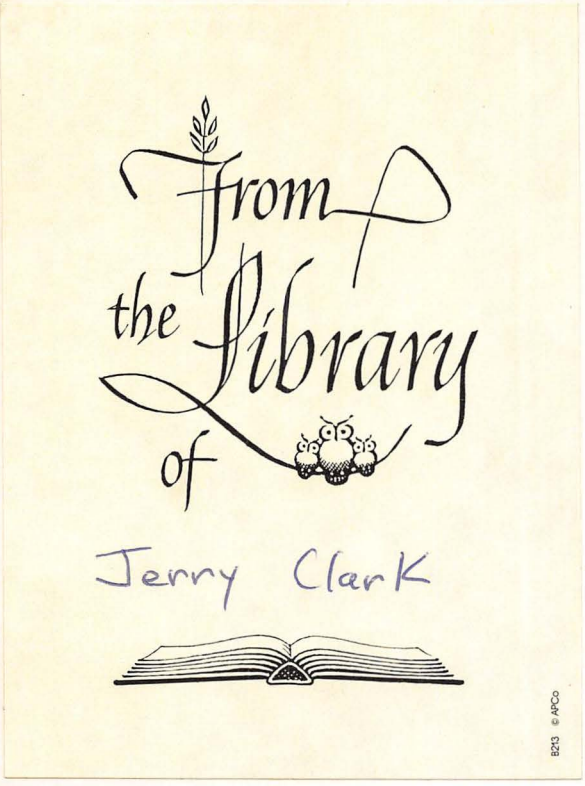
SERIES EDITOR SCOTT KNASTER

Macintosh Inside Out

Elements OF C++ Macintosh® Programming

DAN WESTON


from
the Library
of
Jerry Clark
B213 © APCo

Elements of C++ Macintosh® Programming

Elements of C++ Macintosh® Programming

Dan Weston

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan

Library of Congress Cataloging-in-Publication Data

Weston, Dan.
Elements of C++ Macintosh programming/Dan Weston
p. cm. -- (Macintosh inside out)
ISBN 0-201-55025-3
1. Macintosh (Computer)--Programming. 2. C++ (Computer program language) I. Title II. Series.
QA76.8.M3W48 1990
005.26'5--dc20

90-35408
CIP

Sponsoring Editor: Carole McClendon
Technical Reviewer: John Dance
Cover Design: Ronn Campisi
Text Design: Copenhaver Cumpston
Set in 10.5 point Palatino by Don Huntington

ISBN 0-201-55025-3

ABCDEFGHIJ-MW-943210
First printing, June 1990

This book is dedicated to the members of the nerd breakfast club.

Contents

Preface

Three proverbs for C++ programmers:

If you put five C programmers in a room and ask them to write the same program, you'll get five different programs. If you ask five C++ programmers to write the same program, they'll sit around and argue about the design.
— Wm Leler

It's not the code you write. It's where you put it that matters.
— Steve Splonskowski

Fortran programs can be written in any language.
—Anon

Dan Weston
March 1990

Acknowledgments

Every author says that he or she couldn't have written the book without help. It's true. I am deeply indebted to many people who have generously given time and energy to this book. Jim Berry and Stan Krute read early drafts of the book and were kind enough to be truthful, critical, and encouraging. Steve Splonskowski actually used the examples in this book to write real programs in his work, and in the process found many bugs and design flaws. Their questions and suggestions have dramatically improved the quality of the book. Thanks to Al Smith for telling me how to fix a bug in the printing code. John Dance did a thorough technical review and made many key suggestions that raised the book to a higher level.

Thanks to Scott Knaster for keeping the faith. At Addison-Wesley, Carole McClendon knew how to make a deal quickly when the time was right. Also, many thanks to Joanne Clapp Fullagar for gentle editing, Diane Freed for cooperative production, and Rachel Guichard for everything else.

My family had to live with me while I wrote this book. I am grateful for their patience and understanding and support. Thanks to Sarah and Asa for diverting me so that I couldn't become too demented. Most of all, thanks to Leslie for listening to my dreams.

PART ONE

▶ Getting Your Feet Wet

The first four chapters of this book provide an introduction to C++ and object-oriented programming. Think of it as sticking your toes into a swimming pool, trying to get used to the water before jumping in. These first few chapters will introduce the major techniques, terminology, and syntax that are used throughout the rest of the book to develop large-scale object-oriented programming projects.

Take your time with these chapters, but don't worry if you don't immediately grasp every concept that is introduced. All the concepts that are described in Chapters 1-4 are illustrated by extensive example programs in Chapters 5-14, so you will be able to see the concepts put to work in actual problem-solving situations.

1 ▶ Why C++?

Why should you learn C++? It's a good question. You are probably a busy person. You already know a few computer languages and you can write perfectly good programs using what you already know. Why bother to learn another computer language, especially one as complicated as C++?

This book shows you how C++ can help you write better programs. C++ is not a simple language, but it is easy to use selected parts of the language without first mastering all its subtle features.

The following sections list some reasons, in no particular order of importance, why you should learn C++. Following that are some reasons why you might not want to learn C++. You can decide.

▶ Reasons to Learn C++

C++ can be used just like C. If you are a C programmer already, there is almost no reason not to use C++ instead. C++ can be used just like C, and it will make you a better C programmer because of features like strong type checking and operator overloading. If you have been using another language, like Pascal, and have been thinking of switching to C, learn C++ instead. Chapter 2 describes the features of C++ that make it a worthwhile replacement for C.

C++ supports object-oriented programming. Object-oriented programming can make you more productive, no doubt about it. Personal computers are getting more and more complicated and difficult to program. You need all the help you can get. Object-oriented programming is described in general in Chapter 3 and illustrated in great detail in later chapters.

C++ is going to be a mainstream language. Apple Computer has stated publicly that its new system software will be written in C++. Microsoft and IBM are making strong commitments to C++ for future versions of OS/2. This is not to say that C++ is necessarily the best language — just that its use will be widespread throughout the 90s, just as C was dominant in the 80s. Because C++ will be the standard, more documentation will be available describing how to use the language. Computer companies will define system software calls as C++ functions. Magazines will be filled with articles on C++ and object-oriented programming techniques.

▶ Reasons Not to Learn C++

C++ is hard to learn. C++ is a big, inelegant language. It is easy to get started in C++ but hard to master its many subtleties. C++ should not be your first computer language.

C++ is not for small programs. Because C++ makes you spend more time designing and declaring, it is not very good for small programs (less than a few pages of code) that you might want to knock off in an afternoon. Use some other language for those programs. C++'s advantages show most clearly in larger programming projects.

Compile times for C++ are long. Because C++ is typically implemented as a preprocessor, it adds an extra step to your program's build process. C++ reads your source code and produces standard C code, which it then feeds to a C compiler. All this takes extra time. But whatever time you lose in extra compile time should be regained when you spend less time debugging as your program becomes larger. Once again, the benefits of C++ probably won't be apparent in small projects.

▶ How to Learn C++

This book will teach you how to use C++ by showing lots of examples of C++ code. It is one thing to see a description of a language's features. It is another to see how those features are applied to problem-solving situations. This book applies C++ to the general problem of how to write Macintosh programs. C++ is not a simple language, but this book tries to pick out the features that are particularly useful for Macintosh programming and show those features at work in fully functional programs. There are some features of the language that are not even touched upon using this approach, but I think that you will become comfortable enough with C++ so that you can continue with your own exploration after finishing the book.

Chapters 2 and 3 provide a brief introduction to some of C++'s features. These chapters will acquaint you with terminology and show some short examples, but they are not intended to be comprehensive or deep. Chapter 4 describes several complete example programs using the features described in Chapters 2 and 3. Moving on from Chapter 4, the rest of the book develops increasingly complex programs and classes to illustrate how C++ can be used to solve problems that face Macintosh programmers.

2 ▶ C++: A Better C

As its name suggests, C++ is incrementally derived from the C programming language. One of the design goals of C++ was that it be upwardly compatible with C programs. That is, C programs should be able to be compiled with C++ with minimal changes. That goal has been largely met.

This chapter discusses some of the additions to the C language that make C++ a better version of C. Most of these features are related to strong type checking for function arguments and variable assignments. These features help reduce programming errors and make C++ more suitable for large-scale programming projects with many programmers where strict interfacing guidelines must be enforced.

This chapter does not discuss those C++ features that pertain to object-oriented programming. Those features are described in Chapter 3. Nor does this chapter discuss those features of C++ that are carried over from C.

▶ Comments

One of the most trivial but most welcome new features in C++ is the double-slash (//) comment. This comment indicator tells the compiler to ignore any following text until the end of the line. Old-style C comments, which begin with /* and end with */, are still supported. The big advantage of having // comments, aside from the fact that they are easier to use, is that they can be nested inside old-style comment blocks. For example, consider the following function and its comment, done in the old style.

```
int foo(int a, int b){
   /* here is a comment about the function */
   return a + b;
}
```

If you wanted to comment out the entire function definition with old-style comments, you couldn't: The end comment mark (*/) at the end of the first comment line would prematurely terminate the larger comment block. Old-style comments cannot be nested. In the past, to comment out entire functions or large blocks of code containing comments, C programmers have used #ifdef directives, as shown in the following code.

```
#ifdef NEVER
int foo(int a, int b){
   /* here is a comment about the function */
   return a + b;
}
#endif
```

If the previous function is rewritten with double-slash comments, it is easy to comment out the entire function using old-style comments, as shown by the following code.

```
/*
int foo(int a, int b){
   // here is a comment about the function
   return a + b;
}
*/
```

Most of the programs in this book use the double-slash comments to mark comments in program code. Old-style comments are reserved for commenting out large blocks of code and for those situations where a comment is inserted in the middle of a line but doesn't extend to the end of the line.

▶ Declarations and Definitions

One source of confusion for new C++ programmers is the difference between a declaration and a definition. C++ requires that all functions and variables be declared before they are used. A declaration is a statement that tells the compiler about the variable or function. For variables, the declaration specifies the name of the variable and its data type.

For functions, the declaration specifies the name of the function, its return type and the data types of the function arguments, if any. A declaration does not allocate any space for the variable or function. It is merely a way of specifying the interface for the variable or function.

A definition, on the other hand, explicitly allocates space for the variable or function. A declaration describes the interface. A definition describes the implementation.

Key Point ▶

Definitions cause space to be allocated for a variable, or code to be generated for a function. Declarations do not cause space allocation or code generation.

▶ Variables

For variables, the declaration and definition are most often combined into the same statement. For example, the following statements declare and define variables. Notice that variables can be initialized in the definition statement.

```
int   foo = 5;
Rect  theRect;
float f = 3.66;
```

To declare a variable without actually defining it, you must precede the variable declaration with the extern keyword, as shown in the following declaration statements. Extern tells the compiler that the variable will be defined elsewhere.

```
extern int foo;
extern Rect theRect;
extern float f;
```

▶ Functions

For functions, declaration and definition are more often separated. A function is declared by listing its name, its return type, and the types of its arguments, as shown by the following statements.

```
int  foo(int a, int b);
void bar(int,short,char *);
char * batz(void);
```

Declarations go in .h files

Several observations can be made about the previous function declarations. First, the argument list in a function definition may or may not include names for the arguments, but it must show the type of each argument. Thus, the following function declarations are all equivalent as far as C++ is concerned.

```
int foo(int, int);
int foo(int a, int);
int foo(int, int b);
int foo(int a, int b);
```

(In this book the convention is to include names for function arguments in function declarations unless there is a good reason not to. See the "Compiler Warnings" section of Chapter 4 for an explanation of a situation where argument names should be left out.)

The next observation about function declarations is that the return type can be void. A void function does not return any value.

Finally, notice that the argument list can also be void, indicating that the function does not take any arguments. You can also specify that the function does not take any arguments by simply leaving the argument list empty. Thus, the following two function declarations are equivalent.

```
char * batz(void);
char * batz();
```

Function definitions provide the same information as the function declaration plus a block of code that defines the implementation of the function, as shown by the following function definition. When C++ encounters a function definition, it will use the function block to produce actual code to implement the function.

```
int foo(int a, int b){
   return a + b;
}
```

▶ When to Declare and When to Define

C++ programs are typically broken into several separate source files. These files are compiled separately and then linked together to produce the final application program. Putting related functions and variables into the same source file enables you to reuse that file in other pro-

gramming projects. For example, assume that you have defined a set of functions that calculate the relationship of ohms to volts to current represented by Ohm's law. The following functions can be declared.

```
float GetVolts(float ohms, float current);
float GetCurrent(float volts, float ohms);
float GetOhms(float volts, float current);
```

The declaration of the functions are put into the file elec.h. The definitions of the functions are put into the file elec.cp. The implementation file elec.cp is compiled one time to produce the object file elec.cp.o. This is the file that contains the actual object code for the functions. This file must be linked to other object files to form an application.

The header file, elec.h, which contains the declaration of the functions, can be included in other source files that want to call the electricity functions. The declarations in elec.h tell C++ enough about the functions so that it can do the required type checking on calls to those functions. It is not necessary to know about the definition of the functions. The actual implementation of the functions is immaterial to C++ when it is doing this kind of type checking. Because simple function declarations alone do not cause any code to be generated, you can include a header file in several other files without the penalty of recompiling the functions each time.

Of course, if you include a header file with function declarations, you must link with an object file containing the definitions (implementations) of the functions or the linker will complain that it can't find the missing functions. Chapter 7 describes in detail how to inform the linker what files are necessary to resolve all function references.

Key Point ▶

Function declarations are generally contained in header files (.h) and definitions are generally contained in implementation files (.cp).

Likewise for variables, if you have global variables that you want to include in several other files, you can declare them as extern in a header file and then define them in a separate implementation file. That way, they are defined in only one file but can be referenced from any file that includes the declaration header.

▶ Variable Definitions Anywhere in Code

One big improvement of C++ over C is that in C++ you can define a variable anywhere a normal statement can occur. This is most useful when defining local variables within a function block. In C, all variable declarations in a function must come at the beginning of the function. This causes problems if the function is long because you have to keep looking back at the beginning to check the name of the variable. It also causes a problem when you delete a section of code in a long function definition because you often forget to go back to the beginning and delete the variable definitions used by the deleted code.

C++ allows you to define variables right before you use them. Thus, in a long function definition, you can group individual variable definitions with the code that uses them. This makes your code more readable and maintainable.

A common use of this feature is to define a loop counter variable in the for clause of a for-loop, as shown in the following code fragment.

```
for(int i = 0; i< maxCount; i++){
```

▶ Type Checking

C++ checks the data types of all function arguments. This tends to catch many errors during compilation rather than letting them go until runtime. The reason that C++ requires full argument lists in function declarations is so it can check arguments against the argument list when the function is called.

Strong type checking means that C++ will warn you or refuse to compile function calls where incompatible variables are passed to a function. C++ will do some automatic type conversion for you when the type that you pass can be converted to the type that the function expects. For example, when passing a floating-point number to a function that expects an integer, C++ will automatically truncate the floating-point number to its integer component before passing it to the function. When the conversion will change the value, as it does in floating-point to integer conversion, C++ will issue a warning during compilation.

Strong type checking is a big advantage, but it can be somewhat restrictive at times. Suppose you want to pass a pointer variable to a function that expects a long integer. You know that a pointer is the same size as a long integer, but the compiler complains because it doesn't have a standard way of converting a pointer to a long integer.

You can get around this problem by making an explicit typecast of the pointer variable to a long integer when passing it as a function argument, as shown in the following code fragment.

```
char * p;          // definition of pointer variable
void foo(longl);   // declaration for function with one long arg
foo(p);            // compiler will complain !!!
foo((long)p);      // typecast makes it all OK
```

Typecasting is a way of telling C++ that you know what you are doing. Actually, with C++'s ability to overload functions, you could define another version of the function shown in the previous example so that it could be called with a pointer argument as well as with a long integer argument, thus eliminating the need for an explicit typecast. See the section "Overloaded Functions" in this chapter for details.

▶ Arguments by Reference

When you specify the arguments in a function's argument list, you can follow the argument type name with a & character to indicate that the argument should be passed by reference rather than by value. This means that C++ will pass a pointer to the argument rather than passing a copy of the argument to the function. Thus, you will be able to alter the value of the argument in the function and have those changes show up in the variable that was specified as the argument.

Pass-by-reference is essentially the same as the VAR argument feature of Pascal. Pass-by-reference permits you to specify the name of a variable when specifying an argument rather than needing to specify the address of the variable. It also means that inside the function, you do not have to dereference a pointer to get at the contents of the argument. This is most useful for structures and other complex data types.

Consider the following example. Assume that a data structure has been defined as follows to represent a rectangle.

```
struct Rect {
   short top;
   short left;
   short bottom;
   short right;
};
```

Now consider a function to calculate the area of the rectangle. In C, you would define the function to take a pointer to a Rect, as shown here.

```
int Area(Rect * r){
   return ((r->bottom - r->top) * (r->right - r->left));
}
```

When calling this function, you would have to provide the address of a Rect variable as an argument, as shown in the following code fragment.

```
Rect theRect;
int theArea = Area(&theRect);
```

Using C++'s call-by-reference syntax, you could define the Area function as follows.

```
int Area(Rect& r){
   return ((r.bottom - r.top) * (r.right - r.left));
}
```

When calling this version of the function, you could simply provide the name of a Rect variable as an argument, as shown in the following code fragment.

```
Rect theRect;
int theArea = Area(theRect);
```

The pass-by-reference syntax allows you to avoid the dereferencing and address-of operations that are typically used to pass pointer arguments. Pass-by-reference also forces the caller to allocate the structure that is to be used by the called function. For example, if you used the Area function that takes a pointer argument, there is nothing to prevent you from allocating a Rect pointer variable that doesn't actually point to a valid Rect structure and then erroneously passing that pointer to Area, as shown in the following code.

```
Rect *pRect;
int theArea = Area(pRect); // DANGER: using uninitialized Pointer!
```

▶ Default Argument Values

Another new feature of C++ is the ability to assign default values to function arguments in the declaration of the function. For example, assume that you declared a function that played a sound at a specified sample rate. The function would take two arguments, a pointer to the sound data and an integer specifying the sample rate, as shown in the following declaration.

```
void PlaySound(char * sound, int sampRate);
```

Assume further that the normal sample rate for sounds in your system is 22,000 Hz. You could define a default value for the sample rate argument so that the caller of the function wouldn't have to provide the sample rate argument if the default rate was acceptable. The declaration with the default assignment is shown as follows.

```
void PlaySound(char * sound, int sampRate = 22000);
```

C++ will substitute the default argument value only if the caller does not supply the argument. More than one argument in an argument list can have a default value, but substitution is based on the argument position, so an argument with a default value cannot be followed in the argument list with a nondefault argument. Another way of saying this is that the default arguments must be last in the argument list. Thus, it would be incorrect to declare the PlaySound function as follows.

```
void PlaySound(int sampRate = 22000, char * sound); // INCORRECT !!
```

Key Point ▶

Default argument values can only be specified once, in either the declaration or the definition of the function, but not both. This book uses the convention of specifying the default arguments in the function declaration, but it could be done in the definition instead.

▶ Inline Functions

C++ allows you to define inline functions. Inline functions are not invoked through the normal function call mechanism, which involves stack manipulation and can be rather inefficient for functions that are called often or for small functions where the overhead of the function call is greater than the actual work done by the function. Instead, the body of inline functions is substituted directly into the code at the place where the inline function is called. Any arguments are also type checked and substituted. The substitution happens during compilation; at runtime there is no actual function call, just execution of the substituted code that makes up the function.

Inline functions basically replace the macro facility that is commonly used by C programmers to avoid the overhead of function calls for short, frequently called routines. Inline functions have the advantage over macros of strong type checking for inline function arguments and return types. For example, in a C program you might define the following macros to return the high and low 16-bit words from a 32-bit long integer.

```
#define HiWrd(aLong)  ((short)  (((aLong) >> 16) & 0xFFFF))
#define LoWrd(aLong)  ((short)  ((aLong) & 0xFFFF))
```

In C++ you would define inline functions instead of macros, as shown by the following inline function definitions.

```
inline short HiWrd(long aLong)
   {return (short)(((aLong) >> 16) & 0xFFFF);}
inline short LoWrd(long aLong)
   {return (short)((aLong) & 0xFFFF);}
```

Key Point ▶

The advantage of defining inline functions rather than macros is that C++ checks the argument and return value for inline functions but not for macros.

▶ Const Definitions

In traditional C programs you typically define constants with #define statements, such as the following.

```
#define  MAXSHORT 32767
#define  MINUSONE -1
```

C++ provides a better mechanism for defining constants that allows you to specify the exact data type of the constant as well as its value. In C++ the previous #define statements would be replaced by the following statements.

```
const  short MAXSHORT = 32767;
const  long MINUSONE = -1;
```

Const definitions essentially create read-only variables. A const variable must be initialized when it is defined. It cannot be changed thereafter. The advantage of const definitions over #define statements is that the const variable has a definite data type and can thus be subjected to type checking when it is used as a function argument.

▶ Overloaded Functions

Function overloading in C++ permits you to define several versions of a function, each with a different argument list. The same function name then refers to several different function definitions. The compiler is able to differentiate between the versions by looking at the argument types. For example, look again at the example from the "Type Checking" section of this chapter where it was necessary to do an explicit typecast in order to pass a pointer argument to a function that expected a long integer argument, shown as follows.

```
char * p;           // definition of pointer variable
void foo(long l);   // declaration for function with one long arg
foo(p);             // compiler will complain !!!
foo((long)p);       // typecast makes it all OK
```

By overloading foo so that it can accept a pointer argument, you can avoid the need to typecast, shown as follows.

```
char * p;            // definition of pointer variable
void foo(long l);    // declaration for function with one long arg
void foo(char * c); // OVERLOADED function declaration
foo(p);              // compiler will NOT complain !!!
```

Each overloaded function must have its own definition. Each overloaded version of the function is actually a separate function. All the functions just share the same name. C++ decides which version of the named function to call by matching the arguments provided against the argument lists of the declarations.

▶ Streams

Like C, C++ does not contain any predefined input and output operators. All I/O is handled by library functions. C has the stdio library, with the infamous printf function. C++ supports stdio, but it also defines a new type of I/O called the iostream library. Streams provide the same type of formatting and input/output services as the stdio library, but they also provide better type checking and additional support for user-defined data types.

A stream is a sequence of bytes. Data can be extracted from the stream and placed into program variables with the **extraction** operator (>>). Data can also be inserted into the stream with the **insertion** operator (<<). In a typical UNIX environment, streams can be associated with

the standard input and output I/O channels. For example, standard input is often associated with a user terminal keyboard, and standard output is the terminal screen. Thus, extracting data from the standard input stream retrieves characters typed by the user at the terminal. Inserting characters into the standard output stream sends those characters to the terminal screen.

The iostream library defines three stream variables — *cin*, *cout*, and *cerr* — that correspond to standard input, standard output, and standard error output channels of the system. You can use these predefined variables in your programs to extract and insert data into the standard I/O channels. For example, to send a string to the standard output channel, you could use the following statement.

```
cout << "This data will go to the output channel \n";
```

More important, you can combine insertion operations and use the automatic formatting capabilities of the **insertion** operator to produce composite output, as shown in the following statements.

```
int x = 23;
cout << "The value of x is " << x << ".\n";
```

The previous statements will produce the following output.

```
The value of x is 23.
```

You can see that the **insertion** operator knows how to change the int variable *x* into its textual representation before inserting it into the output stream. Likewise, the **extraction** operator knows how to take a textual representation from an input stream and convert it into a specified destination data type. For example, you could extract an integer from the input stream and place its value into an int variable with the following statements. In addition to inserting and extracting predefined data types, the stream operators can be overloaded to operate on user-defined types.

```
int x;
cin >> x;
```

Chapter 4 shows how to use iostreams in the MPW tools to take advantage of the UNIX-like input and output redirection facilities of the MPW environment. In a typical Macintosh program, however, standard input and output are not supported. Streams are more useful in Macin-

tosh programs when they are associated with files. When it is associated with a file, a stream can be used to insert and extract data from the file. Chapter 13 shows how to modify the basic functionality of a stream so that its output is directed into a debugging window in your program.

▶ Summary

C++ adds many new features to C. Many of those features center around stronger type checking. This makes C++ programs less prone to runtime errors involving faulty function arguments or variable assignments. Most of these improvements were made without sacrificing the efficiency that C is famous for.

Even if C++ added only the features described in this chapter, it would be worthwhile to make the switch from C to C++. But C++ also adds new features that support object-oriented programming. The next chapter describes the object-oriented features of C++. The ability to do object-oriented programming is what makes the move to C++ really exciting.

3 ▶ C++ Objects

This chapter provides an overview of the object-oriented features of C++. Object-oriented programming includes the ability to define classes, create objects that belong to a class, and derive new classes based on existing classes. These techniques can have a dramatic effect on program design.

Object-oriented programming techniques are having the same sort of impact on the programming profession that structured programming had when it was first introduced in the mid 1970s. Object-oriented programming will change the way you write programs. Using object-oriented design techniques will make you a better programmer.

This chapter describes what a class is and discusses the different parts of a class. The second half of the chapter develops a set of classes that encapsulates the idea of a linked list. This example shows how classes often work together to achieve a common goal.

▶ Class Declarations

A class is like a data structure. It has slots for data, called members, and slots for functions, called member functions. When you write a declaration of a class, you are specifying a template for objects that belong to that class. The class declaration tells the compiler how much space to set aside each time an object of that class is created and how to access the members and member functions of that class.

A class enables you to encapsulate data and functions that operate on that data in one place. For example, a simple class declaration for a class that represents a rectangle is shown as follows.

* Classes begin with T
members begin with f

```
class TRect {
public:
   // these are the data members
   short fTop;
   short fLeft;
   short fBottom;
   short fRight;
   // these are the member functions
   short Area(void);
   Boolean PointInRect(Point thePt);
};
```

By The Way ▶

The naming conventions used in this book for classes and members are derived from the MacApp-Object Pascal naming scheme used by Apple. Class names begin with T because in Object Pascal classes are called types. Similarly, members begin with f, since in Object Pascal members are called fields. Naming members this way helps to distinguish between members and local and global variables.

The class declaration gives the name of the class, **TRect**, and then lists the members and member functions for the class. The keyword public: specifies the protection level of the members and member functions. Protection is discussed in the next section.

Notice that each member is preceded by a type designator and that each member function declaration includes a full prototype of the function, including argument types and return value.

Key Point ▶

Don't forget to put a semicolon after the closing brace of a class declaration. This is often confusing to new C++ programmers since the closing brace of a function definition does not need to be followed by a semicolon. Failure to put a closing semicolon after a class declaration can result in mystifying compiler error messages.

▶ Public, Protected, and Private

C++ provides three levels of protection for members and member functions within a class. The default protection level is private. That means that a private member or member function cannot be accessed by any function that is not a member function of that class. This is sometimes known as data hiding, although in C++ functions as well as data can be hidden. The next level of protection is protected. A pro-

tected member or member function is accessible only to other member functions of that class or from classes derived from that class. (Derived classes are described in a later section.) The protected designation is most useful when you plan to derive other classes from your base class. The least restrictive level of protection is public. Public members and member functions can be accessed by any other function.

You control the protection level of members and member functions by inserting the keywords public, protected, or private, followed by a colon, in your class declarations. If none of these keywords is used, the compiler assumes that you want everything to be private. You will usually want at least some of the member functions for a class to be public, otherwise the class will be unusable by other parts of your program. (See the discussion of friends in the next section for a qualification to this statement.)

Key Point ▶

If you don't specify a protection level, C++ will assume that you want private members and member functions by default.

The protection keywords can appear many times in a class declaration. For example, you can switch from public to protected and then go back to public as often as you want within a declaration. The following class declaration shows how this might be done.

```
class TRect {
protected:
   // these data members are protected
   short fTop;
   short fLeft;
   short fBottom;
   short fRight;
public:
   // these member functions are public
   short Area(void);
   Boolean PointInRect(Point thePt);
   void SetRect(short top, short left,
      short bottom, short right);
protected:
   // this member function is protected
   short ComparePoint(Point thePt);
};
```

Using different levels of protection for different parts of your class declaration permits you to hide some the details of your class while making other parts explicitly visible to outside users. The **TList** class described in the last half of this chapter shows how to use protected members to hide implementation details while providing public member functions to use the class. The more elaborate classes developed in Chapters 5 and 6 also show how to use the various protection levels in a class declaration.

▶ Friends

There are cases where you will want to provide special access to protected or private members to a particular function or class but you won't want to open access completely by making the members public. C++ provides the friend keyword to allow you to designate specific functions or classes that can access protected or private members and member functions. This lets you grant special access privileges to specific functions or classes without giving up the general protection enjoyed by the protected and private members. The iterator and list classes described in the last half of this chapter show an example of how to use the friend mechanism.

▶ Accessing Members and Member Functions

Class members and member functions are accessed just as if they were elements of a data structure. For example, if you had a **TRect** variable (going back to the original declaration of **TRect** that had public members and member functions), you could access its members and member functions as shown in the following code fragment.

```
Point p;
TRect theRect;
theRect.fTop = 0;
theRect.fLeft = 0;
theRect.fBottom = 100;
theRect.fRight = 300;
short theArea = theRect.Area();
Boolean inRect = theRect.PointInRect(p);
```

If your variable is a pointer to an object rather than the object itself, you must use the **dereference** operator (->) to access the members and member functions of the object, as shown by the following code.

```
Point p;
TRect theRect;
TRect * ptheRect = &theRect
ptheRect->fTop = 0;
ptheRect->fLeft = 0;
ptheRect->fBottom = 100;
ptheRect->fRight = 300;
short theArea = ptheRect->Area();
Boolean inRect = ptheRect->PointInRect(p);
```

Member functions of a class can access other members and member functions of the same class by name alone, without supplying an object or object pointer. This is shown by the following definition for the **Area** member function in the **TRect** class. **Area** can access the other members of the **TRect** class as if they were simple variables. Member functions can likewise be accessed as if they were simple functions.

```
short TRect::Area(void){
   return ( (fBottom - fTop) * (fRight - fLeft));
}
```

:: denotes access to member variables or other functions in a class.

▶ Derived Classes

One of the most important aspects of object-oriented programming is the notion of inheritance. Once you define a class, you can derive other classes from that class. The derived classes inherit all the members and member functions of the original class. The original class, from which you derive the new classes, is the parent or base class.

A derived class has all the characteristics of its parent class. Typically, you derive one class from another so that you can add or change selected parts of the parent class in the derived class. There are two ways to make changes when deriving a new class. First, you can extend the parent class by adding new members or member functions to the new class. These new members or member functions become part of the derived class in addition to the members and member functions that are inherited from the parent class. The second way to modify a derived class is to override member functions from the parent class. When you override a member function, you supply a new definition for the function. C++ will use that new function definition for the derived class, but the original definition will remain valid for the parent class.

For example, you can derive a round-cornered rectangle class from the original **TRect** class with the following declaration.

```
class TRoundRect : public TRect {
protected:
   // add some new members
   short fHOval;
   short fVOval;
public:
   // and override the area member function
   short Area(void);
};
```

TRoundRect is derived from **TRect**, as indicated by the single colon following the class name **TRoundRect**. The first line of its declaration also includes the keyword public, which specifies that **TRoundRect** is publicly derived from **TRect**. Because **TRoundRect** is publicly derived from **TRect**, objects in the **TRoundRect** class can be treated just like objects from the **TRect** class. That is, you can create a **TRoundRect** object and use it to call member functions or access members that were defined for the **TRect** class. If the public keyword is left off the first line of the class derivation, then the class is privately derived by default, and objects of the derived class cannot be used as if they were objects of the parent class. For privately derived classes, only those members and member functions that are defined (or overridden) in the derived class can be accessed. Member functions of a privately derived class can access the parent class's members and member functions, but users of objects of the derived class cannot use the derived objects to access members and member functions for the parent class. All the examples in this book use public derivation since we want full access to the parent class's members and member functions. It is, however, sometimes useful to hide the details of a base class by using private derivation for its derived classes.

TRoundRect also adds two new members, *fHOval* and *fVOval*, to specify the shape of the round corners of the rectangle. **TRoundRect** inherits the original four data members specifying the corners of the rectangle from **TRect**. Finally, **TRoundRect** overrides the **Area** member function to more accurately calculate the area of the round-cornered rectangle.

Most of the rest of this book is dedicated to showing examples of derived classes. Inheritance, and the ability to modify or extend the behavior of a base class, are the most powerful ideas in object-oriented programming. The examples in Chapters 4-14 illustrate how useful these techniques can be.

▶ Members

The class declaration describes the members that make up the class. Each object of the class that is created gets a copy of all the class members. That is, each instance of a class has its own private copy of the members. For example, if you create two **TRect** objects, as shown by the following code fragment, each object will have separate copies of the *fTop*, *fLeft*, *fBottom*, and *fRight* members.

```
TRect rect1;
TRect rect2;
rect1.fTop = 0;
rect1.fLeft = 0;
rect1.fBottom = 100;
rect1.fRight = 300;
rect2.fTop = 20;
rect2.fLeft = 30;
rect2.fBottom = 400;
rect2.fRight = 100;
```

The members of an object allow that object to maintain its own state, independent of the state of any other objects of the same class that have been created.

▶ Static Members

If you declare a class member with the static keyword, that member is shared by all objects of that class. There is only one copy of the static member. Thus, if one object sets the value of the static member, that value is reflected in all objects of that class. A static member is like a global variable that obeys access and protection rules as if it were a class member. The following declaration adds a static member to the **TRect** class.

```
class TRect {
public:
   // static member
   static short fNumRects;
   // these are the data members
   short fTop;
   short fLeft;
   short fBottom;
   short fRight;
   // these are the member functions
   short Area(void);
   Boolean PointInRect(Point thePt);
};
```

Static members are declared as shown in the declaration of the class. They must be defined separately. A good rule of thumb is to define and initialize static members just like global variables. The following code shows how to define and initialize the *fNumRects* static member for the **TRect** class.

```
short TRect::fNumRects = 0;
```

Once initialized this way, the static member can be modified by any of **TRect**'s member functions. The syntax for accessing a static member from within a class member function is exactly the same as for non-static members. From outside the class, a static member can be accessed by preceding its name with the name of the class and two colons, as shown in the definition statement. Of course, access from outside the class also depends on the protection level (public, protected, or private) of the static member. (See the **TScrollDoc** class in Chapter 10 for an example of how to use a static member.)

▶ Member Functions

All objects of a particular class share the member functions for that class. While each object has a separate copy of the members, it would be too inefficient to give each object copies of the code that implements the member functions for the class. Thus, when you call a member function for an object, it uses the common code that defines the member function for the whole class.

To call a member function for a particular class, you must have an object, or a pointer to an object, that belongs to that class. It is necessary to associate a member function call with a specific object in this way because member functions often alter the state of the object by accessing members of that particular object.

▶ "this" as a Function Argument

Every member function for a class must be declared with a function prototype in the class declaration. This prototype lists the arguments for the member function. C++ adds an additional argument, "this", to the beginning of the argument list for every member function. The "this" argument is a pointer to the individual object that is being used to access the member function. For example, consider the following declaration for the rectangle class.

```
class TRect {
public:
   // these are the data members
   short fTop;
   short fLeft;
   short fBottom;
   short fRight;
   // these are the member functions
   short Area(void);
   Boolean PointInRect(Point thePt);
};
```

The two member functions, **Area** and **PointInRect**, are actually declared as follows after C++ adds the implicit "this" argument.

```
short Area(TRect * this);
Boolean PointInRect(TRect * this, Point thePt);
```

Since the "this" argument is implicit, you do not have to include it in the argument list for your member functions. Likewise, you never have to worry about providing it when calling the member function from C++, since the compiler automatically adds the extra argument when you call the member function.

C++ uses "this" to make sure that access to class members from within the member function affects the members of the particular object that was used to make the member function call. You can use "this" to refer to the object itself within the definition of the member function, although it is not usually necessary. You will need to use "this" only if you want to pass a reference to the object to some other function.

By The Way ▶

The "this" argument to a member function can be used to explicitly show that a member or member function for the class is being accessed. Thus, assuming that a member named *fTop* is defined for a class, this->*fTop* and *fTop* would be equivalent ways of referring to the member from within a member function for that class. Many programmers prefer the more explicit form, and there is no performance penalty for using it.

One consequence of the implicit "this" argument is that you cannot use C++ member functions to write functions that will be called from the Macintosh toolbox. Functions that are called from within the toolbox, such as the scroll action procedures described in Chapter 10, have a strictly defined set of arguments and cannot accommodate the extra implicit argument.

▶ Virtual Functions

When you declare a member function you can tell C++ that you will probably be overriding that function in a derived class by preceding the function declaration with the virtual keyword, as shown by the following class declaration.

```
class TRect {
public:
   // these are the data members
   short fTop;
   short fLeft;
   short fBottom;
   short fRight;
   // these are the member functions
   virtual short Area(void);
   virtual Boolean PointInRect(Point thePt);
};
```

A virtual member function for a class is called through a jump table associated with the class. This makes virtual functions slightly less efficient than nonvirtual functions, which are called with direct function calls. In practice you should not notice the difference in efficiency, although if you find that you are repeatedly calling a virtual function in a computationally intensive situation, you may want to make the function nonvirtual instead.

Key Point ▶ A good rule of thumb is to make a member function virtual if you think you will ever override it in a derived class.

Nonvirtual member function calls are bound at compile time. That is, the compiler knows exactly which function corresponds to the member function call and emits code to make a direct function call. In contrast, virtual function calls are bound at runtime. This means that the compiler can't figure out exactly which function should be called, so it creates code that looks up the function pointer in the jump table for the object. At runtime, the member function call invokes the code to look up the function pointer and then jump to the actual virtual function that is appropriate for that class.

Virtual member functions are necessary since objects in derived classes are often referenced by pointers to the parent class type. A good example of this shows up in the generic application and document classes, **TApp** and **TDoc**, used in Chapters 5-13. The application

class manipulates documents by making calls to **TDoc** member functions. The application class keeps all its document references as **TDoc** pointers. Yet, in practice, these document objects are actually classes derived from **TDoc**. The application treats the documents as if they belonged to the **TDoc** class, but because the document member functions are virtual, the proper overridden member functions for the derived documents get called. Without virtual functions, the original member functions of **TDoc** would always be called when the application used a **TDoc** pointer to make a function call.

Key Point ▶

By making the member function virtual, you are telling the compiler to check the actual class of the object at runtime to ensure that the derived class's member function will be called even if the pointer used to access the member function is declared as a pointer to the base class.

▶ Static Member Functions

A static member function is a member function declared with the static keyword. A static member function in a class can access only the static members of the class. Static member functions provide a way to access static members without needing an actual object of the class.

Static member functions do not have the implicit "this" argument described previously. This means that they cannot access any of the nonstatic members of the class. (See the **TScrollDoc** class in Chapter 10 for an example of how to use a static member function to access a static member.)

▶ Constructors and Destructors

One of the best features of C++ is that it provides a way to define functions that will be automatically called when an object is created and when it is deleted. The function that is called when an object is created is a constructor. Constructors are typically used to initialize the members of the object to their default state. The function that is called when the object is deleted is a destructor. The destructor is typically used to deallocate any memory allocated by the object.

A constructor is a member function that has the same name as its class. A destructor has the same name as its class with a leading tilde (~). For example, to add a constructor and destructor to the **TRect** class described earlier in this chapter, the following declaration would be used.

```
class TRect {
public:
   // these are the data members
   short fTop;
   short fLeft;
   short fBottom;
   short fRight;
   // these are the member functions
   short Area(void);
   Boolean PointInRect(Point thePt);
   // constructor
   TRect(void);
   // destructor
   ~TRect(void);
};
```

Key Point ▶

If the constructor for a class is not public, only friends of that class will be able to create objects of that class, since creating an object is just like calling its constructor.

Creating an object is equivalent to calling its constructor

Constructor automatically initializes members of an object to default values.

Neither a constructor nor a destructor can be declared to return any value as a function result. A constructor can be declared to take arguments, but a destructor cannot have any arguments. The ability to pass arguments to a constructor is a very useful feature, and it is commonly used in C++ programs. For instance, the constructor for the **TRect** class just described would more usefully be declared in the following way.

```
TRect(short top = 0, short left = 0,
   short bottom = 0, short right = 0);
```

When declared this way with default arguments, the user has a choice of providing the coordinates for the rectangle or passing no arguments and accepting the defaults. The following definition for the constructor shows how the arguments could be used to initialize the object members.

```
TRect::TRect(short top, short left, short bottom, short right){
   fTop = top;
   fLeft = left;
   fBottom = bottom;
   fRight = right;
}
```

Constructors and destructors are not required for a class. You can define a constructor without a destructor, or a destructor without a constructor, or neither, for any class.

When a class is derived from another class, it inherits its parent class's constructor and destructor. Parent constructors are invoked before derived constructors. Destructors are invoked in the opposite direction, proceeding from the derived class upward through its parent chain. This topic is discussed in more detail with examples in Chapters 8, 9, and 11.

Since a constructor cannot return any status, it is unwise to do anything in a constructor that can fail. Constructors are typically used only to initialize members, not to allocate memory or read in resources. If your class needs to allocate memory, you should add a separate initialization member function that is called separately after the object has been created. For example, if the **TRect** class needed to allocate memory, you would add a member function, called something like **InitRect** or **IRect**, and call that function after creating a **TRect** object. See the **TDoc** class in Chapter 5 and the **TApp** class in Chapter 6 for examples of initialization functions that are separate from the constructor.

Destructors, on the other hand, are especially good for deallocating memory that has been allocated by the object.

▶ Virtual Destructors

It is important to make a virtual destructor for a class that will be used as a base class for derived classes. This is necessary when the derived objects are referenced with pointers to the base class, as discussed previously in the "Virtual Functions" section. For example, the application class described in Chapter 6 always deletes documents by calling the **delete** operator on a pointer to the base document class, even though the document is actually always an object of a derived class. Because the document class destructor is virtual, however, the correct destructor for the derived class is called.

Key Point ▶ Classes that will be used as base classes for derived classes should have virtual destructors.

▶ Creating Objects

A class declaration describes a template for the creation of objects. During program execution you create objects that belong to one class or another. C++ uses the class declaration to determine how much memory to allocate for the object.

There are two ways to create objects in a C++ program. The first is to define a variable with a particular class type, either as a global variable or as a local variable within a block. For example, to define a **TRect** object this way, you would use a statement like that shown here.

```
void foo(void){
   TRect theRect;  // constructor called here
   // ... do something with theRect here
   // destructor for theRect called here as it goes out of scope
}
```

When C++ sees a variable definition for an object type, it makes the space for the object on the stack and calls the constructor, if any, for the object. Thus, simply defining the variable as shown here is enough to make its constructor execute. Similarly, when an object variable goes out of scope (at the end of the block in which it was defined), its destructor is called automatically by C++. In the preceding example, since the **TRect** variable was defined within a function block, it will go out of scope when the function terminates. Global variables go out of scope when the program terminates. This automatic construction and destruction of objects is one of the big advantages that C++ has over other object-oriented languages such as Object Pascal.

The notion of scope goes deeper than just global variables and local variables. Scope is defined by the smallest block that encloses the definition. Thus, in the following example, the object variable theRect is created when entering the for-loop and destroyed when leaving the loop.

```
void foo(void){
   short i;
   for(i = 1; i < 10; i++){
      TRect theRect; // constructor called here
      // ... do something with theRect here
      // destructor for theRect called when it goes out of scope
   }
   // other function steps here can't access theRect
}
```

The other way to create an object is to declare a variable that is a pointer to the object type and call the C++ **new** operator, which will allocate space for the object on the heap and call the constructor, if any, for the object. In the same way that memory for the object is allocated with **new**, a pointer variable must be explicitly deallocated with the **delete** operator. The constructor for the object is executed when **new** is called, and the destructor is executed when **delete** is called. The destructor is not called automatically when the pointer variable goes out of scope. You must explicitly call **delete** for objects created with **new**. The following code shows how to create and destroy a **TRect** object with **new** and **delete**.

```
void foo(void){
   TRect * theRect = new TRect;  // constructor called here
   // ... do something with theRect here
   // destructor for theRect called when it is deleted
   delete theRect;
}
```

The preceding example shows how **delete** can be called before the function exits to deallocate the **TRect** object created at the beginning of the function. However, you might not want to delete the object at that point. The big advantage of using pointers to objects instead of the objects themselves is that the object can live beyond the scope in which it was originally created. This means that you can create a pointer to an object and pass it around among many functions within a program, deleting it at some later time when appropriate. This can make the object available to many parts of a program without having to make it a global variable.

▶ A Generic List Class

The rest of this chapter develops a few sample classes that can be used together to manage lists of objects. And what would you do with a list of objects? You must be able to add an object, remove an object, and find out how many objects are in the list. These three operations can be represented by the public member function of the class **TList**, as shown by the following partial class declaration.

```
class TList {
public:
   TList(void);  // constructor
   virtual void AddItem(void* item);
   virtual void RemoveItem(void* item);
   int NumItems() { return fNumItems; }
   // more class declarations to come...
```

Notice that the items you add or remove are represented by void pointer (void *) arguments. A void pointer tells the compiler that any sort of pointer will do. Typically, the items added to the list will be referenced by pointers to objects of one type or another. Because the arguments are declared as void pointers, the C++ compiler will let you pass any sort of pointer to those member functions.

Notice also that the member functions **AddItem** and **RemoveItem** are virtual, which means that they can be overridden in classes derived from **TList**. The member function **NumItems** is not virtual since it will never need to be overridden.

The member functions just described tell you how to use the **TList** class. But how will the list actually be implemented? There are options for managing a list: a linked list, a doubly linked list, and a dynamic array. Looking at the simple linked-list option, you can create a new class, **TLink**, to describe each link in the list. Each link has a pointer to the object that represents the item, and a pointer to the next link in the list, as shown by Figure 3-1.

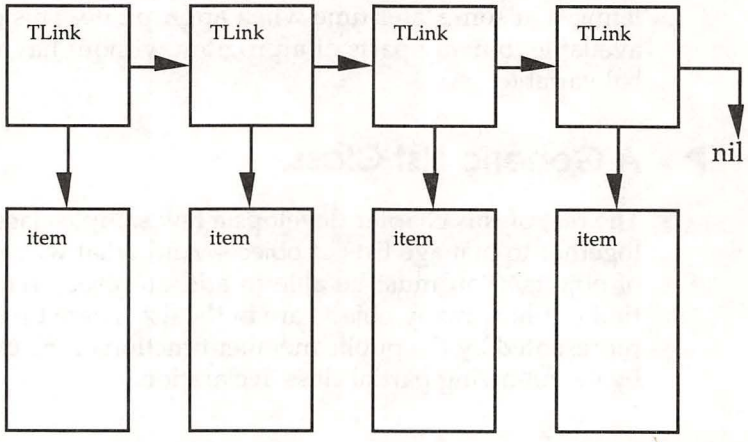


Figure 3-1. A Linked List

The **TLink** class is declared as follows.

```
class TLink {
   TLink* fNext; // the link to the next item
   void*  fItem; // the item this link refers to
public:
   TLink(TLink *n = nil, void *item = nil)
      {fNext = n; fItem = item;}
   TLink* GetNext()
      { return fNext; }
   void* GetItem()
      { return fItem; }
   void SetNext(TLink* aLink)
      { fNext = aLink; }
   void SetItem(void* anItem)
      { fItem = anItem; }
};
```

Notice that the members are private by default, since they are not explicitly public or protected. Access to the private members is provided by public member functions. You can see that all the member functions of **TLink** are defined within the declaration of **TLink**. Because they are defined within the class declaration, the compiler will treat them as inline functions, as described in Chapter 2. This means that they will execute faster than a traditional function call, thus lessening the penalty for not being able to directly access the private members. Notice also the default arguments to the constructor for **TLink**.

The constructor for the **TLink** class is also defined within the class declaration. It takes a link and an item as arguments. It uses the link argument to set the *fNext* member and uses the item argument to set the *fItem* member. Thus, the act of creating a new link and running the constructor establishes the connections between links that make up the list. Figure 3-2 shows what happens when an existing link is used as an argument to create a new link.

Now that the **TLink** class is defined, look again at the complete declaration of the **TList** class, shown as follows.

```
class TList {
protected:
   TLink* fLink; // the first link in our list
   int  fNumItems; // the number of elements in the list
public:
   TList(void);  // constructor
   virtual void AddItem(void* item);
   virtual void RemoveItem(void* item);
   int NumItems() { return fNumItems; }
};
```

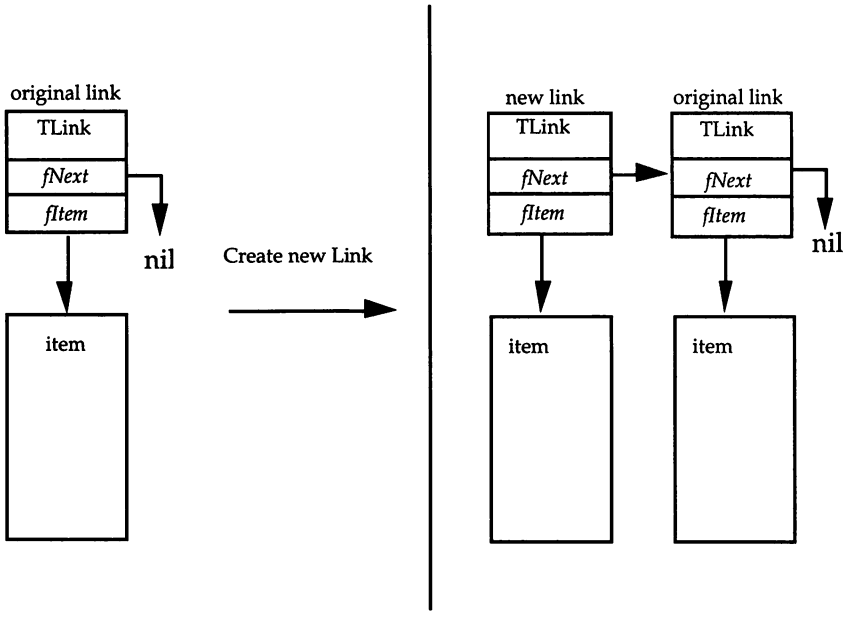


Figure 3-2. Creating a New Link

Two protected members are included in the class to keep track of the first link in the list and the number of items in the list. These members are protected rather than private because they need to be accessible to classes derived from **TList**, as shown in Chapter 6. If the members were private, they would not be accessible to derived classes.

The constructor for the **TList** class is shown as follows. It merely initializes the two members to show that the list is empty.

```
TList::TList(void){
   fLink = nil;
   fNumItems = 0;
}
```

The **AddItem** member function is called with an item pointer as its argument. **AddItem** creates a new **TLink** object to hold the item, sets the *fLink* list member to point to the new link, and increments the *fNumItems* member. **AddItem** creates the new **TLink** object by calling the C++ **new** operator, giving the name of the **TLink** class and the arguments to the **TLink** constructor. The arguments to **TLink**'s constructor

are the current first link in the list and the pointer to the new item. In the process of creating the new link, **TLink**'s constructor uses the link passed as its argument to set the *fNext* member of the new link. Because **AddItem** uses the new link to set the *fItem* member of **TList**, the new item becomes the first item in the list. Figure 3-3 shows what happens when a new item is added to the list.

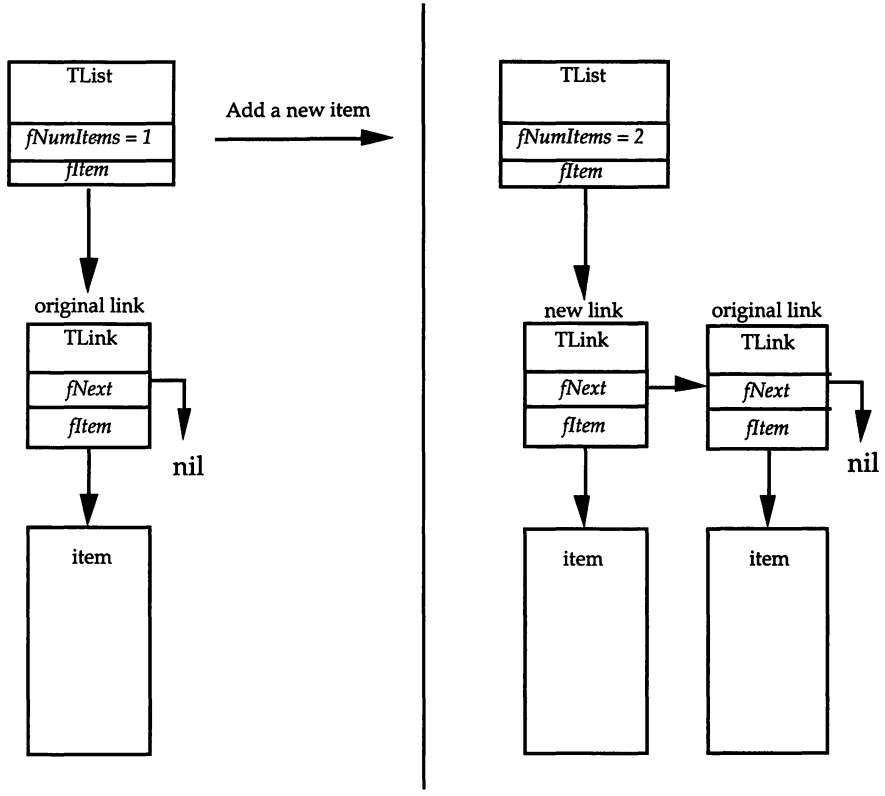


Figure 3-3. Adding an Item to the List

The code for **AddItem** is shown as follows.

```
void TList::AddItem(void* item){
   fLink = new TLink(fLink,item);
   fNumItems++;
}
```

To remove an item from the list, the **RemoveItem** member function must search the list to find a matching item and then delete the link

that holds that item. **RemoveItem** starts at the first link in the list and compares its item to the item that was passed to **RemoveItem** as its argument. If the matching item is the first link in the list, then **RemoveItem** simply sets the *fItem* member of the list to point to the next link in the list, as shown by Figure 3-4.

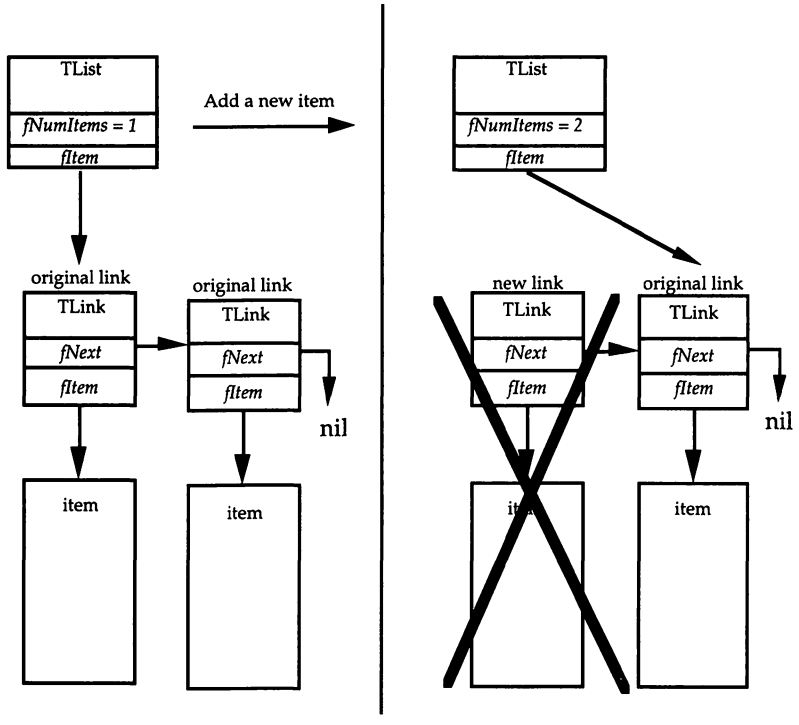


Figure 3-4. Removing the First Item in the List

If the match is not the first link in the list, then **RemoveItem** must change the *fNext* member of the link preceding the removed link so that it points to the link that follows the removed link, as shown by Figure 3-5.

In either case, **RemoveItem** uses the **delete** operator to deallocate the link that it is removing. The **delete** call is necessary since the link object was originally allocated with the **new** operator. The code for **RemoveItem** is shown as follows.

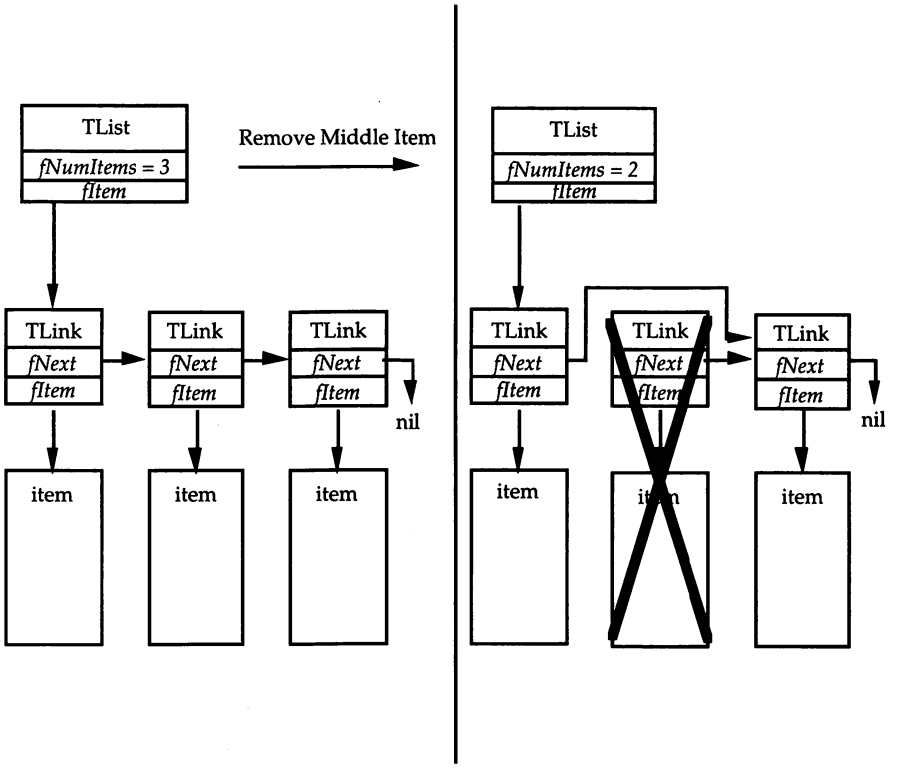


Figure 3-5. Removing the Middle Item in the List

```
void TList::RemoveItem(void* item){
   TLink* temp;
   TLink* last;
   last = nil;
   for (temp = fLink; temp != nil; temp = temp->GetNext())
      if (temp->GetItem() == item){
         // if first item in list, just set first
         if (last == nil)
            fLink = temp->GetNext();
         else
            last->SetNext(temp->GetNext());
         delete temp;
         fNumItems--;
         return;
      } else
         last = temp;
}
```

As defined here, **TList** can be used to keep lists of almost any kind of object. The items in the list are referred to by void pointers, so the list and link classes don't care what the item pointers actually point to. This makes the list class very flexible. Chapter 6 shows how to derive a specialized list class from **TList** to keep a list of a specific kind of object.

Key Point ▶

The interesting thing about the TList class is that the public interface contains no information about how the list is actually implemented. Thus, if you needed to recode TList to use a faster or more memory efficient list management scheme, you could do it without changing the functions that use TList. The implementation details of how the list is managed are hidden from the outside user. This kind of separation of usage and implementation is one hallmark of good design. The public, protected, and private sections of a class declaration make it easier to enforce separation of usage and implementation.

▶ Iterating on a List

It is often useful to be able to perform the same operation on all items in a list. An iterator object is a traditional C++ way of iterating on the items in a list. The iterator is a class that knows how to access the internals of the list class. The iterator is initialized with a pointer to the list and then returns each item in turn, keeping track of how far down the list it has traveled.

The class **TIterator** is declared as follows.

```
class TIterator {
   TLink* fCurLink;
public:
   TIterator(TList* list)
      { fCurLink = list->fLink; }
   void* Next(void);
};
```

It has a single member, *fCurLink*, that keeps track of how far down the list the iteration has progressed. It has a constructor that takes a **TList** pointer as an argument and initializes the *fCurLink* member to point to the first link in the list. It then declares the **Next** member function, which is responsible for returning the next item in the list.

Each time you call **Next**, it returns the item for the current link and then updates the *fCurLink* member to point to the next link in the list. When *fCurLink* is equal to nil, which will happen after the last list item has been returned or if the list is empty, **Next** simply returns nil. The following code shows how **Next** returns the item and advances *fCurLink* to point to the next link.

```
void* TIterator::Next(void){
   TLink* link = fCurLink;
   if (fCurLink) {
      fCurLink = fCurLink->GetNext();
      return (link->GetItem());
   } else
      return nil;
}
```

A typical use of the **TIterator** class is to repeatedly call **Next** until it returns nil, indicating that the end of the list has been reached. Each time **Next** returns a valid pointer to an item, you can do something with that pointer, as shown in the following code fragment. Because **Next** returns a void pointer, you will have to typecast the pointer to point to a particular object type before you can use it. See Chapter 6 for an example of how an iterator can be used with a particular object type.

```
void DoToAll(TList * theList){
   void * theItem;
   // create and initialize an iterator
   TIterator theIterator(theList);
   while(theItem = theIterator.Next() != nil){
      // do something with theItem ...
   }
}
```

If you try to use **TIterator** with **TList** as described in the previous discussion, the compiler will complain that **TIterator** cannot access the *fLink* member of the **TList** class since *fLink* is a protected member. You have three choices here: First, you can make *fLink* a public member. Second, you could define a public access member function to retrieve the value of *fLink*. Third, you could make **TIterator** a friend class to **TList**. The first two solutions defeat the separation of usage and implementation that is supported by the public/private segregation in **TList**. Making **TIterator** a friend is the best solution, since it maintains the protected nature of *fLink* to all other classes. You must change the declaration of **TList** as follows to specify **TIterator** as a friend class.

```
class TList {
protected:
   friend   class TIterator;
   TLink*   fLink;      // the first link in our list
   int      fNumItems;  // the number of elements in the list
public:
   TList(void);     // constructor
   virtual void   AddItem(void* item);
   virtual void   RemoveItem(void* item);
   int NumItems() { return fNumItems; }
};
```

▶ Summary

Object-oriented programming can open up a whole new world of programming ideas to you. The ability to define base classes and then derive other classes from the base classes is an extremely powerful technique for program design.

The public, protected, and private protection levels offered by C++ permit you to hide some of the details of a class's implementation from outside users. This makes it easier to change the internals of a class without changing its external public interface.

This chapter did not describe all the features of C++ that are related to object-oriented programming. Some of the more advanced or subtle features, such as operator overloading and virtual base classes, were left out on purpose to simplify the discussion. The features that were discussed, however, seem to be the most useful for getting started with object-oriented programming in C++, as illustrated by the examples in the remaining chapters.

The rest of this book is filled with examples of how well-designed base classes and derived classes can help you write great programs. You will find that you will spend a considerable amount of time designing the original base classes, but that investment will pay off when you create the derived classes that actually make up your programs. This is a key characteristic of object-oriented programming — you will spend more time designing your program and less time actually implementing it and even less time debugging it.

4 ▶ Writing MPW Tools in C++

The previous chapters discussed general concepts of C++ and gave short example fragments of C++ code. This chapter shows you how to write complete C++ programs that run in the MPW environment. The programs in this chapter are MPW tools, which means that they cannot run outside the MPW environment.

Writing MPW tools is like writing programs in a traditional UNIX environment. You can use streams or printf to print text messages to the screen. MPW windows are like miniterminals in a more traditional text–only computing environment. You can usually take C++ example programs that are intended for traditional systems and compile them unchanged as MPW tools. This makes MPW an ideal learning environment for beginning C++ programmers.

The examples in this chapter focus on the more generic aspects of C++ programming. The programs created here can be run on almost any system that supports C++ with minimal modification. Very few Macintosh-specific features are used in these programs. Chapters 5-14, however, develop classes and programs that address the special characteristics of the Macintosh more directly.

▶ Helloworld

It is traditional that your first C program should print out the words "hello world." The following C program shows how this is usually done.

```
/*
 * HelloWorld.c
 * A simple program in C
 * This program prints the words "hello world" to standard output
 * January 1990, Dan Weston
 */
#include <stdio.h>
void main(){
   printf("hello world\n");
}
```

The following C++ program does the same thing as the C program just shown.

```
// HelloWorld.cp
// A very simple C++ program
// This program prints the words "hello world" to standard output
// January 1990, Dan Weston
#include <iostream.h>
void main(void){
   cout << "hello world\n";
}
```

Comparing the two versions of the program, you can see that both begin with comments that state what the program is supposed to do, who wrote it, and when it was written. Next, notice the #include directives, which tell the compiler to read in a file before attempting to compile the rest of the code. The C program includes the file stdio.h (standard I/O) to get the declarations and definitions that enable it to use the printf function. The C++ program includes the file iostream.h, which contains the declarations and definitions necessary for using streams for simple input and output. In particular, iostream.h contains the definition for a global **ostream** variable named *cout* that is used in the body of the code for output.

Create a new file named Helloworld.cp in MPW and type in the C++ code just shown. The next section explains how to create a makefile and then compile, link, and execute the program.

▶ Creating a Makefile

Once you have created a file containing the code for the C++ Helloworld program, you need to compile it and link it with the proper libraries. The easiest way to do this is to create a makefile for the program using the CreateMake tool that is part of MPW. Choose the Create Build Commands... menu item from the Build menu in MPW to invoke the CreateMake tool, or simply enter the CreateMake... command from any MPW window.

By The Way ▶

The ellipsis (...) after CreateMake tells MPW to show a Commando dialog to get the command arguments for the tool. You type an ellipsis in MPW with the Option-Semicolon key combination.

When you invoke CreateMake, either from the Build menu or directly from an MPW command window, the Commando dialog shown in Figure 4-1 will appear. Fill in the name of the program, Helloworld, select the Tool option for program type, and choose the file Helloworld.cp after clicking on the Source Files... button. When you have done all this, click on the CreateMake button in the lower right corner of the dialog.

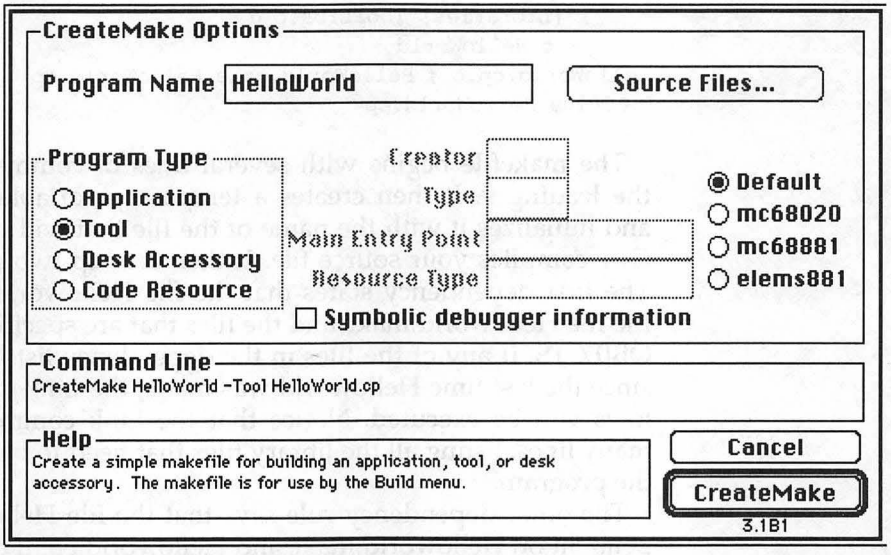


Figure 4-1. The Commando Dialog for CreateMake

CreateMake will examine the information passed to it by the Commando dialog and create a makefile for you. The default name for the makefile is the name of your program with .make appended to it. When given the input specified in the previous paragraph, CreateMake will create the file Helloworld.make, which is listed as follows.

```
#   File:       HelloWorld.make
#   Target:     HelloWorld
#   Sources:    HelloWorld.cp
#   Created:    Monday, January 29, 1990 9:36:51 AM

OBJECTS = HelloWorld.cp.o
HelloWorld ƒƒ HelloWorld.make {OBJECTS}
   Link -w -c 'MPS ' -t MPST ∂
      {OBJECTS} ∂
      "{CLibraries}"CSANELib.o ∂
      "{CLibraries}"Math.o ∂
      "{CLibraries}"CplusLib.o ∂
      #"{CLibraries}"Complex.o ∂
      "{CLibraries}"StdCLib.o ∂
      "{CLibraries}"CInterface.o ∂
      "{Libraries}"Stubs.o ∂
      "{CLibraries}"CRuntime.o ∂
      "{Libraries}"Interface.o ∂
      "{Libraries}"ToolLibs.o ∂
      - o HelloWorld
HelloWorld.cp.o ƒ HelloWorld.make HelloWorld.cp
   CPlus HelloWorld.cp
```

The makefile begins with several lines of comments, identified by the leading #. It then creates a temporary variable name, OBJECTS, and initializes it with the name of the file that will be produced when C++ compiles your source file. It then sets up two dependency rules. The first dependency states that the file Helloworld is dependent on the file Helloworld.make and the files that are specified by the variable OBJECTS. If any of the files in the dependency list has been modified since the last time Helloworld was made, the Link instruction that follows will be executed. Notice that the Link command extends over many lines, listing all the library files that need to be included to make the program.

The other dependency rule says that the file Helloworld.cp.o is dependent on Helloworld.make and Helloworld.cp. If either of these files has been modified since Helloworld.cp.o was last built, the CPlus script will be invoked to recompile the program.

CreateMake is a big help when you are getting started with MPW because it knows lots of details about the build process that would not be immediately obvious to you, such as the list of files that are necessary for the link phase. For the projects in this chapter you can use the makefile created by CreateMake just as it is, or with only minor modifications. It is very helpful to use CreateMake to create the initial makefile and then make modifications to that file to handle special circumstances of your program's dependencies, as illustrated later in this chapter. In later chapters the dependencies become more complex and require custom makefiles.

Key Point ▶

It is important to name your MPW C++ source files with the .cp extension so that CreateMake will be able to differentiate them from normal C files and thus produce the proper instructions for building the program.

Once you have created the makefile, you can build your program by choosing the Build... menu item from the Build menu, specifying the name of the program to the resulting dialog box. Another way of accomplishing the same thing is to issue the command BuildProgram Helloworld from any MPW window.

While the program is being built, MPW will print progress information to the window. When the process is done, you can select the program name and press the Enter key to see your program run. It should respond by printing the words "hello world" in the window from which it is run.

So there you have it — your first C++ program on the Macintosh. Most of the work revolved around creating the makefile and actually building the program rather than writing the program code. The following sections develop more complicated programs, but the process of creating the makefile and building the programs will remain essentially the same.

▶ I/O Redirection in MPW

As we have seen, one big advantage of writing MPW tools is the ability to do simple text output using streams. In addition, MPW supports redirected input and output to tools. Every time a tool runs, it has three predefined streams available to it: *cout* for output, *cin* for input, and *cerr* for error message output. By default, *cin*, *cout*, and *cerr* are attached to the window from which the tool was run. But MPW lets you hook

these standard input and output streams to other sources. The most common use of redirection is to attach *cin* and *cout* to disk files. This is done by using the redirection symbols < for input and > for output. For example, if you invoke the Helloworld program with the command line

```
Helloworld > myFile
```

the output message from Helloworld would go into the file myFile instead of to the window. If the file does not exist, MPW would create it for you. If the file already exists, the output from Helloworld would replace the previous file contents. You can also use the append symbol, >>, to add output to the end of a file instead of replacing it.

I/O redirection is a very productive way to write tools that read input from the standard input stream, perform some operation on the incoming data, and output the converted data to the standard output stream. In the simplest case, where no redirection occurs, the input and output both come from the MPW window where the program was launched (or from a terminal in a more traditional UNIX environment). With redirection, the program can read from one file and output to another file. Because MPW does all the work of opening, and possibly creating, the specified files, the program that does the data conversion can be extremely simple — it merely reads from *cin* and writes to *cout*. The program doesn't need to know the actual source and destination for the data.

To illustrate this concept, consider the following problem. C++ allows you to use a double slash // to mark comments. Most C compilers do not accept double-slash comments (Apple's MPW C compiler being a notable exception). Think C 4.0, which has partial support for object-oriented programming and uses some of the same syntax as C++, does not accept double-slash comments. It would be useful to be able to read in a C++ source file and change all the double-slash comments to traditional C comments, delimited by /* and */.

This kind of operation can be thought of as a filter. The original source file is read, one character at a time. When a double-slash comment is found, it is replaced by a traditional C comment. The filtered characters are written to a new file. Because this program will use standard input and output streams, it won't have to be concerned with file names or with any other file system housekeeping. It will simply get characters from *cin* and put characters to *cout*. MPW will take care of making sure that these streams are attached to the indicated source and destination.

The following program code shows how this filter can be written. Notice that it includes the header file iostream.h so that the predefined stream variables *cin* and *cout* are available. It then repeatedly calls the **Get** member function for *cin* to look at each character from the source file. The **extraction** operator (>>) is not used for input because it skips white space, including new-line characters. Most of the time, the incoming character is simply passed through to the output stream by using the *insertion* operator (<<) on the *cout* stream. Whenever the character coming in looks like it might be the start of a double-slash comment, extra processing is done to replace the C++ comment with the equivalent C comment.

```
////////////////////////////////////////////////////////////////
//
// fixcom.cp
//
// Changes C++ style comments to C comments
//
// Uses cin and cout streams
//
// invoke with redirection from MPW, such as
//
// fixcom < foo.cp > foo.fixed
//
// ©1989 Dan Weston, all rights reserved
//
////////////////////////////////////////////////////////////////

#include <iostream.h>
int main(void){
   char c;
   char nextc;
   while (cin.get(c)){
      if (c != '/'){
         // most chars just pass right through filter
         cout << c;
      } else {
         // this may be a double-slash comment...
         // check next char following first '/'
         cin.get(nextc);
         if (nextc != '/'){
            // not a double-slash comment,
            // just output the '/' and the following char
            cout << c << nextc;
```

```
        } else {
            // it is a double-slash comment,
            // substitute opening C comment
            cout << '/' << '*';
            // pass chars through until end of line
            cin.get(c);
            while (c != '\n'){
               cout << c;
               cin.get(c);
            }
            // now insert a closing comment
            cout << ' ' << '*' << '/';
            // and send the newline char out too
            cout << c;
        } // end nextc != '/' else
    } // end c != '/' else
  } // end while
  // make sure all output is flushed
  cout << flush;
  return 0;
}
```

This program is called fixcom. You can create a makefile for it and build it using the techniques described for the Helloworld program. Once it is built, you can run it with the following command in MPW.

```
fixcom < fixcom.cp > fixcom.fixed
```

This command line redirects input from fixcom.cp and redirects output to fixcom.fixed. The output file is listed as follows. You can see how the C++ comments have all been changed to C comments.

```
/*////////////////////////////////////////////////////////////// */
/* */
/* fixcom.cp */
/* */
/* Changes C++ style comments to C comments  */
/*  */
/* Uses cin and cout streams */
/* */
/* invoke with redirection from MPW, such as */
/* */
```

```
/* fixcom < foo.cp > foo.fixed */
/* */
/* ©1989 Dan Weston, all rights reserved */
/* */
/*////////////////////////////////////////////////////////////// */

#include <iostream.h>
int main(void){
   char c;
   char nextc;
   while (cin.get(c)){
      if (c != '/'){
         /* most chars just pass right through filter */
         cout << c;
      } else {
         /* this may be a double-slash comment... */
         /* check next char following first '/' */
         cin.get(nextc);
         if (nextc != '/'){
            /* not a double-slash comment, */
            /* just output the '/' and the following char */
            cout << c << nextc;
         } else {
            /* it is a double-slash comment,  */
            /* substitute opening C comment */
            cout << '/' << '*';
            /* pass chars through until end of line */
            cin.get(c);
            while (c != '\n'){
               cout << c;
               cin.get(c);
            }
            /* now insert a closing comment */
            cout << ' ' << '*' << '/';
            /* and send the newline char out too */
            cout << c;
         }  /* end nextc != '/' else */
      } /* end of c != '/' else */
   } /* end while */
   /* make sure all output is flushed */
   cout << flush;
   return 0;
}
```

If you invoke fixcom without specifying any redirection for output, the output will go to the window from which the tool was run. If you don't specify any input redirection, the program will accept keyboard input until it sees an end-of-file character, which you can generate in MPW by pressing the Command-Enter key combination. You must either end each input line with the Enter key, or select a range of text and press the Enter key before the text will be passed to the tool through standard input.

Key Point ▶

One important thing to know about keyboard input from an MPW window is that MPW tries to use the current selection in the window as input to the tool when the Enter key is pressed. If there is no current selection, MPW will send the entire line of text containing the insertion point to the tool when you press the Enter key. This can cause problems when you are porting programs from a UNIX environment (for instance, when you are typing in programs from other C++ books). For example, consider a program that prompted the user for a temperature in degrees Celsius and output the temperature in degrees Fahrenheit. It might look something like this.

```
float convert(float temp_c);
float temp_C;
float temp_F;
cout << "Enter temperature in °Celsius: ";
cin >> temp_C;
temp_F = convert(temp_C);
cout << "\n" << temp_C << " °C = " << temp_F << " °F\n";
```

As shown here, the program won't run successfully in an MPW window because the user input is typed in on the same line as the "Enter temperature ..." prompt. If the user selects the input text before pressing Enter, everything works as expected. But if the user simply types in the temperature and presses the Enter key without first selecting the input text, MPW will send the entire line, including the prompt text, to the tool. The input stream will not be able to successfully format the combined prompt text and input value into a float variable, and the program will not give the expected results. The solution to this problem is simple: Just make certain that the user's input is entered on a line by itself. To fix the previous example, add a new line at the end of the prompt, as shown here.

```
cout << "Enter temperature in °Celsius:\n";
```

I/O redirection is a very elegant and powerful way to write filter programs such as fixcom. These tool programs are flexible because they can accept input from a file or directly from the keyboard. Likewise, their output can go to a specified file or to an MPW window.

By The Way ▶

During one particularly acrimonious discussion among several programmers regarding the relative merits of mouse-based editors versus more traditional editors such as vi, a level head in the room remarked that real programmers don't use editors — they just type their code directly to the compiler through the standard input channel. She said she did all her programming by simply invoking the C++ compiler with the following command, without specifying any input file.

```
cplus
```

She then typed in her code directly, ending each line with the Enter key and ending the final line with the Command-Enter key combination. Try it.

▶ Command Line Arguments

Unfortunately, tools that use redirected input and output are, in practice, clumsy to use. Tool programs are more frequently written to accept arguments from the command line. The command line arguments typically specify the names of the input and output files and other options that can change the operation of the tool. For example, if the fixcom program was written to accept command line arguments, it would probably be invoked with the following command.

```
fixcom foo.cp -o foo.fixed
```

The first argument is the name of the input file. The next argument is an option, -o, that tells the tool that the following argument will be the name of the output file. With a command line interface to the tool, it is possible to add options to the tool. For example, it might be handy to add an option to fixcom that caused it to delete comments from source files.

MPW passes command line arguments to the main function of tool programs in two function arguments, *argc* and *argv*. The *argc* argument is the number of arguments on the command line, including the program name. The *argv* argument is a pointer to an array of pointers.

Each pointer in the array points to a C string (null-terminated) containing one argument from the command line.

You must define your main function as follows to be able to process the information passed to your tool from MPW.

```
int main(int argc, char* argv[]){
   // ...
}
```

The first string in the *argv* array is always the name of the program. The rest of the strings are taken from the command line from left to right. For example, the command line

```
fixcom foo.cp -o foo.fixed
```

would result in *argc* and *argv* as shown in Figure 4-2.

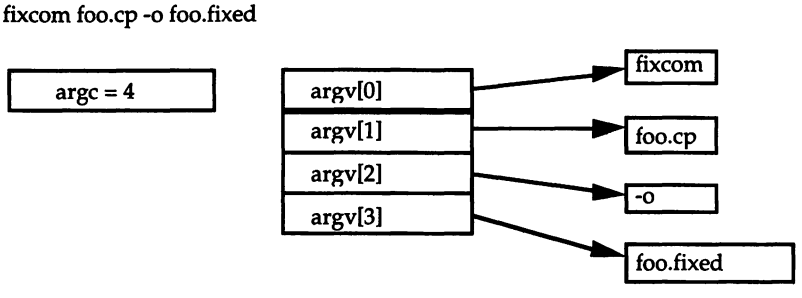


Figure 4-2. Command Line Arguments Passed to a Tool

The next section of this chapter describes a class that knows how to parse arguments off the command line. This class will make it easier to use the command line, allowing you to concentrate on the details that are specific to your particular tool. A later section of this chapter develops a version of fixcom that uses command line arguments.

▶ The TTool Class

The programs described in the previous sections are written in C++ but other than the fact that they use streams for input and output, they could just as well have been written in C. This is one of the main characteristics of C++ — you can use it just like C. But in order to get the full benefit of C++, you should also use the object-oriented aspects of C++.

This section develops a class that can be used to write MPW tool programs. The **TTool** class's main job is to initialize the tool and then pull arguments off the command line and pass them to member functions that can process the arguments as appropriate for the particular tool. This section will describe the **TTool** base class. Later sections will show how to derive classes from **TTool** to implement specific tool programs.

The file TTool.h contains the declaration of the **TTool** class. TTool.h begins by including several header files to get the declarations for Macintosh toolbox and C++ library functions that the **TTool** class uses. The following include statements show the files that are used by **TTool**.

```
#include <Quickdraw.h>
#include <Fonts.h>
#include <CursorCtl.h>
#include <iostream.h>
#include <fstream.h>
#include <FCntl.h>
```

Quickdraw.h, Fonts.h, and CursorCtl.h contain Macintosh-specific function declarations that are used by the tool class. The last three files listed contain C++ library function declarations for the stream and file-related calls that the tool class uses.

The **TTool** class is declared as follows.

```
class TTool {
protected:
   int fArgc;
   int fCurrentArg;
   char ** fArgv;
   char * fProgName;
   char * fNextArg;
public:
   virtual void ITool(int argc, char* argv[]);
   virtual int Run(void);
protected:
   // these three functions will probably NOT be overridden
   virtual char* GetNextArg(void);
   virtual fstream * MakeStream(char * fileName,int permission);
   virtual int ParseArguments(void);
   // these three functions will most likely be overridden
   virtual int SetOption(char * option){return 1;}
   virtual int HandleArg(char * arg){return 1;}
   virtual int DoWork(void){return 0;}
};
```

The class contains five members and eight member functions. They will be explained individually in the following sections. Notice that all the member functions are virtual so that they can be overridden in derived classes if necessary. Notice also that there are only two public member functions; all other members and member functions are protected.

▶ Initializing and Running the Tool

The **ITool** member function is responsible for initializing the tool. It takes two arguments, *argc* and *argv*, which contain the information passed to the tool from the MPW command line, as explained in an earlier section. **ITool** uses these arguments to set the initial value of several of the class members. **ITool** then goes on to call the toolbox functions InitGraf and SetFScaleDisable. These two functions are necessary if your tool will call any QuickDraw routines (including the function Random), so they are included in the base class just in case. Finally, **ITool** calls the MPW tool library functions InitCursorCtl and SpinCursor to initialize the "spinning beach ball" cursor that is typically used in MPW tools. The definition for **ITool** is shown as follows.

```
void TTool::ITool(int argc, char* argv[]){
   fArgc = argc;
   fArgv = argv;
   fProgName = *fArgv++;
   fCurrentArg = 1;
   fNextArg = 0;
   // just in case you want to use QuickDraw
   InitGraf(&qd.thePort);
   // MPW tool documentation says to call this next function
   SetFScaleDisable(true);
   InitCursorCtl(nil);
   SpinCursor(1);
}
```

Some of the member initialization done in **ITool** could have been done in a constructor, but this class does not define a constructor since that would complicate the derivation process. The **TDoc** class in Chapter 5 shows how to use a constructor to perform member initialization.

The **Run** member function is called once the tool has been initialized. It calls the **ParseArguments** member function to process the command line arguments and then calls the **DoWork** member function to actually do the work that the tool is supposed to do. **ParseArguments** and **DoWork** are explained in later sections. The definition for **Run** is shown as follows.

```
int TTool::Run(void){
   if(ParseArguments())
      return DoWork();
   else
      return 1;
}
```

ITool and **Run** are the only two public member functions for **TTool**. Thus, they are the only functions that can be called from outside the class. **TTool** (or one of its derived classes) is usually used as shown in the following code. A **TTool** object is created, **ITool** is called, and then **Run** is called. The result of **Run** is passed back to MPW as the overall result of the tool. Most of the details of how the tool class works internally are hidden from the user in protected member functions.

```
int main(int argc, char* argv[]){
   TTool aTool;
   aTool.ITool(argc,argv);
   return aTool.Run();
}
```

Of course the example just shown wouldn't do anything useful, since the **DoWork** member function is defined by default to do nothing and must be overridden to perform the work of your specific tool.

▶ Parsing Command Line Arguments

As mentioned earlier in this chapter, MPW passes command line arguments to a tool in an array of string pointers and a counter. **ITool** uses this array and counter to initialize the *fArgv* and *fArgc* members. **ITool** also initializes its own counter member, *fCurrentArg*, to keep track of which arguments have been processed. **ITool** takes the first string from the array and uses it as the program name. So by the time **ITool** has finished, *fCurrentArg* is equal to 1 (since the counter is zero-based).

The **GetNextArg** member function is defined to return the next available argument from the *fArgv* array. **GetNextArg** is responsible for updating the *fCurrentArg* member and comparing it to the overall argument count so that it will know when the last argument has been retrieved. **GetNextArg** returns the string pointer for the next argument, or zero if all the arguments have been processed. The definition for **GetNextArg** is shown as follows.

```
char* TTool::GetNextArg(void){
   if(fCurrentArg++ < fArgc){
      fNextArg = *fArgv++;
      return fNextArg;
   } else
      return 0;
}
```

ParseArguments repeatedly calls **GetNextArg** to get each command line argument in turn. It examines the first character of each argument. If the argument begins with a dash, '-', then **ParseArguments** assumes that the argument is an option flag and passes the argument to the **SetOption** member function. Otherwise, the argument is passed to the **HandleArg** member function for processing. Both **SetOption** and **HandleArg** return a value indicating if the argument could be successfully processed, and **ParseArguments** examines these return values to detect any error caused by an invalid argument. The definition of **ParseArguments** is shown as follows.

```
int TTool::ParseArguments(void){
   char * arg;
   while((arg = GetNextArg()) != 0){
      if(*arg == '-') {
         if(SetOption(arg) == 0)
            return 0;
      } else {
         if(HandleArg(arg) == 0)
            return 0;
      }
   }
   // signal success
   return 1;
}
```

SetOption and **HandleArg** are both defined by default to do nothing except return 1, indicating success. If your tool expects to use command line arguments, you must override these two member functions to process arguments and options. A later section of this chapter shows an example of how to do this.

▶ Making Streams for Input and Output

When handling command line arguments, it is often necessary to take a file name from the command line and open the file and create a stream for the file. The **MakeStream** member function is provided in **TTool** for just this purpose. It takes a file name as an argument, opens the file (creating it first if necessary) and then creates an **fstream** attached to the file. An **fstream** is derived from the **iostream** base class. **MakeStream** encapsulates all the gritty details and error-reporting that you don't want to recreate in every tool you write. You will probably not need to override **MakeStream**. Its definition is shown as follows; its use is shown in a later section of this chapter.

```
fstream * TTool::MakeStream(char * fileName,int permission){
   const int BUFFSIZE = 1024;
   int fd = open(fileName,permission);
   if(fd == EOF){
      fd = create(fileName);
   }
   if(fd != EOF){
      char * buff = new char[BUFFSIZE];
      if(buff == 0){
         cerr << "### " << "error making stream\n";
         return 0;
      }
      fstream * fs = new fstream(fd,buff,BUFFSIZE);
      if(fs == 0){
         cerr << "### " << "error making stream\n";
         return 0;
      }
      return fs;
   }
   cerr << "### " << "error opening file " << fileName <<"\n";
   return 0; // failed to open file
}
```

Notice that **MakeStream** uses the predefined error stream *cerr* to report error messages to the user. Unless redirected, these error messages will go to the MPW window from which the tool was run. The *cerr* stream is also a very useful way to display debugging messages while developing an MPW tool.

▶ I/O Redirection with TTool

Now that the **TTool** class is defined, you can use it to write a simple tool. Let's rewrite the fixcom tool described earlier in this chapter. The new tool, fixcom2, will use I/O redirection just like the original fixcom. Another revision of fixcom that takes command line arguments will be developed in a later section.

To create fixcom2 you must derive a new class from **TTool**, overriding the **DoWork** member function and adding a new member function to do the comment filtering.

▶ Deriving a New Class from TTool

Both the declaration and definition of **TTool** are contained in the file TTool.h. Usually, the declaration would be in TTool.h and the definition would be in a file named TTool.cp, but because the function definitions for **TTool** are rather short, it is easier to put them all in the same file with the class declaration. In later chapters, where the declaration for the **TDoc** class might be used in several files, a more rigorous separation of declaration and definition is used.

The definition of the new class for fixcom2 is shown as follows. Notice that you must include the file TTool.h so that you can create derived classes from **TTool**. Notice also that **DoWork** is defined in the declaration to call the **Filter** member function, which is a new function for this new class for fixcom2.

```
////////////////////////////////////////////////////////////////
//
// fixcom2.cp
//
// Changes C++ style comments to C comments
//
// Uses MPW input and output redirection
//
// invoke with redirected input file and output file names
//
// fixcom2 < foo.cp > foo.c
//
// ©1990 Dan Weston, all rights reserved
//
////////////////////////////////////////////////////////////////
#include "TTool.h"
class TFixComment2 : public TTool {
```

```
public:
 int DoWork(void)
  {return Filter(cin,cout);}
protected:
 int Filter(istream& in,ostream& out);
};
```

▶ The Filter Member Function

Filter is the new member function defined for the **TFixComment2** class. It takes an **istream** and an **ostream** as arguments and uses them for input and output. The code for **Filter** is almost exactly like that of the original fixcom program, except that it has been generalized so it uses arguments to identify the input and output streams rather than hard-coding them to *cin* and *cout*. It also includes a call to the tool library function SpinCursor each time a character is retrieved from the input file. SpinCursor spins the beach ball cursor to show the user that the program is still running, and it also allows the user to switch to any other program running under MultiFinder. The code for **Filter** is shown as follows.

```
int TFixComment2::Filter(istream& in,ostream& out){
   char c;
   char nextc;
   while (in.get(c)){
      SpinCursor(1);
      if (c != '/'){
         // most chars just pass right through filter
         out << c;
      } else {
         // this may be a double-slash comment...
         // check next char following first '/'
         in.get(nextc);
         if (nextc != '/'){
            // not a double-slash comment,
            // just output the '/' and the following char
            out << c << nextc;
         } else {
            // it is a double-slash comment,
            // substitute opening C comment
            out << '/' << '*';
            // pass chars through until end of line
            in.get(c);
```

```
            while (c != '\n'){
               out << c;
               in.get(c);
            }
            // now insert a closing comment
            out << ' ' << '*' << '/';
            // and send the newline char out too
            out << c;
         }  // end nextc != '/' else
      } // end of c != '/' else
   } // end while

   // make sure all output is flushed
   out << flush;
   return 0;
}
```

Notice that **Filter** uses reference arguments to identify the in and out streams. Reference arguments are much easier to use than pointers in this function since all stream operators expect to operate on a stream rather than on a pointer to a stream. See Chapter 2 for a discussion of reference arguments.

In fixcom2, the **DoWork** member function calls **Filter** with *cin* and *cout* as arguments, so the effect is exactly the same as in the original fixcom program. However, generalizing **Filter** to use arbitrary input and output streams will pay off in the next version of fixcom, where file names from the command line will be used to create streams for input and output.

▶ The Main Function

As shown in an earlier section, the **TTool** class has only two public member functions. So to use **TTool** or any of its derivatives, you must call **ITool** and then **Run**, as shown by the following definition for the main function of fixcom2.

```
int main(int argc, char* argv[]){
   TFixComment2 fixComTool;
   fixComTool.ITool(argc,argv);
   return fixComTool.Run();
}
```

▶ Makefile for Fixcom2

You build fixcom2 using the same processes described for Helloworld earlier in this chapter. You can use CreateMake to create a makefile for fixcom2 and then build the program. Because CreateMake will not permit you to specify header files on which the program is dependent, you must add those dependencies to the makefile by hand. The modified makefile for fixcom2 is shown as follows. Notice that the file TTool.h has been added to the dependency list for fixcom2.cp.o. It is a common practice to modify makefiles created with CreateMake to make them more accurately reflect the dependencies of your program. Complete code and makefile listings for fixcom2 are listed in Appendix B.

```
#   File:       fixcom2.make
#   Target:     fixcom2
#   Sources:    fixcom2.cp
#   Created:    Friday, January 19, 1990 11:58:21 AM

OBJECTS = fixcom2.cp.o

fixcom2 ƒƒ fixcom2.make {OBJECTS}
   Link -w -c 'MPS ' -t MPST ∂
      {OBJECTS} ∂
      "{CLibraries}"CSANELib.o ∂
      "{CLibraries}"Math.o ∂
      "{CLibraries}"CplusLib.o ∂
      #"{CLibraries}"Complex.o ∂
      "{CLibraries}"StdCLib.o ∂
      "{CLibraries}"CInterface.o ∂
      "{Libraries}"Stubs.o ∂
      "{CLibraries}"CRuntime.o ∂
      "{Libraries}"Interface.o ∂
      "{Libraries}"ToolLibs.o ∂
      - o fixcom2
fixcom2.cp.o ƒ fixcom2.make fixcom2.cp TTool.h
   CPlus  fixcom2.cp
```

▶ Compiler Warnings

As you compile fixcom2.cp, the following warning messages will appear.

```
File "TTool.h"; line 51 # warning: option not used
File "TTool.h"; line 52 # warning: arg not used
```

If you look at lines 51 and 52 in TTool.h, you'll see

```
virtual int SetOption(char * option){return 1;}
virtual int HandleArg(char * arg){return 1;}
```

The warning is generated because **SetOption** and **HandleArg** each take an argument that is not used in the body of the function. Neither of these member functions need their arguments since they are just stubs, but the arguments may be needed when the functions are overridden in derived classes. Sometimes these warning messages can be very helpful in identifying unnecessary function arguments (and local variables that aren't used), but in this case the warnings can be ignored. If the warnings get to be too bothersome, you can turn them off by adding the –w flag when calling the CPlus script.

Key Point ▶

A better way to turn off the unused argument warning is to comment out the name of the argument (leaving its type designator) in the definition of the function. If a function has an unnamed argument in its definition, C++ assumes that the argument won't be used (in fact, it can't be used because it has no name to refer to it in the function body). For example, you could comment out the argument name in **TTool:SetOption**, as shown here, to prevent warnings.

```
virtual int SetOption(char * /*option*/){return 1;}
```

▶ Reading the Command Line with TTool

Fixcom2 used **TTool**, but it didn't show much advantage over the original version largely because **TTool** contains more functionality than fixcom2 needed. The following sections develop a third version of fixcom, fixcom3, that takes arguments from the command line and demonstrates the usefulness of **TTool**. The command line arguments are assumed to be an input file name and an output file name. The output file name is preceded by the -o option flag. If either the input or output file names are not specified, standard input or standard output is used.

You begin fixcom3 by defining a new class derived from **TTool**, as shown in the following code.

```
////////////////////////////////////////////////////////////////
//
// fixcom3.cp
//
// Changes C++ style comments to C comments
//
// Uses MPW command line for input and output file
//
// invoke with input file name and -o output file names:
//
// fixcom3 foo.cp -o foo.c
//
// ©1990 Dan Weston, all rights reserved
//
////////////////////////////////////////////////////////////////
#include "TTool.h"
class TFixComment3 : public TTool {
protected:
   istream *fIn;
   ostream *fOut;
public:
   void ITool(int argc, char* argv[]);
   int DoWork(void);
protected:
   int SetOption(char * option);
   int HandleArg(char * arg);
   int Filter(istream& in,ostream& out);
};
```

Two new members, *fIn* and *fOut*, are added to the new class to hold pointers to the input and output streams. The new class also overrides the **ITool**, **SetOption**, and **HandleArg** member functions. Like fixcom2, fixcom3 adds a **Filter** member function.

▶ Overriding the Initialization Member Function

The overridden **ITool** member function begins by calling the **ITool** function for its parent class. It does this by preceding the name of the function with the name of the parent class, **TTool**, and two colons, as in **TTool::ITool**. After calling the inherited version of the initialization function to take care of the default initialization, the derived version of **ITool** goes on to initialize the *fIn* and *fOut* members so that they point to *cin* and *cout*. These stream pointers are attached to the standard in and out streams by default in case no input or output file is specified on the command line.

```
void TFixComment3::ITool(int argc, char* argv[]){
   // do the inherited stuff first
   TTool::ITool(argc,argv);
   // hook up default input and output
   fIn = &cin;
   fOut = &cout;
}
```

Key Point ▶

The derived ITool member function calls the ITool member function for its parent class, TTool, by preceding the member function name with the class name and two colons, as shown here.

```
TTool::ITool(argc,argv);
```

▶ Processing the Command Line

Once the tool is initialized, **ParseArguments** is called. **ParseArguments** repeatedly takes arguments out of the argument array and passes them to either **SetOption** or **HandleArg**. These two member functions must be overridden to perform the specific command operations required of your tool.

HandleArg assumes that the argument passed to it is the name of the input file. It uses that file name to create a stream by calling the **MakeStream** member function. If the stream is successfully created, **HandleArg** sets the *fIn* member to point to that stream so that it will be used for input later when the **Filter** member function is called. The code for **HandleArg** is shown as follows.

```
int TFixComment3::HandleArg(char * arg){
   // open the file and create a stream for it
   fstream * fs = MakeStream(arg,O_RDONLY);
   if(fs != 0){
      fIn = fs;
      return 1;
   } else
      return 0;
}
```

SetOption is called when **ParseArguments** encounters an argument that begins with a dash (-). In this class, **SetOption** checks the character following the dash to make sure it is an "o". If the option character is valid, then **SetOption** assumes that the next argument is the name of

the output file name. It calls **GetNextArg** to retrieve the output file name from the argument array and attempts to make a stream from it by calling the **MakeStream** member function. If the stream is successfully created, **SetOption** sets the *fOut* member to point to that stream. If the option is not valid, or if the stream cannot be created, **SetOption** returns 0 to indicate failure.

```
int TFixComment3::SetOption(char * option){
    char * arg;
    if(*(++option) == 'o'){
      // get the output file name
      if((arg = GetNextArg()) != 0){
         // open the file and create a stream for it
         fstream * fs = MakeStream(arg,O_WRONLY);
         if(fs != 0){
            fOut = fs;
            return 1;
         } else
            return 0;
      } else{
         cerr << "### " << " missing file name\n" ;
         return 0;
      }
    } else {
      cerr << "### " << option << " is not a valid option\n";
      return 0;
    }
}
```

▶ Calling the DoWork Member Function

If **ParseArguments** does not return an error, the **DoWork** member function is called. Fixcom3 overrides **DoWork** so that it calls the **Filter** member function, supplying the *fIn* and *fOut* members as the input and output stream arguments. Remember that *fIn* and *fOut* were initialized to point to *cin* and *cout*, and then later set to point to input and output files if those files were specified on the command line. The **Filter** function is exactly the same as that used in fixcom2.

```
int TFixComment3::DoWork(void){
   return Filter(*fIn,*fOut);
}
```

Fixcom3 shows how to use the command line parsing capabilities of **TTool** to easily read arguments and attach input and output streams to specified files. This is a very common thing to do in MPW tools. **TTool** uses the object-oriented features of C++ to make writing tools easier.

▶ Summary

MPW tools are a good way to begin your C++ explorations. You can get a feel for how C++ programs are put together and how you can employ the different parts of MPW to create and use makefiles. The programs in this chapter are relatively simple because they rely on MPW to handle most input and output. Once you have mastered the concepts shown in this chapter you will be ready to move on to the more complicated material of the later chapters.

Chapters 5-14 describe how to write stand-alone C++ programs that run independently of the MPW environment and utilize all the features of the Macintosh interface, such as windows and menus. The coming chapters also develop much larger classes to handle the complexities of Macintosh programming.

PART TWO

▶ Total Immersion

The previous four chapters have given you a gentle introduction to C++ and object-oriented programming concepts. The next three chapters will immerse you in a practical exercise to see how C++ can be used to solve big programming problems. This section is like being thrown headfirst into the deep end of a swimming pool.

As mentioned in earlier chapters, C++ is most useful in large, complicated programming projects. The next few chapters display that usefulness by attempting to implement two large classes. You will learn a lot by studying these classes and thinking about the decisions that went into their design. There are things about object-oriented program design that you just can't learn from small program examples.

The chapters in this section develop classes that can be used to help write full-fledged Macintosh programs. The chapters are pretty dense. Be prepared to put the book down halfway through a chapter and pause for reflection. That is alright. The time that you spend working through the example classes in this section and thoroughly understanding them will pay off when you begin to use those classes to develop your own Macintosh programs.

5 ▶ TDoc: The Generic Document Class

This chapter describes a class that encapsulates the concept of a document in a Macintosh application. It is a rather simple model of a document, based on the idea that a document is associated with one file on the disk and has one window in which to display its data, but the model should be sufficient for you to write many useful Macintosh programs. Even though this document model is quite simple, you will find that there is a great deal of code necessary to support a minimal document class. This chapter will take you through the process of designing a large class, while looking at some of the trade-offs you must make between efficiency and flexibility.

This class, named **TDoc**, will be used as the basis for several derived document classes developed in Chapters 7-13. **TDoc** itself is not derived from any other object. It could have been derived from the **HandleObject** type supplied by Apple in order to take advantage of the memory management benefits of handle-based objects, but in Chapter 13 **TDoc** is used to derive a class using multiple inheritance, and **HandleObjects** cannot be used in multiple inheritance hierarchies.

This chapter describes the members and member functions of **TDoc**, but you might also want to look at the code listings in TDoc.h and TDoc.cp in Appendix B to get a better idea of how a large class declaration and definition is put together. You will also want to read Chapter 6, on the **TApp** application class, since the application class makes such extensive use of the document member functions.

▶ TDoc Members

Each document object contains seven members that describe its state. All the members of **TDoc** are *protected*, so they can only be accessed via member functions of **TDoc** or member functions of derived classes. Notice that the members are not *private*, which would mean that derived classes could not access them, or *public*, which would mean that anyone with a reference to the object could change the members.

TDoc's members are declared as follows.

```
class TDoc {
protected:
    OSType      fCreator;
    SFReply     fFileInfo;
    Boolean     fFileOpen;
    short       fRefNum;
    WindowPtr   fDocWindow;
    Boolean     fNeedtoSave;
    Boolean     fNeedtoSaveAs;
```

The first two members, *fCreator* and *fFileInfo*, hold information about the file that contains the data for the document. The next member, *fFileOpen*, tells whether or not the file is currently open. The next member, *fRefNum*, contains the file system reference number for the file if it is open. Other **TDoc** member functions can use this refnum to perform file I/O operations on the file. The next member, *fDocWindow*, refers to the window for the document. Many member functions use this member to operate on the document window. The last two members, *fNeedtoSave* and *fNeedtoSaveAs* tell if the document has changed since the last time it was saved. *fNeedtoSave* is true whenever there is unsaved data in the document. *fNeedtoSaveAs* is true when there is unsaved data and the file is not associated with a disk file (i.e., it was created as a new, blank document). All members are initialized by **TDoc**'s constructor and updated by other member functions during the life of the object.

▶ Constructor and Destructor

As discussed in Chapter 3, the main purpose of a constructor is to initialize members of the object. It is not wise to do anything in a constructor that could fail since there is no way to return an error code from a constructor. The constructor for **TDoc** is declared with the following interface.

```
TDoc(OSType theCreator = '????',SFReply * SFInfo = (SFReply *) nil);
```

The constructor takes two arguments: *theCreator* is an OSType that the document will use if it has to create a new file for the document; *SFInfo* is a pointer to an SFReply structure that describes an existing file for the document. Notice that both arguments have default values in the declaration. This means that you can call the constructor for **TDoc** three different ways, shown as follows.

```
SFReply theInfo;
TDoc * doc1 = new TDoc;  // theCreator and SFInfo both default
TDoc * doc2 = new TDoc('MPNT');  // only SFInfo default
TDoc * doc3 = new TDoc('MPNT',&theInfo);  // neither default
```

Default arguments are substituted in strict left-to-right ordering. Thus, you could *not* call the **TDoc** constructor as follows.

```
TDoc * theDoc = new TDoc(&theReply);    // COMPILER WILL COMPLAIN
```

TDoc objects are typically created by the application in response to the user choosing the New or Open menu items. If the document is created as a new document, then *SFInfo* will be nil, to indicate that there is no existing file for the document. If the document is created from an Open menu choice, then *SFInfo* will point to a valid SFReply containing information about the file obtained from the toolbox function SFGetFile so that the document can later open the file and read and write its contents.

The Macintosh operating system assigns all files a creator signature and a file type signature. This supports the association of applications with document files so that a user can open an application by opening one of its documents in the Finder. File creator and file type signatures also allow for application specific icons on the Finder desktop. The Scribble application, described in Chapter 8, shows how to use unique creator and type identifiers to enable these features.

The *theCreator* argument to **TDoc**'s constructor allows the caller (usually the application object) to specify the creator signature for the document. The document object uses the creator signature in case it needs to create a new file when saving the document.

The constructor for **TDoc** initializes the object members to their default values. The *fNeedtoSaveAs* member is set according to whether or not the document was created from an existing file, as indicated by the value of the *SFInfo* argument (nil indicating that there is no file associated with this document yet). The code for **TDoc**'s constructor is shown as follows.

```
TDoc::TDoc(OSType theCreator,SFReply * SFInfo){
    fCreator = theCreator;
    fNeedtoSave = false;
    fDocWindow = nil;
    fRefNum = 0;
    fFileOpen = false;
    if(SFInfo != nil){
       fNeedtoSaveAs = false;
       fFileInfo = *SFInfo;
    }
    else{
       fNeedtoSaveAs = true;
       fFileInfo.good = false;
    }
}
```

The destructor for **TDoc** is called when a **TDoc** object is deleted. Its responsibility is to dispose of the window associated with the document. The destructor of **TDoc** is declared as follows.

```
// virtual destructor so that derived destructors will be called
virtual ~TDoc(void);
```

The destructor is declared to be virtual so that the destructors for classes derived from **TDoc** will be called when the derived objects are deleted, as explained in Chapter 3. This will be very important in later chapters, which develop classes based on **TDoc**. The destructors for those derived classes will need to be called to deallocate memory holding the data for the derived documents.

Key Point ▶ It is very important to declare the destructor of a class as virtual if you expect to derive any other classes from that class.

The destructor is defined as follows.

```
TDoc::~TDoc(void){
    if(fDocWindow != nil){
        DisposeWindow(fDocWindow);
        fDocWindow = nil;
    }

}
```

▶ Initializing Documents

As previously mentioned, **TDoc** objects are typically created by an application when the user chooses a New or Open menu item. The application creates a **TDoc** object and then calls several other **TDoc** member functions to create a window for the document and perform any other document-specific initialization.

MakeWindow is the **TDoc** member function that creates a window for the document. It is not part of the constructor for two reasons. First, it involves resource and memory allocation when the window is created, so it is possible that it might fail, thus making it dangerous to do within the constructor. Second, it is split out from the constructor to make it easy to change the way derived classes make their document windows. For instance, Chapter 9 describes a derived class that overrides **MakeWindow** to make a dialog window instead of a regular window for the document. If the process of creating a document window had been part of the constructor or part of some other initialization member function, then the dialog class would have had to duplicate much of the surrounding code in order to change the way a window was created. As is often said, it isn't the code that you write that's important, it's where you put it.

MakeWindow takes one argument that specifies whether or not to make a Color QuickDraw-compatible window. The application that creates the document sets this argument based on the availability of Color QuickDraw in the runtime environment. **MakeWindow** returns true if the window is created successfully, false if otherwise. If the window can't be created, this allows the application to abort the document creation process. The code for **MakeWindow** is shown as follows.

```
Boolean TDoc::MakeWindow(Boolean colorWindow){
    if(colorWindow)
        fDocWindow = GetNewCWindow(GetWinID(),nil,(WindowPtr)-1)
    else
        fDocWindow = GetNewWindow(GetWinID(),nil,(WindowPtr)-1);

    return (fDocWindow != nil);
}
```

Notice that **MakeWindow** calls a member function, **GetWinID**, to get the resource ID for the window it is creating. This makes it easier for a derived class to use another resource as the basis of its windows. The default resource for document windows is defined in TDoc.rsrc

(see the "Resources" section of this chapter) as a 'WIND' resource with the ID number rGenericDoc (this constant is defined in TDoc.h). **GetWinID** just returns rGenericDoc by default. If your derived document class wants to use a different resource definition for its windows, you can simply override **GetWinID**, without having to override **MakeWindow**, to return a different ID number. The code for **GetWinID** is shown as follows.

```
short TDoc::GetWinID(void){
    return rGenericDoc;
}
```

The last initialization member function for **TDoc** is **InitDoc**. By default, it returns true to show that initialization is complete. It is mainly a placeholder for derived classes that might want to perform other initialization tasks that could fail. It is called after the document window has been created but before the window has been made visible. **InitDoc** is defined in the declaration of **TDoc**, as shown here.

```
// InitDoc is available for your initialization
// routines that might fail,
// It gets called after the window is created
virtual Boolean InitDoc(void){return true;}
```

See Chapters 9 and 10 for examples of how a derived class can use **InitDoc** for its own purposes.

▶ Maintaining Windows

Once the document is initialized and its window created, the program will need to move the window around and set its title. **TDoc** has several member functions that are provided to manipulate the document window. These functions are defined in the declaration of **TDoc**, shown as follows. Notice that these member functions are not virtual. They cannot be overridden in derived classes. This is done because these member functions are so basic that it should never be necessary to change them, and it is faster to call a nonvirtual member function than a virtual member function.

```
WindowPtr GetDocWindow(void)
        { return fDocWindow; }
void SetDocWindowTitle(Str255 title)
```

```
        {if(fDocWindow)SetWTitle(fDocWindow,title);}
void MoveDocWindow(short h, short v)
        {if(fDocWindow)MoveWindow(fDocWindow,h,v,true);}
void ShowDocWindow(void)
        {if(fDocWindow)ShowWindow(fDocWindow);}
```

Handling Events

TDoc contains many member functions that are called in response both to user actions with the mouse and keyboard and to events generated by the operating system. Typically, the application object (described in Chapter 6) detects the event and calls the appropriate member function for the document object. The following sections describe the event handling member functions for the document class. You will probably override many of these member functions when deriving your own document class.

Update Events

When the window for a document needs to be redrawn, the application will receive an update event for the window. The application determines which document owns the window, and then calls the **DoTheUpdate** member function for that document. **DoTheUpdate** sets the GrafPort to the window and calls the toolbox function BeginUpdate to make sure that the clipping regions are set up for drawing in the window. It then determines the bounding rectangle of the visRegion of the window. This bounding rectangle will often be smaller than the window itself, encompassing only that part of the window that needs to be redrawn. **DoTheUpdate** then passes this rectangle to the **Draw** member function, where the actual drawing of the window contents occurs. Then, **DoTheUpdate** calls the **DoDrawGrowIcon** member function to draw the grow icon. Finally, it calls the toolbox function EndUpdate to tell the system that the window has been redrawn. The code for **DoTheUpdate** is shown as follows.

```
void  TDoc::DoTheUpdate(EventRecord*/*theEvent*/){
    if(fDocWindow != nil){
        SetPort(fDocWindow);
        BeginUpdate(fDocWindow);
        Rect r = (**(fDocWindow->visRgn)).rgnBBox;
        Draw(&r);
```

```
            DoDrawGrowIcon();
            EndUpdate(fDocWindow);
        }
    }
```

The code for the **DoDrawGrowIcon** member function is quite simple. It merely calls the toolbox function DrawGrowIcon. You might wonder why this is packaged as an object member function rather than as a straight toolbox call in **DoTheUpdate**. The reason is that derived classes might want windows without grow boxes (see **TScribbleDoc** in Chapter 8 and **TModelessDoc** in Chapter 9), but they don't want to have to rewrite the entire **DoTheUpdate** member function just to change the way the grow box is drawn. With **DoDrawGrowIcon** split out as a member function, it is easy for the derived class to alter that portion of the update process without having to change **DoTheUpdate**. Even though it is less efficient to go through a member function instead of calling the toolbox directly, the gain in class flexibility is well worth it. The code for **DoDrawGrowIcon** is shown as follows.

```
// override this if you don't want grow box
virtual void DoDrawGrowIcon(void)
    {if(fDocWindow)DrawGrowIcon(fDocWindow);}
```

Key Point ▶ If you find yourself duplicating large parts of a member function when overriding that member function, you should isolate the part that you want to change and split it into a separate member function.

The other member function that is called from **DoTheUpdate** is **Draw**. It is hard to imagine a document class derived from **TDoc** that won't override **Draw**. This member function is responsible for drawing a representation of the document's data. **Draw** should not be concerned with setting the GrafPort before it draws, since this is done for it by **DoTheUpdate**. **Draw** is declared as a protected member function, so that only member functions of **TDoc** and its derived classes will call it; thus it is easier to depend on the GrafPort being set correctly before **Draw** is called.

Draw should not make any reference to the document window, since the **Draw** member function might be called to draw the data into an off-screen GrafPort or a printing GrafPort rather than a window. The rectangle argument specifies the area that needs to be drawn. It allows for optimizations when only a portion of a large data set needs to be drawn. The default definition of **Draw** is shown as follows.

```
void TDoc::Draw(Rect *r){
    EraseRect(r);

}
```

DoTheUpdate will probably not be overridden in most classes derived from **TDoc**. **DoDrawGrowIcon** will only be overridden if you don't want a grow box in your window. **Draw** will almost always be overridden.

▶ Activation

Macintosh programs typically can have more than one document window open at any given time. The window that appears closest to the front is called the *active window*. In the **TDoc/TApp** class structure, the document associated with the active window is called the *current document*. The Macintosh operating system generates activation/deactivation events for a window when it becomes the active window or when it stops being the active window. The application determines which document is associated with the activating or deactivating window and calls the **DoActivate** member function for that document.

Figure 5-1 shows two windows on the Macintosh desktop. You can see that the active window looks different than the inactive window. The active window's title bar is highlighted and its grow box and scroll bars are fully drawn. The inactive window, on the other hand, has an unhighlighted title bar, invisible scroll bars, and its grow box is empty. The operating system takes care of highlighting the title bar when a window becomes active or inactive. It is the program's responsibility to appropriately draw the scroll bars and grow box. (See Chapter 10 for a discussion of how to activate the scroll bars.) The toolbox function DrawGrowIcon checks the activation state of the window when it is called and draws the grow box appropriately. Thus, **TDoc**'s responsibility is to call **DoDrawGrowIcon** when the window changes activation states.

DoActivate handles both activation and deactivation. It calls the **DoDrawGrowIcon** member function in either case, since the grow icon will be redrawn appropriately depending on the activation state of the window.

After redrawing the grow box, **DoActivate** calls the **Activate** or **Deactivate** member function, depending on whether the window is becoming active or inactive. **Activate** is the member function that you will override to provide your document with its specific activation

response. **Deactivate** is the member function that you will override to provide your document with its specific deactivation response. For example, the **TTEDoc** class in Chapter 12 calls the toolbox function TEDeactivate from the **Deactivate** member function and TEActivate from the **Activate** member function.

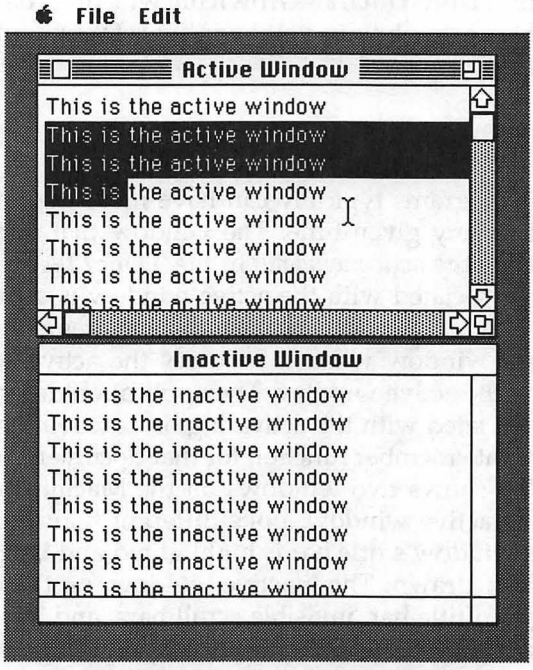


Figure 5-1. Active and Inactive Windows

The code for **DoActivate** is shown as follows.

```
void TDoc::DoActivate(EventRecord* theEvent){
    Boolean activating = theEvent->modifiers & activeFlag;

    // no need to activate if no window
    if(fDocWindow == nil)
        return;

    SetPort(fDocWindow);
    DoDrawGrowIcon();
```

```
    if(activating)
        Activate();
    else
        Deactivate();
}
```

Your derived classes will probably not need to override the **DoActivate** member function since it is applicable to almost all situations. **Activate** and **Deactivate** are both defined as empty functions. They exist to be overridden by derived document classes. The **Activate** and **Deactivate** member functions are split off to allow a convenient spot to put your document-specific code without having to change the common code. **Activate** and **Deactivate** are defined in the declaration of **TDoc** as follows.

```
// override these to de/activate window (TEActivate, etc)
virtual void Activate(void){}
virtual void Deactivate(void){}
```

▶ Growing, Zooming, Dragging

Windows associated with **TDoc** documents can be resized, zoomed, and dragged. All these actions are pretty straightforward. Derived document classes will probably not need to override any of the member functions listed in this section, although they are declared as virtual functions just in case. These member functions encapsulate standard Macintosh interface behavior for a document window. They are called from the application when it detects mouse clicks in the grow box, title bar, or zoom box of a document window.

The **DoGrow** member function has three tasks. First, it must set up a growbounds rectangle that defines the minimum and maximum size that the window can assume. Next, **DoGrow** calls the toolbox function GrowWindow to let the user drag a gray outline to size the window. Finally, it invalidates areas of the window that need to be redrawn and calls the toolbox function SizeWindow to draw the window at its new size.

The top and left coordinates of the growbounds rectangle represent the minimum vertical and horizontal dimensions of the window. The bottom and right coordinates of the rectangle represent the maximum vertical and horizontal dimensions. **DoGrow** calls the toolbox function GetGrayRgn to get a handle to the gray region of the desktop. **DoGrow** uses the bottom and right coordinates of that region's rectangle for its maximum grow size.

The only tricky thing about the code in **DoGrow** is that it invalidates the scroll bar areas of the window before and after the call SizeWindow. This ensures that the window will be properly updated and redrawn at its new size. The code for the **DoGrow** member function is shown as follows.

```
void TDoc::DoGrow(EventRecord* theEvent){
    long result;

    // use desktop gray region as grow limits
    RgnHandle theGrayRgn = GetGrayRgn();

    Rect r = (**theGrayRgn).rgnBBox;
    r.top = GetMinHeight(); r.left = GetMinWidth();

    SetPort(fDocWindow);
    result = GrowWindow(fDocWindow, theEvent->where, &r);
       if ( result != 0 ){
       // invalidate the old scroll bar areas
       r = fDocWindow->portRect;
       r.left = r.right - kScrollBarPos;
       InvalRect(&r);
       r = fDocWindow->portRect
       r.top = r.bottom - kScrollBarPos;
       InvalRect(&r);

       // now make the window the new size
       SizeWindow(fDocWindow, LoWrd(result), HiWrd(result), true);

       // invalidate the new scroll bar areas
       r = fDocWindow->portRect;
       r.left = r.right - kScrollBarPos;
       InvalRect(&r);
       r = fDocWindow->portRect;
       r.top = r.bottom - kScrollBarPos;
       InvalRect(&r);

    }
}
```

The member functions **GetMinHeight** and **GetMinWidth** are used by **DoGrow** to set up the minimum window size in the growbounds rectangle. These two member functions are defined by default to return a value of 75, but you can override them if you want a different minimum window size. They are defined in the declaration of **TDoc** as follows.

```
virtual void GetMinHeight(void){return 75;}
virtual void GetMinWidth(void){return 75;}
```

The **DoZoom** and **DoDrag** member functions implement zooming and dragging as suggested by *Inside Macintosh*. You will probably never need to override these two member functions. They are shown as follows.

```
void TDoc::DoZoom(short partCode){
    if(fDocWindow){
        SetPort(fDocWindow);
        EraseRect(&fDocWindow->portRect);
        ZoomWindow(fDocWindow, partCode,
        fDocWindow == FrontWindow());
        // invalidate the whole content
        InvalRect(&fDocWindow->portRect);
    }
}
void TDoc::DoDrag(EventRecord* theEvent){

    // use desktop gray region as drag limits
    RgnHandle theGrayRgn = GetGrayRgn();
    Rect r = (**theGrayRgn).rgnBBox;

    if(fDocWindow)
        DragWindow(fDocWindow, theEvent->where, &r);
}
```

▶ Mouse Clicks and Key Presses

By default, **TDoc** does not do anything with mouse clicks in the content area of the window. Likewise, it does nothing in response to key presses when the document window is active. Two empty member functions, **DoContent** and **DoKeyDown**, are defined to handle these events. You will probably want to override these member functions in your derived document class to respond to these events. For example, **TScribbleDoc** (in Chapter 8) overrides **DoContent** to track the mouse and let the user draw in the window. **TTEDoc** (in Chapter 12) overrides **DoKeyDown** to call the toolbox function TEClick to add the keypress to the text in the window. **DoContent** and **DoKeyDown** are defined as follows.

```
// override these to respond to clicks and keys
virtual void DoContent(EventRecord* theEvent) {}
virtual void DoKeyDown(EventRecord* theEvent) {}
```

▶ Idle Events

Whenever the application has no other events pending, it calls the **DoIdle** member function of the current document. Idle events are a good time to do background processing and housekeeping tasks. **DoIdle** is defined as an empty function by default, but you may want to override it in your derived document class. The **TTEDoc** class, described in Chapter 12, uses **DoIdle** to call the toolbox function TEIdle. Whatever you do in your **DoIdle** member function, make sure that it doesn't take more than about 1/10 second, since pending user mouse clicks and key presses won't be handled until you return from **DoIdle**.

AdjustCursor is another empty member function defined in TDoc.h. If your document needs to have a special cursor shape depending on the mouse location, you should override **AdjustCursor** and call it from **DoIdle**, passing the mouse position (in local coordinates of the document window) as the argument. **AdjustCursor** can then examine the mouse position and set the cursor accordingly. (See **TTEDoc** in Chapter 12 for an example of how to use **DoIdle** and **AdjustCursor**.) The default definitions of **DoIdle** and **AdjustCursor** are shown as follows:

```
virtual void DoIdle(void) {}
virtual void AdjustCursor(Point where) {}
```

▶ Handling Menus

The document class and the application class (described in Chapter 6) share a common header file, AppDocMenus.h, which defines the menu and item ID numbers for standard Apple, File, and Edit menus, as shown in Figure 5-2.

There are two main tasks relating to menus in a Macintosh application. The first is to adjust the menus to make sure that the user is allowed to choose only those menu items that are appropriate for the current state of the application and document. For example, the Close menu item should be disabled when there are no open documents to be closed. Properly setting up the menus reduces the need for extensive error checking in other parts of the program since it is much harder for the user to choose an inappropriate command. The second type of menu task is to actually handle commands chosen by the user from the enabled menu items.

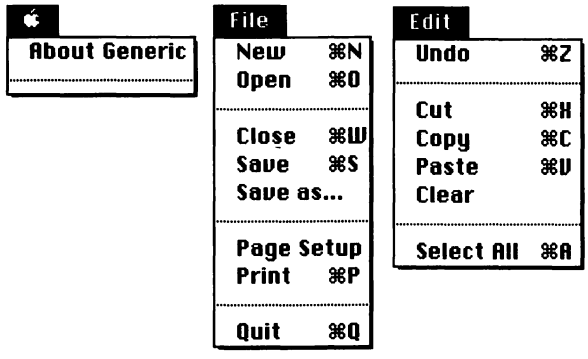


Figure 5-2. Standard menus

The application class and the document class share the responsibility for both types of menu processing. The application class is responsible for adjusting the state of some of the menu items and the document class is responsible for others. For example, the application controls whether or not the New and Open menu items are enabled since these commands trigger the creation of new documents, which is the application's job. On the other hand, the document class is responsible for enabling or disabling the Save and SaveAs menu items since these commands directly affect the current document. Likewise, the application class handles the menu commands for opening new documents, and the document class handles the Save and SaveAs commands. A discussion of the details of both types of menu processing follows.

▶ Adjusting Menus

A short inline utility function, **SetMenuAbility**, is defined to ease the task of enabling or disabling the individual menu items. It uses the toolbox functions EnableItem and DisableItem. Because this function is defined in the header file TDoc.h, it can be used by the application class as well as the document class. The definition of **SetMenuAbility** is shown as follows. Notice that this is not a member function of **TDoc** or any other class.

```
inline void SetMenuAbility(MenuHandle menu,short item,Boolean enable)
    {enable ? EnableItem(menu,item):DisableItem(menu,item);}
```

The document member function **AdjustDocMenus** is called by the application whenever the user is about to make a menu choice. This member function is responsible for enabling or disabling menu items

that depend on the current state of the document. For instance, the Save menu item is enabled only if the *fNeedtoSave* member is true. The definition of **AdjustDocMenus** is shown as follows.

```
void TDoc::AdjustDocMenus(void){
    MenuHandle menu;

// Do the document's portion of the file menu
    menu = GetMHandle(rFileMenu);
    SetMenuAbility(menu,iClose,CanClose());
    SetMenuAbility(menu,iSave,CanSave());
    SetMenuAbility(menu,iSaveAs,CanSaveAs());
    SetMenuAbility(menu,iPageSetup,CanPageSetup());
    SetMenuAbility(menu,iPrint,CanPrint());

// now the edit menu, App handles Paste Item
    menu = GetMHandle(rEdit);
    SetMenuAbility(menu,iUndo,CanUndo());
    SetMenuAbility(menu,iCut,HaveSelection());
    SetMenuAbility(menu,iCopy,HaveSelection());
    SetMenuAbility(menu,iClear,HaveSelection());
    SetMenuAbility(menu,iSelectAll,CanSelectAll());
}
```

You can see that **AdjustDocMenus** calls a separate member function for each of the menu items to determine if it should be enabled or disabled. All of these functions return a Boolean value. These menu query functions are defined in the declaration of **TDoc**, as follows.

```
virtual Boolean CanUndo(void) { return false; }
virtual Boolean HaveSelection(void) { return false; }
virtual Boolean CanPaste(OSType /*theType*/) { return false; }
virtual Boolean CanSelectAll(void){return false;}
virtual Boolean CanClose(void) { return true; }
virtual Boolean CanSave(void) { return fNeedtoSave; }
virtual Boolean CanSaveAs(void) { return false; }
virtual Boolean CanPageSetup(void){return false;}
virtual Boolean CanPrint(void) { return false; }
```

These member functions provide a convenient way for you to change the default enabling/disabling behavior without overriding **AdjustDocMenus**. For example, if your document can handle the Print menu command, then you will override **CanPrint** to return true instead of false. The document classes in Chapters 7-13 show examples of how to override these functions.

The default definition of **AdjustDocMenus** only covers the standard File and Edit menus. If your document class adds other menus you will need to override **AdjustDocMenus** to account for the additional menu items. When overriding **AdjustDocMenus** it is important to call the parent class's version of **AdjustDocMenus** as part of the derived version so that you will inherit the default behavior. For example, the following code skeleton shows how to override **AdjustDocMenus**.

```
void TDerivedDoc::AdjustDocMenus(void){
    // adjust the additional menu items for the derived document
    // ... //
    // and now call the parent class for default behavior
    TDoc::AdjustDocMenus();
}
```

Key Point ▶

The derived **AdjustDocMenus** member function calls the **AdjustDocMenus** member function for its parent class, **TDoc**, by preceding the member function name with the class name and two colons, as shown here:

TDoc::AdjustDocMenus();

The Scribble program in Chapter 8 contains an example of how to override **AdjustDocMenus** to enable and disable menus beyond the standard Edit and File menus.

▶ Handling Menu Commands

The document member function **DoDocMenuCommand** is called by the application whenever the user makes a menu choice, either with the mouse or a command-key equivalent. **DoDocMenuCommand** gets first shot at the menu command to see if it is one of the commands that can be handled by the document without any help from the application. For example, the document is completely responsible for responding to the Save command.

If the document handles the menu command, then **DoDocMenuCommand** returns true so that the application won't try to do any further processing for that command. For each menu item that it handles, **DoDocMenuCommand** calls other document member functions to do the actual processing. Those member functions are described individually in other sections of this chapter.

If **DoDocMenuCommand** returns false, then the application assumes that the menu command is one that it is responsible for and processes the command with its own methods. **DoDocMenuCommand** is defined as follows.

```
Boolean  TDoc::DoDocMenuCommand(short menuID, short menuItem){
   switch ( menuID ){
      case rFileMenu:
         switch ( menuItem ){
            case iSave:
               DoSave();
                  break;
               case iSaveAs:
                  DoSaveAs();
                  break;
               case iPageSetup:
                  DoPageSetup();
                  break;
               case iPrint:
                  DoPrint();
                  break;
               default:  // we didn't handle command
               return false;
            } // end menuItem switch
            return true; // we handled this command

      case rEdit:
         if ( !SystemEdit(menuItem-1) ){
            switch ( menuItem ){
               case iClear:
                  DoClear();
                  break;
               case iSelectAll:
                  DoSelectAll();
                  break;
               default:
                   // we didn't handle command
                   return false;
            } // end menuItem switch
            return true; // we handled this command
         } else
            return true; // SystemEdit handled command

   }// end menuID switch
   // we didn't handle command
   return false;
}
```

Like **AdjustDocMenus**, **DoDocMenuCommand** only knows about the standard File and Edit menus. If your document class adds other menus, you will have to override **DoDocMenuCommand** to process those additional menu items. Like **AdjustDocMenus**, it is important to call the parent class's version of **DoDocMenuCommand** from your derived version so that the default behavior will occur along with any new capabilities. See the Scribble program in Chapter 8 for an example of how to override **DoDocMenuCommand** to process additional menu items.

▶ Cut, Copy, and Paste

The Macintosh clipboard provides a metaphor and mechanism for transferring data from one place to another within an application and between applications. The **TDoc** document class implements clipboard handling by defining member functions for cut, copy, and paste. Clipboard data always has a four-character type designator associated with it. The two most common data types are 'TEXT' and 'PICT'.

Clipboard handling is a cooperative effort between the document class and the application class (see Chapter 6). When the user selects the Copy menu item, the application calls the **DoCopy** member function of the current document, passing it a pointer to a handle to hold the clipboard data and a pointer to a variable to specify the type of data. **DoCopy** allocates memory for the clipped data and passes its handle and data type back to the caller through the arguments. **DoCut** is similar, except that the document also deletes the clipped data after copying it. These two member functions are stubbed out in **TDoc** since they depend on the type of data in the document. You will need to override these member functions if your document supports cutting and pasting to the clipboard. (The TextEdit document class in Chapter 12 shows an example of how to override these member functions.) **DoCut** and **DoCopy** are defined in **TDoc** as follows.

```
// these are all empty, override if you want them
virtual Boolean DoCut(Handle *theData,OSType *theType)
    {*theData = nil;*theType = '????';return false;}
virtual Boolean DoCopy(Handle *theData,OSType *theType)
    {*theData = nil;*theType = '????';return false;}
```

After the application has called **DoCut** or **DoCopy** for a document, it then keeps the resulting clipboard data handle. Later, if the user chooses the Paste menu command, the application will send the data handle and type designator to the document by calling the **DoPaste**

member function for the document. The document takes the pasted data and incorporates it into the document. Like **DoCut** and **DoCopy**, this member function is just a stub in **TDoc**. You will want to override **DoPaste** if your document supports the clipboard. Again, see Chapter 12 for a document class that overrides the **DoPaste** member function. **DoPaste** is defined as follows.

```
virtual void DoPaste(Handle theData,OSType theType) {}
```

The document class deals with the clipboard operations in terms of only handles and data types. It does not interact with the toolbox functions that operate on the Macintosh's system clipboard. The application class is responsible for passing clipboard data on to the system as necessary, as explained in Chapter 6.

There are also three other member functions associated with items in the Edit menu that do not affect the clipboard contents. They are defined as empty functions that you will override if you want this functionality in your document. They are defined as follows.

```
virtual void DoClear(void) {}
virtual void DoSelectAll(void) {}
virtual void DoUndo(void) {}
```

▶ Handling Files

Much of the previous discussion of the **TDoc** class centered on its window and menu handling member functions. The window is merely a way of representing, or visualizing, the document's data to the user. The other major responsibility of the document class is to maintain its data in a disk file. This section discusses the member functions that **TDoc** uses to keep track of the file that is associated with the document.

▶ Opening and Closing Document Files

Two low-level member functions, **OpenDocFile** and **CloseDocFile**, are defined to open and close the document's file. When a document is created from an existing file, **TDoc** keeps open the file associated with the document until the user closes the document. This is a precaution to ensure that other applications in the MultiFinder or AppleShare environment can't modify the file while **TDoc** has the file open. If the document is created as a new, blank document, then it will have no file

associated with it until the user does a SaveAs operation, after which the newly created file is kept open until the document is closed.

OpenDocFile's argument is an SFReply which contains the file name and volume reference number for the file. The return value from **OpenDocFile** is the reference number for the newly opened file, or 0 if the file could not be opened. **OpenDocFile** tries to open the file, based on the information in the SFReply. If the file is already open, **OpenDocFile** calls the utility function **ErrorAlert**, described in a later section of this chapter, to display an alert dialog and then returns 0. If the file cannot be found, which can happen during a SaveAs operation, the **OpenDocFile** tries to create the file and then open the newly created file. If the file is successfully opened, **OpenDocFile** sets the *fFileOpen* and *fRefNum* members of the document and returns the refnum for the file. The code for **OpenDocFile** is shown as follows.

```
short TDoc::OpenDocFile(SFReply * reply){

   short refnum;
   OSErr err = FSOpen((Str255)reply->fName,
        reply->vRefNum,&refnum);
   switch(err){
      case fnfErr: // file not found, create it
         err = Create((Str255)reply->fName,
            reply->vRefNum,
            fCreator,
            GetDocType());
         if(err == noErr){
             err = FSOpen((Str255)reply->fName,
                    reply->vRefNum,
                    &refnum);

             if(err != noErr)
                return 0;
         } else
             return 0;

         // if open was successful, fall through
         // to next case

      case noErr:   // file opened OK
         fFileOpen = true;
         fRefNum = refnum;
         return refnum;

      case opWrErr:
         ErrorAlert(rDocErrorStrings,sFileOpen);
         return 0;
```

```
        default:
            ErrorAlert (rDocErrorStrings,sUnknownErr);
            return 0;

    }
}
```

CloseDocFile is very simple; it just uses the refnum supplied as its argument to close the file, as follows.

```
void TDoc::CloseDocFile(short refNum){
    OSErr err = SClose(refNum);
}
```

▶ Reading and Writing Document Files

Two other low-level member functions, **DoWriteFile** and **DoReadFile**, are defined as empty member functions. You must override these member functions if your derived document class wants to move data to or from a file. Both of these member functions take a refnum argument and return a Boolean to indicate if the operation was successful. Chapters 8, 11, and 12 contain examples of how to override **DoWriteFile** and **DoReadFile**. They are defined in **TDoc** as follows.

```
// override these to read and write files
virtual Boolean ReadDocFile(short /*refnum*/){return true;}
virtual Boolean WriteDocFile(short /*refnum*/){return true;}
```

▶ Saving Document Files

On a higher level, several member functions are defined to be called in response to user menu choices. These member functions include **DoSave**, **DoSaveAs**, and **DoClose**.

DoSaveAs is called when the user chooses the SaveAs menu item, or when the user tries to close a document that has unsaved changes and was created as a blank document. **DoSaveAs** calls the toolbox function SFPutFile to put up a dialog to allow the user to specify a file name and directory for the file. If the user cancels from within the SFPutFile dialog, then **DoSaveAs** returns false and does not do any further processing. If the user does not cancel, then **DoSaveAs** tries to open the specified file by calling **OpenDocFile**, which will create the file if it doesn't already exist. Once the file is open, **DoSaveAs** calls **WriteDocFile** to write the document data to the file and rename the document window

to match the new file name. The code for **DoSaveAs** is shown as follows. Notice that when the data has been successfully written to the file, the *fNeedtoSave* and *fNeedtoSaveAs* members are set to false.

```
Boolean TDoc::DcSaveAs(void){
    SFReply whereToSave;
    Point p;
    Str255 title;

    GetWTitle(fDocWindow,title);

    p.h = 100; p.v = 100;
    SFPutFile(p,
        "\pSave file as...",
        title,
        (DlgHookProcPtr)nil,
        &whereToSave);

    if(! whereToSave.good){
        // the user canceled the SaveAs
        return false;
    }else{
        fFileInfo = whereToSave;
        fRefNum = OpenDocFile(&whereToSave);
        if(fRefNum == 0){
            // file didn't open
            return false;
        }else{
            fFileOpen = true;
            if(! WriteDocFile(fRefNum)){
                // write was unsuccessful
                return false;
            }else{
                fNeedtoSave = false;
                fNeedtoSaveAs = false;
                SetDocWindowTitle(whereToSave.fName);
            }
        }
    }
    return true;    // passed every test for success
}
```

The **DoSave** member function is called when the user chooses the Save menu item or when the user tries to close a document that has unsaved changes. It will call **DoSaveAs** if the *fNeedtoSaveAs* member is true, indicating that the file was created as a blank document and has never been saved. If **DoSaveAs** is not called, then **DoSave** calls **Write-**

DocFile to put the document's data in the file and then sets the *fNeedtoSave* and *fNeedtoSaveAs* members to false to show that the file has been saved. **DoSave** is defined as follows.

```
Boolean TDoc::DoSave(void){
    if(fNeedtoSaveAs)
        return DoSaveAs();

    if(WriteDocFile(fRefNum)){
        fNeedtoSave = false;
        fNeedtoSaveAs = false;
        return true;
    } else
        return false;
}
```

▶ Closing Documents

The application calls **DoClose** when the user chooses the Close menu item or clicks on the close box of a document window. It checks the *fNeedtoSave* member of the document to see if the document has any unsaved changes, and it calls **WantToSave** to display a dialog to give the user a chance to save those changes before actually closing the window, as shown in Figure 5-3.

The code for the **WantToSave** member function is shown as follows.

```
short TDoc::WantToSave(void){
    Str255 title;
    Str255 nullStr;
    *nullStr = 0;

    if(fDocWindow){
        GetWTitle(fDocWindow,title);
        ParamText(title,nullStr,nullStr,nullStr);
    } else
        ParamText(nullStr,nullStr,nullStr,nullStr);
    return Alert(rWantToSave,(ModalFilterProcPtr)nil);
}
```

The **WantToSave** dialog gives the user three choices. The user can say "yes, I want to save," or "no, don't save, but continue," or "cancel the whole operation." If the user clicks the Cancel button, then the **DoClose** operation is canceled. If the user clicks the No button, then **DoClose** proceeds without trying to save the document changes. If the user clicks the Yes button, then **DoClose** tries to save the document before proceeding with closing the document.

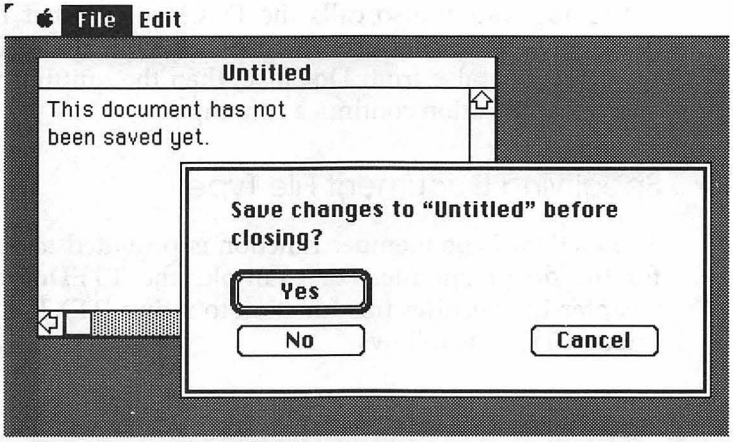


Figure 5-3. The WantToSave Dialog

If the user chooses to save before closing, the **DoSave** member function is called. The user can cancel the operation while saving, so the return code from **DoSave** is checked before proceeding. If all goes well, **DoClose** closes the document file and returns true. The definition of **DoClose** is shown as follows.

```
Boolean TDoc::DoClose(void){
    // give the user a chance to save if necessary
    // and possibly cancel the close operation

    if(fNeedtoSave){
        // ask if they want to save it
        short saveit = WantToSave();
        if(saveit == iCancel)
            return false;
        if(saveit == iYes){
            // User can cancel save at this point too
            if (! DoSave())
                return false;
        }
    }

    //close the file
    if(fFileOpen )
        CloseDocFile(fRefNum);

    // if all goes well, return true
    return true;
}
```

The application also calls the **DoClose** member function for all its documents when the user chooses the Quit menu item. If any document returns false from **DoClose**, then the quitting process is aborted and the application continues running.

▶ Specifying Document File Type

The **GetDocType** member function is provided to specify the file type for the document file. For example, the **TTEDoc** class described in Chapter 12 overrides this function to return 'TEXT'. **GetDocType** is defined in **TDoc** as follows.

```
virtual OSType GetDocType(void){return '????';}
```

You will probably never need to override **DoSave**, **DoSaveAs**, or **DoClose**. They encapsulate the proper logic to deal with unsaved changes in a general way to implement behavior that Macintosh users have come to expect. Because they are part of the **TDoc** base class, this behavior will show up in your derived classes as well. Typically, **DoReadFile**, **DoWriteFile**, and **GetDocType** are the only file-related member functions that you will have to override when deriving your own document class. All the other operations for file maintenance — opening and closing a file, asking the user to save before closing a document with unsaved changes, and getting a new file name during a SaveAs operation — are taken care of by the default member functions of **TDoc**. Your only responsibility is to read and write the data in the file to tell the application the file type for the file.

▶ Printing

TDoc provides two empty member functions, **DoPageSetup** and **DoPrint**, that you can override to provide printing support. These two member functions correspond to the Page Setup and Print menu items, respectively.

When you override **DoPageSetup**, you will call the toolbox Print Manager functions to fill in the values for a print record. You can also define an additional document member to store the print record information.

When overriding **DoPrint**, you can open a printing port for the current printer and then call the **Draw** member function to draw the image of the document's data. Notice that the discussion of the **Draw** member function said that **Draw** should not assume that it is drawing in a window. If you follow that restriction, you should be able to use the **Draw** member function to do your printing as well as your window drawing.

The empty definitions for **DoPageSetup** and **DoPrint** follow. See the PictView program in Chapter 11 for an example of how to implement printing by overriding these member functions.

```
// override these for printing support
virtual void DoPageSetup(void){}
virtual void DoPrint(void) {}
```

▶ Utilities

Three utility functions are defined in TDoc.h and TDoc.cp. These utility functions are not defined as member functions of **TDoc** since they do not need to access any members of the **TDoc** class. They are included in the same files as the **TDoc** class merely for convenience.

HiWrd and **LoWrd** take a 4-byte integer as input and return the upper or lower 2-byte word. They are defined as inline functions for efficiency, as shown here.

```
inline short HiWrd(long aLong)
    {return (short)(((aLong) >> 16) & 0xFFFF);}
inline short LoWrd(long aLong)
    {return (short)((aLong) & 0xFFFF);}
```

In a traditional C program, these two functions would probably be defined as macros. The big disadvantage of macros, however, is that there is no type-checking for arguments or return values for macros.

The other utility function is **ErrorAlert**. It is used by several member functions in **TDoc** to put up an alert to tell the user of an error condition, such as when the user tries to open a file that is already open. **ErrorAlert** pulls in the string resources specified by its arguments and substitutes them into the generic error alert dialog with the toolbox function ParamText. It then displays the alert and waits for the user to click the OK button before going on. The first argument specifies the ID number of the STR# resource and the second argument indicates which string to pull from the STR# resource. This allows derived classes to add their own STR# resources to handle error conditions that do not occur in the base class. Look at the **DoOpenFile** member function in the "Opening and Closing Document Files" section of this chapter for an example of how **ErrorAlert** can be used.

```
void ErrorAlert(short stringsID,short theError){
    short result;
    Str255 theStr;
    Str255 nullStr;
    *nullStr = 0;

    GetIndString(theStr,stringsID,theError);

    ParamText(theStr,nullStr,nullStr,nullStr);
    result = CautionAlert(rErrorAlert,(ModalFilterProcPtr)nil);
}
```

▶ Compiling TDoc

TDoc cannot run as a program by itself. It is designed to work with the **TApp** application class described in the next chapter. You do, however, have to compile TDoc.cp to create an object file, TDoc.cp.o, which can be linked with other files to create a full application.

One crucial element of the compilation process is to specify a segment name for the **TDoc** code. This is done by placing the following compiler directive just before the first line of code.

```
#pragma segment DocSeg
```

This line directs the compiler to put the code that follows into a segment named DocSeg. It is a good idea to break your program into smaller segments to make it more memory efficient. When you link your final program, all the various code segments will be included. You do not need to be concerned about whether a function call is to a function in the same segment or a different segment since the linker takes care of resolving those address calculations.

To compile TDoc.cp, invoke the following command in MPW, making sure, of course, that the current directory contains the file TDoc.cp.

```
CPlus TDoc.cp - o TDoc.cp.o
```

That command above causes the CPlus script to run, which calls CFront to create C code from your C++ code; it then calls the C compiler to compile the final object code. The code is output as TDoc.cp.o, as specified by the -o flag in the command.

▶ Resources

The source code for the **TDoc** class is contained in TDoc.cp and TDoc.h. It is compiled to produce object code, which resides in the file TDoc.cp.o. The object code can be linked with other application code to form the final program. But, a Macintosh program also requires resources to be complete. The code in **TDoc** depends on several resources to function properly. The resources are in the file TDoc.rsrc. Appendix B contains a rez-compatible listing of these resources in the file TDoc.rsrc.r.

The resources in TDoc.rsrc are shown in Table 5-1.

Table 5-1. TDoc Resources

Res. Type	*ID No.*	*Description*
'WIND'	1000	The default window resource for the document window
'ALRT'	255	The generic error alert
'DITL'	255	The dialog item list for the generic alert
'ALRT'	500	An alert that asks if the user wants to save a document
'DITL'	500	The dialog item list for the want-to-save alert
'STR#'	255	A list of strings that can be substituted in the generic alert

You will need to replace some of these resources as you create classes derived from **TDoc**. Chapters 7-13 contain examples of how to replace resources for the **TDoc** class.

▶ Summary

The **TDoc** class provides a core for creating documents in Macintosh applications. The document class is responsible for maintaining a window to display the data and for reading and writing the data to and from the disk.

TDoc is really not very useful in itself. It is designed to be a base class. You use **TDoc** as the basis for derived document classes that fit the kind of data you want to manipulate. Most of its member functions are defined as virtual member functions so that the derived classes can override and extend the class's functionality.

The next chapter describes the **TApp** application class. The application class is closely tied to the document class. Together, they form a good framework on which you can quickly build Macintosh applications.

6 ▶ TApp: The Generic Application Class

TApp is a class that encapsulates the standard behavior of a Macintosh application. It knows how to initialize the toolbox managers and display menus. It supports desk accessories and is compatible with MultiFinder. It can respond to standard File and Edit menu commands by opening and closing documents and performing clipboard operations.

TApp, when used with the **TDoc** class described in Chapter 5, can be the basis for your own Macintosh applications. You can create a derived application class from **TApp**, changing only those member functions of **TApp** that are necessary to match the functionality of your application. You will find that you probably do not have to change **TApp** as much as **TDoc** to get a working application. Chapters 7-13 show examples of how to modify **TApp** to create your own applications.

TApp is very dependent on **TDoc**. It makes calls to **TDoc** member functions to respond to most user events. **TApp** acts mainly as a manager for one or more documents, routing events to the proper document so that they can be processed.

This chapter describes the **TApp** class by discussing its various members and member functions. **TApp** is a very large class, so you might not want to try to read this entire chapter in one sitting. You can also look at the complete code listing for TApp.h and TApp.cp in Appendix B.

▶ Extending TList to Handle Documents

One of the most fundamental tasks of the application class is to manage one or more document objects. To do this, the application class needs to keep a list of all open documents. Chapter 3 developed the **TList** class to manage lists of objects. You can derive a document list class, **TDocList**, from the **TList** class to keep a list of **TDoc** objects. The declaration of **TDocList** is shown here.

```
class TDocList : public TList {

public:

    // add one new member function
    // find the TDocument associated with the window
    TDoc* FindDoc(WindowPtr window);

};
```

TDocList uses all the existing **TList** members and member functions to add and delete objects from the list. It defines one additional member function to make it more useful to the application class. The application's job is to route event messages from the operating system to a document object where the event can be processed. Unfortunately, event reporting in the Macintosh operating system is always tied to the window that should be concerned with the event. The operating system knows nothing about document objects. It is up to the application to determine which document object is associated with the window specified in the event message. To do this, **TDocList** adds a member function, **FindDoc**, that takes a WindowPtr as an argument and searches the document list for the document that is associated with that window. As shown in the following code, **FindDoc** repeatedly calls the **GetNext** member function of **TList** to get a pointer to the next item in the list. It then compares the *fDocWindow* member of the item to see if it is equal to the window argument. It returns a pointer to the item that matches, or nil if no match is found.

```
TDoc* TDocList::FindDoc(WindowPtr window){
    TLink* temp;
    TDoc* tDoc;

    for (temp = fLink; temp != nil; temp = temp->GetNext()){
        tDoc = (TDoc*)temp->GetItem();
```

```
            if (tDoc->GetDocWindow() == window)
                return tDoc;
        }
        return nil;
}
```

Key Point ▶

The **TList** class is implemented with void pointers (void *) so that its internals don't need to be changed to accommodate different object types as list items. This is because the member functions of **TList** never try to access members or member functions of the objects in the list. The **FindDoc** member function of **TDocList**, however, needs to typecast the void pointer returned from **GetNext** to be a pointer to a **TDoc** object (**TDoc ***) so that it can access the *fDocWindow* member of the document object item.

▶ How to Use TApp

TApp is a large class. It contains over fifty member functions, yet fewer than ten of those member functions are public. Most of the complexity of **TApp** is hidden in protected member functions that are called only by other **TApp** member functions. When you use **TApp** to create Macintosh applications you will actually call only a few of **TApp**'s member functions. For example, the following code fragment shows how you could use **TApp** to make a very simple program.

```
void main(void){
    TApp theApp;

    if(theApp.InitApp()){
       theApp.OpenNewDoc();
       theApp.EventLoop();
    }
    theApp.CleanUp();
}
```

The code starts by defining a **TApp** variable. This allocates space for the object and causes its constructor to run. Next, it calls the **InitApp** member function to initialize the application object. This member function returns a Boolean value indicating if the initialization was successful. If the application object is successfully initialized, you then call the

OpenNewDoc member function to create a new, blank document. Next, call the **EventLoop** member function. This member function will continue to handle user input and operating system events until the Quit menu item is chosen. Finally, the **CleanUp** member function is called to allow the application a chance to perform any last chores before the program terminates. Notice also that the destructor for the application object will run when the **TApp** variable goes out of scope when main terminates.

This example created a **TApp** object. Normally, you will first derive your own application class from **TApp** and then define an object of your derived application class rather than **TApp**. When deriving your application class, you will probably also derive a document class from **TDoc**. In your derived application class, you will override the **MakeDoc** member function to create documents of your derived document class rather than the default **TDoc** class. Later sections of this chapter will discuss in more detail the member functions that you will probably override when deriving your own application class from **TApp**, and Chapters 7-13 contain examples of derived application classes.

▶ TApp Members

TApp needs to maintain several members to indicate the state of the application and the environment. The declaration for **TApp**'s members is listed as follows.

```
class TApp {

public:
    //  other classes might like to see this
    SysEnvRec  fenvRec;

protected:
    // members just for TApp and derived classes
    TDocList* fDocList;
    TDoc*     fCurDoc;
    Boolean   fHaveWaitNextEvent;
    Boolean   fDone;
    Boolean   fInBackground;
    Handle    fClipData;
    OSType    fClipType;
    Boolean   fDAonTop;
    short     fLastScrapCount;
```

The first member, *fenvRec*, is filled in when **TApp** calls the toolbox function SysEnvirons. The resulting structure contains information about the current hardware and software environment, including such information as whether or not the current machine has Color QuickDraw. This member is in the public section of the class declaration because functions outside the **TApp** class might want to access this structure to determine characteristics of the environment.

The rest of the members are protected, so that only member functions of **TApp** and its derived classes can access them. The purpose of each member will be discussed in the following sections that describe the various **TApp** member functions.

▶ Clipboard Support

TApp supports the standard Macintosh clipboard model by maintaining several members. *fClipData* is a handle to a block of data holding the data for the clipboard. *fClipType* is an OSType variable that contains the current type designator for the clip data. Standard clipboard data types include 'TEXT' and 'PICT'. Other members that are used for clipboard support are *fDAonTop* and *fLastScrapCount*, which are explained later.

The clipboard model implemented by **TApp** is rather limited in that it only allows an application to support a single clipboard data type. The **CanAcceptClipType** member function returns an OSType value that indicates what kind of clipboard data the application supports. The default definition of **CanAcceptClipType** is shown as follows.

```
OSType TApp::CanAcceptClipType(void){

    return '????';

}
```

Since the default **CanAcceptClipType** member function returns a useless data specifier, you will need to override this member function if your application supports the clipboard. The TextEdit application described in Chapter 12 overrides this member function to return 'TEXT'.

▶ Private Clipboard and System Clipboard

The application keeps a copy of its clipboard data separate from the system clipboard. It uses its private copy when handling cut, copy, and paste commands for its documents. This avoids the overhead of accessing the system clipboard for every cut, copy, or paste operation. The application exchanges its private clipboard data with the system clipboard whenever it gives up or regains control from a desk accessory or another application. This exchange is handled differently depending on whether or not the application is running under MultiFinder. Because there is no way for an application to actually tell whether or not it is running under MultiFinder, **TApp** uses both clipboard strategies all the time. But they do not conflict since the conditions that trigger their activation do not overlap the two environments. That is, the MultiFinder-specific code will never have any effect in a non-MultiFinder runtime environment, and vice versa.

Two member functions are defined to move data between the system clipboard and the application's private clipboard. These member functions are used whether or not MultiFinder is running. The first, **GetClipFromSystem**, calls the **CanAcceptClipType** member function to determine what kind of data the application can use and then calls the toolbox function GetScrap to get the data from the system clipboard. The data returned from GetScrap is used to set the value of the *fClipData* member of the application object, shown as follows.

```
void TApp::GetClipFromSystem(void){

    long offset;
    Handle newData = NewHandle(0);
    OSType newType = CanAcceptClipType();

    long result = GetScrap(newData,newType,&offset);
    if(result > 0){
        if(fClipData != nil)
            DisposHandle(fClipData);
        fClipData = newData;
        fClipType = newType;
    }
}
```

The **GiveClipToSystem** member function takes the application's private clipboard and writes it to the system clipboard with the toolbox function PutScrap. This member function is called whenever the appli-

cation is about to give up control to a desk accessory or to another application under MultiFinder, or because the application is terminating. The other task of **GiveClipToSystem** is to update the *fLastScrapCount* member. The scrapcount is a value maintained by the system clipboard that changes every time the data on the system clipboard changes. **GiveClipToSystem** calls the toolbox function InfoScrap to get the current value of scrapcount (after the data has been written to the clipboard). Later, **TApp** can check the current system scrapcount against *fLastScrapCount* to see if the clipboard contents have changed. The code for **GiveClipToSystem** is shown as follows.

```
void TApp::GiveClipToSystem(void){
    if(fClipData != nil){
        long result = ZeroScrap();
        if(result != noErr)
            return;

        long size = GetHandleSize(fClipData);
        HLock(fClipData);
        result = PutScrap(size,fClipType,*fClipData);
        HUnlock(fClipData);
    }
    // update our scrapcount field so we can tell if scrap
    // has changed later on
    PScrapStuff scrapInfo = InfoScrap();
    fLastScrapCount = scrapInfo->scrapCount;
}
```

▶ The Clipboard without MultiFinder

When the application is not running under MultiFinder, there are four distinct situations where the private and system clipboards must be reconciled, as listed here.

- When the application starts up, it should copy the system clipboard to its private clipboard. This makes the last cut or copy operation in the previous application available to the new application. **TApp** calls **GetClipFromSystem** as part of its initialization.
- When the application terminates, it should copy its private clipboard to the system clipboard. This makes the application's last cut or copy operation available to the next application that runs. **TApp** calls **GiveClipToSystem** as part of its termination sequence.

- When a desk accessory (DA) becomes active, the application should copy its private clipboard to the system clipboard so that the DA can access the last copy or cut operation from the application. **TApp** detects this situation by monitoring and updating the class member *fDAonTop*.
- When an application window becomes active after a desk accessory has been active and the clipboard changed while the DA was active. This allows the last cut or copy operation in the DA to be available to the application. **TApp** detects this situation by monitoring and updating the class members *fDAonTop* and *fLastScrapCount*. It copies the system clipboard to the private clipboard only if the current scrapcount is different than *fLastScrapCount*.

Two key member functions that implement the clipboard scheme for the non-MultiFinder environment are **ClipHasChanged** and **CheckForDASwitch**. **ClipHasChanged** calls the toolbox function InfoScrap to get the current scrapcount value for the system clipboard. It compares that value to the *fLastScrapCount* member. If they are the same, then **ClipHasChanged** returns false, to indicate that the clipboard has not changed. If the two values are different, then **ClipHasChanged** returns true, as follows.

```
Boolean TApp::ClipHasChanged(void){
   PScrapStuff scrapInfo = InfoScrap();
   return (scrapInfo->scrapCount != fLastScrapCount);
}
```

The **CheckForDASwitch** member function is called every time **TApp** fetches a new event in the event loop (the event loop is discussed in a later section of this chapter). It checks to see if the front window belongs to a desk accessory by looking at the windowKind element of the WindowRecord returned by the toolbox function FrontWindow. **TApp** keeps a status member, *fDAonTop*, to indicate whether or not the front window belongs to a desk accessory. This allows it to detect when a desk accessory first becomes active or when one of the application's own windows becomes active after a desk accessory has been on top. When it detects a desk accessory window first becoming active, it calls **GiveClipToSystem**. When it detects one of its own windows becoming active after a desk accessory, it calls **GetClipFromSystem** if the clipboard has changed since it was last written out to the system. The code for **CheckForDASwitch** is shown as follows.

```
void TApp::CheckForDASwitch(WindowPtr theFrontWindow){

   if(theFrontWindow == nil)
      return;

   Boolean DAWindowOnTop;
   DAWindowOnTop = ((WindowPeek)theFrontWindow)->windowKind < 0;

   // if the state has changed since we last checked it, then
   // do clipboard conversion
   if(DAWindowOnTop != fDAonTop){
      fDAonTop = DAWindowOnTop;

      if(DAWindowOnTop)
         // DA is becoming active, give up the clipboard
         GiveClipToSystem();
      else {
         // DA is becoming inactive, reclaim clip if necessary
         if(ClipHasChanged())
            GetClipFromSystem();
      }
   }
}
```

▶ The Clipboard with MultiFinder

When the application is running under MultiFinder, there are also four distinct situations where the private and system clipboard must be reconciled, but two of these situations are defined differently than for the non-MultiFinder environment.

- When the application starts up, it should copy the system clipboard to its private clipboard. This makes the last cut or copy operation in the previous application available to the new application. This is handled just as it is in the non-MultiFinder environment.
- When the application terminates, it should copy its private clipboard to the system clipboard. This makes the application's last cut or copy operation available to the next application that runs. This is handled just as it is in the non-MultiFinder environment.
- When the application is giving up control to another application, the first application should copy its private clipboard to the system clipboard so that the other application can access the last copy or cut operation from the first application. MultiFinder signals this situation

by sending a Suspend event to the first application. A flag in the event message indicates whether or not clipboard conversion is necessary. **TApp** responds to a Suspend event by calling **GiveClipToSystem** if the flag indicates that clipboard conversion is necessary.

- When the application regains control after another application has been active and the clipboard changed while the other application was active. This allows the last cut or copy operation in the other application to be available to the application that is regaining control. MultiFinder signals this situation by sending a Resume event to the application that is regaining control. A flag in the event message indicates whether or not clipboard conversion is necessary. **TApp** responds to a Resume event by calling **GetClipFromSystem** if the flag indicates that clipboard conversion is necessary.

Other details of how **TApp** supports MultiFinder and Suspend and Resume events are discussed in the "MultiFinder Support" section of this chapter.

▶ TApp Constructor and Destructor

The constructor for a **TApp** object is called whenever you define a **TApp** variable or when you create a **TApp** object with the C++ **new** operator. The code fragments that follow show two ways to create **TApp** objects.

```
TApp  theApp;                // define a TApp variable
TApp * ptheApp = new TApp;   // use pointer to TApp
```

In the first case, simply defining a **TApp** variable causes the compiler to allocate space for the object and run its constructor. In the second case, we are defining a variable that is a *pointer* to a **TApp** object. When you define pointer variables, the compiler allocates space for the pointer variable, but you must explicitly ask it to allocate space for the actual object by using the **new** operator. The **new** operator will allocate space for the object and run its constructor.

Key Point ▶ **Because each program can have only one application object, you will usually define it as a simple variable rather than as a pointer. In contrast, document objects are almost always defined as pointer variables and allocated with the new operator.**

The **TApp** constructor has three main tasks. First, it must initialize all the object members and create an empty document list object. Next, it must initialize the Macintosh toolbox managers and investigate the hardware and software environments. Finally, it loads the menus for the application and displays the menu bar. In the earlier discussion of constructors, it was said that you should never do anything that can fail in a constructor. In this case we stretch that rule a bit by loading menu resources in the constructor. But unless you make a mistake when building the program or the available memory is extremely low at program startup, there is no reason why the menu building code should fail, and it is probably safe to do it in the constructor. The code for **TApp**'s constructor is shown as follows.

```
TApp::TApp(void){

   // initialize our class variables
   fCurDoc = nil;
   fDone = false;
   fInBackground = false;
   fClipData = nil;
   fClipType = '????';
   fDAonTop = false;
   fLastScrapCount = 0;

   // initialize Mac Toolbox components
   InitGraf((Ptr) &qd.thePort);
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs((ResumeProcPtr) nil);
   InitCursor();

   (void) SysEnvirons(curSysEnvVers, &fenvRec);

   // expand the heap so new code segments load at the top
   MaxApplZone();

   // allocate an empty document list
   fDocList = new TDocList;

   // check to see if WaitNextEvent is implemented
   fHaveWaitNextEvent = TrapAvailable(_WaitNextEvent, ToolTrap);
```

```
    // read menus into menu bar
    Handle menuBar = GetNewMBar(rMenuBarID);
    // install menus
    SetMenuBar(menuBar);
    DisposHandle(menuBar);

    // add DA names to Apple menu
    AddResMenu(GetMHandle(rAppleMenu), 'DRVR');

    DrawMenuBar();
}
```

Notice that the constructor initializes the menus by loading an 'MBAR' resource with the resource ID number rMenuBar. This constant is defined in TApp.h. If you want to change or add menus to your application, you must still have an 'MBAR' resource with this ID in your application's resource fork. (See the Scribble program in Chapter 8 for an example of how to add menus to an application based on **TApp**.)

A short utility member function, **TrapAvailable**, is defined to tell whether or not a particular toolbox function is available. The constructor calls **TrapAvailable** to set the Boolean member *fHaveWaitNextEvent* to indicate whether or not the toolbox function WaitNextEvent, which is associated with MultiFinder, is present in the system. The **TrapAvailable** member function is shown as follows.

```
Boolean TApp::TrapAvailable(short tNumber,TrapType tType){

    return NGetTrapAddress(tNumber, tType) !=
          GetTrapAddress(_Unimplemented);
}
```

The destructor for **TApp** simply deletes the document list allocated in the destructor. The *fDocList* member holds a pointer to the document list. The destructor uses the **delete** operator to deallocate the list object pointed to by *fDocList*. Notice that **delete** will not crash even if it is passed a pointer equal to zero. The code for **TApp**'s destructor is defined in the declaration of the **TApp** class, as shown here.

```
virtual ~ TApp(void){delete fDocList;}
```

▶ Initializing the Application

The constructor takes care of most of the initialization tasks for the application object, but an additional member function, **InitApp**, is provided to perform other initialization for the application. By default, **InitApp** calls the **GetClipFromSystem** member function to load the system clipboard. **InitApp** returns a Boolean value to indicate if its operations were successful. By default, it returns true. The code for **InitApp** is shown as follows.

```
Boolean TApp::InitApp(void){

   GetClipFromSystem();
   return true;
}
```

By the time **InitApp** is called, you can be sure that the constructor has run and that the *fenvRec* member has been filled in from a call to the toolbox function SysEnvirons. You can override **InitApp** in your derived application class to perform initialization tasks that you don't want to put in the constructor, such as allocating memory or checking the hardware and software environment to see if your application should run, but you must be sure to call the parent class's version of **InitApp** from your derived version. For example, your derived **InitApp** could return false if your application depended on Color QuickDraw and it wasn't available, as shown in the following example.

```
Boolean TMyApp::InitApp(void){

    if(TApp::InitApp())
        return fenvRec.hasColorQD;
    else
        return false;

}
```

Key Point ▶ As shown in Chapters 4 and 5, a derived member function calls the equivalent member function for its parent class by preceding the member function name with the class name and two colons.

▶ Cleaning Up After the Application

The **CleanUp** member function is provided for you to perform any final clean up or memory deallocation before the application terminates. In its default definition it simply calls **GiveClipToSystem** to make sure that the application's private clipboard is made available to subsequent applications, as shown here.

```
void TApp::CleanUp(void){

    GiveClipToSystem();

}
```

If you override **CleanUp**, be sure to call the parent class's **CleanUp** in your version, as shown by the following example.

```
Void TMyApp::CleanUp(void){

    // do your stuff
    // ...

    // and then call the parent class
    TApp::CleanUp();

}
```

▶ Making Documents

One of the main jobs of an application object is to create documents. The member function **MakeDoc** is provided for this purpose. It takes a pointer to an SFReply structure as an argument. As shown in the following declaration, that argument defaults to nil if it is not supplied.

```
virtual TDoc * MakeDoc(SFReply * reply = (SFReply *)nil);
```

The default argument allows you to use **MakeDoc** in two different ways. When you are making a new, blank document that is not associated with an existing file, you can call **MakeDoc** with no argument. If you are creating a document from an existing file, you can pass a pointer to the SFReply that contains the file name and volume reference number for the file.

MakeDoc creates the document dynamically by using the C++ **new** operator in conjunction with the class name of the document it wishes to create. The default definition of **MakeDoc** creates a **TDoc** object, as shown by the following code. You must override **MakeDoc** to create your own derived document type in your application. Chapters 7-13 contain examples of how to override **MakeDoc**.

```
TDoc * TApp::MakeDoc(SFReply * reply){

    return new TDoc(GetCreator(),reply);

}
```

Notice that **MakeDoc** passes two arguments to the constructor for the document object. The first argument is the creator signature for the application, as returned by the member function **GetCreator**. The document will use the creator signature when creating new files so that the files will be associated with the application that created them. The second argument is the SFReply pointer that was passed as the argument to **MakeDoc**. These arguments will be passed to **TDoc**'s constructor. (See Chapter 5 for a description of **TDoc**'s constructor.)

GetCreator is defined by default to return the signature '????'. If your application has a unique signature, override **GetCreator** to return your signature. The Scribble program in Chapter 8 uses a unique application signature to allow it to use its own icons in the Finder and to allow the user to launch the application by opening one of its documents in the Finder.

```
virtual OSType GetCreator(void){return '????';}
```

The **AddDocument** member function provides a way to add a new document to the application's document list and to update the *fCurDoc* member to point to the new document. It is used by other member functions such as **OpenNewDoc** and **OpenOldDoc** after they have created a document object. **AddDocument** also provides a single member function that you can override if you need additional processing when a new document is added to the application. For example, you could use **AddDocument** as a way of updating a menu that listed all open documents. The default definition of **AddDocument** is shown as follows.

```
void TApp::AddDocument(TDoc *theDoc){

    fDocList->AddItem(theDoc);
    fCurDoc = theDoc;
}
```

The next three sections describe three different member functions for creating documents. The first member function creates a blank document when the user chooses the New menu item. The next member function creates a document from an existing file when the user chooses the Open menu item. The last member function creates a document from an existing file when the user starts up the application by opening a document from the Finder. All three of these member functions call **MakeDoc** to actually create the document object; thus, by overriding **MakeDoc** you can control the type of document created in all three situations. Likewise, all three member functions call **AddDocument** to add the new document to the application's document list, so any change to that member function will apply to all three creation member functions.

▶ Opening New Documents

When the user chooses the New menu command, the application object responds by calling the **OpenNewDoc** member function. **OpenNewDoc** calls **MakeDoc** with no argument to create a blank document. It then calls the **MakeWindow** and **InitDoc** document member functions for the new document. If those member functions return true, then it staggers the position of the document window based on how many documents are already open. Once the window, which is still not visible, is positioned, **OpenNewDoc** calls the document member function **ShowDocWindow** to make the window visible. Finally, it calls the application member function **AddDocument** to add the document to the application document list. The code for **OpenNewDoc** is shown as follows.

```
void TApp::OpenNewDoc(void){

    TDoc * newDoc = MakeDoc();
    if(newDoc){
        if((newDoc->MakeWindow(fenvRec.hasColorQD)) &&
            (newDoc->InitDoc())){
            short numDocs = fDocList->NumItems() ;
            newDoc->MoveDocWindow(kHPos + (numDocs * kStagger),
                                  kVPos + (numDocs * kStagger));
            newDoc->ShowDocWindow();
            AddDocument(newDoc);
        } else {
            // MakeWindow or InitDoc failed, but doc created
            delete(newDoc);
        }
    }
}
```

▶ Opening Old Documents

When the user chooses the Open menu command, the application object responds by calling the **OpenOldDoc** member function. **OpenOldDoc** puts up a standard file dialog by calling the toolbox function SFGetFile to allow the user to pick which file to open. The type of files shown in the standard file list is determined by two application member functions, **GetNumFileTypes** and **GetFileTypesList**. By default, **GetNumFileTypes** returns 0 and **GetFileTypesList** returns nil. This causes SFGetFile to display files of all types in the file list. If you want to display files of a certain type only, such as only 'TEXT' or 'PICT' files, you must override **GetNumFileTypes** and **GetFileTypesList**. Chapters 8, 11, and 12 show examples of how to override these member functions to limit the types of files displayed by **OpenOldDoc**. By isolating these actions in separate member functions, you can change the behavior of **OpenOldDoc** without having to actually override **OpenOldDoc**. **GetNumFileTypes** and **GetFileTypesList** are defined in the declaration of **TApp**, shown as follows.

```
virtual int GetNumFileTypes(void){return 0;}
virtual SFTypeList GetFileTypesList(void){return (SFTypeList)nil;}
```

Once the user has dismissed the standard file dialog, **OpenOldDoc** examines the "good" member of the SFReply structure to see if the user canceled the dialog. If good is false, **OpenOldDoc** returns without doing anything more. Otherwise, it assumes that the user chose a file whose name and volume location are contained in the SFReply structure. It then passes that structure to the **InitOldDoc** member function, where the document will be created and the file opened and its contents read in. The code for **OpenOldDoc** is shown as follows.

```
void TApp::OpenOldDoc(void){

    SFReply reply;
    Point p;

    p.h = 100; p.v = 100;
    SFGetFile(p,
          (Str255)"",
          (FileFilterProcPtr)nil,
          GetNumFileTypes(),
          GetFileTypesList(),
          (DlgHookProcPtr)nil,
          &reply);
```

```
    // don't go on if user cancels dialog
    if(! reply.good)
    return;

    (void) InitOldDoc(&reply);

}
```

InitOldDoc is like **OpenNewDoc** in that it creates a document by calling **MakeDoc** and then makes a window and initializes the document. It also staggers the document's window position depending on the number of open documents. In addition, **InitOldDoc** opens the file specified by the SFReply structure. It calls the document member functions **OpenDocFile** and **ReadDocFile**. It also sets the document window title to match the file name. Because so many things potentially can go wrong, the error checking in this member function is rather involved. **InitOldDoc** must carefully clean up, depending on how far it got before the error occurred. The following code shows the definition for **InitOldDoc**.

```
Boolean TApp::InitOldDoc(SFReply * reply){

    TDoc * newDoc = MakeDoc(reply);
    if(newDoc){
        if((newDoc->MakeWindow(fenvRec.hasColorQD))
            && (newDoc->InitDoc())){
            short numDocs = fDocList->NumItems() ;
            newDoc->MoveDocWindow(kHPos + (numDocs * kStagger),
                    kVPos + (numDocs * kStagger));
            newDoc->SetDocWindowTitle((Str255)reply->fName);
            short refNum = newDoc->OpenDocFile(reply);

            if(refNum != 0){
                if(newDoc->ReadDocFile(refNum)){
                    newDoc->ShowDocWindow();
                    AddDocument(newDoc);
                }else{
                   // open was successful, but read failed
                   newDoc->CloseDocFile(refNum);
                   delete(newDoc);
                   return false;
                }
            } else {
                // file not opened successfully, but doc created
                delete(newDoc);
```

```
                return false;
            }
        } else {
            // MakeWindow or InitDoc failed, but doc created
            delete(newDoc);
            return false;
        }
    } else{
        // document not created
        return false;
    }
    // if we get this far, all went well
    return true;
}
```

Key Point ▶

Notice that the same code is used by **OpenNewDoc** and **InitOldDoc** to stagger the window position. This suggests that the code should be split out and put into a separate member function, making it easy for you to modify the staggering behavior without rewriting other parts of the document creation member functions.

▶ Opening Documents from the Finder

The last member function for creating documents will open documents if the user has launched the application by opening one or more of its documents in the Finder. When the user opens an application's document files in the Finder, the Finder launches the application but it doesn't actually open the files. Instead, it places information about the files in a specific area of memory where the application can examine it. It is the application's responsibility to actually open the files. The toolbox functions CountAppFiles and GetAppFiles allow an application to retrieve information about the files that the user asked to be opened.

Usually, you will call the **OpenDocFromFinder** member function after initializing your application. It returns a Boolean result value to indicate whether or not it successfully opened one or more document files from the Finder. A typical scenario at program startup is to open a blank document unless existing files were opened from the Finder, as illustrated by the applications in Chapters 8, 11, and 12.

The **OpenDocFromFinder** member function begins by calling the toolbox function CountAppFiles to see if any files were opened from the Finder. If no files need to be opened, then it simply returns false.

If one or more files need to be opened, then **OpenDocFromFinder** calls the toolbox function GetAppFiles for each file, filling in an AppFile structure with information about the file. The AppFile structure is very similar to an SFReply structure, so **OpenDocFromFinder** copies the information from the AppFile to an SFReply. It then uses the SFReply as the argument to the **InitOldDoc** member function, which was described in the previous section. From this point on, the process of creating the document is the same as it was when the user chose the Open menu item.

One check you must make before calling **InitOldDoc** is to make sure that the file type is one that your application knows how to handle. **OpenOldDoc** was able to filter the file types that appeared in the SFGetFile dialog. Here, however, there is nothing to prevent the user from selecting a group of incompatible files in the Finder and launching the application. Therefore, **OpenDocFromFinder** calls the member function **AcceptableFileType** for each file to make sure that the application can handle the file before trying to open it. **AcceptableFileType** uses the **GetNumFileTypes** and **GetFileTypesList** member functions to get the same list of file types that are used in **OpenOldDoc**. The code for **OpenDocFromFinder** and **AcceptableFileType** is as follows.

```
Boolean TApp::OpenDocFromFinder(void){

    short message;
    short count;
    AppFile theApp;
    SFReply reply;
    Boolean fileOpened = false;

    // see if there are any files to be opened or printed
    CountAppFiles(&message,&count);
    if(count == 0)
        return false;

    for(short i = count;i;i--){
        GetAppFiles(i,&theApp);
        // convert theApp to an SFReply
        reply.good = true;
        reply.fType = theApp.fType;
        reply.vRefNum = theApp.vRefNum;
        reply.version = theApp.versNum;
        unsigned char strLen = theApp.fName[0];
        for(short j = 0; j <= strLen; j++)
            reply.fName[j] = theApp.fName[j];
```

```
        // check here to see if file is an acceptable type
        if(AcceptableFileType(reply.fType))
            // now create the document and open the file
            if(InitOldDoc(&reply))
               fileOpened = true;
    }
    return fileOpened;
}

Boolean TApp::AcceptableFileType(OSType theType){

    int numTypes = GetNumFileTypes();
    OSType *theTypeList = (OSType *)GetFileTypesList();

    if((numTypes == 0) || (theTypeList == nil))
       return true;

    for (int i = 0;i < numTypes; i++){
        if(theType == *theTypeList++)
            return true;
    }
    return false;
}
```

In a simple application, the only document creation member function you must override is **MakeDoc**. Other member functions that you will probably override are **GetNumFileTypes** and **GetFileTypesList** to filter the types of files shown in the standard file dialog. You will probably never need to override the other member functions that control document creation. These member functions are general enough that they should serve in any derived application.

▶ Deleting Documents

There is a single member function, **CloseADoc**, that deletes a document. It is called when the user chooses the Close menu item, when the user clicks on the close box of a document window, or when the application is about to terminate.

CloseADoc calls the **DoClose** member function for the document and checks its return value since the user can cancel a close operation. If the document is closed successfully, **CloseADoc** removes the document from the application's document list. Next, if the document that it just closed was the front document, it sets the application's *fCurDoc*

member to nil. Finally, since the document object was allocated dynamically with the **new** operator, **CloseADoc** uses the **delete** operator to deallocate the object and cause the destructor for the document object to run.

Key Point ▶ **Objects that are allocated dynamically with the new operator must be explicitly deallocated with the delete operator.**

The code for **CloseADoc** is shown as follows. Notice that it returns a Boolean value to indicate if the document was actually closed.

```
Boolean TApp::CloseADoc(TDoc * theDoc){

    if(theDoc != nil)
        if(theDoc->DoClose()){
            fDocList->RemoveItem(theDoc);
            if(theDoc == fCurDoc)
                fCurDoc = nil;
            delete theDoc;
            return true;
        }
    // if we get here, the doc didn't close
    return false;
}
```

It is normally not necessary to override **CloseADoc**, although the Scribble program in Chapter 8 does override it to adjust menus whenever a document is deleted.

▶ Handling Events

Another major task for the application object is responding to user input and operating system events. The member function **EventLoop** is the heart of **TApp**'s event handling capabilities. Once the application is initialized, you call **EventLoop**. This member function repeatedly retrieves and dispatches events until the user chooses the Quit menu item. **EventLoop** is the central switchboard for the application object.

Each time through its loop, **EventLoop** calls the toolbox function FrontWindow to get the WindowPtr to the frontmost window on the screen. It then passes this window to the member function **CheckForDASwitch** to detect when a desk accessory becomes active or inactive, as explained earlier in the "Clipboard Support" section of this chapter.

EventLoop then searches the document list to see if the window matches the window for any of its documents. If a match is found, the member *fCurDoc* is set to point to that document. Thus, *fCurDoc* is always pointing to the document associated with the front window, or it will be nil if the front window doesn't belong to a document (such as when a desk accessory is in front). Many other member functions rely on *fCurDoc* being set correctly since the frontmost document will be given the chance to process most user input. The following code fragment shows how *fCurDoc* is set up in **EventLoop**.

```
void TApp::EventLoop(void){

    int gotEvent;
    EventRecord theEvent;
    WindowPtr theFrontWindow;

    while (fDone == false) {
       theFrontWindow = FrontWindow();

       // find out if a DA is becoming active or inactive
       CheckForDASwitch(theFrontWindow);

       // see if window belongs to a document,
       // FindDoc will return nil if not one of our windows
       fCurDoc = fDocList->FindDoc(theFrontWindow);
```

Once the current document is determined, **EventLoop** calls either WaitNextEvent or GetNextEvent, depending on the setting of the member *fHaveWaitNextEvent*, which was set by the constructor. WaitNextEvent is the MultiFinder-friendly version of GetNextEvent. Either function will retrieve the next pending event from the operating system, returning true if an event was retrieved or false if no event was pending. If no event was received, **EventLoop** calls the **AppIdle** member function and then branches back to the beginning of the loop to fetch another event, as shown by the following code.

```
    if (fHaveWaitNextEvent)
       gotEvent = WaitNextEvent(everyEvent,
                  &theEvent,
                  SleepVal(),
                  (RgnHandle)nil);
    else {
       SystemTask();
       gotEvent = GetNextEvent(everyEvent, theEvent);
    }
```

```
// make sure we got a real event
if (gotEvent == false){
   AppIdle();
   continue;
}
```

Notice that the third argument to WaitNextEvent is specified by calling the member function **SleepVal**. This argument tells WaitNextEvent how long the caller is willing to sleep if no events are pending. Specifying a large value here will make your application give up more time to other applications running under MultiFinder when it doesn't have any pending events. By default, **SleepVal** is defined to return 0. You can override **SleepVal** to return a larger number if your application can afford to give up more idle time to other applications.

If an event was received, **EventLoop** branches to an appropriate event handler member function based on the nature of the event, as shown by the following code. Once the event is handled, **EventLoop** loops back for another event.

```
switch (theEvent.what){
   case nullEvent :
      DoIdle();
      break;
   case mouseDown :
      MouseDown(&theEvent);
      break;
   case mouseUp :
      MouseUp(&theEvent);
      break;
   case keyDown :
   case autoKey :
      KeyDown(&theEvent);
      break;
   case updateEvt :
      UpdateEvt(&theEvent);
      break;
   case diskEvt :
      DiskEvt(&theEvent);
      break;
   case activateEvt :
      ActivateEvt(&theEvent);
      break;
```

```
    case kOSEvent :
      OSEvent(&theEvent);
      break;
    default :
      break;
} // end switch
```

▶ Idle Events

Whenever **EventLoop** tries to retrieve an event and none is pending, it calls the application's **AppIdle** member function. **AppIdle** in turn calls the **DoIdle** member function for the current document object pointed to by *fCurDoc*, as shown here.

```
void TApp::AppIdle(void){
    if (fCurDoc != nil)
        fCurDoc->DoIdle();

}
```

The document closest to the front is given a chance to perform idle time processing. Typical uses for the **DoIdle** document member function are to blink a text insertion cursor or change the mouse pointer shape, as shown by the TextEdit document class in Chapter 12.

You might want to override the application's **AppIdle** member function to call **DoIdle** for all existing documents if your documents used idle time messages to do animation or perform background computation. To accomplish this, you would create an iterator object for the document list, as explained in Chapter 3, and call the **DoIdle** member function for each document in the list, as shown in the following code.

```
void TMyApp::AppIdle(void){

    TIterator iter(fDocList);
    TDoc * nextDoc;

    while (nextDoc = (TDoc *)iter.Next())
        nextDoc->DoIdle();
}
```

▶ Mouse Down Events

When the application event loop detects a mouse down event, it calls the **MouseDown** member function. **MouseDown** calls the toolbox function FindWindow to determine where the mouse down event occurred. It also searches the document list to see if the window where the mouse down event occurred is one of the application's document windows. It then branches to an appropriate member function or toolbox function to handle the event. If the mouse down event was in a document window, **TApp** typically calls a document member function. In other cases, such as a mouse down event in a desk accessory window or in the menu bar, **TApp** handles the event with an application member function or toolbox call. The code for the **MouseDown** member function is shown as follows.

```
void TApp::MouseDown(EventRecord * theEvent){

    WindowPtr theWindow;

    short partCode = FindWindow(theEvent->where, &theWindow);

    TDoc * tempDoc = fDocList->FindDoc(theWindow);

    switch (partCode){
        case inSysWindow :
            SystemClick(theEvent,theWindow);
            break;
        case inMenuBar :
            AdjustMenus();
            long mResult = MenuSelect(theEvent->where);
            if (HiWrd(mResult) != 0){
                DoMenuCommand(HiWrd(mResult),LoWrd(mResult));
                HiliteMenu(0);
            }
            break;
        case inGoAway :
            if TrackGoAway(theWindow, theEvent->where))
                CloseADoc(tempDoc);
            break;
        case inDrag :
            if(tempDoc != nil)
                tempDoc->DoDrag(theEvent);
            break;
```

```
        case inGrow :
            if (tempDoc != nil)
                tempDoc->DoGrow(theEvent);
            break;
        case inZoomIn :
        case inZoomOut :
            if ((TrackBox(theWindow, theEvent->where, partCode)) &&
                (tempDoc != nil))·
                tempDoc->DoZoom(partCode);
            break;
        case inContent :
            if(theWindow != FrontWindow())
                SelectWindow(theWindow);
            else
                if(tempDoc != nil)
                    tempDoc->DoContent(theEvent);
            break;

    }
}
```

One thing to notice is that **MouseDown** calls the member function **AdjustMenus** just before calling the toolbox function MenuSelect in response to a mouse down event in the menu bar. **AdjustMenus** takes care of enabling or disabling individual menu items to reflect the current state of the application. It is discussed in a later section of this chapter.

▶ Key Down Events

Key down events are handled by the **KeyDown** member function. If the command key was pressed along with another key, then **KeyDown** calls the toolbox function MenuKey to handle command-key equivalents to menu items. Notice that it calls the **AdjustMenus** member function before MenuKey to make sure that the enabled menu items are up-to-date.

If the key press was not a menu key, then **KeyDown** calls the **DoKeyDown** member function for the current document to let it handle the event. You will probably never need to override the application's **KeyDown** member function, since most of the functionality for key processing will be in the document. For example, the TextEdit document class responds to key events by calling the toolbox function TEKey. The code for **KeyDown** is shown as follows.

```
void TApp::KeyDown(EventRecord * theEvent){

    char key;
    long mResult;

    key = (char) (theEvent->message & charCodeMask);
    if ((theEvent->modifiers & cmdKey) &&
        (theEvent->what == key Down)){
        // only do command keys if we are not autokeying
        AdjustMenus();    // make sure menus are up-to-date
        mResult = MenuKey(key);
        // if it wasn't a menu key, pass it through
        if (HiWrd(mResult) != 0){
            DoMenuCommand(HiWrd(mResult), LoWrd(mResult));
            HiliteMenu(0);
            return;
        }
    }else
        if (fCurDoc != nil)
            fCurDoc->DoKeyDown(theEvent);
}
```

▶ Activate Events

The application responds to activate events by determining if the window getting the activation message belongs to one of the application's documents. If it does, the application sends the event to the document by calling the document's **DoActivate** member function, as follows.

```
void TApp::ActivateEvt(EventRecord * theEvent){

    WindowPtr theWindow;

    // event record contains window ptr
    theWindow = (WindowPtr) theEvent->message;
    // see if window belongs to a document
    TDoc *tempDoc = fDocList->FindDoc(theWindow);

    if (tempDoc != nil)
        tempDoc->DoActivate(theEvent );
}
```

▶ Update Events

The application responds to update events as it does to activation events, by determining if the window getting the update belongs to one of the application's documents. If so, the application sends the event to the document by calling the document's **DoTheUpdate** member function, as follows.

```
void TApp::UpdateEvt(EventRecord * theEvent){

    WindowPtr theWindow;

    // event record contains window ptr
    theWindow = (WindowPtr) theEvent->message;
    // see if window belongs to a document
    TDoc *tempDoc = fDocList->FindDoc(theWindow);

    if (tempDoc != nil)
        tempDoc->DoTheUpdate(theEvent);
}
```

▶ MultiFinder Support

TApp is designed to be compatible with MultiFinder. The "accept suspend/resume events" bit in the 'SIZE' resource is set (see the "TApp Resources" section of this chapter), causing MultiFinder to send both a suspend event to the application when it is about to give up control to another application and a resume event when it is about to regain control. When **TApp** receives a suspend event, it is responsible for deactivating the front window and for copying the application's private clipboard to the system. When the application receives a resume event, it must activate the front window and copy the system clipboard to its private clipboard if the clipboard has changed while the application was suspended.

The ability to accept suspend and resume events makes it easier and more efficient for MultiFinder to move the application into the background and then back into the foreground. If the application's 'SIZE' resource indicates that it cannot respond to suspend and resume events, then MultiFinder goes through an elaborate ruse to make the application think that a desk accessory window is becoming active when the application is losing control and that a desk accessory window is becoming inactive when the application regains control. This series of events triggers the clipboard conversion and window activation

mechanisms that were discussed earlier in this chapter for the non-MultiFinder environment.

Suspend and resume events are actually sent to the application as a single event type called an OSEvent, with bits in the EventRecord differentiating between suspend and resume. **TApp** defines a set of constants to locate these flag bits in the event. Some flags specify the type of event; another indicates if clipboard conversion is necessary. The **OSEvent** member function decodes the various bits in the EventRecord to determine the exact nature of the event, and then it calls appropriate member functions to process the event. One side effect of this member function is to set the state of the member *fInBackground* to show whether or not the application is in the background.

An OSEvent event is also sent to an application if it specified a mouse region when calling WaitNextEvent and the mouse has moved outside that region. You can process a mouse-moved message just like an Idle event to adjust the mouse cursor shape. Although **TApp** does not use the mouse region feature of WaitNextEvent, it is a good way to reduce the amount of processing time that your application needs while it is waiting for events.

The code for the **OSEvent** member function is shown as follows. Its main feature is that it encapsulates all the details of decoding the flag bits of an OSEvent message.

```
void TApp::OSEvent(EventRecord * theEvent){
    Boolean doConvert;
    unsigned char evType;
    // is it a multifinder event?
    evType = (unsigned char)(theEvent->message >> 24) & 0x00ff;
    switch (evType) {  // high byte of message is type of event
        case kMouseMovedMessage :
            AppIdle(); // mouse-moved is also an idle event
            break;
        case kSuspendResumeMessage :
            doConvert = (theEvent->message & kClipConvertMask) !=
                        0;
            fInBackground = (theEvent->message & kResumeMask) == 0;
            if (fInBackground)
                DoSuspend(theEvent,doConvert);
            else
                 DoResume(theEvent,doConvert);
            break;
    }
}
```

▶ Suspend and Resume

The **DoSuspend** member function is called when the application receives an OSEvent message that indicates it is about to be switched into the background under MultiFinder. This member function receives a pointer to an EventRecord and a Boolean argument indicating whether or not the application should give its clipboard data to the system clipboard. If the application needs to give the clipboard to the system, it calls the **GiveClipToSystem** member function (described in an earlier section of this chapter). It then calls the **DoActivate** member function for the current document to tell it to deactivate. **DoSuspend** clears the activeFlag bit of the modifier member in the EventRecord before passing that structure to **DoActivate**. The code for **DoSuspend** is shown as follows.

```
void TApp::DoSuspend(EventRecord * theEvent,Boolean convertClip){

    if(convertClip)
        GiveClipToSystem();
    if (fCurDoc != nil){
        // tell DoActivate to deactivate
        theEvent->modifiers &= (!activeFlag);
        fCurDoc->DoActivate(theEvent);
    }
}
```

The **DoResume** member function reverses the process outlined for **DoSuspend**. It is called when the application is about to be reactivated under MultiFinder. If the clipboard needs to be updated, **DoResume** calls the **GetClipFromSystem** member function to copy the system clipboard into the application's private clipboard. It then calls the **DoActivate** member function for the current document after first setting the activeFlag bit in the EventRecord. The code for **DoResume** is shown as follows.

```
void TApp::DoResume(EventRecord * theEvent,Boolean convertClip){

    if(convertClip)
        GetClipFromSystem();
    if (fCurDoc != nil){
        // tell DoActivate to activate
        theEvent->modifiers |= activeFlag;
        fCurDoc->DoActivate(theEvent);
    }
}
```

The member functions defined for handling MultiFinder events should not have to be overridden since application behavior in those situations is well defined and will not normally change from one application to the next.

▶ Handling Menus

As explained in Chapter 5, the application class and the document class share responsibility for the menus. The application's constructor loads the menu resources and displays the menu bar. The application also acts as a switchboard for menu commands, handling some commands itself and passing others off to the current document to be processed. During program execution, both the application and document classes adjust the menus to enable and disable menu items based on the current state of the application and the active document.

The default application and document classes know only about the standard Apple, File, and Edit menus, as described in Chapter 5. However, it is easy to override the menu handling member functions to incorporate additional menus, as explained in the following sections and shown in the Scribble program in Chapter 8.

▶ Adjusting Menus

As mentioned in previous sections, the **AdjustMenus** member function is called just before a user makes a menu selection. **AdjustMenus** selectively enables and disables menu items so that only appropriate menu items are enabled when the user makes a selection. For instance, it makes no sense for the Close menu item to be enabled if there is no document currently open.

The application itself is responsible for the state of some of the menu items, such as the New, Open, and Quit items in the File menu. The current document is responsible for enabling many of the other items. **AdjustMenus** calls the current document's member function **AdjustDocMenus** to allow the document to adjust its portion of the menus. If there is no current document, the application disables the items that the document normally controls — unless there is an active desk accessory, in which case the Edit menu items are all enabled.

The code for **AdjustMenus** is shown as follows. Even though it is called between the time the user clicks the mouse on the menu bar and the time when the menu drops down to allow the user to make a choice, it executes quickly enough so that you will not notice any delay.

```
void TApp::AdjustMenus(void){

    MenuHandle menu;

    // first give the current document a chance to adjust the
    // menus
    if(fCurDoc != nil)
        fCurDoc->AdjustDocMenus();

    // Now do the file menu
    menu = GetMHandle(rFileMenu);
    // the app controls whether we can open and new and quit
    SetMenuAbility(menu,iNew,CanNew());
    SetMenuAbility(menu,iOpen,CanOpen());
    SetMenuAbility(menu,iQuit,CanQuit());

    if ( fCurDoc == nil ){
        // no current doc, disable File menu items
        // usually handled by the document
        SetMenuAbility(menu,iClose,false);
        SetMenuAbility(menu,iSave,false);
        SetMenuAbility(menu,iSaveAs,false);
        SetMenuAbility(menu,iPageSetup,false);
        SetMenuAbility(menu,iPrint,false);
    }

    // now the edit menu
    menu = GetMHandle(rEdit);
    // if no current doc, then enable edit menu depending
    // on whether a DA is on top
    if ( fCurDoc == nil ){
        SetMenuAbility(menu,iUndo,fDAonTop);
        SetMenuAbility(menu,iCut,fDAonTop);
        SetMenuAbility(menu,iCopy,fDAonTop);
        SetMenuAbility(menu,iPaste,fDAonTop);
        SetMenuAbility(menu,iClear,fDAonTop);
        SetMenuAbility(menu,iSelectAll,fDAonTop);
    } else {
        // Paste is the one Edit item that the doc can't
        // set by itself
        SetMenuAbility(menu,iPaste,(fClipData != nil) &&
                    (fCurDoc->CanPaste(fClipType)));
    }
}
```

Notice also that the state of each individual menu item is controlled by a short member function such as **CanOpen** and **CanNew**. You can easily control the standard menu items by overriding these member functions rather than having to rewrite **AdjustMenus**.

You can override **AdjustMenus** if your derived application defines additional menus beyond the standard Apple, File, and Edit menus. There are two options available for handling additional menus. If the additional menu items relate to an individual document, such as a Font menu, then the document member functions should probably handle the menu. If, on the other hand, the menu has an effect on the application as a whole, such as a menu that lists all open documents, then application member functions should handle those menu items. Allowing a document class to handle its own menus is particularly useful when the application will support several document types.

If you decide to override **AdjustMenus**, you should include a call to **TApp::AdjustMenus** in your derived version so that the standard File and Edit menu items are enabled and disabled. The following code skeleton shows how you could override **AdjustMenus**.

```
void TDerivedApp::AdjustMenus(void){

    // adjust the additional menu items for the derived application

    // ... //

    // now call the parent class to do default processing
    TApp::AdjustMenus();
}
```

▶ Handling Menu Commands

After the menus have been adjusted, the application must respond to the user's menu item selection. The application member function **DoMenuCommand** is called whenever the user selects a menu command, either with the mouse or with a command-key combination. The main switchboard directs the command to the proper member function for processing. **DoMenuCommand** first calls the document member function **DoDocMenuCommand** to give the document class a chance to process the command. This document member function contains all the information necessary to process menu commands that are unique to that document class. If the current document's **DoDocMenuCommand** returns true, indicating that it handled the command, then **DoMenuCommand** does nothing more. If **DoDocMenuCommand** returns false, indicating that it didn't handle the command, then **DoMenuCommand** proceeds to decode the menu item before dispatching it for processing.

DoMenuCommand handles some of the menu commands by calling application member functions, such as the Open, New, Close, and Quit

menu items. The application also handles the About . . . menu command and menu commands to open a desk accessory. Other menu choices from the standard File and Edit menus are passed on to member functions of the current document. For example, when the user chooses the Cut menu item, the application passes this command on to the current document's **DoCut** member function. The code for **DoMenuCommand** is shown here.

```
void TApp::DoMenuCommand(short menuID, short menuItem){

    short   itemHit;
    Str255  daName;
    short   daRefNum;

 // allow the current doc a chance to handle it first
 if((fCurDoc != nil) &&
    (fCurDoc->DoDocMenuCommand(menuID,menuItem)))
    return;

    switch ( menuID ){
        case rAppleMenu:
            switch ( menuItem ){
                case iAbout:
                    itemHit = Alert(rAboutID, nil);
                    break;
                default:
                    GetItem(GetMHandle(rAppleMenu), menuItem,
                            daName);
                    daRefNum = OpenDeskAcc(daName);
                    break;
            } // end menuItem switch

            break;

        case rFileMenu:
            switch ( menuItem ){
                case iNew:
                    OpenNewDoc();
                    break;
                case iOpen:
                    OpenOldDoc();
                    break;
                case iClose:
                    CloseADoc(fCurDoc);
                    break;
```

```
                case iQuit:
                Quit();
                break;
            } // end menuItem switch

        break;

    case rEdit:
        if ( !SystemEdit(menuItem-1) ){
            switch ( menuItem ){
                case iUndo:
                    DoUndoCmd(fCurDoc);
                    break;
                case iCut:
                    DoCutCmd(fCurDoc);
                    break;
                case iCopy:
                    DoCopyCmd(fCurDoc);
                    break;
                case iPaste:
                    DoPasteCmd(fCurDoc);
                    break;

            } // end menuItem switch
        } // end if

        break;

    } // end menuID switch
}
```

Like the **AdjustMenus** member function, **DoMenuCommand** can be overridden in applications derived from **TApp** if the applications include additional menus. If the new menus affect the application as a whole, then you must override **DoMenuCommand** to process the new menu commands. Otherwise, you can leave **DoMenuCommand** the way it is and put all specialized menu processing into the document class. The Scribble program in Chapter 8 contains an example of how to handle additional menus in this way.

If you do override **DoMenuCommand**, don't forget to include a call to **TApp::DoMenuCommand** to get the default processing for the standard Apple, File, and Edit menu items.

▶ File Menu Commands

As discussed previously, the application responds to some menu commands with application member functions and to others by calling document member functions. This section discusses some of the application member functions used to handle menu commands in the File menu.

The Open and New menu commands are handled by the member functions **OpenOldDoc** and **OpenNewDoc**, respectively. Those member functions are described in previous sections of this chapter.

When the user selects the Quit menu item, the application must make sure that all open documents are safely saved and closed before actually terminating. The **Quit** member function creates an iterator object (iterators are described in Chapter 3) for the application's document list. It then calls the **CloseADoc** member function for each document in the list. In the course of closing a document, **CloseADoc** calls the **DoClose** member function for the document. That member function, in turn, will call the document's **WantToSave** member function if the document has unsaved changes. If the user chooses to cancel during the save operation, we assume that the entire quit process should also be canceled. Thus, if any document is not successfully closed, as indicated by a false return value from **CloseADoc**, then the quit process is aborted. If, in fact, all documents are successfully closed, then **Quit** calls the **ExitLoop** member function, which will break out of the **EventLoop** and cause the program to run to completion. The code for **Quit** is shown as follows.

```
void TApp::Quit(void){
    TIterator iter(fDocList);
    TDoc * nextDoc;
    Boolean OKToQuit = true;

    // ask each doc if it is ready to Quit
    // It is possible that the user may cancel
    // while saving one of these documents,
    // thus aborting the Quit process
    while (nextDoc = (TDoc *)iter.next())
        if(! CloseADoc(nextDoc)){
            OKToQuit = false;
            break;  // don't continue iterating
        }
    If(OKToQuit)
        ExitLoop();
}
```

▶ Edit Menu Commands

Like the member functions that deal with File menu commands, there are also member functions to handle Edit menu commands. **DoMenuCommand** calls these member functions with *fCurDoc* as the argument. The service member functions call the appropriate member functions for the specified document to do most of the actual command processing. It is unlikely that you will ever need to override the member functions described in the following paragraphs.

Cut, Copy, and Paste commands are handled in close conjunction with document member functions. The document member functions for clipboard support expect a handle to a block of clipboard data and a type designator for the data. The application sends its clipboard data to the document for Paste commands and receives data from the document for Cut or Copy commands.

When the user chooses the Paste menu command, **DoMenuCommand** calls the application member function **DoPasteCmd**, passing the current document as an argument. This member function calls the **DoPaste** member function for the specified document, passing the clipboard data and type designator contained in *fClipData* and *fClipType* as arguments. The document takes that data and uses it to complete the Paste command. The code for **DoPasteCmd** is shown as follows. See Chapter 5 for a discussion of the document's **DoPaste** member function.

```
void TApp::DoPasteCmd(TDoc * theDoc){

    if(theDoc != nil)
        theDoc->DoPaste(fClipData,fClipType);
}
```

When the user chooses the Cut menu command, **DoMenuCommand** calls the application member function **DoCutCmd**, passing the current document as an argument. **DoCutCmd** calls the **DoCut** member function for the document. **DoCut** fills in a handle and data type argument with the result of the cut operation. If the document member function returns true, indicating that the cut operation was successful, then **DoCutCmd** disposes of the old clipboard data and assigns the new clipboard data and type designator to the *fClipData* and *fClipType* members. For Copy menu commands, the **DoCopyCmd** member function does essentially the same thing, except that it calls the document's **DoCopy** member function instead of **DoCut**. The code for these two application member functions is shown as follows. See Chapter 5 for a description of the document member functions for cutting and copying.

```
void TApp::DoCutCmd(TDoc * theDoc){

    Handle newData;
    OSType newType;

    if(theDoc != nil)
        if(theDoc->DoCut(&newData,&newType)){
            //get rid of old clip data if DoCut succeeds
            if(fClipData != nil)
                DisposHandle(fClipData);
            fClipData = newData;
            fClipType = newType;
        }
}

void TApp::DoCopyCmd(TDoc * theDoc){

    Handle newData;
    OSType newType;

    if(theDoc != nil)
        if(theDoc->DoCopy(&newData,&newType)){
            //get rid of old clip data if DoCopy succeeds
            if(fClipData != nil)
                DisposHandle(fClipData);
            fClipData = newData;
            fClipType = newType;
        }
}
```

It is unlikely that you will need to override any of the application member functions for handling Edit menu commands. Most of the cut and paste functionality that you need to change is in the document class. See Chapter 12 for an example of how the TextEdit application and document work together to support cut, copy, and paste.

▶ TApp Resources

Like **TDoc**, the **TApp** class depends on several resources. These resources are defined in the file TApp.rsrc. Appendix B contains a rez-compatible listing of the resources in the file TApp.rsrc.r. The resources are listed in Table 6-1. You will need to replace some of the resources for **TApp** with your own application-specific resources. Chapters 7-13 contain examples of how to replace the default resources for **TApp**.

Table 6-1. TApp Resources

Res. Type	*ID No.*	*Description*
'ALRT'	128	Generic About alert window, replace for your application
'DITL'	128	Generic About alert dialog item list, replace for your application
'MBAR'	128	MenuBar for application, replace if you add menus
'MENU'	128	Apple menu, replace to change About item
'MENU'	129	File menu, probably won't replace
'MENU'	130	Edit menu, probably won't replace
'SIZE'	-1	MultiFinder resource, replace to change memory required

▶ Compiling TApp

Like **TDoc**, **TApp** cannot run as a program by itself. It is designed to work with **TDoc**. You do, however, have to compile TApp.cp to create an object file, TApp.cp.o, which can be linked with other files to create a full application.

You also need to make sure that the application object code is placed in its own segment by placing the following compiler directive just before the first line of code. The justifications for segmenting your code are discussed in Chapter 5.

```
#pragma segment AppSeg
```

To compile TApp.cp, invoke the following command in MPW, making sure, of course, that the current directory contains the file TApp.cp:

```
CPlus TApp.cp -o TApp.cp.o
```

That command causes the CPlus script to run, which calls CFront to create C code from your C++ code, and then calls the C compiler to compile the final object code. The code is output as TApp.cp.o, as specified by the -o flag in the command.

▶ Summary

TApp is by far the biggest class that is defined in this book. It encapsulates many fundamental principles of Macintosh programming so that you will never again have to be concerned with details like where particular flag bits are in an OSEvent. It also takes care of the clipboard and menus and opening documents.

There are actually very few member functions in **TApp** that you will have to override in a typical application. But you will almost surely override **MakeDoc** so that it creates one of your derived document objects instead of the generic **TDoc** document. Other member functions that you will probably override specify the type of data that you will support on the clipboard and adjust menus and respond to menu commands.

The hardest part is over. Object-oriented programming often includes a steep learning curve to get started and become familiar with a set of base classes. Now that you have developed a familiarity with C++ techniques and have developed a base document and application class, you are ready to start enjoying the benefits of object-oriented programming. Most of the rest of this book develops applications based on **TApp** and **TDoc**. Those chapters show how to override selected member functions in the application and document base classes to create a variety of programs.

7 ▶ Helloworld, Revisited

The previous two chapters have described, in great detail, an application class and a document class that can make Macintosh programming easier. Now it is time to show how to use those classes. This chapter develops a simple program based on **TApp** and **TDoc**. The program will display multiple windows with the message "hello world" in each window, as shown in Figure 7-1.

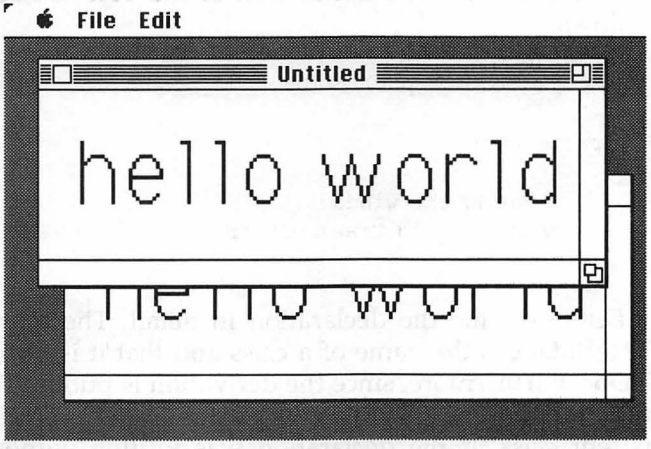


Figure 7-1. The Helloworld2 Application

This chapter describes the parts necessary to build the Helloworld2 application. These parts include the C++ code, resource definitions, and the makefile. Although the Helloworld2 application is very simple, the mechanics of building it will be applicable to more complex applications that appear later in this book. The complete code for the Helloworld2 application is listed in Appendix B.

▶ Subclassing TApp and TDoc

The key to the power of object-oriented programming is the ability to derive new classes from existing classes. A derived class inherits all the behavior of its parent class except for those member functions that you choose to override in the derived class. For the Helloworld2 application, we will derive a new application class from **TApp** and a new document class from **TDoc**.

▶ THelloDoc

The new document class is called **THelloDoc**. The only difference between it and its parent, **TDoc**, is the **Draw** member function. **THelloDoc** documents respond to **Draw** messages by drawing the words "hello world" in the document window. All other behavior is just like the parent class. The declaration of the **THelloDoc** class is shown as follows.

```
class THelloDoc : public TDoc{

    protected:

        // draw the window
        virtual void Draw(Rect *r);
};
```

Let's examine the declaration in detail. The first line declares that **THelloDoc** is the name of a class and that it is derived from the class **TDoc**. Furthermore, since the derivation is public, **THelloDoc** member functions will be able to access all public and protected members of the parent class. If the declaration was written without the public keyword, as shown here

```
class THelloDoc : TDoc{
```

then the users of **THelloDoc** documents could not access the members or member functions of **TDoc**. This is not what we want in this case. We want **THelloDoc** to be just like **TDoc** except in the way it draws the contents of its document windows. Notice that if **TDoc** had any private members or member functions (which it doesn't), **THelloDoc** would not be able to access them, even though it is declared as a public derivation. Private members and member functions are never visible to derived classes.

Key Point ▶

It is a common mistake to forget the public keyword when deriving a class. Remember that if no keyword appears, the default is private.

In the protected section of **THelloDoc**, the **Draw** member function is declared to indicate that this version of **Draw** is to be called rather than the original version in **TDoc**. In order to override the original **Draw** member function, you must declare the new **Draw** with the same argument types as in the original. For example, if you declared **Draw** without the Rect argument, C++ would think that this was simply another version of the **Draw** member function with a different argument list. You want to replace the original **Draw** rather than provide an alternate version, so you must use the same argument list.

Key Point ▶

Although it is highly recommended that the names of the arguments to a derived member function be the same as the parent member function, it is not necessary. Only the types of the arguments are significant. If the original **Draw** member function is declared in **TDoc** as

```
virtual void Draw(Rect *r);
```

then any of the following would override the **Draw** member function in the derived class.

```
virtual void Draw(Rect *r);
virtual void Draw(Rect *theRect);
virtual void Draw(Rect *);
```

This flexibility in defining function interfaces can be useful to suppress certain compiler warnings about unused arguments, as explained in the "Compiler Warnings" section of Chapter 4.

▶ THelloApp

Our new application class is called **THelloApp**. It overrides the member function **MakeDoc** to make a **THelloDoc** object instead of a **TDoc** object. The declaration of **THelloApp** is shown as follows. Notice that the SFReply * argument to **MakeDoc** will be assigned a value of nil if no argument is passed to the member function.

```
class THelloApp : public TApp{
   protected:
      // make our kind of document
      virtual TDoc * MakeDoc(SFReply * reply = (SFReply *)nil);
};
```

THelloApp is declared to be a class that is publicly derived from the **TApp** class. In its protected section we override the **MakeDoc** member function. **MakeDoc** is declared with the same argument types as in **TApp** so that the new version will override the original version, just as discussed for the **THelloDoc Draw** member function. **MakeDoc** is a protected member function since it is called only from other member functions of **TApp**.

▶ Making New THelloDoc Objects

THelloApp overrides the member function **MakeDoc** so that it will make **THelloDoc** objects rather than the default **TDoc** objects. You will want to override **MakeDoc** in almost any program that is based on **TApp** and **TDoc**. The code for overriding **MakeDoc** is shown as follows.

```
TDoc * THelloApp::MakeDoc(SFReply * reply){
   return new THelloDoc();
}
```

MakeDoc takes a pointer to an SFReply structure (which defaults to nil if no pointer is supplied, as specified in the original declaration of the member function) and creates a new **THelloDoc** object. We did not declare a constructor for **THelloDoc** objects, so the constructor for the parent class, **TDoc**, is called by default. Since the constructor for a **TDoc** object has default assignments for both of its arguments (see Chapter 5 for discussion of the **TDoc** constructor), we can also create a new **THelloDoc** without supplying any arguments.

▶ The Draw Member Function

The only part of **THelloDoc** that is different from **TDoc** is its **Draw** member function. By the time **Draw** is called, it assumes that the GrafPort is set correctly. The **Draw** member function is responsible for drawing only the contents of the window. The code for **Draw** is shown as follows.

```
void THelloDoc::Draw(Rect *r){

    EraseRect(r);
    TextSize(48);
    MoveTo(20,65);
    DrawString("\phello world");

}
```

By the Way ▶

C and C++ expect strings to end in the null character. Pascal expects strings to begin with a byte that specifies the length of the string. Toolbox functions such as DrawString expect Pascal strings as arguments. You can tell C++ that you want a string constant to be encoded in Pascal string format by starting the string with \p, as in "\phello world".

You can see that it is quite simpleminded. You might want to spice it up a bit by centering the text or changing the font size since the windows are resizable. Better yet, load the string from a resource so that the program could be easily translated into a foreign language.

▶ The Helloworld2 Main Program

Once the **THelloApp** and **THelloDoc** classes are declared, we can use them in an application. The code for the Helloworld2's main function is shown as follows.

```
void main(void){
    THelloApp theApp;
    // initialize the application
    if(theApp.InitApp()){
        // open one window to start with
        theApp.OpenNewDoc();
```

```
        // Start our main event loop running.
        // This won't return until user quits
        theApp.EventLoop();

        //now clean up
        theApp.CleanUp();
    }
}
```

You declare a **THelloApp** variable at the beginning of main. This causes a **THelloApp** object to be created on the stack (in the local variable space). The constructor for *theApp* will be automatically called when main is entered. Likewise, the destructor for *theApp* will automatically be called when you exit main and the variable goes out of scope. This sort of automatic construction and destruction is one of C++'s most useful features.

Key Point ▶

The automatic construction and destruction of C++ objects based on scope does not occur for C++ object pointer variables. For example, if you defined *theApp* to be a pointer to a TApp object, you would have to explicitly use the new operator to allocate space for it on the heap and cause its constructor to run. When you were done, you would have to use the delete operator to invoke its destructor and deallocate its space, as shown here.

```
int main(void){
    THelloApp * theApp = new THelloApp;
    // initialize the application
    if(theApp->InitApp()){
        // open one window to start with
        theApp->OpenNewDoc();
        // Start our main event loop running.
        // This won't return until user quits
        theApp->EventLoop();
        //now clean up
        theApp->CleanUp();
    }
    // delete theApp so destructor will run
    delete theApp;
}
```

Next, we call **InitApp** to further initialize the application object and check the result of that member function before proceeding. In your applications, you might override **InitApp** to check the current configuration to see if your program should run. For example, you could check for the presence of Color QuickDraw, and return false if it wasn't available, thus aborting the program.

If **InitDoc** returns true, we call **OpenNewDoc** to create a new document. **OpenNewDoc** calls **MakeDoc**, which will make a **THelloDoc** object since we have overridden **MakeDoc**. Once the first document is made, we drop into the main event loop for the application object. When the user finally chooses the Quit menu item, we drop out of **EventLoop** and call **CleanUp**.

This main program seems quite simple, given the fact that the program supports multiple, resizable windows and is MultiFinder-friendly. This is where object-oriented programming begins to really pay off. Most of the functionality of this program is encapsulated in the **TApp** and **TDoc** classes. You only needed to make minimal changes to create this program.

▶ The Helloworld2 Resources

In order to make a Macintosh program, you must gather code and resources together into one file. The **TApp** and **TDoc** classes depend on certain resources (such as menus and windows), being present in the resource file of the program. The resources for the Helloworld2 program are made up of resources from the TApp.rsrc and TDoc.rsrc files, and some resources that are unique to this application. The use of resources is similar to the use of the code for the classes. We only need to define our own resources where those resources are different from the resources defined for the parent classes. The resources that we create specifically for Helloworld2 are in a file named Helloworld2.rsrc. You can use ResEdit to create these application-specific resources, or you can use rez to compile the resources listed in the file Helloworld2.rsrc.r in Appendix B.

The first resource that Helloworld2 overrides is the Apple menu. We want the first item in the Apple menu to read About Helloworld2 . . . instead of About Generic. To do this, define a menu resource with the resource ID number rAppleMenu. (This constant is defined in TApp.h.)

We also want to override the About dialog that describes our application. The resources that define the About box are 'ALRT' and 'DITL' resources with the resource ID number rAbout. (This constant is defined in TApp.h.)

Finally, we want a different window resource than the default defined in TDoc.rsrc. We define a 'WIND' resource with the resource ID number rGenericDoc. (This constant defined in TDoc.h.)

The file Helloworld2.r uses rez to collect all the required resources from the parent resource files and the specific resource file for Helloworld. It uses include statements to load resources from the specified files, as shown in the following code. Default resources come from the parent class resource files TApp.rsrc and TDoc.rsrc. Overridden resources come from the application resource file Helloworld2.rsrc.

```
include "TApp.rsrc" ;
include "TDoc.rsrc" ;
include "Helloworld2.rsrc";
```

Key Point ▶

The order of inclusion in rez files is important because resources with the same type and ID number that are included last will replace resources from the earlier files. Thus, in the preceding example, resources from Helloworld2.rsrc will overwrite similar resources from TApp.rsrc and TDoc.rsrc.

By the Way ▶

Note the difference between #include and include statements in rez source files. Statements that begin with #include are used to bring in text files that contain resource definitions. Include statements are used to read in resources in binary form from the specified file. Note also that include statements end in a semicolon while #include statements do not.

▶ Helloworld2 Makefile: Putting It All Together

Now it is time to build the Helloworld2 application. Many tools and files are required to build an application in MPW. The makefile coordinates all this activity. Figure 7-2 shows the source files and tools required to build the Helloworld2 application.

The MPW Make tool reads a makefile, examines the dependencies specified therein, and decides which tools need to be run to build the program.

Unfortunately, the CreateMake tool that we used in Chapter 4 is not smart enough to write a makefile for the Helloworld2 application because it is not able to specify multiple directories for source and include files. You will have to use the makefile discussed here as a template for creating your own makefiles.

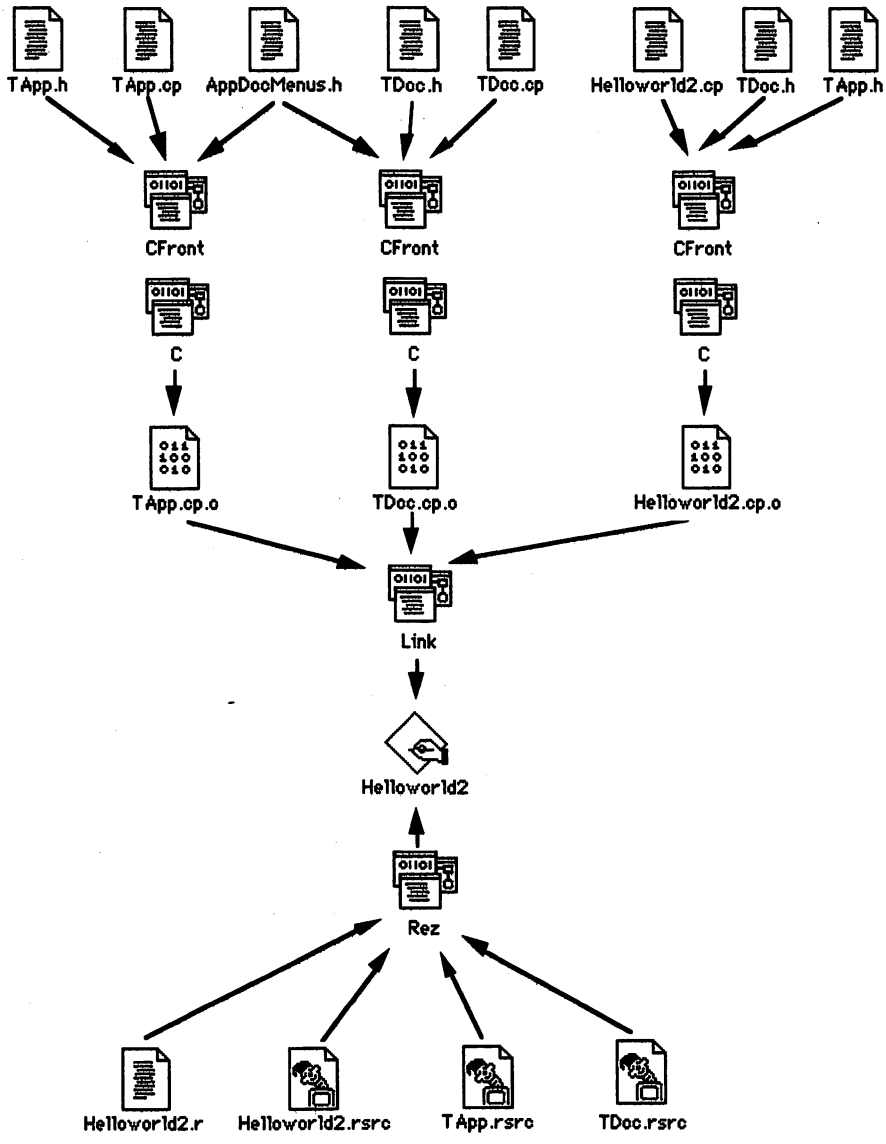


Figure 7-2. Dependencies for Helloworld

The makefile for the application is named Helloworld2.make. We start the makefile by specifying the directory where the sources and object files for **TApp** and **TDoc** reside. This directory tells C++ where to search for the include files TApp.h and TDoc.h. It also tells the linker where to find the object files TApp.cp.o and TDoc.cp.o., and it tells rez where to find the files TApp.rsrc and TDoc.rsrc. Specifying a separate directory for the application and document class sources means that you don't have to copy all these files to the directory where your application is. This saves disk space and also allows you to keep one master copy of the parent class files. Keeping these files in a separate directory is also the reason why you can't use CreateMake to write the makefile for Helloworld2. The following statement in the makefile defines the path name for the directory containing the **TApp** and **TDoc** code and resources.

```
# tell cplus and rez where to find included files for TApp and TDoc
AppObjectDir = ::App-Doc:
```

The directory is specified with a partial path name beginning with a double colon, which means to back up one directory level and start down the new path. This assumes that you have a directory structure where the App-Doc folder and Helloworld2 folder are both in the same parent folder, as shown by Figure 7-3. If your disk is structured differently, you will have to make changes as necessary to specify the correct directory path.

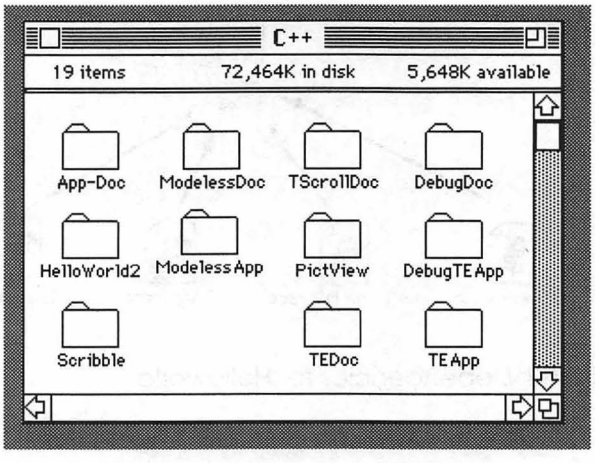


Figure 7-3. Directory Organization

Next, use the directory specified above and other flags to create option specifications for CPlus, link, and rez. The -sym on option tells the compiler and linker to create symbol tables so that the program can be debugged using SADE. This option slows down the compilation and link processes, so you may want to specify -sym off if you aren't using SADE.

```
# use SADE symbol generation
SymOpts = -sym on
```

The -i option for CPlus and rez tells these tools where to look for header files included with the #include directive. The -s option for rez tells it where to look for resources included with the include directive We also pass the SADE symbol definition option that we just defined through to C++.

```
# options for C++
CPlusOptions = {SymOpts} -i "{AppObjectDir}"

# options for rez
RezOptions = -s "{AppObjectDir}" -i "{AppObjectDir}"
```

For the linker, specify the -msg nodup option to suppress warnings about duplicate label definitions during the link phase. These duplicate labels happen because C++ sometimes creates virtual tables for the same class from several different source files. The duplicate label messages can be ignored, so we suppress their output. The other option that we pass to the linker is the SADE symbol definition option, which was defined previously.

```
# options for the linker,
# nodup ignores duplicate labels in object files
LinkOptions = -msg nodup {SymOpts}
```

Next, define a rule that tells Make how to compile C++ files that are out of date. Make contains a default rule for C++ files, but we want to change the rule to include the CPlusOptions that we have defined. The rule says that files that end in cp.o are dependent on similarly named files that end in .cp. For example, the file MyFile.cp.o is assumed, by this rule, to be dependent on MyFile.cp.

```
# We need to change this rule to include CPlusOptions
.cp.o ƒ .cp
    CPlus {default}.cp -o {default}.cp.o {CPlusOptions}
```

Next, we define a list of object files that are needed to build the application. Note that TApp.cp.o and TDoc.cp.o are preceded by the variable that we defined for the App-Doc directory. Helloworld2.cp.o is assumed to be in the current directory. We also define a list of resource files that are needed for the application.

```
Objects = ∂
   "{AppObjectDir}"TApp.cp.o ∂
   "{AppObjectDir}"TDoc.cp.o ∂
   Helloworld2.cp.o

ResourceFiles = ∂
   "{AppObjectDir}"TApp.rsrc ∂
   "{AppObjectDir}"TDoc.rsrc ∂
   Helloworld2.rsrc
```

By the Way ▶

The line continuation character, ∂, is used to show that the next line is actually a continuation of the current line. This allows you to break up long lines of multiple file names into a more readable format. Press Option-d on the keyboard to get a ∂ character.

The default dependency rule for C++ files described previously tells Make what shell commands are necessary to compile C++ source files, but it does not adequately describe all the dependencies for individual C++ files. We must list the dependencies for **TApp** and **TDoc** so that they will be correctly rebuilt whenever any of the files on which they depend are changed.

The dependency rules for TDoc.cp.o and TApp.cp.o are listed as follows. Notice that the dependencies do not include any shell commands to invoke if the dependent files are out of date, so Make uses the commands listed in the default rule.

```
# dependency rules for TDoc and TApp
"{AppObjectDir}"TDoc.cp.o ƒ "{AppObjectDir}"TDoc.cp ∂
      "{AppObjectDir}"TDoc.h ∂
      "{AppObjectDir}"AppDocMenus.h

"{AppObjectDir}"TApp.cp.o ƒ "{AppObjectDir}"TApp.cp ∂
      "{AppObjectDir}"TApp.h ∂
      "{AppObjectDir}"TDoc.h ∂
      "{AppObjectDir}"AppDocMenus.h
```

Finally, define the dependencies for the Helloworld2 application. Helloworld.cp.o is dependent on Helloworld2.cp, TApp.h, TDoc.h, and

Helloworld2.make. If any one of these files has a modification date later than Helloworld2.cp.o, then CPlus will be invoked to recompile Helloworld2.cp.

```
Helloworld.cp.o ƒ Helloworld2.cp ∂
        "{AppObjectDir}"TApp.h ∂
        "{AppObjectDir}"TDoc.h ∂
        Helloworld2.make
```

The application itself, Helloworld2, is defined in multiple dependency rules, as signaled by *ƒƒ* rather than *ƒ*. The first dependency says that Helloworld2 is dependent on all of the object files in the object file list Objects, which was previously defined, and on Helloworld2.make. If this dependency rule is triggered, the program will be relinked with the appropriate object files. After the application file is linked, the SetFile tool is run to set the type, creator, and bundle bit attribute of the application.

```
Helloworld2 ƒƒ {Objects} Helloworld2.make
    Link - o {Targ} {LinkOptions} ∂
        {Objects} ∂
        "{CLibraries}"CPlusLib.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Interface.o
    SetFile {Targ} -t APPL -c '????' -a B
```

The second dependency rule for Helloworld lists the resource files on which Helloworld depends. If any of these files has changed, then rez is invoked to bring the resources in Helloworld2 up-to-date, as directed by the following dependency statement.

```
Helloworld2 ƒƒ Helloworld2.r ∂
        "{AppObjectDir}"TApp.rh ∂
        "{AppObjectDir}"TDoc.rh ∂
        {ResourceFiles}
    Rez -append - o {Targ} {RezOptions} Helloworld2.r
```

All of these definitions and dependencies are combined in a file named Helloworld2.make. When you want to build the Helloworld2 application, you choose the Build... menu command from MPW. When the dialog comes up, specify Helloworld2 as the file you want to build, and then sit back to watch the build process. Make sure that the current directory is set to the folder containing Helloworld2.cp, Helloworld2.r, Helloworld2.rsrc, and Helloworld2.make before trying to build the application.

▶ Debugging Helloworld2 with SADE

Once you have built your program, you can run it. In the unlikely event that the program does not run correctly, you can use SADE to debug it. SADE is a symbolic debugger, which means that it will display your C++ code rather than assembly language while you debug your application.

To use SADE you must compile and link your program with the -sym on option. You must also prepare a short script that tells SADE where to find the source files for your program. Open SADE and create a new file named Helloworld2.sade. (Make sure that you are running MultiFinder, or SADE won't allow you to debug your program.) Type the following lines into the Helloworld2.sade window.

```
directory 'hd:mpw:C++:helloworld2'
sourcepath '::helloworld2','::App-Doc'
target 'Helloworld2'
open source ('Helloworld2.cp')
```

Save the file, select all the lines you just typed, and press the Enter key. This causes SADE to execute the selected text, much like MPW would. In fact, you will notice many similarities between the user interface of MPW and SADE.

Let's examine the commands in Helloworld2.sade. The first line

```
directory 'hd:mpw:C++:helloworld2'
```

sets the current directory to the directory containing the Helloworld application. You should use a complete path name to set the directory here.

The next line

```
sourcepath '::helloworld2','::App-Doc'
```

tells SADE where to look for the source files when it debugs the program. Notice that we specify two directories with relative path names — one for Helloworld2.cp and the other for TApp.cp and TDoc.cp.

The third line

```
target 'Helloworld2'
```

tells SADE the name of the application that you will be debugging. It uses that name to find the symbol file, which in this case will be named Helloworld2.SYM.

The last line

```
open source ('Helloworld2.cp')
```

tells SADE to open the specified source file.

Once the source file, Helloworld2.cp, is open, you can select a line of the source code and choose the GoTill . . . menu command in SADE. SADE will then start executing your program until it reaches the selected line of code. It will then halt your program and display the source file containing the selected line. You can then step through your code, one line at a time, or select another line and GoTill . . . . While stopped, you can also select a variable name and ask SADE to display its value.

SADE is actually capable of much more than described here. It is a very powerful debugging environment, worthy of an entire book in itself. The simple commands described here will get you into SADE and let you watch your code execute and see the value of your variables. This can be extremely helpful in finding bugs. As you gain experience with SADE, you may want to study its documentation to see how its more powerful features can be used.

▶ Summary

Helloworld2 is a very simple program. It builds upon the **TApp** and **TDoc** classes by deriving new application and document classes. These new classes inherit most of the behavior of their parent class, changing only those member functions necessary to give them their own characteristic behavior.

You have seen how to declare and define a derived class. You have also seen how to combine resources from several sources into the final application.

The makefile for even a simple program like Helloworld2 is rather complicated. But the good news is that the makefile for more complicated programs will be about the same as the makefile for Helloworld2. You now know as much about makefiles as you will probably ever need to know.

Finally, we discussed the basic operation of SADE. The ability to symbolically debug your programs will become more important as your applications become bigger and more complex.

Helloworld2 illustrates just how much work is required to get the simplest C++ program running. Think of the time you spent developing the **TApp** and **TDoc** classes and all the intricacies of the makefile. That is why I suggested that C++ is not suitable for quick projects, at

least at first. Now, however, the worst is over. Once the foundation of good base classes has been laid, you can use C++ to dramatically increase your programming productivity. In the next several chapters we will begin to develop more complicated applications. You will see that the complicated programs aren't much harder than the simple programs. That is where C++ really pays off.

PART THREE

▶ Swimming

The previous section was like being thrown into the deep end of a pool; this section is like swimming. Now you have all the basic classes and concepts that you need to start using C++ to write Macintosh programs. You can relax and start to see the benefits of all the hard work you did in the first part of this book.

The chapters in this section build upon the application and document classes presented in the previous section. The projects described in this section show how the original base classes can be extended to provide new functionality without sacrificing the old. This is the heart of object-oriented programming. This is how object-oriented programming is a good idea. This is how object-oriented programming will make you both more productive and a better programmer.

The last chapter of this section develops a program using the MacApp class library instead of the application and document classes used in the other chapters. MacApp is a much more fully developed class system. If you get serious about object-oriented programming on the Macintosh, you will probably want to investigate MacApp further.

8 ▶ Scribble

Scribble is a simple drawing program based on the **TApp** and **TDoc** classes. It can display multiple windows in which the user can draw freehand with the mouse. Scribble can save the drawings to the disk and can later open the files for further drawing. It also adds a menu from which the user can choose several pen sizes and pen patterns. The Scribble screen is shown in Figure 8-1.

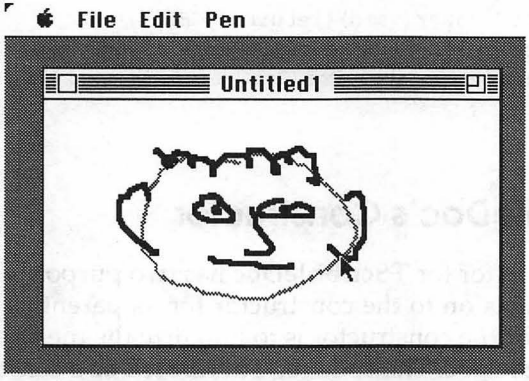


Figure 8-1. The Scribble Application

The Scribble program shows how to extend **TApp** and **TDoc** to use application-specific menus and how to read and write disk files. These two capabilities are essential when using **TApp** and **TDoc** for anything but the most trivial program. Scribble also shows how to define an

application-specific file type and creator so that a user can open the application by opening one or more of its documents from the Finder.

This chapter develops the two new classes, **TScribbleDoc** and **TScribbleApp**, in the context of the application as a whole, so the discussion is somewhat interwoven. See Appendix B for a complete code listing for the classes and the Scribble application.

▶ Making a TScribbleDoc

Just as you did in the Helloworld program in Chapter 7, you need to override **MakeDoc** to make a document that is specific to the application rather than the generic **TDoc**. One difference between this version of **MakeDoc** and the one shown in the Helloworld2 application is that we want to pass actual arguments to the **TScribbleDoc** constructor rather than settling for the defaults. You call **GetCreator** to pass the creator identifier for the Scribble application, 'SCBL', to the new document so that it will be able to correctly set the creator when saving files. You also pass an SFReply pointer to the new document. The *reply* argument will be nil if you are opening a new, blank document, but it will point to an SFReply structure if you are opening an existing document that the user selected with SFGetFile. The code for **MakeDoc** and **GetCreator** is shown as follows.

```
virtual GetCreator(void){return 'SCBL';}
TDoc * TScribbleApp::MakeDoc(SFReply * reply){
    return new TScribbleDoc(GetCreator(),reply);
}
```

▶ TScribbleDoc's Constructor

The constructor for **TScribbleDoc** has two purposes. The first is to pass its arguments on to the constructor for its parent class, **TDoc**. The second task for the constructor is to initialize the members that are specific to **TScribbleDoc**. The code for **TScribbleDoc**'s constructor is shown as follows.

```
TScribbleDoc::TScribbleDoc(OSType theCreator,SFReply *reply ):
    TDoc(theCreator,reply) {
    fPenSize = 2;
    fPattern = patGray;
    fPic = nil;
}
```

The colon following the argument list for **TScribbleDoc** tells the compiler to look for instructions on how to invoke the parent class constructor. In this example, we specify that the constructor for the **TDoc** class be called with *theCreator* and *reply* as arguments. The **TScribbleDoc** constructor passes its arguments off to the constructor for its parent class.

Key Point ▶

Arguments can be passed to the constructor of a parent class by following the argument list of the derived constructor with a colon and an argument list for the parent class. For example, **TScribbleDoc** passes arguments to its parent, **TDoc**, with the following constructor definition.

```
TScribbleDoc::TScribbleDoc(OSType theCreator,SFReply *reply ):
    TDoc(theCreator,reply) {/**/}
```

The constructor for the parent class is executed before the body of the constructor for the derived class. In the case of **TScribbleDoc**, this is of no consequence. But in some classes it can be crucial to remember that the parent constructor has already run to completion before the derived constructor body is executed.

The drawing in a Scribble window is controlled by two members of the **TScribbleDoc** class, *fPenSize* and *fPattern*. The pen size is initialized by the constructor to 2 pixels by 2 pixels and the pen pattern is initialized to gray.

The pen pattern member is an enumerated type that we define to keep track of the possible patterns for a Scribble document. The declaration for the enumerated type is shown as follows.

```
enum penPat{patBlack, patGray, patWhite};
```

Finally, the *fPic* member is a PicHandle that contains a QuickDraw picture of the window's contents. When the document is initialized, the picture is empty, so we set it to nil. Other member functions of the document will set it to refer to a valid picture.

Key Point ▶

If the derived class has any members that are not in the parent class, the derived constructor should initialize them.

▶ Scribbling Functions

As previously discussed, drawing is controlled by the *fPenSize* and *fPattern* members of **TScribbleDoc**. These members are initialized by the constructor and changed when a user chooses items from the Pen menu. Since these members are private, we provide public member functions of **TScribbleDoc** to set and get their values, as shown here.

```
// new member functions related to pen menu
void SetPenSize(short p){fPenSize = p;}
void SetPenPat(penPat p){fPattern = p;}
short GetPenSize(void){return fPenSize;}
penPat GetPenPat(void){return fPattern;}
```

A **TScribbleDoc** document needs to respond to mouse clicks in the content area of its window by allowing the user to draw. It does this by overriding the **DoContent** member function. The application object will take care of calling **DoContent** for the appropriate document when mouse down events occur in a document window.

DoContent sets the GrafPort to the document window, and then it sets the pen size and pen pattern attributes of the port depending on the setting of the *fSize* and *fPattern* members of the document. These settings can be changed when a user chooses an item from the Pen menu. Choosing the menu item only changes the setting of the member; it does not actually change the GrafPort attributes. The attributes must be reset each time you start to draw, in case the member settings have changed since the last time you drew.

Next, **DoContent** determines the starting point from the EventRecord and starts tracking the mouse, drawing lines as fast as it can while the mouse moves. When the user lets up on the button, we set the *fNeedToSave* member to true to show that the document has changed and should be saved before being closed.

Finally, **DoContent** captures the image in the window as a QuickDraw picture so that it can be redrawn if the window is obscured by another window or dialog. **DoContent** checks if there is a current picture in the *fPic* member, and deletes the picture if it is non-nil. Next, **DoContent** calls the toolbox function CopyBits to copy the contents of the window into a new picture. The picture handle is stored in the *fPic* member. The **Draw** member function, explained later, uses the *fPic* picture to redraw the window during update events.

This technique for capturing the window contents has a significant weakness; it captures only what is visible in the window. If the window is made smaller and more drawing occurs, all the previous drawing

that is outside the new window boundaries will be lost the next time the window snapshot is taken. A more serious approach would be to maintain an off-screen bitmap copy of the entire drawing and use CopyBits to move portions of the drawing back and forth to the window.

The code for **DoContent** is shown as follows.

```
void TScribbleDoc::DoContent(EventRecord* theEvent){

    Point newPoint;

    if(fDocWindow){
        SetPort(fDocWindow);
        PenSize(fPenSize,fPenSize);

        if(fPattern == patBlack)
            PenPat(qd.black);
        if(fPattern == patGray)
            PenPat(qd.gray);
        if(fPattern == patWhite)
            PenPat(qd.white);


        GlobalToLocal(&theEvent->where);
        MoveTo(theEvent->where.h, theEvent->where.v);
        do{
            GetMouse(&newPoint);
            LineTo(newPoint.h, newPoint.v);
        }while(StillDown());

        fNeedtoSave = true;

        // now take a snapshot of window
        if(fPic != nil)
            KillPicture(fPic);
        fPic = OpenPicture(&fDocWindow->portRect);
        CopyBits(&fDocWindow->portBits,
                 &fDocWindow->portBits,
                 &fDocWindow->portRect,
                 &fDocWindow->portRect,
                 srcCopy,
                 (RgnHandle)nil);
        ClosePicture();
    }
}
```

As mentioned previously, **TScribbleDoc** also overrides the **Draw** member function to redraw the window contents during update events. The **Draw** member function uses the *fPic* member as a PicHandle and calls the toolbox function DrawPicture to draw the picture into the window. Notice that we extract the original rectangle from the picture and use it as the destination rectangle. Another option would be to use the window's portRect as the destination, which would result in the picture being scaled to fit in the window. The code for the **Draw** member function is as follows.

```
void TScribbleDoc::Draw(Rect *r){

    if(fPic != nil)
        DrawPicture(fPic,&((**fPic).picFrame));

}
```

TScribbleDoc overrides **DoDrawGrowIcon** so that the grow box is not drawn in the window. The user can still resize the window, but the grow box is not drawn so as not to overwrite the picture in the window.

▶ File Type and Creator

Scribble reads and writes disk files that enable it to save and restore drawings. To do this it must create files with a file type and file creator that are specific to Scribble. File type and file creator identifiers are four-character constants such as 'TEXT', 'PICT', and 'MSWD'. The Finder uses the file type and creator to link applications with their own document files. The connection between a document file and its creator allows a user to start up an application by opening one of its documents in the Finder. When the user opens a document in the Finder, the Finder launches the application that has the same creator signature as the document. File types allow applications that support more than one document type to differentiate among its documents, and it also allows applications that share a file type to share document files. The Scribble application uses 'SCBL' as the creator and 'SPCT' for the file type for its files.

TApp and **TDoc** have generic support built in for application-specific file types and creators, but you need to override several member functions in both the application and document classes to specify the file type and creator you will be using in your application.

In the document class, you override **GetDocType** to return the file type for your document. Other parts of the **TApp** and **TDoc** objects call this member function when they need to know the type of the document files. **TScribbleDoc** overrides **GetDocType** as follows.

```
virtual OSType GetDocType(){return 'SPCT';}
```

In the application class, you must override the **GetCreator** member function to return the creator identifier for the application. An application can have only one creator signature, although it might support several different document types and file types. **TScribbleApp** overrides **GetCreator** as follows.

```
virtual OSType GetCreator(void){return 'SCBL';}
```

File types are also used when **TApp** calls SFGetFile to put up a standard file dialog to enable the user to open a file. If your application supports a specific file type, then you want files of only that type to be displayed in the dialog. **TApp** contains two member functions, **GetNumFileTypes** and **GetFileTypesList**, that you will override to specify which file types should be displayed in the file dialog. Since Scribble only supports one document type, **GetNumFileTypes** is overridden to return 1, as follows.

```
// file info for SFGetFile
virtual int GetNumFileTypes(void){return 1;};
```

GetFileTypesList is overridden to return a pointer to an array of file type identifiers. We declare a global variable, *gtheTypes*, as an SFTypeList and initialize it with a single file type, 'SPCT', as shown in the following code. (Notice the ability to initialize an array as part of its definition.) **GetFileTypesList** then returns a pointer to *gtheTypes*, as shown here.

```
// define as a global variable
SFTypeList gtheTypes = {'SPCT'};

// override this member function to return ptr to types list
SFTypeList TScribbleApp::GetFileTypesList(void){
    return gtheTypes;
}
```

▶ Opening Files from the Finder

Once you have defined a file type and creator for your application's documents, you will be able to take advantage of **TApp**'s mechanism for opening documents from the Finder. If the application was launched by a user opening one or more documents from the Finder, **OpenDocFromFinder** opens the documents and returns true. Scribble checks the return value from **OpenDocFromFinder** so that it doesn't open a blank document at startup if other documents are already open.

```
if(! theApp.OpenDocFromFinder())
    theApp.OpenNewDoc();
```

▶ Reading and Writing Scribble Files

The final **TDoc** member functions you need to override to support document files are **WriteDocFile** and **ReadDocFile**. These two member functions handle the low-level tasks of transferring the document data to and from the actual disk file. Other member functions take care of creating, opening, and closing the files, and these do not typically need to be overridden. But **TDoc** has no way of knowing the structure of the content of the files, so you must provide this functionality.

WriteDocFile takes the content of the *fPic* handle, which contains the QuickDraw picture of the window contents, and writes it to the file. The code is straightforward, as shown here. Notice that we reset the file position to the beginning of the file before writing. This precaution is necessary since the file remains open as long as the document is open and other operations (such as the original read operations) could have moved the position indicator in the file. **WriteDocFile** returns true if the file operations are successful, false if otherwise. Other member functions that call **WriteDocFile** use the result to determine if the larger operation (such as a save operation) was successful.

```
Boolean TScribbleDoc::WriteDocFile(short refNum){

    if(fDocWindow){
        if(fPic != nil){
            long len = GetHandleSize((Handle)fPic);
            HLock((Handle)fPic);
            OSErr err = SetFPos(refNum,fsFromStart,0);
            err = FSWrite(refNum,&len,(Ptr)*fPic);
            HUnlock((Handle)fPic);
```

```
            if(err == noErr)
                return true;
            else
                return false;
        }
    }
    // if there ain't no window...
    return false;
}
```

ReadDocFile reads the contents of a document file and places it in a handle which is then used as a QuickDraw picture of the window contents. **ReadDocFile** starts by allocating a new handle to hold the file contents. Notice that it displays an error alert inherited from **TDoc** if it can't allocate enough memory. If the memory allocation is successful, **ReadDocFile** reads the data from the file and extracts the picFrame rectangle from the picture data. It then attempts to make the document window the same size as the picture, limited by the size of the screen. Like **WriteDocFile**, **ReadDocFile** returns true or false to indicate if the file operations were successful or not. The following code shows the definition of **ReadDocFile**.

```
Boolean TScribbleDoc::ReadDocFile(short refNum){
const short kHAdjust = 50;
const short kWAdjust = 40;
   if(fDocWindow){
      long len;
      OSErr err = GetEOF(refNum,&len);
      Handle thePic = NewHandle(len);
      if(thePic == nil){
         ErrorAlert(rDocErrorStrings,sNoMem);
         return false;
      }

      HLock(thePic);
      err = SetFPos(refNum,fsFromStart,0);
      err = FSRead(refNum,&len,(Ptr)*thePic);
      HUnlock(thePic);
      if(err == noErr){
         // now make the window the size of the picture
         Rect r = ( **((PicHandle)thePic)).picFrame;
         short height = r.bottom - r.top;
         short width = r.right - r.left;
         r = qd.screenBits.bounds;
```

```
            height = min(height ,r.bottom - r.top - kHAdjust);
            width = min(width, r.right - r.left - kWAdjust);
            SizeWindow(fDocWindow, width, height, true);

            // set the member to reference Picture
            fPic = (PicHandle)thePic;
            return true;
         } else {
            DisposHandle(thePic);
            return false;
         }
      }
      // if there ain't no window...
      return false;
}
```

All the other details of handling files are handled by the default member functions of **TApp** and **TDoc**. You have to override only those member functions that pertain to the specific nature of your files, and the base classes do the rest. For example, if the user tries to close a Scribble document that has unsaved changes, **TDoc** will ask if the file should be saved before closing. This also will happen if the user tries to quit with unsaved documents. All of that functionality comes from the base classes.

▶ Handling Menus

There are two main areas where Scribble must deal with menus. First, since it supports reading and writing files, it must make sure that the Open, Save, and SaveAs menu items of **TApp** are enabled and disabled appropriately. Second, it must support the Pen menu, about which the parent class knows nothing.

▶ Handling Menu Commands

You must override the document member function **DoDocMenuCommand** to handle selections from the Pen menu, which the **TDoc** class knows nothing about. **DoDocMenuCommand** is called from the application class member function **DoMenuCommand**. The derived **DoDocMenuCommand** for Scribble documents returns true if the menu item is from the Pen menu, thus telling the application that it doesn't need to do any further processing with the menu command. If the menu item isn't in the Pen menu, then you pass the command on to the

parent class by calling **TDoc::DoDocMenuCommand** in case the menu item is one of the default items handled by the document class. The code for **DoDocMenuCommand** is shown as follows.

```
Boolean TScribbleDoc::DoDocMenuCommand (short menuID, short
menuItem){

   if( menuID == rPenMenu){
      switch ( menuItem ){
         case i1X1:
            SetPenSize(1) ;
            break;
         case i2X2:
            SetPenSize(2) ;
            break;
         case i3X3:
            SetPenSize(3) ;
            break;
         case iBlack:
            SetPenPat(patBlack) ;
            break;
         case iGray:
            SetPenPat(patGray) ;
            break;
         case iWhite:
            SetPenPat(patWhite) ;
            break;
         default:
            return false; // this should never happen
      }
      // tell the app that we handled this menu item
      return true;
   } else{
      // it's not one of our menus, give the parent class a chance
      return TDoc::DoDocMenuCommand(menuID,menuItem);
   }
}
```

▶ Adjusting Menus

As explained in Chapters 5 and 6, the application and document classes enable individual menu items by calling query member functions. The Open menu is controlled by the **TApp** member function **CanOpen**, which returns false by default in the definition of **TApp**. Since we want the Scribble application to be able to use the Open menu command, we must override **CanOpen** in **TScribbleApp** so that it returns true. **CanOpen** is defined in the declaration of **TScribbleApp** as follows.

```
virtual Boolean CanOpen (void){return true;}
```

Likewise, the **TDoc** member functions **CanSave** and **CanSaveAs** control the Save and SaveAs menu items. You must override **CanSaveAs** to return true, since it returns false by default. You don't have to override **CanSave**, because it is defined in **TDoc** to return the *fNeedtoSave* member of the document. Thus, as long as you correctly set *fNeedtoSave* to true whenever the document has unsaved changes (**TScribbleDoc** does this in the **DoContent** member function), the **CanSave** member function will work correctly without needing to be overridden. **CanSaveAs** is defined in the declaration of **TScribbleDoc** as follows.

```
virtual Boolean CanSaveAs(void){return true;}
```

Just as you extended the document's **DoDocMenuCommand** to include the Pen menu, so you must extend its **AdjustDocMenus** member function in the same way. The derived **AdjustDocMenus** makes calls to document member functions to determine the current pen width and pen pattern and sets the checkmarks on the menu items accordingly so that the menu reflects the settings of the current document. The definition for **AdjustDocMenus** is shown as follows. Notice that it also calls **AdjustDocMenus** for the parent class to make sure that the default processing gets done.

```
void TScribbleDoc::AdjustDocMenus(void){

   // Do the pen menu
   MenuHandle menu = GetMHandle(rPenMenu);

   CheckItem(menu,i1X1,GetPenSize() == 1);
   CheckItem(menu,i2X2,GetPenSize() == 2);
   CheckItem(menu,i3X3,GetPenSize() == 3);
   CheckItem(menu,iBlack,GetPenPat() == patBlack);
```

```
    CheckItem(menu,iGray,GetPenPat() == patGray);
    CheckItem(menu,iWhite,GetPenPat() == patWhite);

    // now let the parent class have a shot at the menus
    TDoc::AdjustDocMenus();
}
```

By default, **AdjustDocMenus** is called whenever the user clicks the mouse button in the menu bar, just before the menu is pulled down. This works fine as long as you are only enabling and disabling individual menu items. In Scribble, however, you want to disable the entire Pen menu, including the title in the menu bar, when there is no active Scribble document. If you specify item number 0 to the toolbox function DisableItem, it will disable all the items, including the menu title. However, the title will not be redrawn in its disabled state unless you also call the toolbox function DrawMenuBar. Similarly, passing item number 0 to the toolbox function EnableItem will enable all items and the menu title, but it will not show the change until you call DrawMenuBar.

It is annoying to redraw the menu bar every time the user makes a menu selection. What we really want is for the menu to change only when a Scribble document is activated or deactivated. When a Scribble document is activated, it should enable the entire Pen menu and redraw the menu bar. When the document is disabled, it should disable the entire Pen menu and redraw the menu bar. We can extend the document member functions **Activate** and **Deactivate** to perform these tasks. In addition, since there is no deactivation event when a window is closed, we must also override the **DoClose** document member function to disable the Pen menu and redraw the menu bar.

TScribbleDoc defines a utility member function, **TogglePenMenu**, to enable or disable the menu and redraw the menu bar, shown as follows.

```
void TScribbleDoc::TogglePenMenu(Boolean enable){

    MenuHandle menu = GetMHandle(rPenMenu);
    SetMenuAbility(menu,kEveryItem,enable);
    DrawMenuBar();
}
```

TScribbleDoc overrides the **Activate**, **Deactivate**, and **DoClose** member functions to call **TogglePenMenu** in addition to their default actions. Each of these functions first calls the parent class's version of the function to get the default processing and then goes on to call **TogglePenMenu**. This is another example of extending the functionality of a class. These three member functions are overridden as follows.

```
void TScribbleDoc::Activate(void){
   TDoc::Activate();
   TogglePenMenu(true);
}

void TScribbleDoc::Deactivate(void){
   TDoc::Deactivate();
   TogglePenMenu(false);
}

Boolean TScribbleDoc::DoClose(void){
   if(TDoc::DoClose()){
      TogglePenMenu(false);
      return true;
   } else
      return false;
}
```

▶ The Scribble Application

The main application for Scribble is exactly like the main application for the Helloworld2 application described in Chapter 7, with one exception. If Scribble was launched by the user opening a Scribble document from the Finder, then we shouldn't open a blank document at program startup since one or more existing documents will be opened instead. You decide whether to open a blank document by looking at the return value from **OpenDocFromFinder**. The code for the main program is shown as follows.

```
void main(void) {
   TScribbleApp theApp;
   // initialize the application
   if(theApp.InitApp()){
      // open one window to start with
      // unless files were opened from the Finder
      if(! theApp.OpenDocFromFinder())
         theApp.OpenNewDoc();
      // Start our main event loop running
      theApp.EventLoop();
      //now clean up
      theApp.CleanUp();
   }
}
```

▶ Scribble Resources

The Scribble program's Pen menu is defined in the file Scribble.rsrc. Just as you did in the Helloworld2 program, you include resources from TApp.rsrc and TDoc.rsrc first, and then include the resources from Scribble.rsrc to replace those resources you want to change.

Scribble.rsrc contains both an Apple menu with the correct About . . . item and the Pen menu. In order to include the Pen menu, the default 'MBAR' resource in TApp.rsrc must be replaced by a new 'MBAR' resource that contains the Apple, File, Edit, and Pen menu identifiers. You can define the new 'MBAR' in Scribble.rsrc and include it in the application to replace the 'MBAR' from TApp.rsrc. You also want to override the default About dialog, so we define a new 'ALRT' and 'DITL' resource in Scribble.rsrc and include these instead to replace the equivalent resources from TApp.rsrc. Finally, Scribble.rsrc contains 'BNDL' and 'ICN#' resources that allow the Finder to display the Scribble application and its documents with their own icons. The rez source file for Scribble is shown here:

```
include "TApp.rsrc" ;
include "TDoc.rsrc" ;
include "Scribble.rsrc" ;
```

▶ The Scribble Makefile

As promised, the makefile for Scribble is almost exactly the same as the makefile for the Helloworld2 application described in Chapter 7. See Appendix B for a complete listing of Scribble.make. The most important difference is that the creator must be set correctly for the application. The following SetFile command, which comes just after the Link step, shows how this is done. Notice also that we set the Bundle attribute of the application so that the Finder will be able to connect the application to its document files.

```
SetFile {Targ} -t APPL -c SCBL -a B
```

▶ Summary

The main features illustrated by the Scribble program are how to read and write application-specific files and how to incorporate application-specific menus. Both of these techniques are crucial to developing useful applications based on **TApp** and **TDoc**.

For an application to use its own files, its document class must override **ReadDocFile** and **WriteDocFile**. The document class must also tell the application that it can save files by returning true for the **CanSaveAs** member function. One other responsibility of the document class is to override **GetDocType** to return the OSType of the files that it uses.

Likewise, for an application to use its own files the application class must override **CanOpen** to return true so that the Open menu item will be enabled. It must also override **GetNumFileTypes** and **GetFileTypesList** to provide the parameters for SFGetFile. Finally, it must override **MakeDoc** to provide the creator signature to the new document so that newly created files will have the correct creator.

If the document and application classes override the member functions named in the previous two paragraphs, then opening documents by double-clicking on a document in the Finder is automatically supported by **TApp**. The only responsibility of the derived application is to check the return value of **OpenDocFromFinder** to see if files actually were opened with the application. Scribble uses this technique to decide whether to open a blank document at startup.

For a document class to use its own menus, the derived document class must override **AdjustDocMenus** and **DoDocMenuCommand**. These overridden member functions do the processing that is necessary for the document-specific menus, and then call the inherited version of the member functions to perform the general menu processing in **TDoc.** Similarly, the application class can extend **AdjustMenus** and **DoMenuCommand** to handle additional menus that pertain to the application. This technique of extending the function by calling the parent's member function as part of the derived class's member function is very important in object-oriented programming.

9 ▶ Modeless Dialog Documents

This chapter develops a modeless document class, **TModelessDoc**, derived from the generic document class described in Chapter 5. The modeless document class allows you to create a document whose window is a modeless dialog instead of a standard document window. This lets you use the toolbox Dialog Manager to handle buttons and other types of controls within the window.

As its name suggests, this class creates modeless dialogs. A modeless dialog acts just like a normal document window in that it can be moved about the screen and go behind another window when the other window is selected. In contrast, a modal dialog will not permit the user to select any other window until the dialog is dismissed.

This class is relatively simple to derive from **TDoc**. You need to change the member functions that create the document window and handle events. The public interface for the class will be the same as the public interface for **TDoc**, so the application class described in Chapter 6 will be able to use documents from the **TModelessDoc** class in exactly the same way as it uses documents from **TDoc**. The ability to substitute a derived class in place of the parent class is one of the best reasons to use an object-oriented language.

TModelessDoc is intended to be a base class. That is, it is not very useful unless you derive your own class from it. But you will find that deriving a class from **TModelessDoc** typically requires that you override only one or two member functions and define the resources for your dialog window. The second half of this chapter develops a class derived from **TModelessDoc** and uses it in a small application. The complete code for the **TModelessDoc** class and the application that uses it are listed in Appendix B.

▶ TModelessDoc Constructor and Destructor

The constructor for **TModelessDoc** does nothing except pass its arguments on to its parent class, **TDoc**. The syntax for passing constructor arguments to a parent class is described for **TScribbleDoc**'s constructor in Chapter 8. One difference between the example in Chapter 8 and the constructor defined here is that this definition is done within the declaration of the class. In other words, **TModelessDoc**'s constructor is declared and defined at the same time in the file TModelessDoc.h. The ability to combine declaration and definition is especially useful when the body of the member function is empty, as it is here for **TModelessDoc**'s constructor.

```
TModelessDoc(OSType theCreator = '????',
             SFReply * SFInfo = (SFReply *)nil):
        TDoc(theCreator,SFInfo)
{}
```

The destructor for **TModelessDoc** calls the toolbox function DisposDialog to delete the dialog window that was allocated for the document. (See the next section for a discussion of how the dialog window is created.) Notice that we typecast the *fDocWindow* member, which is declared as a WindowPtr, to be a DialogPtr so that we can pass it to DisposDialog. The definition for **TModelessDoc**'s destructor is shown as follows. Like our other derived document classes, this destructor is declared to be virtual so that it will be called even though the document is accessed through a **TDoc** pointer.

```
TModelessDoc::~TModelessDoc(void){

   if(fDocWindow){
      DisposDialog((DialogPtr)fDocWindow);
      fDocWindow = nil;
   }
}
```

The destructor for the parent class, **TDoc**, will be called automatically after the destructor for **TModelessDoc**. For this reason, **TModelessDoc**'s destructor sets the *fDocWindow* member to nil so that **TDoc**'s destructor will not try to delete a window that has already been deleted.

▶ Making the Dialog Window

TModelessDoc overrides the **MakeWindow** member function to create a dialog window instead of a normal document window. It calls the toolbox function GetNewDialog to load the specified 'DLOG' and 'DITL' resources. The code for **MakeWindow** is shown as follows.

```
Boolean TModelessDoc::MakeWindow(Boolean /*colorWindow*/){

      fDocWindow = (WindowPtr)GetNewDialog(GetWinID(),nil,
                                           (WindowPtr)-1);

      return (fDocWindow != nil);

}
```

Notice that we don't use the *colorWindow* argument to **MakeWindow**, so we comment it out to prevent compiler warnings. There is no special toolbox call to create color dialogs. If you want a color dialog you must create a 'dctb' (dialog color table) resource with the same ID number as the 'DLOG' resource. GetNewDialog automatically makes a color dialog if it finds a matching 'dctb' resource when creating the dialog.

Notice also that we typecast the DialogPtr returned by GetNewDialog to be a WindowPtr so that we can assign it to the *fDocWindow* member of the document.

MakeWindow calls **GetWinID** to get the resource ID when it calls GetNewDialog. **TModelessDoc** overrides **GetWinID** to return the constant rGenericDialog, which is defined in TModelessDoc.h. You will want to override this function if you define your own 'DLOG' and 'DITL' resources with a different ID number. **GetWinID** is shown as follows. You don't have to override **GetWinID** if you define your 'DLOG' and 'DITL' resources with an ID number of rGenericDialog. The default definition of **GetWinID** is also shown.

```
short TModelessDoc::GetWinID(void){
   return rGenericDialog;
}
```

▶ Handling Events

One reason to use dialogs instead of regular document windows is that the toolbox Dialog Manager will take care of much of the event handling that you would normally have to do for a regular window. The following sections describe how to modify the event handling member functions of **TDoc** to use the Dialog Manager.

▶ Using the Dialog Manager

The Dialog Manager functions IsDialogEvent and DialogSelect are especially designed to support modeless dialogs. Each time you receive an event you pass the event to IsDialogEvent to see if the event involves a dialog window. If IsDialogEvent returns true, you call DialogSelect, passing the event as an argument along with pointers to a DialogPtr variable and a short integer variable. If the event involves an enabled item in the dialog, such as a mouse click on a button or a text entry in a text edit control, then DialogSelect returns true and fills in the DialogPtr and short integer arguments to indicate the identity of the dialog and the item affected by the event. For update and activate events, DialogSelect updates or activates the dialog window and returns false, indicating that no further processing of this event is necessary.

TModelessDoc defines a new member function, **DoDialogEvent**, to handle all events. It calls IsDialogEvent and DialogSelect to determine if the event affects an enabled dialog item and to automatically handle updates and activation events. If an enabled item is involved in the event, then it calls another new member function, **DoItemHit**, to specifically respond to the event. The code for **DoDialogEvent** is shown as follows.

```
void TModelessDoc::DoDialogEvent (EventRecord* theEvent)(

   short itemHit;
   DialogPtr theDialog;

   if (IsDialogEvent(theEvent)){
      if(DialogSelect(theEvent,&theDialog,&itemHit))
         DoItemHit(theDialog,itemHit);
   }
}
```

▶ Responding to Individual Item Hits

DoDialogEvent calls the **DoItemHit** member function when it detects an event that affects a particular dialog item. **DoItemHit** is responsible for responding to the event. For example, **DoItemHit** might respond to a click on the OK button of a dialog by initiating some process. **DoItemHit** is the member function that you must override in your derived dialog document class. It is the member function that embodies information about the function of individual dialog items in your document. **DoItemHit** is defined, as follows, as an empty function in the declaration of **TModelessDoc.** The sample derived class described in the second half of this chapter shows an example of how to override **DoItemHit.**

```
virtual void DoItemHit(DialogPtr theDialog,short theItem){}
```

▶ Receiving Events from TApp

DoDialogEvent and **DoItemHit** are both new member functions that are unique to this class. They are both declared in the protected section of the class declaration. Because they are protected, the application class cannot call them directly. You must override the public event handling member functions for the document class so that they call **DoDialogEvent**. Thus, the application treats the dialog document just like a regular document, but the dialog document uses its own internal implementation to handle the events that are passed to it by the application. **TModelessDoc** overrides the following four event handling member functions so that they pass the event on to **DoDialogEvent.**

```
virtual void DoActivate(EventRecord* theEvent)
     {DoDialogEvent(theEvent);}
virtual void DoTheUpdate(EventRecord* theEvent)
     {DoDialogEvent(theEvent);}
virtual void DoContent(EventRecord* theEvent)
     {DoDialogEvent(theEvent);}
virtual void DoKeyDown(EventRecord* theEvent)
     {DoDialogEvent(theEvent);}
```

In addition, you must also override the **DoIdle** member function so that it calls **DoDialogEvent**. This is necessary so that the insertion point will blink in edit text items in the dialog. Since **DoIdle** does not have an EventRecord argument, you must create one as a local variable, set the *what* field of the record to be a null event, and then pass the event on to **DoDialogSelect**. The code for **DoIdle** is shown as follows.

```
void TModelessDoc::DoIdle(void){

   EventRecord theEvent;
   theEvent.what = nullEvent;

   DoDialogEvent(&theEvent);

}
```

▶ Dialog Windows Shouldn't Be Resized

One final issue for event handling involves the ability to resize the window. Normally, a dialog window should not be resized, so you need to override the **DoGrow** and **DoDrawGrowIcon** member functions to disable this functionality. Both of the member functions are redefined as empty member functions, as shown here. If you want to be able to resize your dialog windows, then override these member functions in your derived class to restore the functionality that was present in the original **TDoc** class.

```
// disable grow actions
virtual void DoGrow(EventRecord* theEvent)
        {}
virtual void DoDrawGrowIcon(void)
        {}
```

▶ Using TModelessDoc: A Sample Application

The rest of this chapter describes a sample application that uses a document class derived from the modeless dialog class. The derived dialog class is very simple; it contains a single button and a user item that fills a rectangle with a series of ever smaller rectangles, as shown in Figure 9-1.

Even though this application is very simple, it illustrates several key points related to using the modeless dialog class. First, it shows how to use dialog user items. User items enable you to install your own procedures to draw in the dialog window. User items give you almost unlimited opportunities to customize the look of your dialogs and make them appear more professional. Next, the derived dialog document class shows how to override the **DoItemHit** member function to respond to user input in your dialog. You must override this member function in

your dialog class since the base class defines this member function as empty. Finally, the sample class shows how to define 'DLOG' and 'DITL' resources for the dialog. You must define these resources for your derived class since the base class contains no default resources.

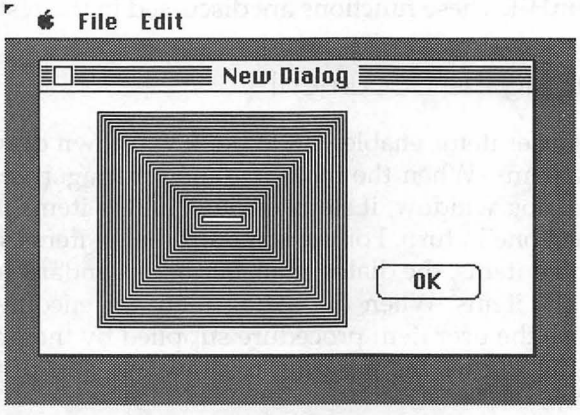


Figure 9-1. A Sample Dialog

▶ TSampDlg

TSampDlg is a class derived from **TModelessDoc**. The declaration of **TSampDlg** is shown as follows.

```
class TSampDlg : public TModelessDoc{

public:
   TSampDlg(OSType creator,SFReply * theReply);
   virtual Boolean InitDoc(void);

protected:
   virtual void DoItemHit(DialogPtr theDialog,short theItem);

};
```

The constructor for **TSampDlg** has no purpose other than to pass its arguments on to the parent class, as illustrated by the following definition.

```
TSampDlg::TSampDlg(OSType creator,SFReply * theReply):
   TModelessDoc(creator,theReply){
}
```

TSampDlg overrides only two other member functions, **InitDoc** and **DoItemHit**. These functions are discussed in the rest of this chapter.

▶ Initializing the Document

Dialog user items enable you to hook your own drawing procedures to dialog items. When the toolbox dialog manager gets an update event for a dialog window, it steps through all the items in the dialog, drawing each one in turn. For the standard dialog items such as buttons and static text items, the dialog manager uses standard toolbox functions to draw the items. When the dialog manager encounters a user item, it employs the user item procedure supplied by the program to draw the item; this allows the program to draw any sort of image to represent the item.

The biggest danger in this scheme is that the user item and its drawing procedure must be hooked up at runtime, before the first update event for the dialog. If the dialog manager tries to draw the user item before you have specified the drawing procedure, it will try to use a nil procedure pointer and will crash.

TSampDlg uses the **InitDoc** member function to set up the procedure pointer for the user item in the sample dialog. Remember that **InitDoc** is called after the window has been created but before it has been made visible. This is the perfect time to hook the user item to its drawing procedure. The toolbox functions GetDItem and SetDItem are used to first gather information about the user item and then attach a pointer to the user item drawing procedure to the user item. We define a constant, iUserItem, to specify the item number of the user item.

By the Way ▶ **Dialog item numbers are assigned in the order in which the items are defined in the 'DITL' resource for the dialog.**

The code for **InitDoc** is shown as follows. Notice that it calls the inherited version of **InitDoc** first to take care of any initialization for its parent classes. It then calls GetDItem to fill in some variables with information about the specified dialog item. It then calls SetDItem, supplying a pointer to the user item procedure as the fourth argument.

The function prototype for the user item procedure is declared at the beginning of **InitDoc** so that it can be used as a procedure pointer when calling SetDItem. The prototype tells the compiler that somewhere else there is a function named UserItemProc that uses the Pascal calling method and has two arguments. This function is actually defined later in the same source file, although it could alternately be defined in another source file.

```
Boolean TSampDlg::InitDoc(void){

   // install user item proc
   void pascal UserItemProc(WindowPtr theWindow,short theItem);

   Rect theRect;
   short theType;
   Handle theItem;

   if(TModelessDoc::InitDoc()){
      GetDItem((DialogPtr)fDocWindow,
         iUserItem,
         &theType,
         &theItem,
         &theRect);
      SetDItem((DialogPtr)fDocWindow,
         iUserItem,
         theType,
         (Handle)UserItemProc,
         &theRect);

      return true;
   }

   else
      return false;

}
```

The fourth argument to the toolbox function SetDItem is supposed to be a handle. We must therefore typecast our procedure pointer, UserItemProc, to be a handle before the compiler will accept it. This typecast is needed to correctly interface with the toolbox call, but it causes the following compiler warning.

```
File "ModelessApp.cp"; line 115 # warning:  pointer to function
cast to pointer to non function
```

You can ignore this warning. The typecast is necessary because SetDItem is used for many purposes and for most, the fourth argument is a handle. However, when hooking up a user item to a user item procedure, SetDItem needs a function pointer instead of a handle.

Key Point ▶

Because the toolbox interface is defined in Pascal, it is not possible to declare two versions of SetDItem. If the toolbox were defined in C++, this problem could be avoided by overloading SetDItem so that one version had a handle as the fourth argument and the other version had a function pointer.

▶ Dialog User Item

The arguments for the user item procedure are defined in the "Dialog Manager" section of *Inside Macintosh*. It must use the Pascal calling convention and take two arguments — a window pointer and an item number. The sample user item procedure uses the dialog pointer and item number to call GetDItem to get the rectangle that encloses the user item. It then draws ever smaller rectangles to fill the item area. The code for the user item procedure is shown as follows. Notice that it is not a member function, since a member function would have an extra, implicit argument pointing to the object, as discussed in Chapter 3.

```
void pascal UserItemProc(WindowPtr theWindow, short theItem){
   Rect r;
   short theType;
   Handle theItemH;
   short width;

   GetDItem((DialogPtr)theWindow,
      theItem,
      &theType,
      &theItemH,
      &r);
   width = (r.right - r.left);
   EraseRect(&r);
   FrameRect(&r);
   for(short i = width / 2; i > 0; i -= 2){
      InsetRect(&r,2,2);
      FrameRect(&r);
   }
}
```

▶ Handling User Input

As mentioned in the discussion of the base class, you must override the **DoItemHit** member function to respond to user input. This function is called whenever there is a mouse click or key press in an enabled dialog item. The arguments are a DialogPtr and the item number. You can use the item number to determine what action to take. The DialogPtr can be useful if you want to set the state of radio buttons or check boxes. The code for the sample **DoItemHit** is shown as follows. It simply beeps when the OK button is clicked.

```
void TSampDlg::DoItemHit(DialogPtr/*theDialog*/,short theItem){
   if(theItem == iOK)
      SysBeep(1);
}
```

▶ TModelessApp

You must define the minimal derived application class to tie everything together. Like other application classes that you have seen in previous chapters, this class must override the **MakeDoc** member function to create the desired type of document. The declaration of the new application class and the definition of its **MakeDoc** member function are listed as follows.

```
class TModelessApp : public TApp{
   virtual TDoc * MakeDoc(SFReply * reply = (SFReply *) nil);
};
TDoc * TModelessApp::MakeDoc(SFReply * reply){
   return new TSampDlg(GetCreator(),reply);
}
```

▶ Application Resources

The derived dialog document class that is defined for this application needs to have its own 'DLOG' and 'DITL' resources. If you define them both with the ID number rGenericDialog (defined as 1000 in TModelessDoc.h), you will not have to override **GetWinID**. Appendix B contains the definition for the sample 'DLOG' and 'DITL' resources in the file ModelessApp.rsrc.r.

The file ModelessApp.r lists all the resource files necessary to create the application resources. As before, they are listed in a particular order so that the resources in the last files will replace equivalent resources in the first files. The listing for ModelessApp.r is as follows.

```
// ModelessApp.r
// gather the resources for simple program
// that uses TModelessDoc's

include "TApp.rsrc";
include "TDoc.rsrc";
include "ModelessApp.rsrc";
```

▶ Making the Sample Application

The makefile for this sample application is similar to those in previous chapters. Since **TSampDlg** is derived from **TModelessDoc**, you need to include the directory for **TModelessDoc** in the search path for C++ and the linker. Just as in Chapter 6, where we defined a path name for the directory containing **TApp** and **TDoc**, you can define an additional directory path for the modeless dialog class, as follows.

```
# tell cplus and rez where to find included files for TApp, TDoc,
# and TModelessDoc
AppObjectDir = ::App-Doc:
ModelessObjDir = ::ModelessDoc:
```

Next, tell C++ to search that directory when looking for include files by adding the ModelessDoc directory to CPlusOptions, as follows.

```
# options for C++, where to look for include files
CPlusOptions = {SymOpts} -i "{AppObjectDir}" -i "{ModelessObjDir}"
```

Then you add the object file for **TModelessDoc** to the list of object files, and you include the object file for the sample application ModelessApp, shown as follows.

```
Objects = ∂
   "{AppObjectDir}"TApp.cp.o ∂
   "{AppObjectDir}"TDoc.cp.o ∂
   "{ModelessObjDir}"TModelessDoc.cp.o ∂
   ModelessApp.cp.o
```

Because ModelessApp uses the **TModelessDoc** class, you must include a dependency rule for the new document class so that it will be rebuilt correctly when any of its dependent files are changed. TModelessDoc.cp.o is dependent on TModelessDoc.cp and TModelessDoc.h and also on TDoc.h, as shown by the following dependency rule.

```
# dependency rules for TModelessDoc
"{ModelessObjDir}"TModelessDoc.cp.o ƒ ∂
          "{ModelessObjDir}"TModelessDoc.cp ∂
          "{ModelessObjDir}"TModelessDoc.h ∂
          "{AppObjectDir}"TDoc.h
```

These are the major changes you must make to the makefile. They deal largely with the problem of how to include header files and object modules from an additional directory. See the complete text of ModelessApp.make in Appendix B for the finer details of putting the application together.

▶ Summary

TModelessDoc is an easy modification of **TDoc** that enables you to create and manage documents that utilize features of the toolbox Dialog Manager. It gives you the ability to use buttons, text boxes, and user items in document windows.

This chapter discussed the **TModelessDoc** class as a base class and then developed a derived class that could be used in an application. The derived class showed how to use standard dialog items like a button and also how to include user items to give your dialogs a customized look.

The ability to adapt the original **TDoc** class to use dialogs instead of regular windows shows the flexibility of a good base class. If these changes had been more difficult to make, it would have indicated that the design of the original class was at fault.

TModelessDoc was derived from **TDoc** to serve as a new base class for other derived document classes. It will not be developed any further in this book. The next chapter also develops a new document class derived from **TDoc**. That new class will be used as the base class for the document classes derived in Chapters 11-13.

10 ▶ TScrollDoc: The Generalized Scrolling Document Class

This chapter describes the **TScrollDoc** class. **TScrollDoc** is derived from the **TDoc** class presented in Chapter 5. **TScrollDoc** supports scroll bars in a window very generally, so it can be used as the basis for many types of documents that need to scroll images that are larger than the window itself.

Supporting scroll bars and scrolling images is one of the hardest parts of Macintosh programming. It involves defining functions that are called from the toolbox and requires well-coordinated error checking to keep the image and the scroll bars in sync. **TScrollDoc** encapsulates fundamental concepts about scrolling and the QuickDraw coordinate system, and it also implements most of the trickiest scrolling routines that are needed to support scroll bars.

The scrolling member functions contained in this class were originally developed as part of the TextEdit document class described in Chapter 12. But in the course of creating that class, I decided it would be better to separate the scrolling member functions from the text handling member functions. The result was a scrolling class that supported the TextEdit class and could also be used to support a document class that displayed 'PICT' images, as shown in Chapter 11. My main motivation for isolating and generalizing the scrolling code was that I never wanted to have to write scrolling code again.

One of the goals of **TScrollDoc** is to hide most of the complexity of scrolling. Just because a class has a complex implementation is no reason for it to have a complicated external interface. You will find the **TScrollDoc** class easy to use. The member functions that you have to

override when creating your derived scrolling classes are well isolated and typically trivial to implement. You should not have to rewrite the hard parts.

This chapter discusses general scrolling concepts and various member functions of **TScrollDoc**. The complete code for **TScrollDoc** is listed in Appendix B.

▶ Overview of Scrolling

Scrolling is a way to display an image in a window when the image is larger than the window. Think of a window as a stationary piece of cardboard with a rectangular hole cut in its center. Think of the image as a large piece of paper that sits behind the cardboard window. You can see only that portion of the image that sits behind the hole in the cardboard. To see other parts of the image, you can slide the paper beneath the cardboard so that another portion of the image comes into view through the hole. Scrolling is analogous to moving the paper beneath the cardboard.

Let's translate cardboard and paper into QuickDraw GrafPorts and coordinate systems. A window displays images by drawing into a GrafPort. The interior of the window is analogous to the hole in the cardboard. By default, the coordinate system for a window's GrafPort has its origin in the upper left corner of the interior of the window, as shown in Figure 10-1. Images that are drawn in the window are positioned according to the coordinate system of the window's GrafPort. The large 'PICT' image in Figure 10-1 is drawn into the rectangle (0,0,450,400) (top left, bottom right).

Now consider the fact that the origin of a window's GrafPort does not always have to be in the upper left corner of the window. It is possible to specify the coordinates of the upper left corner of the window with the toolbox function SetOrigin. Figure 10-2 shows how a large 'PICT' image would be displayed if the window's upper left corner was set to the coordinate (300,145) (vertical, horizontal). The 'PICT' is still drawn into the rectangle (0,0,450,400), but this rectangle is now in a different place relative to the interior of the window. The coordinate system of the window's GrafPort has been moved up and to the left, just like the piece of paper beneath the cardboard.

TScrollDoc uses coordinate system offsets to implement scrolling. The class keeps track of the current coordinates of the top left corner of the window and calls SetOrigin to slide the window's GrafPort to match these offsets. User clicks on the scroll bars cause the class to adjust the coordinate offsets and redraw the image. This scrolling strategy turns out to be general enough to apply to all sorts of image data.

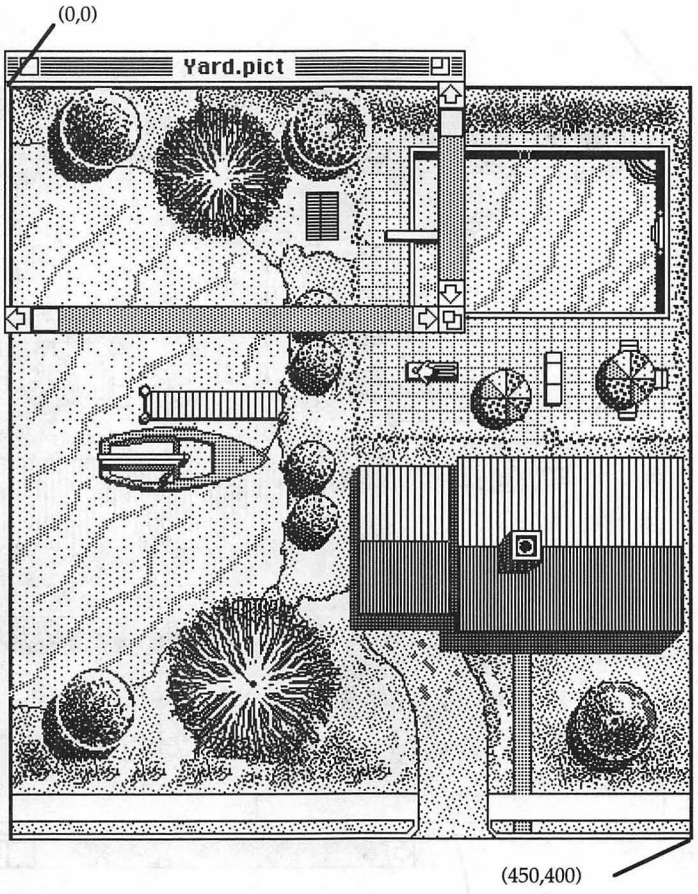


Figure 10-1. A Window and a Large Image

▶ Scroll Bars

Scroll bars are a convenient way for a user to control scrolling in a window, but they are not absolutely necessary. Remember, for example, that the original MacPaint program had no scroll bars, yet the user could view a selected portion of a full-page document in a small, fixed-size window. MacPaint let the user move the image with a "hand" cursor instead of scroll bars. The hand cursor helped the user visualize, in a very direct way, the metaphor of sliding a piece of paper beneath the window until the desired portion came into view.

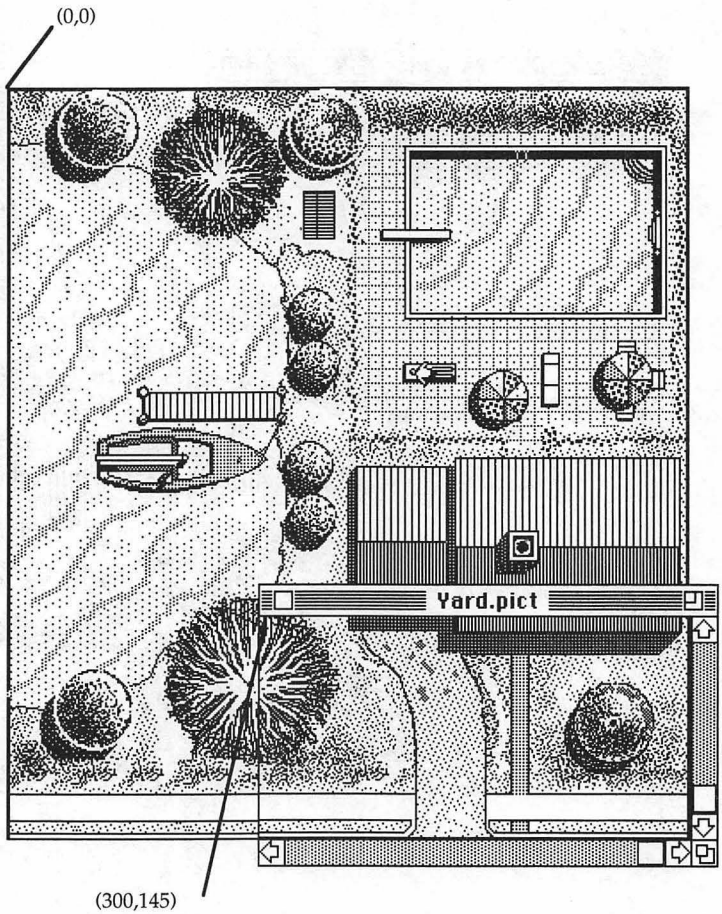


Figure 10-2. An Offset Window and a Large Image

Scroll bars perform the same function as the hand cursor. They tell the program how far to slide the image. Scroll bars contain several distinct parts, as shown in Figure 10-3.

The program can determine what part of a scroll bar was clicked by a user by calling the toolbox function FindControl. The program can then scroll the image by the appropriate amount, depending on which part of the scroll bar was clicked. Clicks on the up or down arrows typically cause the image to scroll by a small amount, such as one line of text in a text window. Clicks on the page up or page down areas normally result in scrolling equal to the width or height of the window. Dragging the thumb permits the user to scroll to an arbitrary location in the image.

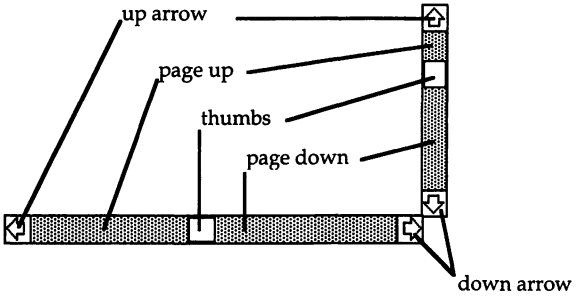


Figure 10-3. Scroll Bar Parts

Scroll bars also maintain three state variables: minimum value, maximum value, and current value. The thumb position indicates the current scroll bar value relative to the minimum and maximum values. **TScrollDoc** always sets the minimum value of a scroll bar to 0. It sets the maximum value of a scroll bar to the vertical or horizontal dimension, in pixels, of the image, minus the height or width of the window. Looking back at Figure 10-1, the image is 450 pixels tall and the window 150 pixels tall, so the maximum value of the vertical scroll bar would be 300. **TScrollDoc** uses the current value of the scroll bar to hold the current vertical or horizontal offset in the GrafPort's coordinate system. In Figure 10-2, the current value of the vertical scroll bar is 300 and the horizontal scroll bar is 145.

▶ Members

TScrollDoc is derived from **TDoc**, so it contains all the members of **TDoc**. It also defines four additional members that are specific to its scrolling features. Two of the members contain ControlHandles that refer to the vertical and horizontal scroll bars of the document window. The other two members contain integers that keep track of the amount of horizontal and vertical offset in the coordinate system that has occurred as a result of scrolling, as described in the previous section. The declaration of **TScrollDoc**'s members is shown as follows.

```
class TScrollDoc : public TDoc {

protected:
   ControlHandle fHorizScrollBar;
   ControlHandle fVertScrollBar ;
   short fVOffset;
   short fHOffset;
```

All of these members are declared as protected so that they can only be accessed by member functions of **TScrollDoc** and its derived classes. We do provide public member functions, however, to get the value of the scroll bar ControlHandles since they need to be available to functions that are not class member functions (this is explained in the "Scroll Action Procedure" section of this chapter). The two member functions to return the ControlHandles are defined in the declaration of **TScrollDoc** shown here.

```
public:

ControlHandle GetVScroll(void){return fVertScrollBar;}
ControlHandle GetHScroll(void){return fHorizScrollBar;}
```

▶ Static Member and Static Member Function

In addition to the four members described in the previous section, **TScrollDoc** defines a static member that is a pointer to a **TScrollDoc** object. This member, *fCurrScrollDoc*, is declared in the declaration of **TScrollDoc** in the file TScrollDoc.h as shown here. Notice that *fCurrScrollDoc* is protected so that it can't be accessed from outside the class.

```
class TScrollDoc : public TDoc {
protected:

    ControlHandle fHorizScrollBar;
    ControlHandle fVertScrollBar;
    short fVOffset;
    short fHOffset;
    static TScrollDoc *fCurrScrollDoc;
```

It is defined and initialized in TScrollDoc.cp with the following statement.

```
TScrollDoc *TScrollDoc::fCurrScrollDoc = nil;
```

This definition statement causes the compiler to allocate space for the static member when TScrollDoc.cp is compiled. As explained in Chapter 3, static members are essentially global variables that obey the access rules of class members.

fCurrScrollDoc is initialized to nil in its definition statement. Subsequently, it is set to the value of the currently active document when-

ever a new document becomes active (see the description of the **Activate** member function later in this chapter). All scrolling document objects share a single instance of *fCurrScrollDoc*; thus its value always points to the current scrolling document.

TScrollDoc also defines a static member function, **GetCurrScrollDoc**, to allow functions from outside the class to get the value of *fCurrScrollDoc*, which is protected. The key attribute of a static member function is that you don't need to have an object reference to call the static member function. To call a regular member function, you must have a specific object, or a pointer to an object, of that class. With a static member function, however, you can call the function simply by preceding its name with the name of the class and two colons, as follows.

```
// call the static member function
TScrollDoc * theCurrScrollDoc = TScroll::GetCurrScrollDoc();
```

GetCurrScrollDoc is defined in the declaration of **TScrollDoc** as follows. Notice the static keyword before the declaration.

```
static  TScrollDoc *GetCurrScrollDoc(void){return fCurrScrollDoc;}
```

Because **GetCurrScrollDoc** is a static member function, it is possible for functions that do not have an object reference to call it and gain access to the current scrolling document. This is necessary because certain parts of the scrolling mechanism, explained in the "ScrollAction Procedure" section of this chapter, cannot be written as object member functions and cannot gain access to the scrolling member functions without first getting a pointer to the current scrolling document from **GetCurrScrollDoc**.

The static member and static member function enable the parts of the scrolling code that are not member functions to utilize the member functions of the scrolling class. This could also have been accomplished by using a global variable to hold a pointer to the current scrolling document, but a static member can be protected while a global can be altered by any piece of code.

Key Point ▶

Static members and static member functions are often preferable to global variables because static members can be protected from unauthorized modification.

▶ Constructor and Destructor

The constructor for **TScrollDoc** merely initializes the members that are unique to this class and passes its two arguments on to its parent class, **TDoc**. The mechanism for passing arguments to parent constructors is described in more detail in Chapter 8. The code for **TScrollDoc**'s constructor is shown as follows.

```
TScrollDoc::TScrollDoc(OSType theCreator,SFReply * SFInfo):
   TDoc(theCreator,SFInfo){
   fHorizScrollBar = nil;
   fVertScrollBar = nil;
   fVOffset = 0;
   fHOffset = 0;
}
```

The destructor is defined as follows as an empty member function. Since **TScrollDoc** will be used as a base class for other document classes, the destructor is virtual to ensure that the destructor for the derived class will be called, as explained in Chapter 3.

```
// virtual destructor so that derived destructors will be called
virtual ~TScrollDoc(void){}
```

▶ Initialization

TScrollDoc overrides the **InitDoc** member function to create the two scroll bars for the document window. It calls the **InitDoc** member function for its parent class first, to make sure that the default initialization is successful before it proceeds. The application class calls **InitDoc** for all new documents. If the new document is a member of the **TScrollDoc** class, then the version of **InitDoc** shown as follows will be executed.

```
Boolean TScrollDoc::InitDoc(void){
   if( ! TDoc::InitDoc())
      return false;
   if(fDocWindow != nil){
      SetPort(fDocWindow);
      fHorizScrollBar = GetNewControl(rHScroll,fDocWindow);
      fVertScrollBar = GetNewControl(rVScroll,fDocWindow);
      SizeScrollBars();
      SynchScrollBars();
   }
   return ((fHorizScrollBar != nil) && .(fVertScrollBar != nil));
}
```

InitDoc calls the toolbox function GetNewControl to load in the resources for the scroll bars. It then calls two member functions, **SizeScrollBars** and **SynchScrollBars** to fit the scroll bars to the window. These two member functions are explained in a later section of this chapter.

InitDoc returns true if both scroll bars were created successfully. One problem with this class is that there is no way to specify whether or not you want both scroll bars. What if you wanted a window without the horizontal scroll bar? The problem is that the original design of **TDoc** did not deal with scroll bars, so there are no arguments to any of its initialization member functions to specify how many scroll bars to install. This problem is perpetuated because **TApp** treats all documents like members of the **TDoc** class. You could solve this by adding parameters to the **InitDoc** member function to control scroll bar creation, but you would also have to change the **TApp** class to pass these parameters when it calls **InitDoc**.

You might want to make these changes. Essentially, you want to create a derived application class that uses **TScrollDoc** instead of **TDoc** as its base document class.

▶ Geometry

TScrollDoc defines several new member functions that are concerned with the dimensions of the image and the viewing area of the window. Some other member functions specify how far to scroll in response to various scrolling actions. These are the member functions that you will typically override when deriving a class from **TScrollDoc**. Other scrolling member functions in the class use these member functions to get the specific characteristics of your image so that it can be scrolled.

The **GetContentRect** member function fills in a rectangle that defines the area of the document window where the image is displayed. By default, we take the portRect of the window and subtract the scroll bars, leaving just the interior of the window. **GetContentRect** is a critical member function since it is used by many other member functions in the class. It is not necessary to override this member function if your image occupies all of the window except the scroll bars, but you will want to override it if your image takes up more or less space than that. For example, the PictView document class in Chapter 11 does not override **GetContentRect**, but the TextEdit class in Chapter 12 does override it since the text window has a narrow top and left margin where no text is ever displayed. You could also override **GetContentRect** if your derived class had a palette area, like MacDraw, within each window. The definition of **GetContentRect** is shown as follows.

```
void TScrollDoc::GetContentRect(Rect & r){

   // how big is the content area of the window, discounting the
   // scroll bars
   r = fDocWindow->portRect;
   if(fVertScrollBar != nil)
      r.right -= kScrollBarPos;
   if(fHorizScrollBar != nil)
      r.bottom -= kScrollBarPos;
}
```

The next two member functions, **GetVertSize** and **GetHorizSize**, return the vertical and horizontal dimensions, in pixels, of the document image. You must override these member functions since they return 0 by default. For example, the PictView document class in Chapter 11 returns the width and height of the picture's bounding rectangle. The default definitions of **GetVertSize** and **GetHorizSize** are shown as follows.

```
// these two member functions must be overridden
virtual short GetVertSize(void){return 0;}
virtual short GetHorizSize(void){return 0;}
```

Finally, four member functions are defined to tell the scrolling routines how far to scroll when the user clicks on the up or down scroll arrows or the page scroll areas of the scroll bar.

You must override the first two of these member functions. They specify how far to scroll in response to a click on the up or down arrows of the scroll bar, which is known as a line scroll. The member function **GetVertLineScrollAmount** gives the amount to scroll vertically and **GetHorizLineScrollAmount** gives the amount for horizontal scrolling. By default these member functions return 0. The TextEdit document class overrides these member functions to return the number of pixels in one line of text. The default definitions are shown as follows.

```
virtual short GetVertLineScrollAmount(void){return 0;}
virtual short GetHorizLineScrollAmount(void){return 0;}
```

The other two member functions that specify how much to scroll are **GetVertPageScrollAmount** and **GetHorizPageScrollAmount**. These two member functions return the number of pixels to scroll, either vertically or horizontally, in response to a click on the page scroll area of the scroll bar. By default they return the height or width of the content area of the window, minus a small constant value to give some overlap. Essentially, they scroll the image one full window at a time. Since

most Macintosh users have come to expect this behavior, you will probably not need to override these member functions. The definitions of **GetVertPageScrollAmount** and **GetHorizPageScrollAmount** are shown as follows.

```
short TScrollDoc::GetVertPageScrollAmount(void){
   Rect r;
   GetContentRect(r);
   return r.bottom - r.top - kScrollOverlap;
}

short TScrollDoc::GetHorizPageScrollAmount(void){
   Rect r;
   GetContentRect(r);
   return r.right - r.left - kScrollOverlap;
}
```

▶ Coordinate Offset and Focus

As mentioned earlier in this chapter, the key to the scrolling techniques of this class is changing the coordinate system of the document window's GrafPort. The contents of the window are drawn within a coordinate system that is offset to reflect how far the image has been scrolled. Whenever we are about to draw the contents of a window, we need to make sure that the origin of the window is set to reflect this coordinate offset. The member function **FocusOnContent** uses the *fVOffset* and *fHOffset* members (of the document) to set the coordinate origin with the toolbox function SetOrigin. It also sets the clip region of the window to encompass just the content area of the window, excluding the scroll bars. This ensures that drawing the document's image will not infringe upon the scroll bars. The code for **FocusOnContent** is shown as follows.

```
void TScrollDoc::FocusOnContent(){
   SetPort(fDocWindow);
   SetOrigin(fHOffset,fVOffset);
   Rect r;
   GetContentRect(r);
   ClipRect(&r);
}
```

TScrollDoc also defines a member function, **FocusOnWindow**, that we call whenever we are about to draw some of the structural parts of the window, such as the scroll bars or the grow box. This member function sets the origin of the window back to 0,0 and opens up the clip region to include the scroll bar areas. **FocusOnWindow** is shown as follows.

```
void TScrollDoc::FocusOnWindow(){
   SetPort(fDocWindow);
   SetOrigin(0,0);
   Rect r = fDocWindow->portRect;
   ClipRect(&r);
}
```

These two member functions allow **TScrollDoc** to maintain two distinct coordinate systems for the document window. When drawing the image contents, it offsets the coordinates in accordance with how far the image has been scrolled. When drawing the structural parts of the window, it reverts to a fixed coordinate system with no offset. Other member functions that draw in the window are responsible for calling **FocusOnContent** or **FocusOnWindow**, as appropriate, to make sure that both the coordinate system is aligned with the part of the window that they want to draw in and the clip region is set properly.

▶ Managing the Scroll Bars

We define three member functions to manage the scroll bars. The first member function, **SizeScrollBars**, is called when the document window is first created and whenever it is resized thereafter. **SizeScrollBars** first calls **FocusOnWindow** to reset the coordinate system origin to 0,0. Then it uses the portRect of the window to calculate the position and size of the horizontal and vertical scroll bars. Notice that after a scroll bar is drawn in its new position, we call the toolbox function ValidRect so that the scroll bar won't be needlessly drawn again as part of an update event. The code for **SizeScrollBars** is shown as follows. You will probably never have to override this member function.

```
void TScrollDoc::SizeScrollBars(void){

   if(fDocWindow != nil){
      FocusOnWindow();
      Rect r = fDocWindow->portRect;

      if(fVertScrollBar != nil){
         HideControl(fVertScrollBar);
         SizeControl(fVertScrollBar,
               kScrollBarWidth,
               (r.bottom - r.top - kScrollBarPos) + 2);

         MoveControl(fVertScrollBar,
               r.right - kScrollBarPos,
               -1);
         ShowControl(fVertScrollBar);
         ValidRect(&(**fVertScrollBar).contrlRect);
      }
      if(fHorizScrollBar != nil){
         HideControl(fHorizScrollBar);
         SizeControl(fHorizScrollBar,
               (r.right - r.left - kScrollBarPos) + 2,
               kScrollBarWidth);

         MoveControl(fHorizScrollBar,
               -1,
               r.bottom - r.top - kScrollBarPos);
         ShowControl(fHorizScrollBar);
         ValidRect(&(**fHorizScrollBar).contrlRect);
      }
   }
}
```

The next member function is **AdjustScrollBars**. It is responsible for adjusting the minimum and maximum values of the scroll bars. Remember from our previous discussion that the maximum value of a scroll bar in a **TScrollDoc** document is equal to the width or height of the image minus the width or height of the content area of the window. Thus, any time the size of the window changes or the dimensions of the image change we must adjust the scroll bars. **AdjustScrollBars** calculates the maximum value of the horizontal and vertical scroll bars by subtracting the dimensions of the content rectangle of the window from the image dimensions returned by the member functions **GetVertSize** and **GetHorizSize**, as shown here in the first part of **AdjustScrollBars**.

```
void TScrollDoc::AdjustScrollBars(void){

   // don't activate the scroll bars until
   // the data extends beyond the window boundaries
   // If currentCtlValue is greater than new ctlmax,
   // scroll image to bring it in line
   Rect r ;
   GetContentRect(r);
   short dh,dv;
   dh = dv = 0;
   short currentValue;
   short newMax;

   // now ask the document how big its image is
   // first for the vertical dimension
   if(fVertScrollBar != nil){
      currentValue = GetCtlValue(fVertScrollBar);
      newMax = GetVertSize() - (r.bottom - r.top);
      if(newMax < 0)
         newMax = 0;
      if(currentValue > newMax)
         dv = currentValue - newMax;
      SetCtlMax(fVertScrollBar,newMax);
   }

   // and then the horizontal dimension
   if(fHorizScrollBar != nil){
      currentValue = GetCtlValue(fHorizScrollBar);
      newMax = GetHorizSize() - (r.right - r.left);
      if(newMax < 0)
         newMax = 0;
      if(currentValue > newMax)
         dh = currentValue - newMax;
      SetCtlMax(fHorizScrollBar,newMax);
   }
```

AdjustScrollBars then compares these new maximum values to the current control values of the scroll bars. If the current scroll bar value is greater than the new maximum, we must scroll the image back so that the image is in synch with the scroll bars. This happens when the window is made larger or the image becomes smaller. We don't actually want the image to be redrawn by the scrolling operation at this point. We want to affect only the underlying data structures and coordinate offset values, so we shut the clip region down to an empty rectangle be-

fore calling **ScrollContents**. After scrolling, we reset the clip region to its former setting. We invalidate the entire window so that the image will be redrawn in its proper place by an update event. The second half of **AdjustScrollBars** is shown as follows.

```
    // adjust the position of the image if the window
    // has gotten bigger than the image.
    if(dh | dv){
        FocusOnContent();

        // invalidate the whole content area
        GetContentRect(&r);
        InvalRect(&r);

        // shut the clip region down to zero
        // so that the scrolling won't actually
        // draw in the window; wait for update instead
        RgnHandle oldClip = NewRgn();
        GetClip(oldClip);
        SetRect(&r,0,0,0,0);
        ClipRect(&r);
        ScrollContents(dh,dv);
        // now reset the clip region
        SetClip(oldClip);
        DisposeRgn(oldClip);
    }
}
```

You will probably never need to override **AdjustScrollBars**.

The final member function for managing the scroll bars is **SetScrollBarValues**. This member function calculates the current value for the scroll bar indicator thumbs. The thumbs indicate how far the image has been scrolled. In the default case, the value of the horizontal scroll bar is equal to the horizontal offset value in the member *fHOffset* and the vertical scroll bar is equal to *fVOffset*. **SetScrollBarValues** calls the toolbox function SetCtlValue to set the scroll bar value. It calls **FocusOnWindow** first, to make sure that the clip region includes the scroll bars so that any change in the thumb position will be drawn in the scroll bar. If you use the default scrolling member functions described later in this chapter, you will probably not have to override **SetScrollBarValues**. The TextEdit document class described in Chapter 12 uses different scrolling techniques and therefore must override **SetScrollBarValues**. The default definition for **SetScrollBarValues** is shown as follows.

```
void TScrollDoc::SetScrollBarValues(void){
   FocusOnWindow();
   if(fHorizScrollBar != nil)
      SetCtlValue(fHorizScrollBar,fHOffset);
   if(fVertScrollBar != nil)
      SetCtlValue(fVertScrollBar,fVOffset);
}
```

Since **AdjustScrollBars** and **SetScrollBarValues** are often called in tandem, **TScrollDoc** defines a utility member function, **SynchScrollBars**, to call them together, shown by the following code.

```
void TScrollDoc::SynchScrollBars(void){
   AdjustScrollBars();
   SetScrollBarValues();
}
```

▶ Handling Events

TScrollDoc is derived from **TDoc**, so it has all the same event handling member functions of the parent class. It overrides many of the event member functions to account for the scroll bars. The various event member functions are described in the sections that follow.

▶ Activation/Deactivation

When the document window is becoming active, the **Activate** member function must call the toolbox function ShowControl to make the scroll bars active. Before calling ShowControl, **Activate** calls the **FocusOnWindow** member function to make certain that the clip region of the window includes the scroll bars so that they will be redrawn to show their active status. **Activate** must also set the static member *fCurrScrollDoc* to "this" to refer to the document object that is becoming active. This static member allows functions that are not document member functions, such as the scroll action procedure described later in this chapter, to have access to **TScrollDoc** member functions for the currently active document. The **Activate** member function, which is an empty function in the **TDoc** class, is overridden in **TScrollDoc** as follows.

```
void TScrollDoc::Activate(void){
   FocusOnWindow();
   if(fVertScrollBar != nil)
      ShowControl(fVertScrollBar);
   if(fHorizScrollBar != nil)
      ShowControl(fHorizScrollBar);
   // set up static member so that scroll action proc
   // can access object member functions
   fCurrScrollDoc = this;
}
```

Likewise, when the document window is becoming inactive, the **Deactivate** member function calls HideControl to make the scroll bars inactive. Like the activation case, **Deactivate** focuses on the entire window so that the drawing necessary to deactivate the scroll bars will show up. The **Deactivate** member function is shown as follows.

```
void TScrollDoc::Deactivate(void){
   FocusOnWindow();
   if(fVertScrollBar != nil)
      HideControl(fVertScrollBar);
   if(fHorizScrollBar != nil)
      HideControl(fHorizScrollBar);
}
```

▶ Updates

TScrollDoc overrides the **DoTheUpdate** member function to include extra code to adjust the window focus and coordinate system and to draw the scroll bars with the toolbox function DrawControls. **DoTheUpdate** calls the **FocusOnContent** member function before calling the **Draw** member function to draw the contents of the window, and then it calls **FocusOnWindow** before drawing the scroll bars and grow box. The code for **DoTheUpdate** is shown as follows. As with the **TDoc** class, you will want to override the **Draw** member function to do the actual drawing in the window, and you will probably not need to override **DoTheUpdate**.

```
void TScrollDoc::DoTheUpdate(EventRecord * /*theEvent*/){

    if(fDocWindow != nil){
        FocusOnContent();
        BeginUpdate(fDocWindow);
```

```
        Rect r = (**(fDocWindow->visRgn)).rgnBBox;
        Draw(&r);
        FocusOnWindow();
        DrawControls(fDocWindow);
        DoDrawGrowIcon();
        EndUpdate(fDocWindow);
    }
}
```

▶ Mouse Clicks on Content

In the original **TDoc** class, it was enough to know that the mouse had been clicked on the content area of the window. In that class, the content area included the area occupied by the scroll bars in the **TScrollDoc** class. Now we need to have a more precise division of labor when responding to mouse clicks. **TScrollDoc** overrides the **DoContent** member function to determine if the mouse click is on the scroll bars or the content area of the window. It also defines two new member functions, **ContentClick** and **ScrollClick**, to handle the two possible cases. The overridden **DoContent** is shown here. It uses the **GetContentRect** member function to determine the boundaries between the content area and the scroll bars.

```
void TScrollDoc::DoContent(EventRecord* theEvent){

   FocusOnWindow();
   GlobalToLocal(&theEvent->where);
   Rect contents;
   GetContentRect(contents);
   if(PtInRect(theEvent->where,&contents))
      ContentClick(theEvent);
   else
      ScrollClick(theEvent);
}
```

ContentClick is defined as an empty member function that you will want to override if your document responds to clicks on the content area. The TextEdit document class in Chapter 12 overrides **ContentClick** to call the toolbox function TEClick.

ScrollClick is defined to call the toolbox function FindControl to determine on which part of the scroll bar the mouse was clicked. It uses

the part code returned from FindControl to branch to the appropriate scrolling member function. (The scrolling member functions are described in a later section of this chapter.) You will probably not need to override **ScrollClick**; any change in behavior that you want is more likely to be in the individual scrolling member functions than in **ScrollClick**. The definition of **ScrollClick** is shown as follows.

```
void TScrollDoc::ScrollClick(EventRecord *theEvent){
   ControlHandle whichControl;
   short part;
   FocusOnWindow();
   if(part = FindControl(theEvent>where,fDocWindow,&whichControl)){
      switch (part){
         case inThumb:
            DoThumbScroll(whichControl,theEvent->where);
            break;

         case inUpButton:
         case inDownButton:
            DoButtonScroll(whichControl,theEvent->where);
            break;

         case inPageUp:
         case inPageDown:
            DoPageScroll(whichControl,part);
            break;

      }
   }
}
```

▶ Zooming and Growing

The member functions for growing and zooming a window both call the equivalent member function for the parent class to do the default processing for the document window and then call additional member functions to resize and adjust the scroll bars. The definitions for **DoGrow** and **DoZoom** are shown as follows.

```
void TScrollDoc::DoGrow(EventRecord* theEvent){

   FocusOnWindow();
   // call the parent class
   TDoc::DoGrow(theEvent);
   SizeScrollBars();
   SynchScrollBars();
}

void TScrollDoc::DoZoom(short partCode){

   FocusOnWindow();
   // call the parent class
   TDoc::DoZoom(partCode);
   SizeScrollBars();
   SynchScrollBars();
}
```

Key Point ▶

Calling the parent member function first and then performing other tasks enables the derived member function to extend the functionality of the parent member function rather than replace it.

▶ Scrolling

The member functions that deal with scrolling and user input in the scroll bars are divided up so that you will typically override only one or two member functions related to the low-level mechanisms that are used to scroll your image. The rest of the member functions are written to be general enough so that they should not need to be overridden. The more general member functions are dependent on the more specific, low-level utility member functions, so that any changes in the utility functions are reflected in the general member functions. This is an area where a well-designed base class can save a lot of work when you are creating derived classes. Try to isolate the parts of your class that will change in utility member functions and then write the rest of your member functions so that they call those utility member functions. That way, the changes you make to the utility functions will show up in all member functions that use the utility functions.

▶ Scrolling Utility Functions

ScrollContents is the utility member function that actually scrolls the image within the window. It is responsible for changing the *fHOffset* and *fVOffset* members, moving the bits within the window and forcing the window to be redrawn as necessary to show parts of the image that have just scrolled into view. All other member functions that respond to user input in the scroll bars eventually call **ScrollContents** to actually scroll the image in the window. The default definition of **ScrollContents** uses the toolbox function ScrollRect to scroll the bits of the image by the specified vertical and horizontal amounts. ScrollRect fills in a region handle with a region representing the area that scrolls into view and thus needs to be redrawn. **ScrollContents** invalidates this region to add it to the window's update region and forces an immediate update by calling the **DoTheUpdate** member function.

After scrolling the image, **ScrollContents** changes *fVOffset* and *fHOffset* to reflect the distance that the image has scrolled. **ScrollContents** is defined as follows.

```
void TScrollDoc::ScrollContents(short dh,short dv){

   // determine the area to scroll
   Rect r;
   GetContentRect(r);

   // now scroll the image
   RgnHandle updateRgn = NewRgn();
   ScrollRect(&r,dh,dv,updateRgn);

   // keep track of how far off the origin we are
   fVOffset -= dv;
   fHOffset -= dh;

   // tell window to redraw uncovered content
   InvalRgn(updateRgn);

   // now force the update area to be drawn
   DoTheUpdate((EventRecord *)nil);

   // dispose of the region
   DisposeRgn(updateRgn);

}
```

The default definition of **ScrollContents** will be sufficient to support scrolling in many classes derived from **TScrollDoc**. For example, the 'PICT' viewer document class described in Chapter 11 does not change **ScrollContents**. For some derived classes, however, you may want to override this member function to enable a different scrolling technique. For instance, the TextEdit document class in Chapter 12 uses the toolbox function TEScroll instead of ScrollRect and does not use *fVOffset* and *fHOffset* to keep track of how much the image has scrolled, relying instead on the destination and viewing rectangles that the toolbox TextEdit routines maintain. If you decide to override **ScrollContents**, there is a good chance that you will also override **SetScrollBarValues**, since that function assumes that *fVOffset* and *fHOffset* are set correctly by **ScrollContents**. See the TextEdit class in Chapter 12 for an example of how to implement a custom scrolling technique.

The other scrolling utility function that is defined is **Scroll**. This member function will scroll the image either horizontally or vertically and make sure that the scroll bar values remain in synch with the image. One of the hardest things about using scroll bars is making certain that you don't scroll past the endpoints. That is, when the scroll bar thumb is all the way to the top of the scroll bar, you don't want to scroll any further when the user clicks the mouse on the up button. All of the endpoint checking code is isolated in **Scroll**.

Scroll can be used to perform general image scrolling and scroll bar updating. For example, it is called from the member functions that respond to mouse clicks on the scroll bars, but it can also be called from other parts of the program, such as when a user drags a text selection outside the window in a TextEdit document (see Chapter 12). **Scroll** generalizes scrolling so that it doesn't necessarily have to originate with some action in a scroll bar, yet it still ensures that the scroll bars are in synch with the image.

The first part of **Scroll**, shown as follows, saves the clip region of the window so that it can be reinstated later. It then does the endpoint checking to make sure that the specified scroll amount will not take the image beyond the minimum or maximum values of the scroll bar.

```
void TScrollDoc::Scroll(ControlHandle theControl,short change){

   // This routine changes the value of the scroll bar
   // and scrolls the contents.
   // It can be used for arbitrary scrolling,
   // either from scroll bar action procs
```

```
   // or while dragging the mouse outside window.
   // save current clip region
   RgnHandle oldClip = NewRgn();
   GetClip(oldClip);

   short diff = 0;
   short oldValue = GetCtlValue(theControl);
   short newValue = oldValue + change;

   // check for endpoint
   if (change < 0){
      short minValue = GetCtlMin(theControl);
      if(newValue < minValue)
      newValue = minValue;
   } else {
      short maxValue = GetCtlMax(theControl);
      // figure the new value and check for endpoint
      if(newValue > maxValue)
         newValue = maxValue;
   }
   diff = oldValue - newValue;
```

Next, **Scroll** calls **FocusOnContent** and **ScrollContents** to scroll the image within the window. It then calls **FocusOnWindow** and **SetScrollBarValues** to make certain that the scroll bar thumbs reflect the new position of the image. Finally, it resets the clip region for the window to its original setting. The second half of **Scroll** is shown as follows.

```
   // do the scrolling and set the new scroll bar values
   FocusOnContent();
   if(theControl == fHorizScrollBar)
     ScrollContents(diff,0);
   if(theControl == fVertScrollBar)
      ScrollContents(0,diff);

   FocusOnWindow();
   SetScrollBarValues();

   // restore old clip region
   SetClip(oldClip);
   DisposeRgn(oldClip);

}
```

You probably won't need to override **Scroll**, even if your derived class uses a different scrolling technique than **TScrollDoc** uses. It is more likely that you will override **ScrollContents** and **SetScrollBarValues** to implement the low-level details of your scrolling strategy. Since **Scroll** calls these member functions to do the actual scrolling, changes you make in the low-level functions will carry through to **Scroll**.

▶ Thumb Scroll

The **DoThumbScroll** member function is defined to respond to mouse clicks on the thumb of a scroll bar. It calls the toolbox function TrackControl to enable the user to move the thumb within the scroll bar. When the user releases the mouse button, TrackControl returns with the scroll bar value set to reflect the new position of the thumb. We call the toolbox function GetCtlValue to get the new value and compare it to the thumb position before we began tracking. We use the difference between the old and new thumb positions as the argument to **ScrollContents**, which actually scrolls the image to its new position. The code for **DoThumbScroll** is shown here.

```
void TScrollDoc::DoThumbScroll(ControlHandle theControl,
   Point localPt){

   short oldValue = GetCtlValue(theControl);
   short trackResult = TrackControl(theControl,localPt,nil);
   if(trackResult != 0){
      short newValue = GetCtlValue(theControl);
      short diff = oldValue - newValue;
      FocusOnContent();
      if(theControl == fHorizScrollBar)
         ScrollContents(diff,0);
      if(theControl == fVertScrollBar)
         ScrollContents(0,diff);
      FocusOnWindow();
   }
}
```

You will not need to override **DoThumbScroll** unless you want to have immediate feedback while the user is moving the thumb. With the default implementation, the image is not scrolled until the user releases the mouse button. It is possible to actually scroll the image while tracking the thumb movement by supplying an action procedure

pointer as the third argument to TrackControl. We supply nil for this argument, so there is no other activity while tracking the thumb. You might want to write an action procedure and override **DoThumbScroll** to supply the procedure pointer when calling TrackControl. (See the action procedure defined for up and down arrow scrolling in a later section of this chapter and look in the "Control Manager" chapter of *Inside Macintosh* for more information about action procedures.)

▶ Page Scroll

DoPageScroll is a member function that is called when the user presses the mouse button on the page up or page down areas of a scroll bar of a **TScrollDoc** document. It calls **GetVertPageScrollAmount** or **GetHorizPageScrollAmount** to determine how far to scroll, and then it repeatedly calls the **Scroll** member function for as long as the user clicks the mouse button on that area of the scroll bar. **Scroll** takes care of calling **ScrollContents** to scroll the image and **SetScrollBarValues** to update the thumb position so that the user gets visual feedback as the scrolling operation proceeds. You will probably have no reason to override **DoPageScroll**. Its definition is shown as follows.

```
void TScrollDoc::DoPageScroll(ControlHandle theControl,short part){

   short scrollAmount;
   Point thePt;
   short currentPart;

   if((theControl == fVertScrollBar))
      scrollAmount = GetVertPageScrollAmount();
   else
      scrollAmount = GetHorizPageScrollAmount();

   // repeat as long as user holds down mouse button
   do {
      GetMouse(&thePt);
      currentPart = TestControl(theControl,thePt);
      if(currentPart == part){
         if(currentPart == inPageUp)
            Scroll(theControl,-scrollAmount);
         if(currentPart == inPageDown)
            Scroll(theControl,scrollAmount);
      }
   }while(Button());
}
```

▶ Button Scroll

Finally, the **DoButtonScroll** member function is called when the user clicks the mouse button on an up or down arrow of a scroll bar. Its sole task is to call the toolbox function TrackControl, supplying a pointer to a scroll action procedure where most of the scrolling will take place. The scroll action procedure is described in the next section. It is unlikely that you will need to override the default definition of **DoButtonScroll**, which is shown as follows.

```
void TScrollDoc::DoButtonScroll(ControlHandle theControl,
                                Point localPt){
   // declare the action procedure
   pascal void ActionProc(ControlHandle theControl,short partCode);
   short result = TrackControl(theControl,
                               localPt,
                               (ProcPtr)ActionProc);
}
```

▶ Scroll Action Procedure

One of the key elements of supporting scroll bars is the scroll action procedure. This function is called repeatedly by the toolbox function TrackControl while the user clicks the mouse button on the up or down arrows of a scroll bar. The scroll action procedure can scroll the image and change the scroll bar value. The scroll action procedure must be defined as a function with two arguments. It must also be defined with the pascal keyword so that it will obey the calling conventions like a function written in Pascal, since that is what the toolbox expects. The two arguments are a handle to the scroll bar on which the mouse is clicked and a part code indicating where the mouse is at the current time. The part argument allows you to stop scrolling when the user moves the mouse off of the up or down arrow and resume scrolling when it moves back on.

The scroll action procedure cannot be implemented as a class member function because all member functions have a hidden parameter, "this," that is a pointer to the object itself. Since the action procedure is called from the toolbox, there is no way to get this hidden argument passed to an action procedure written as a class member function. Thus, we have to define the scroll action procedure as a simple function, shown in the following code.

```
pascal void ActionProc(ControlHandle theControl,short partCode){
   TScrollDoc * theCurrScrollDoc = TScrollDoc::GetCurrScrollDoc();
   short scrollAmount = 0;
   if(theControl == theCurrScrollDoc->GetVScroll())
      scrollAmount = theCurrScrollDoc->GetVertLineScrollAmount();
   if(theControl == theCurrScrollDoc->GetHScroll())
      scrollAmount = theCurrScrollDoc->GetHorizLineScrollAmount();

   if(partCode == inUpButton)
      theCurrScrollDoc->Scroll(theControl,-scrollAmount);
   if(partCode == inDownButton)
      theCurrScrollDoc->Scroll(theControl,scrollAmount);
}
```

The action procedure uses the static member function **GetCurrScrollDoc** to retrieve the *fCurrScrollDoc* static member to call member functions for the current scrolling document object. Remember that *fCurrScrollDoc* is set to point to the currently active scroll document by the **Activate** member function. The action procedure examines the control handle passed to it from the toolbox to see if it is the vertical or horizontal scroll bar for our document. It then asks the document how far to scroll by calling either **GetHorizLineScrollAmount** or **GetVertLineScrollAmount**. The action procedure then looks at the *partCode* argument to decide whether to scroll with a positive or negative scroll amount. It calls the **Scroll** member function to actually scroll the image and update the scroll bar values. Because the action procedure calls TScrollDoc member functions to decide how much to scroll and also to do the actual scrolling, you should not have to rewrite it. It is sufficient to override some of the member functions that this function uses in order to change its behavior to suit your particular document needs.

▶ TScrollDoc Resources

TScrollDoc.rsrc contains two control definitions, one for the vertical scroll bar and one for the horizontal scroll bar. The **InitDoc** member function for **TScrollDoc** loads these controls with the toolbox function GetNewControl. These two controls are defined to occupy an empty rectangle (0,0,0,0) and the minimum, maximum, and current control values are also defined as zero. The scroll bars are resized and positioned and the control values adjusted when the document is initialized and as the contents of the document change or the window size changes.

You will probably not need to change the resources for **TScrollDoc**. Your only responsibility will be to include TScrollDoc.rsrc with the resources for your program. See Chapters 11-13 for examples of how to include these resources in your program.

▶ Summary

The most fundamental idea of the scrolling class is to use the coordinate offsetting capabilities of QuickDraw to do most of the scrolling. This simple technique can be applied to scrolling almost any kind of image in a window.

One interesting C++ technique used in the scrolling class is the use of a static member and static member function in place of a global variable. You should consider using a static member whenever you are tempted to use a global variable in your own classes.

The **TScrollDoc** class attempts to take a particularly difficult aspect of Macintosh programming and encapsulate it so that you don't have to be concerned with too many grimy details. It was originally part of the TextEdit document class but was separated into its own class to maximize its usefulness for nontext documents.

A major design goal for the class is to channel most processing through a few low-level utility member functions that are clearly identified so that a minimal amount of overriding is necessary to adapt the class to different types of data. Chapters 11 and 12 show two examples of document classes derived from **TScrollDoc**. The document class in Chapter 11 displays 'PICT' images and scrolls them with almost no change to the default scrolling mechanisms of **TScrollDoc**. The class in Chapter 12 displays text with the toolbox TextEdit functions. This class uses a different scrolling strategy yet still requires minimal changes to **TScrollDoc**.

11 ▶ PictView: Using the TScrollDoc Class

This chapter describes a program that can read 'PICT' files and display them in movable, scrollable windows. It will also print 'PICT' images to any standard printer that is attached to the Macintosh. This program uses the **TScrollDoc** class described in the previous chapter as a base class for the 'PICT' document. The code that we must write for the PictView program is remarkably short. Instead, we will rely on the parent classes for most of the program's functionality. Once again, you will be able to see how the hard work that you put into good base classes can really pay off when you are building on top of those classes.

'PICT' files are a standard Macintosh way of saving graphic images. Most graphics programs can read and write 'PICT' files. The 'PICT' file is a way of saving a QuickDraw picture from memory onto the disk. The 'PICT' format is flexible enough to display black-and-white and color pictures. The PictView application screen is shown in Figure 11-1.

This chapter discusses the members and member functions necessary to create the 'PICT' application and document class. Appendix B contains a complete code listing of the file PictView.cp.

▶ The TPICTApp Class

The application class for PictView is very simple. It is derived from the **TApp** class and inherits all the default behavior of **TApp**. There are just a few things that need to change. First, you must override **MakeDoc** to create **TPICTDoc** documents, shown as follows.

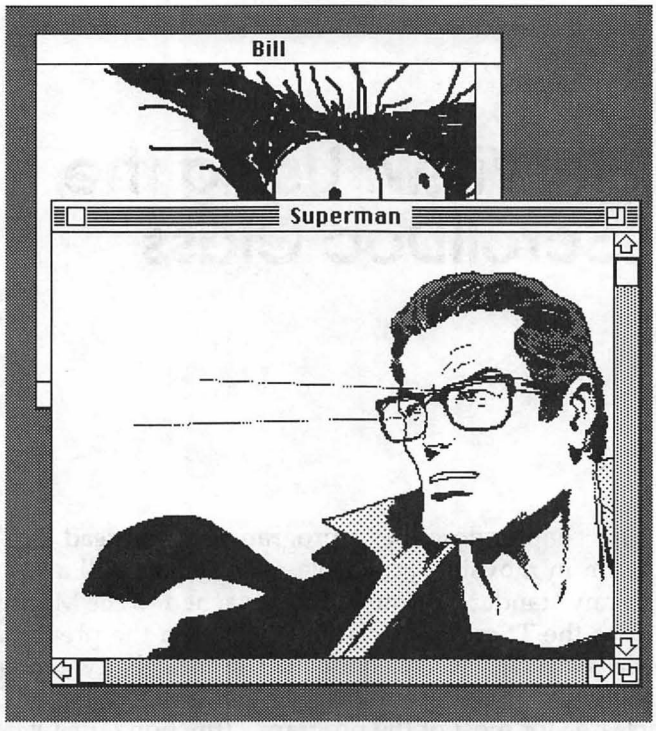


Figure 11-1. The PictView Application

```
TDoc * TPICTApp::MakeDoc(SFReply * reply){
   return new TPICTDoc(GetCreator(),reply);
}
```

Next, override those member functions that configure SFGetFile so that it will display only 'PICT' files for the user to open. Since we want to open only one kind of document, we override **GetNumFileTypes** to return 1. We put the one file type that we want to open, 'PICT', into a global array and return a pointer to the array from the overridden **GetFileTypesList** member function. Both member functions and the global variable definition are shown as follows. **GetNumFileTypes** is defined within the declaration of **TPICTApp** and **GetFileTypesList** is defined outside the declaration. The global list of file types is defined in the global area of the program.

```
// defined within TPICTApp declaration
virtual int GetNumFileTypes(void){return 1;}
```

```
// Global list of file types for SFGetFile
SFTypeList gtheTypes = {'PICT'};

SFTypeList TPICTApp::GetFileTypesList(void){
   return gtheTypes;
}
```

Finally, **TPICTApp** enables the Open menu item by overriding the **CanOpen** member function, and disables the New menu item by overriding the **CanNew** member function, as shown by the following definitions. New is disabled since PictView cannot create new documents. It can only read existing 'PICT' files created by other applications.

```
virtual Boolean CanOpen(void){return true;}
virtual Boolean CanNew(void){return false;}
```

▶ The TPICTDoc Class

The **TPICTDoc** class defines the behavior of documents in PictView. **TPICTDoc** is derived from **TScrollDoc**, which is described in Chapter 10. Because **TScrollDoc** is derived from **TDoc**, **TPICTDoc** inherits the behavior from **TDoc** that **TScrollDoc** does not override. Most of the functionality of **TPICTDoc** is inherited from its parent classes. The main work that we need to do inside **TPICTDoc** is to read 'PICT' files, draw the document picture in the window, tell **TScrollDoc** the dimensions of the document's picture, and print the picture. The declaration of **TPICTDoc** is shown as follows. Notice how it switches back and forth between public and protected sections for the members and member function declarations.

```
class TPICTDoc : public TScrollDoc{
protected:
   Handle   fPict;
   Handle   fHeader;
   THPrint  fPrintRecord;

public:
   TPICTDoc(OSType theCreator = '????',
            SFReply *reply = (SFReply*) nil);
   virtual ~TPICTDoc();
   virtual Boolean InitDoc(void);

protected:
```

```
    // routines you must override to support scrolling
    virtual short GetVertSize(void);
    virtual short GetHorizSize(void);
    virtual short GetVertLineScrollAmount(void){return 16;}
    virtual short GetHorizLineScrollAmount(void){return 16;}
    // draw the picture
    void Draw(Rect *r);

public:
    // This is the file type of the document
    virtual OSType GetDocType(){return 'PICT';}
    // this function reads in the file
    virtual Boolean ReadDocFile(short refNum);
    // disable the SaveAs menu
    virtual Boolean CanSaveAs(void){return false;}
    virtual void DoPageSetup(void);
    virtual void DoPrint(void);
    virtual Boolean CanPrint(void){return true;}
    virtual Boolean CanPageSetup(void){return true;}
};
```

▶ Constructing the Document

The constructor for the **TPICTDoc** class is very similar to the constructor for the **TScribbleDoc** class described in Chapter 8. It passes its arguments on to its parent, **TScrollDoc**, and then initializes its members to nil. The chain of inherited constructors extends back through **TScrollDoc** to **TDoc**, which is the progenitor of both **TScrollDoc** and **TPICTDoc**. Just as **TPICTDoc** passes its arguments on to **TScrollDoc**, so **TScrollDoc** will pass those arguments on to **TDoc**. The body of the constructor for **TDoc** will be executed first, the body of **TScrollDoc**'s constructor will be next, and the body of **TPICTDoc**'s constructor will be last.

```
TPICTDoc::TPICTDoc(OSType theCreator,SFReply *reply ):
    TScrollDoc(theCreator,reply) {

    fPict = nil;
    fHeader = nil;
    fPrintRecord = nil;
}
```

Key Point ▶ Constructors for parent classes execute before the constructor for a derived class.

▶ Initializing the Document

You override the **InitDoc** member function to allocate and initialize a print record for the document. This task is not done in the constructor since it is possible that the memory allocation for the print record can fail. The print record is allocated as a handle, the size of which is determined by the expression sizeof(TPrint). The resulting handle is stored in the *fPrintRecord* member of the document. If the handle is successfully created, **InitDoc** opens the toolbox Print Manager by calling PrOpen. It then calls PrintDefault to fill in the print record with default values, such as page size and orientation, for the current printer. The toolbox function PrClose closes the Print Manager. The Print Manager will be reopened later when the document needs to do other printing operations. The code for **InitDoc** is shown as follows. Notice that it calls its parent class first to take care of the parent's initialization.

```
Boolean TPICTDoc::InitDoc(void){

   if(TScrollDoc::InitDoc()){
      fPrintRecord = (THPrint)NewHandle(sizeof(TPrint));
      if(fPrintRecord != nil){
         PrOpen();
         PrintDefault(fPrintRecord);
         PrClose();
         return true;
      }
   }
   // or if something went wrong
   return false;
}
```

▶ Destroying the Document

The destructor for the **TPICTDoc** object is called when the document object is deleted. This usually happens when the user closes the document, but it can also happen if an error occurs while reading a file when the document is first created. In either case, we check the *fPict*, *fHeader*, and *fPrintRecord* handles and deallocate the memory if the handles are non-nil, as shown here.

```
TPICTDoc::~TPICTDoc(){

   if(fHeader != nil){
      DisposHandle(fHeader);
      fHeader = nil;
   }
   if(fPict != nil){
      KillPicture((PicHandle)fPict);
      fPict = nil;
   }
   if(fPrintRecord != nil){
      DisposHandle((Handle)fPrintRecord);
      fPrintRecord = nil;
   }

}
```

The destructors for the parent classes will also be automatically called when a **TPICTDoc** object is deleted. The order of destructor invocation is the opposite of the constructors, so **TPICTDoc**'s destructor is first, **TScrollDoc**'s is next, and **TDoc**'s is last.

Key Point ▶ Destructors for derived classes execute before the destructors for parent classes.

▶ Reading 'PICT' Files

'PICT' files have a 512-byte header followed by the bytes of a QuickDraw picture. The header contains application-specific information about the picture that PictView can ignore. (MacDraw, for example, puts extra information about the picture in the header.) There is no particular reason to keep the header around, although if you modified this program so that it could write files as well as read them, you would want to save the header so that it could be written back out with the file.

TPICTDoc overrides **ReadDocFile** to read the data from the file. It allocates handles for the header and the picture data and reads in the header and picture. The handle to the picture data can then be treated as a PicHandle and passed to the toolbox function DrawPicture to draw the picture.

If the memory allocations and file operations are successful, **ReadDocFile** assigns the two handles to the *fHeader* and *fPict* members of

the document object so that other document member functions can access them. It also calls the **AdjustScrollBars** member function so that the scroll bars will reflect the dimensions of the picture. **ReadDocFile** is shown as follows.

```
Boolean TPICTDoc::ReadDocFile(short refNum){

   if(fDocWindow){

      long pictLength;
      long headerLength = kPictHeaderSize;
      OSErr err = GetEOF(refNum,&pictLength);
      pictLength -= kPictHeaderSize;
      Handle thePic = NewHandle(pictLength);
      if(thePic == nil){
         ErrorAlert(rDocErrorStrings,sNoMem);
         return false;
      }
      Handle theHeader = NewHandle(headerLength);
      if(theHeader == nil){
         ErrorAlert(rDocErrorStrings,sNoMem);
         DisposHandle(thePic);
         return false;
      }
      HLock(theHeader);
      HLock(thePic);
      err = SetFPos(refNum,fsFromStart,0);
      err = FSRead(refNum,&headerLength,(Ptr)*theHeader);
      err = FSRead(refNum,&pictLength,(Ptr)*thePic);
      HUnlock(thePic);
      HUnlock(theHeader);
      if(err == noErr){
         fPict = thePic;
         fHeader = theHeader;
         AdjustScrollBars();
         return true;
      } else {
         DisposHandle(thePic);
         DisposHandle(theHeader);
         return false;
      }
   }
   // if there ain't no window...
   return false;
}
```

▶ Drawing the Picture

Because we maintain a handle to the picture as a member of the document object, we can draw the window simply by calling the toolbox functions EraseRect and DrawPicture. This is the same technique used in the Scribble program in Chapter 8. Notice that **Draw** makes no reference to the window because it is also used to draw the picture when printing the document. **Draw** is overridden as follows.

```
void TPICTDoc::Draw(Rect *r){

   if(fPict != nil){
      EraseRect(r);
      DrawPicture((PicHandle)fPict,
               &((**) (PicHandle)fPict)).picFrame));
   }
}
```

▶ Handling Scrolling

Because the scrolling routines in **TScrollDoc** are written to apply generally to all sorts of data within windows, there is actually very little that you need to do to support scrolling in PictView. You must override both the member functions that tell **TScrollDoc** how tall and wide the picture is and those that tell how far to scroll when the user clicks on the up or down arrows of a scroll bar.

GetVertSize and **GetHorizSize** determine the height and width of the picture by examining the picFrame field of the QuickDraw picture data structure, as shown here.

```
short TPICTDoc::GetVertSize(void){
   Rect r ;
   if(fPict){
      r = (**((PicHandle)fPict)).picFrame;
      return r.bottom - r.top;
   }else
      return 0;
}

short TPICTDoc::GetHorizSize(void){
   Rect r ;
   if(fPict){
```

```
        r = (**((PicHandle)fPict)).picFrame;
        return r.right - r.left;
    }else
        return 0;
}
```

The amount to scroll for an up or down arrow click is arbitrary; we chose 16 pixels in this program, but you could use any reasonable value. **TScrollDoc** overrides the member functions that return these values with definitions in the declaration of the **TPICTDoc** class, as follows.

```
virtual short GetVertLineScrollAmount(void){return 16;}
virtual short GetHorizLineScrollAmount(void){return 16;}
```

The amount to scroll when the user clicks on the page up or page down area of a scroll bar is set by default in **TScrollDoc** to be the current width or height of the window, so you don't need to override the member functions that provide those values.

All the other actions for scrolling do not need to be overridden. It was hard work putting the **TScrollDoc** class together, but you won't ever have to do that work again. Now you can reap the fruits of your labor.

▶ Printing the Document

As discussed in a previous section, the **InitDoc** member function is overridden to allocate and initialize a print record for each document. The print record contains all the information that the Print Manager needs to print the document.

TPICTDoc also supports printing by overriding the **CanPrint**, **DoPrint**, **CanPageSetup**, and **DoPageSetup** member functions, as described in the following sections.

▶ PageSetup

The print record allocated and initialized by **InitDoc** contains the default settings for the printer at the time **InitDoc** executed. The print record must be changed if the user does not want the default settings. For example, the default page orientation for most printers is portrait mode, but the user might want to print a document in landscape mode.

The toolbox function PrStlDialog can be called to let the user change the contents of the print record. It displays a style dialog that permits the user to specify characteristics such as page orientation and print quality. Different printers will have printer-specific options, such as color options for color printers, in this dialog.

The **DoPageSetup** member function is called by the application when the user chooses the PageSetup menu item. It calls the toolbox function PrOpen to open the Print Manager. Then it calls PrStlDialog, passing in the *fPrintRecord* member. PrStlDialog will display the style dialog and accept the user's choices. If the user cancels the dialog, then PrStlDialog returns false and does not change any values in the print record. If the user clicks the OK button of the style dialog, then PrStlDialog fills in the print record to reflect the user's choices in the dialog and returns true. Since the print record is not changed if PrStlDialog returns false, it is not necessary to check the function result, although you might want to know if the user has changed the print record so that you could save it with the document. The code for **DoPageSetup** is shown as follows.

```
void TPICTDoc::DoPageSetup(void){

   // open the print manager
   PrOpen();
   // put up the style dialog
   (void) PrStlDialog(fPrintRecord);
   // and close the print manager
   PrClose();
}
```

Key Point ▶

You can explicitly ignore the result of a function by placing a "(void)" typecast in front of the function call, as shown here.

```
(void) PrStlDialog(fPrintRecord);
```

Of course, the **CanPageSetup** member function must be overridden to return true, as follows, so that the PageSetup menu item will be enabled.

```
virtual Boolean CanPageSetup(void){return true;}
```

▶ Printing

A document is printed when the user chooses the Print menu item. The **CanPrint** member function must be overridden so that the Print menu item will be enabled, as shown here.

```
virtual Boolean CanPrint(void){return true;}
```

The **DoPrint** member function is overridden to actually print the picture. It begins by opening the Print Manager with the toolbox function PrOpen. Next, it passes the *fPrintRecord* member to the PrValidate function to see if the print record is compatible with the current printer. PrValidate returns false if the print record is valid (meaning, I suppose, that no validation was necessary). If the print record is valid, then **DoPrint** proceeds to the next step in the printing process. If the print record is invalid, which can happen if the user employs the Chooser to change printers after the document is initialized, then **DoPrint** passes the print record to PrStlDialog to allow the user to make new choices based on the new printer. The result of PrStlDialog is checked because if the user cancels that dialog in this context, then the entire printing operation should be canceled.

Once the print record is in order, **DoPrint** calls the Print Manager function PrJobDialog. This function displays a dialog that prompts the user to specify the number of copies and page range for the print job. This dialog should be displayed each time the document is printed. The first part of **DoPrint**, which validates the print record and displays the job dialog, is shown as follows. Notice how it calls PrClose if the user cancels either the style or job dialog.

```
void TPICTDoc::DoPrint(void){

   TPPrPort printPort;
   // open the Print Manager
   PrOpen();
   // if print record doesn't match printer,
   // put up the style dialog
   if(PrValidate(fPrintRecord))
      // if user cancels style dialog, cancel all printing
      if( ! PrStlDialog(fPrintRecord)){
         PrClose();
         return;
      } else
         fSavePageSetup = true;
```

```
// Always put up the job dialog,
// check to see if user cancels
if(! PrJobDialog(fPrintRecord)){
   PrClose();
   return;
}
```

Now that all the print record manipulation is out of the way, **DoPrint** can go ahead and print the document. It first calls PrOpenDoc to create a printing GrafPort. PrOpenDoc creates the GrafPort and sets it to be the current port, so all subsequent drawing operations will go to this GrafPort. PrOpenPage is then called to prepare a page for printing. The **Draw** member function is then called to draw into that page. Because the GrafPort is set to the printing port, all the QuickDraw functions in **Draw** will go to the printing port instead of the window. An empty rectangle is passed into **Draw** since it uses the rectangle argument to determine how much area to erase before drawing the picture and it is not necessary to erase anything when printing.

After the image is drawn, **DoPrint** calls PrClosePage to output the page. Since this program only prints one page per document, **DoPrint** then calls PrCloseDoc to close the printing GrafPort. If the program printed more than one page per document, then each page would be drawn with **Draw** and output with PrClosePage before PrCloseDoc was called. After closing the printing GrafPort, **DoPrint** calls PrPicFile if the print record indicates that the Print Manager has stored the printing output in a spool file. The Print Manager spools its output when printing to the ImageWriter, but doesn't spool when printing to the LaserWriter. PrPicFile sends the accumulated printing output in the spoolfile to the printer. The Print Manager spool file should not be confused with the print spooling offered by MultiFinder for the LaserWriter.

DoPrint finishes by calling PrClose to close the Print Manager. The second half of **DoPrint** is shown as follows.

```
// now open the printing port
printPort = PrOpenDoc(fPrintRecord,nil,nil);

// open a page
PrOpenPage(printPort,nil);

// draw the image
// use an empty rect to avoid unnecessary EraseRect
Rect r;
```

```
SetRect(&r,0,0,0,0);
Draw(&r);

// close the page
PrClosePage(printPort);


// close the printing port
PrCloseDoc(printPort);

// call PrPicFile for spooled printing (imagewriter)
if((**fPrintRecord).prJob.bJDocLoop != 0){
   TPrStatus status;
   PrPicFile(fPrintRecord,nil,nil,nil,&status);
}

// close the Print Manager
PrClose();
```

DoPrint will only print one page, even if the picture is larger than the page. Parts of the picture that extend beyond the page boundaries are simply clipped off. Supporting multiple page printing is not really much harder than single page printing, but it is beyond the scope of this chapter. MacApp automatically supports multipage printing, as you will see in Chapter 14.

▶ The PictView Main Program

The main program code for PictView is almost identical to the other programs based on **TDoc** and **TApp**. One difference is that it prompts the user to open an existing 'PICT' document at startup instead of opening a new, blank document if no documents were opened from the Finder. This is because PictView is a read-only program; it cannot create new documents. It only reads 'PICT' files created from other applications. The main program code for PictView is shown as follows.

```
void main(void)
{
   TPICTApp theApp;

   // initialize the application
   if(theApp.InitApp()){
```

```
        // allow the user to open a 'PICT' file first thing
        if(! theApp.OpenDocFromFinder())
           theApp.OpenOldDoc();

        // Start our main event loop running.
        theApp.EventLoop();

        //now clean up
        theApp.CleanUp();

    }
}
```

▶ PictView Resources

The resources for PictView are handled in the same way as the other object programs in this book. Resources from the base classes are included first, followed by the resources specific to the current application.

PictView.rsrc contains a new Apple menu with a correct About item and an 'ALRT' and 'DITL' for the About dialog. Otherwise, it relies on the resources of its base classes. Notice that TScrollDoc.rsrc is included as well, since **TPICTDoc** is derived from **TScrollDoc**.

```
/* PictView.r rez source for the PictView application */

include "TApp.rsrc" ;
include "TDoc.rsrc";
include "TScrollDoc.rsrc";
include "PictView.rsrc" ;
```

▶ The PictView Makefile

The makefile for PictView is very similar to those seen for previous programs in this book. Since **TPICTDoc** is derived from **TScrollDoc**, you need to include dependencies and directory information for the scrolling document object and resources.

First, define the directory path to the folder containing the code and resources to **TScrollDoc**. Then add this path to the options for CPlus and rez, as shown here.

```
# tell cplus and rez where to find included files for TApp, TDoc,
# and TScrollDoc
```

```
AppObjectDir = ::App/Doc:
ScrollObjDir = ::TScrollDoc:

# options for C++, where to look for include files
CPlusOptions = {SymOpts} -i "{AppObjectDir}"∂
     -i "{ScrollObjDir}"

# options for rez, where to look for include and #include files
RezOptions = -s "{AppObjectDir}" -s "{ScrollObjDir}"∂
   -i "{AppObjectDir}" -i "{ScrollObjDir}"
```

Next, add TScrollDoc.cp.o and TScrollDoc.rsrc to the list of objects and resource files. You must also include a dependency rule for TScrollDoc.cp.o, as follows.

```
Objects = ∂
   "{AppObjectDir}"TApp.cp.o ∂
   "{AppObjectDir}"TDoc.cp.o ∂
   "{ScrollObjDir}"TScrollDoc.cp.o ∂
   PictView.cp.o

ResourceFiles = ∂
   "{AppObjectDir}"TApp.rsrc ∂
   "{AppObjectDir}"TDoc.rsrc ∂
   "{ScrollObjDir}"TScrollDoc.rsrc ∂
   PictView.rsrc

# dependency rules for TScrollDoc
"{ScrollObjDir}"TScrollDoc.cp.o ƒ ∂
          "{ScrollObjDir}"TScrollDoc.cp ∂
          "{ScrollObjDir}"TScrollDoc.h ∂
          "{AppObjectDir}"TDoc.h
```

Finally, define a dependency rule for PictView.cp.p to shown that it is dependent on TScrollDoc.h as well as the other normal files, as shown here.

```
PictView.cp.o  ƒ PictView.cp ∂
        "{AppObjectDir}"TApp.h ∂
        "{AppObjectDir}"TDoc.h ∂
        "{ScrollObjDir}"TScrollDoc.h ∂
        PictView.make
```

All other parts of the makefile are the same as we've seen before. See Appendix B for a complete listing of PictView.make.

▶ Summary

By now you should be convinced of the value of object-oriented programming. The PictView program involves a minimal amount of code and explanation, yet it displays a remarkable amount of functionality. All of this is based on the strength of the base classes that we have developed in the previous chapters.

This program demonstrates how to use the **TScrollDoc** class to automate scrolling. Hopefully, you will be able to use it for your own projects so that you never have to write your own scroll bar routines. That is the point of object-oriented programming; why reinvent the wheel?

PictView also shows how to implement simple printing for documents by overriding the **DoPrint** and **DoPageSetup** member functions.

Chapter 14 describes essentially the same program using MacApp. It shows many similar strategies and capabilities, although MacApp surpasses **TApp** and **TDoc** in many ways.

12 ▶ Text Edit Document

This chapter develops a document class derived from **TScrollDoc** that knows how to use the toolbox TextEdit functions to manage and edit text. This document class, named **TTEDoc**, is able to read and write text files, display the text in a resizable and scrollable window, and support the normal Macintosh text editing operations such as selection and cut, copy, and paste.

This chapter continues to build on the base classes **TDoc** and **TScrollDoc**. This heredity model is the key power behind object-oriented programming. As shown in earlier chapters, you change only those aspects of the base classes that are necessary for the new class's distinctive behavior. Everything else remains the same.

The last sections of this chapter describe a simple application that uses **TTEDoc** documents to create a multiwindow text editor. Although **TTEDoc** is fully functional as it is written, it can also be used as a base class for derived classes that need text editing capabilities. Chapter 13 develops a debugging window that is derived from **TTEDoc** and the C++ **ostream** class to provide a convenient way to display debugging output within the Macintosh window environment.

The complete source code for the **TTEDoc** class and the application that uses it is included in Appendix B.

▶ Overview of Toolbox TextEdit

The Macintosh ROM contains a set of functions that implement the fundamental aspects of Macintosh text editing. These functions are collectively know as the Text Edit Manager, or TE. They provide a convenient way for Macintosh programmers to implement text editing in a fashion that is consistent with the Macintosh user interface guidelines.

TE depends on two overlapping rectangles to define how the text is displayed in a window. The view rectangle specifies the coordinates of the rectangle in which the visible portion of the text is displayed. It is typically equivalent to the content area of the window. The destination rectangle, on the other hand, defines the ultimate size of the rectangle in which the text is formatted. The destination rectangle is normally larger than the view rectangle, and its right edge determines where the text automatically wraps to the next line. Figure 12-1 shows a typical view and destination rectangle for a text edit document. Since rectangle coordinates are signed 16-bit numbers, the right and bottom coordinates of the destination rectangle in Figure 12-1 are as large as they can be.

TE also provides for scrolling with the function TEScroll. TE keeps track of scrolling by offsetting the destination rectangle in relation to the view rectangle. For example, when a TE document is first created, the top left corner of the view and destination rectangles are the same, as shown in Figure 12-1. As the text is scrolled toward the end of the document, the top coordinate of the destination rectangle will become negatively offset from the view rectangle, as shown in Figure 12-2.

TE uses the difference between the view and destination rectangles to determine which portion of the text is visible in the view rectangle at any one time. It is similar to, although not exactly the same as, the offset technique we used in the **TScrollDoc** class in Chapter 10. Because TE has its own mechanism for keeping track of scrolling, **TTEDoc** needs to modify the scrolling routines of **TScrollDoc** somewhat in order to take advantage of TE's built-in capabilities. But these modifications prove to be very simple, thus once again demonstrating the flexibility of a well-designed base class.

TE has some real limitations, including an absolute 32767 character limit (and a much smaller realistic limit before performance slows intolerably). You would not want to base the next heavyweight word processor on TE. Despite its shortcomings, however, TE remains a very useful tool for simple text display and editing.

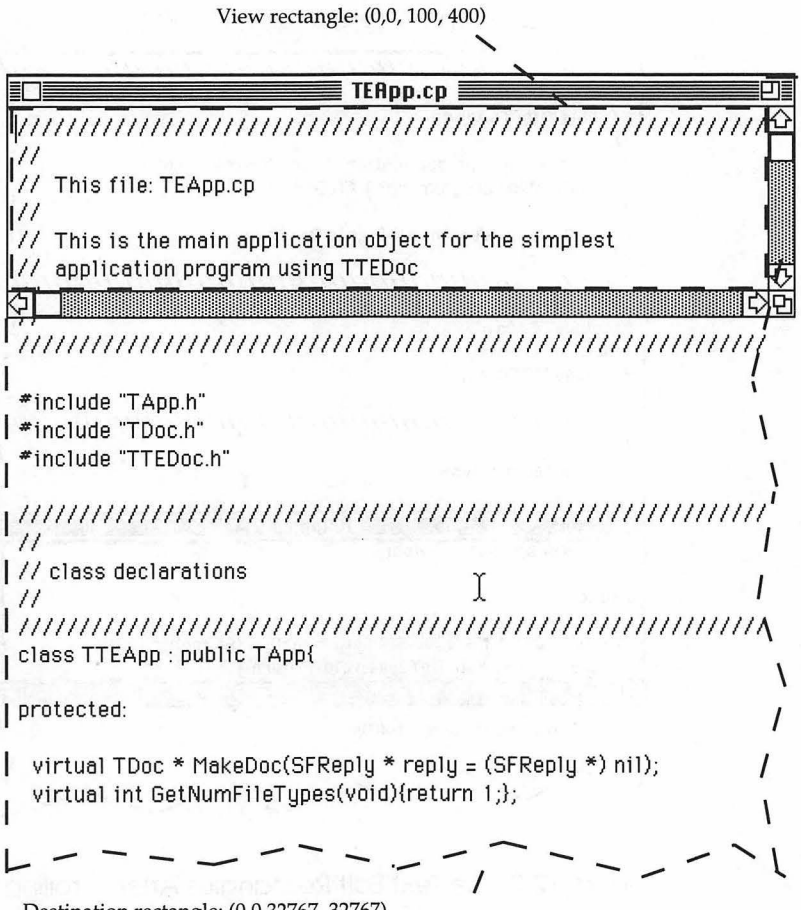


Figure 12-1. The Text Edit View and Destination Rectangles

▶ TTEDoc Members

TTEDoc defines only one new member, *fTEHandle*, to hold a handle to the TERecord that is allocated for the toolbox TextEdit functions. All other members of **TDoc** and **TScrollDoc** are also present since this class is derived from **TScrollDoc**, which in turn is derived from **TDoc**. The *fTEHandle* member is declared in the protected section of the declaration, shown as follows, so that it cannot be accessed from outside the class.

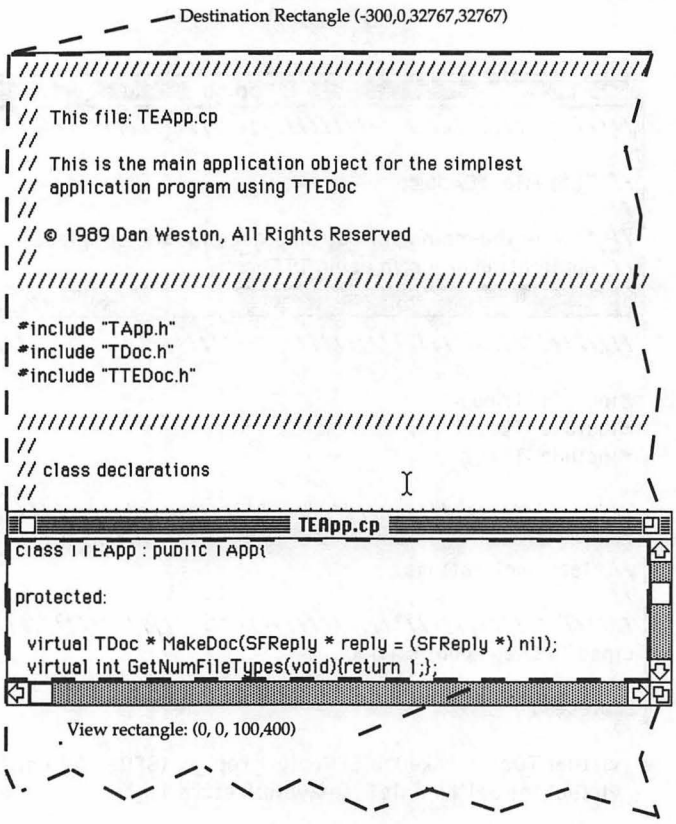


Figure 12-2. The Text Edit Rectangles After Scrolling

```
class TTEDoc : public TScrollDoc {
protected:
   TEHandle fTEHandle;
```

▶ TTEDoc Constructor and Destructor

The constructor for **TTEDoc** simply passes its arguments to its parent class and initializes *fTEHandle* to nil. This is in keeping with the minimalist behavior of constructors that has been advocated throughout this book. The real work of initializing the class will take place in the **InitDoc** member function, which is overridden in the next section. **TTEDoc**'s constructor passes its arguments to **TScrollDoc**, which in turn passes them to **TDoc**. The body of the constructor for **TDoc** will execute first, then **TScrollDoc** will execute, and finally **TTEDoc**.

```
TTEDoc::TTEDoc(OSType theCreator,SFReply * SFInfo):
   TScrollDoc(theCreator,SFInfo){
   fTEHandle = nil;
}
```

The destructor for **TTEDoc** is responsible for calling the toolbox function TEDispose to deallocate the text editing memory used by the toolbox TextEdit functions. The other deallocation tasks, such as for the window and the scroll bars, will be performed by the destructors for **TScrollDoc** and **TDoc**. Remember that destructors are executed in the reverse order of constructors, with **TTEDoc**'s destructor running first, then upward through the parent class chain so that **TDoc**'s constructor will be the last to run. The definition for **TTEDoc**'s destructor is shown as follows. Like all other document classes in this book, its destructor is declared as virtual so that it will be invoked even though the application keeps track of all documents as generic **TDoc** pointers.

```
TTEDoc::~TTEDoc(void){
   if(fTEHandle != nil){
      TEDispose(fTEHandle);
      fTEHandle = nil;
   }
}
```

▶ Initializing the Document

We override the **InitDoc** member function to initialize the text edit capabilities of the class. **InitDoc** begins by calling the parent class's **InitDoc** member function, specified as **TScrollDoc::InitDoc**. If the parent class's initialization function returns true, then we proceed with our specific initialization tasks.

Key Point ▶

Remember from Chapter 10 that **TScrollDoc**'s version of **InitDoc** calls **TDoc::InitDoc** so that the entire chain of initialization will occur. Notice, however, that while you must explicitly call the parent class's version of **InitDoc**, the constructors for parent classes are automatically invoked when the constructor for a derived class is executed.

InitDoc sets up destination and view rectangles for TE based on the document's window size (remember that **InitDoc** is not called until the document window has been created). The bottom and right coordinates of the destination rectangle are set to the maximum short integer

(32767), which means that there will be no word wrap at the right edge. This is typical of most simple text editors (as opposed to word processors, where word wrap is expected). The top and left coordinates of the destination rectangle are inset by four pixels to provide a small margin at the top and left edge of the window. We define two constants to specify the maximum integer and margin size, as shown here.

```
const short kMaxShort = 32767;
const short kTEMargin = 4;
```

Key Point

In traditional C programs, you normally define constants with #define statements, such as

```
#define kMaxShort 32767
#define kTEMargin 4
```

In C++, it is recommended that you use a const definition instead of #define whenever possible, as shown here.

```
const short kMaxShort = 32767;
const short kTEMargin = 4;
```

After setting up the view and destination rectangles, **InitDoc** calls the toolbox function TENew to create a new TERecord. TENew returns a TEHandle that we install in the *fTEHandle* member. After creating the TERecord, **InitDoc** calls the member function **SetTERect** to set the size of the view rectangle to fit exactly in the content area of the window.

Next, **InitDoc** calls the toolbox function TEAutoView to tell TE that we want it to do automatic scrolling whenever the insertion cursor moves outside the view rectangle. We also call the toolbox function SetClikLoop to install our own click loop procedure that TE will call whenever the user drags a selection outside the view rectangle. These two topics will be discussed in more detail in later sections of this chapter.

Finally, **InitDoc** returns true if the TEHandle was successfully allocated. The definition for the overridden version of **InitDoc** is shown as follows.

```
Boolean TTEDoc::InitDoc(void){

   Rect view,dest;
   if(TScrollDoc::InitDoc()){
      SetPort(fDocWindow);
      view = dest = fDocWindow->portRect;
      dest.left += kTEMargin;
      dest.top += kTEMargin;
      dest.right = kMaxShort;
      dest.bottom = kMaxShort;
      fTEHandle = TENew(&dest,&view);
      SetTERect();

      TEAutoView(true,fTEHandle);

      // install the click loop procedure
      SetClikLoop(MyClickLoop,fTEHandle);
   }
   return (fTEHandle != nil);
}
```

▶ Scrolling the Text

As mentioned in previous sections, TE has its own scrolling mechanisms so **TTEDoc** needs to override some of the scrolling member functions of **TScrollDoc**. The most important member function to override is **ScrollContents**. This is the member function that actually scrolls the image in the window and changes the offset values. **TScrollDoc** calls the toolbox function ScrollRect and also changes the *fVOffset* and *fHOffset* members of the document to reflect the amount that the image has scrolled. **TTEDoc**, on the other hand, calls the toolbox routine TEScroll instead of ScrollRect and does not change the *fVOffset* and *fHOffset* members. **TTEDoc** doesn't use the offset members because TE takes care of the scrolling position by changing the destination rectangle in relation to the view rectangle. **TTEDoc** doesn't have to keep track of the offset itself. The new definition for **ScrollContents** is shown as follows.

```
void TTEDoc::ScrollContents(short dh,short dv){
   if(fTEHandle != nil)
      TEScroll(dh,dv,fTEHandle);

}
```

The other scrolling member function that **TTEDoc** must change is **SetScrollBarValues**. **TScrollDoc** uses the value of the *fVOffset* and *fHOffset* members to determine the values of the vertical and horizontal scroll bars. In **TTEDoc**, those two members are not used and will never change from their initial zero values. Thus, **TTEDoc** must look in the TERecord for the difference between the destination and view rectangles to determine how much the text has scrolled. This is very similar, conceptually, to the offset model of **TScrollDoc**, but it relies on the internal mechanics of TE. Since **TTEDoc** lets TE take care of the scrolling, it must also use TE's data structures to determine how to set the scroll bar values. **SetScrollBarValues** is overridden as follows.

```
void TTEDoc::SetScrollBarValues(void){
   Rect visible = (**fTEHandle).viewRect;
   Rect dest = (**fTEHandle).destRect;

   short vPos = visible.top - dest.top;
   short hPos = visible.left - dest.left;
   FocusOnWindow();
   SetCtlValue(fHorizScrollBar,hPos);
   SetCtlValue(fVertScrollBar,vPos);

}
```

Notice that because the destination rectangle's top and left coordinates will typically be negative when the text has been scrolled, the resulting scroll bar values will be positive, since we subtract the negative values from the view rectangle's coordinates. This sort of negative-positive confusion is one of the hardest things to get right in any sort of scrolling scheme.

▶ AutoScrolling

TE contains built-in mechanisms for scrolling the text whenever the insertion point moves outside the visible region. This can happen when the user presses one of the arrow keys on the keyboard, or when a cut or paste operation changes the text selection. You enable those built-in scrolling mechanisms by calling the toolbox function TEAutoView, as described earlier in the section on initializing the document.

Autoscrolling is great, but it's tricky: you have to make sure that the scroll bars stay in synch with the text as it scrolls. TE doesn't know anything about scroll bars, so it doesn't do anything to them when it autoscrolls the text. It is up to **TTEDoc** to reset the scroll bars when it

thinks that the text might have scrolled. **TTEDoc** does this by calling the **SynchScrollBars** member function after any operation that may have caused autoscrolling. For example, it calls **SynchScrollBars** after every keystroke and cut or paste operation.

TE also provides autoscrolling when the user drags the mouse outside the view rectangle in the process of selecting text. This capability is particularly useful for dialog text edit controls that have no scroll bars, but it is less useful for our purposes because **TTEDoc** can't reset the scroll bars until the user lets up on the mouse button and the document gets a chance to call **SynchScrollBars**. It is better to be able to change the scroll bar settings while the user is dragging the mouse and the text is scrolling. In fact, the user finds such immediate feedback very useful.

In order to manipulate the scroll bars during drag scrolling, we must tell TE to use our procedure to scroll the text rather than its default autoscroll procedure. We can do this by passing a procedure pointer to the toolbox function SetClikLoop, as explained earlier in the section on initializing the document. Our click loop procedure cannot be a member function, so it must use the static member function **GetCurrScrollDoc** (explained in Chapter 10) to gain access to the members and member functions of the current document.

When **TTEDoc** detects a mouse click in the text area of its window, it calls the toolbox function TEClick. TEClick tracks the mouse as long as the button is held down, changing the selection as required. If the mouse moves outside the view rectangle, TEClick calls our click loop procedure repeatedly until the mouse moves back into the view rectangle. The click loop procedure examines the current mouse location and calls the **Scroll** member function to scroll the text to follow the mouse. If the mouse is outside the view rectangle, the text is scrolled toward the mouse. Remember that **Scroll** will scroll the image and adjust the scroll bars, so the user will get instant feedback from the scroll bars as the text scrolls. The code for our click loop procedure is shown as follows.

```
pascal Boolean MyClickLoop(void){
   Point where;
   Rect view;
   TScrollDoc * theCurrScrollDoc =
                    TScrollDoc::GetCurrScrollDoc ();
   theCurrScrollDoc->GetContentRect(view);

   GetMouse(&where);
   if( where.v > view.bottom){
```

```
      theCurrScrollDoc->Scroll(theCurrScrollDoc->GetVScroll(),
         theCurrScrollDoc->GetVertLineScrollAmount());
   }
   if(where.h > view.right){
      theCurrScrollDoc->Scroll(theCurrScrollDoc->GetHScroll(),
         theCurrScrollDoc->GetHorizLineScrollAmount());
   }
   if(where.v < view.top){
      theCurrScrollDoc->Scroll(theCurrScrollDoc->GetVScroll(),
         -(theCurrScrollDoc->GetVertLineScrollAmount()));
   }
   if(where.h < view.left){
      theCurrScrollDoc->Scroll(theCurrScrollDoc->GetHScroll(),
         -(theCurrScrollDoc->GetHorizLineScrollAmount()));
   }
   return true;
}
```

▶ Document Dimensions

Another set of member functions that **TTEDoc** must override to support text scrolling are those that specify the dimensions of the document image. Text, in this case, is just an image made up of characters drawn within a specified rectangle. It is really no different than the pictures displayed by PictView in Chapter 11.

TTEDoc begins by overriding **GetContentRect** to take away 4 pixels from the top and left edge of the content area of the window. This moves the text slightly down and to the right, away from the window edge, to make it more readable. The overridden version of **GetContentRect** first calls **TScrollDoc::GetContentRect** to get the default calculation for the content area, and then adjusts the resulting rectangle to allow for the text margins, as shown in the following code. Once again, you see an example of using the parent class member function and then extending its functionality.

```
void TTEDoc::GetContentRect(Rect &r){
   // ask the base class how big the rect is
   TScrollDoc::GetContentRect(r);
   // and now take away the TE margins
   r.left += kTEMargin;
   r.top += kTEMargin;
}
```

Next, **TTEDoc** defines a new member function, **SetTERect**, to make sure that the viewRect in the TERecord reflects the current size of the document window's content area. This member function is called whenever the size of the window changes, such as when you are growing or zooming the window. The definition for **SetTERect** is shown as follows.

```
void TTEDoc::SetTERect(void){
   if(fTEHandle != nil){
      // set up the view rect
      Rect r;
      GetContentRect(r);
      (**fTEHandle).viewRect = r;
   }
}
```

As we have defined **TTEDoc**, the right edge of the destination rectangle is set permanently to its maximum setting to defeat word wrap. You could change **SetTERect** to also modify the destination rectangle if you wanted a text edit document where the text automatically wrapped at the edge of the window.

GetVertSize and **GetHorizSize** must be overridden to tell the scrolling member functions the height and width of the text image. We get the vertical size by multiplying the number of lines of text times the number of pixels in each line. We get those values from the TERecord referenced by the *fTEHandle* member. The vertical dimension of the text will change as more lines of text are added or removed. The definition for **GetVertSize** is shown as follows.

```
short TTEDoc::GetVertSize(void){
   return ((**fTEHandle).nLines * (**fTEHandle).lineHeight) ;
}
```

The horizontal size of the text image does not change, unlike the vertical dimension. **TTEDoc** sets the width to match the width of the TE destination rectangle, as shown in the following definition. Even if you decided to create a TE document class where the destination rectangle was the same width as the view rectangle in order to enable automatic word wrap, the method of calculating the horizontal size of the image in the following code would still work.

```
short TTEDoc::GetHorizSize(void){
   return (**fTEHandle).destRect.right -
          (**fTEHandle).destRect. left;
}
```

Finally, **TTEDoc** overrides two other member functions to tell the scrolling functions how far to scroll when the user clicks on the up or down arrow of a scroll bar. These member functions return zero by default in **TScrollDoc**, so you must override them in any derived class. **GetVertLineScrollAmount** examines the *fTEHandle* member and returns the height of one line of text, in pixels, as the amount for vertical scrolling. The line height is also used for horizontal scrolling, although other values might be appropriate as well. **GetVertLineScrollAmount** and **GetHorizLineScrollAmount** are defined as follows.

```
short TTEDoc::GetVertLineScrollAmount(void){
    if(fTEHandle != nil)
        return (**fTEHandle).lineHeight;
    else
        return 0;
}

short TTEDoc::GetHorizLineScrollAmount(void){
    if(fTEHandle != nil)
        return (**fTEHandle).lineHeight;
    else
        return 0;
}
```

▶ Changing the Text

Once you have initialized the TextEdit document, there are three ways to add or take away text. First, the document must take keyboard input to add or delete characters. Second, it must be able to accept arbitrary chunks of text and add them to the document, such as when a text file is read and its contents displayed in the document. Last, it must support the cut, copy, and paste operations.

▶ Managing Selections

The *current selection* is the range of text that will be affected by the next text editing operation. The selection can be a single insertion point between two characters or it can include one or more characters. TE uses inverse highlighting to indicate the extent of the current selection. **TTEDoc** overrides several member functions to enable the selection capabilities.

First, to allow the user to select text by dragging the mouse, **TTEDoc** calls the toolbox function TEClick from the **ContentClick** member function in response to mouse down events in the document window. TEClick automatically tracks the mouse and changes the selection range as the mouse moves. **ContentClick** is described in more detail in a later section of this chapter.

Next, **TTEDoc** must override the **CanSelectAll** member function to tell the application that it can perform the SelectAll menu command. It must also override the **DoSelectAll** member function to perform that operation, shown as follows. **DoSelectAll** calls the toolbox function TESetSelect, indicating that the selection should encompass all the text by setting the selection start to zero and the selection end to the maximum short integer value (32767).

```
virtual Boolean CanSelectAll(void)
   {return true;}
void TTEDoc::DoSelectAll(void){
   if(fTEHandle)
      TESetSelect(0,kMaxShort,fTEHandle);
}
```

Finally, **TTEDoc** overrides the **HaveSelection** member function to indicate whether or not the selection includes at least one character. The application uses this information to enable or disable the Cut, Copy, and Clear menu items. Obviously, it makes no sense to perform one of these operations if the selection is empty. The code for **HaveSelection** is shown as follows.

```
Boolean TTEDoc::HaveSelection(void){
   if(fTEHandle)
      return ((**fTEHandle).selStart != (**fTEHandle).selEnd);
   else
      return false;
}
```

▶ Accepting Keyboard Input

TTEDoc overrides the **DoKeyDown** member function to enable keyboard input to the document. The application class calls **DoKeyDown** for the current document whenever a key down event occurs. **TTEDoc** extracts the ASCII code for the key from the EventRecord and passes it

to the toolbox function TEKey, where it will be entered into the text. If the key is the delete character, then the character just before the insertion point is deleted, or the current selection range is deleted. Arrow keys cause the insertion point to move appropriately and the text will autoscroll if the insertion point moves outside the view rectangle. Other valid characters are added to the text at the insertion point, or they replace the current selection. **TTEDoc** relies heavily on the built-in capabilities of TE to process keyboard input.

After each keystroke, **DoKeyDown** sets the *fNeedtoSave* member to true to show that the document contents have changed. Actually, it would be better to check the character and not set *fNeedtoSave* when an arrow key is pressed. **DoKeyDown** also calls the **SynchScrollBars** member function after each keystroke in case the size of the text image has changed (lines added or taken away) or the text has autoscrolled when the insertion point moved outside the view rectangle. The definition of **DoKeyDown** is shown as follows.

```
void TTEDoc::DoKeyDown(EventRecord* theEvent){
   if(fTEHandle){
      TEKey(LoWrd(theEvent->message),fTEHandle);
      fNeedtoSave = true;
      // reset the scroll bars since the key press
      // may have caused the text to scroll or added
      // text
      SynchScrollBars();
   }
}
```

▶ Adding Arbitrary Text

TTEDoc defines a new member function, **AddText**, to insert arbitrary blocks of text into the document. It calls the toolbox function TEInsert to add the text to the TERecord and sets the *fNeedtoSave* member to show that the document has unsaved changes. It then calls the toolbox function TESelView so that the text will autoscroll if the new insertion point is outside the view rectangle. Finally, it calls **SynchScrollBars** to adjust the scroll bars in case the new text has changed the size of the text image or caused the text to autoscroll. The code for **AddText** is shown as follows. This member function makes it easy for the program (rather than the user) to write text to the document, such as in the debugging document described in Chapter 13.

```
void TTEDoc::AddText(Ptr text,long len){
   if(fTEHandle != nil){
      TEInsert(text,len,fTEHandle);
      fNeedtoSave = true;
      TESelView(fTEHandle);
      SynchScrollBars();
   }
}
```

▶ Cut, Copy, Paste

TE supports cut, copy, and paste with its own private clipboard, known as the TEScrap. Several TE toolbox functions are available for cut, copy, and paste operations. **TTEDoc** uses these TE toolbox functions, but it must also adhere to the more general clipboard model defined in **TDoc** and **TApp**. Thus, much of the work in **TTEDoc** to support the clipboard is dedicated to transferring data from the TEScrap to the **TDoc** clipboard model and back again.

The first clipboard member function to override is **CanPaste**. This function tells the application if the document can accept a particular clipboard data type. **TTEDoc** returns true if the type is 'TEXT', false if otherwise, as follows.

```
virtual Boolean CanPaste(OSType theType)
                { return (theType == 'TEXT'); }
```

Next, **TTEDoc** overrides the **DoClear** member function. This function is quite simple since it doesn't affect the clipboard. Its only purpose is to delete the current selection. **DoClear** calls the toolbox function TEDelete to delete the selection, and then it sets the *fNeedtoSave* member to true to show that the document contents have changed. Finally, **DoClear** calls **SynchScrollBars** to make sure that the scroll bars are in synch with the new text image size and scroll position.

```
void TTEDoc::DoClear(void){
   if(fTEHandle){
      TEDelete(fTEHandle);
      fNeedtoSave = true;
      SynchScrollBars();
   }
}
```

The **DoCopy** member function is declared in **TDoc** to take two arguments. The first argument is a pointer to a handle that the function is supposed to fill in with a handle to the data that is being copied. **DoCopy** is responsible for allocating the memory for the copied data and passing the handle to the data back to the caller through the first argument. The second argument is a pointer to an OSType variable that **DoCopy** should fill in with the data type of the copied data. This is a very general model for clipboard operations.

When **TTEDoc** overrides **DoCopy**, it first calls the toolbox function TECopy to take the current text selection and place it in the TEScrap. Next, it sets the clip type argument to 'TEXT' and the data handle argument to nil. **DoCopy** then calls the toolbox function TEScrapHandle to get the handle to the data that was just copied by TECopy. It then calls the toolbox function HandToHand to copy the data specified by that handle into a new block. **DoCopy** passes that new handle back to the caller by setting the handle indicated by the first argument. Essentially, **DoCopy** is allowing TE to take care of all the details regarding the copy operation on the document text and then copying TE's copy of the data into a new handle and passing the new handle back to the application. **DoCopy** is defined as follows.

```
Boolean TTEDoc::DoCopy(Handle *theData,OSType *theType){
   if(fTEHandle){
      // put data on TEScrap
      TECopy(fTEHandle);

      // set theType
      *theType = 'TEXT';

      // do this in case we fail
      *theData = nil;

      // copy the handle to the data
      Handle TEData = TEScrapHandle();
      OSErr err = HandToHand(&TEData);
      if(err != noErr)
         return false;
      *theData = TEData;

      return true;
   }
}
```

Notice that **DoCopy** does not change the text in the document; it only copies the selection to the clipboard. **DoCut**, on the other hand, deletes the selection after copying it. **DoCut** is easy to implement since it simply calls the **DoCopy** and **DoClear** member functions that we have already defined, as follows.

```
Boolean TTEDoc::DoCut(Handle *theData,OSType *theType){
   Boolean  result;
   if (result = DoCopy(theData,theType))
      DoClear();
   return result;
}
```

Finally, **TTEDoc** must override the **DoPaste** member function to implement the paste operation. This function receives a handle to the data to be pasted from the application. It puts this data handle in the low-memory global TEScrapHandle to tell TE that this is the data it should use for subsequent TE paste operations. Notice that **DoPaste** has to set the low-memory global directly, since there is no toolbox function to do it for us. Once **DoPaste** sets the TEScrapHandle to point to the appropriate data, it can call the toolbox function TEPaste to perform the paste operation on the document text. It also sets the *fNeedToSave* member and calls **SynchScrollBars** to finish the operation. **DoPaste** is shown as follows.

```
void TTEDoc::DoPaste(Handle theData,OSType theType){

   if((fTEHandle) && (theType == 'TEXT')){
      // put data in TEScrap
      long scrapLen = GetHandleSize(theData);
      TESetScrapLen(scrapLen);

      // set low-memory TEScrap handle with our data handle
      Handle * TEScrapHandle = (Handle *) TEScrpHandle;
      *TEScrapHandle = theData;

      // now go ahead and paste
      TEPaste(fTEHandle);

      fNeedtoSave = true;
      SynchScrollBars();
   }
}
```

▶ Handling Events

Several of the event handling member functions defined in **TDoc** and **TScrollDoc** need to be overridden in **TTEDoc** to account for the text editing capabilities in the derived class. These member functions are discussed in the following sections. In most cases **TTEDoc** is extending the functionality of the base class member functions — first it calls the base class function and then it proceeds with the text-specific operations to complete the function.

▶ Activation/Deactivation

TE has built-in support for activation and deactivation. When a text document window becomes active, its selection must be highlighted and the insertion cursor shown. When the window is deactivated, the selection must be unhighlighted and the cursor hidden. **TTEDoc** overrides the **Activate** and **Deactivate** member functions to call the toolbox functions TEActivate and TEDeactivate, respectively. The definitions for **Activate** and **Deactivate** are shown as follows.

```
void TTEDoc::Activate(void){
   TScrollDoc::Activate();
   if(fTEHandle)
      TEActivate(fTEHandle);
}

void TTEDoc::Deactivate(void){
   TScrollDoc::Deactivate();
   if(fTEHandle)
      TEDeactivate(fTEHandle);
}
```

Notice that each of these overridden functions begins by calling the matching member function for the parent class, **TScrollDoc**, in order to get the default behavior before going on to the processing necessary for the derived class.

▶ Drawing the Text

The **Draw** member function is called in response to update events in the document window. **TTEDoc** simply erases the specified rectangle and draws the text by calling the toolbox routine TEUpdate to draw the text, shown as follows.

```
void TTEDoc::Draw(Rect *r){
   if(fTEHandle){
      EraseRect(r);
      TEUpdate(r,fTEHandle);
   }
}
```

▶ Mouse Clicks on Text

As mentioned in an earlier section, **TTEDoc** calls the toolbox function TEClick from the **ContentClick** member function. If the user clicks on a location in the text, the insertion point is moved to that location. If the user clicks and holds the mouse button, TEClick tracks the mouse and changes the selection while the user is dragging the mouse. We want our click loop procedure, which is called from TEClick while the mouse button is held down, to detect when the mouse leaves the view rectangle and to scroll the text appropriately to extend the selection.

TTEDoc installed our click loop procedure during document initialization with the toolbox function SetClikLoop. **TTEDoc** must turn off TE's built-in default autoscrolling while the mouse is being dragged in order to get the custom click loop procedure called. When **TTEDoc** responds to keystrokes, the default autoscrolling is sufficient since the document gets a chance to adjust the scroll bars after each keystroke. But when the mouse is being dragged, we want to use our own click loop procedure instead of the default procedure because the default procedure doesn't allow us to adjust the scroll bars while the mouse button is being held down. Our custom click loop procedure scrolls the text and adjusts the scroll bars repeatedly while the mouse button is being held down, thus giving the user immediate feedback.

ContentClick calls the toolbox function TEAutoView with false as the first argument in order to turn off TE's autoscrolling functions. Then it calls TEClick to track the mouse as long as the button is being held down. TEClick will call our click loop procedure while tracking the mouse to give us a chance to scroll the text and update the scroll bars. After TEClick returns, we turn the autoscroll features back on by calling TEAutoView with true for the first argument. The code for **ContentClick** is shown as follows.

```
void TTEDoc::ContentClick(EventRecord *theEvent){
   Boolean shiftKeyDown = ((theEvent->modifiers & shiftKey) != 0);
   if(fTEHandle){
      // turn off autoscrolling, we do it ourselves for clicking
      TEAutoView(false,fTEHandle);
      TEClick(theEvent->where,shiftKeyDown,fTEHandle);
```

```
      TEAutoView(true,fTEHandle);
   }
}
```

▶ Idle Events: Adjusting the Cursor

The application calls the **DoIdle** member function for the current document whenever it sees that no other events are waiting in the event queue. **TTEDoc** overrides the **DoIdle** function to call the toolbox function TEIdle, which blinks the insertion cursor. It also gets the current mouse location, in local coordinates, and passes that location to the **AdjustCursor** member function. **AdjustCursor** sets the mouse cursor to the I-beam cursor shape that is normally associated with text insertion when the mouse is within the content area of the document window. Otherwise it sets the cursor to the arrow shape with the toolbox function InitCursor. The code for **DoIdle** and **AdjustCursor** is shown as follows.

```
void TTEDoc::DoIdle(void){
   TScrollDoc::DoIdle();
   if(fTEHandle){
      TEIdle(fTEHandle);
   }

   GrafPtr oldPort;
   GetPort(&oldPort);
   SetPort(fDocWindow);
   Point thePt;
   GetMouse(&thePt);
   AdjustCursor(thePt);
   SetPort(oldPort);
}

void TTEDoc::AdjustCursor(Point where){
   Rect r;
   // decide if it is in content or scroll bars
   GetContentRect(r);
   if(PtInRect(where,&r)){
      CursHandle IBeam = GetCursor(iBeamCursor);
      if(IBeam != nil){
         SetCursor(*IBeam);
      }
   }else
      // it must be in the scroll bars or grow box
      InitCursor();
}
```

Growing and Zooming

The code for growing and zooming the document window is simple. In both cases, you just call the matching member function from the parent class to take care of resizing the window and the scroll bars and then call **SetTERect** to adjust the view rectangle for the text to fit the new window size. The code for **DoGrow** and **DoZoom** is shown as follows. Once again, you can see the power of inheritance.

```
void TTEDoc::DoGrow(EventRecord*  theEvent){
   // call the parent class, this will adjust scroll bars
   TScrollDoc::DoGrow(theEvent);
   // adjust the TE rectangle
   SetTERect();
}
void TTEDoc::DoZoom(short partCode){
   // call the parent class, this will adjust scroll bars
   TScrollDoc::DoZoom(partCode);
   // adjust the TE rectangle
   SetTERect();
}
```

File Operations

In addition to the text editing capabilities described in the previous sections, **TTEDoc** can also read and write text files. As with the TScribbleDoc class in Chapter 8, you need to override only a few member functions to enable the document's file operations.

GetDocType is the member function that tells the application the file type of the document's files. **TTEDoc** overrides this method to return 'TEXT'. **GetDocType** is defined in the declaration of **TTEDoc** as follows.

```
virtual OSType GetDocType(){return 'TEXT';}
```

TTEDoc also overrides the **CanSaveAs** member function to return true so that the SaveAs menu item will be enabled whenever a **TTEDoc** document is active. The code for **CanSaveAs** is defined within the declaration of the **TTEDoc** class as follows.

```
virtual Boolean CanSaveAs(void){return true;}
```

As explained in Chapter 8, the **CanSave** member function is defined by default to depend on the value of the *fNeedtoSave* member, so **TTEDoc** doesn't need to override **CanSave** to enable the Save menu item; it will be enabled whenever the *fNeedtoSave* member for the current document is true.

▶ Reading 'TEXT' Files

By the time the **ReadDocFile** member function is called, the file is already open. **ReadDocFile** must determine how many bytes are in the file and allocate sufficient memory to read the entire file. Because of the 32767 character limit for TE, **ReadDocFile** will not read more than that many characters. Once the memory has been successfully allocated, **ReadDocFile** sets the file position to the beginning of the file and reads the entire file into the handle that was just allocated. Assuming that there are no errors, **ReadDocFile** uses that handle to set the text for the document with the toolbox function TESetText. Since TESetText makes a copy of the specified text, we dispose of the handle once the text has been passed to TE. Finally, **ReadDocFile** sets the text selection position to zero and calls **SynchScrollBars** so that the scroll bars will reflect the amount of text in the document. The code for **ReadDocFile** is shown as follows.

```
Boolean TTEDoc::ReadDocFile(short refNum){
   if((fDocWindow) && (fTEHandle != nil)){
      long len;
      OSErr err = GetEOF(refNum,&len);
      // truncate to TE limits
      if(len > kMaxShort){
         len = kMaxShort;
      }
      Handle thetext = NewHandle(len);
      if(thetext == nil){
          ErrorAlert(rDocErrorStrings,sNoMem);
          return false;
      }
      HLock(thetext);
      err = SetFPos(refNum,fsFromStart,0);
      err = FSRead(refNum,&len,(Ptr)*thetext);
      HUnlock(thetext);
      if(err == noErr){
         // set the text in the TERecord
         HLock(thetext);
         TESetText(*thetext,len,fTEHandle);
         HUnlock(thetext);
         DisposHandle(thetext);

         // set the selection to the first char
         TESetSelect(0,0,fTEHandle);
         SynchScrollBars();
         return true;
```

```
      } else {
         DisposHandle(thetext);
         return false;
      }
   }
   // if there ain't no window...
   return false;
}
```

▶ Writing 'TEXT' Files

Like **ReadDocFile**, the file is already open by the time **WriteDocFile** is called. **TTEDoc**'s only responsibility in this member function is to write the text from the document out to the file. The toolbox function TEGetText returns a handle to the text and **WriteDocFile** writes the contents of that handle out to the file, resetting the file position to zero before writing. That's all there is to it. The code for **WriteDocFile** is shown as follows.

```
Boolean TTEDoc::WriteDocFile(short refNum){
   if((fDocWindow != nil) && (fTEHandle != nil)){
      long len = (long)(**fTEHandle).teLength;
      CharsHandle thetext = TEGetText(fTEHandle);
      HLock((Handle)thetext);
      OSErr err = SetFPos(refNum,fsFromStart,0);
      err = FSWrite(refNum,&len,(Ptr)*thetext);
      HUnlock((Handle)thetext);
      if(err == noErr)
         return true;
      else
         return false;
   }
   // if there ain't no window...
   return false;
}
```

▶ Using TTEDoc: TTEApp

The remainder of this chapter is devoted to developing a simple application that uses the **TTEDoc** class. A derived application class, **TTEApp**, is described along with a main program and makefile to put it all together. The TE application is similar to the applications developed in Chapters 7-9 and 11, so it will be treated rather briefly here.

▶ The TTEApp Class

You must define a derived application class for the TE application. Its main jobs are to create TE documents and configure SFGetFile to read 'TEXT' files. It also must override the **CanAcceptClipType** member function to specify that the application will support 'TEXT' clipboard data. The declaration of **TTEApp** is shown as follows.

```
class TTEApp : public TApp{
protected:
   virtual TDoc * MakeDoc(SFReply * reply =  SFReply *) nil);
   virtual SFTypeList GetFileTypesList(void);
   virtual int GetNumFileTypes(void){return 1;};
   virtual Boolean CanOpen(void){return true;}
   virtual OSType CanAcceptClipType(void){return 'TEXT';}
};
```

As always, you override **MakeDoc** to create a specific document type rather than the default **TDoc**. In **TTEApp**, you want to make **TTEDoc** documents, as follows.

```
TDoc * TTEApp::MakeDoc(SFReply * reply){
   return new TTEDoc(GetCreator(),reply);
}
```

You also must define a global variable to hold the file types list containing one element specifying that you want to read 'TEXT' files, as shown here.

```
SFTypeList gtheTypes = {'TEXT'};
```

Likewise, you override **GetFileTypesList** to return a pointer to the file types list just defined.

```
SFTypeList TTEApp::GetFileTypesList(void){
   return gtheTypes;
}
```

And you must override **GetNumFileTypes** to return 1 to indicate that you only want to read one type of file, as follows.

```
virtual int GetNumFileTypes(void){return 1;};
```

Since **TTEApp** can open existing text files, you override **CanOpen** to return true, as shown here.

```
virtual Boolean CanOpen(void){return true;}
```

And finally, you override **CanAcceptClipType** to return 'TEXT', indicating that the application can accept data from the system clipboard if it has the clipboard data type 'TEXT'.

```
virtual OSType CanAcceptClipType(void){return 'TEXT';}
```

As you can see, the modifications necessary to adapt the application base class to work with **TTEDoc** are minimal and easy to implement.

▶ The Main Program

The main program for the text application is exactly like the main program for the Scribble program in Chapter 8, except that here you create an application object from the **TTEApp** class instead of the **TScribbleApp** class. The code for the main program is shown as follows.

```
void main(void)
{
   // create an instance of TTEApp
   TTEApp theApp;
   // initialize the application
   if(theApp.InitApp()){
      // open one window to start with,
      // unless we got files from the Finder
      if(! theApp.OpenDocFromFinder())
         theApp.OpenNewDoc();
      // run the event loop until user quits
      theApp.EventLoop();
      //now clean up
      theApp.CleanUp();
   }
}
```

▶ TEApp Resources

The resources for the TE application are pulled from TApp.rsrc, TDoc.rsrc, TScrollDoc.rsrc, and TEApp.rsrc. The resources in the first three files were discussed in the chapters that described the associated classes. The last file, TEApp.rsrc, contains a new Apple menu with the text of the About item changed and a new 'ALRT' and 'DITL' to display the correct information about the program when the user chooses the About item. These are the minimal resource changes necessary to create a derived application. The file TEApp.r contains the following statements to include the resources from all four files. Remember that the last resources included replace equivalent resources read from earlier files.

```
include "TApp.rsrc";
include "TDoc.rsrc" ;
include "TScrollDoc.rsrc";
include "TEApp.rsrc";
```

▶ The TEApp Makefile

The makefile for the TE application is also very similar to those seen for the other applications in this book. One addition is the definition of a new source path for the **TTEDoc** class. C++, the Linker, and rez need to know about the **TTEDoc** folder so they can find the header files and resources necessary to build the complete application. The following statement is added at the beginning of the makefile to define the relative path to the folder containing the **TTEDoc** class source code and resources.

```
TEObjDir = ::TEDoc:
```

You then add that source path to the options for C++ and rez, as follows.

```
# options for C++, where to look for include files
CPlusOptions = {SymOpts} ∂
         -i "{AppObjectDir}"∂
         -i "{TEObjDir}"∂
         -i "{ScrollObjDir}"

RezOptions = -s "{AppObjectDir}" -s "{ScrollObjDir}"∂
         -i "{AppObjectDir}"  -i "{ScrollObjDir}"
```

The object code for the **TTEDoc** class must also be added to the list of object code files for the finished program, as shown here.

```
Objects = ∂
   {AppObjectDir}"TDoc.cp.o ∂
   "{ScrollObjDir}"TScrollDoc.cp.o ∂
   "{TEObjDir}"TTEDoc.cp.o ∂
   TEApp.cp.o
```

The resource file, TEApp.rsrc, must be included in the list of resource files that are combined in the final program, as follows.

```
ResourceFiles = ∂
   "{AppObjectDir}"TApp.rsrc ∂
```

```
"{AppObjectDir}"TDoc.rsrc ∂
"{ScrollObjDir}"TScrollDoc.rsrc ∂
TEApp.rsrc
```

You must define a new dependency rule for the **TTEDoc** class, shown as follows. Notice that TTEDoc.cp.o is dependent on TScrollDoc.h and TDoc.h, showing its inheritance lineage.

```
# dependency rules for TTEDoc
"{TEObjDir}"TTEDoc.cp.o ƒ "{TEObjDir}"TTEDoc.cp ∂
                  "{TEObjDir}"TTEDoc.h ∂
                  "{ScrollObjDir}"TScrollDoc.h ∂
                  "{AppObjectDir}"TDoc.h
```

And finally, the dependencies for **TEApp.cp.o** are defined with the following statement.

```
TEApp.cp.o  ƒ TEApp.cp ∂
           "{AppObjectDir}"TApp.h ∂
           "{AppObjectDir}"TDoc.h ∂
           "{ScrollObjDir}"TScrollDoc.h ∂
           "{TEObjDir}"TTEDoc.h ∂
           "TEApp.make
```

See Appendix B for a complete listing of TEApp.make, as this section has discussed only the parts of the file that are different from other makefiles described in previous chapters.

▶ Summary

This chapter described a TextEdit document class built on top of the existing **TDoc** and **TScrollDoc** base classes. **TTEDoc** knows how to edit text and read and write text files. It also shows how to utilize the clipboard member functions to implement cut, copy, and paste operations.

TTEDoc is a very useful class. Almost every program can use some sort of simple text editing capability. Because it is completely self-contained, **TTEDoc** documents can be used along with other document types in an application that supported more than one document type.

TTEDoc also shows how to adapt **TScrollDoc** to a different scrolling mechanism. It uses the built-in scrolling from the toolbox TE functions within the structure of the **TScrollDoc** class member functions to provide scrolling text. This is much easier than trying to implement scrolling text from scratch.

This chapter also built a small application to use **TTEDoc**. With minimal effort, the **TApp** class was modified to support the text editing documents. The result was a multiwindow editor.

TTEDoc can be used as a functional class just the way it is, or it can be used as a base class to derive specialized text editing documents. The next chapter uses multiple inheritance to create a document class based on **TTEDoc** and the **ostream** class that is part of the C++ I/O library. The result is a text document that you can treat just like a stream.

13 ▶ TDebugDoc: Streams and Multiple Inheritance

This chapter develops a new document class, **TDebugDoc**, based on the **TTEDoc** class described in the preceding chapter. In addition, **TDebugDoc** is also derived from the C++ **ostream** class. The ability to inherit from more than one class is called *multiple inheritance*, and it is one of the most far-reaching features of C++ 2.0. Thus, **TDebugDoc** has all the members and member functions of the **TTEDoc** class plus all the members and member functions of the **ostream** class. You can treat a **TDebugDoc** object just like a **TTEDoc**, or just like an **ostream**. It embodies the characteristics of both parent classes.

TDebugDoc can be used to write debugging messages into a window during program execution. Because it inherits from **TTEDoc**, it knows how to display and edit text in a scrollable window, and it can be manipulated by the application just like other documents for activation and update purposes. And, because it inherits from the **ostream** class, it knows how to format numbers and strings for textual output. By combining these capabilities, you can use all the formatting strengths of streams and all the display features of **TTEDoc**.

You will need to modify the base classes very little to create this new hybrid class. Streams typically use the standard input and output channels of the operating environment, as we saw when using streams with MPW in Chapter 4. In this chapter we will modify the stream so that it sends its output to a text window. The techniques shown in this chapter can be extended and applied to create other stream derivatives, such as a stream class that sends its output over a network.

The complete source code for this class and an application that demonstrates how to use it are listed in Appendix B.

▶ About Multiple Inheritance

C++ 2.0 allows a class to be derived from more than one parent class, which can be an extremely useful design tool when you are creating new classes. You can combine capabilities from different classes into new classes with characteristics of all the parent classes. And if the new class is derived publicly from the parent classes, it will have access to all the public and protected members and member functions of its parents.

The basic idea behind multiple inheritance is easy to grasp, but there are a few snags that you must avoid when combining classes. First, the same member name or member function name in two or more parent classes causes ambiguity when the derived class tries to access the shared name. Although you can get around this ambiguity, it is best to eliminate it altogether by using dissimilar member names in your parent classes. The second restriction is that Apple's C++ does not allow multiple inheritance from parent classes that are derived from the HandleObject class or the PascalObject class. This is the reason I did not use HandleObjects as the basis for **TDoc** in this book, even though HandleObjects use memory more efficiently than regular objects do. (If you don't expect to use multiple inheritance, it would be a good idea to redefine **TDoc** as a HandleObject.)

There is much more to say about multiple inheritance, but it is beyond the scope of this chapter. Here we will simply show how to create a working class using multiple inheritance and leave the discussion of the finer points to other references.

▶ Streams and Streambufs

Chapter 4 contained many examples of using streams to produce formatted output in the MPW environment. The current chapter concentrates on the internal workings of streams in order to modify their behavior.

The two main stream classes that are defined in the C++ stream library are **ostream** and **istream**. **Ostream** handles output and **istream** handles input. A third class, **iostream**, is available for streams that need both input and output.

Stream I/O is built in two layers — one for formatting and the other for raw character input and output. The **ostream** and **istream** classes are responsible for formatting. The **streambuf** class is responsible for raw input and output. For output, the stream takes the insertion argument and formats it into a sequence of characters. It then passes those characters on to the **streambuf** for consumption. Similarly for input,

characters pass into the **streambuf** from some source, such as the keyboard or a disk file, and are withdrawn from the **streambuf** by the stream. Once the stream withdraws the characters from the **streambuf**, it scans them and converts them into the format required by the extraction argument.

When used for input, a **streambuf** is said to produce characters, and when used for output, the **streambuf** is said to consume characters. For example, consuming the characters in the **streambuf** might entail writing the characters to a disk file, or sending them over the serial port or to the standard I/O channel of MPW. All stream objects have an associated **streambuf** that is required as an argument when the stream is created.

The formatting capabilities of streams are rarely modified directly. Rather, you will normally extend the formatting abilities by overloading the **insertion** (<<) and **extraction** (>>) operators to handle your own data types and classes in addition to the predefined formatting definitions.

In contrast, the consumption and production of characters by the **streambuf** class are often replaced by totally new capabilities in order to redirect stream I/O. By deriving new classes from **streambuf**, you can change the raw consumption and production mechanisms for the **streambuf**.

Two member functions of the **streambuf** class are crucial to redefining the production and consumption mechanisms. The first member function, **overflow**, contains the low-level code that actually consumes the characters, whether to a disk file or some other destination. The other function, **underflow**, contains the code that actually produces the characters from their source. So, by overriding **underflow** and **overflow**, you can control where the characters come from and where they go.

A **streambuf** object typically has an associated area of memory where it buffers characters so that the actual input or output does not have to proceed one character at a time. For example, if a **streambuf** has an 80-character buffer, it could accept up to 80 characters before **overflow** needed to be called. Of course, you can also flush the buffer even if it is not completely full. It is also possible to create a **streambuf** that has no buffer. An unbuffered **streambuf** performs raw I/O each time a character is inserted or extracted.

The following section develops a derived **streambuf** class that knows how to output its characters to a **TTEDoc** window. That derived **streambuf** will then be associated with an **ostream** so that all insertion operations on that stream will appear in the text window. You can also find more details on streams and **streambufs** in the "iostream Examples" section of the MPW C++ manual.

Overflow is the only member function that we need to override in the **streambuf** class to redirect output. If we wanted to use this **streambuf** for input as well, we would also override the **underflow** member function. The following sections show how to associate the derived **streambuf** with a stream and how to send output to that stream.

▶ TDebugDoc

TDebugDoc is a text editing document. **TDebugDoc** is an **ostream**. **TDebugDoc** is both. This is the wonder of multiple inheritance. We derive **TDebugDoc** from the **TTEDoc** class to get all of its text editing and window management functions. We also derive from the **ostream** class to get all of that class's output capabilities. We want **TDebugDoc** to have access to the public and protected members of the parent classes, so we declare that it is publicly derived from both parents. The declaration of **TDebugDoc** is shown as follows.

```
class TDebugDoc : public TTEDoc,public ostream {
protected:
    TWindowStreamBuff * fBuff;
public:
    TDebugDoc::TDebugDoc(TWindowStreamBuff *buff,
                         OSType theCreator = '????',
                         SFReply * SFInfo = (SFReply *)nil);
    virtual ~TDebugDoc(void);
    virtual short GetWinID(void){return rDebugDoc;}
    // do this so Close menu isn't active
    // when debug window is on top
    virtual Boolean CanClose(void) { return false; };
};
```

Notice that on the first line we state that **TDebugDoc** has two paren classes, and that the public keyword is used for each parent class, ind pendent of the other. The list of parent classes is separated by comma You can have more than two parent classes.

Key Point ▶

To declare a derived class with multiple inheritance, list the paren classes, separated by commas, in the first line of the derived class declaration, as shown here.

```
class TDebugDoc : public TTEDoc,  public ostream {
    // ...
}
```

The declaration includes a constructor and destructor for **TDebugDoc** and a new member, *fBuff*, which holds a pointer to a **TWindowStreamBuff**. We use this pointer to shut down the **streambuf** when the document is deleted so that no further output will be attempted.

TDebugDoc also overrides the member functions **GetWinID** and **CanClose**. **GetWinID** is defined in the class declaration to return the ID number of the 'WIND' resource for the debugging document window. **TDebugDoc** uses a different resource from the default document window since we don't want the debugging window to have a close box. Toward this same end, **CanClose** is overridden to return false so that the Close menu item is disabled whenever the debugging document is the active document.

▶ TDebugDoc Constructor and Destructor

Like other derived constructors that have been described in this book, the constructor for **TDebugDoc** must pass its arguments on to its parent classes. When there is more than one parent class, the syntax for passing arguments to them is similar to the method used with one parent except that you use a list of parent classes, separated by commas, instead of just one class. Each parent class is listed along with the arguments it is to receive.

TDebugDoc's constructor takes three arguments. The first is a pointer to a **TWindowStreamBuff**. This **streambuf** must be created before we can create a **TDebugDoc** because it is required as an argument to the parent **ostream** class. **TDebugDoc** also uses the **streambuf** argument to initialize the *fBuff* member. The other two arguments are a creator designator and an SFReply pointer, which are passed to the document parent class. The definition for **TDebugDoc**'s constructor is as follows.

```
TDebugDoc::TDebugDoc(TWindowStreamBuff *buff,
                     OSType theCreator,SFReply * SFInfo):
        TTEDoc(theCreator,SFInfo),
        ostream(buff){
    // save a reference to the streambuffer
    // so we can disable it when the window closes
    fBuff = buff;
}
```

Key Point ▶

To pass constructor arguments on to multiple parent classes, use a list of parent classes, separated by commas, with the arguments for each parent class enclosed in parentheses, as shown here.

```
TDebugDoc::TDebugDoc(TWindowStreamBuff *buff,
                     OSType theCreator,SFReply * SFInfo):
              TTEDoc(theCreator,SFInfo),
              ostream(buff){
   // ...
}
```

The destructor for **TDebugDoc** is declared as a virtual function so that it will be called even though the application will delete it with a generic **TDoc** pointer. Its sole task is to set the **TWindowStreamBuff**'s *fTEDoc* member to nil so that the **streambuf** will not try to do anymore output. Remember that **TWindowStreamBuff**'s overflow function always checks the *fTEDoc* member before attempting to send text to the document. When the document is being deleted, it must make sure that no more output is sent to it. The code for **~TDebugDoc** is shown as follows.

```
TDebugDoc::~TDebugDoc(void){
    // disable the streambuff so it won't
    // try to output to a deleted document
    fBuff->fTEDoc = nil;
}
```

▶ Making Debug Documents

MakeDebugDoc is a utility function that knows how to create **TDebugDoc** objects and add them to the application's document list. It creates a **TWindowStreamBuff** after first allocating memory for the **streambuf**'s buffer. Once the **TWindowStreamBuff** is created, it is used as an argument to create a new **TDebugDoc**. The resulting **TDebugDoc** is then initialized and added to the application's document list. Once the document is part of the application's document list, it will be treated like any other document derived from **TDoc**.

The **TWindowStreamBuff** is used as an argument to create the **TDebugDoc** document so that the **ostream** part of **TDebugDoc** will have a **streambuf** to send its characters to. But the **TWindowStreamBuff** object itself must contain a reference to the **TDebugDoc** document so

that it has a place to send its final output. This sort of circular referencing can be tricky to initialize. The final task of **MakeDebugDoc** is to set the *fTEDoc* member of the **TWindowStreamBuff** object to point to the **TDebugDoc** object that was just created. The code for **MakeDebugDoc** is shown as follows.

```
TDebugDoc * MakeDebugDoc(TApp * theApp){
    // grab some memory for the stream buffer
    char * theBuffer = new char[kBufferSize];
    if(!theBuffer)
        return nil;
    // create the streambuffer
    TWindowStreamBuff *buff = new TWindowStreamBuff(theBuffer,
            kBufferSize);
    // and pass it to the new DebugDoc's constructor
    TDebugDoc * temp = new TDebugDoc(buff);
    if(! temp)
        return nil;
    // make the window
    if( temp->MakeWindow(theApp->fenvRec.hasColorQD) &&
            temp->InitDoc()){
        temp->ShowDocWindow();
        theApp->AddDocument(temp);
        // connect the streambuff to the TTEDoc document
        buff->fTEDoc = temp;
        return temp;
    } else
        return nil;
}
```

▶ Using Debug Documents

Once you have defined the **TDebugDoc** class as described in previous sections, you can use it in your programs to provide diagnostic text output while the program executes. A typical use of **TDebugDoc** is to declare a single global pointer to a **TDebugDoc** object. The **MakeDebugDoc** function is used to create a debugging document and it is assigned to the global variable. Then other functions in your program can access the global variable as if it were a pointer to an **ostream**. The following code shows how to declare the global variable and create the debugging document within the context of an application that also supports regular **TTEDoc** documents.

```
SFTypeList gtheTypes = {'TEXT'};
TDebugDoc *gdebugDoc = nil;
void main(void){
    // create an instance of TTEApp
    TTEApp theApp;
    // initialize the application
    if(theApp.InitApp()){
        gdebugDoc = MakeDebugDoc(&theApp);
        // open one window to start with,
        // unless we got files from the Finder
        if(! theApp.OpenDocFromFinder())
            theApp.OpenNewDoc();
        // run the event loop until user quits
        theApp.EventLoop();
        //now clean up
        theApp.CleanUp();
    }
}
```

▶ Sending Output to Debug Documents

Once the debugging document has been created and assigned to a global variable, as shown in the previous section, other parts of the program can use it just like a stream for character output. For example, you could override the **MakeDoc** member function of the application to write diagnostic information each time it made a new document, as follows. Notice that you must dereference the pointer to the **ostream** since the **insertion** operator works on stream objects rather than stream pointers.

```
TDoc * TTEApp::MakeDoc(SFReply * reply){
    TTEDoc * temp = new TTEDoc(GetCreator(),reply);
    *gdebugDoc << "Making a new document, address = "
        << (int)temp
        << endl;
    return temp;
}
```

That's all there is to it. The debugging document is a stream, so it embodies all the formatting capabilities of streams. But it is also a document, with that class's window management and file handling functionality. Best of all, it will coexist with other document types in an application so that you can add debugging support to any application based on **TApp** and **TDoc**.

▶ TDebugDoc Resources

The only new resource necessary for a **TDebugDoc** is a 'WIND' resource with an ID number of rDebugDoc (defined as a constant equal to 1000 in TDebugDoc.h). This window resource creates a window with no close box. It uses a different ID number from the default document window ID number so that they can coexist in the same application.

If you use **TDebugDoc** in your application, you must also add TDebugDoc.rsrc to your list of resources. See the files DebugTEApp.make and DebugTEApp.r in Appendix B for details.

▶ Summary

Multiple inheritance enables you to create classes that inherit members and member functions from more than one parent class. The example used here was a document class that was also a stream. The result is a document that can handle stream operations to produce debugging output in a Macintosh window. This can be a very useful class since the Macintosh doesn't normally support this sort of output.

This chapter also demonstrated how the **streambuf** class can be modified to redirect the output of a stream. Sending stream output to a text window is a useful technique, but it would be just as easy to create a **streambuf** that sent its output through the serial port to a debugging terminal. You might want to try overriding the **overflow** member function to achieve this effect.

This is the final chapter that will use **TApp** and **TDoc**. These classes and their derivatives have been very useful for learning the basics of C++, and they are quite useful for creating programs on the Macintosh. But they do have many shortcomings as the programs become more complex. The next chapter describes the MacApp class library. MacApp is a much more complete and fully developed class system. You will find that MacApp is a better way to go if you want to develop industrial-strength applications.

14 ▶ MacApp and PictView

This chapter reimplements the PictView program, originally developed in Chapter 11 using the **TApp** and **TScrollDoc** classes, using the MacApp class library. MacApp is a set of classes from Apple Computer that encapsulates much of the standard behavior of Macintosh applications and documents. In many ways, the application and document classes described in Chapters 5-13 are stripped-down versions of MacApp. But MacApp is a much more fully developed class library. It has been under development at Apple since 1984, so the class design has had a chance to mature and the bugs have been worked out through extensive testing and application use.

The PictView program developed in this chapter is named MAPictView to distinguish it from the program developed in Chapter 11. MAPictView has essentially the same functionality as the original PictView program, which allows you to compare the coding requirements for the **TApp-TDoc** class system and MacApp. You will find that MacApp and **TApp** and **TDoc** are similar in many ways and dissimilar in others. One thing you will find is that MacApp is more complicated than **TApp** or **TDoc**. This can be good and bad. **TApp** and **TDoc** were designed to be useful teaching tools, so many features that would distract from their teaching role were left out. MacApp, on the other hand, was designed to help programmers write robust application programs. MacApp has it all. As a consequence, MacApp is harder to learn than **TApp** or **TDoc**. But if your goal is to write the next blockbuster commercial application, MacApp is a good choice.

You might also be tempted to implement your own application and document class system, based on your experience with the classes in

Chapters 5-13. Although it is a lot of fun to create your own base classes, your time will probably be better spent learning a well-designed and well-tested class library like MacApp and then applying your creative efforts to building on top of those classes to address the needs of your specific application. Apple has devoted five or six years and untold engineering hours to developing and testing MacApp, so it makes sense to take advantage of all that work and experience.

The program developed in this chapter is very simple, and it does not begin to explain or utilize all the features of MacApp. It will, however, give you a taste of MacApp programming. The complete code is listed in Appendix B. If you are interested in learning more about MacApp, see the MacApp documentation from Apple or Wilson, Rosenstein, and Shafer's excellent book *Programming with MacApp*, (Addison-Wesley, 1990).

▶ Overview of MacApp

MacApp is a set of classes that provide a skeleton on which to build a Macintosh application. It has a document class and an application class, similar to the ones described in Chapters 5 and 6. The application class of Chapter 6 and the MacApp application class are similar in form and function. They are both responsible for dispatching events and managing a list of documents. The document class from Chapter 5 and the MacApp document class are quite different. While the **TDoc** class in Chapter 5 took care of managing the document data and displaying it, MacApp creates an additional class, **TView**, that takes the display responsibilities away from the document class. In MacApp, the document class is responsible for reading and writing data from the disk, and the view class is responsible for displaying the data.

MacApp uses the view class pervasively to implement display objects on the screen. Windows are derived from the view class. Controls such as buttons, scroll bars, and edit text boxes are also derived from the view class. Scrollers are special views that control scrolling images. A typical document window is made up of a window, a scroller, and an application-specific view to display the document data. The sample program described in this chapter will illustrate how to subclass the application, document, and view classes to create a new application.

In addition to the application, document, and view classes, MacApp provides many other support classes to make Macintosh programming easier. One of the biggest strengths of MacApp is a well-designed error handling mechanism. Although the program in this chapter will not explore the error-handling aspects of MacApp in any detail, it does call some of the error-catching functions provided by MacApp.

▶ MacApp and C++

The classes that make up the MacApp class library are written in Object Pascal. Apple had to make several additions to its version of C++ in order to make the use of MacApp (and other classes written in Object Pascal) possible from C++.

Object Pascal and C++ use very different mechanisms for calling virtual functions. Apple has added a C++ base class called **PascalObject** so that any C++ classes derived from **PascalObject** use the function-dispatching mechanisms of Object Pascal. This means that classes derived from **PascalObject** in C++ can be called from Object Pascal and that classes written in Object Pascal can be called from C++ as if they were derived from the C++ class **PascalObject.**

This provides an elegant way to access all of MacApp from your C++ programs. Apple has created a set of header files that declare the MacApp classes as descendents of **PascalObject**. All of the members and member functions in the MacApp classes are represented in the C++ header files. For example, the following Object Pascal code declares the interface to the class **Foo**.

```
Foo = OBJECT
   PROCEDURE Foo.Func1(var1 : INTEGER; var2 : Rect);
   FUNCTION  Foo.Func2(var1 : LongInt; var2 : INTEGER): INTEGER;
END;
```

In C++ , **Foo** would be declared in the following way.

```
class Foo : public PascalObject {
public:
   virtual pascal void Func1(short var1, Rect * var2);
   virtual pascal short Func2(long var1, short var2);
};
```

Notice how all the member functions from an Object Pascal class are defined as virtual when translated into a C++ declaration. Notice also that all the member functions are declared with the pascal keyword so that they will use the Pascal calling conventions for arguments and function results. Pascal data types must be converted to the equivalent C++ types: a Pascal INTEGER becomes a C++ short and so on.

Apple has also added the inherited keyword to C++ to give compatibility with a similar capability in Object Pascal. C++ classes that are derived from the base class **PascalObject** can use the inherited keyword to reference their parent class. In contrast, C++ classes that are not derived from **PascalObject** must explicitly use the parent class name to call the parent's member function.

▶ TPICTDocument

The **TPICTDocument** class is responsible for reading the data off the disk and making the window and view to display the data. It is declared as a derivative of the MacApp class **TDocument**, as follows.

```
class TPICTDocument : public TDocument {
   public:
      Handle   fPICTData;    // The PICT owned by the document
      Handle   fPICTHeader;  // header for PICT file
      // Initialization and freeing
      virtual pascal void IPICTDocument(void);
      virtual pascal void Free(void);

      // disable Save and SaveAs menu items
      virtual pascal void DoSetupMenus(void);
      // read the file
      virtual pascal void DoRead(short aRefNum,Boolean rsrcExists,
                                 Boolean forPrinting);
      // Making views and windows
      virtual pascal void DoMakeViews(Boolean forPrinting);
      // Inspecting
      virtual pascal void Fields(pascal void (*DoToField (
                                 StringPtr fieldName,
                                 Ptr fieldAddr,
                                 short fieldType,
                                 void *DoToField_StaticLink),
                                 void *DoToField_StaticLink);
};
```

TPICTDocument declares two new members, *fPICTData* and *fPICTHeader*, similar to the **TPICTDoc** class described in Chapter 11. The overridden member functions are discussed in the following sections.

▶ Initializing the Document

The **IPICTDocument** member function is defined to do the initialization specific to **TPICTDocument**. It begins by calling the **IDocument** member function to take care of all the default initialization for the **TDocument** class, and then it goes on to initialize its class-specific members. This member function serves the same purpose that the constructor and the **InitDoc** member function served in the **TPICTDoc** class described in Chapter 11. Although it is possible to define constructors for your MacApp classes written in C++, it is generally not

done — those portions of MacApp written in Object Pascal don't have constructors and it would be confusing to mix the two initialization styles. The code for **IPICTDocument** is shown as follows.

```
#pragma segment AOpen
pascal void TPICTDocument::IPICTDocument(void){
   // do the inherited stuff
   IDocument(kFileType,
      kSignature,
      kUsesDataFork,
      ! kUsesRsrcFork,
      kDataOpen,
      ! kRsrcOpen);
   // and now do our specific members
   fPICTData = nil;
   fPICTHeader = nil;
}
```

Key Point ▶

Notice the #pragma segment AOpen directive just before the beginning of the **IPICTDocument** definition. MacApp programs are conventionally segmented on a function-by-function basis to allow finer control of segment contents.

▶ Making the Views

The document object is responsible for creating the views that display the document's data. **TPICTDocument** calls the MacApp utility function NewTemplateWindow to read in a view resource and create the document window, the scroller, and **TPICTView**. NewTemplateWindow returns a pointer to the view object that represents the window (the window class is derived from the view class). **DoMakeViews** then calls the **FindSubView** member function for the window to extract the **TPICTView** from the window. Each view in the view resource is given a four-character identifier. **DoMakeViews** uses the identifier for the **TPICTView** view to retrieve a pointer to that view object.

DoMakeViews then creates a print handler, which is a MacApp class that knows how to print views. This is the only thing that you need to do to support printing in most MacApp programs. The print handler is initialized with a pointer to the view that it will print. Later, when the user chooses the Print menu command, the application will call the appropriate member functions for the print handler to print the view. Automatic support for multipage printing is included in the

print handler. Compare this to the printing code that was needed to enable printing in Chapter 11.

The code for **DoMakeViews** is shown as follows.

```
#pragma segment AOpen
pascal void TPICTDocument::DoMakeViews(Boolean /*forPrinting*/){
   TView *theWindow,*thePictView;
   TStdPrintHandler *aHandler;
   theWindow = NewTemplateWindow(kWindowRsrcID, this);
   FailNIL(theWindow);
   thePictView = theWindow->FindSubView('PicV');
   aHandler = new TStdPrintHandler;
   FailNIL(aHandler);
   aHandler->IStdPrintHandler(this,
                        thePictView,
                        ! kSquareDots,
                        kFixedSize,
                        ! kFixedSize);
   ShowReverted();
}
```

Notice how **DoMakeViews** calls the MacApp utility FailNIL each time to check that a new object was successfully created. This built-in error handling capability is one of MacApp's greatest strengths.

▶ Adjusting the Menus

TPICTDocument overrides the **DoSetUpMenus** member function so that the Save and SaveAs menu items in the File menu will be disabled. In MacApp, menu items are associated with command numbers, so the code to enable or disable individual items uses the command number for the item rather than the position number of the item within the menu. This is a great improvement over the methods used in Chapters 5-13, since it means that you can add or delete menu items without having to rewrite all the menu adjustment code.

The code for **DoSetUpMenus** is shown as follows. Notice that it calls the inherited version from its parent class before doing the class-specific processing.

```
#pragma segment ARes
pascal void TPICTDocument::DoSetupMenus(void){
   inherited::DoSetupMenus();
   Enable(cSaveAs, false);
   Enable(cSaveCopy, false);
}
```

▶ Reading 'PICT' Files

The document member function that is used to read the picture data off the disk is essentially the same as that described in Chapter 11. **DoRead** needs to allocate enough memory to hold the picture header and the actual bytes of the picture. One difference between the function shown here and the one in Chapter 11 is that this version of **DoRead** checks for errors with the MacApp utilities FailNIL and FailOSErr, which makes error-checking much cleaner and easier. The code for **DoRead** is shown as follows.

```
#pragma segment AReadFile
pascal void TPICTDocument::DoRead(short aRefNum,
                    Boolean /*rsrcExists*/,
                    Boolean /*forPrinting*/){
   long  pictSize;
   long  headerSize = kPictHeaderSize;
   // calculate size of file, subtract header size
   FailOSErr(GetEOF(aRefNum, &pictSize));
   pictSize = pictSize - kPictHeaderSize;
   // allocate memory for header and pict
   fPICTHeader = NewPermHandle(kPictHeaderSize);
   FailNIL(fPICTHeader);
   fPICTData = NewPermHandle(pictSize);
   FailNIL(fPICTData);
   // now read header and pict
   HLock(fPICTHeader);
   FailOSErr(FSRead(aRefNum, &headerSize, *fPICTHeader));
   HUnlock(fPICTHeader);
   HLock(fPICTData);
   FailOSErr(FSRead(aRefNum, &pictSize, *fPICTData));
   HUnlock(fPICTData);
}
```

▶ Closing the Document

TPICTDocument overrides the **Free** member function so that it can deallocate the memory for the picture and the picture header when the document is closed. This is similar to the work done by the destructor in the document class in Chapter 11. But destructors are rarely used for MacApp classes written in C++. The code for **Free** is shown as follows. Notice that it calls **inherited::Free** to make sure that its parent classes also have a chance to clean up and deallocate their memory.

```
# pragma segment AClose
pascal void TPICTDocument::Free(void){
   if(fPICTData != nil){
      DisposHandle(fPICTData);
      fPICTData = nil;
   }
   if(fPICTHeader != nil){
      DisposHandle(fPICTHeader);
      fPICTHeader = nil;
   }
   inherited::Free();
}
```

▶ Inspecting the Document

MacApp includes a facility for inspecting objects while an application is running. You can choose to build your MacApp program with debugging support included during development and then build it with all the debugging code stripped out for final production, as described in a later section of this chapter. When the debugging support is included, an extra menu is included that lets you to open inspector windows. Figure 14-1 shows an inspector window examining the contents of a **TPICTDocument** object for an open document.

You can see from Figure 14-1 that the inspector lists all the members for the selected object. Notice that the members for each parent class are also shown, all the way back to the ultimate parent class.

In order to allow your class to be inspected in this way, you must override the **Fields** member function for the class. **Fields** receives a pointer to a function as its first argument. The function, called DoToField, is responsible for drawing the information in the inspector window. **Fields** first passes the name of the class to DoToField. **Fields** then passes information about each of its members to this function. The class that is being inspected doesn't need to know anything about how the inspector operates. It only supplies information about its name and members to DoToField. After the class has called DoToField for all its members, it calls **inherited::Fields** to invoke the **Fields** member function for its parent class. This invocation chain continues up through the parent lineage, resulting in the display of all members, as shown in Figure 14-1.

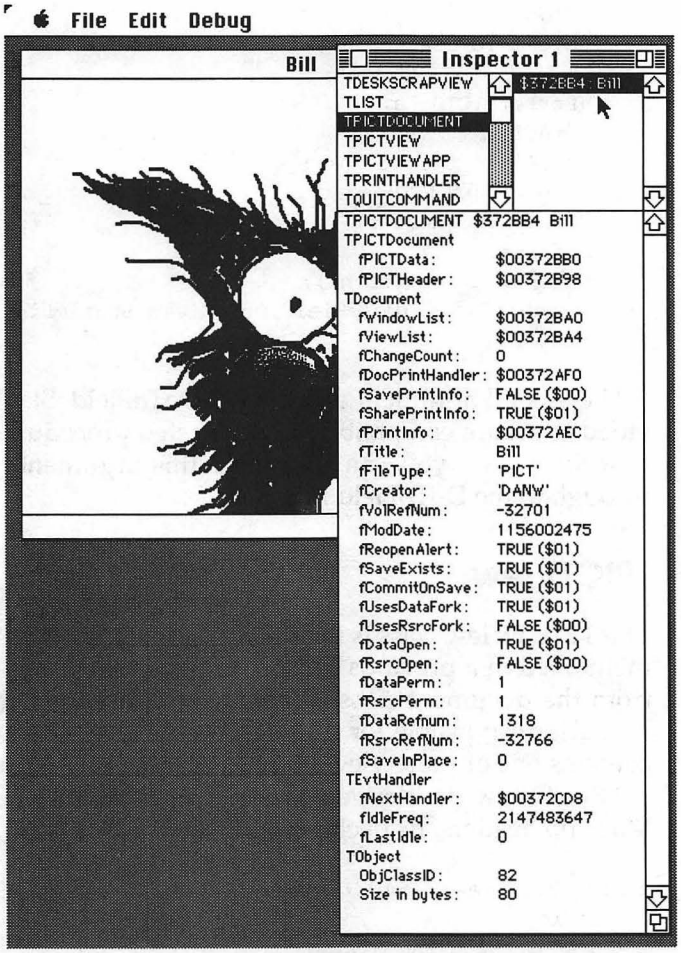


Figure 14-1. An Inspector Window with a TPICTDocument

The code for **TPICTDocument::Fields** is shown as follows.

```
#pragma segment AFields
pascal void TPICTDocument::Fields(pascal void (*DoToField)(
                        StringPtr fieldName,
                        Ptr fieldAddr,
                        short fieldType,
                        void *DoToField_StaticLink),
                        void *DoToField_StaticLink){
 DoToField("\pTPICTDocument",
     nil,
```

```
    bClass,
    DoToField_StaticLink);
 DoToField("\pfPICTData",
    (Ptr)&fPICTData,
    bHandle,
    DoToField_StaticLink);
 DoToField("\pfPICTHeader",
    (Ptr)&fPICTHeader,
    bHandle,
    DoToField_StaticLink);
 inherited::Fields(DoToField,DoToField_StaticLink);
}
```

The second argument to **Fields** is DoToField_StaticLink. This is provided to assure compatibility with nested procedures written in Object Pascal. In C++ you can disregard this argument, simply passing it through to the DoToField function.

▶ TPICTView

The **TPICTView** class is the primary view that displays the picture. As mentioned in a previous section, MacApp separates the display chores from the document class. Whereas in Chapter 11 the document class was also responsible for drawing the picture, in MacApp a document contains one or more view objects that take care of the drawing.

TPICTView is derived from **TView**, which is defined in the MacApp headers. Its declaration is shown as follows.

```
class TPICTView : public TView {
  public:
     // drawing and sizing
     virtual pascal void CalcMinSize(VPoint *minSize)
     virtual pascal void Draw(Rect *area);
};
```

From the declaration you can see that **TPICTView** overrides two member functions. These functions are discussed separately in the following sections.

You might notice that this class does not override the **Fields** member function, as in the previous two classes. This is because **TPICTView** does not define any new members, so its parent class's **Fields** will be sufficient to permit the class to be inspected. You only need to override **Fields** when you define new members for a class.

▶ Calculating View Dimensions

TPICTView is contained within a window. It is also contained within a special MacApp view class called a scroller. The scroller is responsible for managing the scroll bars and performing the coordinate system manipulations that enable the image in the view to scroll. MacApp uses a coordinate offset mechanism very similar to that used by **TScrollDoc**, as explained in Chapter 10. MacApp, however, develops those ideas further and encapsulates them, separate from the document class and the view class, in the scroller class. Scrollers are one of the trickiest parts of MacApp to understand, but fortunately it is seldom necessary to modify scrollers directly so they are actually quite easy to use.

TPICTView must override the **CalcMinSize** member function to tell the scroller how big the view's image is. This function is similar to the **GetVertSize** and **GetHorizSize** member functions for **TScrollDoc**, as explained in Chapter 10. As you saw in the original PictView program in Chapter 11, the dimensions of the image is extracted from the picFrame field of the Picture data structure. **CalcMinSize** uses the *fDocument* member (which is declared as a part of the base class **TView**) to get access to the document and then retrieves the picture from the *fPICTData* member of the document. Because *fDocument* is declared as a pointer to a **TDocument** object, **CalcMinSize** must typecast it to be a pointer to a **TPICTDocument** so that it can access the document's *fPICTData* member, which is specific to the **TPICTDocument** class. **CalcMinSize** then looks at the picFrame field of the Picture and calculates the width and height of the image. These dimensions are passed back to the caller by setting the VPoint argument to **CalcMinSize**, shown as follows.

```
#pragma segment ARes
pascal void TPICTView::CalcMinSize(VPoint *minSize){
   short hSize,vSize;
   TPICTDocument * PICTDoc = (TPICTDocument *)fDocument;
   if(PICTDoc->fPICTData) {
      hSize = (**((PicHandle)PICTDoc->fPICTData)).picFrame.right
              - (**((PicHandle)PICTDoc->fPICTData)).picFrame.left;
      vSize = (**((PicHandle)PICTDoc->fPICTData)).picFrame.bottom
              - (**((PicHandle)PICTDoc->fPICTData)).picFrame.top;
      SetVPt(minSize, hSize,vSize);
   } else
      SetVPt(minSize,0,0);
}
```

The scroller calls **CalcMinSize** to determine the minimum and maximum values for the scroll bars in much the same way that **TScrollDoc** used **GetVertSize** and **GetHorizSize** in Chapters 10 and 11.

▶ Drawing the View

Because views are responsible for displaying data on the screen, **TPICTView** must override the **Draw** member function. The code for **Draw** is essentially the same as that shown in the PictView program in Chapter 11. Just as you saw in **CalcMinSize**, the view must go to the document to get the picture data. **Draw** passes the picture to the toolbox function DrawPicture, as shown here.

```
#pragma segment ARes
pascal void TPICTView::Draw(Rect * /*area*/){
   TPICTDocument * PICTDoc = (TPICTDocument *)fDocument;
   if(PICTDoc->fPICTData)
      DrawPicture((PicHandle)(PICTDoc->fPICTData),
              &(**((PicHandle)PICTDoc->fPICTData)).picFrame);
}
```

▶ TPICTViewApp

The third class that is necessary for the MAPictView program is an application class derived from the MacApp class **TApplication**. It is similar in functionality to the **TApp** class developed in Chapter 6. The declaration for **TPICTViewApp** is shown as follows. The individual member functions that are defined in this class are discussed in the following sections.

```
class TPICTViewApp: public TApplication {
   public:
   // Initialize the Application
   virtual pascal void IPICTViewApp(void);
   // Launches a TPICTDocument
   virtual pascal struct TDocument *DoMakeDocument(CmdNumber
      itsCmdNumber);
   // disable the new menu item
   virtual pascal void DoSetupMenus(void);
   // Prevents empty document on launch
   virtual pascal void OpenNew(CmdNumber itsCmdNumber);
};
```

▶ Initializing the Application

TPICTViewApp defines a new initialization function, **IPICTViewApp**, to initialize the application and register the **TPICTView** class so that it won't be stripped out by the linker. The default application initialization is invoked by calling the **IApplication** member function. **IPICTViewApp** then initializes the global variable *gStaggerCount*, which is used to control the stagger position of new windows as they are opened.

Next, **IPICTViewApp** calls the **new** operator to create a **TPICTView** object. This is necessary because the linker will automatically strip out all code for classes that don't actually get created by the program. If the linker doesn't see an object of a particular class created with the **new** operator in your code, it assumes that the class isn't used and strips out the code for the class during the link process. This can be very useful when you are linking with large libraries that contain many classes, most of which you won't use. But it can also strip out needed code when objects of a class are created implicitly so that the linker can't detect their use.

Objects belonging to the other two classes that are unique to this program, **TPICTDocument** and **TPICTViewApp**, are created with the **new** operator in other parts of the program, but a **TPICTView** object is never created directly. Instead, it is automatically created when NewTemplateWindow reads in the resource template for the window and its views. Therefore, it is necessary to include code that explicitly creates a **TPICTView** object with the **new** operator so that the linker will not strip out the code for the class. You don't have to delete the object since the creation code is contained in a conditional block that will never be executed.

This method of protecting classes from the linker may change in future versions of MacApp or the linker, but for now it is a necessary kludge. See the MacApp documentation for more details on this problem. The code for **IPICTViewApp** is shown as follows.

```
#pragma segment AInit
pascal void TPICTViewApp::IPICTViewApp(void){
   IApplication(kFileType);
   gStaggerCount = 0;

   // So the linker doesn't dead strip class info
   if( gDeadStripSuppression) {
      TPICTView *aPICTView;
```

```
        aPICTView = new TPICTView;

    }
}
```

▶ Making a Document

Just as the **TApp** class described in Chapter 6 had the member function **MakeDoc** to create new documents, so the MacApp class **TApplication** has the **DoMakeDocument** member function. **TPICTViewApp** overrides **DoMakeDocument** to create and initialize a **TPICTDocument**. The new document is responsible for creating its window and views, as explained in a previous section. The code for **DoMakeDocument** is shown as follows.

```
#pragma segment AOpen
pascal struct TDocument *TPICTViewApp::DoMakeDocument(CmdNumber
                    /*itsCmdNumber*/){
   TPICTDocument *aPICTDocument;
   aPICTDocument = new TPICTDocument;
   FailNIL(aPICTDocument);
   aPICTDocument->IPICTDocument();
   return aPICTDocument;
}
```

Normally, MacApp creates a new, blank document at program startup by calling the **OpenNew** member function. Since the MAPictView program can work only with existing files, we don't want to open a blank document at startup. **TPICTViewApp** overrides **OpenNew** so that it displays a standard file dialog to let the user open an existing file instead of a blank document. The code for **OpenNew** is shown as follows.

```
#pragma segment AOpen
pascal void TPICTViewApp::OpenNew( CmdNumber /*itsCmdNumber*/){
   AppFile anAppFile;
   if (ChooseDocument( cFinderOpen, &anAppFile ))
      OpenOld( cFinderOpen, &anAppFile );
}
```

▶ Adjusting the Menus

Just as the document class overrode **DoSetUpMenus** to disable the Save and SaveAs menu items, so the application class overrides **DoSetUpMenus** so that the New menu item is disabled. The New menu item must be disabled for this program because it can only read existing 'PICT' files, not create new ones. The application class is responsible for disabling New since the application is the class that would respond to the New menu command, just as the document class must disable Save and SaveAs since it responds to those menu commands. The code that **TPICTViewApp** uses to override **DoSetUpMenus** is shown as follows. Notice that the inherited version of the function is called before the class-specific operations are performed.

```
#pragma segment ARes
pascal void TPICTViewApp::DoSetupMenus(void){
 inherited::DoSetupMenus();
 Enable(cNew,false);
}
```

▶ The MAPictView Main Program

You must write a main function to create the application object, initialize it, and start it running. Before creating the application object, you must initialize the toolbox and MacApp.

One interesting function that MacApp provides is ValidateConfiguration. You can specify many different environmental requirements for your program by setting flags during the build process. For example, your application might require Color QuickDraw to function properly. ValidateConfiguration checks the current execution environment against those requirements and returns true only if all those requirements are met. If the environment is unsuitable to the program, ValidateConfiguration will return false, an error alert will be displayed, and the program will terminate.

The code for the main function for the MAPictView program is shown as follows. It is similar in intent to the main functions described in Chapters 7-13.

```
TPICTViewApp *gPICTViewApp;
#pragma segment Main
void main(void){
   InitToolBox();// Essential toolbox and utilities initialization
   // Make sure we can run
   if(ValidateConfiguration(&gConfiguration)){
      InitUMacApp(8)  //Initialize MacApp; 8 calls to MoreMasters
      InitUPrinting();// Initialize the Printing unit
      gPICTViewApp = new TPICTViewApp;

      FailNIL(gPICTViewApp);
      gPICTViewApp->IPICTViewApp(); // Initialize the application
      gPICTViewApp->Run();    // Run the application
   } else
      StdAlert(phUnsupportedConfiguration);
}
```

▶ MAPictView Resources

As mentioned in a previous section, MacApp allows you to define resources that describe an arrangement of views in a window. This resource can then be loaded and all the views initialized with a single call to NewTemplateWindow. This makes it much easier to create windows with multiple views.

The ViewEdit program that comes with MacApp is an interactive resource editor, much like ResEdit, except that it knows about view resources and about the predefined view classes that are part of the MacApp class library. You can use ViewEdit to create your view resources and then include those resources in your applications program.

The view resource for the MAPictView program includes three views. The first view is the window. Inside the window view is the scroller view, and inside the scroller view is the **TPICTView** view. The following resource definition defines these three views in relation to each other. Each view definition contains information about the size and location of the view, the four-character identifier for the view and its parent view, and other class-specific initialization information.

```
resource 'view' (kWindowRsrcID, purgable) {
   {
   root, 'WIND', { 50, 40 }, { 250, 450 },
      sizeVariable, sizeVariable, shown, enabled,
   Window {
      "",
      zoomDocProc, goAwayBox, resizable,
```

```
      modeless, ignoreFirstClick, freeOnClosing,
      disposeOnFree, closesDocument, openWithDocument,
      dontAdaptToScreen, stagger,
      forceOnScreen, dontCenter, noID, ""
};
   'WIND', 'SCLR', { 0, 0 }, { 250-kSBarSizeMinus1,
   450-kSBarSizeMinus1 },
   sizeRelSuperView, sizeRelSuperView, shown, enabled,
      Scroller {
         "",
         vertScrollBar, horzScrollBar, 0, 0, 16, 16,
         vertConstrain, horzConstrain, { 0, 0, 0, 0 }
      };
   'SCLR', 'PicV', { 0, 0 }, { 116, 1020 },
         sizeVariable, sizeVariable, shown, enabled,
      View
         {"TPICTView"}
   }
};
```

The complete resource definition file for MAPictView is listed in Appendix B in the file MAPictView.r. It includes many default resources from MacApp and defines some application-specific resources.

▶ Building MAPictView

MacApp includes a sophisticated build system so that you don't have to write a makefile for most MacApp projects. The key to using MacApp's built-in build capabilities is to name your source files in accordance with Apple's file naming conventions, as shown in Table 14-1.

Assuming that the name of your application is MAPictView, you would create the following files.

Table 14-1. MacApp Naming Conventions

File Name	*File Contents*
UMAPictView.h	class declarations (interface)
UMAPictView.cp	definition of member functions (implementation)
MMAPictView.cp	main function
MAPictView.r	resource definitions
MAPictView	application

If you name your files in this way, the default build instructions in the MABuild tool will automatically build your program. You can invoke MABuild directly from the MPW command line, or you can use the MPW Build... menu command, supplying the name of the application that you want to build. In either case, MABuild will take care of examining the dependencies and rebuilding your application.

Starting with version 2.0b12, MABuild does not include debugging code in your MacApp application by default. The debugging code allows you to use the inspector and the MacApp debugger, but it adds about 150 Kilobytes to your application and slows down its overall performance. To build a version of your application with the debugging code, add the -debug option when you invoke MABuild.

▶ Summary

This chapter has touched on some of the fundamental aspects of programming with MacApp and C++. MacApp is a fully developed class library designed to make it easier to write Macintosh programs. It provides base classes that you can subclass to create your own applications.

The MAPictView program developed in this chapter is similar to the program described in Chapter 11, and the application classes in this chapter and in Chapter 11 are essentially the same. But in Chapter 11 the document class handled all the data management and display tasks; in MacApp the document class gives up its display chores to the view class.

Many important aspects of MacApp programming are not covered in this chapter. The most important of these are command objects, which provide an elegant mechanism for doing and undoing commands in response to user actions. You are encouraged to explore MacApp further. It is a rich storehouse of Macintosh programming gems and a great foundation on which to build your programs. Taking advantage of this accumulated wisdom is the biggest benefit of object-oriented programming.

Afterword by Scott Knaster

A "Cult Classic" is a work of art that's appreciated and revered by a (usually) small but devoted band of followers. Cult classics often are created as labors of love. Although their commercial success may not be great, devotees of cult classics are fanatical.

The strange little community of Macintosh programmers has produced a few cult classic books in the years since the Macintosh was introduced. Two of the greatest of these are Dan Weston's famous assembly language books. I remember reading these books back when they came out and being astonished to discover that there was someone outside the cubicles of Apple who knew so much about what was going on with such an arcane subject. I also loved Dan's conversational style of presenting very technical material.

When *Macintosh Inside Out* was started, Dan was one of the first authors we talked to about writing a book for the new series. For some reason, Dan has always refused to leave his beautiful home in the idyllic northwest for the traffic-choked, smog-encrusted, overpriced hell of Silicon Valley, so a *Macintosh Inside Out* book was an ideal chance for Dan to do something without having to leave his favored environment.

It turned out that he was working with C++, a topic we definitely wanted to cover in the series. I was really looking forward to seeing Dan's explanation of what the heck is going on in C++.

The book Dan wrote (this one) turned out to be much more than an overview of C++. As I read Dan's manuscript, I was amazed to discover that he had done something that I'd never seen done in a book before: He had taken the time to implement a real live class library,

and he had taken the reader along on the journey, acting as a tour guide and explaining what he was doing along the way. This is great! It's wonderful to get your feet wet (to borrow one of Dan's metaphors) in object programming by watching as a class library is designed by someone who really knows how to do it.

Dan's writing skill and attention to detail are going to make this book another cult classic, but the importance of C++ is going to make this particular cult very large. I hope you've enjoyed learning from Dan about what goes on inside the mind of an object designer and that you have fun with C++.

Scott Knaster
Macintosh Inside Out Series Editor

Appendix A
Think C 4.0 and C++

About the same time that Apple released its version of MPW C++, Symantec released Think C version 4.0. This version of Think C included many object-oriented extensions, but it is not a full C++ implementation. This appendix discusses some of the similarities and differences between Think C 4.0 and MPW C++.

Think C: C+-

Think C includes many object-oriented programming features that are similar to C++, but it is not C++. In particular, several key C++ features are missing from Think C, as shown in the following list.

- No double-slash (//) comments.
- No const definitions.
- No default argument values.
- New and delete are not operators in Think C. They are functions, so parentheses are always required for arguments to new and delete in Think C.
- No protected and private protection levels in classes. All members and member functions are public.
- No constructors and destructors.
- No multiple inheritance.

Think C has no virtual keyword. All member functions are virtual automatically. This can cause problems if you are trying to port code from Think C to MPW C++. One solution is to make the

```
#define virtual
```

definition in your Think C files and then add virtual to all your Think C member function declarations so that when they are moved to C++ you will be able to override them in derived classes.

Porting code from Think C to MPW C++ is not easy. Because Think C is a subset of C++, it is much easier to port code from Think C to MPW C++ than the other way around. But you must be careful to pay attention to the small differences between Think C and C++. For example, an int is 16 bits in Think C and 32 bits in C++. Another potential problem is that Think C uses different names for some of its Macintosh include files.

Think C's biggest advantage over MPW is that Think C compiles and links code much faster than MPW does. Think C was originally called Lightspeed C for good reason. The ability to turn code around in a hurry is especially valuable when you are learning a language.

Think C has excellent debugging support with its source level debugger. The Think C debugger is much easier to use than the SADE debugger in MPW C++.

All-in-all, Think C provides a great way to get into object-oriented programming. If you can get by without some of the C++ features that Think C leaves out, you should find that Think C is more than adequate for all your programming needs.

▶ The Think C Class Library

You get a big bonus with Think C: the Think Class Library, written by Greg Dow. This class library contains numerous classes that help you write Macintosh applications. It is like a simplified MacApp. Actually, because it is somewhat simpler than MacApp, it may be preferable to MacApp for most programmers. The Think Class Library is a great way to make a Macintosh program. Several nontrivial example programs are also provided to show how to use the classes in the library.

Appendix B
Source Code Listings

Appendix B is a chapter-by-chapter compilation of all source code for the programs contained in this book. (You can also order a source code disk of these program listings; please see the tear-out order card at the end of this book.) The following list identifies each program included in Appendix B, followed by the page number in parentheses on which each program begins.

▶ Chapter 4 Programs:

▶ Chapter 5 Programs:

▶ Chapter 6 Programs:

▶ Chapter 7 Programs:

▶ Chapter 8 Programs:

▶ Chapter 9 Programs:

▶ Chapter 10 Programs:

▶ Chapter 11 Programs:

▶ Chapter 12 Programs:

▶ Chapter 13 Programs:

▶ Chapter 14 Programs:

```
// HelloWorld.cp
// A very simple C++ program
// This program prints the words "hello world" to standard output
// January 1990, Dan Weston

#include <iostream.h>

void main(void){

    cout << "hello world\n";

}
```

```
#    File:        HelloWorld.make
#    Target:      HelloWorld
#    Sources:     HelloWorld.cp
#    Created:     Monday, January 29, 1990 9:36:51 AM


OBJECTS = HelloWorld.cp.o



HelloWorld ƒƒ HelloWorld.make {OBJECTS}
    Link -w -c 'MPS ' -t MPST ∂
        {OBJECTS} ∂
        "{CLibraries}"CSANELib.o ∂
        "{CLibraries}"Math.o ∂
        "{CLibraries}"CplusLib.o ∂
        #"{CLibraries}"Complex.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Stubs.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{Libraries}"Interface.o ∂
        "{Libraries}"ToolLibs.o ∂
        -o HelloWorld
HelloWorld.cp.o ƒ HelloWorld.make HelloWorld.cp
     CPlus  HelloWorld.cp
```

```
//////////////////////////////////////////////////////////////////
//
// fixcom.cp
//
// Changes C++ style comments to C comments
//
// Uses cin and cout streams
//
// invoke with redirection from MPW, such as
//
// fixcom < foo.cp > foo.fixed
//
// ©1990 Dan Weston, all rights reserved
//
//////////////////////////////////////////////////////////////////


#include <iostream.h>

int main(void){

    char c;
    char nextc;

    while (cin.get(c)){
        if (c != '/'){
            // most chars just pass right through filter
            cout << c;
        } else {
            // this may be a double slash comment...
            // check next char following first '/'
            cin.get(nextc);
            if (nextc != '/'){
                // not a double slash comment,
                // just output the '/' and the following char
                cout << c << nextc;

            } else {
                // it is a double slash comment,
                // substitute opening C comment
                cout << '/' << '*';

                // pass chars through until end of line
                cin.get(c);
                while (c != '\n'){
                    cout << c;
                    cin.get(c);
                }

                // now insert a closing comment
                cout << ' ' << '*' << '/';
```

```
                // and send the newline char out too
                cout << c;

            }  // end nextc != '/' else

        }   // end of c != '/' else

    }   // end while

    // make sure all output is flushed
    cout << flush;
    return 0;
}
```

fixcom.make **Page 1**

```
#   File:       fixcom.make
#   Target:     fixcom
#   Sources:    fixcom.cp
#   Created:    Monday, January 29, 1990 10:49:49 AM


OBJECTS = fixcom.cp.o



fixcom ƒƒ fixcom.make {OBJECTS}
    Link -w -c 'MPS ' -t MPST ∂
        {OBJECTS} ∂
        "{CLibraries}"CSANELib.o ∂
        "{CLibraries}"Math.o ∂
        "{CLibraries}"CplusLib.o ∂
        #"{CLibraries}"Complex.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Stubs.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{Libraries}"Interface.o ∂
        "{Libraries}"ToolLibs.o ∂
        -o fixcom
fixcom.cp.o ƒ fixcom.make fixcom.cp
     CPlus  fixcom.cp
```

```
////////////////////////////////////////////////////////////////////////
//
// TTool.h
//
// A simple class for writing MPW tools
//
// Include TTool.h in your tool program,
// derive a class from TTool,
//  Override SetOption to process arguments that begin with '-'
//  Override HandleArg to process all other arguments
//  Override DoWork to do the actual work of the tool
//
//  ©1990 Dan Weston, all rights reserved
//
////////////////////////////////////////////////////////////////////////

#include <Quickdraw.h>
#include <Fonts.h>
#include <CursorCtl.h>
#include <iostream.h>
#include <fstream.h>
#include <FCntl.h>

////////////////////////////////////////////////////////////////////////
//
// class TTool
//
////////////////////////////////////////////////////////////////////////
class TTool {

protected:

    int fArgc;
    int fCurrentArg;
    char ** fArgv;
    char * fProgName;
    char * fNextArg;

public:

    virtual void ITool(int argc, char* argv[]);
    virtual int Run(void);

protected:

    virtual char* GetNextArg(void);
    virtual fstream * MakeStream(char * fileName,int permission);
    virtual int ParseArguments(void);

    virtual int SetOption(char * /*option*/){return 1;}
    virtual int HandleArg(char * /*arg*/){return 1;}
```

```
    virtual int DoWork(void){return 0;}
};

////////////////////////////////////////////////////////////////////////
//
// TTool::ITool
//
////////////////////////////////////////////////////////////////////////
void TTool::ITool(int argc, char* argv[]){

    fArgc = argc;
    fArgv = argv;

    fProgName = *fArgv++;
    fCurrentArg = 1;
    fNextArg = 0;

    // just in case you want to use Quickdraw
    InitGraf(&qd.thePort);

    // MPW tool documentation says to call this next function
    SetFScaleDisable(true);

    InitCursorCtl(nil);
    SpinCursor(1);
}
////////////////////////////////////////////////////////////////////////
//
// TTool::Run
//
////////////////////////////////////////////////////////////////////////
int TTool::Run(void){

    if(ParseArguments())
        return DoWork();
    else
        return 1;
}
////////////////////////////////////////////////////////////////////////
//
// TTool::ParseArguments
//
////////////////////////////////////////////////////////////////////////
int TTool::ParseArguments(void){

    char * arg;
    while((arg = GetNextArg()) != 0){
        if(*arg == '-') {
            if(SetOption(arg) == 0)
                return 0;
        } else {
```

```
            if(HandleArg(arg) == 0)
                return 0;
        }
    }
    // signal success
    return 1;
}
////////////////////////////////////////////////////////////////////////
//
// TTool::GetNextArg
//
////////////////////////////////////////////////////////////////////////
char* TTool::GetNextArg(void){

    if(fCurrentArg++ < fArgc){
        fNextArg = *fArgv++;
        return fNextArg;
    } else
        return 0;
}
////////////////////////////////////////////////////////////////////////
//
// TTool::MakeStream
//
////////////////////////////////////////////////////////////////////////
fstream * TTool::MakeStream(char * fileName,int permission){

    const int BUFFSIZE = 1024;

    int fd = open(fileName,permission);
    if(fd == EOF){
        fd = creat(fileName);
    }
    if(fd != EOF){
        char * buff = new char[BUFFSIZE];
        if(buff == 0){
            cerr << "### " << "error making stream\n";
            return 0;
        }
        fstream * fs = new fstream(fd,buff,BUFFSIZE);
        if(fs == 0){
            cerr << "### " << "error making stream\n";
            return 0;
        }
        return fs;
    }
    cerr << "### " << "error opening file " << fileName <<"\n";
    return 0;   // failed to open file
}
```

```
////////////////////////////////////////////////////////////////
//
// fixcom2.cp
//
// Changes C++ style comments to C comments
//
// Uses MPW input and output redirection
//
// invoke with redirected input file and output file names
//
// fixcom2 < foo.cp > foo.c
//
// ©1990 Dan Weston, all rights reserved
//
////////////////////////////////////////////////////////////////

#include "TTool.h"

class TFixComment2 : public TTool {

public:

    int DoWork(void)
        {return Filter(cin,cout);}

protected:

    int Filter(istream& in,ostream& out);

};


int TFixComment2::Filter(istream& in,ostream& out){

    char c;
    char nextc;

    while (in.get(c)){
        SpinCursor(1);
        if (c != '/'){
            // most chars just pass right through filter
            out << c;
        } else {
            // this may be a double slash comment...
            // check next char following first '/'
            in.get(nextc);
            if (nextc != '/'){
                // not a double slash comment,
                // just output the '/' and the following char
                out << c << nextc;
```

```
            } else {
                // it is a double slash comment,
                // substitute opening C comment
                out << '/' << '*';

                // pass chars through until end of line
                in.get(c);
                while (c != '\n'){
                    out << c;
                    in.get(c);
                }

                // now insert a closing comment
                out << ' ' << '*' << '/';

                // and send the newline char out too
                out << c;

            }  // end nextc != '/' else

        }   // end of c != '/' else

    }   // end while

    // make sure all output is flushed
    out << flush;
    return 0;
}


int main(int argc, char* argv[]){

    TFixComment2 fixComTool;

    fixComTool.ITool(argc,argv);
    return fixComTool.Run();

}
```

fixcom2.make **Page 1**

```
#   File:       fixcom2.make
#   Target:     fixcom2
#   Sources:    fixcom2.cp
#   Created:    Friday, January 19, 1990 11:58:21 AM


OBJECTS = fixcom2.cp.o



fixcom2 ƒƒ fixcom2.make {OBJECTS}
    Link -w -c 'MPS ' -t MPST ∂
        {OBJECTS} ∂
        "{CLibraries}"CSANELib.o ∂
        "{CLibraries}"Math.o ∂
        "{CLibraries}"CplusLib.o ∂
        #"{CLibraries}"Complex.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Stubs.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{Libraries}"Interface.o ∂
        "{Libraries}"ToolLibs.o ∂
        -o fixcom2
fixcom2.cp.o ƒ fixcom2.make fixcom2.cp TTool.h
     CPlus  fixcom2.cp
```

```
////////////////////////////////////////////////////////////////
//
// fixcom3.cp
//
// Changes C++ style comments to C comments
//
// Uses MPW command line for input and output file
//
// invoke with input file name and -o output file names:
//
// fixcom2 foo.cp -o foo.c
//
// if input or output file is not specified, standard in or out is used
//
// ©1990 Dan Weston, all rights reserved
//
////////////////////////////////////////////////////////////////

#include "TTool.h"

class TFixComment3 : public TTool {

protected:
    istream *fIn;
    ostream *fOut;

public:
    void ITool(int argc, char* argv[]);

    int DoWork(void);


protected:

    int SetOption(char * option);
    int HandleArg(char * arg);

    int Filter(istream& in,ostream& out);

};

void TFixComment3::ITool(int argc, char* argv[]){

    // do the inherited stuff first
    TTool::ITool(argc,argv);

    // hook up default input and output
    fIn = &cin;
    fOut = &cout;

}
```

```
int TFixComment3::DoWork(void){
    return Filter(*fIn,*fOut);
}

int TFixComment3::SetOption(char * option){

    char * arg;

    if(*(++option) == 'o'){
        // get the output file name
        if((arg = GetNextArg()) != 0){
            // open the file and create a stream for it
            fstream * fs = MakeStream(arg,O_WRONLY);
            if(fs != 0){
                fOut = fs;
                return 1;
            } else
                return 0;
        } else{
            cerr << "### " << " missing file name\n" ;
            return 0;
        }
    } else {
        cerr << "### " << option << " is not a valid option\n";
        return 0;
    }
}

int TFixComment3::HandleArg(char * arg){

    // open the file and create a stream for it
    fstream * fs = MakeStream(arg,O_RDONLY);
    if(fs != 0){
        fIn = fs;
        return 1;
    } else
        return 0;
}


int TFixComment3::Filter(istream& in,ostream& out){

    char c;
    char nextc;

    while (in.get(c)){
        SpinCursor(1);
        if (c != '/'){
            // most chars just pass right through filter
            out << c;
```

```
        } else {
            // this may be a double slash comment...
            // check next char following first '/'
            in.get(nextc);
            if (nextc != '/'){
                // not a double slash comment,
                // just output the '/' and the following char
                out << c << nextc;

            } else {
                // it is a double slash comment,
                // substitute opening C comment
                out << '/' << '*';

                // pass chars through until end of line
                in.get(c);
                while (c != '\n'){
                    out << c;
                    in.get(c);
                }

                // now insert a closing comment
                out << ' ' << '*' << '/';

                // and send the newline char out too
                out << c;

            }  // end nextc != '/' else

        }   // end of c != '/' else

    }   // end while

    // make sure all output is flushed
    out << flush;
    return 0;
}

int main(int argc, char* argv[]){

    TFixComment3 fixComTool;

    fixComTool.ITool(argc,argv);
    return fixComTool.Run();

}
```

```
#   File:       fixcom3.make
#   Target:     fixcom3
#   Sources:    fixcom3.cp
#   Created:    Friday, January 19, 1990 11:58:21 AM


OBJECTS = fixcom3.cp.o



fixcom3 ƒƒ fixcom3.make {OBJECTS}
    Link -w -c 'MPS ' -t MPST ∂
        {OBJECTS} ∂
        "{CLibraries}"CSANELib.o ∂
        "{CLibraries}"Math.o ∂
        "{CLibraries}"CplusLib.o ∂
        #"{CLibraries}"Complex.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Stubs.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{Libraries}"Interface.o ∂
        "{Libraries}"ToolLibs.o ∂
        -o fixcom3
fixcom3.cp.o ƒ fixcom3.make fixcom3.cp TTool.h
     CPlus  fixcom3.cp
```

```
//////////////////////////////////////////////////////////////////////////
//
// This is the generic document object
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////////

#ifndef TDoc_Defs
#define TDoc_Defs

// Include necessary interface files
#include <Types.h>
#include <Quickdraw.h>
#include <Windows.h>
#include <Packages.h>
#include <Menus.h>

//////////////////////////////////////////////////////////////////////////
//
// constants
//
//////////////////////////////////////////////////////////////////////////

const short kScrollBarWidth = 16;
const short kScrollBarPos = kScrollBarWidth -1;

const short rErrorAlert = 255;
const short rDocErrorStrings = 255;
const short sNoMem = 1;
const short sFileOpen = 2;
const short sUnknownErr = 3;

const short rGenericDoc  = 1000;

const short rWantToSave = 500;
const short iYes = 1;
const short iNo = 3;
const short iCancel = 2;

//////////////////////////////////////////////////////////////////////////
//
// utility routines
//
//////////////////////////////////////////////////////////////////////////

// Define HiWrd and LoWrd function inline for efficiency
inline short HiWrd(long aLong){return (short)(((aLong) >> 16) & 0xFFFF);}
inline short LoWrd(long aLong){return (short)((aLong) & 0xFFFF);}

void ErrorAlert(short StringsID,short theErrorID);
```

```
inline void SetMenuAbility(MenuHandle menu,short item,Boolean enable)
    {enable ? EnableItem(menu,item):DisableItem(menu,item);}

//////////////////////////////////////////////////////////////////////////
//
// class declarations
//
//////////////////////////////////////////////////////////////////////////

class TDoc {
protected:
    OSType      fCreator;
    SFReply     fFileInfo;
    Boolean     fFileOpen;
    short       fRefNum;
    WindowPtr   fDocWindow;
    Boolean     fNeedtoSave;
    Boolean     fNeedtoSaveAs;

public:

    // SFInfo will be non-nil when opening an existing document
    TDoc(OSType theCreator = '????',SFReply * SFInfo = (SFReply *) nil);

    // virtual destructor so that derived destructors will be called
    virtual ~TDoc(void);

    // called by TApp when making a document,
    // you probably won't override this
    virtual Boolean MakeWindow(Boolean colorWindow );

    // override to get your own WIND resource read in
    virtual short GetWinID(void);

    // InitDoc is available for your initialization
    // routines that might fail,
    // It gets called after the window is created
    virtual Boolean TDoc::InitDoc(void){return true;};

    // utilities that manipulate the window
    // can't be overridden
    WindowPtr GetDocWindow(void) { return fDocWindow; }
    void SetDocWindowTitle(Str255 title)
        {if(fDocWindow)SetWTitle(fDocWindow,title);}
    void MoveDocWindow(short h, short v)
        {if(fDocWindow)MoveWindow(fDocWindow,h,v,true);}
    void ShowDocWindow(void)
        {if(fDocWindow)ShowWindow(fDocWindow);}

// Event actions
```

```
    // this probably won't be overridden
    virtual void DoTheUpdate(EventRecord* theEvent);

protected:
    // override this to draw window contents
    virtual void Draw(Rect *r);

public:

    // override this if you don't want grow box
    virtual void DoDrawGrowIcon(void)
        {if(fDocWindow)DrawGrowIcon(fDocWindow);}

    virtual void DoActivate(EventRecord* theEvent);

    // override these to de/activate window (TEActivate, etc)
    virtual void Activate(void){}
    virtual void Deactivate(void){}

    // override these if you don't want default behavior
    virtual void DoZoom(short partCode) ;
    virtual void DoGrow(EventRecord* theEvent);
    virtual void DoDrag(EventRecord* theEvent);
    virtual short GetMinHeight(void){return 75;}
    virtual short GetMinWidth(void){return 75;}

    // override these to respond to clicks and keys
    virtual void DoContent(EventRecord* theEvent){}
    virtual void DoKeyDown(EventRecord* theEvent) {}

    virtual void DoIdle(void) {}
    virtual void AdjustCursor(Point where) {}  // where is in local coords

// Edit menu and clipboard actions

public:     // public functions are called from App class

    virtual void DoUndo(void) {}
    virtual Boolean DoCut(Handle *theData,OSType *theType)
        {*theData = nil;*theType = '????';return false;}
    virtual Boolean DoCopy(Handle *theData,OSType *theType)
        {*theData = nil;*theType = '????';return false;}
    virtual void DoPaste(Handle theData,OSType theType) {}

protected:  // protected functions are only called from Doc class

    virtual void DoClear(void) {}
    virtual void DoSelectAll(void) {}

public:
```

```
    // called by app to handle doc specific menu items
    // return true if you handle the menu command
    virtual Boolean DoDocMenuCommand(short menuID, short menuItem);
    virtual void AdjustDocMenus(void);

    // query state of document - useful for adjusting menu state
    // override if you can do any of these operations
    virtual Boolean CanUndo(void) { return false; }
    virtual Boolean HaveSelection(void) { return false; }
    virtual Boolean CanPaste(OSType /*theType*/) { return false; }
    virtual Boolean CanSelectAll(void){return false;}
    virtual Boolean CanClose(void) { return true; }
    virtual Boolean CanSaveAs(void) { return false; }
    virtual Boolean CanPageSetup(void){return false;}
    virtual Boolean CanPrint(void) { return false; }

    // this probably won't be overridden
    virtual Boolean CanSave(void) { return fNeedtoSave; }

    // override these to read and write files
    virtual Boolean ReadDocFile(short /*refNum*/){return true;}
    virtual Boolean WriteDocFile(short /*refNum*/){return true;}

    // override this if you have your own file type, such as TEXT
    virtual OSType GetDocType(void){return '????';}

    // these probably don't need to be changed
    virtual short OpenDocFile(SFReply * reply);
    virtual void CloseDocFile(short refNum);

    // these are called when user chooses associated menu item
    // these probably don't need to be changed
    virtual Boolean DoClose(void);
    virtual Boolean DoSave(void) ;
    virtual Boolean DoSaveAs(void) ;

    // this is called when user tries to close a doc
    // with unsaved changes
    // this probably doesn't need to be changed
    virtual short WantToSave(void);

    // override these for printing support
    virtual void DoPageSetup(void){}
    virtual void DoPrint(void) {}
};
#endif TDoc_Defs
```

```
//////////////////////////////////////////////////////////////////////////
//
// This is the generic document object
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////////

// Mac Includes
#include <Types.h>
#include <Windows.h>
#include <OSUtils.h>
#include <ToolUtils.h>
#include <Dialogs.h>
#include <Files.h>
#include <Errors.h>
#include <SysEqu.h>
#include <Desk.h>

#include "TDoc.h"
#include "AppDocMenus.h"

// define the segment for the TDoc class
#pragma segment DocSeg

//////////////////////////////////////////////////////////////////////////
//
// ErrorAlert
//
//////////////////////////////////////////////////////////////////////////
void ErrorAlert(short stringsID,short theError){

    short result;
    Str255 theStr;
    Str255 nullStr;
    *nullStr = 0;

    GetIndString(theStr,stringsID,theError);

    ParamText(theStr,nullStr,nullStr,nullStr);
    result = CautionAlert(rErrorAlert,(ModalFilterProcPtr)nil);

}
//////////////////////////////////////////////////////////////////////////
//
// TDoc::TDoc
//
//////////////////////////////////////////////////////////////////////////
TDoc::TDoc(OSType theCreator,SFReply * SFInfo){

    fCreator = theCreator;
```

```
    fNeedtoSave = false;
    fDocWindow = nil;
    fRefNum = 0;
    fFileOpen = false;
    if(SFInfo != nil){
        fNeedtoSaveAs = false;
        fFileInfo = *SFInfo;
    }
    else{
        fNeedtoSaveAs = true;
        fFileInfo.good = false;
    }
}
////////////////////////////////////////////////////////////////////////
//
// TDoc::~TDoc
//
////////////////////////////////////////////////////////////////////////
TDoc::~TDoc(void){
    if(fDocWindow != nil){
        DisposeWindow(fDocWindow);
        fDocWindow = nil;
    }

}
////////////////////////////////////////////////////////////////////////
//
//  TDoc::GetWinID
//
////////////////////////////////////////////////////////////////////////
short TDoc::GetWinID(void){
    return rGenericDoc;
}

////////////////////////////////////////////////////////////////////////
//
// TDoc::MakeWindow
//
////////////////////////////////////////////////////////////////////////
Boolean TDoc::MakeWindow(Boolean colorWindow){

    if(colorWindow)
        fDocWindow = GetNewCWindow(GetWinID(),nil,(WindowPtr)-1);
    else
        fDocWindow = GetNewWindow(GetWinID(),nil,(WindowPtr)-1);

    return (fDocWindow != nil);

}

////////////////////////////////////////////////////////////////////////
```

```
//
// TDoc::DoActivate
//
//////////////////////////////////////////////////////////////////////
void TDoc::DoActivate(EventRecord* theEvent){

    Boolean activating = theEvent->modifiers & activeFlag;

    // no need to activate if no window
    if(fDocWindow == nil)
        return;

    SetPort(fDocWindow);
    DoDrawGrowIcon();

    if(activating)
        Activate();
    else
        Deactivate();
}

//////////////////////////////////////////////////////////////////////
//
// TDoc::DoTheUpdate
//
//////////////////////////////////////////////////////////////////////
void TDoc::DoTheUpdate(EventRecord* /*theEvent*/){

    if(fDocWindow != nil){
        SetPort(fDocWindow);
        BeginUpdate(fDocWindow);
        Rect r = (**(fDocWindow->visRgn)).rgnBBox;
        Draw(&r);
        DoDrawGrowIcon();
        EndUpdate(fDocWindow);
    }
}
//////////////////////////////////////////////////////////////////////
//
// TDoc::Draw
//
//////////////////////////////////////////////////////////////////////
void TDoc::Draw(Rect *r){

    EraseRect(r);

}

//////////////////////////////////////////////////////////////////////
//
// TDoc::DoGrow
```

```
//
//////////////////////////////////////////////////////////////////////
void TDoc::DoGrow(EventRecord* theEvent){
    long result;

    // use desktop gray region as grow limits
    RgnHandle theGrayRgn = GetGrayRgn();

    Rect r = (**theGrayRgn).rgnBBox;
    r.top = GetMinHeight(); r.left = GetMinWidth();

    SetPort(fDocWindow);
    result = GrowWindow(fDocWindow, theEvent->where, &r);
    if ( result != 0 ){
        // invalidate the old scroll bar areas
        r = fDocWindow->portRect;
        r.left = r.right - kScrollBarPos;
        InvalRect(&r);
        r = fDocWindow->portRect;
        r.top = r.bottom - kScrollBarPos;
        InvalRect(&r);

        // now make the window the new size
        SizeWindow(fDocWindow, LoWrd(result), HiWrd(result), true);

        // invalidate the new scroll bar areas
        r = fDocWindow->portRect;
        r.left = r.right - kScrollBarPos;
        InvalRect(&r);
        r = fDocWindow->portRect;
        r.top = r.bottom - kScrollBarPos;
        InvalRect(&r);

    }
}
//////////////////////////////////////////////////////////////////////
//
// TDoc::DoZoom
//
//////////////////////////////////////////////////////////////////////
void TDoc::DoZoom(short partCode){
    if(fDocWindow){
        SetPort(fDocWindow);
        EraseRect(&fDocWindow->portRect);
        ZoomWindow(fDocWindow, partCode, fDocWindow == FrontWindow());

        // invalidate the whole content
        InvalRect(&fDocWindow->portRect);
    }
}
```

```
//////////////////////////////////////////////////////////////////////////
//
// TDoc::DoDrag
//
//////////////////////////////////////////////////////////////////////////
void TDoc::DoDrag(EventRecord* theEvent){

    // use desktop gray region as drag limits
    RgnHandle theGrayRgn = GetGrayRgn();
    Rect r = (**theGrayRgn).rgnBBox;

    if(fDocWindow)
        DragWindow(fDocWindow, theEvent->where, &r);
}
//////////////////////////////////////////////////////////////////////////
//
// TDoc::DoDocMenuCommand
//
//////////////////////////////////////////////////////////////////////////
Boolean TDoc::DoDocMenuCommand(short menuID, short menuItem){

    switch ( menuID ){
        case rFileMenu:
            switch ( menuItem ){
                case iSave:
                    DoSave();
                    break;
                case iSaveAs:
                    DoSaveAs();
                    break;
                case iPageSetup:
                    DoPageSetup();
                    break;
                case iPrint:
                    DoPrint();
                    break;
                default:
                    // we didn't handle command
                    return false;
            } // end menuItem switch
            return true;    // we handled this command

        case rEdit:
            if ( !SystemEdit(menuItem-1) ){
                switch ( menuItem ){
                    case iClear:
                        DoClear();
                        break;
                    case iSelectAll:
                        DoSelectAll();
                        break;
```

```
                    default:
                        // we didn't handle command
                        return false;
                } // end menuItem switch
                return true;    // we handled this command
            } else
                return true; // SystemEdit handled command

    }   // end menuID switch
    // we didn't handle command
    return false;
}

//////////////////////////////////////////////////////////////////////////
//
// TDoc::AdjustDocMenus
//
//////////////////////////////////////////////////////////////////////////
void TDoc::AdjustDocMenus(void){

    MenuHandle  menu;

// Do the document's portion of the file menu
    menu = GetMHandle(rFileMenu);
    SetMenuAbility(menu,iClose,CanClose());
    SetMenuAbility(menu,iSave,CanSave());
    SetMenuAbility(menu,iSaveAs,CanSaveAs());
    SetMenuAbility(menu,iPageSetup,CanPageSetup());
    SetMenuAbility(menu,iPrint,CanPrint());

// now the edit menu, App handles Paste Item
    menu = GetMHandle(rEdit);
    SetMenuAbility(menu,iUndo,CanUndo());
    SetMenuAbility(menu,iCut,HaveSelection());
    SetMenuAbility(menu,iCopy,HaveSelection());
    SetMenuAbility(menu,iClear,HaveSelection());
    SetMenuAbility(menu,iSelectAll,CanSelectAll());
}

//////////////////////////////////////////////////////////////////////////
//
// TDoc::OpenDocFile
//
//////////////////////////////////////////////////////////////////////////
short TDoc::OpenDocFile(SFReply * reply){

    short refnum;
    OSErr err = FSOpen((Str255)reply->fName,reply->vRefNum,&refnum);
    switch(err){
        case fnfErr:    // file not found, create it
            err = Create((Str255)reply->fName,
```

```
                            reply->vRefNum,
                            fCreator,
                            GetDocType());
            if(err == noErr){
                err = FSOpen((Str255)reply->fName,
                             reply->vRefNum,
                             &refnum);

                if(err != noErr)
                    return 0;
            } else
                return 0;

            // if open was successful, fall through
            // to next case

        case noErr:          // file opened OK
            fFileOpen = true;
            fRefNum = refnum;
            return refnum;

        case opWrErr:
            ErrorAlert(rDocErrorStrings,sFileOpen);
            return 0;

        default:
            ErrorAlert(rDocErrorStrings,sUnknownErr);
            return 0;

    }

}
////////////////////////////////////////////////////////////////////////
//
// TDoc::CloseDocFile
//
////////////////////////////////////////////////////////////////////////
void TDoc::CloseDocFile(short refNum){

    OSErr err = FSClose(refNum);

}

////////////////////////////////////////////////////////////////////////
//
// TDoc::DoClose
//
////////////////////////////////////////////////////////////////////////
Boolean TDoc::DoClose(void){

    // you could give the user a chance to save if necessary
```

```
    // and possibly cancel the close operation

    if(fNeedtoSave){
        // ask if they want to save it
        short saveit = WantToSave();
        if(saveit == iCancel)
            return false;
        if(saveit == iYes){
            // User can cancel save at this point too
            if (! DoSave())
                    return false;
        }
    }

    //close the file
    if(fFileOpen )
        CloseDocFile(fRefNum);

    // if all goes well, return true
    return true;
}
////////////////////////////////////////////////////////////////////////
//
// TDoc::WantToSave
//
////////////////////////////////////////////////////////////////////////
short TDoc::WantToSave(void){

    Str255 title;
    Str255 nullStr;
    *nullStr = 0;

    if(fDocWindow){
        GetWTitle(fDocWindow,title);
        ParamText(title,nullStr,nullStr,nullStr);
    } else
        ParamText(nullStr,nullStr,nullStr,nullStr);

    return Alert(rWantToSave,(ModalFilterProcPtr)nil);

}
////////////////////////////////////////////////////////////////////////
//
// TDoc::DoSaveAs
//
////////////////////////////////////////////////////////////////////////
Boolean TDoc::DoSaveAs(void){

    SFReply whereToSave;
    Point p;
    Str255  title;
```

```
    GetWTitle(fDocWindow,title);

    p.h = 100; p.v = 100;
    SFPutFile(p,
            "\pSave file as...",
            title,
            (DlgHookProcPtr)nil,
            &whereToSave);

    if(! whereToSave.good){
        // the user canceled the SaveAs
        return false;
    }else{
        fFileInfo = whereToSave;
        fRefNum = OpenDocFile(&whereToSave);
        if(fRefNum == 0){
            // file didn't open
            return false;
        }else{
            fFileOpen = true;
            if(! WriteDocFile(fRefNum)){
                // write was unsuccessful
                return false;
            }else{
                fNeedtoSave = false;
                fNeedtoSaveAs = false;
                SetDocWindowTitle(whereToSave.fName);
            }
        }
    }
    return true;    // passed every test for success
}
////////////////////////////////////////////////////////////////////////
//
// TDoc::DoSave
//
////////////////////////////////////////////////////////////////////////
Boolean TDoc::DoSave(void){

    if(fNeedtoSaveAs)
        return DoSaveAs();

    if(WriteDocFile(fRefNum)){
        fNeedtoSave = false;
        fNeedtoSaveAs = false;
        return true;
    } else
        return false;
}
```

```
/* TDoc.rsrc.r
 *
 * rez source for TDoc resources
 *
 * © 1990 Dan Weston All rights reserved
 *
 * Build it with the following rez command
 *
 *  rez TDoc.rsrc.r -o TDoc.rsrc -t rsrc -c RSED
 *
 */


#include "types.r"


resource 'WIND' (1000) {
    {40, 40, 182, 304},
    zoomDocProc,
    invisible,
    goAway,
    0x0,
    Untitled
};

resource 'ALRT' (500, "phSaveChanges", purgeable) {
    {100, 110, 220, 402},
    500,
    {   /* array: 4 elements */
        /* [1] */
        OK, visible, silent,
        /* [2] */
        OK, visible, silent,
        /* [3] */
        OK, visible, silent,
        /* [4] */
        OK, visible, silent
    }
};

resource 'ALRT' (255, "ErrorAlert", purgeable) {
    {100, 110, 220, 402},
    255,
    {   /* array: 4 elements */
        /* [1] */
        OK, visible, silent,
        /* [2] */
        OK, visible, silent,
        /* [3] */
        OK, visible, silent,
        /* [4] */
```

```
        OK, visible, silent
    }
};

resource 'DITL' (500, "phSaveChanges", purgeable) {
    {   /* array DITLarray: 4 elements */
        /* [1] */
        {58, 25, 76, 99},
        Button {
            enabled,
            "Yes"
        },
        /* [2] */
        {86, 195, 104, 269},
        Button {
            enabled,
            "Cancel"
        },
        /* [3] */
        {86, 25, 104, 99},
        Button {
            enabled,
            "No"
        },
        /* [4] */
        {12, 20, 53, 279},
        StaticText {
            disabled,
            "Save changes to "^0" before closing?"
        }
    }
};

resource 'DITL' (255, "phSaveChanges", purgeable) {
    {   /* array DITLarray: 2 elements */
        /* [1] */
        {85, 199, 105, 259},
        Button {
            enabled,
            "OK"
        },
        /* [2] */
        {11, 96, 76, 284},
        StaticText {
            disabled,
            "^0"
        }
    }
};

resource 'STR#' (255) {
```

```
    {   /* array StringArray: 3 elements */
        /* [1] */
        "Not enough memory to complete this opera"
        "tion.",
        /* [2] */
        "That file is already open.",
        /* [3] */
        "Unknown error."
    }
};
```

```
////////////////////////////////////////////////////////////////////////
//
// Menu ID constants shared by TApp and TDoc
//
////////////////////////////////////////////////////////////////////////

const short rMenuBarID   = 128;             /* application's menu bar */


const short rAppleMenu  = 128;      /* Apple menu */
const short iAbout  = 1;

const short rFileMenu = 129;        /* File menu */
const short  iNew = 1;
const short  iOpen = 2;
// -----
const short  iClose = 4;
const short  iSave = 5;
const short  iSaveAs = 6;
//----
const short iPageSetup   = 8;   //should be 7
const short iPrint = 9;
//----
const short iQuit = 11;         //should be 10

const short rEdit = 130;        /* Edit menu */
const short iUndo = 1;
//----
const short iCut = 3;
const short iCopy = 4;
const short iPaste = 5;
const short iClear = 6;
//----
const short  iSelectAll = 8;
```

```
////////////////////////////////////////////////////////////////////////
//
// This is the generic application object
//
// © 1990 Dan Weston, All Rights Reserved
//
////////////////////////////////////////////////////////////////////////

#ifndef TApp_H
#define TApp_H

#include <Types.h>
#include <Desk.h>
#include <Events.h>
#include <Menus.h>
#include <OSUtils.h>

// we need definitions of Document class
#include "TDoc.h"


////////////////////////////////////////////////////////////////////////
//
// class TLink
// TLink is a utility class that is used by
// the TList class below.
//
////////////////////////////////////////////////////////////////////////
class TLink {
    TLink*      fNext;  // the link to the next item
    void*       fItem;  // the item this link refers to

public:

    TLink(TLink *n = nil, void *item = nil)
        {fNext = n; fItem = item;}

    TLink* GetNext()
        { return fNext; }
    void* GetItem()
        { return fItem; }
    void SetNext(TLink* aLink)
        { fNext = aLink; }
    void SetItem(void* anItem)
        { fItem = anItem; }
};

////////////////////////////////////////////////////////////////////////
//
// class TList
//
```

```
//////////////////////////////////////////////////////////////////////

class TList {
protected:
    friend          class TIterator;
    TLink*          fLink;          // the first link in our list
    int             fNumItems;      // the number of elements in the list

public:
    TList(void);                    // constructor

    virtual void        AddItem(void* item);
    virtual void        RemoveItem(void* item);
    int NumItems() { return fNumItems; }

};

//////////////////////////////////////////////////////////////////////
//
// class TIterator
//
//////////////////////////////////////////////////////////////////////
class TIterator {
    TLink*  fCurLink;

public:
    TIterator(TList* list) { fCurLink = list->fLink; }
    void* Next(void);
};

//////////////////////////////////////////////////////////////////////
//
// class TDocList
//
//////////////////////////////////////////////////////////////////////
class TDocList : public TList {

public:

    // add one new member function
    // find the TDocument associated with the window
    TDoc*   FindDoc(WindowPtr window);


};

//////////////////////////////////////////////////////////////////////
//
//  class TApp
//
//////////////////////////////////////////////////////////////////////
```

```
class TApp {

public:

    //  other classes might like to see this
    SysEnvRec       fenvRec;

protected:

    // members just for TApp and derived classes
    TDocList*       fDocList;
    TDoc*           fCurDoc;
    Boolean         fHaveWaitNextEvent;
    Boolean         fDone;
    Boolean         fInBackground;
    Handle          fClipData;
    OSType          fClipType;
    Boolean         fDAonTop;
    short           fLastScrapCount;

public:

    // constructor needs to be public
    TApp(void);

    virtual ~TApp(void){delete fDocList;}

    // These three member functions get called from main()
    virtual Boolean InitApp(void);
    virtual void EventLoop(void);
    virtual void CleanUp(void);

    // these functions create documents, and manipulate
    // the document list
    // probably won't be overridden
    virtual Boolean OpenDocFromFinder(void);
    virtual void OpenNewDoc(void);
    virtual void OpenOldDoc(void);

    // add a document to app's document list
    virtual void AddDocument(TDoc * theDoc);

protected:

    // override this function to make the kind of document
    // supported by your application
    virtual TDoc * MakeDoc(SFReply * reply = (SFReply *)nil);

    // override this if your app has a unique creator signature
    virtual OSType GetCreator(void){return '????';}
```

```
    // routines to override to configure SFGetFile
    // override these to specify the type of files you can open
    virtual int GetNumFileTypes(void){return 0;};
    virtual SFTypeList GetFileTypesList(void){return (SFTypeList)nil;}
    virtual Boolean AcceptableFileType(OSType theType);

    // common code for OpenDocFromFinder and OpenOldDoc
    virtual Boolean InitOldDoc(SFReply * reply);

// event handlers you shouldn't need to override in a typical application
// these call event handlers for documents, where real functionality is
    virtual void OSEvent(EventRecord * theEvent);
    virtual void MouseDown(EventRecord * theEvent);
    virtual void KeyDown(EventRecord * theEvent);
    virtual void ActivateEvt(EventRecord * theEvent);
    virtual void UpdateEvt(EventRecord * theEvent);
    virtual void ExitLoop(void){fDone = true;}
    virtual void AppIdle(void);

// override these if you need to respond to
// mouseups or disk inserted events
    virtual void MouseUp(EventRecord * theEvent){}
    virtual void DiskEvt(EventRecord * theEvent){}

// MultiFinder friendly functions
    virtual void DoSuspend(EventRecord * theEvent,Boolean convertClip);
    virtual void DoResume(EventRecord * theEvent,Boolean convertClip);

    // how long to sleep in WaitNextEvent
    virtual unsigned long SleepVal(void) { return 0; }

// menu functions

    // handles standard DA, File, and Edit menus
    // override to handle additional menus
    // but call TApp::DoMenuCommand if your derived function
    // doesn't handle menu choice
    virtual void DoMenuCommand(short menuID, short menuItem);

    // functions that control state of menu items
    virtual void AdjustMenus(void);
    virtual Boolean CanNew(void){return true;}
    virtual Boolean CanOpen(void){return false;}
    virtual Boolean CanQuit(void){return true;}

    // responses for File Menu items
    virtual Boolean CloseADoc(TDoc * theDoc);
    virtual void Quit(void);

// clipboard support
```

```
    virtual void    GetClipFromSystem(void);
    virtual void    GiveClipToSystem(void);
    virtual OSType  CanAcceptClipType(void);
    virtual Boolean ClipHasChanged(void);
    virtual void CheckForDASwitch(WindowPtr theFrontWindow);

    // responses to Edit menu items
    virtual void DoUndoCmd(TDoc * theDoc);
    virtual void DoCutCmd(TDoc * theDoc);
    virtual void DoCopyCmd(TDoc * theDoc);
    virtual void DoPasteCmd(TDoc * theDoc);

    // Utility routine
    Boolean TrapAvailable(short tNumber,TrapType tType);

};

#endif TApp_H
```

```
//////////////////////////////////////////////////////////////////////////
//
// This is the generic application object
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////////

// Mac Includes
#include <Types.h>
#include <Events.h>
#include <Windows.h>
#include <Menus.h>
#include <Dialogs.h>
#include <Desk.h>
#include <ToolUtils.h>
#include <Fonts.h>
#include <Memory.h>
#include <OSUtils.h>
#include <Traps.h>
#include <SegLoad.h>
#include <Scrap.h>

// our includes
#include "TApp.h"
#include "AppDocMenus.h"

// define the segment for the TApp class
#pragma segment AppSeg

//////////////////////////////////////////////////////////////////////////
//
//  constants
//
//////////////////////////////////////////////////////////////////////////

const short rAboutID    = 128;              /* about alert */

const short kOSEvent = app4Evt;
const short kSuspendResumeMessage = 0x01;
const short kClipConvertMask = 0x02;
const short kResumeMask = 0x01;
const short kMouseMovedMessage = 0xFA;

const short kStagger = 20;
const short kHPos    = 20;
const short kVPos    = 50;


//////////////////////////////////////////////////////////////////////////
//
```

```
// TList::TList
//
//////////////////////////////////////////////////////////////////////////
TList::TList(void){
    fLink = nil;
    fNumItems = 0;
}


//////////////////////////////////////////////////////////////////////////
//
// TList::AddItem
//
//////////////////////////////////////////////////////////////////////////
void TList::AddItem(void* item){
    TLink* temp;

    temp = new TLink(fLink,item);
    fLink = temp;
    fNumItems++;
}
//////////////////////////////////////////////////////////////////////////
//
// TList::RemoveItem
//
//////////////////////////////////////////////////////////////////////////
void TList::RemoveItem(void* item){
    TLink* temp;
    TLink* last;

    last = nil;
    for (temp = fLink; temp != nil; temp = temp->GetNext())
        if (temp->GetItem() == item){
            // if first item in list, just set first
            if (last == nil)
                fLink = temp->GetNext();
            else
                last->SetNext(temp->GetNext());
            delete temp;
            fNumItems--;
            return;
        } else
            last = temp;
}

//////////////////////////////////////////////////////////////////////////
//
// TDocList::FindDoc
// find the TDoc associated with the window
//
//////////////////////////////////////////////////////////////////////////
```

```
TDoc* TDocList::FindDoc(WindowPtr window){
    TLink* temp;
    TDoc* tDoc;

    for (temp = fLink; temp != nil; temp = temp->GetNext()){
        tDoc = (TDoc*)temp->GetItem();
        if (tDoc->GetDocWindow() == window)
          return tDoc;
     }
    return nil;
}

//////////////////////////////////////////////////////////////////////////
//
// TIterator::Next
//
//////////////////////////////////////////////////////////////////////////
void* TIterator::Next(void){
    TLink* link = fCurLink;

    if (fCurLink) {
        fCurLink = fCurLink->GetNext();
        return (link->GetItem());
    } else
        return nil;
}


//////////////////////////////////////////////////////////////////////////
//
// TApp::TApp
//
//////////////////////////////////////////////////////////////////////////
TApp::TApp(void){

    // initialize our class variables
    fCurDoc = nil;
    fDone = false;
    fInBackground = false;
    fClipData = nil;
    fClipType = '????';
    fDAonTop = false;
    fLastScrapCount = 0;

    // initialize Mac Toolbox components
    InitGraf((Ptr) &qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs((ResumeProcPtr) nil);
```

```
    InitCursor();

    (void) SysEnvirons(curSysEnvVers, &fenvRec);

    // expand the heap so new code segments load at the top
    MaxApplZone();

    // allocate an empty document list
    fDocList = new TDocList;

    // check to see if WaitNextEvent is implemented
    fHaveWaitNextEvent = TrapAvailable(_WaitNextEvent, ToolTrap);

    // read menus into menu bar
    Handle  menuBar = GetNewMBar(rMenuBarID);
    // install menus
    SetMenuBar(menuBar);
    DisposHandle(menuBar);

    // add DA names to Apple menu
    AddResMenu(GetMHandle(rAppleMenu), 'DRVR');

    DrawMenuBar();
}

////////////////////////////////////////////////////////////////////////
//
// TApp::InitApp
//
////////////////////////////////////////////////////////////////////////
Boolean TApp::InitApp(void){

    GetClipFromSystem();
    return true;
}

////////////////////////////////////////////////////////////////////////
//
// TApp::CleanUp
//
////////////////////////////////////////////////////////////////////////
void TApp::CleanUp(void){

    GiveClipToSystem();

}

////////////////////////////////////////////////////////////////////////
//
// TApp::CheckForDASwitch
//
```

```
////////////////////////////////////////////////////////////////////////
void TApp::CheckForDASwitch(WindowPtr   theFrontWindow){

    if(theFrontWindow == nil)
        return;

    Boolean DAWindowOnTop;
    DAWindowOnTop = ((WindowPeek)theFrontWindow)->windowKind < 0;

    // if the state has changed since we last checked it, then
    // do clipboard conversion
    if(DAWindowOnTop != fDAonTop){
        fDAonTop = DAWindowOnTop;

        if(DAWindowOnTop)
            // DA is becoming active, give up the clipboard
            GiveClipToSystem();
        else {
            // DA is becoming inactive, reclaim clip if necessary
            if(ClipHasChanged())
                GetClipFromSystem();
        }
    }
}
////////////////////////////////////////////////////////////////////////
//
// TApp::ClipHasChanged
//
////////////////////////////////////////////////////////////////////////
Boolean TApp::ClipHasChanged(void){

    PScrapStuff scrapInfo = InfoScrap();

    return (scrapInfo->scrapCount != fLastScrapCount);


}

////////////////////////////////////////////////////////////////////////
//
// TApp::GiveClipToSystem
//
////////////////////////////////////////////////////////////////////////
void TApp::GiveClipToSystem(void){

    if(fClipData != nil){
        long result = ZeroScrap();
        if(result != noErr)
            return;

        long size = GetHandleSize(fClipData);
```

```
        HLock(fClipData);
        result = PutScrap(size,fClipType,*fClipData);
        HUnlock(fClipData);
    }
    // update our scrapcount field so we can tell if scrap
    // has changed later on
    PScrapStuff scrapInfo = InfoScrap();
    fLastScrapCount = scrapInfo->scrapCount;

}
////////////////////////////////////////////////////////////////////////
//
// TApp::GetClipFromSystem
//
////////////////////////////////////////////////////////////////////////
void TApp::GetClipFromSystem(void){

    long    offset;
    Handle newData = NewHandle(0);
    OSType  newType = CanAcceptClipType();

    long result = GetScrap(newData,newType,&offset);
    if(result > 0){
        if(fClipData != nil)
            DisposHandle(fClipData);
        fClipData = newData;
        fClipType = newType;
    }
}
////////////////////////////////////////////////////////////////////////
//
// TApp::CanAcceptClipType
//
////////////////////////////////////////////////////////////////////////
OSType TApp::CanAcceptClipType(void){

    return '????';

}

////////////////////////////////////////////////////////////////////////
//
// TApp::EventLoop
//
////////////////////////////////////////////////////////////////////////
void TApp::EventLoop(void){

    int gotEvent;
    EventRecord theEvent;
    WindowPtr   theFrontWindow;
```

```
    while (fDone == false){
        theFrontWindow = FrontWindow();

        // find out if a DA is becoming active or inactive
        CheckForDASwitch(theFrontWindow);

        // see if window belongs to a document,
        // FindDoc will return nil if not one of our windows
        fCurDoc = fDocList->FindDoc(theFrontWindow);

        if (fHaveWaitNextEvent)
            gotEvent = WaitNextEvent(everyEvent,
                                     &theEvent,
                                     SleepVal(),
                                     (RgnHandle)nil);
        else {
            SystemTask();
            gotEvent = GetNextEvent(everyEvent, &theEvent);
        }

        // make sure we got a real event
        if (gotEvent == false){
            AppIdle();
            continue;
        }

        switch (theEvent.what){
            case nullEvent :
                AppIdle();
                break;
            case mouseDown :
                MouseDown(&theEvent);
                break;
            case mouseUp :
                MouseUp(&theEvent);
                break;
            case keyDown :
            case autoKey :
                KeyDown(&theEvent);
                break;
            case updateEvt :
                UpdateEvt(&theEvent);
                break;
            case diskEvt :
                DiskEvt(&theEvent);
                break;
            case activateEvt :
                ActivateEvt(&theEvent);
                break;
            case kOSEvent :
                OSEvent(&theEvent);
```

```
                break;
            default :
                break;
        }  // end switch
    }   // end while
}

//////////////////////////////////////////////////////////////////////////
//
// TApp::AddDocument
//
//////////////////////////////////////////////////////////////////////////
void TApp::AddDocument(TDoc *theDoc){

    fDocList->AddItem(theDoc);
    fCurDoc = theDoc;
}


//////////////////////////////////////////////////////////////////////////
//
// TApp::OpenDocFromFinder
//
//////////////////////////////////////////////////////////////////////////
Boolean TApp::OpenDocFromFinder(void){

    short   message;
    short   count;
    AppFile theApp;
    SFReply reply;
    Boolean fileOpened = false;

    // see if there are any files to be opened or printed
    CountAppFiles(&message,&count);
    if(count == 0)
        return false;

    for(short i = count;i;i--){
        GetAppFiles(i,&theApp);
        // convert theApp to an SFReply
        reply.good = true;
        reply.fType = theApp.fType;
        reply.vRefNum = theApp.vRefNum;
        reply.version = theApp.versNum;
        unsigned char strLen = theApp.fName[0];
        for(short j = 0; j <= strLen; j++)
            reply.fName[j] = theApp.fName[j];

        // check here to see if file is an acceptable type
        if(AcceptableFileType(reply.fType))
            // now create the document and open the file
```

```
            if(InitOldDoc(&reply))
                fileOpened = true;
    }
    return fileOpened;
}
////////////////////////////////////////////////////////////////////////
//
// TApp::OpenNewDoc
// Creates a new document object, staggers it, and adds it to doclist
//
////////////////////////////////////////////////////////////////////////
void  TApp::OpenNewDoc(void){

    TDoc * newDoc = MakeDoc();
    if(newDoc){
        if((newDoc->MakeWindow(fenvRec.hasColorQD)) &&
            (newDoc->InitDoc())){
            short numDocs = fDocList->NumItems() ;
            newDoc->MoveDocWindow(kHPos + (numDocs * kStagger),
                                  kVPos + (numDocs * kStagger));
            newDoc->ShowDocWindow();
            AddDocument(newDoc);
        } else {
            // MakeWindow or InitDoc failed, but doc created
            delete(newDoc);
        }
    }
}

////////////////////////////////////////////////////////////////////////
//
// TApp::OpenOldDoc
//
////////////////////////////////////////////////////////////////////////
void TApp::OpenOldDoc(void){

    SFReply reply;
    Point   p;

    p.h = 100; p.v = 100;
    SFGetFile(p,
            (Str255)"",
            (FileFilterProcPtr)nil,
            GetNumFileTypes(),
            GetFileTypesList(),
            (DlgHookProcPtr)nil,
            &reply);

    // don't go on if user cancels dialog
    if(! reply.good)
        return;
```

```
    (void)InitOldDoc(&reply);

}


////////////////////////////////////////////////////////////////////////
//
// TApp::InitOldDoc
//
// Creates a new document object, reads in data for it,
// sets the window title to file name, staggers it,
// and adds it to doclist

//  This routine is shared by both OpenDocFromFinder
//  and OpenOldDoc
//
////////////////////////////////////////////////////////////////////////
Boolean TApp::InitOldDoc(SFReply * reply){

    TDoc * newDoc = MakeDoc(reply);
    if(newDoc){
        if((newDoc->MakeWindow(fenvRec.hasColorQD))
            && (newDoc->InitDoc())){
            short numDocs = fDocList->NumItems() ;
            newDoc->MoveDocWindow(kHPos + (numDocs * kStagger),
                                  kVPos + (numDocs * kStagger));
            newDoc->SetDocWindowTitle((Str255)reply->fName);
            short refNum = newDoc->OpenDocFile(reply);

            if(refNum != 0){
                if(newDoc->ReadDocFile(refNum)){
                    newDoc->ShowDocWindow();
                    AddDocument(newDoc);
                }else{
                    // open was successful, but read failed
                    newDoc->CloseDocFile(refNum);
                    delete(newDoc);
                    return false;
                }
            } else {
                // file not opened successfully, but doc created
                delete(newDoc);
                return false;
            }
        } else {
            // MakeWindow or InitDoc failed, but doc created
            delete(newDoc);
            return false;
        }
    } else{
```

```
        // document not created
        return false;
    }
    // if we get this far, all went well
    return true;
}

////////////////////////////////////////////////////////////////////////
//
// TApp::MakeDoc
// Override this function to make the type of document
// that your application uses
//
////////////////////////////////////////////////////////////////////////
TDoc * TApp::MakeDoc(SFReply * reply){

    return new TDoc(GetCreator(),reply);

}

////////////////////////////////////////////////////////////////////////
//
// TApp::AcceptableFileType
//
////////////////////////////////////////////////////////////////////////
Boolean TApp::AcceptableFileType(OSType theType){

    int numTypes = GetNumFileTypes();
    OSType *theTypeList = (OSType *)GetFileTypesList();

    if((numTypes == 0) || (theTypeList == nil))
        return true;

    for (int i = 0;i < numTypes; i++){
        if(theType == *theTypeList++)
            return true;
    }
    return false;
}

////////////////////////////////////////////////////////////////////////
//
// TApp::OSEvent
//
////////////////////////////////////////////////////////////////////////
void TApp::OSEvent(EventRecord * theEvent){

    Boolean doConvert;
    unsigned char evType;

    // is it a multifinder event?
```

```
    evType = (unsigned char)(theEvent->message >> 24) & 0x00ff;
    switch (evType) {   // high byte of message is type of event
        case kMouseMovedMessage :
            AppIdle();          // mouse-moved is also an idle event
            break;
        case kSuspendResumeMessage :
            doConvert = (theEvent->message & kClipConvertMask) != 0;
            fInBackground = (theEvent->message & kResumeMask) == 0;
            if (fInBackground)
                DoSuspend(theEvent,doConvert);
            else
                DoResume(theEvent,doConvert);
            break;
    }
}

////////////////////////////////////////////////////////////////////////
//
// TApp::MouseDown
//
////////////////////////////////////////////////////////////////////////
void TApp::MouseDown(EventRecord * theEvent){

    WindowPtr theWindow;

    short partCode = FindWindow(theEvent->where, &theWindow);

    TDoc * tempDoc = fDocList->FindDoc(theWindow);

    switch (partCode){
        case inSysWindow :
            SystemClick(theEvent,theWindow);
            break;
        case inMenuBar :
            AdjustMenus();
            long mResult = MenuSelect(theEvent->where);
            if (HiWrd(mResult) != 0){
              DoMenuCommand(HiWrd(mResult),LoWrd(mResult));
              HiliteMenu(0);
            }
            break;
        case inGoAway :
            if (TrackGoAway(theWindow, theEvent->where))
                CloseADoc(tempDoc);
            break;
        case inDrag :
            if(tempDoc != nil)
                tempDoc->DoDrag(theEvent);
            break;
        case inGrow :
            if (tempDoc != nil)
```

```
            tempDoc->DoGrow(theEvent);
        break;
    case inZoomIn :
    case inZoomOut :
        if ((TrackBox(theWindow, theEvent->where, partCode)) &&
            (tempDoc != nil))
          tempDoc->DoZoom(partCode);
        break;
    case inContent :
        if(theWindow != FrontWindow())
            SelectWindow(theWindow);
        else
            if(tempDoc != nil)
                tempDoc->DoContent(theEvent);
        break;

  }
}

//////////////////////////////////////////////////////////////////////
//
// TApp::KeyDown
//
//////////////////////////////////////////////////////////////////////
void TApp::KeyDown(EventRecord * theEvent){

    char key;
    long mResult;

    key = (char) (theEvent->message & charCodeMask);
    if ((theEvent->modifiers & cmdKey) &&
        (theEvent->what == keyDown)){
        // only do command keys if we are not autokeying
        AdjustMenus();              // make sure menus are up to date
        mResult = MenuKey(key);
        // if it wasn't a menu key, pass it through
        if (HiWrd(mResult) != 0){
            DoMenuCommand(HiWrd(mResult), LoWrd(mResult));
            HiliteMenu(0);
            return;
         }
    }else
        if (fCurDoc != nil)
            fCurDoc->DoKeyDown(theEvent);
}
//////////////////////////////////////////////////////////////////////
//
// TApp::ActivateEvt
//
//////////////////////////////////////////////////////////////////////
void TApp::ActivateEvt(EventRecord * theEvent){
```

```
    WindowPtr theWindow;

    // event record contains window ptr
    theWindow = (WindowPtr) theEvent->message;

    // see if window belongs to a document
    TDoc *tempDoc = fDocList->FindDoc(theWindow);

    if (tempDoc != nil)
      tempDoc->DoActivate(theEvent );
}
////////////////////////////////////////////////////////////////////////
//
// TApp::UpdateEvt
//
////////////////////////////////////////////////////////////////////////
void TApp::UpdateEvt(EventRecord * theEvent){

    WindowPtr theWindow;

    // event record contains window ptr
    theWindow = (WindowPtr) theEvent->message;

    // see if window belongs to a document
    TDoc *tempDoc = fDocList->FindDoc(theWindow);

    if (tempDoc != nil)
        tempDoc->DoTheUpdate(theEvent);

}

////////////////////////////////////////////////////////////////////////
//
// TApp::AppIdle
//
////////////////////////////////////////////////////////////////////////
void TApp::AppIdle(void){

    if (fCurDoc != nil)
        fCurDoc->DoIdle();

}


////////////////////////////////////////////////////////////////////////
//
// TApp::DoSuspend
//
////////////////////////////////////////////////////////////////////////
void TApp::DoSuspend(EventRecord * theEvent,Boolean convertClip){
```

```
    if(convertClip)
        GiveClipToSystem();

    if (fCurDoc != nil){
        // tell DoActivate to deactivate
        theEvent->modifiers &= (!activeFlag);
        fCurDoc->DoActivate(theEvent );
    }
}
////////////////////////////////////////////////////////////////////////
//
// TApp::DoResume
//
////////////////////////////////////////////////////////////////////////
void TApp::DoResume(EventRecord * theEvent,Boolean convertClip){

    if(convertClip)
        GetClipFromSystem();

    if (fCurDoc != nil){
        // tell DoActivate to activate
        theEvent->modifiers |= activeFlag;
        fCurDoc->DoActivate(theEvent);
    }
}

////////////////////////////////////////////////////////////////////////
//
// TApp::AdjustMenus
// Enable and disable menus based on the current state.
//
////////////////////////////////////////////////////////////////////////
void TApp::AdjustMenus(void){

    MenuHandle  menu;

    // first give the current document a chance to adjust the menus
    if(fCurDoc != nil)
        fCurDoc->AdjustDocMenus();

    // Now do the file menu
    menu = GetMHandle(rFileMenu);
    // the app controls whether we can open and new and quit
    SetMenuAbility(menu,iNew,CanNew());
    SetMenuAbility(menu,iOpen,CanOpen());
    SetMenuAbility(menu,iQuit,CanQuit());

    if ( fCurDoc == nil ){
        // no current doc, disable File menu items
        // usually handled by the document
```

```
        SetMenuAbility(menu,iClose,false);
        SetMenuAbility(menu,iSave,false);
        SetMenuAbility(menu,iSaveAs,false);
        SetMenuAbility(menu,iPageSetup,false);
        SetMenuAbility(menu,iPrint,false);
    }

    // now the edit menu
    menu = GetMHandle(rEdit);
    // if no current doc, then enable edit menu depending
    // on whether a DA is on top
    if ( fCurDoc == nil ){
        SetMenuAbility(menu,iUndo,fDAonTop);
        SetMenuAbility(menu,iCut,fDAonTop);
        SetMenuAbility(menu,iCopy,fDAonTop);
        SetMenuAbility(menu,iPaste,fDAonTop);
        SetMenuAbility(menu,iClear,fDAonTop);
        SetMenuAbility(menu,iSelectAll,fDAonTop);
    } else {
        // Paste is the one Edit item that the doc can't
        // set by itself
        SetMenuAbility(menu,iPaste,(fClipData != nil) &&
                                (fCurDoc->CanPaste(fClipType)));
    }
}
//////////////////////////////////////////////////////////////////////////
//
// TApp::DoMenuCommand
//
//////////////////////////////////////////////////////////////////////////
void TApp::DoMenuCommand(short menuID, short menuItem){

    short       itemHit;
    Str255      daName;
    short       daRefNum;

    // allow the current doc a chance to handle it first
    if((fCurDoc != nil) &&
        (fCurDoc->DoDocMenuCommand(menuID,menuItem)))
        return;

    switch ( menuID ){
        case rAppleMenu:
            switch ( menuItem ){
                case iAbout:
                    itemHit = Alert(rAboutID, nil);
                    break;
                default:
                    GetItem(GetMHandle(rAppleMenu), menuItem, daName);
                    daRefNum = OpenDeskAcc(daName);
                    break;
```

```
            }  // end menuItem switch

            break;

        case rFileMenu:
            switch ( menuItem ){
                case iNew:
                    OpenNewDoc();
                    break;
                case iOpen:
                    OpenOldDoc();
                    break;
                case iClose:
                    CloseADoc(fCurDoc);
                    break;
                case iQuit:
                    Quit();
                    break;
            } // end menuItem switch

            break;

        case rEdit:
            if ( !SystemEdit(menuItem-1) ){
                switch ( menuItem ){
                    case iUndo:
                        DoUndoCmd(fCurDoc);
                        break;
                    case iCut:
                        DoCutCmd(fCurDoc);
                        break;
                    case iCopy:
                        DoCopyCmd(fCurDoc);
                        break;
                    case iPaste:
                        DoPasteCmd(fCurDoc);
                        break;

                } // end menuItem switch
            }   // end if

            break;

    }   // end menuID switch
}

////////////////////////////////////////////////////////////////////////
//
// TApp::CloseADoc
//
////////////////////////////////////////////////////////////////////////
```

```
Boolean TApp::CloseADoc(TDoc * theDoc){

    if(theDoc != nil)
        if(theDoc->DoClose()){
            fDocList->RemoveItem(theDoc);
            if(theDoc == fCurDoc)
                fCurDoc = nil;
            delete theDoc;
            return true;
        }
    // if we get here, the doc didn't close
    return false;
}

////////////////////////////////////////////////////////////////////////
//
// TApp::Quit
//
////////////////////////////////////////////////////////////////////////
void TApp::Quit(void){

    TIterator iter(fDocList);
    TDoc * nextDoc;
    Boolean OKToQuit = true;

    // ask each doc if it is ready to Quit
    // It is possible that the user may cancel
    // while saving one of these documents,
    // thus aborting the Quit process
    while (nextDoc = (TDoc *)iter.Next())
        if(! CloseADoc(nextDoc)){
            OKToQuit = false;
            break;  // don't continue iterating
        }

    if(OKToQuit)
        ExitLoop();

}

////////////////////////////////////////////////////////////////////////
//
// TApp::DoUndoCmd
//
////////////////////////////////////////////////////////////////////////
void TApp::DoUndoCmd(TDoc * theDoc){

    if(theDoc != nil)
        theDoc->DoUndo();

}
```

```
//////////////////////////////////////////////////////////////////////
//
// TApp::DoCutCmd
//
//////////////////////////////////////////////////////////////////////
void TApp::DoCutCmd(TDoc * theDoc){

    Handle  newData;
    OSType  newType;

    if(theDoc != nil)
        if(theDoc->DoCut(&newData,&newType)){
            //get rid of old clip data if DoCut succeeds
            if(fClipData != nil)
                DisposHandle(fClipData);
            fClipData = newData;
            fClipType = newType;
        }
}
//////////////////////////////////////////////////////////////////////
//
// TApp::DoCopyCmd
//
//////////////////////////////////////////////////////////////////////
void TApp::DoCopyCmd(TDoc * theDoc){

    Handle  newData;
    OSType  newType;

    if(theDoc != nil)
        if(theDoc->DoCopy(&newData,&newType)){
            //get rid of old clip data if DoCopy succeeds
            if(fClipData != nil)
                DisposHandle(fClipData);
            fClipData = newData;
            fClipType = newType;
        }
}
//////////////////////////////////////////////////////////////////////
//
// TApp::DoPasteCmd
//
//////////////////////////////////////////////////////////////////////
void TApp::DoPasteCmd(TDoc * theDoc){

    if(theDoc != nil)
        theDoc->DoPaste(fClipData,fClipType);

}
//////////////////////////////////////////////////////////////////////
```

```
//
// TApp::TrapAvailable
//
//////////////////////////////////////////////////////////////////////
Boolean TApp::TrapAvailable(short tNumber,TrapType tType){

    return NGetTrapAddress(tNumber, tType) !=
            GetTrapAddress(_Unimplemented);
}
```

```
/* TApp.rsrc.r
 *
 * rez source for TApp resources
 *
 * © 1990 Dan Weston All rights reserved
 *
 * Build it with the following rez command
 *
 *  rez TApp.rsrc.r -o TApp.rsrc -t rsrc -c RSED
 *
 */


#include "types.r"


resource 'ALRT' (128, purgeable) {
    {68, 76, 172, 376},
    128,
    {   /* array: 4 elements */
        /* [1] */
        OK, visible, silent,
        /* [2] */
        OK, visible, silent,
        /* [3] */
        OK, visible, silent,
        /* [4] */
        OK, visible, silent
    }
};

resource 'DITL' (128, purgeable) {
    {   /* array DITLarray: 5 elements */
        /* [1] */
        {65, 199, 85, 279},
        Button {
            enabled,
            "OK"
        },
        /* [2] */
        {8, 8, 24, 304},
        StaticText {
            disabled,
            "Generic Application"
        },
        /* [3] */
        {44, 199, 59, 245},
        StaticText {
            disabled,
            ""
        },
```

```
        /* [4] */
        {42, 34, 58, 162},
        StaticText {
            disabled,
            "version number ??"
        },
        /* [5] */
        {65, 34, 89, 165},
        StaticText {
            disabled,
            "Date completed"
        }
    }
};

resource 'MENU' (128, preload) {
    128,
    textMenuProc,
    0x7FFFFFFD,
    enabled,
    apple,
    {   /* array: 2 elements */
        /* [1] */
        "About Generic", noIcon, noKey, noMark, plain,
        /* [2] */
        "-", noIcon, noKey, noMark, plain
    }
};

resource 'MENU' (129, preload) {
    129,
    textMenuProc,
    0x7FFFFDBB,
    enabled,
    "File",
    {   /* array: 11 elements */
        /* [1] */
        "New", noIcon, "N", noMark, plain,
        /* [2] */
        "Open", noIcon, "O", noMark, plain,
        /* [3] */
        "-", noIcon, noKey, noMark, plain,
        /* [4] */
        "Close", noIcon, "W", noMark, plain,
        /* [5] */
        "Save", noIcon, "S", noMark, plain,
        /* [6] */
        "Save as...", noIcon, noKey, noMark, plain,
        /* [7] */
        "-", noIcon, noKey, noMark, plain,
        /* [8] */
```

```
        "Page Setup", noIcon, noKey, noMark, plain,
        /* [9] */
        "Print", noIcon, "P", noMark, plain,
        /* [10] */
        "-", noIcon, noKey, noMark, plain,
        /* [11] */
        "Quit", noIcon, "Q", noMark, plain
    }
};

resource 'MENU' (130, preload) {
    130,
    textMenuProc,
    0xBD,
    enabled,
    "Edit",
    {   /* array: 8 elements */
        /* [1] */
        "Undo", noIcon, "Z", noMark, plain,
        /* [2] */
        "-", noIcon, noKey, noMark, plain,
        /* [3] */
        "Cut", noIcon, "X", noMark, plain,
        /* [4] */
        "Copy", noIcon, "C", noMark, plain,
        /* [5] */
        "Paste", noIcon, "V", noMark, plain,
        /* [6] */
        "Clear", noIcon, noKey, noMark, plain,
        /* [7] */
        "-", noIcon, noKey, noMark, plain,
        /* [8] */
        "Select All", noIcon, "A", noMark, plain
    }
};

resource 'MBAR' (128, preload) {
    {   /* array MenuArray: 3 elements */
        /* [1] */
        128,
        /* [2] */
        129,
        /* [3] */
        130
    }
};


resource 'SIZE' (-1) {
    dontSaveScreen,
    acceptSuspendResumeEvents,
```

```
    enableOptionSwitch,
    canBackground,
    multiFinderAware,
    backgroundAndForeground,
    dontGetFrontClicks,
    ignoreChildDiedEvents,
    is32BitCompatible,
    reserved,
    reserved,
    reserved,
    reserved,
    reserved,
    reserved,
    reserved,
    384000,
    384000
};
```

```
//////////////////////////////////////////////////////////////////////
//
//  This file: Helloworld2.cp
//
//  This is the source for the
//  Helloworld2 program
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////

#include <Windows.h>
#include <Fonts.h>

#include "TApp.h"
#include "TDoc.h"

//////////////////////////////////////////////////////////////////////
//
// class declarations
//
//////////////////////////////////////////////////////////////////////

class THelloDoc : public TDoc{

    protected:

        // draw the window
        virtual void Draw(Rect *r);
};

class THelloApp : public TApp{

    protected:
        // make our kind of document
        virtual TDoc * MakeDoc(SFReply * reply = (SFReply *)nil);
};

//////////////////////////////////////////////////////////////////////
//
// main
//
//////////////////////////////////////////////////////////////////////
void main(void){

    THelloApp theApp;

    // initialize the application
    if(theApp.InitApp()){

        // open one window to start with
```

```
        theApp.OpenNewDoc();

        // Start our main event loop running.
        // This won't return until user quits
        theApp.EventLoop();

        //now clean up
        theApp.CleanUp();
    }
}

////////////////////////////////////////////////////////////////////////
//
// THelloApp::MakeDoc
//
////////////////////////////////////////////////////////////////////////
TDoc * THelloApp::MakeDoc(SFReply * /*reply*/){

    return new THelloDoc();

}

////////////////////////////////////////////////////////////////////////
//
// THelloDoc::Draw
//
////////////////////////////////////////////////////////////////////////
void THelloDoc::Draw(Rect * r){

    EraseRect(r);
    TextSize(48);
    MoveTo(20,65);
    DrawString("\phello world");

}
```

```
#-------------------------------------------------------------------------
# Make file for the simplest program using the App and Doc objects
# To use it, use the MPW "Build..." command from the build menu,
# specifying "Helloworld2" and the target file

#© 1990 Dan Weston, All rights reserved

# tell cplus and rez where to find included files for TApp and TDoc
AppObjectDir = ::App-Doc:

# use SADE symbol generation, -sym off will result in faster builds
SymOpts = -sym on

# options for C++, where to look for include files
CPlusOptions = {SymOpts} -i "{AppObjectDir}"

# options for the linker

LinkOptions = -msg nodup {SymOpts}

# options for rez, where to look for include and #include files
RezOptions = -s "{AppObjectDir}" -i "{AppObjectDir}"

# We need to change this rule to include CPlusOptions
.cp.o  ƒ  .cp
    CPlus {default}.cp -o {default}.cp.o {CPlusOptions}

Objects =  ∂
    "{AppObjectDir}"TApp.cp.o ∂
    "{AppObjectDir}"TDoc.cp.o ∂
    Helloworld2.cp.o

ResourceFiles = ∂
    "{AppObjectDir}"TApp.rsrc ∂
    "{AppObjectDir}"TDoc.rsrc ∂
    Helloworld2.rsrc

# dependency rules for TDoc and TApp
"{AppObjectDir}"TDoc.cp.o ƒ "{AppObjectDir}"TDoc.cp ∂
                            "{AppObjectDir}"TDoc.h ∂
                            "{AppObjectDir}"AppDocMenus.h

"{AppObjectDir}"TApp.cp.o ƒ "{AppObjectDir}"TApp.cp ∂
                            "{AppObjectDir}"TApp.h ∂
                            "{AppObjectDir}"TDoc.h ∂
                            "{AppObjectDir}"AppDocMenus.h

# dependency rules for Helloworld2
Helloworld2.cp.o  ƒ Helloworld2.cp ∂
            "{AppObjectDir}"TApp.h ∂
            "{AppObjectDir}"TDoc.h ∂
```

```
            Helloworld2.make

Helloworld2 ƒƒ {Objects} Helloworld2.make
    Link -o {Targ} {LinkOptions} ∂
        {Objects} ∂
        "{CLibraries}"CPlusLib.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Interface.o
    SetFile {Targ} -t APPL -c '????' -a B

Helloworld2 ƒƒ Helloworld2.r ∂
        {ResourceFiles} ∂
        Helloworld2.make
    Rez -append -o {Targ} {RezOptions} Helloworld2.r
```

```
/* Helloworld2.r  rez source for a simple generic application */

// the order of includes is important, since the last
// resources included will replace resources loaded
// before.  We use this to override certain resources in
// TApp and TDoc with the resources in Helloworld.
include "TApp.rsrc";
include "TDoc.rsrc";
include "Helloworld2.rsrc";
```

```
/* HelloWorld2.rsrc.r
 *
 * rez source for HelloWorld2 resources
 *
 * © 1990 Dan Weston All rights reserved
 *
 * Build it with the following rez command
 *
 *  rez HelloWorld2.rsrc.r -o HelloWorld2.rsrc -t rsrc -c RSED
 *
 */


#include "types.r"

resource 'ALRT' (128, purgeable) {
    {68, 76, 172, 376},
    128,
    {   /* array: 4 elements */
        /* [1] */
        OK, visible, silent,
        /* [2] */
        OK, visible, silent,
        /* [3] */
        OK, visible, silent,
        /* [4] */
        OK, visible, silent
    }
};

resource 'DITL' (128, purgeable) {
    {   /* array DITLarray: 5 elements */
        /* [1] */
        {65, 199, 85, 279},
        Button {
            enabled,
            "OK"
        },
        /* [2] */
        {8, 8, 24, 304},
        StaticText {
            disabled,
            "Helloworld2: a very simple application."
        },
        /* [3] */
        {44, 199, 59, 245},
        StaticText {
            disabled,
            ""
        },
        /* [4] */
```

```
        {42, 34, 58, 162},
        StaticText {
            disabled,
            "version 1.0"
        },
        /* [5] */
        {65, 34, 89, 165},
        StaticText {
            disabled,
            "January 15, 1990"
        }
    }
};

resource 'MENU' (128, preload) {
    128,
    textMenuProc,
    0x7FFFFFFD,
    enabled,
    apple,
    {   /* array: 2 elements */
        /* [1] */
        "About Helloworld2", noIcon, noKey, noMark, plain,
        /* [2] */
        "-", noIcon, noKey, noMark, plain
    }
};

resource 'WIND' (1000) {
    {40, 40, 152, 368},
    zoomDocProc,
    invisible,
    goAway,
    0x0,
    Untitled
};
```

Helloworld2.sade **Page 1**

```
directory 'hd:mpw:C++:helloworld2:'

sourcepath  '::App/Doc:', ∂
            '::helloworld2:'

target 'helloworld2'

open source ('helloworld2.cp')
```

```
//////////////////////////////////////////////////////////////////////////
//
// This file: Scribble.cp
//
// This is the main application object for the Scribble program
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////////

#include <Quickdraw.h>
#include <Windows.h>
#include <Memory.h>
#include <Files.h>
#include <Errors.h>

#include "TApp.h"
#include "TDoc.h"

//////////////////////////////////////////////////////////////////////////
//
// constants
//
//////////////////////////////////////////////////////////////////////////

const short  rPenMenu = 131;
const short  i1X1 = 1;
const short  i2X2 = 2;
const short  i3X3 = 3;
const short  iBlack = 5;
const short  iGray = 6;
const short  iWhite = 7;

static const short kEveryItem = 0;

//////////////////////////////////////////////////////////////////////////
//
// utility functions
//
//////////////////////////////////////////////////////////////////////////

inline short min(short a, short b){return (a < b ? a : b);}

//////////////////////////////////////////////////////////////////////////
//
// class declarations
//
//////////////////////////////////////////////////////////////////////////
enum penPat{patBlack,patGray, patWhite};

class TScribbleDoc : public TDoc{
```

```
private:

    short   fPenSize;
    penPat  fPattern;
    PicHandle   fPic;

public:

    TScribbleDoc(OSType theCreator = '????',
                SFReply *reply = (SFReply *)nil);


    // this method does the doodling
    virtual void DoContent(EventRecord *theEvent);

    // take care of our document-specific menus
    virtual void AdjustDocMenus(void);
    virtual Boolean DoDocMenuCommand(short menuID, short menuItem);

protected:

    // draw the picture during updates
    virtual void Draw(Rect *r);

public:

    // make this do nothing so that grow box isn't drawn
    virtual void DoDrawGrowIcon(void){};

    // This is the file type of the document
    virtual OSType GetDocType(){return 'SPCT';}

    // file related methods
    virtual Boolean ReadDocFile(short refNum);
    virtual Boolean WriteDocFile(short refNum);
    virtual Boolean CanSaveAs(void){return true;}

    // override these methods to fiddle with pen menu
    // so that it is enabled when a scribble doc is active
    // and disabled when a scribble doc is disabled or close
    virtual void Activate(void);
    virtual void Deactivate(void);
    virtual Boolean DoClose(void);


protected:
    // new method to actually enable and disable pen menu
    virtual void TogglePenMenu(Boolean  enable);

    // new methods related to pen menu
    void SetPenSize(short p){fPenSize = p;}
```

```
    void SetPenPat(penPat p){fPattern = p;}
    short GetPenSize(void){return fPenSize;}
    penPat GetPenPat(void){return fPattern;}


};


class TScribbleApp : public TApp{

protected:

    // make our kind of document
    virtual TDoc * MakeDoc(SFReply * reply = (SFReply *)nil);

    // yes, we can open old documents
    virtual Boolean CanOpen(void){return true;}

    // file info for SFGetFile and CreateFile
    virtual OSType GetCreator(void){return 'SCBL';}
    virtual int GetNumFileTypes(void){return 1;};
    virtual SFTypeList GetFileTypesList(void);

};


//////////////////////////////////////////////////////////////////////////
//
// Globals
//
//////////////////////////////////////////////////////////////////////////

SFTypeList gtheTypes = {'SPCT'};


//////////////////////////////////////////////////////////////////////////
//
// main
//
//////////////////////////////////////////////////////////////////////////
void main(void)
{
    TScribbleApp theApp;

    // initialize the application
    if(theApp.InitApp()){

        // open one window to start with
        if(! theApp.OpenDocFromFinder())
            theApp.OpenNewDoc();
```

```
        // Start our main event loop running.
        // This won't return until user quits
        theApp.EventLoop();

        //now clean up
        theApp.CleanUp();
    }
}

//////////////////////////////////////////////////////////////////////////
//
// TScribbleApp::MakeDoc
//
//////////////////////////////////////////////////////////////////////////
TDoc * TScribbleApp::MakeDoc(SFReply * reply){

    return new TScribbleDoc(GetCreator(),reply);

}
//////////////////////////////////////////////////////////////////////////
//
// TScribbleApp::GetFileTypesList
//
//////////////////////////////////////////////////////////////////////////
SFTypeList TScribbleApp::GetFileTypesList(void){
    return gtheTypes;
}

//////////////////////////////////////////////////////////////////////////
//
// TScribbleDoc::DoDocMenuCommand
//
//////////////////////////////////////////////////////////////////////////
Boolean TScribbleDoc::DoDocMenuCommand(short menuID, short menuItem){

    if( menuID == rPenMenu){
        switch ( menuItem ){
            case i1X1:
                SetPenSize(1) ;
                break;
            case i2X2:
                SetPenSize(2) ;
                break;
            case i3X3:
                SetPenSize(3) ;
                break;
            case iBlack:
                SetPenPat(patBlack) ;
                break;
            case iGray:
                SetPenPat(patGray) ;
```

```
                break;
            case iWhite:
                SetPenPat(patWhite) ;
                break;
            default:
                return false;   // this should never happen
        }
        // tell the app that we handled this menu item
        return true;
    } else{
        // its not one of our menus, give the parent class a chance
        return TDoc::DoDocMenuCommand(menuID,menuItem);
    }
}
////////////////////////////////////////////////////////////////////////
//
// TScribbleDoc::AdjustDocMenus
//
////////////////////////////////////////////////////////////////////////
void TScribbleDoc::AdjustDocMenus(void){

    // Do the pen menu
    MenuHandle  menu = GetMHandle(rPenMenu);

    CheckItem(menu,i1X1,GetPenSize() == 1);
    CheckItem(menu,i2X2,GetPenSize() == 2);
    CheckItem(menu,i3X3,GetPenSize() == 3);
    CheckItem(menu,iBlack,GetPenPat() == patBlack);
    CheckItem(menu,iGray,GetPenPat() == patGray);
    CheckItem(menu,iWhite,GetPenPat() == patWhite);

    // now let the parent class have a shot at the menus
    TDoc::AdjustDocMenus();

}
////////////////////////////////////////////////////////////////////////
//
// TScribbleDoc::TogglePenMenu
//
////////////////////////////////////////////////////////////////////////
void TScribbleDoc::TogglePenMenu(Boolean enable){

    MenuHandle  menu = GetMHandle(rPenMenu);

    SetMenuAbility(menu,kEveryItem,enable);
    DrawMenuBar();
}

////////////////////////////////////////////////////////////////////////
//
// TScribbleDoc::Activate
```

```
//
//////////////////////////////////////////////////////////////////////////
void TScribbleDoc::Activate(void){

    TDoc::Activate();
    TogglePenMenu(true);
}

//////////////////////////////////////////////////////////////////////////
//
// TScribbleDoc::Deactivate
//
//////////////////////////////////////////////////////////////////////////
void TScribbleDoc::Deactivate(void){

    TDoc::Deactivate();
    TogglePenMenu(false);

}
//////////////////////////////////////////////////////////////////////////
//
// TScribbleDoc::DoClose
//
//////////////////////////////////////////////////////////////////////////
Boolean TScribbleDoc::DoClose(void){

    if(TDoc::DoClose()){
        TogglePenMenu(false);
        return true;
    } else
        return false;
}

//////////////////////////////////////////////////////////////////////////
//
// TScribbleDoc::TScribbleDoc
//
//////////////////////////////////////////////////////////////////////////
TScribbleDoc::TScribbleDoc(OSType theCreator,SFReply *reply ):
    TDoc(theCreator,reply) {

    fPenSize = 2;
    fPattern = patGray;
    fPic = nil;

}
//////////////////////////////////////////////////////////////////////////
//
// TScribbleDoc::Draw
//
//////////////////////////////////////////////////////////////////////////
```

```
void TScribbleDoc::Draw(Rect * /*r*/){

    if(fPic != nil)
        DrawPicture(fPic,&((**fPic).picFrame));

}
//////////////////////////////////////////////////////////////////////
//
// TScribbleDoc::DoContent
//
//////////////////////////////////////////////////////////////////////
void TScribbleDoc::DoContent(EventRecord* theEvent) {

    Point newPoint;

    if(fDocWindow){
        SetPort(fDocWindow);
        PenSize(fPenSize,fPenSize);

        if(fPattern == patBlack)
            PenPat(qd.black);
        if(fPattern == patGray)
            PenPat(qd.gray);
        if(fPattern == patWhite)
            PenPat(qd.white);


        GlobalToLocal(&theEvent->where);
        MoveTo(theEvent->where.h, theEvent->where.v);
        do{
            GetMouse(&newPoint);
            LineTo(newPoint.h, newPoint.v);
        }while(StillDown());

        fNeedtoSave = true;

        // now take a snap shot of window
        if(fPic != nil)
            KillPicture(fPic);
        fPic = OpenPicture(&fDocWindow->portRect);
        CopyBits(&fDocWindow->portBits,
                &fDocWindow->portBits,
                &fDocWindow->portRect,
                &fDocWindow->portRect,
                srcCopy,
                (RgnHandle)nil);
        ClosePicture();
    }
}

//////////////////////////////////////////////////////////////////////
```

```
//
// TScribbleDoc::ReadDocFile
//
//////////////////////////////////////////////////////////////////////////
Boolean TScribbleDoc::ReadDocFile(short refNum){
const short kHAdjust = 50;
const short kWAdjust = 40;

    if(fDocWindow){

        long len;
        OSErr err = GetEOF(refNum,&len);
        Handle thePic = NewHandle(len);
        if(thePic == nil){
            ErrorAlert(rDocErrorStrings,sNoMem);
            return false;
        }

        HLock(thePic);
        err = SetFPos(refNum,fsFromStart,0);
        err = FSRead(refNum,&len,(Ptr)*thePic);
        HUnlock(thePic);
        if(err == noErr){
            // now make the window the size of the picture
            Rect    r = (**((PicHandle)thePic)).picFrame;
            short height = r.bottom - r.top;
            short width = r.right - r.left;
            r = qd.screenBits.bounds;

            height = min(height ,r.bottom - r.top - kHAdjust);
            width = min(width, r.right - r.left - kWAdjust);
            SizeWindow(fDocWindow, width, height, true);

            // set the member to reference Picture
            fPic = (PicHandle)thePic;
            return true;
        } else {
            DisposHandle(thePic);
            return false;
        }
    }
    // if there ain't no window...
    return false;
}

//////////////////////////////////////////////////////////////////////////
//
// TScribbleDoc::WriteDocFile
//
//////////////////////////////////////////////////////////////////////////
Boolean TScribbleDoc::WriteDocFile(short refNum){
```

```
    if(fDocWindow){
        if(fPic != nil){
            long len = GetHandleSize((Handle)fPic);
            HLock((Handle)fPic);
            OSErr err = SetFPos(refNum,fsFromStart,0);
            err = FSWrite(refNum,&len,(Ptr)*fPic);
            HUnlock((Handle)fPic);
            if(err == noErr)
                return true;
            else
                return false;

        }
    }
    // if there ain't no window...
    return false;

}
```

```
#--------------------------------------------------------------------
# Make file for the Scribble program using the App and Doc objects
# To use it, use the MPW "Build..." command from the build menu,
# specifying "Scribble" and the target file

#© 1990 Dan Weston, All rights reserved

# tell cplus and rez where to find included files for TApp and TDoc
AppObjectDir = ::App-Doc:

# use SADE symbol generation, -sym off will result in faster builds
SymOpts = -sym on

# options for C++, where to look for include files
CPlusOptions = {SymOpts} -i "{AppObjectDir}"

# options for the linker

LinkOptions = -msg nodup {SymOpts}

# options for rez, where to look for include and #include files
RezOptions = -s "{AppObjectDir}" -i "{AppObjectDir}"

# We need to change this rule to include CPlusOptions
.cp.o  ƒ  .cp
    CPlus {default}.cp -o {default}.cp.o {CPlusOptions}

.cp.o  ƒ  .h
    CPlus {default}.cp -o {default}.cp.o {CPlusOptions}

Objects = ∂
    "{AppObjectDir}"TApp.cp.o ∂
    "{AppObjectDir}"TDoc.cp.o ∂
    Scribble.cp.o

ResourceFiles = ∂
    "{AppObjectDir}"TApp.rsrc ∂
    "{AppObjectDir}"TDoc.rsrc ∂
    Scribble.rsrc

# dependency rules for TDoc and TApp
"{AppObjectDir}"TDoc.cp.o ƒ "{AppObjectDir}"TDoc.cp ∂
                            "{AppObjectDir}"TDoc.h ∂
                            "{AppObjectDir}"AppDocMenus.h

"{AppObjectDir}"TApp.cp.o ƒ "{AppObjectDir}"TApp.cp ∂
                            "{AppObjectDir}"TApp.h ∂
                            "{AppObjectDir}"TDoc.h ∂
                            "{AppObjectDir}"AppDocMenus.h

# dependency rules for Scribble
```

```
Scribble.cp.o  ƒ Scribble.cp ∂
            "{AppObjectDir}"TApp.h ∂
            "{AppObjectDir}"TDoc.h ∂
            Scribble.make

Scribble ƒƒ {Objects} Scribble.make
    Link -o {Targ} {LinkOptions} ∂
        {Objects} ∂
        "{CLibraries}"CPlusLib.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Interface.o
    SetFile {Targ} -t APPL -c SCBL -a B

Scribble ƒƒ Scribble.r ∂
        {ResourceFiles}
    Rez -append -o {Targ} {RezOptions} Scribble.r
```

```
/* Scribble.r  rez source for the scribble application */

include "TApp.rsrc";
include "TDoc.rsrc";
include "Scribble.rsrc";
```

```
/* Scribble.rsrc.r
 *
 * rez source for Scribble resources
 *
 * © 1990 Dan Weston All rights reserved
 *
 * Build it with the following rez command
 *
 *  rez Scribble.rsrc.r -o Scribble.rsrc -t rsrc -c RSED
 *
 */

#include "types.r"

resource 'MENU' (128, preload) {
    128,
    textMenuProc,
    0x7FFFFFFD,
    enabled,
    apple,
    {   /* array: 2 elements */
        /* [1] */
        "About Scribble...", noIcon, noKey, noMark, plain,
        /* [2] */
        "-", noIcon, noKey, noMark, plain
    }
};

resource 'MENU' (131) {
    131,
    textMenuProc,
    allEnabled,
    enabled,
    "Pen",
    {   /* array: 7 elements */
        /* [1] */
        "1 X 1", noIcon, noKey, check, plain,
        /* [2] */
        "2 X 2", noIcon, noKey, noMark, plain,
        /* [3] */
        "3 X 3", noIcon, noKey, noMark, plain,
        /* [4] */
        "-", noIcon, noKey, noMark, plain,
        /* [5] */
        "Black", noIcon, noKey, check, plain,
        /* [6] */
        "Gray", noIcon, noKey, noMark, plain,
        /* [7] */
        "White", noIcon, noKey, noMark, plain
    }
};
```

```
resource 'DITL' (128, purgeable) {
    {   /* array DITLarray: 5 elements */
        /* [1] */
        {65, 199, 85, 279},
        Button {
            enabled,
            "OK"
        },
        /* [2] */
        {8, 8, 24, 304},
        StaticText {
            disabled,
            "Scribble: a very simple application."
        },
        /* [3] */
        {44, 199, 59, 245},
        StaticText {
            disabled,
            ""
        },
        /* [4] */
        {42, 34, 58, 162},
        StaticText {
            disabled,
            "version 1.0"
        },
        /* [5] */
        {65, 34, 89, 165},
        StaticText {
            disabled,
            "February 15, 1990"
        }
    }
};

resource 'ALRT' (128, purgeable) {
    {68, 76, 172, 376},
    128,
    {   /* array: 4 elements */
        /* [1] */
        OK, visible, silent,
        /* [2] */
        OK, visible, silent,
        /* [3] */
        OK, visible, silent,
        /* [4] */
        OK, visible, silent
    }
};
```

```
resource 'BNDL' (128) {
    'SCBL',
    0,
    {   /* array TypeArray: 2 elements */
        /* [1] */
        'ICN#',
        {   /* array IDArray: 2 elements */
            /* [1] */
            0, 128,
            /* [2] */
            1, 129
        },
        /* [2] */
        'FREF',
        {   /* array IDArray: 2 elements */
            /* [1] */
            0, 128,
            /* [2] */
            1, 129
        }
    }
};

resource 'FREF' (129) {
    'SPCT',
    1,
    ""
};

resource 'FREF' (128) {
    'APPL',
    0,
    ""
};

resource 'ICN#' (129, preload) {
    {   /* array: 2 elements */
        /* [1] */
        $"0FFF FE00 0800 0300 0812 6280 089B 4E40"
        $"0889 4A20 08CB 4A10 0A5F FBF8 0B70 1CE8"
        $"0920 0708 09C0 0228 08C0 03E8 0CC0 0118"
        $"0B0E 71F8 088E 7088 088E F088 0880 8088"
        $"0880 C088 0880 4088 0843 C088 0840 0108"
        $"0820 1108 082C 7108 0823 C308 0810 0608"
        $"0818 1C08 080F F008 0800 0008 0800 0008"
        $"0800 0008 0800 0008 0800 0008 0FFF FFF8",
        /* [2] */
        $"0FFF FE00 0FFF FF00 0FFF FF80 0FFF FFC0"
        $"0FFF FFE0 0FFF FFF0 0FFF FFF8 0FFF FFF8"
        $"0FFF FFF8 0FFF FFF8 0FFF FFF8 0FFF FFF8"
        $"0FFF FFF8 0FFF FFF8 0FFF FFF8 0FFF FFF8"
```

```
        $"0FFF FFF8 0FFF FFF8 0FFF FFF8 0FFF FFF8"
        $"0FFF FFF8 0FFF FFF8 0FFF FFF8 0FFF FFF8"
        $"0FFF FFF8 0FFF FFF8 0FFF FFF8 0FFF FFF8"
        $"0FFF FFF8 0FFF FFF8 0FFF FFF8 0FFF FFF8"
    }
};

resource 'ICN#' (128, preload) {
    {   /* array: 2 elements */
        /* [1] */
        $"0001 0000 0002 8000 0004 4000 0008 2000"
        $"0010 1000 0025 4800 0055 5400 0097 D200"
        $"015C 7500 0270 3480 0540 1C40 09C0 0B20"
        $"1680 0E10 239D C808 411D FF04 809D C082"
        $"4086 8041 2085 3022 1087 C814 087E 7F8F"
        $"044A 3007 0227 0007 0111 8007 008F E007"
        $"0040 1FE7 0020 021F 0010 0407 0008 0800"
        $"0004 1000 0002 2000 0001 4000 0000 80",
        /* [2] */
        $"0001 0000 0003 8000 0007 C000 000F E000"
        $"001F F000 003F F800 007F FC00 00FF FE00"
        $"01FF FF00 03FF FF80 07FF FFC0 0FFF FFE0"
        $"1FFF FFF0 3FFF FFF8 7FFF FFFC FFFF FFFE"
        $"7FFF FFFF 3FFF FFFE 1FFF FFFC 0FFF FFFF"
        $"07FF FFFF 03FF FFFF 01FF FFFF 00FF FFFF"
        $"007F FFFF 003F FE1F 001F FC07 000F F800"
        $"0007 F000 0003 E000 0001 C000 0000 80"
    }
};

resource 'MBAR' (128) {
    {   /* array MenuArray: 4 elements */
        /* [1] */
        128,
        /* [2] */
        129,
        /* [3] */
        130,
        /* [4] */
        131
    }
};

data 'SCBL' (0) {
    $"1F53 6372 6962 626C 652C 4665 6220 3135"
    /* .Scribble,Feb 15 */
    $"2C31 3939 3020 4461 6E20 5765 7374 6F6E"
    /* ,1990 Dan Weston */
};
```

```
directory 'hd:mpw:C++:Scribble:'

sourcepath '::App/Doc:', ∂
           '::Scribble:'

target "Scribble"       # assumes symbol file is 'application name.SYM'

open source ('Scribble.cp')
```

```
////////////////////////////////////////////////////////////////////////
//
// This is the generic modeless dialog document object
//
// © 1990 Dan Weston, All Rights Reserved
//
////////////////////////////////////////////////////////////////////////


#ifndef TModelessDoc_Defs
#define TModelessDoc_Defs

// Include necessary interface files
#include <Types.h>
#include <Quickdraw.h>
#include <Windows.h>
#include <Packages.h>
#include <TDoc.h>


////////////////////////////////////////////////////////////////////////
//
// constants
//
////////////////////////////////////////////////////////////////////////

const short rGenericDialog = 1000;


////////////////////////////////////////////////////////////////////////
//
// class declarations
//
////////////////////////////////////////////////////////////////////////

class TModelessDoc : public TDoc {

public:

    // SFInfo will be non-nil when opening an existing document
    TModelessDoc(OSType theCreator = '????',
                 SFReply * SFInfo = (SFReply *)nil):
        TDoc(theCreator,SFInfo){};

    // virtual destructor so that derived destructors will be called
    virtual ~TModelessDoc();

    // called by TApp when making a document,
    virtual Boolean MakeWindow(Boolean colorWindow );

    virtual short GetWinID(void);
```

```
// Event actions
    virtual void DoIdle(void);

    virtual void DoActivate(EventRecord* theEvent)
        {DoDialogEvent(theEvent);}
    virtual void DoTheUpdate(EventRecord* theEvent)
        {DoDialogEvent(theEvent);}
    virtual void DoContent(EventRecord* theEvent)
        {DoDialogEvent(theEvent);}
    virtual void DoKeyDown(EventRecord* theEvent)
        {DoDialogEvent(theEvent);}

    // disable grow actions
    virtual void DoGrow(EventRecord* theEvent){}
    virtual void DoDrawGrowIcon(void)
        {}

protected:
    virtual void DoDialogEvent(EventRecord* theEvent);
    virtual void DoItemHit(DialogPtr theDialog,short theItem){}

};

#endif TModelessDoc_Defs
```

```
//////////////////////////////////////////////////////////////////////////
//
// This is the generic modeless dialog document object
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////////

// Mac Includes
#include <Types.h>
#include <Windows.h>
#include <OSUtils.h>
#include <Files.h>
#include <Errors.h>

#include "TModelessDoc.h"

// define a segment for the Modeless Doc code
#pragma segment ModelessSeg

//////////////////////////////////////////////////////////////////////////
//
// TModelessDoc::GetWinID
//
//////////////////////////////////////////////////////////////////////////
short TModelessDoc::GetWinID(void){
    return rGenericDialog;
}

//////////////////////////////////////////////////////////////////////////
//
// TModelessDoc::MakeWindow
//
//////////////////////////////////////////////////////////////////////////
Boolean TModelessDoc::MakeWindow(Boolean /*colorWindow*/){

    fDocWindow = (WindowPtr)GetNewDialog(GetWinID(),nil,(WindowPtr)-1);

    return (fDocWindow != nil);

}

//////////////////////////////////////////////////////////////////////////
//
// TModelessDoc::~TModelessDoc
//
//////////////////////////////////////////////////////////////////////////
TModelessDoc::~TModelessDoc(void){

    if(fDocWindow){
        DisposDialog((DialogPtr)fDocWindow);
```

```
        fDocWindow = nil;
    }
}
////////////////////////////////////////////////////////////////////////
//
// TModelessDoc::DoDialogEvent
//
////////////////////////////////////////////////////////////////////////
void TModelessDoc::DoDialogEvent(EventRecord* theEvent){

    short       itemHit;
    DialogPtr   theDialog;

    if (IsDialogEvent(theEvent)){
        if(DialogSelect(theEvent,&theDialog,&itemHit))
            DoItemHit(theDialog,itemHit);
    }
}

////////////////////////////////////////////////////////////////////////
//
// TModelessDoc::DoIdle
//
////////////////////////////////////////////////////////////////////////
void TModelessDoc::DoIdle(void){

    // this is necessary so that the insertion point
    //  will blink in edit text dialog items
    EventRecord theEvent;
    theEvent.what = nullEvent;

    DoDialogEvent(&theEvent);
}
```

```
////////////////////////////////////////////////////////////////////////
//
//
//  This file: ModelessApp.cp   -   C++ Source
//
//  This is the main application object a simple
//  application program using TModelessDoc
//
// © 1990 Dan Weston, All Rights Reserved
//
////////////////////////////////////////////////////////////////////////
#include "TApp.h"
#include "TDoc.h"
#include "TModelessDoc.h"

////////////////////////////////////////////////////////////////////////
//
// constants
//
////////////////////////////////////////////////////////////////////////

const short  iOK = 1;
const short  iUserItem = 2;

////////////////////////////////////////////////////////////////////////
//
// class declarations
//
////////////////////////////////////////////////////////////////////////

class TModelessApp : public TApp{

    virtual TDoc * MakeDoc(SFReply * reply = (SFReply *) nil);

};

class TSampDlg : public TModelessDoc{

public:
    TSampDlg(OSType creator,SFReply * theReply);

    virtual Boolean InitDoc(void);

protected:
    virtual void DoItemHit(DialogPtr theDialog,short theItem);

};
////////////////////////////////////////////////////////////////////////
//
// main
//
```

```
////////////////////////////////////////////////////////////////////////
void main(void)
{
    // create an instance of TModelessApp
    TModelessApp theApp ;

    // initialize the application
    if(theApp.InitApp()){

        // open one window to start with,
        // unless we got files from the Finder
        if(! theApp.OpenDocFromFinder())
            theApp.OpenNewDoc();

        // run the event loop until user quits
        theApp.EventLoop();

        //now clean up
        theApp.CleanUp();
    }
}

////////////////////////////////////////////////////////////////////////
//
// TModelessApp::MakeDoc
//
////////////////////////////////////////////////////////////////////////
TDoc * TModelessApp::MakeDoc(SFReply * reply){

    return new TSampDlg(GetCreator(),reply);

}

////////////////////////////////////////////////////////////////////////
//
// TSampDlg::TSampDlg
//
////////////////////////////////////////////////////////////////////////
TSampDlg::TSampDlg(OSType creator,SFReply * theReply):
    TModelessDoc(creator,theReply){

}

////////////////////////////////////////////////////////////////////////
//
// TSampDlg::InitDoc
//
////////////////////////////////////////////////////////////////////////
Boolean TSampDlg::InitDoc(void){

    // install user item proc
```

```
    void pascal UserItemProc(WindowPtr theWindow,short theItem);

    Rect    theRect;
    short   theType;
    Handle  theItem;

    if(TModelessDoc::InitDoc()){
        GetDItem((DialogPtr)fDocWindow,
                iUserItem,
                &theType,
                &theItem,
                &theRect);
        SetDItem((DialogPtr)fDocWindow,
                iUserItem,
                theType,
                (Handle)UserItemProc,
                &theRect);

        return true;
    }

    else
        return false;

}

////////////////////////////////////////////////////////////////////////
//
// TSampDlg::DoItemHit
//
////////////////////////////////////////////////////////////////////////
void TSampDlg::DoItemHit(DialogPtr /*theDialog*/,short theItem){

    if(theItem == iOK)
        SysBeep(1);

}

////////////////////////////////////////////////////////////////////////
//
// UserItemProc
//
////////////////////////////////////////////////////////////////////////
void pascal UserItemProc(WindowPtr theWindow, short theItem){

    Rect    r;
    short   theType;
    Handle  theItemH;
    short   width;

    GetDItem((DialogPtr)theWindow,
```

```
            theItem,
            &theType,
            &theItemH,
            &r);
    width = (r.right - r.left) ;

    EraseRect(&r);
    FrameRect(&r);
    for(short i = width / 2; i > 0; i -= 2){
        InsetRect(&r,2,2);
        FrameRect(&r);
    }
}
```

ModelessApp.make **Page 1**

```
#-------------------------------------------------------------------
# Make file for the simplest program using the ModelessDoc objects
# To use it, use the MPW "Build..." command from the build menu,
# specifying "ModelessApp" and the target file

#© 1990 Dan Weston, All rights reserved

# tell cplus and rez where to find included files for TApp,TDoc,
# and TModelessDoc
AppObjectDir = ::App-Doc:
ModelessObjDir = ::ModelessDoc:

# use SADE symbol generation, -sym off will result in faster builds
SymOpts = -sym on

# options for C++, where to look for include files
CPlusOptions = {SymOpts} -i "{AppObjectDir}" -i "{ModelessObjDir}"

# options for the linker

LinkOptions = -msg nodup {SymOpts}

# options for rez, where to look for include and #include files
RezOptions = -s "{AppObjectDir}" -i "{AppObjectDir}"

# We need to change this rule to include CPlusOptions
.cp.o  ƒ  .cp
    CPlus {default}.cp -o {default}.cp.o {CPlusOptions}

Objects = ∂
    "{AppObjectDir}"TApp.cp.o ∂
    "{AppObjectDir}"TDoc.cp.o ∂
    "{ModelessObjDir}"TModelessDoc.cp.o ∂
    ModelessApp.cp.o

ResourceFiles = ∂
    "{AppObjectDir}"TApp.rsrc ∂
    "{AppObjectDir}"TDoc.rsrc ∂
    ModelessApp.rsrc

# dependency rules for TDoc and TApp
"{AppObjectDir}"TDoc.cp.o ƒ "{AppObjectDir}"TDoc.cp ∂
                           "{AppObjectDir}"TDoc.h ∂
                           "{AppObjectDir}"AppDocMenus.h

"{AppObjectDir}"TApp.cp.o ƒ "{AppObjectDir}"TApp.cp ∂
                           "{AppObjectDir}"TApp.h ∂
                           "{AppObjectDir}"TDoc.h ∂
                           "{AppObjectDir}"AppDocMenus.h

# dependency rules for TModelessDoc
```

```
"{ModelessObjDir}"TModelessDoc.cp.o ƒ ∂
                "{ModelessObjDir}"TModelessDoc.cp ∂
                "{ModelessObjDir}"TModelessDoc.h ∂
                "{AppObjectDir}"TDoc.h

# dependency rules for ModelessApp
ModelessApp.cp.o  ƒ ModelessApp.cp ∂
            "{AppObjectDir}"TApp.h ∂
            "{AppObjectDir}"TDoc.h ∂
            "{ModelessObjDir}"TModelessDoc.h ∂
            ModelessApp.make

ModelessApp ƒƒ {Objects} ModelessApp.make
    Link -o {Targ} {LinkOptions} ∂
        {Objects} ∂
        "{CLibraries}"CPlusLib.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Interface.o
    SetFile {Targ} -t APPL -c '????' -a B

ModelessApp ƒƒ ModelessApp.r ∂
        {ResourceFiles}
    Rez -append -o {Targ} {RezOptions} ModelessApp.r
```

```
// ModelessApp.r
// gather the resources for simple program
// that uses TModelessDoc's

include "TApp.rsrc";
include "TDoc.rsrc";
include "ModelessApp.rsrc";
```

```
/* modelessapp.rsrc.r
 *
 * rez source for modelessapp resources
 *
 * © 1990 Dan Weston All rights reserved
 *
 * Build it with the following rez command
 *
 *  rez modelessapp.rsrc.r -o modelessapp.rsrc -t rsrc -c RSED
 *
 */

#include "types.r"

resource 'DITL' (1000) {
    {   /* array DITLarray: 2 elements */
        /* [1] */
        {99, 194, 119, 254},
        Button {
            enabled,
            "OK"
        },
        /* [2] */
        {11, 33, 135, 177},
        UserItem {
            enabled
        }
    }
};

resource 'DLOG' (1000) {
    {40, 40, 190, 324},
    documentProc,
    invisible,
    goAway,
    0x0,
    1000,
    "New Dialog"
};
```

ModelessApp.sade **Page 1**

```
directory 'hd:mpw:C++:ModelessApp:'

sourcepath '::App/Doc:',∂
        '::ModelessDoc:',∂
        '::ModelessApp:'

target 'ModelessApp'    # assumes symbol file is 'application name.SYM'

open source ('ModelessApp.cp')
```

```
//////////////////////////////////////////////////////////////////////
//
//  This is the generic scrolling document object
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////

#ifndef TScrollDoc_Defs
#define TScrollDoc_Defs

// Include necessary interface files
#include <Types.h>
#include <Quickdraw.h>
#include <Windows.h>
#include <Packages.h>
#include <TDoc.h>

//////////////////////////////////////////////////////////////////////
//
// class declarations
//
//////////////////////////////////////////////////////////////////////

class TScrollDoc : public TDoc {
protected:
    ControlHandle   fHorizScrollBar;
    ControlHandle   fVertScrollBar ;

    short   fVOffset;
    short   fHOffset;

    static TScrollDoc *fCurrScrollDoc;

public:
    static TScrollDoc *GetCurrScrollDoc(void){return fCurrScrollDoc;}

    ControlHandle   GetVScroll(void){return fVertScrollBar;}
    ControlHandle   GetHScroll(void){return fHorizScrollBar;}

    // SFInfo will be non-nil when opening an existing document
    TScrollDoc(OSType theCreator = '????',
               SFReply * SFInfo = (SFReply *)nil);

    // virtual destructor so that derived destructors will be called
    virtual ~TScrollDoc(){}

    virtual Boolean InitDoc(void);

// Event actions that are different from TDoc
    // you probably won't need to override these
```

```
    virtual void DoContent(EventRecord* theEvent);
    virtual void DoTheUpdate(EventRecord *theEvent);
    virtual void DoZoom(short partCode);
    virtual void DoGrow(EventRecord *theEvent);

    // override these for activation deactivation stuff
    // be sure and call TScrollDoc::Activate or
    // TScrollDoc::Deactivate in your override functions.
    virtual void Activate(void);
    virtual void Deactivate(void);

protected:
// new functions to support scrolling
    // you probably won't need to override these
    virtual void ScrollClick(EventRecord *theEvent);

    virtual void DoThumbScroll(ControlHandle theControl,Point localPt);
    virtual void DoPageScroll(ControlHandle theControl,short part);
    virtual void DoButtonScroll(ControlHandle theControl,Point localPt);

    virtual void SizeScrollBars(void);
    virtual void AdjustScrollBars(void);
    virtual void SetScrollBarValues(void);
    virtual void SynchScrollBars(void);

public:
    virtual void FocusOnWindow();
    virtual void FocusOnContent();
    virtual void Scroll(ControlHandle theControl,short change);
    virtual void GetContentRect(Rect& r);

    // routines you must override
    virtual short GetVertSize(void){return 0;}
    virtual short GetHorizSize(void){return 0;}
    virtual short GetVertLineScrollAmount(void){return 0;}
    virtual short GetHorizLineScrollAmount(void){return 0;}

    // override these only if you don't want a page
    // scroll to be one window full
    virtual short GetVertPageScrollAmount(void);
    virtual short GetHorizPageScrollAmount(void);

    // routines you might override
    virtual void ContentClick(EventRecord *theEvent){}
    virtual void ScrollContents(short dh,short dv);
};
#endif TScrollDoc_Defs
```

```
//////////////////////////////////////////////////////////////////////////
//
//  This is the generic scrolling document object
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////////

#include <Types.h>
#include <Windows.h>
#include <OSUtils.h>
#include <Files.h>
#include <Errors.h>
#include <Memory.h>
#include <SysEqu.h>

#include "TScrollDoc.h"

// define a segment for the ScrollDoc code
#pragma segment ScrollSeg

//////////////////////////////////////////////////////////////////////////
//
// constants
//
//////////////////////////////////////////////////////////////////////////

// how much of the screen to leave during page scroll
const short kScrollOverlap = 16;

// resource ID's for scroll bars
const rVScroll =    128;
const rHScroll =    129;

//////////////////////////////////////////////////////////////////////////
//
// static members are defined like globals
//
//////////////////////////////////////////////////////////////////////////
TScrollDoc *TScrollDoc::fCurrScrollDoc = nil;

//////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::TScrollDoc
//
//////////////////////////////////////////////////////////////////////////
TScrollDoc::TScrollDoc(OSType theCreator,SFReply * SFInfo):
    TDoc(theCreator,SFInfo){

    fHorizScrollBar = nil;
    fVertScrollBar = nil;
```

```
    fVOffset = 0;
    fHOffset = 0;
}
//////////////////////////////////////////////////////////////////////
//
// TScrollDoc::InitDoc
//
//////////////////////////////////////////////////////////////////////
Boolean TScrollDoc::InitDoc(void){

    if( ! TDoc::InitDoc())
        return false;

    if(fDocWindow != nil){
        SetPort(fDocWindow);
        fHorizScrollBar = GetNewControl(rHScroll,fDocWindow);
        fVertScrollBar = GetNewControl(rVScroll,fDocWindow);
        SizeScrollBars();
        SynchScrollBars();
    }
    return ((fHorizScrollBar != nil) && (fVertScrollBar != nil));
}
//////////////////////////////////////////////////////////////////////
//
// TScrollDoc::SizeScrollBars
//
//////////////////////////////////////////////////////////////////////
void TScrollDoc::SizeScrollBars(void){

    if(fDocWindow != nil){
        FocusOnWindow();
        Rect r = fDocWindow->portRect;

        if(fVertScrollBar != nil){
            HideControl(fVertScrollBar);
            SizeControl(fVertScrollBar,
                        kScrollBarWidth,
                        (r.bottom - r.top - kScrollBarPos) + 2);

            MoveControl(fVertScrollBar,
                        r.right - kScrollBarPos,
                        -1);
            ShowControl(fVertScrollBar);
            ValidRect(&(**fVertScrollBar).contrlRect);
        }

        if(fHorizScrollBar != nil){
            HideControl(fHorizScrollBar);
            SizeControl(fHorizScrollBar,
                        (r.right - r.left - kScrollBarPos) + 2,
                        kScrollBarWidth);
```

```
            MoveControl(fHorizScrollBar,
                        -1,
                        r.bottom - r.top - kScrollBarPos);
            ShowControl(fHorizScrollBar);
            ValidRect(&(**fHorizScrollBar).contrlRect);
        }
    }
}
////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::AdjustScrollBars
//
////////////////////////////////////////////////////////////////////////
void TScrollDoc::AdjustScrollBars(void){

    // don't activate the scroll bars until
    // the data extends beyond the window boundaries
    // If currentCtlValue is greater than new ctlmax ,
    // scroll image to bring it in line
    Rect r ;
    GetContentRect(r);
    short dh,dv;
    dh =  dv = 0;
    short currentValue;
    short newMax;
    // now ask the document how big its image is
    // first for the vertical dimension
    if(fVertScrollBar != nil){
        currentValue = GetCtlValue(fVertScrollBar);
        newMax = GetVertSize() - (r.bottom - r.top);
        if(newMax < 0)
            newMax = 0;
        if(currentValue > newMax)
            dv = currentValue - newMax;
        SetCtlMax(fVertScrollBar,newMax);
    }

    if(fHorizScrollBar != nil){
        currentValue = GetCtlValue(fHorizScrollBar);
        newMax = GetHorizSize() - (r.right - r.left);
        if(newMax < 0)
            newMax = 0;
        if(currentValue > newMax)
            dh = currentValue - newMax;
        SetCtlMax(fHorizScrollBar,newMax);
    }

    // adjust the position of the image if the window
    // has gotten bigger than the image.
    if(dh | dv){
```

```
        FocusOnContent();
        // invalidate the whole content area
        GetContentRect(r);
        InvalRect(&r);

        // shut the clip region down to zero
        // so that the scrolling won't actually
        // draw in the window, wait for update instead
        RgnHandle   oldClip = NewRgn();
        GetClip(oldClip);
        SetRect(&r,0,0,0,0);
        ClipRect(&r);
        ScrollContents(dh,dv);
        // now reset the clip region
        SetClip(oldClip);
        DisposeRgn(oldClip);
    }
}
////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::SynchScrollBars
//
////////////////////////////////////////////////////////////////////////
void TScrollDoc::SynchScrollBars(void){

    AdjustScrollBars();
    SetScrollBarValues();

}
////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::SetScrollBarValues
//
////////////////////////////////////////////////////////////////////////
void TScrollDoc::SetScrollBarValues(void){

    FocusOnWindow();
    if(fHorizScrollBar != nil)
        SetCtlValue(fHorizScrollBar,fHOffset);
    if(fVertScrollBar != nil)
        SetCtlValue(fVertScrollBar,fVOffset);
}
////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::FocusOnWindow
//
////////////////////////////////////////////////////////////////////////
void TScrollDoc::FocusOnWindow(){

    SetPort(fDocWindow);
    SetOrigin(0,0);
```

```
    Rect r = fDocWindow->portRect;
    ClipRect(&r);
}
////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::FocusOnContent
//
////////////////////////////////////////////////////////////////////////
void TScrollDoc::FocusOnContent(){

    SetPort(fDocWindow);
    SetOrigin(fHOffset,fVOffset);
    Rect r;
    GetContentRect(r);
    ClipRect(&r);
}
////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::GetVertPageScrollAmount
//
////////////////////////////////////////////////////////////////////////
short TScrollDoc::GetVertPageScrollAmount(void){

    Rect r;
    GetContentRect(r);
    return r.bottom - r.top - kScrollOverlap;

}
////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::GetHorizPageScrollAmount
//
////////////////////////////////////////////////////////////////////////
short TScrollDoc::GetHorizPageScrollAmount(void){

    Rect r;
    GetContentRect(r);
    return r.right - r.left - kScrollOverlap;

}
////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::GetContentRect
//
////////////////////////////////////////////////////////////////////////
void TScrollDoc::GetContentRect(Rect& r){

    // how big is the content area of the window, discounting the
    // scroll bars
    r = fDocWindow->portRect;
    if(fVertScrollBar != nil)
```

```
        r.right -= kScrollBarPos;
    if(fHorizScrollBar != nil)
        r.bottom -= kScrollBarPos;

}
//////////////////////////////////////////////////////////////////////
//
// TScrollDoc::ScrollClick
//
//////////////////////////////////////////////////////////////////////
void TScrollDoc::ScrollClick(EventRecord *theEvent){

    ControlHandle whichControl;
    short part;
    FocusOnWindow();
    if(part = FindControl(theEvent->where,fDocWindow,&whichControl)){
        switch (part){
            case inThumb:
                DoThumbScroll(whichControl,theEvent->where);
                break;

            case inUpButton:
            case inDownButton:
                DoButtonScroll(whichControl,theEvent->where);
                break;

            case inPageUp:
            case inPageDown:
                DoPageScroll(whichControl,part);
                break;

        }
    }
}
//////////////////////////////////////////////////////////////////////
//
// TScrollDoc::DoButtonScroll
//
//////////////////////////////////////////////////////////////////////
void TScrollDoc::DoButtonScroll(ControlHandle theControl,Point localPt){

    // declare the action procedure
    pascal void ActionProc(ControlHandle theControl,short partCode);

    short result = TrackControl(theControl,
                                localPt,
                                (ProcPtr)ActionProc);

}
//////////////////////////////////////////////////////////////////////
//
```

```
// TScrollDoc::DoPageScroll
//
//////////////////////////////////////////////////////////////////////
void TScrollDoc::DoPageScroll(ControlHandle theControl,short part){

    short scrollAmount;
    Point thePt;
    short currentPart;

    if((theControl == fVertScrollBar))
        scrollAmount = GetVertPageScrollAmount();
    else
        scrollAmount = GetHorizPageScrollAmount();

    // repeat as long as user holds down mouse button
    do {
        GetMouse(&thePt);
        currentPart = TestControl(theControl,thePt);
        if(currentPart == part){
            if(currentPart == inPageUp)
                Scroll(theControl,-scrollAmount);
            if(currentPart == inPageDown)
                Scroll(theControl,scrollAmount);
        }
    }while(Button());
}
//////////////////////////////////////////////////////////////////////
//
// TScrollDoc::Scroll
//
//////////////////////////////////////////////////////////////////////
void TScrollDoc::Scroll(ControlHandle theControl,short change){

    // this routine changes the value of the scroll bar
    // and scrolls the contents,
    // it can be used for arbitrary scrolling,
    // either from scroll bar action procs
    // or while dragging mouse outside window

    // save current clip region
    RgnHandle oldClip = NewRgn();
    GetClip(oldClip);

    short diff = 0;
    short oldValue = GetCtlValue(theControl);
    short newValue = oldValue + change;

    // check for endpoint
    if (change < 0){
        short minValue = GetCtlMin(theControl);
        if(newValue < minValue)
```

```
            newValue = minValue;
    } else {
        short maxValue = GetCtlMax(theControl);
        // figure the new value and check for endpoint
        if(newValue > maxValue)
            newValue = maxValue;
    }
    diff = oldValue - newValue;

    // do the scrolling and set the new scroll bar values
    FocusOnContent();
    if(theControl == fHorizScrollBar)
        ScrollContents(diff,0);
    if(theControl == fVertScrollBar)
        ScrollContents(0,diff);

    FocusOnWindow();
    SetScrollBarValues();

    // restore old clip region
    SetClip(oldClip);
    DisposeRgn(oldClip);

}
//////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::DoThumbScroll
//
//////////////////////////////////////////////////////////////////////////
void TScrollDoc::DoThumbScroll(ControlHandle theControl,Point localPt){

    short oldValue = GetCtlValue(theControl);
    short trackResult = TrackControl(theControl,localPt,nil);
    if(trackResult != 0){
        short newValue = GetCtlValue(theControl);
        short diff = oldValue - newValue;
        FocusOnContent();
        if(theControl == fHorizScrollBar)
            ScrollContents(diff,0);
        if(theControl == fVertScrollBar)
            ScrollContents(0,diff);
        FocusOnWindow();
    }
}
//////////////////////////////////////////////////////////////////////////
//
// ActionProc
//
//////////////////////////////////////////////////////////////////////////
pascal void ActionProc(ControlHandle theControl,short partCode){
```

```
    // use static member function to get static member
    TScrollDoc * theCurrScrollDoc = TScrollDoc::GetCurrScrollDoc();

    short scrollAmount = 0;
    if(theControl == theCurrScrollDoc->GetVScroll())
        scrollAmount = theCurrScrollDoc->GetVertLineScrollAmount();
    if(theControl == theCurrScrollDoc->GetHScroll())
        scrollAmount = theCurrScrollDoc->GetHorizLineScrollAmount();

    if(partCode == inUpButton)
        theCurrScrollDoc->Scroll(theControl,-scrollAmount);
    if(partCode == inDownButton)
        theCurrScrollDoc->Scroll(theControl,scrollAmount);
}
//////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::ScrollContents
//
//////////////////////////////////////////////////////////////////////////
void TScrollDoc::ScrollContents(short dh,short dv){

    // determine the area to scroll
    Rect r;
    GetContentRect(r);

    // now scroll the image
    RgnHandle updateRgn = NewRgn();
    ScrollRect(&r,dh,dv,updateRgn);

    // keep track of how far off the origin we are
    fVOffset -= dv;
    fHOffset -= dh;

    // tell window to redraw uncovered content
    InvalRgn(updateRgn);

    // now force the update area to be drawn
    DoTheUpdate((EventRecord *)nil);

    // dispose of the region
    DisposeRgn(updateRgn);

}
//////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::Activate
//
//////////////////////////////////////////////////////////////////////////
void TScrollDoc::Activate(void){
    FocusOnWindow();
```

```
    if(fVertScrollBar != nil)
        ShowControl(fVertScrollBar);
    if(fHorizScrollBar != nil)
        ShowControl(fHorizScrollBar);

    // set up static member so that scroll action proc can access
    // member functions
    fCurrScrollDoc = this;

}
//////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::Deactivate
//
//////////////////////////////////////////////////////////////////////////
void TScrollDoc::Deactivate(void){
    FocusOnWindow();
    if(fVertScrollBar != nil)
        HideControl(fVertScrollBar);
    if(fHorizScrollBar != nil)
        HideControl(fHorizScrollBar);

}
//////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::DoTheUpdate
//
//////////////////////////////////////////////////////////////////////////
void TScrollDoc::DoTheUpdate(EventRecord * /*theEvent*/){

    if(fDocWindow != nil){
        FocusOnContent();
        BeginUpdate(fDocWindow);
        Rect r = (**(fDocWindow->visRgn)).rgnBBox;
        Draw(&r);
        FocusOnWindow();
        DrawControls(fDocWindow);
        DoDrawGrowIcon();
        EndUpdate(fDocWindow);
    }
}
//////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::DoContent
//
//////////////////////////////////////////////////////////////////////////
void TScrollDoc::DoContent(EventRecord* theEvent){

    FocusOnWindow();
    GlobalToLocal(&theEvent->where);
    Rect contents;
```

```
    GetContentRect(contents);
    if(PtInRect(theEvent->where,&contents)){
        FocusOnContent();
        ContentClick(theEvent);
    } else
        ScrollClick(theEvent);
}
////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::DoZoom
//
////////////////////////////////////////////////////////////////////////
void TScrollDoc::DoZoom(short partCode){

    FocusOnWindow();
    // call the parent class
    TDoc::DoZoom(partCode);

    SizeScrollBars();
    SynchScrollBars();

}
////////////////////////////////////////////////////////////////////////
//
// TScrollDoc::DoGrow
//
////////////////////////////////////////////////////////////////////////
void TScrollDoc::DoGrow(EventRecord* theEvent){

    FocusOnWindow();

    // call the parent class
    TDoc::DoGrow(theEvent);

    SizeScrollBars();
    SynchScrollBars();

}
```

```
/* TScrollDoc.rsrc.r
 *
 * rez source for TScrollDoc resources
 *
 * © 1990 Dan Weston All rights reserved
 *
 * Build it with the following rez command
 *
 *  rez TScrollDoc.rsrc.r -o TScrollDoc.rsrc -t rsrc -c RSED
 *
 */


#include "types.r"


resource 'CNTL' (129) {
    {0, 0, 0, 0},
    0,
    visible,
    0,
    0,
    scrollBarProc,
    0,
    ""
};

resource 'CNTL' (128) {
    {0, 0, 0, 0},
    0,
    visible,
    0,
    0,
    scrollBarProc,
    0,
    ""
};
```

```
//////////////////////////////////////////////////////////////////////////
//
// This file: PictView.cp   -   C++ Source
//
// This is the main application object for the PictView program
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////////

#include <Quickdraw.h>
#include <Windows.h>
#include <Memory.h>
#include <Files.h>
#include <Errors.h>
#include <Printing.h>

#include "TApp.h"
#include "TScrollDoc.h"

//////////////////////////////////////////////////////////////////////////
//
// class declarations
//
//////////////////////////////////////////////////////////////////////////

const long kPictHeaderSize = 512;

class TPICTDoc : public TScrollDoc{

protected:

    Handle      fPict;
    Handle      fHeader;
    THPrint     fPrintRecord;

public:

    TPICTDoc(OSType theCreator = '????',SFReply *reply = (SFReply *)nil);

    virtual ~TPICTDoc();

    virtual Boolean InitDoc(void);

protected:

    // routines you must override
    virtual short GetVertSize(void);
    virtual short GetHorizSize(void);
    virtual short GetVertLineScrollAmount(void){return 16;}
    virtual short GetHorizLineScrollAmount(void){return 16;}
```

```
    // draw the picture
    void Draw(Rect *r);

public:

    // This is the file type of the document
    virtual OSType GetDocType(){return 'PICT';}

    // this function reads in the file
    virtual Boolean ReadDocFile(short refNum);

    // disable the SaveAs menu
    virtual Boolean CanSaveAs(void){return false;}

    virtual void DoPageSetup(void);
    virtual void DoPrint(void);

    virtual Boolean CanPrint(void){return true;}
    virtual Boolean CanPageSetup(void){return true;}

};

class TPICTApp : public TApp{

protected:
    // make our kind of document
    virtual TDoc * MakeDoc(SFReply * reply = (SFReply *)nil);

    virtual Boolean CanOpen(void){return true;}

    // configure SFGetFile
    virtual int GetNumFileTypes(void){return 1;}
    virtual SFTypeList GetFileTypesList(void);

    // disable the New menu
    virtual Boolean CanNew(void){return false;}

};

////////////////////////////////////////////////////////////////////////
//
// Globals
//
////////////////////////////////////////////////////////////////////////

// list of file types for SFGetFile
SFTypeList gtheTypes = {'PICT'};

////////////////////////////////////////////////////////////////////////
//
```

```
// main
//
//////////////////////////////////////////////////////////////////////////
void main(void)
{
    TPICTApp theApp;

    // initialize the application
    if(theApp.InitApp()){

        // allow the user to open a PICT file first thing
        if(! theApp.OpenDocFromFinder())
            theApp.OpenOldDoc();

        // Start our main event loop running.
        // This won't return until user quits
        theApp.EventLoop();

        //now clean up
        theApp.CleanUp();
    }
}

//////////////////////////////////////////////////////////////////////////
//
// TPICTApp::MakeDoc
//
//////////////////////////////////////////////////////////////////////////
TDoc * TPICTApp::MakeDoc(SFReply * reply){

    return new TPICTDoc(GetCreator(),reply);

}
//////////////////////////////////////////////////////////////////////////
//
// TPICTApp::GetFileTypesList
//
//////////////////////////////////////////////////////////////////////////
SFTypeList TPICTApp::GetFileTypesList(void){
    return gtheTypes;
}


//////////////////////////////////////////////////////////////////////////
//
// TPICTDoc::TPICTDoc
//
//////////////////////////////////////////////////////////////////////////
TPICTDoc::TPICTDoc(OSType theCreator,SFReply *reply ):
    TScrollDoc(theCreator,reply) {
```

```
    fPict = nil;
    fHeader = nil;
    fPrintRecord = nil;
}

////////////////////////////////////////////////////////////////////////
//
// TPICTDoc::~TPICTDoc
//
////////////////////////////////////////////////////////////////////////
TPICTDoc::~TPICTDoc(){

    if(fHeader != nil){
        DisposHandle(fHeader);
        fHeader = nil;
    }
    if(fPict != nil){
        KillPicture((PicHandle)fPict);
        fPict = nil;
    }
    if(fPrintRecord != nil){
        DisposHandle((Handle)fPrintRecord);
        fPrintRecord = nil;
    }
}
////////////////////////////////////////////////////////////////////////
//
// TPICTDoc::InitDoc
//
////////////////////////////////////////////////////////////////////////
Boolean TPICTDoc::InitDoc(void){

    if(TScrollDoc::InitDoc()){
        fPrintRecord = (THPrint)NewHandle(sizeof(TPrint));
        if(fPrintRecord != nil){
            PrOpen();
            PrintDefault(fPrintRecord);
            PrClose();
            return true;
        }
    }

    // or if something went wrong
    return false;
}
////////////////////////////////////////////////////////////////////////
//
// TPICTDoc::Draw
//
////////////////////////////////////////////////////////////////////////
void TPICTDoc::Draw(Rect *r){
```

```
    if(fPict != nil){
        EraseRect(r);
        DrawPicture((PicHandle)fPict,
                    &((**((PicHandle)fPict)).picFrame));
    }
}

////////////////////////////////////////////////////////////////////////
//
// TPICTDoc::ReadDocFile
//
////////////////////////////////////////////////////////////////////////
Boolean TPICTDoc::ReadDocFile(short refNum){

    if(fDocWindow){

        long pictLength;
        long headerLength = kPictHeaderSize;

        OSErr err = GetEOF(refNum,&pictLength);
        pictLength -= kPictHeaderSize;
        Handle thePic = NewHandle(pictLength);
        if(thePic == nil){
            ErrorAlert(rDocErrorStrings,sNoMem);
            return false;
        }

        Handle theHeader = NewHandle(headerLength);
        if(theHeader == nil){
            ErrorAlert(rDocErrorStrings,sNoMem);
            DisposHandle(thePic);
            return false;
        }
        HLock(theHeader);
        HLock(thePic);
        err = SetFPos(refNum,fsFromStart,0);
        err = FSRead(refNum,&headerLength,(Ptr)*theHeader);
        err = FSRead(refNum,&pictLength,(Ptr)*thePic);
        HUnlock(thePic);
        HUnlock(theHeader);

        if(err == noErr){
            fPict = thePic;
            fHeader = theHeader;
            AdjustScrollBars();
            return true;
        } else {
            DisposHandle(thePic);
            DisposHandle(theHeader);
            return false;
```

```
        }
    }
    // if there ain't no window...
    return false;
}


////////////////////////////////////////////////////////////////////////
//
// TPICTDoc::GetVertSize
//
////////////////////////////////////////////////////////////////////////
short TPICTDoc::GetVertSize(void){

    Rect    r ;
    if(fPict){
        r = (**((PicHandle)fPict)).picFrame;
        return r.bottom - r.top;
    }else
        return 0;

}
////////////////////////////////////////////////////////////////////////
//
// TPICTDoc::GetHorizSize
//
////////////////////////////////////////////////////////////////////////
short TPICTDoc::GetHorizSize(void){

    Rect    r ;
    if(fPict){
        r = (**((PicHandle)fPict)).picFrame;
        return r.right - r.left;
    }else
        return 0;


}

////////////////////////////////////////////////////////////////////////
//
// TPICTDoc::DoPageSetup
//
////////////////////////////////////////////////////////////////////////
void TPICTDoc::DoPageSetup(void){

    // open the print manager
    PrOpen();

    // put up the style dialog
    (void)PrStlDialog(fPrintRecord);
```

```
    // and close the print manager
    PrClose();

}
//////////////////////////////////////////////////////////////////////////
//
// TPICTDoc::DoPrint
//
//////////////////////////////////////////////////////////////////////////
void TPICTDoc::DoPrint(void){

    TPPrPort    printPort;

    // open the print manager
    PrOpen();

    // if print record doesn't match printer,
    // put up the style dialog
    if(PrValidate(fPrintRecord))
        // if user cancels style dialog, cancel all printing
        if( ! PrStlDialog(fPrintRecord)){
            PrClose();
            return;
        }

    // Always put up the job dialog,
    // check to see if user cancels
    if(! PrJobDialog(fPrintRecord)){
        PrClose();
        return;
    }

    // now open the printing port
    printPort = PrOpenDoc(fPrintRecord,nil,nil);

    // open a page
    PrOpenPage(printPort,nil);

    // draw the image
    // use an empty rect to avoid unnecessary EraseRect
    Rect r;
    SetRect(&r,0,0,0,0);
    Draw(&r);

    // close the page
    PrClosePage(printPort);

    // close the printing port
    PrCloseDoc(printPort);
```

```
    // call PrPicFile for spooled printing (imagewriter)
    if((**fPrintRecord).prJob.bJDocLoop != 0){
        TPrStatus status;
        PrPicFile(fPrintRecord,nil,nil,nil,&status);
    }

    // close the print manager
    PrClose();
}
```

```
#-------------------------------------------------------------------
# Make file for a simple program using TScrollDoc
# To use it, use the MPW "Build..." command from the build menu,
# specifying "PictView" and the target file

#© 1990 Dan Weston, All rights reserved

# tell cplus and rez where to find included files for TApp,TDoc,
# and TScrollDoc
AppObjectDir = ::App-Doc:
ScrollObjDir = ::TScrollDoc:

# use SADE symbol generation, -sym off will result in faster builds
SymOpts = -sym on

# options for C++, where to look for include files
CPlusOptions = {SymOpts} -i "{AppObjectDir}"∂
                         -i "{ScrollObjDir}"

# options for the linker

LinkOptions = -msg nodup {SymOpts}

# options for rez, where to look for include and #include files
RezOptions = -s "{AppObjectDir}" -s "{ScrollObjDir}"∂
             -i "{AppObjectDir}" -i "{ScrollObjDir}"

# We need to change this rule to include CPlusOptions
.cp.o  ƒ  .cp
    CPlus {default}.cp -o {default}.cp.o {CPlusOptions}

Objects = ∂
    "{AppObjectDir}"TApp.cp.o ∂
    "{AppObjectDir}"TDoc.cp.o ∂
    "{ScrollObjDir}"TScrollDoc.cp.o ∂
    PictView.cp.o

ResourceFiles = ∂
    "{AppObjectDir}"TApp.rsrc ∂
    "{AppObjectDir}"TDoc.rsrc ∂
    "{ScrollObjDir}"TScrollDoc.rsrc ∂
    PictView.rsrc

# dependency rules for TDoc and TApp
"{AppObjectDir}"TDoc.cp.o ƒ "{AppObjectDir}"TDoc.cp ∂
                            "{AppObjectDir}"TDoc.h ∂
                            "{AppObjectDir}"AppDocMenus.h

"{AppObjectDir}"TApp.cp.o ƒ "{AppObjectDir}"TApp.cp ∂
                            "{AppObjectDir}"TApp.h ∂
                            "{AppObjectDir}"TDoc.h ∂
```

```
                              "{AppObjectDir}"AppDocMenus.h

# dependency rules for TScrollDoc
"{ScrollObjDir}"TScrollDoc.cp.o ƒ ∂
                "{ScrollObjDir}"TScrollDoc.cp ∂
                "{ScrollObjDir}"TScrollDoc.h ∂
                "{AppObjectDir}"TDoc.h

# dependency rules for PictView
PictView.cp.o  ƒ PictView.cp ∂
            "{AppObjectDir}"TApp.h ∂
            "{AppObjectDir}"TDoc.h ∂
            "{ScrollObjDir}"TScrollDoc.h ∂
            PictView.make

PictView ƒƒ {Objects} PictView.make
    Link -o {Targ} {LinkOptions} ∂
        {Objects} ∂
        "{CLibraries}"CPlusLib.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Interface.o
    SetFile {Targ} -t APPL -c '????' -a B

PictView ƒƒ PictView.r ∂
        {ResourceFiles}
    Rez -append -o {Targ} {RezOptions} PictView.r
```

```
/* PictView.r  rez source for the PictView application */

include "TApp.rsrc" ;
include "TDoc.rsrc";
include "TScrollDoc.rsrc";
include "PictView.rsrc" ;
```

```
/* PictView.rsrc.r
 *
 * rez source for PictView resources
 *
 * © 1990 Dan Weston All rights reserved
 *
 * Build it with the following rez command
 *
 *  rez PictView.rsrc.r -o PictView.rsrc -t rsrc -c RSED
 *
 */


#include "types.r"


resource 'MENU' (128, preload) {
    128,
    textMenuProc,
    0x7FFFFFFD,
    enabled,
    apple,
    {   /* array: 2 elements */
        /* [1] */
        "About PictView...", noIcon, noKey, noMark, plain,
        /* [2] */
        "-", noIcon, noKey, noMark, plain
    }
};

resource 'DITL' (128, purgeable) {
    {   /* array DITLarray: 5 elements */
        /* [1] */
        {76, 195, 96, 275},
        Button {
            enabled,
            "OK"
        },
        /* [2] */
        {9, 10, 48, 292},
        StaticText {
            disabled,
            "PictView: a simple application to view "
            "PICT files."
        },
        /* [3] */
        {55, 199, 70, 245},
        StaticText {
            disabled,
            ""
        },
```

```
//////////////////////////////////////////////////////////////////////////
//
//  This is the generic text edit object
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////////

// Mac Includes
#include <Types.h>
#include <Windows.h>
#include <OSUtils.h>
#include <Files.h>
#include <Errors.h>
#include <Memory.h>
#include <SysEqu.h>
#include <ToolUtils.h>

#include "TTEDoc.h"

// define the segment for the TEDoc classs
#pragma segment TEDocSeg

//////////////////////////////////////////////////////////////////////////
//
// constants
//
//////////////////////////////////////////////////////////////////////////

const short kTEMargin = 4;
const short kMaxShort = 32767;

//////////////////////////////////////////////////////////////////////////
//
// TTEDoc::TTEDoc
//
//////////////////////////////////////////////////////////////////////////
TTEDoc::TTEDoc(OSType theCreator,SFReply * SFInfo):
    TScrollDoc(theCreator,SFInfo){

    fTEHandle = nil;

}

//////////////////////////////////////////////////////////////////////////
//
// TTEDoc::InitDoc
//
//////////////////////////////////////////////////////////////////////////
Boolean TTEDoc::InitDoc(void){
```

```
    Rect view,dest;
    if(TScrollDoc::InitDoc()){
        SetPort(fDocWindow);
        view =  dest = fDocWindow->portRect;
        dest.left += kTEMargin;
        dest.top += kTEMargin;
        dest.right = kMaxShort;
        dest.bottom = kMaxShort;
        fTEHandle = TENew(&dest,&view);
        SetTERect();

        TEAutoView(true,fTEHandle);

        // install the click loop procedure
        SetClikLoop(MyClickLoop,fTEHandle);
    }
    return (fTEHandle != nil);
}

////////////////////////////////////////////////////////////////////////
//
// TTEDoc::~TTEDoc
//
////////////////////////////////////////////////////////////////////////
TTEDoc::~TTEDoc(void){

    if(fTEHandle != nil){
        TEDispose(fTEHandle);
        fTEHandle = nil;
    }
}

////////////////////////////////////////////////////////////////////////
//
// TTEDoc::ScrollContents
//
////////////////////////////////////////////////////////////////////////
void TTEDoc::ScrollContents(short dh,short dv){

    if(fTEHandle != nil)
        TEScroll(dh,dv,fTEHandle);

}

////////////////////////////////////////////////////////////////////////
//
// TTEDoc::SetScrollBarValues
//
////////////////////////////////////////////////////////////////////////
void TTEDoc::SetScrollBarValues(void){
```

```
    Rect visible = (**fTEHandle).viewRect;
    Rect dest = (**fTEHandle).destRect;

    short vPos = visible.top - dest.top;
    short hPos = visible.left - dest.left;

    FocusOnWindow();
    SetCtlValue(fHorizScrollBar,hPos);
    SetCtlValue(fVertScrollBar,vPos);

}

//////////////////////////////////////////////////////////////////////
//
// TTEDoc::GetContentRect
//
//////////////////////////////////////////////////////////////////////
void TTEDoc::GetContentRect(Rect& r){

    // ask the base class how big the rect is
    TScrollDoc::GetContentRect(r);

    // and now take away the TE margins
    r.left += kTEMargin;
    r.top += kTEMargin;

}

//////////////////////////////////////////////////////////////////////
//
// TTEDoc::SetTERect
//
//////////////////////////////////////////////////////////////////////
void TTEDoc::SetTERect(void){

    if(fTEHandle != nil){
        // set up the view rect
        Rect r;
        GetContentRect(r);
        (**fTEHandle).viewRect = r;

    }
}
//////////////////////////////////////////////////////////////////////
//
// TTEDoc::GetVertSize
//
//////////////////////////////////////////////////////////////////////
short TTEDoc::GetVertSize(void){

    return ((**fTEHandle).nLines * (**fTEHandle).lineHeight) ;
```

```
}

//////////////////////////////////////////////////////////////////////
//
// TTEDoc::GetHorizSize
//
//////////////////////////////////////////////////////////////////////
short TTEDoc::GetHorizSize(void){

    return (**fTEHandle).destRect.right - (**fTEHandle).destRect.left;

}
//////////////////////////////////////////////////////////////////////
//
// TTEDoc::GetVertLineScrollAmount
//
//////////////////////////////////////////////////////////////////////
short TTEDoc::GetVertLineScrollAmount(void){

    if(fTEHandle != nil)
        return (**fTEHandle).lineHeight;
    else
        return 0;

}
//////////////////////////////////////////////////////////////////////
//
// TTEDoc::GetHorizLineScrollAmount
//
//////////////////////////////////////////////////////////////////////
short TTEDoc::GetHorizLineScrollAmount(void){

    if(fTEHandle != nil)
        return (**fTEHandle).lineHeight;
    else
        return 0;

}

//////////////////////////////////////////////////////////////////////
//
// TTEDoc::AddText
//
//////////////////////////////////////////////////////////////////////
void TTEDoc::AddText(Ptr text,long len){

    if(fTEHandle != nil){
        TEInsert(text,len,fTEHandle);
        fNeedtoSave = true;
        TESelView(fTEHandle);
```

```
        SynchScrollBars();
    }
}

//////////////////////////////////////////////////////////////////////////
//
// TTEDoc::HaveSelection
//
//////////////////////////////////////////////////////////////////////////
Boolean TTEDoc::HaveSelection(void){

    if(fTEHandle)
        return ((**fTEHandle).selStart != (**fTEHandle).selEnd);
    else
        return false;

}
//////////////////////////////////////////////////////////////////////////
//
// TTEDoc::DoCut
//
//////////////////////////////////////////////////////////////////////////
Boolean TTEDoc::DoCut(Handle *theData,OSType *theType){

    Boolean result;
    if (result = DoCopy(theData,theType))
        DoClear();

    return result;
}
//////////////////////////////////////////////////////////////////////////
//
// TTEDoc::DoCopy
//
//////////////////////////////////////////////////////////////////////////
Boolean TTEDoc::DoCopy(Handle *theData,OSType *theType){

    if(fTEHandle){
        // put data on TEScrap
        TECopy(fTEHandle);

        // set theType
        *theType = 'TEXT';

        // do this in case we fail
        *theData = nil;

        // copy the handle to the data
        Handle TEData = TEScrapHandle();
        OSErr err = HandToHand(&TEData);
        if(err != noErr)
```

```
            return false;
        *theData = TEData;

        return true;
    }
}
////////////////////////////////////////////////////////////////////////
//
// TTEDoc::DoPaste
//
////////////////////////////////////////////////////////////////////////
void TTEDoc::DoPaste(Handle theData,OSType theType){

    if((fTEHandle) && (theType == 'TEXT')){
        // put data in TEScrap
        long scrapLen = GetHandleSize(theData);
        TESetScrapLen(scrapLen);

        // set low memory TEScrap handle with our data handle
        Handle * TEScrapHandle = (Handle *) TEScrpHandle;
        *TEScrapHandle = theData;

        // now go ahead and paste
        TEPaste(fTEHandle);

        fNeedtoSave = true;
        SynchScrollBars();
    }
}

////////////////////////////////////////////////////////////////////////
//
// TTEDoc::DoClear
//
////////////////////////////////////////////////////////////////////////
void TTEDoc::DoClear(void){

    if(fTEHandle){
        TEDelete(fTEHandle);
        fNeedtoSave = true;
        SynchScrollBars();
    }
}
////////////////////////////////////////////////////////////////////////
//
// TTEDoc::DoSelectAll
//
////////////////////////////////////////////////////////////////////////
void TTEDoc::DoSelectAll(void){

    if(fTEHandle)
```

```
        TESetSelect(0,kMaxShort,fTEHandle);
}

//////////////////////////////////////////////////////////////////////
//
// TTEDoc::Activate
//
//////////////////////////////////////////////////////////////////////
void TTEDoc::Activate(void){

    TScrollDoc::Activate();
    if(fTEHandle)
        TEActivate(fTEHandle);

}
//////////////////////////////////////////////////////////////////////
//
// TTEDoc::Deactivate
//
//////////////////////////////////////////////////////////////////////
void TTEDoc::Deactivate(void){

    TScrollDoc::Deactivate();
    if(fTEHandle)
        TEDeactivate(fTEHandle);

}

//////////////////////////////////////////////////////////////////////
//
// TTEDoc::Draw
//
//////////////////////////////////////////////////////////////////////
void TTEDoc::Draw(Rect *r){

    if(fTEHandle){
        EraseRect(r);
        TEUpdate(r,fTEHandle);
    }
}
//////////////////////////////////////////////////////////////////////
//
// TTEDoc::ContentClick
//
//////////////////////////////////////////////////////////////////////
void TTEDoc::ContentClick(EventRecord *theEvent){

    Boolean shiftKeyDown = ((theEvent->modifiers & shiftKey) != 0);
    if(fTEHandle){
        // turn off auto scrolling, we do it ourselves for clicking
        TEAutoView(false,fTEHandle);
```

```
        TEClick(theEvent->where,shiftKeyDown,fTEHandle);
        TEAutoView(true,fTEHandle);
    }
}

////////////////////////////////////////////////////////////////////////
//
// TTEDoc::DoKeyDown
//
////////////////////////////////////////////////////////////////////////
void TTEDoc::DoKeyDown(EventRecord* theEvent){

    if(fTEHandle){
        TEKey(LoWrd(theEvent->message),fTEHandle);
        fNeedtoSave = true;

        // reset the scroll bars since the key press
        // may have caused the text to scroll or added
        // text
        FocusOnWindow();
        SynchScrollBars();
    }
}


////////////////////////////////////////////////////////////////////////
//
// TTEDoc::DoIdle
//
////////////////////////////////////////////////////////////////////////
void TTEDoc::DoIdle(void){

    TScrollDoc::DoIdle();
    if(fTEHandle){
        TEIdle(fTEHandle);
    }

    GrafPtr oldPort;
    GetPort(&oldPort);
    SetPort(fDocWindow);
    Point thePt;
    GetMouse(&thePt);
    AdjustCursor(thePt);
    SetPort(oldPort);
}
////////////////////////////////////////////////////////////////////////
//
// TTEDoc::AdjustCursor
//
////////////////////////////////////////////////////////////////////////
void TTEDoc::AdjustCursor(Point where){
```

```
    Rect r;
    // decide if it is in content or scroll bars
    GetContentRect(r);
    if(PtInRect(where,&r)){
        CursHandle IBeam = GetCursor(iBeamCursor);
        if(IBeam != nil){
            SetCursor(*IBeam);
        }
    }else
        // it must be in the scroll bars or grow box
        InitCursor();
}
////////////////////////////////////////////////////////////////////////
//
// TTEDoc::DoZoom
//
////////////////////////////////////////////////////////////////////////
void TTEDoc::DoZoom(short partCode){

    // call the parent class, this will adjust scroll bars
    TScrollDoc::DoZoom(partCode);

    // adjust the TE rectangle
    SetTERect();

}
////////////////////////////////////////////////////////////////////////
//
// TTEDoc::DoGrow
//
////////////////////////////////////////////////////////////////////////
void TTEDoc::DoGrow(EventRecord* theEvent){
    // call the parent class, this will adjust scroll bars
    TScrollDoc::DoGrow(theEvent);

    // adjust the TE rectangle
    SetTERect();

}
////////////////////////////////////////////////////////////////////////
//
// TTEDoc::ReadDocFile
//
////////////////////////////////////////////////////////////////////////
Boolean TTEDoc::ReadDocFile(short refNum){

    if((fDocWindow) && (fTEHandle != nil)){

        long len;
        OSErr err = GetEOF(refNum,&len);
```

```
        // truncate to TE limits
        if(len > kMaxShort){
            len = kMaxShort;
        }
        Handle thetext = NewHandle(len);
        if(thetext == nil){
            ErrorAlert(rDocErrorStrings,sNoMem);
            return false;
        }

        HLock(thetext);
        err = SetFPos(refNum,fsFromStart,0);
        err = FSRead(refNum,&len,(Ptr)*thetext);
        HUnlock(thetext);
        if(err == noErr){
            // add the text to the document
            HLock(thetext);
            TESetText(*thetext,len,fTEHandle);
            HUnlock(thetext);
            DisposHandle(thetext);

            // set the selection to the first char
            TESetSelect(0,0,fTEHandle);

            // adjust scroll bars to new text
            SynchScrollBars();

            return true;

        } else {
            DisposHandle(thetext);
            return false;
        }
    }
    // if there ain't no window...
    return false;
}

////////////////////////////////////////////////////////////////////////
//
// TTEDoc::WriteDocFile
//
////////////////////////////////////////////////////////////////////////
Boolean TTEDoc::WriteDocFile(short refNum){

    if((fDocWindow != nil) && (fTEHandle != nil)){
        long len = (long)(**fTEHandle).teLength;
        CharsHandle thetext = TEGetText(fTEHandle);
        HLock((Handle)thetext);
        OSErr err = SetFPos(refNum,fsFromStart,0);
```

```
        err = FSWrite(refNum,&len,(Ptr)*thetext);
        HUnlock((Handle)thetext);
        if(err == noErr)
            return true;
        else
            return false;
    }
    // if there ain't no window...
    return false;

}
////////////////////////////////////////////////////////////////////////
//
// MyClickLoop
//
////////////////////////////////////////////////////////////////////////
pascal Boolean MyClickLoop(void){
    Point where;
    Rect  view;

    // use static member function to get static member
    TScrollDoc * theCurrScrollDoc = TScrollDoc::GetCurrScrollDoc();

    theCurrScrollDoc->GetContentRect(view);

    GetMouse(&where);
    if( where.v > view.bottom){
        theCurrScrollDoc->Scroll(theCurrScrollDoc->GetVScroll(),
                theCurrScrollDoc->GetVertLineScrollAmount());
    }
    if(where.h > view.right){
        theCurrScrollDoc->Scroll(theCurrScrollDoc->GetHScroll(),
                theCurrScrollDoc->GetHorizLineScrollAmount());
    }
    if(where.v < view.top){
        theCurrScrollDoc->Scroll(theCurrScrollDoc->GetVScroll(),
                -(theCurrScrollDoc->GetVertLineScrollAmount()));
    }
    if(where.h < view.left){
        theCurrScrollDoc->Scroll(theCurrScrollDoc->GetHScroll(),
                -(theCurrScrollDoc->GetHorizLineScrollAmount()));
    }

    return true;
}
```

```
//////////////////////////////////////////////////////////////////////
//
//  This file: TEApp.cp
//
//  This is the main application object for the simplest
//  application program using TTEDoc
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////

#include "TApp.h"
#include "TDoc.h"
#include "TTEDoc.h"

//////////////////////////////////////////////////////////////////////
//
// class declarations
//
//////////////////////////////////////////////////////////////////////
class TTEApp : public TApp{

protected:

    virtual TDoc * MakeDoc(SFReply * reply = (SFReply *) nil);
    virtual int GetNumFileTypes(void){return 1;};
    virtual SFTypeList GetFileTypesList(void);
    virtual Boolean CanOpen(void){return true;}
    virtual OSType CanAcceptClipType(void){return 'TEXT';}

};

//////////////////////////////////////////////////////////////////////
//
// Globals
//
//////////////////////////////////////////////////////////////////////

SFTypeList gtheTypes = {'TEXT'};

//////////////////////////////////////////////////////////////////////
//
// main
//
//////////////////////////////////////////////////////////////////////
void main(void)
{
    // create an instance of TTEApp
    TTEApp theApp;

    // initialize the application
```

```
    if(theApp.InitApp()){

        // open one window to start with,
        // unless we got files from the Finder
        if(! theApp.OpenDocFromFinder())
            theApp.OpenNewDoc();

        // run the event loop until user quits
        theApp.EventLoop();

        //now clean up
        theApp.CleanUp();
    }
}

////////////////////////////////////////////////////////////////////////
//
// TTEApp::MakeDoc
//
////////////////////////////////////////////////////////////////////////
TDoc * TTEApp::MakeDoc(SFReply * reply){

    return new TTEDoc(GetCreator(),reply);

}
////////////////////////////////////////////////////////////////////////
//
// TTEApp::GetFileTypesList
//
////////////////////////////////////////////////////////////////////////
SFTypeList TTEApp::GetFileTypesList(void){
    return gtheTypes;
}
```

```
#--------------------------------------------------------------------
# Make file for a simple program using TTEDoc
# To use it, use the MPW "Build..." command from the build menu,
# specifying "TEApp" and the target file

#© 1990 Dan Weston, All rights reserved

# tell cplus and rez where to find included files for TApp,TDoc,
# TScrollDoc, and TTEDoc
AppObjectDir = ::App-Doc:
ScrollObjDir = ::TScrollDoc:
TEObjDir = ::TEDoc:

# use SADE symbol generation, -sym off will result in faster builds
SymOpts = -sym on

# options for C++, where to look for include files
CPlusOptions = {SymOpts} ∂
        -i "{AppObjectDir}"∂
        -i "{TEObjDir}"∂
        -i "{ScrollObjDir}"

# options for the linker
LinkOptions = -msg nodup {SymOpts}

# options for rez, where to look for include and #include files
RezOptions = -s "{AppObjectDir}" -s "{ScrollObjDir}"∂
            -i "{AppObjectDir}"  -i "{ScrollObjDir}"

# We need to change this rule to include CPlusOptions
.cp.o   ƒ   .cp
    CPlus {default}.cp -o {default}.cp.o {CPlusOptions}

Objects =   ∂
    "{AppObjectDir}"TApp.cp.o ∂
    "{AppObjectDir}"TDoc.cp.o ∂
    "{ScrollObjDir}"TScrollDoc.cp.o ∂
    "{TEObjDir}"TTEDoc.cp.o ∂
    TEApp.cp.o

ResourceFiles = ∂
    "{AppObjectDir}"TApp.rsrc ∂
    "{AppObjectDir}"TDoc.rsrc ∂
    "{ScrollObjDir}"TScrollDoc.rsrc ∂
    TEApp.rsrc

# dependency rules for TDoc and TApp
"{AppObjectDir}"TDoc.cp.o ƒ "{AppObjectDir}"TDoc.cp ∂
                            "{AppObjectDir}"TDoc.h ∂
                            "{AppObjectDir}"AppDocMenus.h
```

```
"{AppObjectDir}"TApp.cp.o ƒ "{AppObjectDir}"TApp.cp ∂
                           "{AppObjectDir}"TApp.h ∂
                           "{AppObjectDir}"TDoc.h ∂
                           "{AppObjectDir}"AppDocMenus.h

# dependency rules for TScrollDoc
"{ScrollObjDir}"TScrollDoc.cp.o ƒ ∂
                "{ScrollObjDir}"TScrollDoc.cp ∂
                "{ScrollObjDir}"TScrollDoc.h ∂
                "{AppObjectDir}"TDoc.h

# dependency rules for TTEDoc
"{TEObjDir}"TTEDoc.cp.o ƒ "{TEObjDir}"TTEDoc.cp ∂
                        "{TEObjDir}"TTEDoc.h ∂
                        "{ScrollObjDir}"TScrollDoc.h ∂
                        "{AppObjectDir}"TDoc.h

# dependency rules for TEApp
TEApp.cp.o  ƒ TEApp.cp ∂
            "{AppObjectDir}"TApp.h ∂
            "{AppObjectDir}"TDoc.h ∂
            "{ScrollObjDir}"TScrollDoc.h ∂
            "{TEObjDir}"TTEDoc.h ∂
            TEApp.make

TEApp ƒƒ {Objects} TEApp.make
    Link -o {Targ} {LinkOptions} ∂
        {Objects} ∂
        "{CLibraries}"CPlusLib.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Interface.o
    SetFile {Targ} -t APPL -c '????' -a B

TEApp ƒƒ TEApp.r ∂
        {ResourceFiles}
    Rez -append -o {Targ} {RezOptions} TEApp.r
```

```
// TEApp.r  rez source for the simplest program
// that uses TTEDoc

include "TApp.rsrc";
include "TDoc.rsrc" ;
include "TScrollDoc.rsrc";
include "TEApp.rsrc" ;
```

```
/* TEApp.rsrc.r
 *
 * rez source for TEApp resources
 *
 * © 1990 Dan Weston All rights reserved
 *
 * Build it with the following rez command
 *
 *  rez TEApp.rsrc.r -o TEApp.rsrc -t rsrc -c RSED
 *
 */

resource 'MENU' (128, preload) {
    128,
    textMenuProc,
    0x7FFFFFFD,
    enabled,
    apple,
    {   /* array: 2 elements */
        /* [1] */
        "About TEApp...", noIcon, noKey, noMark, plain,
        /* [2] */
        "-", noIcon, noKey, noMark, plain
    }
};

resource 'ALRT' (128, purgeable) {
    {68, 76, 172, 376},
    128,
    {   /* array: 4 elements */
        /* [1] */
        OK, visible, silent,
        /* [2] */
        OK, visible, silent,
        /* [3] */
        OK, visible, silent,
        /* [4] */
        OK, visible, silent
    }
};

resource 'DITL' (128, purgeable) {
    {   /* array DITLarray: 5 elements */
        /* [1] */
        {65, 199, 85, 279},
        Button {
            enabled,
            "OK"
        },
        /* [2] */
        {8, 8, 24, 304},
```

```
        StaticText {
            disabled,
            "TEApp: a very simple text edit applicati"
            "on."
        },
        /* [3] */
        {44, 199, 59, 245},
        StaticText {
            disabled,
            ""
        },
        /* [4] */
        {42, 34, 58, 162},
        StaticText {
            disabled,
            "version 1.0"
        },
        /* [5] */
        {65, 34, 89, 165},
        StaticText {
            disabled,
            "April 15, 1990"
        }
    }
};
```

```
directory 'hd:mpw:C++:TEApp:'

sourcepath '::App-Doc:',∂
           '::TEDoc:',∂
           '::TScrollDoc:',∂
           '::TEApp:'

target 'TEApp'

open source ('TEApp.cp')
```

```
////////////////////////////////////////////////////////////////////////
//
// TDebugDoc.h
//
// A document/window that draws stream text in the window
//
// © 1990 Dan Weston, All Rights Reserved
//
////////////////////////////////////////////////////////////////////////

#include <QuickDraw.h>
#include <Events.h>
#include <Windows.h>
#include <iostream.h>

#include "TTEDoc.h"

#ifndef TDebugDoc_Defs
#define TDebugDoc_Defs


const short  rDebugDoc  = 5555;
const int    kBufferSize =  80;

////////////////////////////////////////////////////////////////////////
//
// class TWindowStreamBuff
//
////////////////////////////////////////////////////////////////////////
class TWindowStreamBuff: public streambuf{

public:
    TTEDoc *    fTEDoc;

    int overflow(int c = EOF);

    TWindowStreamBuff(char *p, int len);
};

////////////////////////////////////////////////////////////////////////
//
// class TDebugDoc
//
////////////////////////////////////////////////////////////////////////
class TDebugDoc : public TTEDoc,public ostream{

protected:

    TWindowStreamBuff * fBuff;

public:
```

```
    TDebugDoc::TDebugDoc(TWindowStreamBuff *buff,
                         OSType theCreator = '????',
                         SFReply * SFInfo = (SFReply *)nil);

    virtual ~TDebugDoc(void);

    virtual short GetWinID(void){return rDebugDoc;}

    // do this so Close menu isn't active
    // when debug window is on top
    virtual Boolean CanClose(void) { return false; };

};

// utility routine to make a DebugDoc and add it to doc list
TDebugDoc * MakeDebugDoc(TApp * theApp);


#endif TDebugDoc_Defs
```

```
//////////////////////////////////////////////////////////////////////////
//
// TDebugDoc.cp
//
// A document/window that draws stream text in the window
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////////

#include <Types.h>
#include <QuickDraw.h>
#include <Events.h>
#include <Windows.h>
#include <OSUtils.h>
#include <Strings.h>

#include "TApp.h"
#include "TDebugDoc.h"

// define a segment for DebugDoc
#pragma segment DebugDocSeg

//////////////////////////////////////////////////////////////////////////
//
// TDebugDoc::TDebugDoc
//
//////////////////////////////////////////////////////////////////////////
TDebugDoc::TDebugDoc(TWindowStreamBuff *buff,
                     OSType theCreator,SFReply * SFInfo):
    TTEDoc(theCreator,SFInfo),ostream(buff){

    // save a reference to the streambuffer
    // so we can disable it when the window closes
    fBuff = buff;


}
//////////////////////////////////////////////////////////////////////////
//
// TDebugDoc::~TDebugDoc
//
//////////////////////////////////////////////////////////////////////////
TDebugDoc::~TDebugDoc(void){

    // disable the streambuff so it won't
    // try to output to a deleted document
    fBuff->fTEDoc = nil;

}
```

```
//////////////////////////////////////////////////////////////////////
//
// MakeDebugDoc: A utility routine to make a debugging document
//
//////////////////////////////////////////////////////////////////////
TDebugDoc * MakeDebugDoc(TApp * theApp){

    // grab some memory for the stream buffer
    char * theBuffer = new char[kBufferSize];

    if(!theBuffer)
        return nil;

    // create the streambuffer
    TWindowStreamBuff *buff = new TWindowStreamBuff(theBuffer,
                                                    kBufferSize);

    // and pass it to the new DebugDoc's constructor
    TDebugDoc * temp = new TDebugDoc(buff);

    if(! temp)
        return nil;

    // make the window
    if( temp->MakeWindow(theApp->fenvRec.hasColorQD) &&
            temp->InitDoc()){
        temp->ShowDocWindow();
        theApp->AddDocument(temp);

        // connect the streambuff to the TEDocument
        buff->fTEDoc = temp;
        return temp;
    } else
        return nil;

}

//////////////////////////////////////////////////////////////////////
//
// TWindowStreamBuff::TWindowStreamBuff
//
//////////////////////////////////////////////////////////////////////
TWindowStreamBuff::TWindowStreamBuff(char *p, int len):
    streambuf(p,len){

    fTEDoc = nil;
}
//////////////////////////////////////////////////////////////////////
//
// TWindowStreamBuff::overflow
//
```

```
//////////////////////////////////////////////////////////////////////////
int TWindowStreamBuff::overflow(int c){

    if(fTEDoc){
        // how many chars?
        long len = pptr() - base();

        // add them to the text document
        fTEDoc->AddText((Ptr)base(),len);

        // reset everything
        setp(base(),epptr());
        if (c != EOF)
            sputc(c);

        return 0;
    }
    return EOF;
}
```

```
/* TDebugDoc.rsrc.r
 *
 * rez source for TDebugDoc resources
 *
 * © 1990 Dan Weston All rights reserved
 *
 * Build it with the following rez command
 *
 *  rez TDebugDoc.rsrc.r -o TDebugDoc.rsrc -t rsrc -c RSED
 *
 */


#include "types.r"

resource 'WIND' (5555) {
    {192, 20, 268, 280},
    documentProc,
    visible,
    noGoAway,
    0x0,
    "Debugging..."
};
```

```
////////////////////////////////////////////////////////////////////////
//
//
//  This file: DebugTEApp.cp
//
//  This is the main application object for the simplest
//  application program using TTEDoc and TDebugDoc
//
// © 1990 Dan Weston, All Rights Reserved
//
////////////////////////////////////////////////////////////////////////

#include "TApp.h"
#include "TDoc.h"
#include "TTEDoc.h"
#include "TDebugDoc.h"

////////////////////////////////////////////////////////////////////////
//
// class declarations
//
////////////////////////////////////////////////////////////////////////

class TTEApp : public TApp{

protected:

    virtual TDoc * MakeDoc(SFReply * reply = (SFReply *) nil);
    virtual int GetNumFileTypes(void){return 1;};
    virtual SFTypeList GetFileTypesList(void);
    virtual Boolean CanOpen(void){return true;}
    virtual OSType CanAcceptClipType(void){return 'TEXT';}
};

////////////////////////////////////////////////////////////////////////
//
// Globals
//
////////////////////////////////////////////////////////////////////////

SFTypeList gtheTypes = {'TEXT'};

TDebugDoc *gdebugDoc = nil;

////////////////////////////////////////////////////////////////////////
//
// main
//
////////////////////////////////////////////////////////////////////////
void main(void)
{
```

```
    // create an instance of TTEApp
    TTEApp theApp;

    // initialize the application
    if(theApp.InitApp()){

        gdebugDoc = MakeDebugDoc(&theApp);

        // open one window to start with,
        // unless we got files from the Finder
        if(! theApp.OpenDocFromFinder())
            theApp.OpenNewDoc();

        // run the event loop until user quits
        theApp.EventLoop();

        //now clean up
        theApp.CleanUp();
    }
}

////////////////////////////////////////////////////////////////////////
//
// TTEApp::MakeDoc
//
////////////////////////////////////////////////////////////////////////
TDoc * TTEApp::MakeDoc(SFReply * reply){

    TTEDoc * temp = new TTEDoc(GetCreator(),reply);
    *gdebugDoc << "Making a new document, address = "
               << (int)temp << endl;
    return temp;

}
////////////////////////////////////////////////////////////////////////
//
// TTEApp::GetFileTypesList
//
////////////////////////////////////////////////////////////////////////
SFTypeList TTEApp::GetFileTypesList(void){
    return gtheTypes;
}
```

DebugTEApp.make **Page 1**

```
#-------------------------------------------------------------------------
# Make file for a simple program using TDebugDoc
# To use it, use the MPW "Build..." command from the build menu,
# specifying "DebugTEApp" and the target file
#© 1990 Dan Weston, All rights reserved

# tell cplus and rez where to find included files for TApp,TDoc
# TScrollDoc, TTEDoc, and TDebugDoc
AppObjectDir = ::App-Doc:
ScrollObjDir = ::TScrollDoc:
TEObjDir = ::TEDoc:
DebugObjDir = ::DebugDoc:

# use SADE symbol generation, -sym off will result in faster builds
SymOpts = -sym on

# options for C++, where to look for include files
CPlusOptions = {SymOpts} -i "{AppObjectDir}"∂
                         -i "{TEObjDir}"∂
                         -i "{ScrollObjDir}"∂
                         -i "{DebugObjDir}"

# options for the linker
LinkOptions = -msg nodup {SymOpts}

# options for rez, where to look for include and #include files
RezOptions = -s "{AppObjectDir}" ∂
             -s "{ScrollObjDir}"∂
             -s "{DebugObjDir}" ∂
             -i "{AppObjectDir}" ∂
             -i "{ScrollObjDir}" ∂
             -i "{DebugObjDir}"

# We need to change this rule to include CPlusOptions
.cp.o   ƒ   .cp
    CPlus {default}.cp -o {default}.cp.o {CPlusOptions}

Objects =   ∂
    "{AppObjectDir}"TApp.cp.o ∂
    "{AppObjectDir}"TDoc.cp.o ∂
    "{ScrollObjDir}"TScrollDoc.cp.o ∂
    "{TEObjDir}"TTEDoc.cp.o ∂
    "{DebugObjDir}"TDebugDoc.cp.o ∂
    DebugTEApp.cp.o

ResourceFiles = ∂
    "{AppObjectDir}"TApp.rsrc ∂
    "{AppObjectDir}"TDoc.rsrc ∂
    "{ScrollObjDir}"TScrollDoc.cp.o ∂
    "{DebugObjDir}"TDebugDoc.rsrc
```

```
# dependency rules for TDoc and TApp
"{AppObjectDir}"TDoc.cp.o ƒ "{AppObjectDir}"TDoc.cp ∂
                           "{AppObjectDir}"TDoc.h ∂
                           "{AppObjectDir}"AppDocMenus.h

"{AppObjectDir}"TApp.cp.o ƒ "{AppObjectDir}"TApp.cp ∂
                           "{AppObjectDir}"TApp.h ∂
                           "{AppObjectDir}"TDoc.h ∂
                           "{AppObjectDir}"AppDocMenus.h

# dependency rules for TScrollDoc
"{ScrollObjDir}"TScrollDoc.cp.o ƒ ∂
                "{ScrollObjDir}"TScrollDoc.cp ∂
                "{ScrollObjDir}"TScrollDoc.h ∂
                "{AppObjectDir}"TDoc.h

# dependency rules for TEDoc
"{TEObjDir}"TTEDoc.cp.o ƒ "{TEObjDir}"TTEDoc.cp ∂
                        "{TEObjDir}"TTEDoc.h ∂
                        "{ScrollObjDir}"TScrollDoc.h ∂
                        "{AppObjectDir}"TDoc.h

# dependency rules for TDebugDoc
"{DebugObjDir}"TDebugDoc.cp.o ƒ ∂
                           "{DebugObjDir}"TDebugDoc.cp ∂
                           "{DebugObjDir}"TDebugDoc.h ∂
                           "{TEObjDir}"TTEDoc.h ∂
                           "{ScrollObjDir}"TScrollDoc.h ∂
                           "{AppObjectDir}"TDoc.h

# dependency rules for DebugTEApp
DebugTEApp.cp.o  ƒ DebugTEApp.cp ∂
            "{AppObjectDir}"TApp.h ∂
            "{AppObjectDir}"TDoc.h ∂
            "{TEObjDir}"TTEDoc.h ∂
            DebugTEApp.make

DebugTEApp ƒƒ {Objects} DebugTEApp.make
    Link -o {Targ} {LinkOptions} ∂
        {Objects} ∂
        "{CLibraries}"CPlusLib.o ∂
        "{CLibraries}"CRuntime.o ∂
        "{CLibraries}"StdCLib.o ∂
        "{CLibraries}"CInterface.o ∂
        "{Libraries}"Interface.o
    SetFile {Targ} -t APPL -c '????' -a B

DebugTEApp ƒƒ DebugTEApp.r ∂
        {ResourceFiles}
    Rez -append -o {Targ} {RezOptions} DebugTEApp.r
```

```
// TEApp.r  rez source for the simple
// that uses TTEDoc

include "TApp.rsrc";
include "TDoc.rsrc" ;
include "TScrollDoc.rsrc" ;
include "TDebugDoc.rsrc";
```

```
directory 'hd:mpw:C++:DebugTEApp:'

sourcepath '::App/Doc:',∂
           '::TEDoc:',∂
           '::DebugDoc:', ∂
           '::DebugTEApp:'

target 'DebugTEApp'

open source ('DebugTEApp.cp')
```

```
//////////////////////////////////////////////////////////////////////
//
// UMAPICTView.h
// This is the interface of the application objects for the
// MAPictView program
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////

class TPICTViewApp: public TApplication {
    public:

    // Initialize the Application
    virtual pascal void IPICTViewApp(void);

    // Launches a TPICTDocument
    virtual pascal struct TDocument *DoMakeDocument(CmdNumber
                                                    itsCmdNumber);

    // disable the new menu item
    virtual pascal void DoSetupMenus(void);

    // Prevents empty document on launch
    virtual pascal void OpenNew(CmdNumber itsCmdNumber);

};

class TPICTDocument : public TDocument {

    public:
        Handle          fPICTData;  // The PICT owned by the document
        Handle          fPICTHeader; // header for PICT file

        // Initialization and freeing
        virtual pascal void IPICTDocument(void);
        virtual pascal void Free(void);

        // disable Save and SaveAs menu items
        virtual pascal void DoSetupMenus(void);

        // read the file
        virtual pascal void DoRead(short aRefNum,Boolean rsrcExists,
                    Boolean forPrinting);

        // Making views and windows
        virtual pascal void DoMakeViews(Boolean forPrinting);

        // Inspecting
        virtual pascal void Fields(pascal void (*DoToField)(
                                StringPtr fieldName,
```

```
                                Ptr fieldAddr,
                                short fieldType,
                                void *DoToField_StaticLink),
                                void *DoToField_StaticLink);
};

class   TPICTView : public TView {

    public:

        // drawing and sizing
        virtual pascal void CalcMinSize(VPoint *minSize);

        virtual pascal void Draw(Rect *area);


};
```

```
//////////////////////////////////////////////////////////////////////////
//
// UMAPICTView.cp
// This is the implementation of the application objects for the
// MAPictView program
//
// The interface for these objects is in UMAPICTView.h
//
// © 1990 Dan Weston, All Rights Reserved
//
//////////////////////////////////////////////////////////////////////////

#include    <UMacApp.h>
#include    <UPrinting.h>

#include    <ToolUtils.h>

#include    <UMAPictView.h>

//////////////////////////////////////////////////////////////////////////
//
// constants
//
//////////////////////////////////////////////////////////////////////////

// application signature
const   OSType  kSignature = 'DANW';

// file-type code for saved disk files
const   OSType  kFileType = 'PICT';

//'view' template for a PICTView window
const   short   kWindowRsrcID = 1004;

// how much to stagger doc windows
const short kStaggerAmount = 16;

// ALRT for file-too-big
const short     kFileTooBig = 1000;

// size of PICT file header
const long      kPictHeaderSize = 512;

//////////////////////////////////////////////////////////////////////////
//
// globals
//
//////////////////////////////////////////////////////////////////////////

short   gStaggerCount;
```

```
#pragma segment AInit
//////////////////////////////////////////////////////////////////////
//
// TPICTViewApp::IPICTViewApp
//
//////////////////////////////////////////////////////////////////////
pascal void TPICTViewApp::IPICTViewApp(void){

    IApplication(kFileType);
    gStaggerCount = 0;


    // So the linker doesn't dead strip class info.
    // (gDeadStripSuppression is never true, so the
    // code never actually executes at run time.)
    if( gDeadStripSuppression) {
        TPICTView *aPICTView;
        aPICTView = new TPICTView;
    }

}

#pragma segment AOpen
//////////////////////////////////////////////////////////////////////
//
// TPICTViewApplication::DoMakeDocument
//
//////////////////////////////////////////////////////////////////////
pascal struct TDocument *TPICTViewApp::DoMakeDocument(CmdNumber
                                                      /*itsCmdNumber*/){

    TPICTDocument   *aPICTDocument;

    aPICTDocument = new TPICTDocument;
    FailNIL(aPICTDocument);
    aPICTDocument->IPICTDocument();
    return aPICTDocument;
}


#pragma segment AOpen
//////////////////////////////////////////////////////////////////////
//
// TPICTViewApplication::OpenNew
//
//////////////////////////////////////////////////////////////////////
pascal void TPICTViewApp::OpenNew( CmdNumber /*itsCmdNumber*/){

    AppFile anAppFile;

    if (ChooseDocument( cFinderOpen, &anAppFile ))
```

```
        OpenOld( cFinderOpen, &anAppFile );

}

#pragma segment ARes
//////////////////////////////////////////////////////////////////////////
//
// TPICTViewApp::DoSetupMenus
//
//////////////////////////////////////////////////////////////////////////
pascal void TPICTViewApp::DoSetupMenus(void){

    inherited::DoSetupMenus();

    Enable(cNew,false);

}


#pragma segment AOpen
//////////////////////////////////////////////////////////////////////////
//
// TPICTDocument::IPICTDocument
//
//////////////////////////////////////////////////////////////////////////
pascal void TPICTDocument::IPICTDocument(void){

    // do the inherited stuff
    IDocument(kFileType,
            kSignature,
            kUsesDataFork,
            ! kUsesRsrcFork,
            kDataOpen,
            ! kRsrcOpen);

    // and now do our specific members
    fPICTData = nil;
    fPICTHeader = nil;
}


#pragma segment AClose
//////////////////////////////////////////////////////////////////////////
//
// TPICTDocument::Free
//
//////////////////////////////////////////////////////////////////////////
pascal void TPICTDocument::Free(void){

    if(fPICTData != nil){
        DisposHandle(fPICTData);
```

```
        fPICTData = nil;
    }
    if(fPICTHeader != nil){
        DisposHandle(fPICTHeader);
        fPICTHeader = nil;
    }
    inherited::Free();
}

#pragma segment AOpen
//////////////////////////////////////////////////////////////////////
//
// TPICTDocument::DoMakeViews
//
//////////////////////////////////////////////////////////////////////
pascal void TPICTDocument::DoMakeViews(Boolean /*forPrinting*/){

    TView       *theWindow,*thePictView;
    TStdPrintHandler    *aHandler;

    theWindow = NewTemplateWindow(kWindowRsrcID, this);

    FailNIL(theWindow);
    thePictView = theWindow->FindSubView('PicV');

    aHandler = new TStdPrintHandler;
    FailNIL(aHandler);
    aHandler->IStdPrintHandler(this,
                               thePictView,
                               ! kSquareDots,
                               kFixedSize,
                               ! kFixedSize);
    ShowReverted();
}

#pragma segment AReadFile
//////////////////////////////////////////////////////////////////////
//
// TPICTDocument::DoRead
//
//////////////////////////////////////////////////////////////////////
pascal void TPICTDocument::DoRead(short aRefNum,
                               Boolean /*rsrcExists*/,
                               Boolean /*forPrinting*/){

    long        pictSize;
    long        headerSize = kPictHeaderSize;

    // calculate size of file, subtract header size
    FailOSErr(GetEOF(aRefNum, &pictSize));
    pictSize = pictSize - kPictHeaderSize;
```

```
    // allocate memory for header and pict
    fPICTHeader = NewPermHandle(kPictHeaderSize);
    FailNIL(fPICTHeader);

    fPICTData = NewPermHandle(pictSize);
    FailNIL(fPICTData);

    // now read header and pict
    HLock(fPICTHeader);
    FailOSErr(FSRead(aRefNum, &headerSize, *fPICTHeader));
    HUnlock(fPICTHeader);

    HLock(fPICTData);
    FailOSErr(FSRead(aRefNum, &pictSize, *fPICTData));
    HUnlock(fPICTData);
}

#pragma segment ARes
//////////////////////////////////////////////////////////////////////////
//
// TPICTDocument::DoSetupMenus
//
//////////////////////////////////////////////////////////////////////////
pascal void TPICTDocument::DoSetupMenus(void){

    inherited::DoSetupMenus();

    Enable(cSaveAs, false);
    Enable(cSaveCopy, false);

}

#pragma segment AFields
//////////////////////////////////////////////////////////////////////////
//
// TPICTDocument::Fields
//
//////////////////////////////////////////////////////////////////////////
pascal void TPICTDocument::Fields(pascal void (*DoToField)(
                                  StringPtr fieldName,
                                  Ptr fieldAddr,
                                  short fieldType,
                                  void *DoToField_StaticLink),
                                  void *DoToField_StaticLink){

    DoToField("\pTPICTDocument", nil,
            bClass,DoToField_StaticLink);
    DoToField("\pfPICTData", (Ptr)&fPICTData,
            bHandle,DoToField_StaticLink);
    DoToField("\pfPICTHeader", (Ptr)&fPICTHeader,
```

```
            bHandle,DoToField_StaticLink);
    inherited::Fields(DoToField,DoToField_StaticLink);
}

#pragma segment ARes
//////////////////////////////////////////////////////////////////////
//
// TPICTView::CalcMinSize
//
//////////////////////////////////////////////////////////////////////
pascal void TPICTView::CalcMinSize(VPoint *minSize){

    short   hSize,vSize;
    TPICTDocument * PICTDoc = (TPICTDocument *)fDocument;

    if(PICTDoc->fPICTData) {
            hSize = (**((PicHandle)PICTDoc->fPICTData)).picFrame.right
                - (**((PicHandle)PICTDoc->fPICTData)).picFrame.left;
            vSize = (**((PicHandle)PICTDoc->fPICTData)).picFrame.bottom
                - (**((PicHandle)PICTDoc->fPICTData)).picFrame.top;

            SetVPt(minSize, hSize,vSize);

    } else
        SetVPt(minSize,0,0);
}

#pragma segment ARes
//////////////////////////////////////////////////////////////////////
//
// TPICTView::Draw
//
//////////////////////////////////////////////////////////////////////
pascal void TPICTView::Draw(Rect * /*area*/){

    TPICTDocument * PICTDoc = (TPICTDocument *)fDocument;

    if(PICTDoc->fPICTData)
        DrawPicture((PicHandle)(PICTDoc->fPICTData),
                    &(**((PicHandle)PICTDoc->fPICTData)).picFrame);
}
```

```
////////////////////////////////////////////////////////////////////////
//
// MMAPICTView.cp
// This is the main program for MAPICTView
//
// © 1990 Dan Weston, All Rights Reserved
//
////////////////////////////////////////////////////////////////////////



#include <UMacApp.h>
#include <UPrinting.h>

#include <UMAPictView.h>
////////////////////////////////////////////////////////////////////////
//
// globals
//
////////////////////////////////////////////////////////////////////////

TPICTViewApp    *gPICTViewApp;

#pragma segment Main
////////////////////////////////////////////////////////////////////////
//
// main
//
////////////////////////////////////////////////////////////////////////
void main(void){

    InitToolBox();  // Essential toolbox and utilities initialization

    if(ValidateConfiguration(&gConfiguration)){ // Make sure we can run

        InitUMacApp(8);      //Initialize MacApp; 8 calls to MoreMasters
        InitUPrinting();     // Initialize the Printing unit

        gPICTViewApp = new TPICTViewApp;
        FailNIL(gPICTViewApp);
        gPICTViewApp->IPICTViewApp();
        gPICTViewApp->Run();
    } else
        StdAlert(phUnsupportedConfiguration);
}
```

```
/* • Auto-Include the requirements for this source */
#ifndef __TYPES.R__
#include "Types.r"
#endif

#ifndef __MacAppTypes__
#include "MacAppTypes.r"
#endif

#if qTemplateViews
#ifndef __ViewTypes__
#include "ViewTypes.r"
#endif
#endif

#if qDebug
include "Debug.rsrc";
#endif
include "MacApp.rsrc";
include "Printing.rsrc";

include $$Shell("ObjApp")"MAPictView" 'CODE';


/* Resource ids */

/* The 'File is too large' alert */
#define kFileTooBig     1000
/* resource ID of window */
#define kWindowRsrcID   1004


resource 'view' (kWindowRsrcID, purgeable) {
    {
    root, 'WIND', { 50, 40 }, { 250, 450 }, sizeVariable,
                            sizeVariable, shown, enabled,
    Window {
        "",
            zoomDocProc, goAwayBox, resizable, modeless,
            ignoreFirstClick, freeOnClosing,
            disposeOnFree, closesDocument,
            openWithDocument, dontAdaptToScreen, stagger,
            forceOnScreen, dontCenter, noID, "" };

    'WIND', 'SCLR', { 0, 0 }, { 250-kSBarSizeMinus1,
                                450-kSBarSizeMinus1 },
    sizeRelSuperView, sizeRelSuperView, shown, enabled,
    Scroller {
        "",
            vertScrollBar, horzScrollBar, 0, 0, 16, 16,
```

```
            vertConstrain, horzConstrain, { 0, 0, 0, 0 } };

    'SCLR', 'PicV', { 0, 0 }, { 116, 1020 },
            sizeVariable, sizeVariable, shown, enabled,
    View {"TPICTView"}
    }
};



resource 'SIZE' (-1) {
    saveScreen,
    acceptSuspendResumeEvents,
    enableOptionSwitch,
    canBackground,
    MultiFinderAware,
    backgroundAndForeground,
    dontGetFrontClicks,
    ignoreChildDiedEvents,
    is32BitCompatible,
    reserved,
    reserved,
    reserved,
    reserved,
    reserved,
    reserved,
    reserved,
#if qdebug
    1024 * 1024,
    760 * 1024
#else
    1024 * 1024,
    760 * 1024
#endif
};

/*
    Printing to the LaserWriter is the time when the most
    temporary memory is in use.  We need the segments in
    use at that time
*/

resource 'seg!' (256, purgeable) {
    {
        "GWriteFile";
        "GClipboard";
        "GNonRes";
        "GFile";
        "GSelCommand";
        "GTerminate";
        "GClose";
```

```
        "GDoCommand";
    }
};

resource 'DITL' (phAboutApp, purgeable) {
    {
/* [ 1] */  {160, 182, 180, 262},
            Button {
                enabled,
                "OK"
            };
/* [ 2] */  {10, 75, 150, 320},
            StaticText {
                disabled,
                    "This sample program views \nPICT files."
                    "\n\nThis program was written "
                    "with MacApp® © 1990 Dan Weston,"
                    "\n© 1985-1989 Apple Computer, Inc."
            };
/* [ 3] */  {10, 20, 42, 52},
            Icon {
                disabled,
                1
            }
    }
};

// Grab the default about box
include "Defaults.rsrc"  'ALRT' (phAboutApp);

/*
    Used when the user attempts to
    read a file larger than we can handle
*/

resource 'DITL' (kFileTooBig, purgeable) {
    {
/* [ 1] */  {82, 198, 100, 272},
                Button {
                    enabled,
                    "OK"
                };
/* [ 2] */  {10, 70, 77, 272},
                StaticText {
                disabled,
                "PICT View can't read the entire "
                "file because it is too long."
                };
/* [ 3] */  {10, 20, 42, 52},
                Icon {
                    disabled,
```

```
                    0
                }
    }
};

resource 'ALRT' (kFileTooBig, purgeable) {
    {100, 110, 210, 402},
    kFileTooBig,
    {
/* [ 1] */  OK, visible, silent;
/* [ 2] */  OK, visible, silent;
/* [ 3] */  OK, visible, silent;
/* [ 4] */  OK, visible, silent
    }
};

// Grab the default Apple/File menus
include "Defaults.rsrc" 'cmnu' (mApple);
include "Defaults.rsrc" 'cmnu' (mFile);
include "Defaults.rsrc" 'cmnu' (mEdit);

/* Displayed menus on a non-hierarchical system */
resource 'MBAR' (kMBarDisplayed) {
    {mApple; mFile; mEdit}
};


// Grab the default credits
include "Defaults.rsrc"  'STR#' (kDefaultCredits);

// Get the default MacApp® application icon
// and necessary bundling rsrcs
include "Defaults.rsrc"  'MApp' (0);
include "Defaults.rsrc"  'FREF' (128);
include "Defaults.rsrc"  'BNDL' (128);
include "Defaults.rsrc"  'ICN#' (128);

// Get the default Version resources
// Application or file specific
include "Defaults.rsrc"  'vers' (1);

// Overall package
include "Defaults.rsrc"  'vers' (2);
```

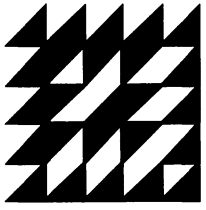
Index

C++ Source Code Disk Available

Complete source code for all programs in *Elements of C++ Macintosh Programming* on an 800K Macintosh floppy. Requires MPW C++ 3.1b1, or later.

Leave the typing to us — Save time and worry

Just $35.00, postage and handling included.
Mastercard and Visa accepted.
Send in the coupon below or phone in your order today.



Nerdworks
3410 SW Water Ave.
Portland, OR 97201
(503) 274-9577

Name _____

Company _____

Address _____

City, State, Zip _____

Date _____

___$35.00 check enclosed ___Visa ___Mastercard

card #_____ exp date _____

signature _____

Macintosh Inside Out

> $22.95 FPT USA
$29.95 CANADA

Elements of C++ Macintosh® Programming

DAN WESTON

The C++ programming language is recognized as the future of Macintosh® programming, and it is rapidly becoming the standard programming language for Apple® Computer, Inc. Every Macintosh programmer, both beginning and advanced, needs to begin exploring the power of C++.

Elements of C++ Macintosh Programming teaches Macintosh programmers just what they need to know to take the step forward from programming with C to programming with C++. It is also the perfect guide for programmers already programming with C++. This book teaches the basic elements of C++ programming, concentrating on the object-oriented programming style and syntax. Through numerous hands-on examples, both beginning and more experienced programmers will learn how to design practical and effective programs with C++, including the newest version, Release 2.

You will also learn how to:

- Create document and application classes that simplify programming
- Derive document classes that can scroll pictures and text
- Use streams for easily displaying debugging information
- Create powerful hybrid classes using multiple inheritance
- Use C++ Release 2 with MacApp® version 2.

In addition, the appendices include the complete original source code used in the examples in the book. This hands-on approach to C++ makes **Elements of C++ Macintosh Programming** an invaluable guide for all Macintosh programmers.

Dan Weston is an expert on C++ Macintosh programming. He is a Macintosh software developer and programming instructor and is a member of Apple's Certified Developers Group. He is also the author of *The Complete Book of Macintosh Assembly Language Programming.*

Author photograph by Eric Edwards
Cover design by Ronn Campisi
Addison-Wesley Publishing Company, Inc.

ISBN 0-201-55025-3
55025