# develop

The Apple Technical Journal




**Issue 2** April 1990

**EDITORIAL**

Editor in Chief's Clothing *Louella Pizzuti*
Developmental Editors *Lorraine Anderson and Carol Westberg*
Editorial Assistant *Susan Marsland*
Editorial Assistant *Lenore Zelony*
Manager, Developer Press *David Krathwohl*

**ART & PRODUCTION**

Design/Art Direction *Joss Parsey*
Technical Illustration *Cleo Huggins*
Production *Bruce Potterton*
Printer *Craftsman Press*
Film Preparation *FilmCraft*
Photographer *Ralph Portillo*
Circulation Management *Dee Kiamy*
Online Production *Cassi Carpenter*

**SPECIAL THANKS**

Mark Kieling

**TECHNICAL REVIEWERS**

Tim Enwall
Craig Prouse
Jon Zap
Gregg Williams
Jim Straus
Andy Shebanow
Larry Rosenstein
Lew Rollins
Rob Neville
P. Nagarajan
Robin Myers
Bob Martin
Don Loomis
John Harvey
Jerry Godes
Dennis Gibbs
Matt Deatherage
Richard Clark
Mary Chan
Bob Campbell
Jeremy Bornstein
Rick Blair
Brian Bechtel
Pete Alexander


**COVER**

Illustrator *Cleo Huggins*

In spite of 4 years at R.I.S.D. and 10 years of experience (including an MS indigital typography from Stanford), Cleo still creates. Of course, she's got the perfect environment at home: a parrot that talks and a boyfriend that doesn't. If she runs into any ancient Egyptions (and she might), she's ready with an 800-character hieroglyphic font she created for the Mac. Pretty cool. One thing, though: she laughs at Louella's jokes (sometimes she even thinks they're funny).

# Content

LOUELLA PIZZUTI

Dear Readers,

Wow! Your response to the first issue of develop was overwhelming! You sent almost two hundred letters and Links telling me what you liked, what you'd like changed, and what articles you'd like to see. I've published (and answered) a representative sampling in the Letters section and will continue to let you know what happens with your suggestions. Article-specific questions are answered (by the original authors) in the Macintosh Q & A section.

This issue of develop revolves around object programming. If you've been reading much of what's been coming out of Apple lately, you will have noticed quite a push toward object programming. Apple has invested years of R & D in object programming and we're working to get you to take advantage of the work we've done.

Unless you're already an object-based programmer, this issue probably won't solve any problems you're struggling with right now. I'm hoping, however, that it helps to convince you that object programming can save you time and effort in the long run and that it's worth the investment it will take to learn about a new environment. Not only will object programming help you to write and to maintain today's applications, but it will also help to prepare you for tomorrow's system software.

Providing you with code and ideas that serve you well when new products come out is one of my top priorities, so look past today and into tomorrow and use the code and ideas we give you.

Keep those cards and letters coming!

Louella Pizzuti
**Editor**

# LETTERS

## THE GOOD

Wow!
—Jim Russell

We just received the first issue of develop. It is just what we need. In fact, it couldn't come at a better time. Anyway, please continue to do it—we need it! Great idea, great layout, great topics, etc....
—Daniel Tapie

Thanks a lot for the complimentary first issue of develop. It makes me feel healthy with the funny introduction of the authors, wealthy with its luxury, and happy for your name is nicer than ever on the first page.
—Philippe

I believe you have a winner here! develop has helped me in just the first 10 minutes. I needed help with the Palette Manager and I found it here!

Might I suggest that the CD-ROM envelope be perforated? This would allow easy removal of the packet and alleviate the "now I have the CD-ROM out, but the packet is still in the way when I flip through the pages" syndrome.
—Bryan Carter

Fantastic idea!!! Keep up the good work. This first issue was FULL of good, timely, and useful advice (code). Thanks for your efforts to help developers.
—Ken Duncan

Excellent magazine! I really enjoyed the articles, source code examples, backgrounds on the authors, etc. I predict that develop will establish a new standard of excellence in technical support literature. But please find a better font for the source code listings; the font is so faint that I found myself suffering from eye strain after a short while.
—Ken Friedenbach

What a fantastic idea! The graphic design in develop is so good that it almost distracts from the content. Keep up the good work and please send the next issue (this one was stolen…).
—Jean-Michel Karr

## THE BAD

develop has NOTHING to offer an Apple II owner/developer. Apple no longer has my respect as the founder of the home computing and friendly interfaces. I do not want a Mac and the way I feel at this moment I would not have a Mac if you gave it to me. Yes, I am angry!
—R. L. Woodworth

The outstanding quality of the premier issue of develop was overshadowed by its content. It would have been appropriate in a premier issue to devote approximately equal space to BOTH Apple lines of computers. Apple II support in develop would surely help Apple and encourage those who acknowledge the IIs as very respectable computers. May the Apple II and Macintosh lines BOTH enjoy continued and increasing success in their respective markets!
—Steven Gozdziewski



**COMMENTS** We welcome timely letters to the editor, especially from readers wishing to react to articles that we publish in develop. Letters should be addressed to Louella Pizzuti; 20525 Mariani Ave. M/S 75-3B; Cupertino, CA 95014 (AppleLink Pizzuti1). All letters should include name and company name as well as address and phone number. Letters may be excerpted or edited for clarity and space.

**SUBSCRIPTION INFORMATION** Please address all subscription (and subscription-related) enquiries to:

develop
Apple Computer, Inc.
P.O. Box 531
Mt. Morris, IL 61054 USA
AppleLink: DEV.SUBS

*Like other commodities,* develop *is influenced by the laws of supply and demand. I publish articles based on what I have and on what I believe the development community needs. There is a definite need for Apple II information, and* develop *will continue to be a forum in which to meet that need, but the mix of Apple II and Macintosh articles will continue to reflect the mix of developers and their needs (and available articles), rather than an absolute 50-50 balance.*
*—Louella*

The situation on technical information has just gotten worse with develop. We now have *Inside Macintosh*, Technical Notes, the Q & A stack, develop, and many other documents available through APDA. All of these have information not found anywhere else. It is a total nightmare when I want to find all the information on a particular topic. I really, really, REALLY would like ALL Macintosh technical information in one regularly updated reference. Get rid of all the others.
—Tim Fredenburg

*Would you settle for having all of the pivotal information in one place (like a CD-ROM disc, for example), and for working toward an indexing scheme that would let you find out all of the documents (and pieces of sample code) that related to your topic of interest? We believe that the documents deserve (and need) to have lives of their own to address the needs of the folks who don't want to know absolutely everything about everything, but we also believe that an intelligent indexer would simplify things immensely. This issue's CD-ROM disc contains much of the information you'd like to see combined, and the next issue will have our first crack at the intelligent indexer. All comments and suggestions welcome.*
*—Louella*

I'm curious to find out WHY Apple decided to go with a CD-ROM disc. I understand that the disc holds gobs of data (which I would imagine goes mostly unused every issue); however, I wonder HOW MANY of the Mac and Apple II developers actually own a CD-ROM drive. Apple may be trying to encourage developers to utilize this technology, but at the moment it looks like they are just flapping their wings in the breeze. I've got a great idea! Everyone who can't use the CD-ROM disc should mail it back to Apple. A small, silent protest. :-)
—Lynda

*The CD-ROM gives us room to archive all of the old issues of* develop *(which allows us to update and to correct mistakes every quarter), to publish code that would never fit in our hundred-some pages, and to explore what can be done when space is not a problem. This issue of* develop*, the disc includes not only* develop *and all of its associated code, but also SpInside Macintosh (a HyperCard stack-based version of Inside Macintosh volumes I-V), the Macintosh Technical Notes stack, and the Macintosh DTS Q & A stack, and of course, the ever-popular audio track.*
*—Louella*

## THE UNCLASSIFIABLE

Egad! I received my developer's package this noontime and found develop included in the package. While I haven't had quite the time I wanted to examine the material, I did notice two serious technical errors. To whit:

1. Louella, you've got to be under forty! That pale gray print for the listings is—at least to my forty+ eyes, well nigh invisible. Please take pity on us old codgers and not blind us!

2. Catching penguins is far easier than you suggest. When I was stationed at McMurdo Sound a few decades ago, we simply recorded the sound of the local pod of Orcas and set up large speakers on three sides of a square. When we turned the recordings on, the poor pennies thought their mortal enemies were coming ashore and accordingly raced out of danger—right into our nets!

This boiled the bejabbers out of the biologists, but made our day! Ever see a penguin race?
—Robert Smith

*I'm afraid I'm guilty on both counts: I am under forty (although I hope the code in this issue is easier on your eyes), and I did not fully test all of the questions and answers. In the future, I'll try to convince management that testing penguin-catching techniques is worth the trip. I'm sure your letter will help.*
*—Louella*

I noticed on page 2 that two "Spirit Guides" were listed. While trying to figure out what these might be, I thought of three possibilities: continuity editors (insure that each article adheres to a common theme); channelers or spiritists (as in New Age, religion, occult); or testers of wines or vineyards. So, now that I've had my guesses, could you tell me what the real answer is and what relation they have to your magazine? Or is it all just a joke to see who REALLY reads your journal?
—Rex Bontrager

*Margery and Lou both contributed immensely to actually getting develop into print; without the two of them, it would probably still be a bunch of manuscript pages sitting in my office and I'd still be thinking that printing was the opposite of cursive. As for the heading, they both make me smile and so does calling them Spirit Guides. (And it is good to see who's actually reading the masthead.)*
*—Louella*

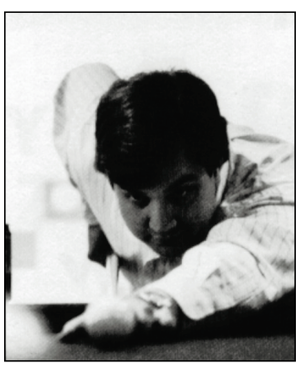
## QUESTIONS

I like develop. It's cool. But what's the deal with the code contained therein (on CD-ROM)? Can we use it? Can we distribute it? Both of those (at least the first) would seem to be the intent of develop. But the lawyer's funfest at the back would seem to say otherwise. I wondered about this before I saw the article in MacWeek, but now I'm really confused. Is use of code contained on the CD-ROM as limited as seems to be implied by the CD-ROM container's text? Or what?
—Robert

*You can freely use, copy, and distribute the code that's included in develop. Many thanks to Teri Drenker in Apple's software licensing group for this issue's revised licensing agreement.*
*—Louella*

# USING C++ OBJECTS IN A HANDLE-BASED WORLD

*Although C++ is a powerful and flexible language, its image of the world inside your computer was shaped by operating systems that were a bit more "traditional" than the Macintosh's. As a result, C++ routines that work fine for MPW tools can cause severe problems when used in a Macintosh application. But there are ways around these problems. This article describes a technique that allows you to use normal C++ objects in your Macintosh applications without undue discomfort.*



ANDY SHEBANOW

Using C++ objects in the handle-based world of the Macintosh Memory Manager can get pretty tricky at times. The Apple extension to C++ that solves the memory allocation problems you're bound to run into can create other headaches for you if you need to use one or more of several important C++ features in your program. In this article, you'll learn about the memory allocation problems you can expect to encounter when you create objects in C++. You'll also learn how to get around these problems while still retaining the use of important C++ features, by creating a special class **`PtrObject`**. You'll see a sample program that uses **`PtrObject`**, and you'll learn how to implement the class.

## PROBLEMS WITH MEMORY ALLOCATION FOR OBJECTS IN C++

In C++, objects are created dynamically with the **`new`** operator, and disposed of with the **`delete`** operator when you've finished with them, like this:

```
TMyObject* aMyObjectRef;
aMyObjectRef = new TMyObject;     // Create a TMyObject object.
aMyObject->AReallyCoolRoutine(); // Do something useful...
delete aMyObject;                 // Delete the object.
```



**ANDY SHEBANOW**, a DTS engineer, wrote this article for the best of reasons: "The beer people had their say in the last issue, and it's about time the Mountain Dew people spoke up." His highly developed personal skills have earned him the affectionate nickname "The Shebanator." After working for a medical imaging company, he joined Apple twenty-odd months ago. It's been so long since he was outside that he's forgotten what his hobbies are; he vaguely remembers something about driving cars at excessive and/or illegal speeds. •

When you use these operators, C++ transforms them into calls to **operator new** and **operator delete**. The default versions of **operator new** and **operator delete** provided in the C++ library use the C Standard Library routines **malloc** and **free**, respectively, to allocate and deallocate the memory needed to store the object. (Actually, the **calloc** routine is called to allocate the memory, but **calloc** just turns around and calls **malloc** to do the real work.)

These routines work fine for MPW tools, but they can cause the following severe problems when used in a Macintosh application:

- *Heap fragmentation.* Nonrelocatable memory for your objects can be allocated in the middle of your heap, preventing the Mac's Memory Manager from compacting memory properly.
- *Heap space permanently wasted.* Because calloc and free manage their own list of free memory blocks and never return unused space to the Mac's Memory Manager, if you create a lot of C++ objects your program can run out of memory and crash even though you have plenty of free memory available. (See the sidebar "Everything You Didn't Want to Know About **malloc** and **free**" for more information on malloc internals.)

Fortunately, you can override the default versions of **operator new** and **operator delete** in your own classes to get explicit control over memory allocation. To help you do this, Apple extended C++ to include a predefined base class called **HandleObject**. If classes you define inherit from **HandleObject**, the Mac's **NewHandle** and **DisposHandle** traps are called instead of the default **operator new** and **operator delete** routines.

While this solves the memory problems just mentioned, it also precludes the use of several important C++ features. Here is a partial list of the restrictions that apply when classes you define inherit from **HandleObject**:

- It is an error to declare global variables, local variables, arrays, members, or parameters of handle-based classes (rather than pointers to them).
- Multiple inheritance cannot be used with handle-based classes.
- Handle-based objects can be created only by the **new** operator. The only use of a dereferenced handle-based class pointer (for example, ***x**) is to refer to a field in the class (for example, ***x.y** or **x->y**).
- It is not possible to allocate an array of handle-based objects—for example, **new->MyObjects[10]**.

As you can see, there are quite a few useful things you can't do in C++ if you use **HandleObject**. Most programs should be able to live with these restrictions, but if your program needs to use multiple inheritance or arrays of objects, a different solution is called for.



Consult the MPW C++ *Reference*, available from APDA as part of the MPW C++ package, for a full list of restrictions on C++ features in effect when classes you define inherit from **HandleObject**.•

## EVERYTHING YOU DIDN'T WANT TO KNOW ABOUT MALLOC AND FREE

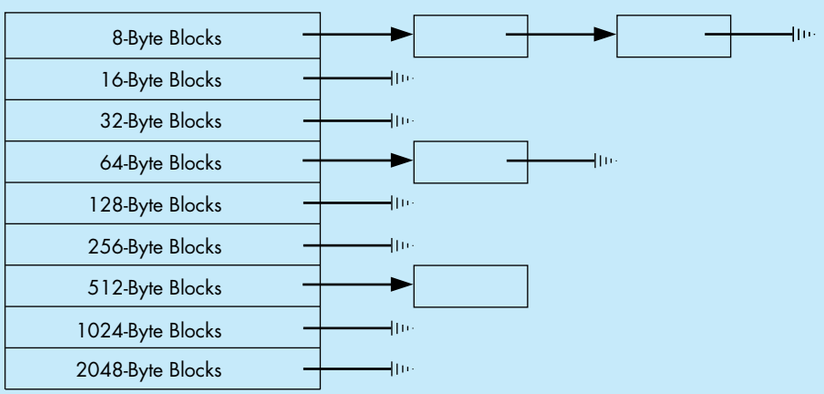
Why do the **malloc** and **free** routines wreak so much havoc in a Macintosh application? The main reason is that these routines were originally written for UNIX systems, which have no built-in memory allocation facilities. So these library routines ended up doing everything themselves, including free list management.

This isn't all bad, since these routines are simpler and faster than their Macintosh Memory Manager equivalents, but they can cause the severe problems listed earlier in this article for a Macintosh application. The worst part is that these problems can occur even if your application doesn't call **malloc** directly. In many situations, C++ calls **malloc** for you, as do many of the other routines in the standard library.

Here's how it all works (in MPW, at least):

When you request some memory from **malloc**, it rounds the size up to the nearest power of 2 (8-byte minimum, ID checked at the door). If you ask for more than 2048 bytes, **malloc** just calls **NewPtr** to allocate the memory, and **DisposPtr** to get rid of it. Otherwise, **malloc** checks its internal free list looking for blocks of the specified size. If it doesn't find any blocks of that size, it allocates a chunk of memory with **NewPtr** big enough to hold 2K worth of blocks (plus 2 bytes overhead per block), and adds the new blocks to the free list for that size. It then returns you the first block off of the free list.

When you dispose of memory with the **free** routine, it looks at the block header to determine which free list to put the block in, and inserts it into the list (sorted by block address to allow for more intelligent freestore management in the future). Here's what a small free list looks like:



MPW (and other Mac development systems) provides versions of these routines to make life easier for people who are porting code from UNIX systems (or MS-DOS, OS/2, etc.). However, since the **malloc** routine calls **NewPtr** rather indiscrimately, it can cause blocks to be allocated in very inconvenient places inside your heap, and once these blocks have been allocated, they are never disposed of.

In just one possible scenario, the user opens a large document with your program SuperOOPWrite and creates 1000 standard C++ objects (each allocated by **malloc**) to represent the elements of the document, each about 100 bytes long. **malloc** asks **NewPtr** to create 63 blocks of memory (about 2K each), and you have about 100K less free memory than you used to. Now the user closes the document, and you dutifully dispose of all of your objects. Guess what? You still have 100K less memory available to you as far as the Mac's Memory Manager is concerned, your heap is chock full of 2K nonrelocatable blocks, and you don't have any way to preflight memory for the next time the user wants to open a document.

By the way, if this algorithm sounds familiar to you, it's because the MPW code is based on a public domain version of **malloc** written by Chris Kingsley.

## THE SOLUTION: CREATING A SPECIAL CLASS PTROBJECT

The solution to memory allocation problems when you can't use **HandleObject** is to create a special class **PtrObject** analogous to the **HandleObject** class. This class overrides both **operator new and operator delete**, so that real Memory Manager pointers are used instead of the pointers returned by the default **operator new**. **PtrObject** also supports the allocation of objects into a separate heap, which further reduces memory fragmentation.

The method functions of the class **PtrObject** are as follows:

**AllocHeap** This function creates a separate heap. All descendants of class **PtrObject** created after calling this function will use this heap. If you do not call this function in your program, the default (application) heap will be used.

**DisposeHeap** This function disposes of the heap allocated by a previous call to **AllocHeap**. You should call this function before quitting your application. Any **PtrObject**s created inside the heap will be invalid, so make sure that you aren't using any of those objects anymore (neither **operator delete** nor the destructor for these objects will be called).

| | |
|---|---|
| **FreeMemory** | This function returns the amount of free space in the **PtrObject** heap, as returned by the trap **FreeMem**. If no separate heap exists, this function will return the amount of free memory in the default (application) heap. |
| **MaxMemory** | This function returns the size of the largest free block in the **PtrObject** heap, as returned by the trap **MaxMem**. If no separate heap exists, this function will return the amount of free memory in the default (application) heap. |
| **operator new** | This function is called by the C++ compiler to allocate memory for **PtrObject**s. You never need to call it directly. |
| **operator delete** | This function is called by the C++ compiler to deallocate memory used by **PtrObject**s. You never need to call it directly. |

Here is the class declaration for **PtrObject**, which would normally be found in the header file PtrObject.h.

```
class PtrObject {
public:
      static OSErr AllocHeap(size_t heapSize);
      // Create a heap heapSize bytes long to allocate
      // objects in.

      static void  DisposeHeap();
      // Free up the heap allocated by a previous call
      // to AllocHeap.

      static long  FreeMemory();
      // Return the total amount of free space in the heap.

      static Size  MaxMemory();
      // Return the size of the largest free block in the heap.

      void*        operator new(size_t size);
      void         operator delete(void* p);
      // These are our special allocation and
      // deallocation operators.

private:
      static THz   fZone;
      // Our private zone pointer.
};
```

Notice that the **AllocHeap**, **DisposeHeap**, **FreeMemory**, and **MaxMemory** calls are all *static member functions*, and that the **fZone** variable is a *static data member*. In C++, static members are shared across all instances of a class. You should use static members in place of global variables and functions whenever possible, since they have limited scope (which means fewer name conflicts) and they are logically tied to the class in which they are declared (which means more readable source code). To call a static member function, the syntax is

```
ClassName::StaticFunctionName(/* parameters, if any */);
```

## A SAMPLE PROGRAM USING PTROBJECT

Now that you've seen the interface to the **PtrObject** class, here is a small sample application that uses it. This program isn't very useful—all it does is define a subclass of **PtrObject**, create an instance of that object, and call one of its methods.

The first thing we have to do is the standard setup for a Macintosh application, which in this case means including all of the needed header files for the Macintosh Toolbox and the C Standard Library:

```
// TestPtrObject.cp
#include <Types.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <SegLoad.h>
#include <Events.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <Memory.h>
#include <OSUtils.h>
#include <stdio.h>
#include <string.h>
#include <stddef.h>
```

Next we include the header file for the **PtrObject** class (just shown), and define a new class **TLout** that is derived from it:

```
#include "PtrObject.h"

// A small class that contains some data and a constructor,
// but spends all of its time on street corners cadging
// cigarettes instead of doing useful work.
```

```
class TLout : public PtrObject {
public:
       TLout() { DoCadge(); };    // Our constructor.
       virtual void DoCadge();    // A rude member function.
private:
       char          fArray[256];
};

void TLout::DoCadge()
{
       strcpy(fArray,"Hey buddy, spare a cig?");
}
```

That's all it takes to define a class with the correct memory management behavior. Here is the main program, which uses our newly defined **TLout** class:

```
void InitToolbox();        // Forward declaration.

main()
{
       // We need this much space to store the objects
       // we're going to initialize - in this case, 16KBytes.
       const size_t kDefaultHeapSize = 0x4000;

       InitToolbox();      // Initialize Mac Toolbox (ho hum).

       // Create a heap for PtrObjects to live in.
       OSErr heapErr = PtrObject::AllocHeap(kDefaultHeapSize);

       // If we got an error, quit - this isn't a real
       // application, so we don't need error handling, right?
       if (heapErr != noErr)
          ExitToShell();

       // Create an object - will go in separate heap automatically.
       TLout* aLout = new TLout;  // Do that voodoo that TLouts do…
       if (aLout != nil)
        {
           aLout->DoCadge();
           delete aLout;          // Delete our object now that we have finished with it.
        }

       // Dispose of the heap.
       PtrObject::DisposeHeap();
       ExitToShell();
}
```

One important thing needs to be pointed out here: you need to call **PtrObject::AllocHeap** as early in your program as possible, or the newly created heap may fragment your application heap.

Finally, just for completeness, here's the implementation of the **InitToolbox** routine, which makes sure that all of the necessary pieces of the Mac Toolbox are initialized:

```
void InitToolbox()
{
	// Standard Macintosh initialization.
	InitGraf((Ptr) &qd.thePort);
	InitFonts();
	InitWindows();
	InitMenus();
	TEInit();
	InitDialogs(nil);
	InitCursor();
	MaxApplZone();
}
```

## THE SAMPLE PROGRAM'S MAKEFILE

```
# TestPtrObject.make
# by Andrew Shebanow (with some help from the CreateMake script)

OBJECTS = ∂
	PtrObject.cp.o ∂
	TestPtrObject.cp.o

SymOptions = -sym on
CPlusOptions = {SymOptions}

{OBJECTS} ƒƒ PtrObject.h

TestPtrObject ƒƒ {OBJECTS}
	Link -w {SymOptions} -mf {OBJECTS} ∂
		"{CLibraries}"CSANELib.o ∂
		"{CLibraries}"Math.o ∂
		"{CLibraries}"CPlusLib.o ∂
		"{CLibraries}"StdCLib.o ∂
		"{CLibraries}"CInterface.o ∂
		"{CLibraries}"CRuntime.o ∂
		"{Libraries}"Interface.o ∂
		-o TestPtrObject

PtrObject.cp.o ƒ PtrObject.cp PtrObject.h
TestPtrObject.cp.o ƒ TestPtrObject.cp PtrObject.h
```

## IMPLEMENTING PTROBJECT

Now that we've seen how to use class **PtrObject**, we need to implement it. We must first include all the necessary header files and allocate our static member data:

```
// PtrObject.cp
#include <Memory.h>
#include <Errors.h>
#include <stdio.h>
#include <stddef.h>
#include "PtrObject.h"

// Static data members actually need to be declared outside of the class
// definition in order to have space allocated.
THz PtrObject::fZone = nil;
```

Next we have the **AllocHeap** function.

```
OSErr PtrObject::AllocHeap(size_t heapSize)
{
       // By default, the heap gets kNumDfltMasters master pointers. A small number,
       // but it shouldn't matter, since we will only be allocating Ptrs in this heap,
       // and Ptrs don't use master pointers.
       const short kNumDfltMasters = 16;

       // This magic number from Inside Mac, vol. II, chapter 1, is the amount of space
       // required for the zone header and trailer, and the master pointer block. We add
       // this to the requested heap size to compensate.
       const size_t kZoneOverhead = 64 + 8 + (sizeof(long) * kNumDfltMasters);

       heapSize += kZoneOverhead;       // Factor in overhead.

       // Allocate space for the zone.
       Ptr zonePtr = NewPtr(heapSize);
       if (!zonePtr)              // if alloc fails, return error
           return MemError();     // Get a pointer to the end of the heap.

       Ptr limitPtr = (Ptr) (((ptrdiff_t) zonePtr) + heapSize);

       // Initialize the zone.
       InitZone(nil, kNumDfltMasters, limitPtr, zonePtr);

       // Save the zone pointer in our static class variable.
       fZone = (THz) zonePtr;
       return noErr;
}
```

The **DisposeHeap** member function is much simpler. It just checks to see if we allocated a zone in the past, and if so, it disposes of the heap's memory. This will destroy any objects that were allocated inside the heap, which could be dangerous, so be careful when you call this routine.

```
void PtrObject::DisposeHeap()
{
       // If zone actually exists, dispose of it.
       if (fZone)
        {
           DisposPtr((Ptr) fZone);
           fZone = nil;
        }
}
```

Next we have the **FreeMemory and MaxMemory** functions. We'll show them together, since they are almost identical. The only thing of note here is the way we switch in our special heap if it exists.

```
long PtrObject::FreeMemory()
{
       THz savedZone;
                                       // Before we can return the amount of free
                                       // memory, we need to switch to the correct zone.
       if (fZone)
        {
           savedZone = GetZone();      // Save current zone.
           SetZone(fZone);             // Make our zone current.
        }

       long free = FreeMem();          // Get total free space.

       if (fZone)
         SetZone(savedZone);           // Restore previous zone.
       return free;
}

Size PtrObject::MaxMemory()
{
       THz savedZone;
                                       // Before we can return the maximum block size,
                                       // we need to switch to the correct zone.
       if (fZone)
        {
           savedZone = GetZone();      // Save current zone.
           SetZone(fZone);             // Make our zone current.
        }

       Size tSize;                     // We know the heap can't grow,
                                       // but we have to have a temp
                                       // variable to satisfy the Toolbox.
       Size max = MaxMem(&tSize);      // Get size of biggest block.
       if (fZone)
         SetZone(savedZone);           // Restore previous zone.
       return max;
}
```

Now we get to the heart of the class, the **operator new** function. Like the **FreeMemory** call, **operator new** switches to our private heap before it actually allocates memory, and restores the previous heap when it is done. The actual memory allocation is done by a call to everyone's favorite Macintosh trap, **NewPtr**.

```
void* PtrObject::operator new(size_t size)
{
       THz savedZone;

       // before we can allocate memory, we need to switch to the correct zone
       if (fZone)
        {
           savedZone = GetZone();        // Save current zone.
           SetZone(fZone);               // Make our zone current.
        }
       Ptr p = NewPtr(size);             // Allocate memory for object.
       if (fZone)
         SetZone(savedZone);             // Restore previous zone.
       return p;
}
```

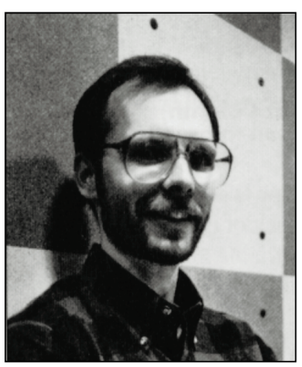
Last, we have the **operator delete** function. All it does is dispose of the memory occupied by the object. We don't need to swap in the private heap here, since the Memory Manager keeps track of the heap that the pointer belongs to for us.

```
void PtrObject::operator delete(void* p)
{
       DisposPtr((Ptr) p);               // This works regardless of the zone
                                         // the pointer was allocated in.
}
```

That's all there is to the **PtrObject** class. If you wish to explore the stranger side of C++ (multiple inheritance and so on), you should use it, since it allows your creations to live in the complicated world of the Macintosh Memory Manager.

# USING OBJECTS SAFELY IN OBJECT PASCAL

*In Object Pascal, objects are just like handles in that they refer to relocatable blocks of memory. To use objects safely, the programmer needs to recognize that the Macintosh Memory Manager can move the block of memory referred to by an object or handle, although only at well-defined times. This article gives guidelines for the safe use of objects in Object Pascal.*



CURT BIANCHI

The simplicity and elegance of Object Pascal's syntax is a two-edged sword. On the one hand, it makes Object Pascal feel like a natural extension to Pascal; on the other, it can lull a programmer into a false sense of security. For although the syntax of Object Pascal treats objects as though they were statically allocated, the fact is that in Object Pascal, objects are *always* allocated as relocatable blocks (handles, in the vernacular) in the application heap. Thus, when you write Object Pascal programs for the Macintosh, you must be eternally aware that objects are handles, and program accordingly. This article tells you how to do that with MPW Pascal and TML Pascal, two compilers that can be used with MacApp in the MPW environment. In addition, it gives some tips for using handles outside the context of objects.

## HOW OBJECT PASCAL IMPLEMENTS OBJECTS: A CAUTIONARY TALE

To get an idea of how Object Pascal implements objects, let's compare the code fragments in Figure 1. Each column of code accomplishes the same thing: the definition and use of a data structure representing a graphical shape. The only difference is that the left column is implemented with objects, while the right column is implemented with handles. The code in these two columns is very similar, and a comparison of the two reveals what goes on behind the scenes.



**CURT BIANCHI**, displaced Lakers fan, has never met a taco he didn't like. He's been at Apple more than three years, where he first worked on MacApp and now concentrates on future system software. This southern California native (he asks you not to hold that against him) earned a BSICS in 1981 from the University of California-Irvine, followed by stints at Link Systems and Monogram Software, and self-employment doing software odd jobs, including working on computer dating software. His hobbies include music, photographing trains, avoiding serious injury on the basketball court, and rooting for the Lakers from afar.

```
 1 TYPE                                        1 TYPE
 2   TShape = OBJECT (TObject)                 2   TShapeHdl = ^TShapePtr;
 3        fBounds:   Rect;                     3   TShapePtr = ^TShape;
 4        fColor:    RGBColor;                 4   TShape = RECORD
 5        END;                                 5        fBounds:   Rect;
 6                                             6        fColor:    RGBColor;
 7                                             7        END;
 8                                             8
 9 VAR                                         9 VAR
10   aShape:         TShape;                  10   aShape:         TShapeHdl;
11  sameShape, copiedShape: TShape;           11  sameShape, copiedShape: TShapeHdl;
12                                            12
13 BEGIN                                      13 BEGIN
14 NEW(aShape);                               14 aShape := TShapeHdl(NewHandle(SIZEOF(TShape)));
15 FailNIL(aShape);                           15 FailNIL(aShape);
16                                            16
17 aShape.fBounds := gZeroRect;               17 aShape^^.fBounds := gZeroRect;
18 aShape.fColor := gRGBBlack;                18 aShape^^.fColor := gRGBBlack;
19                                            19
20 sameShape := aShape;                       20 sameShape := aShape;
21                                            21
22 copiedShape := TShape(aShape.Clone);       22 copiedShape := aShape;
23                                            23 FailOSErr(HandToHand(Handle(copiedShape)));
24 FailNIL(copiedShape);                      24 FailNIL(copiedShape);
25                                            25
26 END;                                       26 END;
```

**Figure 1.**
**A Comparison of Code Implemented with Objects (Left Column) vs. Handles (Right Column)**

The first thing to observe is that any variable of an object type is actually a *reference* to an object. That is, the variable is a handle that refers to a block of memory containing the object's data. Thus, in the left column the value of the variable **aShape** is a handle. It contains the address of a master pointer that in turn points to the object's data. The size of the variable **aShape** is four bytes—the size of an address and not the size of the object itself. This is very much the same as the right column, in which the variable **aShape** is explicitly declared to be a handle. In fact, the only difference between the two is that the object version of **TShape** has an implicit field containing the object's class ID, located just before the first declared field. The class ID is an integer value that allows the object's type to be identified at run time.

Line 14 of each column shows how a **TShape** data structure is created. Since handles must be dynamically allocated in the heap, it follows that objects must be dynamically allocated as well. This is the purpose of the call to **NEW** in the left column. Note that **NEW** works completely differently for objects and for other kinds

of memory allocation. For objects, **NEW** generates a call to the internal library procedure **%_OBNEW**, which, aside from some debugging details, simply calls **NewHandle**, just like the handle-based code on the right does.

The call to **FailNIL** in line 15 detects the case where allocation of the object or handle fails. **FailNIL** is part of MacApp's failure-handling library and will be discussed in greater detail later.

Lines 17 and 18 reference fields of **aShape**. In the object code, the syntax leads you to believe that no handle dereferencing takes place, but of course we know better. What the Pascal compiler does is to implicitly dereference the handle for you. In other words, it does the very same thing as the code in the right column does explicitly.

Line 20 assigns one object *reference* to another, causing both **aShape** and **sameShape** to refer to the *same* object. Line 22 (plus 23 in the right column) produces *another* shape whose contents are exactly the same as **aShape**. In the object case, the **Clone** method is used to produce a copy of the object referenced by **aShape**; **copiedShape** is assigned a reference to the newly created object. **Clone** is implemented by calling the Toolbox routine **HandToHand**, as is used in the right column. (**FailOSErr** is a MacApp routine that checks the result of **HandToHand**.) Since copying an object (or a handle) requires a memory allocation for the new object, **FailNIL** is used to ensure that the copy succeeded.

The moral of this story is that you have to be very careful about how you use objects. For example, you must remember that every time you refer to a field of an object, you're really dereferencing a handle. If you're not careful, you're likely to wind up with a corrupt heap.

## A PRIMER ON HANDLES AND THEIR PITFALLS

Handles have some interesting properties. If you've done any serious programming on the Macintosh (and I don't mean HyperTalk), then you know what I mean. If not, then (1) you've been spared the sorrows of a corrupt heap, and (2) you ought to get *How to Write Macintosh Software*, 2nd ed., by Scott Knaster (Hayden Books, 1988). Chapters 2 and 3 tell you all you need to know about handles. In the meantime, I'll give you a thumbnail description.

In the heap, relocatable blocks of memory are referenced by double indirection, as shown in Figure 2. The first pointer (called the handle) points to a nonrelocatable pointer (called the master pointer), which in turn points to a block of memory. The Memory Manager can move the block of memory, and when this happens the address in the master pointer is changed to the block's new address.

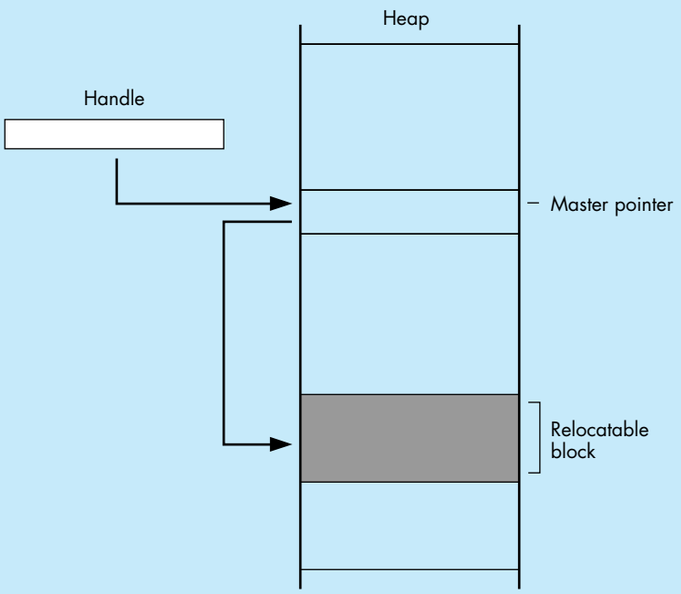



**Figure 2.**
**A Handle to a Relocatable Block**

This doesn't create a problem as long as you access the block via the handle. However, at times it's necessary or desirable for the sake of efficiency to dereference the handle—that is, make a copy of the block's master pointer, and then use that pointer to access the block by single indirection. And even this isn't a problem—as long as the block of memory doesn't move.

Well, we have bad news: it's bound to move at some point, when the Memory Manager needs to compact the heap. When this happens, the master pointer itself is correctly updated, but your copy of it is left dangling.

Now for the good news: relocatable blocks of memory only move at certain well-defined times. Thus, the key to dereferencing handles is knowing when the blocks of memory they point to may move.

Oh, and one more bit of bad news: the Memory Manager has no garbage collection. This means you're responsible for disposing of handles when you've finished with them, and making sure you don't leave any dangling pointers.

## PRACTICING SAFE OBJECT USAGE

Because the Memory Manager moves blocks of memory only at certain well-defined times, it's possible to come up with reliable guidelines for safe object usage. Keep these guidelines firmly in mind anytime you program in Object Pascal:

1. *Don't* pass fields of objects as parameters to any routine unless you know it's safe.

   In Pascal, when a routine is called, each parameter is passed by value or by address. Passing a parameter by value pushes a copy of the parameter's value onto the stack. Passing a parameter by address pushes the parameter's address onto the stack. (This should immediately trigger a handle alert in your head.) Passing the *value* of an object field is no problem. But passing the *address* of an object field on the stack is a potentially unsafe situation. That's because the address points to a memory location within an object—in other words, the object is dereferenced. If the object should happen to get relocated, the address won't point into the object anymore.

   Because there's no way to predict what the address points to after memory relocation, and hence no way to predict the effect of using the address, all manner of strange things can occur. Making this type of bug extra difficult to track down is the fact that passing parameters unsafely works most of the time—it only fails when the heap is so full that the Memory Manager must relocate memory to satisfy a request. You do not want these kinds of bugs in your program.

   Fortunately, Object Pascal programmers have a big advantage over their conventional colleagues: the compiler actually tells you when a field of an object is used in a potentially unsafe way. This occurs for **VAR** parameters, which by definition are passed by address, and for non-**VAR** parameters whose size is greater than four bytes. The latter case is because the compiler actually passes such parameters by address, expecting the called routine to use the address to make a local copy of the data.

   If you stop to think about it, this error message is a really nice feature. Especially when compared to what the compiler does when any other handle is unsafely dereferenced, which is nothing. Nada. Zip. Even the most experienced and handle-cognizant of programmers occasionally writes code that unsafely dereferences a handle.

   Let's look at an example. Consider the following definitions:

```
TYPE
  TShape = OBJECT (TObject)
     fBounds:   Rect;
     fColor:    RGBColor;
     END;

VAR
     aShape:    TShape;
```

Attempting to compile the line

```
OffsetRect(aShape.fBounds, 10, 20);
```

results in the following error:

```
#   OffsetRect(aShape.fBounds, 10, 20);
#               ?
### pascal - Error 815 Unsafe use of an object field as a var
or > 4 byte parameter
```

In other words, this line of code has dereferenced the object's handle at a time when the object may move while it is dereferenced. In this case, the address of the field **fBounds** is computed and passed to **OffsetRect**. If **aShape** were to move before **OffsetRect** used it, then the computed address wouldn't point at **fBounds** anymore. Bombs away! Maybe the message ought to read "Error 815 You are about to commit yourself to spending an indeterminant number of days working with Macsbug. Please reconsider."

A simple way of avoiding the error is to avoid using the field as a parameter. Instead, use a temporary variable:

```
VAR
  r:    Rect;

r := aShape.fBounds;
OffsetRect(r, 10, 20);
aShape.fBounds := r;
```

While this construct is guaranteed to be safe, it could be rather onerous if you had to do this every time you wanted to use an object field as a parameter. Actually, it turns out that many cases can easily be identified as safe because the routine being called doesn't trigger memory relocation. But how do you *know* when it's safe? Mostly, you need to know what causes objects to move.

Memory relocation can be triggered if (a) the called routine is in a different segment from the caller, since loading a segment may trigger memory relocation; (b) the called routine calls a ROM routine that triggers relocation; or (c) the called routine calls another routine that fits the criteria of (a) or (b). In the case of **OffsetRect**, it's in ROM so it won't require a segment load, and it is a ROM routine that doesn't move memory. (I know that because it isn't listed in Appendix A of the *Inside Macintosh XRef*.)

When you do know it's safe (as with **OffsetRect**), you can turn off the compiler's parameter checking, effectively telling the compiler to keep quiet because you know what you're doing. Do this by using the **$H** compiler directive:

```
 {$Push}{$H-}
OffsetRect(aShape.fBounds, 10, 20);
 {$Pop}
```

The first line turns off parameter checking. **$Push** saves the state of the compiler directives; **$H-** tells the compiler not to check parameters for unsafe usage. In the second line the object field is used as a parameter. Because **$H-** was issued, no compiler error is generated. The third line uses **$Pop** to restore the state of the directives at the time the last **$Push** was issued.

The trick, of course, is in knowing when to use **$H** and when to use a copy of the object field instead. Based on the three causes of memory relocation, it's possible to identify the conditions in which you should avoid passing the field of an object as a parameter.

*Don't* pass a field of an object as a **VAR** parameter, or a field greater than four bytes in size, in these circumstances:

a. When the called routine is listed in Appendix A of the *Inside Macintosh XRef*.

   Appendix A lists routines defined in *Inside Macintosh*, volumes I-V, that may trigger memory relocation. These include all system-defined routines, such as those in ROM and in packages. Any routine defined in *Inside Macintosh*, volumes I-V, that does not appear in Appendix A will not trigger memory relocation. (A similar appendix appears in each *Inside Macintosh* book, but only applies to that book. So use the appendix in the *XRef* because it applies to all five volumes.)

b. When the called routine is in a different, nonresident segment from the code generating the call.

   Calling a routine in another segment may require loading it into memory, potentially triggering memory relocation. If the called routine is in the same segment as the caller, then the segment must already be in memory and you're safe. If the called routine is in a different segment from the caller, you're still safe if the called routine's segment is a resident.

   Resident segments are defined by MacApp® to be segments that are loaded into memory when the program starts up, and that stay in memory throughout the life of the program. Thus calling a routine in a resident segment never requires loading it into memory. If you know a routine is in a resident segment, you can call it without worrying about a segment load relocating memory. If you're not sure a routine is in a resident segment, play it safe.

c. When the called routine is in the same segment as the caller, but the called

routine indirectly causes segment loads by calling routines in other, nonresident segments. If you don't know whether a routine does this, then play it safe.

d. When the called routine calls ROM routines that may trigger memory relocation. Again, if you're not sure, play it safe.

2. Don't pass a field of an object or handle as the parameter to **NEW**.

   The parameter to **NEW** is a **VAR** parameter, and since **NEW** calls **NewHandle**, it most definitely may trigger memory relocation. Unfortunately, MPW Pascal compilers before version 3.1 didn't generate an error for **NEW** when you passed an object field as the parameter, and TML Pascal version 3.0 still doesn't. So be careful.

   Note that MacApp contains *functions* that allocate objects as well. The **Clone** method copies an object, returning a reference to the copy as its result. **NewObjectByClassName** and **NewObjectByClassId** create new objects. Because of the way the compiler generates code for functions, it *is* safe to assign a function result to the field of an object.

3. Call **FailNIL** after *every* attempt to create an object, copy an object, or create a handle or pointer.

   **FailNIL**, defined in MacApp's **UFailure** unit, has a single parameter—a reference to an object, handle, or pointer. If that reference is **NIL** then **FailNIL** signals failure, essentially causing the application to back out of what it was doing and resume processing events. Calling **FailNIL** is how you verify that a memory allocation request actually succeeded. It works because the Memory Manager returns **NIL** if there isn't enough memory to satisfy a memory allocation. Since heaps have a finite amount of space, the potential exists that *any* allocation request can fail. So check *each and every* request just to make sure. Failure to heed this advice leads to bus errors and address errors when your program tries to dereference **NIL** handles.

   Be aware that this description of **FailNIL** just scratches the surface of MacApp's failure-handling facilities. The MacApp technical manuals go into greater detail than space permits in this article.

4. Remember to free objects when you've finished with them, *but only when you've finished with them!*

   The Macintosh doesn't have automatic garbage collection, so you're responsible for freeing (disposing of) any objects you create. Failing to free objects when you've finished with them leads to a heap that slowly fills up with garbage, eventually suffocating the application.

Keep in mind that when you free an object, any references to it are no longer valid and should be set to **NIL**. Nasty things happen if you use references to objects that no longer exist. If you're lucky, the MacApp debugger will stop your program the first time you refer to a nonexistent object. But sometimes even the MacApp debugger gets fooled. This happens if the memory occupied by the freed object hasn't yet been written over, or even worse, if another object or handle was allocated using the same master pointer as the object that was freed. (Try debugging that sometime!)

5. If necessary, ensure that an object doesn't move by locking it.

   Sometimes it really makes life easier to ensure that an object won't be moved no matter what happens. Objects, like handles, can be locked. In fact, MacApp provides a method for this purpose. It's called **Lock** and it's defined in **TObject** so it can be used on any object. **Lock** takes a **Boolean** value as its only parameter, which when true "locks" the object's location in memory, and when false makes the object relocatable again. **Lock** returns a **Boolean** result that indicates whether the object was locked when **Lock** was called. This is handy because you can lock an object, do what you need to do, then *restore the object's lock state to what it was before.*

```
VAR
  wasLocked:    BOOLEAN;

BEGIN
  wasLocked := anObject.Lock(TRUE);
  {do what you need to do}
  wasLocked := anObject.Lock(wasLocked);
END;
```

   If you're not using MacApp, you can lock an object by casting it to be a handle and using **HLock** and **HUnlock**, the Memory Manager routines for locking and unlocking handles:

```
HLock(Handle(anObject));
{ Do what you need to do. }
HUnlock(Handle(anObject));
```

   Keep in mind that it's unwise to lock objects (or handles) for long periods of time. Nonrelocatable objects cause heap fragmentation, which reduces the effectiveness of the heap. (For further details, see Richard Clark's article "The Secret Life of the Memory Manager" in this issue.)

## PRACTICING SAFE HANDLE USAGE

Even in an object-based program, it's occasionally necessary to use handles instead of objects. While handle usage is subject to the same guidelines just described for objects, there are some additional wrinkles:

1. Don't count on the compiler to tell you when a handle's field is used unsafely.

   Unlike for fields of objects, the compiler doesn't produce an error when passing a field of a handle by address. All of the same problems with using fields of objects apply to fields of handles, but since the compiler offers no help in detecting unsafe uses, it's completely up to you to ensure that you use fields of handles safely. Chalk one up for objects.

2. Beware of **WITH** statements that dereference handles. For example:

```
TYPE
  TShapeHdl = ^TShapePtr;
  TShapePtr = ^TShape;
  TShape = RECORD
       fBounds:   Rect;
       fColor:    RGBColor;
       END;

VAR
  aShape: TShapeHdl;

 BEGIN
  aShape := NewHandle(SIZEOF(TShape));
  FailNIL(aShape);

  WITH aShape^^ DO
    BEGIN
    fBounds := gZeroRect;
    fColor := gRGBBlack;
    END;
END;
```

   Not only does the **WITH** statement simplify the Pascal text, it also lets the compiler perform code optimizations. Specifically, it stores the address from **aShape^^** in a register so that it can be reused throughout the scope of the **WITH** without being recomputed. As you might imagine, any operation that triggers memory relocation within the scope of the **WITH** will invalidate the address contained in the register. Bad news.

   By the way, using **WITH** statements on objects is okay! The compiler recognizes that the **WITH** is dereferencing an object and makes sure safe code is generated. Objects 2, handles 0.

3. Don't assign a function result to a field of a handle unless you know the calling function won't trigger memory relocation.

   For example, using the shape definitions given above, this is a potentially unsafe use of a handle field:

   ```
   aShape^^.fBounds := FunctionThatReturnsARect;
   ```

   The problem is that the Pascal compiler dereferences the handle **aShape** *before* calling the function. Thus, a function that triggers memory relocation or is in another segment will invalidate the address obtained by dereferencing the handle.

   Once again, this type of usage is okay for objects. As for **WITH** statements, the compiler recognizes when a function result is assigned to an object field and ensures that safe code is generated.

## IN CONCLUSION

In conclusion, you *can* avoid the pitfalls in writing Object Pascal programs, by understanding how objects work and by using the guidelines described in this article. Then instead of having to spend undue time debugging, you can relax and enjoy the advantages of object-based programming.

# THE SECRET LIFE OF THE MEMORY MANAGER

*The Macintosh Memory Manager has changed in some subtle ways since it was documented in Inside Macintosh. This, combined with the difficulty of observing what the Memory Manager actually does, has led to a general misunderstanding of how the Memory Manager works. This article first discusses some common myths about the Memory Manager, then describes some ways to avoid memory-related errors and control fragmentation without sacrificing execution speed.*

**RICHARD CLARK**

Few parts of the Macintosh operating system raise as many questions as the Memory Manager. Since the contents of RAM change dynamically, it's hard to really examine the Memory Manager's behavior. This, combined with the unusual concept of relocatable blocks and the fact that the Memory Manager is used by most of the operating system, has left many Macintosh programmers confused about the behavior of the Memory Manager and, more important, about the impact of this behavior on their applications.

## MYTHS ABOUT THE MEMORY MANAGER

Several myths have grown up around the Memory Manager, serving to increase the confusion about its real behavior. Three of the most prevalent—but mistaken—beliefs are that (1) the Memory Manager will move and delete blocks, and otherwise mangle the heap, at random; (2) using nonrelocatable blocks will cause serious heap fragmentation; and (3) if you use Virtual Memory you don't need to worry about the Memory Manager. We'll demolish each of these myths in turn.



**RICHARD "TIGGER" CLARK** wears brightly colored clothes, writes odd graffiti, tells horrible puns, and is amazingly graceful when running for the bus. He earned a BS in social science (which he says is a hybrid psychology/computer science degree) from the University of California-Irvine in 1985. When he's not teaching at Developer U, you can find him stunt-kite flying (sometimes indoors), mountain climbing (sometimes indoors), or collecting Disney memorabilia. An avid reader, he has totally worn out his copy of *Winnie the Pooh* in Latin. In his time he has been a Valley Boy, a Macintosh repairman, a software developer, King Henry VIII's head steward, a Renaissance bishop, and probably a few other things he won't tell us about. But hey, he's from southern California. •

### MYTH 1: THE MEMORY MANAGER WILL MOVE AND DELETE BLOCKS, AND OTHERWISE MANGLE THE HEAP, AT RANDOM

This simply isn't so. The Memory Manager is in fact quite predictable. It only moves blocks under these circumstances:

- When your application calls a routine that allocates new blocks or enlarges existing ones, when you request that blocks be moved, or when your application calls a routine that in turn calls a ROM routine that may trigger block relocation. Appendix A of the *Inside Macintosh XRef* lists all routines defined in *Inside Macintosh* that may cause blocks to move.
- When the called routine is in a different segment from the code that makes the call, or when the called routine is in the same segment as the caller, but the called routine calls a routine or routines in a different segment. If a called routine lies in a different code segment, the Segment Loader may need to call the code segment in from disk and/or move it to the top of the heap. Either of these actions can cause blocks to move.

### MYTH 2: USING NONRELOCATABLE BLOCKS WILL CAUSE SERIOUS MEMORY FRAGMENTATION

This is a half-truth at best. The Memory Manager actually does a good job of allocating nonrelocatable blocks, but can fragment the heap when these blocks are deallocated and new ones allocated. Similar problems can happen when you start locking relocatable blocks.

This myth actually has a basis in reality, as the earliest versions of the Memory Manager did a poor job of allocating nonrelocatable blocks. Before the 128K ROMs (introduced with the Macintosh 512Ke and Macintosh Plus), the Memory Manager would not move a relocatable block around a nonrelocatable block in its quest to allocate a new nonrelocatable block. This made the heap into a patchwork of relocatable and nonrelocatable blocks, and caused general fragmentation problems, as illustrated in Figure 1.

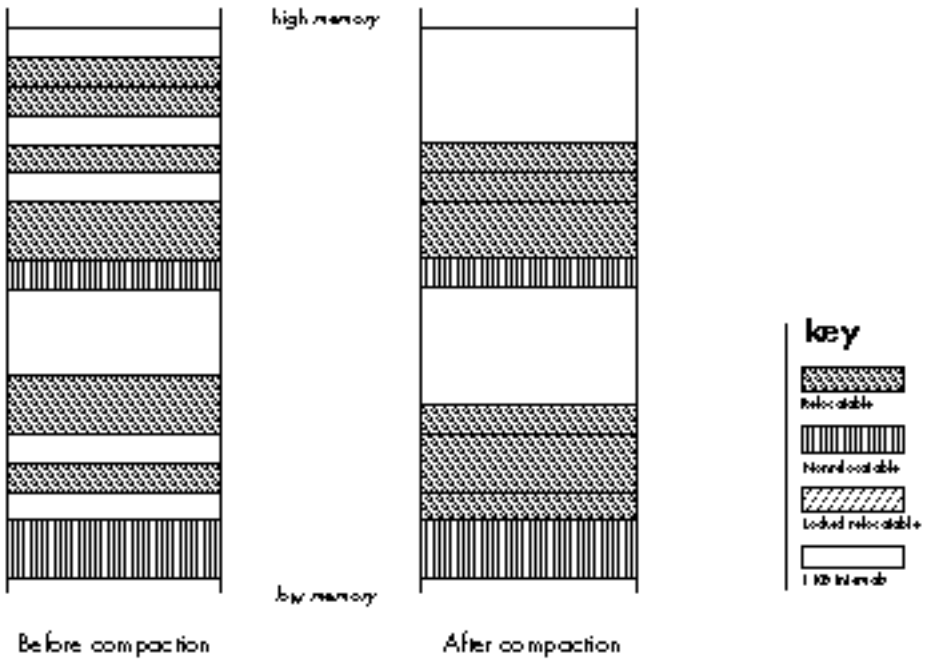



**Figure 1.**
**Fragmentation of Free Space**

But that has long since changed, as **`NewPtr`** will now move a relocatable block around a nonrelocatable block when allocating memory. This tends to partition the heap into two active areas, with all of the nonrelocatable blocks at the bottom of the heap, and the relocatable blocks located immediately above. (See the sidebar "How the Memory Manager Allocates Heap Space" for further details.)

## HOW THE MEMORY MANAGER ALLOCATES HEAP SPACE

The Memory Manager uses two basic techniques to create space for blocks on the heap: compaction and reservation. It uses compaction to create space for new relocatable blocks, and reservation to create space for new nonrelocatable blocks.

When your application (or the operating system) calls **`NewHandle`** to allocate a new relocatable block, the Memory Manager first looks for a large enough space to hold a block of the requested size. If a large enough space is found (and it need not be a perfect fit), the block is allocated. If there is not enough free space to satisfy the request, compaction takes place—relocatable blocks are moved downward (toward low memory) to make space for the new block.

As a rule, the Memory Manager allocates new relocatable blocks as low in the heap as possible without compaction. If the heap must be compacted, the Memory Manager begins with the lowest blocks and gradually works its way upward until it has created a large enough free space to accommodate the new relocatable block or until the entire heap has been compacted.

On the other hand, when your application (or the operating system) calls **`NewPtr`** to allocate a new nonrelocatable block, the Memory Manager calls **`ResrvMem`** to create an empty space at the bottom of the heap for the new nonrelocatable block. This technique is known as reservation (after the call), although you won't find this term anywhere in *Inside Macintosh*.

The Memory Manager always allocates nonrelocatable blocks as low as possible on the heap, even if it means that other blocks have to be moved. In the case shown in Figure 2, the Memory Manager has to move a relocatable block twice when the user allocates two nonrelocatable blocks. Note that each time the 4KB relocatable block is moved, it leaves a 4KB space behind. This is a result of the way the Memory Manager reserves memory. It first moves the block upward into the first free area above its former position large enough to hold it, then uses the old space for the new block.

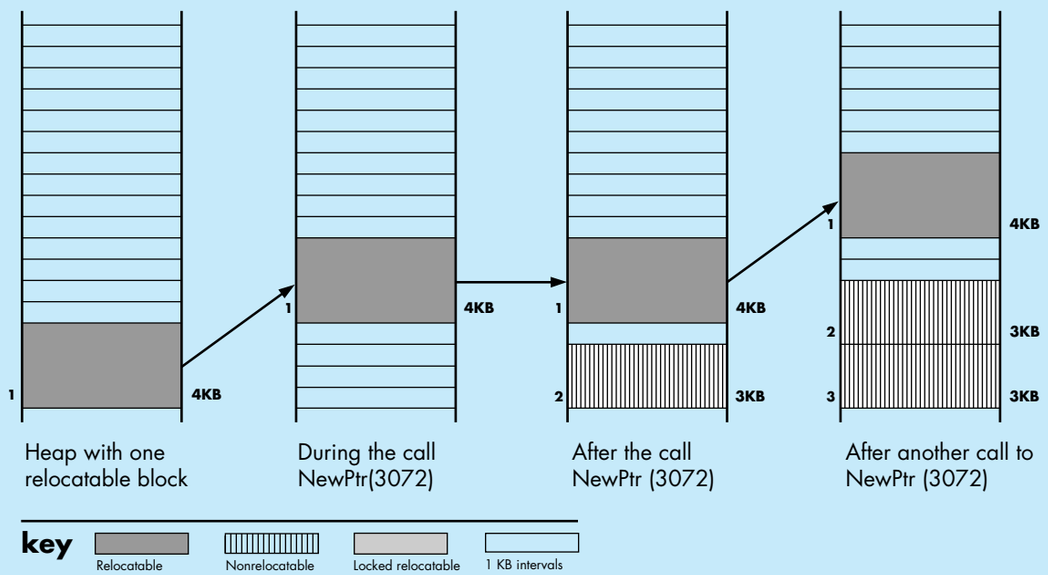



**Figure 2.**
**The Effect of Allocating Nonrelocatable Blocks**

In summary, allocating a new nonrelocatable block is likely to move other (relocatable) blocks upward, while allocating a new relocatable block may cause compaction, which moves relocatable blocks downward.

On the other hand, for all of the improvements in allocation of nonrelocatable blocks, there is still a problem with *de*allocation of these blocks. Since the Memory Manager uses a "find the first free block that fulfills the request" strategy (as opposed to "find a block that fits the request exactly"), if you allocate a subsequent block that is smaller than the block you just deleted, the heap will become fragmented and the amount of usable memory will likely decrease, as illustrated in Figure 3.

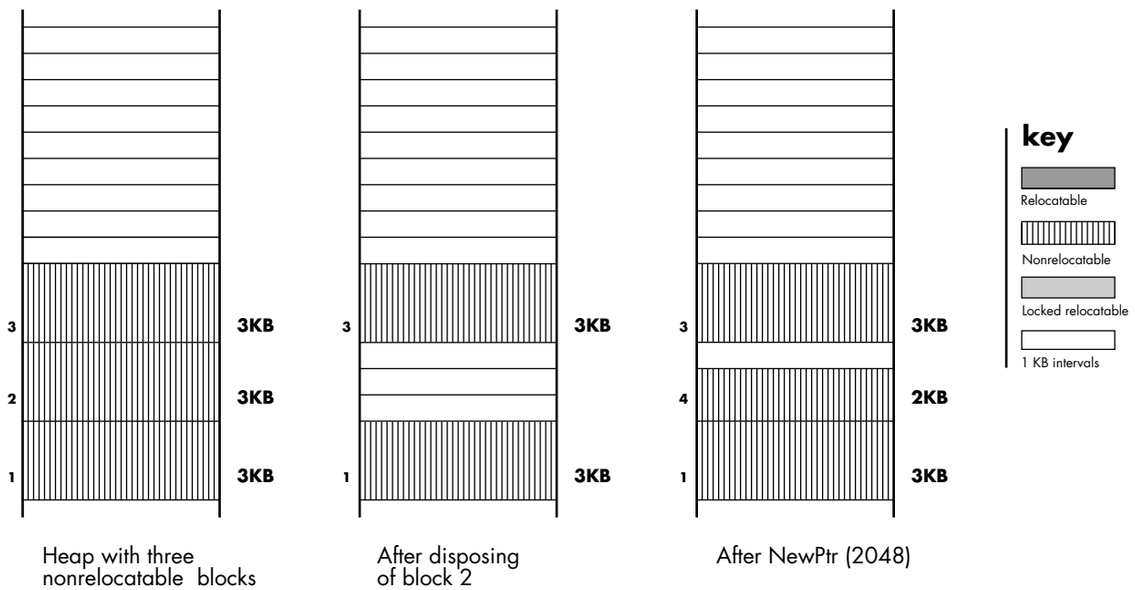



**Figure 3.**
**The Effect of Deallocating and Reallocating a Nonrelocatable Block**

Locking too many relocatable blocks can cause the same kind of fragmentation problems as deallocating and reallocating nonrelocatable blocks. A well-trained programmer uses the call **`MoveHHi`** to move a relocatable block to the top of the heap before locking it. This has the effect of partitioning the heap into four areas, as shown in Figure 4. The idea of using **`MoveHHi`** is to keep the contiguous free space as large as possible. However, **`MoveHHi`** will only move a block upward until it meets either a nonrelocatable block or a locked relocatable block. Unlike **`NewPtr`** (and **`ResrvMem`**), **`MoveHHi`** will not move a relocatable block around one that is not relocatable.

Even if you succeed in moving a relocatable block to the top of the heap, your problems are far from over. Unlocking or deleting locked blocks can also cause fragmentation, unless they are unlocked beginning with the lowest locked block. In

the case illustrated in Figure 4, unlocking and deleting blocks in the middle of the locked area has resulted in heap fragmentation. The relocatable blocks thus trapped in the middle won't be moved until the locked block below them is unlocked.

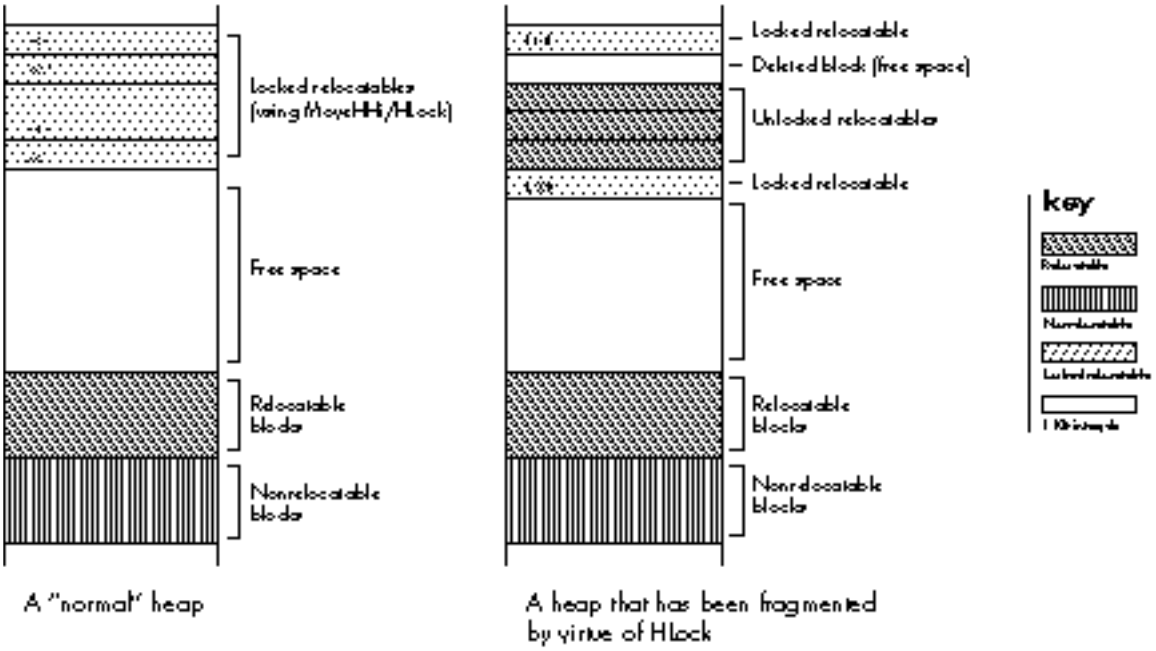



**Figure 4.**
**The Effect of Unlocking Locked Blocks**

### MYTH 3: IF YOU USE VIRTUAL MEMORY, YOU DON'T NEED TO WORRY ABOUT THE MEMORY MANAGER

Many people believe that the wide availability of Virtual Memory will remove the need for careful memory management. Wrong! The Virtual Memory system is based on a series of "pages" of memory that can be swapped to and from the disk, rather than on individual blocks of memory. If you fragment RAM, you also "fragment" the contents of the swap file and gain nothing. In fact, Virtual Memory makes careful memory management even more critical, for two reasons. First, fragmenting the swap file will degrade system performance worse than fragmenting physical memory will, since disk access speeds are obviously slower than the RAM access speed. Second, the combination of Virtual Memory and MultiFinder encourages users to run more programs at the same time than they used to, and users often reduce the partition sizes of their applications to squeeze in "one more program."

## THE EXPERT'S GUIDE TO MEMORY MANAGEMENT

Now you know that the Memory Manager moves blocks of memory only at certain well-defined times; that nonrelocatable blocks can be allocated without causing serious fragmentation in the heap, although deallocation and reallocation of these blocks, and locking too many relocatable blocks, can cause problems; and that use of Virtual Memory makes careful memory management even more important. It's time to put this knowledge into action. In this section, you'll learn how you can work cooperatively with the Memory Manager to increase the efficiency and robustness of your applications.

### TO AVOID DANGLING POINTERS

As every programmer learns early on, the gravest side effect of the Memory Manager's penchant for moving blocks of memory is the peril of dangling pointers. (For a refresher on how these come about, see the sidebar entitled "A Primer on Handles and Their Pitfalls" in Curt Bianchi's article "Using Objects Safely in Object Pascal" in this issue.) And the best defense against having to spend hours—or days—debugging errors caused by dangling pointers is to anticipate situations in which block movement might occur, and if it does occur, will throw a monkey wrench into the works. In these situations, much grief can be saved by using a temporary local or global variable to store a duplicate of the relocatable block. (Note, though, that this trick only works properly if the block can stand on its own—that is, it's not part of a linked list.)

### ABOUT LOCAL AND GLOBAL VARIABLES

While in this article we're primarily interested in information stored on the heap, there are actually three places you can store information in memory: in a relocatable or nonrelocatable block on the heap, in a local variable, or in a global variable. In terms of storage efficiency, relocatable blocks are your best bet. But if you need to store information in an area that will not move, you can use local or global variables.

Local variables are allocated on the machine's stack, and only exist as long as the enclosing procedure is running. Global variables are stored in a special block above the top of the application's stack and heap, and exist as long as the program is running. Both of these areas share one disadvantage: limited space. You can only allocate 32KB of global variables, and the maximum available stack space typically varies between 8KB and 24KB, depending on the machine, the operating system version, and whether or not the application has requested a larger stack when launched.

Some of the situations that might get you into trouble are well documented, such as the use of the WITH statement in Pascal. Other dangerous situations are less obvious, so we'll explore them here.

**Be careful when evaluating expressions.** There are times when evaluating a seemingly innocent expression might have serious side effects. For example, look at the following code:

```
TYPE
 windowInfoHdl = ^windowInfoPtr;
 windowInfoPtr = ^windowInfo;
 windowInfo = RECORD
  aControlHdl: ControlHandle;
  aWindowPtr:  WindowPtr;
 END;

VAR
 myHandle : windowInfoHdl;

BEGIN
 myHandle := windowInfoHdl(
           NewHandle(sizeof(windowInfo)));
 { The next 2 statements have problems. }
 myHandle^^.aWindowPtr := GetNewWindow(1000, NIL, WindowPtr(-1));
 myHandle^^.aControlHdl := GetNewControl(1000, myHandle^^.aWindowPtr);
END;
```

In Pascal, the above statements would probably cause a run-time error. The problem is in the expression "**myHandle^^.something :=**" as the compiler evaluates expressions from left to right and calculates the address on the left side of the assignment statement *before* making the toolbox call. When **GetNewWindow** is called, **myHandle^^** is moved (we passed in **NIL** to force a call to **NewPtr**) *and the address on the left-hand side is no longer valid!* This means that the returned **WindowPtr** will be written into the wrong area of memory, and the program will probably crash.

While both statements suffer from the same basic problem, the first one is more likely to cause a crash than the second one and is therefore easier to debug. Why is this?

The statement containing **GetNewWindow** will make a call to **NewPtr** to allocate a nonrelocatable block at the bottom of the heap, forcing relocatable blocks upward in the process. The other statement, containing **GetNewControl**, allocates a relocatable block, which usually appears above the existing blocks, with block movement happening only if a compaction is required.

While this problem occurs most frequently in Pascal, C programs are not immune. Most C compilers on the Macintosh evaluate the right-hand side of an assignment before the left-hand side—which avoids this problem entirely—*but the order of evaluation is not guaranteed by the ANSI standard.*

This problem can be solved easily by using a temporary variable. The following code avoids the problem:

```
VAR
 myHandle:    windowInfoHdl;
 aWindowPtr:   WindowPtr;         { This is allocated on the }
                                  { stack, so it won't move. }
 aControlHandle: ControlHandle;   { Also on the stack. }

BEGIN
 myHandle := windowInfoHdl(NewHandle(sizeof(windowInfo)));

 { Copy the result into a temporary variable, then copy }
 { that into the relocatable block. }
 aWindowPtr := GetNewWindow(1000, NIL, WindowPtr(-1));
 myHandle^^.aWindowPtr := aWindowPtr;

 aControlHandle := GetNewControl(1000, aWindowPtr);
 myHandle^^.aControlHdl := aControlHandle;
END;
```

**Be careful when using callback routines.** When you pass pointers to your routines, say as a ROM callback routine, and your routines are in multiple segments, you need to be careful.

The following code is fine now, but we'll soon edit it to demonstrate the problem:

```
{$S Main }
PROCEDURE MyCallback(ctl: ControlHandle; part: INTEGER);
{ This represents a callback routine used for continuous }
{ tracking in controls. }
BEGIN
 { Do whatever you need to do. }
END;

PROCEDURE HandleMyControl(theControl: ControlHandle;
            pt: Point);
BEGIN
 part := TrackControl(theControl, pt, @MyCallback);
END;
```

The expression **`@MyCallback`** pushes the address of the callback routine onto the stack before calling **`TrackControl`**. If the two routines are in the same segment, as in the preceding example, all is fine. The segment is locked in memory when **`@MyCallback`** is both evaluated and used; therefore, the address is valid. If the two routines are in different segments, this also works, as the compiler takes the address of the jump table entry for **`MyCallback`**.

In some cases, and especially in C, you may choose to set up a table of procedure addresses. But if you store the address of the routine into a variable, strange things may happen. Take a look at the following code:

```
{ ——————————————————————— }
{ For an example, we'll place the addresses of two control }
{ tracking routines into an array, then use them. }

VAR
 gCallbackArray: ARRAY [1..2] OF ProcPtr;

{ ——————————————————————— }
{$S Segment1 }

 PROCEDURE MyVScrollCallback(theControl: ControlHandle;
              part: INTEGER);
 BEGIN
  { This will get called if our control is a vertical }
  { scrollbar. }
 END;


 PROCEDURE MyVScrollCallback(theControl: ControlHandle;
              part: INTEGER);
 BEGIN
  { This will get called if our control is a horizontal }
  { scrollbar. }
 END;

 PROCEDURE InitCallbackArray;
 { Fill in the addresses in the global "Callback" array. }
 BEGIN
  { Problem: Since we're in the same segment, these aren't }
  { addresses of the jump table entries, but are absolute }
  { locations in RAM! If the segment moves (i.e., if }
  { UnloadSeg is called), the addresses will be invalid. }
  gCallbackArray[1] := @MyVScrollCallback;
  gCallbackArray[2] := @MyHScrollCallback;
 END;
END.

{ ——————————————————————— }
{$S Main }

PROCEDURE HandleAScrollbar(theControl: ControlHandle;
            pt: Point);
{ We'll call this if the user clicks in our scrollbar (except }
{ if she clicks in the thumb, which uses a different kind of }
{ callback.) If it's a vertical scrollbar, use one callback; }
{ if horizontal, use the other. }
```

```
VAR
 part:    INTEGER;
 theCallback: ProcPtr;
 isVertical: Boolean;
 aRect:    Rect;
 cntlWidth:  INTEGER;

BEGIN
 aRect := theControl^^.cntrlRect;
 cntlWidth := aRect.right - aRect.left;
 isVertical := cntlWidth = 16;
 IF isVertical THEN
  part := TrackControl(theControl, pt, gCallbackArray[1])
 ELSE
  part := TrackControl(theControl, pt, gCallbackArray[2])
 { The TrackControl calls will probably crash if }
 { Segment1 has been unloaded since the table was built. }
 { You'll have a wonderful time trying to find the bug! }
END;
```

When setting up a table of such procedure addresses, or even a single global variable, you should do one of the following things: (1) make sure that the setup procedure is in a different segment from the procedures being called, thus insuring that you get the address of a jump table entry; (2) keep *everything* in one segment and never unload it; or (3) always load the segment and build the table before using any of the addresses (and make sure that the segment doesn't get unloaded in the meantime).

**Be careful when passing parameters.** Another problem area occurs when you pass parameters to routines that allocate or move memory. Can you spot the problem in the following code?

```
PROCEDURE ValidateControl(theControl: ControlHandle);
BEGIN
 ValidRect(theControl^^.contrlRect);
END;
```

`ValidRect` receives the address of a rectangle, which is pushed onto the stack before the trap is called. The problem is that before `ValidRect` uses the rectangle's address, it often allocates memory of its own, which can cause `theControl^^` to move and therefore invalidate the rectangle's address.

This problem happens when you pass (1) any parameter larger than four bytes, or (2) any `VAR` parameter. Again, the solution requires a temporary variable:

```
PROCEDURE ValidateControl(theControl: ControlHandle);
VAR
 r : Rect; { r is stack-based, so it doesn't move. }
```

```
BEGIN
 r := theControl^^.contrlRect;
 ValidRect(r);
END;
```

Pascal compilers often avoid this problem for user-defined functions by making a local copy of non-**VAR** parameters that are passed by address. The ROM doesn't make such a copy, so you need to be careful. This is discussed at length by Scott Knaster in *How to Write Macintosh Software*, 2nd ed. (Hayden Books, 1988).

### TO CONTROL HEAP FRAGMENTATION

As you will recall, heap fragmentation can be caused by (1) deallocating and reallocating nonrelocatable blocks, and (2) locking too many relocatable blocks. To keep heap fragmentation under control, follow a few simple rules.

**Use nonrelocatable blocks sparingly.** To avoid the potential problems that deallocation and reallocation of nonrelocatable blocks can cause, you should theoretically use relocatable blocks for everything. However, in practice, there are areas where you *must* use nonrelocatable blocks, such as for **GrafPorts** and **WindowRecords**. In light of this reality, here are three suggestions to help you control fragmentation.

First, remember that you should not choose to use nonrelocatable blocks lightly. Use them only when the Macintosh operating system requires them, or when you can *demonstrate* a severe performance penalty for using relocatable blocks.

Second, avoid allocating nonrelocatable blocks unless they will never be deleted. If you know about such blocks ahead of time, then you can allocate them at program start-up. This works well if you'll have a single large "image buffer" or the like, or a limit on the number of available windows. In these cases, allocating your large fixed blocks at start-up time will avoid potential fragmentation problems.

Third, if you must allocate and deallocate nonrelocatable blocks on demand, you can add some additional memory management code of your own. When you want to deallocate a block of RAM, you can add it to a linked list of free blocks (that you maintain), and then check this list for a free block of the exact size you need each time you want to allocate a new block. Of course, this works best if the range of block sizes you support is limited, and you still have to decide what to do if the block you want doesn't fit any of the free blocks exactly. If you have to allocate a large number of nonrelocatable blocks, or have other special needs, you should consider allocating a large block of memory and doing your own memory management within that. Donald Knuth's book *The Art of Computer Programming*, volume 2, 2nd ed. (Addison-Wesley, 1973) contains a useful overview of memory management techniques under the heading "Dynamic Storage Allocation" (pp. 435-55 and 460-61).

Note that this strategy of reusing nonrelocatable blocks works best under the 128K ROM (and later) Memory Manager, since that version does the best job of allocating nonrelocatable blocks. If you plan to write software under the 64K ROMs (Macintosh 128K or 512K), you should consult Scott Knaster's *How to Write Macintosh Software*, which describes a strategy that does a better job with the old Memory Manager than this strategy does.

**Lock selectively and consider alternatives.** Fear of dangling pointers often drives new programmers to lock down everything in sight, quickly fragmenting the heap and impeding the application's performance. More experienced programmers try to avoid locking relocatable blocks, preferring instead to predict when the Memory Manager will move blocks of memory and then only locking a relocatable block if they must. If done infrequently, locking has a negligible impact on your application.

If you must lock a relocatable block, you should unlock it as soon as possible. This will lessen the probability of another block being moved in underneath (by **MoveHHi**) and locked. Also, if you move and lock several blocks together, you should unlock all of them together, or at least in the reverse of the order in which they were moved high. This will help ensure that the free area is kept together in the heap.

As an alternative to locking relocatable blocks, consider using temporary variables. We've already seen the use of temporary variables for such small items as window pointers and rectangles, but this approach can also be used for entire structures. Using temporary variables can simplify your code by removing the need for **HLock** and **HUnlock** calls. For example, many programs use a window's reference constant (**RefCon**) field to hold a handle to a data structure. Programs that do so look something like this:

```
TYPE
 windowInfoHdl = ^windowInfoPtr;
 windowInfoPtr = ^windowInfo;
 windowInfo = RECORD
  rectArray: ARRAY [1..10] OF Rect;
 END;

PROCEDURE UpdateWindow(wp: WindowPtr);
{ The window's RefCon contains a handle to the data structure
shown }
{ above. The rectArray field contains an array of rectangles that
}
{ we want to draw. }

VAR
 myHandle: windowInfoHdl;
 count:   INTEGER;

BEGIN
```

```
 { Get the window information, then lock the block so that it }
 { doesn't move while drawing the rectangles. }
 myHandle := windowInfoHdl(GetWRefCon(wp));
 MoveHHi(Handle(myHandle));
 HLock(Handle(myHandle));

 BeginUpdate(wp);
 FOR count := 1 TO 10 DO
  { Working with the heap-based window information, draw each }
  { rectangle. }
  FrameRect(myHandle^^.rectArray[count]);
 EndUpdate(wp);
 HUnlock(Handle(myHandle));
END;
```

Notice that we had to perform several type casts, and use **MoveHHi**, **HLock**, and **HUnlock**. Now, let's see how this would look using a temporary variable:

```
{ Type declarations omitted for brevity. }

PROCEDURE GetWindowInfo (wp: WindowPtr; VAR info: windowInfo);
{ Utility routine to make a copy (usually stack-based) of our }
{ window information structure. }
VAR
 myHandle: windowInfoHdl;

BEGIN
 { First, do a little error checking. }
 IF (wp <> NIL) THEN BEGIN
  myHandle := windowInfoHdl(GetWRefCon(wp));
  { You can incorporate extra error checking here. For example, }
  { this is a good place to compare the handle's size to the }
  { window information structure's size, or to verify that the }
  { contents of the block are legal values. }
  {                              }
  { Next, go ahead and copy the contents of the relocatable }
  { block to the specified location. We don't have to lock }
  { things down, since BlockMove won't cause compaction. }
  BlockMove(Ptr(myHandle^), @info, sizeof(windowInfo));
 END;
END;

PROCEDURE UpdateWindow(wp: WindowPtr);
VAR
 info: windowInfo;  { This storage is on the stack, therefore it }
                    { won't move. }
```

```
BEGIN
 GetWindowInfo(wp, info);  { Get a copy of the window information.
}
 BeginUpdate(wp);
 FOR count := 1 TO 10 DO
  { Working with the stack-based copy of the window information, }
  { draw each rectangle. }
  FrameRect(info.rectArray[count]);
 EndUpdate(wp);
END;
```

This approach has two major advantages: safety and code simplification. If you have one central routine that gets the window information (and another similar one to set it), you can add quite a bit of error checking and catch a large number of potential errors. Speed shouldn't be a problem, as the single **`BlockMove`** operation is generally faster than the corresponding **`MoveHHi`** since the latter may need to move an old relocatable block out of the way first.

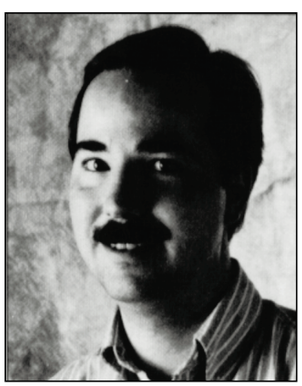
Of course, you have to be *extremely* careful when using this technique, as it is easy to exceed the stack size limit when using recursive or heavily nested procedures. If you have a series of nested procedures that all use the window information structure, you can get the structure in the topmost procedure and pass the block down as a **`VAR`** parameter (pass-by-address in C) so that an extra copy of the data structure isn't made.

## FINAL WORDS OF ADVICE

In this article, we've taken a quick look inside the Memory Manager, but we have not been able to cover everything. If you want to have a fuller understanding of Macintosh memory management, there are a few things you can do. First, reread chapter 3 of *Inside Macintosh*, volume I, and chapter 1 of *Inside Macintosh*, volume II. Next, take a look at Scott Knaster's *How to Write Macintosh Software*, mentioned earlier, which has an excellent discussion of memory management. (In fact, I recommend the book highly to anybody who wants a better understanding of developing and debugging Macintosh software.) Finally, examine the Memory Manager's behavior in real-life situations. develop, the disc, contains the source and object code for the Heap Demo application, which sets up a small heap independent of the main application heap and allows you to manipulate handles and pointers in that environment. If you do these things, you'll be well on the way to mastering the Memory Manager.

# SPEED YOUR SOFTWARE DEVELOPMENT WITH MacApp

*Using MacApp, Apple's object-based application framework, saves time and effort for programmers, and results in an application with the authentic Macintosh look and feel. Developing a Macintosh application can become a simple matter of selecting and integrating functionally specific routines with MacApp and letting MacApp take care of the user interface and other standard application behavior, as this article shows.*



CHRIS KNEPPER

Wouldn't it be nice if you could develop a Macintosh application using previously existing routines? Think of the time and effort you could save if you were able to integrate functionally specific routines from an application you'd written for another platform. Or if you were able to obtain such routines from a public source and use them in your Macintosh application. Or if you could develop such routines yourself, in a language of your choice, and then use them in multiple applications.

And wouldn't it be nice if you had available to you libraries of routines that did the tedious work of creating the interface Macintosh users have come to expect? You wouldn't have to spend time and effort making sure your application did all the things a well-behaved Macintosh application should do.

Dream no more. MacApp makes all of this possible.

## INTRODUCING MacApp

MacApp is an application framework—a skeletal structure for a program that must be fleshed out before it is useful. The bones of this skeletal structure are the MacApp libraries, which handle standard application behavior such as initialization; accessing documents; managing user interface components, such as windows, buttons, scrollbars, and text; managing memory; and handling user input. You flesh out this skeleton by adding functionally specific routines and application-specific code. Figure 1 shows how these pieces fit together.



**CHRIS KNEPPER** is this issue's token beer connoisseur. He's proud that he has never drunk a Mountain Dew in his life, and never plans to. He received a BSEE/CS from Stanford in 1984, and since then has worked in a Dickensian industrial sweat shop (he won't say exactly why or where) and at a small Macintosh consulting firm. Since he came to Apple in 1986, he has done a variety of jobs including software testing, developer technical support, and work for Apple Integrated Systems—all the while consuming record amounts of coffee. When he's not hanging out at local breweries, he's reading, cycling, rooting for the San Francisco Giants, or playing his favorite sport. What is it? Hint—he's been saving his pennies for his dream vanity plate:TNSNE1. . . •

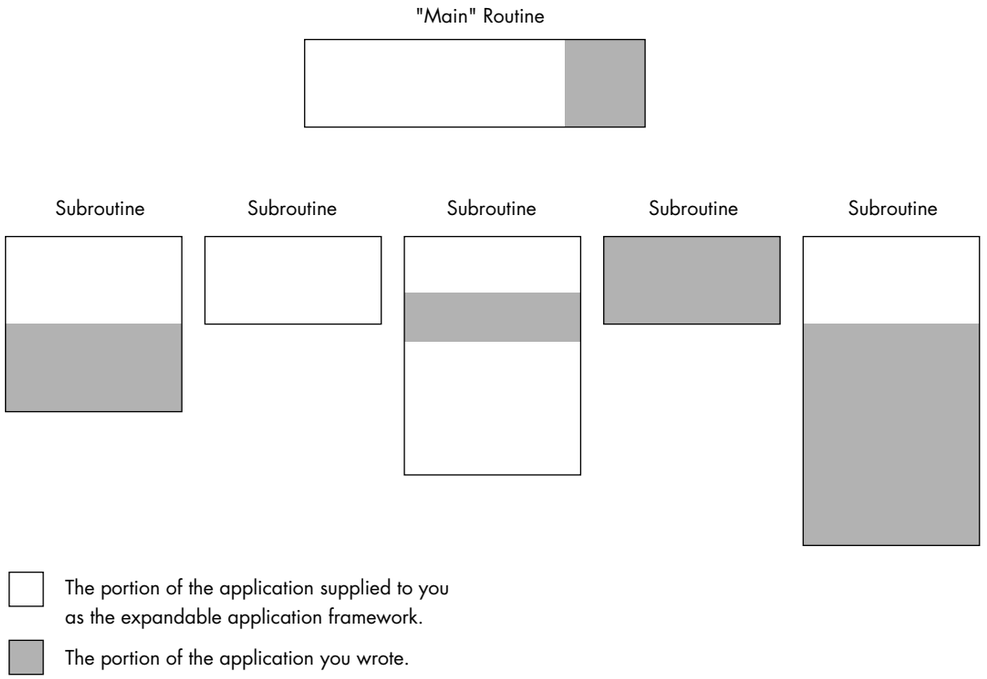



**Figure 1.**
**How MacApp Relates to Your Application**

Because of Apple's commitment to MacApp, the MacApp libraries have been maintained and have matured over time. This has produced libraries that are both versatile, having been used in many applications to address a variety of needs, and robust, because they've been tested and debugged in hundreds of applications and on a wide variety of Macintosh configurations. You can use the code as is or modify pieces that don't meet your needs exactly.

The MacApp libraries are written in Object Pascal, and are distributed via APDA along with interfaces in Object Pascal and C++. Also, p1 Modula-2, Version 4.1, an object language based on Modula-2 now available from the MacApp Developer's Association is fully compatible with MacApp 2.0 and includes interfaces to the MacApp libraries. MPW allows you to develop in Object Pascal, C, C++, FORTRAN, and Modula-2 and still get the benefits of MacApp.



**MacApp**, its manuals, and other useful books on MacApp and object programming are available from APDA. •

## WHAT MacApp CAN DO FOR YOU

MacApp can speed your application development process and help you create more robust applications for the Macintosh. Specifically, MacApp manages the user interface, handles events, implements memory management services, manages printing services, provides basic debugging services, and gives you high-level access to code via Mouser. In addition, when you use MacApp, a number of support organizations and class libraries are available to you. We'll take a closer look at each of these benefits.

**Manages the user interface.** Macintosh users are a demanding audience, having grown accustomed to the Macintosh's distinctive look and feel. Apple has explicitly defined the elements of this look and feel in its Human Interface Guidelines, available from APDA. If your application is to succeed, it must conform to these guidelines. The most significant benefit of using MacApp as your application framework is that it provides for all aspects of Apple's Human Interface Guidelines. MacApp handles user interaction, creates draggable, resizable windows, supports pull-down menus, and provides default behavior for a number of contingencies.

Furthermore, MacApp ships with a tool called ViewEdit that enables you to graphically manipulate and edit the user interface aspects of your software, such as the location, size, and text of buttons and scrollable lists. Creating a dialog box with various controls becomes a simple matter of sketching out these items much as you would sketch a drawing with MacDraw. Figure 2 shows the ViewEdit editing window from the DemoDialogs example that comes with MacApp, offering the programmer the chance to edit the Save As dialog.

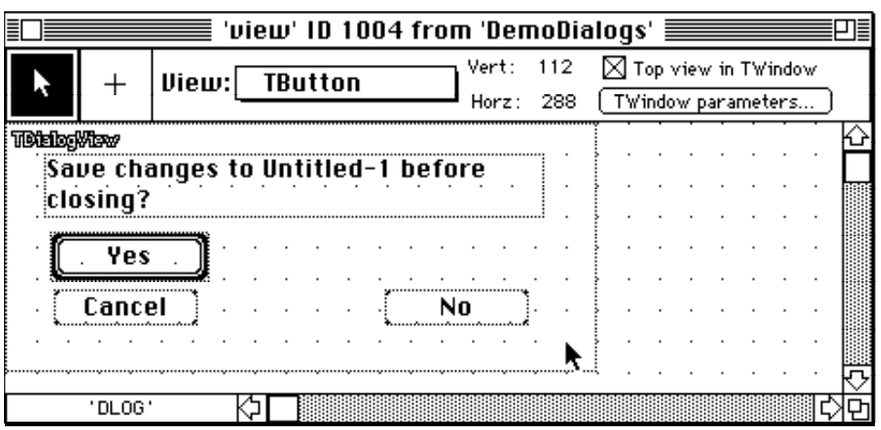



**Figure 2.**
**Editing the Save As Dialog in the ViewEdit Window**

**Note** that whatever work you do with MacApp and Object Pascal is restricted to the Macintosh, because Apple's implementation of Object Pascal has not been endorsed by other vendors on other platforms. If you want to eventually use your code on another platform, consider coding in C++, as C++ compilers are available on other platforms. Of course, you will need to be careful how you structure your application as it develops on the Macintosh (and vice versa) to ensure compatibility across hardware platforms. •

**Handles events.** User interaction produces events that an application gets through the Main Event Loop. Programming this code from scratch is both time-consuming and difficult. MacApp frees you from this requirement, managing the extensive code to handle events and dispatching them accordingly.

**Implements memory management services.** The most difficult part of Macintosh programming, as veteran Macintosh programmers will attest, is careful memory management. Memory management services are fully implemented in MacApp, along with support for failure notification and a simple but elegant mechanism for recovering from failure conditions, such as a memory allocation failure in low-memory situations.

**Manages printing services.** Most Macintosh applications require some degree of printing services. Writing good printing code is difficult and demanding. MacApp makes the job of providing printing capabilities in an application easy, freeing most developers from the necessity of writing even a single line of printing code. MacApp's generalized printing model correctly manages most printing needs. It provides support for monochrome and color printing and for the print dialog boxes, and provides a default notification when the application is busy printing.

**Provides basic debugging services.** Debugging is always a chore. But MacApp eases this chore by supplying a built-in debugger that provides basic debugging services, such as a notification each time a code segment is loaded, and a built-in inspector that allows you to inspect your objects dynamically. Also, SADE 1.1, Apple's standard debugging environment and an excellent debugging tool, supports source code debugging of MacApp applications.

**Gives you high-level access to code via Mouser.** MacApp ships with a tool called Mouser that allows you to access both the MacApp libraries and your source code by class, method, and field. For details, see the sidebar "About Mouser" by Mary Boetcher, author of Mouser.

**Makes support organizations available to you.** When you program for the Macintosh, you can turn to a number of organizations for support. The MacApp Developer's Association (MADA) provides regular newsletters, source code disks, and MacApp tools for developers. Also, a large developer group address on AppleLink called MacApp.Tech$ provides quick answers to technical questions. Many of MacApp's current and former engineers appear on this group address to answer questions.

**Makes class libraries available to you.** Last but not least, if you program with MacApp, you can use existing class libraries from a variety of sources. MacApp comes bundled with five fully functional demo applications in Object Pascal and three in C++. Code can be copied and pasted from these examples into your application. MADA maintains a catalog of powerful classes that are available for purchase, such as an offscreen-imaging unit to improve graphics rendering, a database unit to integrate database capabilities into your application, and several more.

## **ABOUT MOUSER** by Mary "Mouser Woman" Boetcher

Mouser is a browser, a program for viewing and editing source code files.

The difference between a browser and an editor is that the browser "knows" something about the structure of the language and/or development system the code is written in.

Mouser knows about the structure of Object Pascal and C++ programs, and can use this information to allow you to quickly navigate among the classes and methods of a program. The leftmost pane of the browser window displays a list of the program's classes. Clicking on a class name brings up lists of the class's methods and fields. You can then click on a method or field name to see its source code.

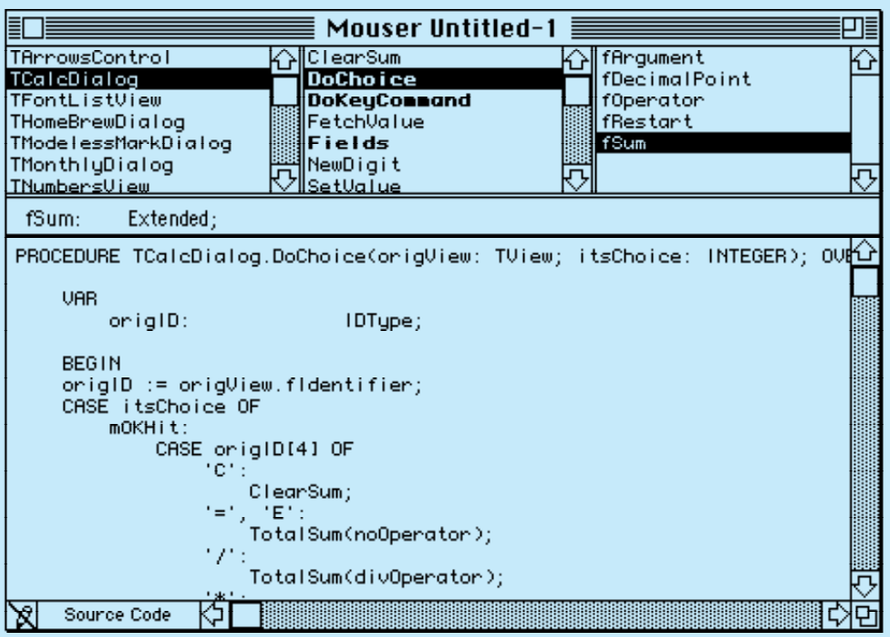



**Figure 3**
**Mouser Provides High-level Access to code**

Mouser provides a number of commands for getting information about your program.

You can get a list of

- all the methods that reference a particular string
- all the methods with a particular name
- all the fields or methods of a class, including those of its superclasses

You can also find out

- which segment a method is in
- which source file a method or class definition is in very useful for MacApp!)
- the inherited form of a method
- the source code for some selected text

## MacApp's FLEXIBILITY

What if there's something your application needs to do slightly differently from the MacApp libraries? The fact that these libraries are written in an object-based language (Object Pascal) means that you can easily modify the pieces of the libraries that don't exactly meet your needs. (If you're not familiar with object programming, and words like *object*, *class*, *subclass*, *superclass*, *method*, and *inheritance* mean nothing to you, you might want to consult the section entitled "Basic Concepts of the Object-Based Approach" in Brian Wilkerson's article "How to Design an Object-Based Application" in this issue for help in understanding the next couple of paragraphs.)

Suppose, for example, you want to add a Preferences item to the standard File menu supported by MacApp. In MacApp, the **TApplication.DoMenuCommand** method (or member function, in C++ terminology) handles the standard menu items (those creating a new document, opening a document, quitting the application, and so on). In your subclass **TMyApplication** of the MacApp class **TApplication**, you define a method that will override the inherited **DoMenuCommand** method to handle the case where the user selects Preferences from the File menu. If the item the user selects from the File menu is not Preferences, then your method simply calls the inherited version of the method so **TApplication** can handle the menu selection.

The following simple method allocates a Preferences command object if Preferences is chosen from the File menu, and otherwise calls the inherited version of the method:

```
FUNCTION TMyApplication.DoMenuCommand(aCmdNumber: CmdNumber):
TCommand; OVERRIDE;
      VAR
            aPreferencesCommand:      TPreferencesCommand;
      BEGIN
      DoMenuCommand := NIL;
      CASE aCmdNumber OF
            cPreferences:
                  BEGIN
                  New(aPreferencesCommand);
                  FailNil(aPreferencesCommand);

aPreferencesCommand.IPreferencesCommand(aCmdNumber);
                  DoMenuCommand := aPreferencesCommand;
                  END;
            OTHERWISE
                  DoMenuCommand := INHERITED
DoMenuCommand(aCmdNumber);
      END;   { CASE aCmdNumber }
      END;   { TMyApplication.DoMenuCommand }
```

Note that for this example to work, you would also have to add the Preferences item to the **cmnu** resource of the application, and override **DoSetupMenus** in **TMyApplication** to enable the menu item.



**You** can contact MADA at P.O. Box 23; Everett, WA; 98206; phone (206) 252-6946; AppleLink address MADA. To join the AppleLink group address MacApp.Tech$, contact AppleLink address MacApp.Admin. •

## WHAT IT TAKES TO USE MacApp

While MacApp will save you time and effort in the long run, you must invest time and effort up front to learn how to use it. If you are new to the Macintosh, you face two steep learning curves: first learning the Macintosh (the Toolbox, operating system, and user interface) and then learning MacApp. Learning MacApp also requires learning object programming.

But don't let this discourage you. Apple Developer University offers excellent introductory courses on the Macintosh programming environment and on MacApp. These courses make the learning process easier and provide programming labs in which you can immediately apply what you learn. Furthermore, using Mouser to browse the MacApp classes can help speed your learning. Finally, the MacApp example applications are a rich source of ideas and examples of how to implement a wide variety of features. And by the end of this article, if you read the next section carefully and try the exercise I lead you through, you will already have some familiarity with MacApp.

## NOW, AN EXAMPLE

Now that you know what MacApp can do for you, and what you must do for MacApp, let's look at an example of how you might use MacApp to develop an application that integrates previously existing routines.

Say we want to develop a database package for the Macintosh based on an application we've developed for another platform. The application has many capabilities that we can reuse (such as b-tree creation and management, graphing, searching, and sorting) and some capabilities that we should not use (such as window management and data entry screens). In this example we'll focus on reusing the graphing capability.

The source code for the graphing capability is in two files of graphics routines written in C. We've been careful in the design of our graphics routines, ensuring that they make no assumptions about their environment, such as graphics parameters or hardware attributes. For example, the routines avoid drawing and instead have an interface that describes what should be drawn. This lets the application that uses the routines determine how the drawing should occur.

We start, then, with our graphics routines. We will create a class that encapsulates the services offered by these routines. Then, you will learn step by step how to seamlessly integrate this code into a MacApp sample application, using MPW.

### START WITH YOUR ROUTINES

Our graphics routines reside in a set of two files: Graph.h, which contains the interfaces to the routines, and Graph.c, which contains the source to the graphics routines. You'll find a complete listing of these files on develop, the disc.

For this example, these files reside on the Macintosh. In your case, the files you want to use may reside on another platform. To transfer your files to the Macintosh, you should consider either a disk transfer or a file transfer. A disk transfer, to transfer the files from another disk, is best accomplished with a utility such as Apple File Exchange. A file transfer is best accomplished with either a terminal emulator, to download the file using standard file transfer protocols, or a file server, such as AppleShare, to access the other platform over AppleTalk and transfer the files.

Now I'll point out selected features of our files Graph.h and Graph.c.

The header file Graph.h contains some type and constant declarations, including the following:

```
#define kMaxPoints 20 /* Maximum number of points we support. */
```

This file also contains some type definitions, like these:

```
typedef enum {kBar, kStackedBar, kPie, kLine} GraphType; /* These are the kinds of graphs
that the graph routines support; only the bar graph is implemented for this example. */
typedef GraphValue GData[kMaxPoints-1]; /* Zero-based array of points. */
typedef struct {
  GraphType  thisGraph;    /* Type of graph it is. */
  short      numPoints;    /* Number of points in this graph. */
  short      top;
  short      left;
  short      bottom;
  short      right;        /* The graph's rectangle with respect */
                           /*to which our graph is computed. */
       short  graphYMax;   /* The graph's maximum Y coord value. */
       short  graphYMin;   /* the graph's minimum Y coord value. */
                           /* Use these to scale the graph. */
       GData  graphItems;  /* The data points in the graph. */
} GraphStruct, *GraphStructPtr;
```

Finally, Graph.h also contains some function declarations, such as:

```
GraphStructPtr DoGraphInit( GraphType whichGraphType );
void DoGraphSetGraphRect( short top, short left, short bottom, short right, GraphStructPtr
graphStorage );
void DoGraphSetPoint( short which, short value, GraphStructPtr graphStorage );
```

The actual routines are implemented in Graph.c. Here's a sample from this file:

```
GraphStructPtr DoGraphInit( GraphType whichGraphType )
{
       GraphStructPtr      graphStorage = 0;
       short               counter;
       GraphValue          aGraphValue;
```

```
        if (!(graphStorage = (GraphStructPtr) malloc(sizeof (GraphStruct))))
                return 0;                       /* Error... */
        switch ( whichGraphType ) {
                case kBar:
                        graphStorage->numPoints = graphStorage->top = graphStorage->left =
                        graphStorage->bottom = graphStorage->right =
                        graphStorage->graphYMax = graphStorage->graphYMin = 0;
                        for (counter = 0; counter   GRAPHITEMS[COUNTER];
                                aGraphValue.whichOne = aGraphValue.value =
                                aGraphValue.top = aGraphValue.left =
                                aGraphValue.right = aGraphValue.bottom = 0;
                        }
                        break;
                case kStackedBar:
                case kPie:
                case kLine:
                        /* These are unsupported in this version. */
                        break;
        }
        return graphStorage;
}
```

### CREATE A CLASS TO ENCAPSULATE SERVICES

We next encapsulate the services of our graphics routines in a C++ class. To do this requires changes to our header. We modify our header files by surrounding our function declarations with the extern "C" directive as follows:

```
#ifdef __cplusplus

extern "C" {
#endif
// Function declarations go here, for example:
GraphStructPtr DoGraphInit( short graphType );
// and so on.
#ifdef __cplusplus
}
#endif
```

This ensures that when CFront, the C++ preprocessor, reads in this header, it won't mangle the names of our C routines.

Next we create a "wrapper object" for these routines. In essence, this is a class that can be used to define objects that provide all the services of the graph routines. This class can then be used in a MacApp application. Such a class can be defined in Object Pascal or C++. In this example, we'll create a class in C++ that provides the services of the graph routines.

To create the C++ wrapper object—**TGraph**—for our graph routines, we make two new files: UGraph.h and UGraph.cp (following MacApp's naming convention). The

first file contains the class definition, and the second contains the class implementation. See develop, the disk, for a complete listing of these two files.

Creating these files is a three-step procedure, as follows:

1. After putting our copyright notices at the top of these two files, we put in UGraph.h the following basic structure:

```
#ifndef __UGRAPH__
#define __UGRAPH__
// · Auto-Include the requirements for this unit's interface.
#ifndef __UMacApp__
#include "UMacApp.h"
#endif
#include "Graph.h"
// The interface to this class goes here.
#endif
```

   This allows the MPW C++ compiler to perform at its best by only making it do the work to include this unit's interface (and the requirements for this unit's interface) when it's not already included.

2. Next, we create the definition for **`TGraph`** and put the definition in UGraph.h.

   To do this, we must choose which class **`TGraph`** will descend from. Since graphs are things that are drawn on the screen and are viewed, we decide to make the graph class descend from MacApp's **`TView`** class. Ideally, we would create a generalized base class for a graph, such as **`TGraph`** descended from **`TView`**, and then create specialized subclasses of **`TGraph`** for the various kinds of graphs. A bar graph—**`TBarGraph`**—would descend from **`TGraph`**; a line chart—**`TLineGraph`**—would descend from **`TGraph`**; and so on. However, to keep this example simple, we'll make the bar graph class descend directly from **`TView`**.

Here's the class definition we come up with:

```
class TGraph : public TView {
 public:
    virtual pascal void IRes(TDocument *itsDocument, TView *itsSuperView,
                             Ptr *itsParams);
    // Initialize the graph view from its resource template.
    virtual pascal void SetGraphRect(Rect graphRect);
    // Initialize the graph data structure to be the size of this view.
    virtual pascal void SetPoint( short which, long value );
    // Set a point to a value.
    virtual pascal short GetNumPoints();
    // Return the number of points in the graph.
    virtual pascal void ComputeBars(Boolean redraw);
    // The graph library computes each of the bars for this graph
    // and if redraw is TRUE forces the view to redraw itself.
    virtual pascal void GetCoordinateRange(Rect *coordRange);
    // Return min & max Y coordinates, and min & max X coordinates,
    // useful for labeling the axes of the graph.
    virtual pascal void Draw(Rect *area);
    // Draw the graph.

    virtual pascal void Free(void);
    // Free the data allocated by this class.
 private:
    GraphStructPtr      fData;
};
```

There are several things to note about this wrapper object.

First, note that the class functions don't map one-to-one with the graph routines. Rather, there is an attempt to abstract from the routines various services available for this class. For example, instead of retrieving the maximum value of a point on the Y-axis with a call to the routine **`DoGraphGetYMax`**, we abstract from this routine the notion of retrieving the range of values on both axes (useful in setting up labels on the axes), and implement the class member function **`GetCoordinateRange`**, which retrieves the range of values on the X- and Y-axes and returns the result in a **`Rect`**.

Also note that instead of retrieving a specific bar by calling the routine **`DoGraphGetBar`**, we attempt to hide that activity behind the class member function **`Draw`**, which simply draws the graph, iterating over all bars in the bar graph.

And note that since this class descends directly from **TView**, three member functions in this class definition override **TView's** member functions: **IRes**, which initializes the view and calls the graph routine to allocate and initialize the graph data structure; **Draw**, which does the work of drawing the graph; and **Free**, which calls the graph routines to dispose of the graph data structure.

3. Finally, we create the file UGraph.cp, which contains the implementation of the class **TGraph** in C++. The first thing to add here (after the copyright notice) is an #include so that the header file is included:

```
#ifndef __UGRAPH__
#include "UGraph.h"
#endif
```

This ensures that the **TGraph** implementation "sees" its class definition, as well as any other necessary definitions. We then list the implementation of the **TGraph** class in the body of the file UGraph.cp. The **TGraph::IRes** member function implemented in this file might look something like this:

```
pascal void

TGraph::IRes(TDocument *itsDocument, TView *itsSuperView, Ptr
*itsParams)

{
  GraphStructPtr aGraphStructPtr;
  Rect aRect;

  inherited::IRes(itsDocument, itsSuperView, itsParams);
  aGraphStructPtr = DoGraphInit(kBar);
  fData = aGraphStructPtr;
  aRect = gZeroRect;
  if (Focus())
    GetQDExtent(&aRect);
  SetGraphRect(aRect);
}
```

This member function initializes the **TView** object by calling its inherited **IRes** member function and then initializes the graph routines by calling **DoGraphInit.** This view object then attempts to get information about its graphics environment and lets the routines set up various values for this environment.

### INTEGRATE THE CLASS INTO AN APPLICATION

At this point, we have a C++ class that encapsulates the services offered by our graphics routines. To see how you can use this C++ class in a MacApp application, try the following exercise. In this exercise, you'll modify the C++ version of the DemoDialogs sample application that comes with MacApp 2.0, by adding a bar graph to the Monthly Values Dialog. All the files you need are on develop, the disc.

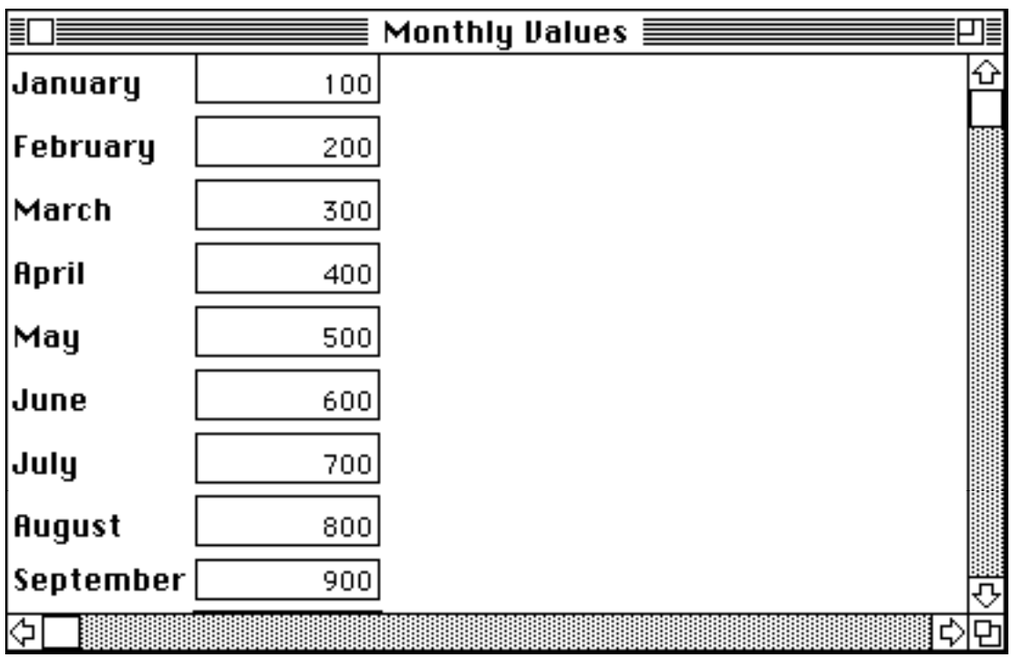



**Figure 4.**
**The Monthly Values Dialog Before Modification**

1. Put the following files in the DemoDialogs folder: Graph.h, Graph.c, UGraph.h, UGraph.cp. Use MPW to set this folder as the current directory.

2. Open the file DemoDialogs.r and locate the Monthly Values Dialog resource description:

```
resource 'view' (cMonthlyDialog, purgeable) {
  {
  ...
  }
};
```

3. Increase the width of the **DialogView** from **500** to **600** as follows:

```
'SCLR', 'DLOG',     { 0, 0 }, { 1000, 600 },
```

4. Add to the end of this resource description (that is, inside the second-to-last right curly brace) the following description:

```
        ;
        'DLOG', 'graf', { 25, 300 }, { 300, 300 },
                sizeFixed, sizeFixed, shown, disabled,
        View { "TGraph" }
```

   This puts a **TGraph** view in the Monthly Values Dialog and ensures that the **TGraph** object is allocated and initialized via its **IRes** member function when the dialog is created.

5. To make sure that the DemoDialogs headers know about your graph routines, add the following line at the end of the list of **#include** files at the top of the file UDemoDialogs.h:

```
#include "UGraph.h"
```

6. Since your graph view is a subview of the Monthly Values Dialog, modify that class so that the graph recomputes and redraws whenever a new monthly value is typed in. To do this, modify the file UDemoDialogs.h by adding the following to the **TMonthlyDialog** class:

```
virtual pascal Boolean DeselectCurrentEditText(void);
```

7. Then add the implementation for this member function to the file UDemoDialogs.cp:

```
pascal Boolean

TMonthlyDialog::DeselectCurrentEditText(void)

{
     TGraph        *aGraph;
     TNumberText   *theNumberText;

     aGraph = (TGraph *) FindSubView('graf');
     for (short which = 0; which   fIdentifier == gMonthIDs[which]) {
                  theNumberText = (TNumberText *) FindSubView(gMonthIDs[which]);
                  if (theNumberText)
                         aGraph->SetPoint(which+1, theNumberText->GetValue());
                  aGraph->ComputeBars(kRedraw);
                  break;
            }
     }
     return inherited::DeselectCurrentEditText();
}
```

8. Next, to prevent the linker from stripping the **TGraph** class, modify **TTestApplication::ITestApplication** to include the following variable:

```
TGraph *aGraph;
```

   and to allocate this variable within the **gDeadStripSuppression** section at the end of this function:

```
if (gDeadStripSuppression) {
  ...
  aGraph = new TGraph;
}
```

9. Then, so that the graph points are set up as the Monthly Values are set up, modify **TMonthlyDialog::StuffValues** as follows:

```
pascal void
TMonthlyDialog::StuffValues()

{
  TGraph *aGraph;
  TNumberText *aNumberText;

  aGraph = (TGraph *) FindSubView('graf');
  for (short i = 0; i
  SetValue(gMonthlyValues[i], kDontRedraw);
    aGraph->SetPoint(i+1, gMonthlyValues[i]);
  }
  aGraph->ComputeBars(kDontRedraw);
}
```

10. Finally, so that the graph unit and the file of graph routines are built when this example is built, add the following lines to the end of the file DemoDialogs MAMake:

```
OtherLinkFiles = ∂
                                    "{CLibraries}"StdCLib.o ∂
                                    "{ObjApp}UGraph.cp.o" ∂
                                    "{ObjApp}Graph.c.o"
"{ObjApp}Graph.c.o" ƒ ∂
                                    "{SrcApp}Graph.c" ∂
                                    "{SrcApp}Graph.h"
  {MAEcho} {EchoOptions} "Compiling:   Graph.c"
  {MAC} "{SrcApp}Graph.c" ∂
              -i "{SrcApp}" ∂
              -i "{CIncludes}" ∂
              -i "{MACIncludes}" ∂
              -o "{ObjApp}Graph.c.o" ∂
              {COptions} ∂
              {OtherCOptions}
"{ObjApp}UGraph.cp.o" ƒ ∂
                                    "{SrcApp}UGraph.h" ∂
                                    "{SrcApp}Graph.h" ∂
                                    {MacAppIntf}
```

11. Compile DemoDialogs and run it. You will see your graph class at work calling your graph routines and drawing the graph in the Monthly Values Dialog.

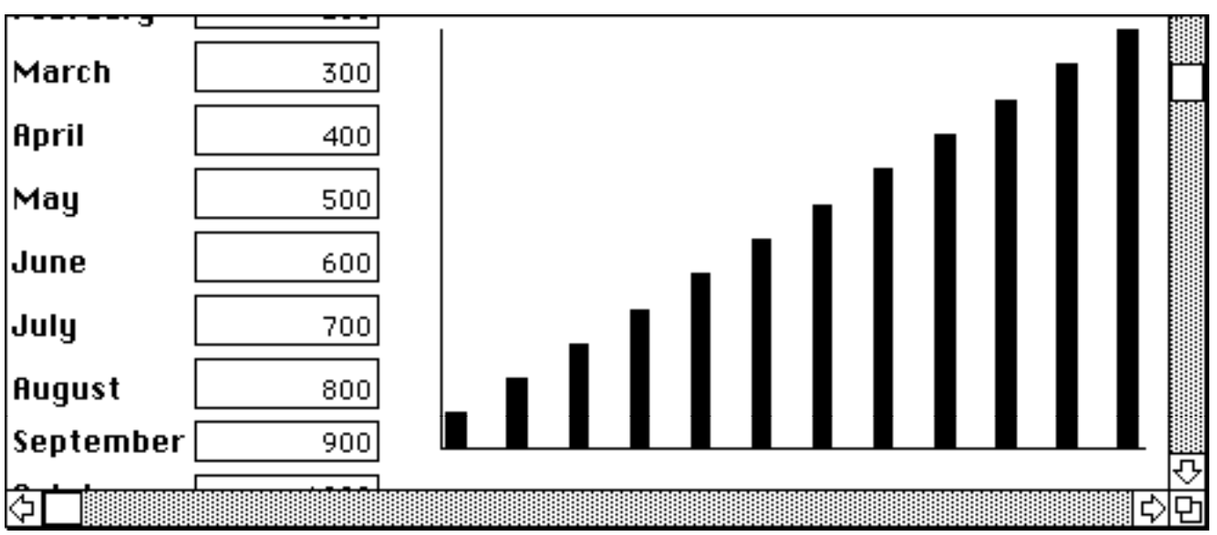



**Figure 5. The Monthly Values Dialog After Modification**

## SUMMARY

You've learned that using MacApp as your framework in developing a Macintosh application not only enables you to reuse your functionally specific code routines, but also saves you time and effort by providing standard application behavior. You've also learned that because the MacApp libraries are written in an object-based language (Object Pascal), you can easily modify the pieces that don't meet your needs. You've watched as we've created a class to encapsulate the functionality of an existing group of routines, and you've gone through the process of integrating this class into a MacApp sample application. Now you're ready to integrate some of your own code into the MacApp sample application on develop, the disc. With the help of MacApp, you'll soon be reusing your code in Macintosh applications that present users with the interface they know and love.

# DEVELOP, THE DISC: FEATURED ARTISTS

*We're professionals. We're entrepreneurs. We quit our real jobs. We have our own company. It's called Cyan.*

*Cyan.*

**RAND & ROBYN MILLER**

What a cool name. We thought of that name ourselves. It doesn't really mean anything to us personally, but man, is that a cool name or what?

We also have Macs. Our Macs are cool too. We do things on our Macs we used to only dream about. We do graphics; we do animation; we do code; we do music. Some people call those things multimedia, and we'll call it that too if it makes us sound more professional.

Cyan actually started out by creating some Desktop Worlds (hey, we coined that one). The first one was the Manhole, and after that came Cosmic Osmo. We decided to make Cosmic Osmo bigger and better, and we ended up with something so big that we could either fit it on 179 floppies or on one CD-ROM. We picked the CD-ROM.

So that leads to our point. This CD-ROM has got a few of the songs we did for our CD-ROM version of Cosmic Osmo. We used our Macs. We also used some awesome equipment and software to help our Macs get the job done. First we discovered MIDI. We composed and sequenced the songs using Master Tracks Pro. We used a Yamaha Clavinova as our MIDI keyboard, and got a whole slew of incredible sounds out of a little MIDI module called th Roland U-110. We completed all the songs, stuck 'em on a diskette and headed for Hidden Forest Studios in Longview, Texas (in the legendary Pine-woods area of deep East Texas).

Now it just so happens that Hidden Forest Studios has Macs too. So we stuck our diskette into their Macs, and suddenly we had a whole studio at our disposal. We clicked here and double-clicked there and moved sounds from one module to another until things sounded tight. With the addition of some live instruments our studio work was complete.



Dancing Beetle
(© 1989 by Cyan)
written and sequenced by Robyn Miller
saxophone: Kyle Stroud
piano solo: Gary Boren

Cosmic Osmo Blues (Rock Version)
(© 1989 by Shep Lovick, Robyn Miller, Rand Miller)
written and sequenced by Shep Lovick
saxophone: Kyle Stroud
horn arrangement and horn synths: Gary Boren

Frozen Jam
(© 1989 by Cyan)
written and sequenced by Robyn Miller
saxophone: Kyle Stroud

all songs produced by Gary Boren and Robyn Miller

We sampled the completed music into our Cosmis Osmo stack using Farallon's MacRecorder and coded things appropriately. Now, when you tune the radio that you just built on the fourth floor of the science planet, you might just hear the flighty rifts of Dancing Beetle. Or when you climb to the highest tower of Queen Osmorella's castle on the urban planet you'll hear the echoing chords of Tower Mist. In addition, the songs are included on audio tracks of the Cosmic Osmo CD-ROM (so you can crank 'em out on your mega-stereo in the den).

Cyan is Robyn Miller and Rand Miller (we're brothers). Cyan is located in Spokane, Washington (capitol of the legendary Inland Empired of the Pacific Northwest). AppleLink us any time at CYAN.

## Macintosh Q&A

**Q**

*How do I get the new GWorld calls into my life and my applications? Can I use them on a Macintosh Plus? Where are they documented?*

**A**

The new GWorld routines are available only under 32 QD, this in itself implies Color QuickDraw Macintoshes (Macintosh II and such) since 32 QD can only run when Color QuickDraw is present; at the moment the calls are documented in the 32-Bit QuickDraw release notes which are available through APDA. A future release of Inside Macintosh will surely document the new set of calls.

If you want to implement offscreen BitMaps and would like to have access to offscreen routines for non color Macintoshes, you may want to check the sample program OffScreen from the Developer Technical Support Sample Collection. OffScreen exemplifies the use of the OffScreen Unit which implements the same type of calls for QuickDraw machines (Macintosh Plus, SE and Portable) and Color QuickDraw Macintoshes not running

with 32 QD installed.

**Q**

*Dave, I read your article on compatibility and gotta say it carries the most real info with the least frosting of any System 7.0 article I've seen. I have a couple of questions related to files and low memory globals:*

*1) On p.57 you say "use SFGetFile and SFPutFile" I assume that includes SFPGetFile and SFPPutFile.*

*2) On p.67 you say "avoid reliance on low-memory globals." I'm using a SFPGetFile to specify a directory instead of a file and am faking it out by sending a cancel (thru my dialog hook) when my Use Directory button is hit. Since the SFReply.vRefNum is invalid for a cancel, I'm using the SFSaveDisk (and perhaps the CurDiskStore for something else later) low-memory global. Are these safe? If not, is there any better way to do this to avoid using those globals?*

**A**

1) True.

2) If use of a low memory global is unavoidable, well, it's unavoidable. SFSaveDisk is a good example. But if alternatives exist, you should use them. Apple is not going to eliminate low memory globals overnight (that would kill just about every application out there), but the day may come when you will no longer be able to access them directly. (Why might this have to change? Well, one problem with shared low memory is it makes fully protected address spaces for applications a very difficult proposition.)

**Q**

*How can I allocate offscreen pixmaps in MultiFinder memory?*

## A

Version 1.2 of 32-Bit QuickDraw (which is available on the System 6.0.5 disks) has been updated to let you allocate offscreen pixmaps in MultiFinder memory.

## Q

*How can I tell which directory my application is in?*

## A

GetVol returns the default volume. When an application is started, the default volume is set to the directory that contains the application. If the application calls GetVol before changing the default volume, it will have the directory for the folder containing the application.

## Q

*How do I find the size of a volume?*

## A

To find the size of a volume, call PBHGetVInfo. The size is the product of the number of allocation blocks and the allocation block size. (ioVNmAlBlks * ioVAlBlkSiz) It is important to remember - especially in Pascal - that ioVNmAlBlks is an UNSIGNED quantity. If you don't take that into account, you often get negative results.

See page IV-130 and IV-123-124 of Inside Macintosh for more details.

## Q

*I'm writing an application where I'd like to keep track of file locations after a reboot. What should I use in addition to the file name?*

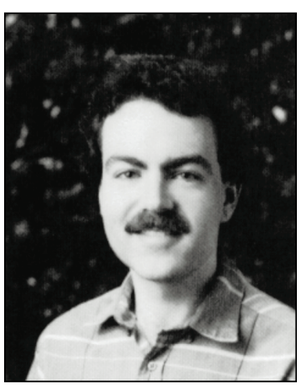
## A

You need to save the name of the volume, the DirID, and the name of the file. Don't save the vrefnum from SFGetFile. The vrefnum that you get from SFGetFile is actually (under HFS) a working directory reference number, and will change when the working directory is closed (e.g. when you turn your machine off).

You use PBGetWDInfo (Inside Macintosh, page IV-159) to get the volume name (in ioNamePtr) and DirId (in ioWDDirID). To get the ioVRefNum given the name of the volume, the DirID, and the name of the file, use PBOpenWD (Inside Macintosh, page IV-158).

# HOW TO DESIGN AN OBJECT-BASED APPLICATION

*The object-based approach promises to make software easier to reuse, refine, test, maintain, and extend. But simply implementing an application in an object-based language does not guarantee these benefits. They can only be achieved if the implementation is based on a sound object-based design. This article presents a process for creating such a design.*



**BRIAN WILKERSON**

As every programmer knows, software applications are becoming increasingly complex, and as a result, increasingly expensive to build and maintain. The good news is that if you are willing to spend the time to carefully develop an object-based design for your software, implementation can proceed smoothly and quickly, and the resulting software will be relatively easy to reuse, refine, test, maintain, and extend. This article gives an overview of the object-based approach and then describes step by step a two-phase process for designing an object-based application.

## BASIC CONCEPTS OF THE OBJECT-BASED APPROACH

Programmers familiar with non-object-based languages are used to dividing information into two distinct kinds: functions and data. Procedural programming, based on this division of information, focuses on *how* to accomplish the goals of the program. It begins by identifying the high-level tasks that need to be performed, and then decomposing each task into smaller tasks until the level of the language statement is reached. Procedural programming concerns itself almost immediately with the implementation of the program: the steps that compose each function, and the particulars of the data to be operated upon.

By contrast, the focus of the object-based approach is more abstract. It asks first about the intention of the program: asking *what*, not *how*. It views the programming process as one of modeling the world. It begins by identifying the things inhabiting



**BRIAN WILKERSON** is an object-oriented systems specialist for Instantiations, Inc., a consulting firm in Portland, Oregon. He studied computer science at the University of Alberta. After receiving his degree, he worked for Tektronix prior to joining his current company. Brian has developed a course about object-oriented design at Instantiations, and has also co-authored a book entitled Designing Object-Oriented Software, to be published this spring by Prentice-Hall. When he's not writing or working, he enjoys day hiking and attending jazz concerts. •

the part of the world being modeled, and the behavior of those things, both as individuals and with respect to the other things in the world.

The object-based approach uses abstraction to manage the complexity inherent in real-world problems. An abstraction is a simplified picture of the world, arrived at by generalizing about details. The object-based approach relies on abstraction mechanisms such as encapsulation, information hiding, polymorphism, and inheritance.

### ENCAPSULATION

Encapsulation is the enclosing of a number of separate related things within a single physical or conceptual capsule. For example, a telephone number encapsulates individual digits at a higher, abstract level at which the numbers form a single entity. When you think of your telephone number, you don't think of it as seven separate digits. You think of it as a single unit that happens, almost incidentally, to be composed of seven digits.

### INFORMATION HIDING

Encapsulation makes complexity more manageable, but it doesn't reduce the amount of visible detail. Information hiding takes encapsulation a step further, reducing complexity by hiding some or all of the things that have been encapsulated. For example, when you use a compact disc player, you don't generally think of all the electrical and mechanical components within it. You don't need to know how it works. What's important is what it does: it plays the music you want to hear.

### OBJECTS

An object is an encapsulation of data and the functions that manipulate that data. But more than that, an object hides the data and possibly some of the functions, revealing only those functions that need to be made available to other objects. The set of visible functions defined by an object is referred to as the behavior of the object.

The data and functions that are hidden within an object define the implementation of that object. That is, they define *how* that object does what it does. The behavior of an object defines *what* the object does. In keeping with the abstract nature of the object-based approach, object-based design focuses exclusively on the behavior of objects.

The object-based approach views a program as a collection of objects that interact with other objects to accomplish the goals of the program. Objects interact with other objects by sending those objects messages requesting that a publicly visible function be executed. A message specifies the name of the function being requested and the arguments required by the receiver of the message to execute the function.

### POLYMORPHISM

Polymorphism is an abstraction mechanism by which two or more different kinds of objects can respond to the same message, each in its own way. This means that an object can send a message to another without knowing how the receiver will respond, or what other messages the receiver might also understand. The sending object just needs to know that many different kinds of objects can be defined to respond to the message being sent and that the receiver is one of those.

### CLASSES

A class is a specification of the behavior of an arbitrary number of similar objects. Objects that share the same behavior are said to belong to the same class. The objects that belong to a class are referred to as instances of that class. The process of dynamically creating objects is known as instantiating a class.

Classes are another abstraction mechanism. They allow us to focus on the kinds of objects in an application rather than on the individual objects.

Throughout the remainder of this article, when we refer to some aspect of a class, we mean the definition of that aspect of the instances of the class. For example, when we refer to the behavior of a class, we mean the definition of the behavior of the instances of that class. The meaning should be clear from context.

### INHERITANCE

Inheritance is an abstraction mechanism by which new classes can be derived from existing ones, thereby "inheriting" both data and functions. The inheritor (called a subclass) reuses the code that it inherits from its superclass. Again, in the design phase, we are only interested in the inheritance of behavior.

### THE CLIENT-SERVER MODEL

The model we use for our object-based design views the world as a system of objects collaborating to perform the work required of them: the client-server model.

The client-server model is a description of the interaction between two entities: the client and the server. A client makes requests of the server to perform services. A server provides a set of services upon request.

The ways in which the client can interact with the server are described by a contract: a description of the requests that can be made of the server by the client. Both must fulfill the contract: the client by making only those requests it specifies, and the server by correctly responding to those requests.

In an object-based design, both client and server are objects. Any object can act as either a client or a server at any given time. The design focuses on the contract between clients and servers by asking (1) what actions each object is responsible for, and (2) what information each object shares.

## THE BENEFITS OF OBJECT-BASED DESIGN

If you spend a meaningful amount of time on carefully developing an object-based design for your software, implementation can proceed more smoothly and quickly than it would for a traditional procedural program. The resulting software can also be easier to test, maintain, refine, and extend.

An object-based design can improve implementation by encapsulating pieces of the program into components that can be implemented without considering the interactions with the rest of the system. If an interface between components then seems wrong for some reason, the system can be changed at just that one point; other parts of the system are not affected.

A careful design can also make it easier to test the application. Classes can be isolated and tested one at a time. An error can more easily be traced to a specific class. Classes can be shown to function before being plugged into the rest of the system.

Similarly, the rigorous specification of the interfaces between classes allows testers to more easily spot discrepancies between the output of one component and the input required to another. Such a careful specification of the interfaces requires a complete understanding of the responsibilities of each component. Holes in the system—places where a responsibility was omitted by the specification, or stated ambiguously, or made part of the wrong class—can more easily be spotted and filled.

After the application has been implemented, it's also easier to maintain. Encapsulation and information hiding rigidly constrain the patterns of communication within the application, so that they can be understood more easily. This makes it easier to determine where a problem lies and where any ramifications may appear after you fix the problem. In this way, you can guard against the notorious problem of one bug fix introducing other bugs.

A system that can be understood can also be refined and extended. If the interfaces between classes have been rigorously controlled, new portions of the system can be created to use the same interfaces, but to do different things with them. You can also add new classes that respond to old requests in ways appropriate to the new system of which they are now a part. Functionality can thereby be increased at far less cost.

In sum, object-based design enables us to build classes that can be depended upon to behave in certain ways, and to know what state results from that behavior. Such classes can be reused in every application that can make use of this behavior and knowledge. With careful thought, you can construct classes that will be useful to many applications.

## A TWO-PHASE PROCESS FOR DESIGNING AN APPLICATION

The remainder of this article describes a process for creating object-based designs. The result of this process, an object-based design, consists of a structure of classes modeling the problem, a description of the public behavior of those classes—their responsibilities, and a description of the patterns of communication among the classes.

The design process we use has two phases:

1. An initial exploration of the possibilities, which produces a preliminary design.
2. A rigorous analysis of the preliminary design.

Both of these phases play critical roles in the object-based design process.

The exploratory phase of object-based design concentrates on identifying the classes, assigning responsibilities to those classes, and determining which other classes collaborate with them to fulfill those responsibilities. At this stage of the engineering process, very little effort has been invested in any specific design. It is therefore relatively cheap to play with the possibilities, trying out various ways to configure your system. A little time spent exploring at this point can lead to a lot of time and effort saved later, as it will be easier to reuse parts of the design, or to refine and extend it.

The results of exploration, however, must be carefully pruned and edited. No one can count on getting it right the first time. The preliminary design must be critically examined, to maximize both encapsulation and inheritance. Only in this way can the use of object-based design fulfill its promise of producing software that is easy to reuse, refine, test, maintain, and extend.

To illustrate the design process described here, we'll use the example of a spreadsheet program, the specification for which appears in the sidebar. This example is too simple to be a true application, but for the purposes of this article it will give you a feeling for how to use the design process.

## THE SPREADSHEET SPECIFICATION

The spreadsheet program is an application that allows users to create and edit electronic spreadsheets.

Users can create new spreadsheets. Existing spreadsheets can be opened from and saved to files.

### The Spreadsheet

A spreadsheet is a collection of cells arranged in rows and columns.

Rows and columns consist of cells and have names. The name of a column is the letter C followed by the ordinal position of that column. The name of a row is the letter R followed by the ordinal position of that row.

Each cell has a name and a value. The name of a cell is the concatenation of the name of the cell's column and the name of the cell's row, in either order.

There are three different types of cells: numeric, text, and expression.

Numeric cells contain numeric values. The user can specify the format in which the value of a numeric cell is displayed. There are four different formats:

- integer (no decimal point)
- monetary (two places after the decimal point, preceded by a dollar sign)
- real (one or more places after the decimal point)
- scientific (as a value between zero and one, and an exponent)

Text cells contain arbitrary text, except that the first character cannot be an equal sign. The text can be formatted to be left aligned, centered, right aligned, or fully justified.

Expression cells contain a formula, but display the value of the formula. Formulas are entered as text, using the syntax defined by the following syntax definition:

```
<formula> ::= '=' <expression>
<expression> ::= [<expression> ',']
<simple expression>
<simple expression> ::= [<simple
expression> <additive operator>] <term>
<term> ::= [<term> <multiplicative
operator>] <factor>
<factor> ::= <constant> | <cell name> |
'(' <expression> ')'
<constant> ::= <number> | <text>
<additive operator> ::= '+' | '-'
<multiplicative operator> ::= '*' | '/'
```

Arguments to additive and multiplicative operators must be numeric. The result is a number. Arguments to the comma operator (text concatenation) may be either text or numbers, the numbers being converted to a textual representation in the latter case. The result is text.

The value of an expression cell can be formatted either as numeric cells or text cells, depending on the type of the result.

### Operations

Users must be able to select rectangular groups of cells, from individual cells to the entire spreadsheet, including rows and columns. Selected cells can be cut, copied, and pasted. At least one cell must be selected at all times.

If one or more complete rows or columns are selected and cut, the rows or columns are removed from the spreadsheet. If one or more rows or columns are pasted, they are inserted to the left of or above the topmost selected row or column, respectively.

If a portion of some rows and columns is cut, the values in those cells are removed, but the empty cells remain. If such a portion is pasted, the values of the same shape of cells are replaced with the values of the cells, with the upper leftmost cell in the paste buffer being aligned with the upper leftmost cell of the selected cells.

Users must have the ability to edit the values in individual cells, and to force recomputation of the values shown in expression cells.

## THE EXPLORATORY PHASE

The exploratory phase of object-based design consists of three steps:

1. Finding the classes.
2. Assigning responsibilities to them.
3. Determining the collaborations required to fulfill those responsibilities.

Let's look at each of these steps individually.

### FINDING CLASSES

Choosing the classes of objects that make up your application is a key part of modeling it. The classes should define the essence of your application; they should emphasize the important aspects, and discard irrelevancies.

**Generate candidate classes.** When you start your design, you frequently have nothing more than a specification outlining the functionality envisioned for the system as a whole. If that's all you have, that's what you start with. Begin by reading the specification until you are familiar with it. Now reread the specification, taking note of every noun or noun phrase in the document. These are your candidate classes.

The following list results from doing this with the spreadsheet specification:

spreadsheet program
application
user
electronic spreadsheet
new spreadsheet
existing spreadsheet
files
spreadsheet
cell
row
column
name
ordinal position
value
numeric cell
numeric value
format
integer format
decimal point
monetary format
dollar sign
real format
scientific format
zero
one
exponent
text cell
arbitrary text
first character
equal sign
text
left-aligned text
centered text
right-aligned text
justified text
expression cell
formula
expression
simple expression
additive operator
term
multiplicative operator
factor
constant
cell name
number
argument
comma operator
textual representation
type
rectangular group of cells
individual cell
entire spreadsheet
selected cell

**Choose classes from candidates.** Once you have a list of possible classes, you must decide which of them will become part of the model you are designing. The following guidelines are useful in choosing which noun phrases represent classes and which are spurious.

- Model physical objects, such as windows on the display or printers on the network. The cells of a spreadsheet can be thought of in this way, so we tentatively create the class Cell.
- If more than one word is used for the same concept, choose the one that is most meaningful in terms of the rest of the system.

  For example, "application" really means "the spreadsheet program" in this context. The phrase that best describes the meaning is kept, while the rest are discarded. In some cases, none of the phases is appropriate, so a new one must be created.

  Our list is full of such synonyms and naming problems. Following is a list of the words with synonyms. The words that remain candidates are followed by the synonyms we have rejected, indented below them.

  spreadsheet
  - electronic spreadsheet
  - new spreadsheet
  - existing spreadsheet
  - entire spreadsheet

  cell
  - numeric cell
  - text cell
  - expression cell

  name
  - cell name

  value
  - numeric value

  text
  - arbitrary text
  - textual representation

  cell group
  - rectangular group of cells
  - individual cell
  - selected cells

  In addition, one more naming problem exists. Rows and columns are just rectangular groups of cells. The general concept capturing this commonality is "cell group."

- Be wary of adjective-noun phrases. An adjective-noun combination can mean a different kind of object, a different use of the same object, or it could be utterly irrelevant. Ask if the object represented by the noun behaves differently when the adjective is applied to it. If the use of the adjective signals that the behavior of the object is different, then make a new class.
- Model categories of objects. Such categories represent abstract superclasses, and should therefore be modeled. "Expression" and "value" are examples of abstract superclasses; there are several different kinds of each in the spreadsheet.

- Model known interfaces to the outside world, such as physical devices, a windowing system, or the operating system, as fully as your initial understanding allows. The interface to the outside world in this case is represented by the noun "file," which becomes a candidate class

.

Don't model things outside the application. Our list includes a variety of things obviously outside the bounds of the system, such as "user," "first character," and "type."

- Model the values of attributes of objects, but not the attributes themselves. For example, each cell has a name. The name is an attribute whose value is a string of characters. Therefore, the class String should be created, but there will be no class called Name; it will be an attribute of the class Cell. The following table contains the phrases representing attributes, the class(es) of objects having that attribute, and the class(es) of the value of the attribute.

| **Attribute Phrase** | **Defining Classes** | **Value Classes** |
|---|---|---|
| name | Cell | String |
| value | Cell | Number, Text, Expression |
| format | Cell | ? |
| decimal point | Number | none |
| exponent | Real Format | none |
| argument | Expression | Expression |

How should formats be represented? The format controls the way in which the value is displayed. This kind of control is usually handled by either sending different messages to a class, or sending the same message to different classes. The former is preferred in this case because the format can change independent of the value. The format, therefore, should be the message with which the value is displayed (or an encoding of it if the target language does not support messages as objects). We can therefore discard all of the noun phrases representing types of formats, which were the following:

| | | |
|---|---|---|
| integer format | scientific format | right-aligned text |
| monetary format | left-aligned text | justified text |
| real format | centered text | |

We replace the phrases describing the syntactic representation of expressions, given below, with classes representing the semantic structure of expressions.

| | | |
|---|---|---|
| formula | simple expression | multiplicative operator |
| expression | additive operator | factor |
| comma operator | term | constant |

**Record classes.** When you have identified the first, tentative list of classes, they need to be recorded. For each class, take a 4" x 6" index card, and write the class name at the top of the card. You should use index cards to record classes because they are compact, easy to manipulate, and easy to modify or discard. Each index card will eventually contain the kinds of information indicated in Figure 1.

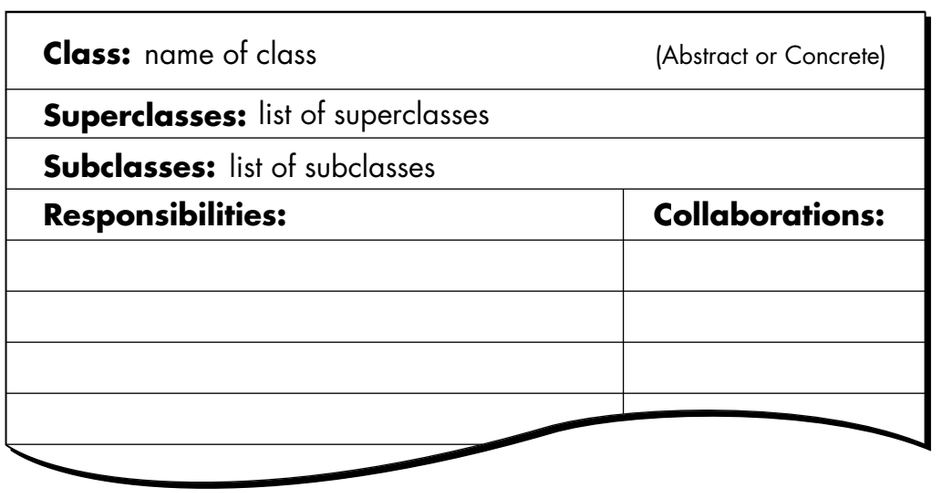



**Figure 1.**
**Contents of Each Index Card**

Some classes will be missing and others will be eliminated later, but don't worry. Your design will go through many stages on its way to completion, and you will have ample opportunity to revise.

The final list of candidate classes is as follows:

| | | |
|---|---|---|
| Binary Expression | Computed Value | Spreadsheet |
| Cell | Constant Expression | Textual Value |
| Cell Group | Expression | Value |
| Cell Reference | File | |
| Expression | Numeric Value | |

### ASSIGNING RESPONSIBILITIES

You have now found the classes in your system. Next you must decide what behavior each of them is going to be responsible for.

Responsibilities include two key items:

- the information other classes can ask for from a class
- the actions a class can perform

Responsibilities are meant to convey a sense of the purpose of a class and its place in the system. As you seek to identify responsibilities, use the conceptual model of the client-server contract. The contract between two classes represents the list of services one class can request of another. A service can be either the performance of some action or the return of some information. If an object provides a service, that is one of its responsibilities. All of the services listed in a particular contract are the responsibilities of the server for that contract.

**Find responsibilities.** To find responsibilities, return to the specification. This time, take note of all the verbs. Use your judgment to determine if each represents an action that some class within the system must perform. Also use the work you just performed when you identified classes. The fact that you identified a class indicates that you saw a need for it to fulfill at least one responsibility. The name you chose for that class probably suggests that responsibility, and possibly others.
From the specification, we can derive the following candidate responsibilities:

| | | |
|---|---|---|
| open from a file | columns have cells | remove columns |
| save to a file | specify cell format | insert rows |
| maintain a collection of cells | numbers convert to text | insert columns |
| cell have names | users select rectangular groups of cells | remove values of cells |
| cell have values | cut selected cells | replace values of cells |
| rows have names | copy selected cells | edit values of cells |
| rows have cells | paste cells | recompute values of cells |
| columns have names | remove rows | |

**Assign responsibilities to classes.** Once you have listed a number of candidate responsibilities for the classes in your application, you can go about assigning each responsibility to the appropriate class.
The following guidelines can prove useful as you seek to apportion the responsibilities to each class.

- Distribute system intelligence as evenly as possible. A system can be thought of as having a certain amount of intelligence, such intelligence being what the system knows and what actions it can perform. Within any system, some classes of objects can be viewed as being relatively "smart," while others seem less so. Distributing the intelligence embodied within your system among a variety of classes allows each class to know about relatively fewer things, thus producing a more flexible system, and one that is easier to modify.
- Keep behavior with related information, if any. If a class is responsible for knowing certain information, it is logical also to assign it the responsibility of performing any operations necessary upon that information. Conversely, if a class requires certain information in order to perform some operation for which it is responsible, it is logical (other things being equal) to assign it the responsibility for maintaining the information as well.
- Keep information about one thing in one place. In general, the responsibility for knowing specific information should not be shared. Sharing information implies a duplication that could lead to inconsistency.
- Share responsibilities among related objects. Occasionally, you may discover that a certain responsibility seems to be several responsibilities, or a compound responsibility, that is best divided or shared among two or more classes.

We now assign the responsibilities from the spreadsheet program.

open from a file
save to a file
It isn't clear which class to assign these responsibilities to. The closest candidate we have now is Spreadsheet itself, but it should be used to represent just the spreadsheet, not the full set of editing capabilities implied by the system. We therefore want a class that represents the application itself. Let's call this class Spreadsheet Editor, and assign these responsibilities to it.

maintain a collection of cells
This is clearly a responsibility of the class Spreadsheet. After all, that is the class that must maintain the collection; therefore, it should have the responsibility for maintaining the information as well.

cells have names
cells have values
By the same token, the responsibility for maintaining this information belongs to the class Cell.

rows have cells
columns have cells
The responsibility for maintaining this information belongs to the class Cell Group. However, because rows and columns are merely different groupings of cells, we might wish to rephrase this responsibility more generally, stating that the class Cell Group knows which cells it contains.

At this point, we might notice that the responsibilities of Spreadsheet and the responsibilities of Cell Group are very similar; they both maintain information about the cells they contain. We should view Spreadsheet as being composed of a group of cells, rather than maintaining a collection of individual cells. The responsibility of the class Spreadsheet is to know the *group* of cells of which it is composed. The responsibility of the clas Cell Group need not change.

rows have names
columns have names
Names of rows and columns appear merely by way of explaining how cells get named. Cells must maintain their names, as we mentioned above, but row and column names are irrelevant, and do not need to be maintained by any class. There is no responsibility for maintaining this information.

specify cell forma
This is actually a compound responsibility. The Spreadsheet Editor allows the user to specify the cell format, but the Cell must maintain its format thereafter.

numbers convert to text
The responsibility for performing this conversion belongs to the class Numeric Value.

users select rectangular groups of cells
cut selected cells
copy selected cells
paste cells
remove rows
remove columns
insert rows
insert columns
remove values of cells
replace values of cells
edit values of cells
The responsibility for receiving user input belongs to the class Spreadsheet Editor. Many of these responsibilities imply that other classes must perform other operations as well. We shall return to this point later, when we discuss collaborations.

recompute values of cells
This is also a compound responsibility. The Spreadsheet Editor allows the user to request that the values be recomputed, but the Expression must perform the actual computation.

**Record responsibilities.** As you assign responsibilities to specific classes, record them on the card for that class, under the class name, on the left edge.

### DETERMINING COLLABORATIONS

A collaboration is a request made of one object by another. It is the embodiment of the requests specified in the client-server contract. A single collaboration flows in one direction—from the client to the server. Every collaboration is associated with a single responsibility. It fulfills, or contributes to the fulfillment of, that responsibility.

Collaborations are important because the pattern of collaborations within your application reveals how control and information will flow during execution. Identifying collaborations between classes allows you to identify paths of communication between classes. Finding such paths will ultimately allow you to identify subsystems of collaborating classes. Finding such subsystems is one way in which you will later be able to further abstract your application.

**Identify collaborations.** To identify collaborations, ask the following questions for each responsibility of each class:

- Is the class capable of fulfilling this responsibility itself?
- If not, what is needed?
- From what other class can it acquire what it needs?

Let's look at each of the responsibilities assigned to the classes in the spreadsheet application. In general, responsibilities to maintain information require no collaborations. Unless they are an exception, we will not discuss such responsibilities.

Expression:
compute values
This generic operation requires no collaborations. However, subclasses of the class Expression require collaborations in order to fulfill their specific responsibilities, as described later.

Binary Expression:
compute values
This operation requires a collaboration with the expressions representing the arguments to the binary operator. These expressions may be a member of any subclass of the class Expression. We therefore record a collaboration with the class Expression.

Cell Reference Expression:
compute values
This operation requires a collaboration with the cell being referenced, an instance of the class Cell.

Numeric Value:
convert to text
This operation occurs during the computation of expressions. It requires no collaborations.

Spreadsheet Editor:
open from a file
save to a file
Clearly, this involves a collaboration with the class File.

allow user to specify cell format
This responsibility involves a collaboration with the class Cell so that the format will be remembered.

users select rectangular groups of cells
cut selected cells
copy selected cells
paste cells
remove rows
remove columns
insert rows
insert columns
remove values of cells
replace values of cells
edit values of cells
allow user to request to recompute values of cells
The Spreadsheet Editor is responsible for interpreting user input. It must then inform the spreadsheet that it has changed, requiring a collaboration with the class Spreadsheet. Responsibilities that alter cells or groups of cells must similarly collaborate with the classes Cell or Cell Group.

**Record collaborations.** Record these classes as collaborations on the card for that class directly opposite the responsibility the collaboration supports. Check to see that a corresponding responsibility exists for every collaboration you record. Remember, however, that a collaboration might be with a subclass, but the responsibility might be recorded on the superclass card instead.

**Design walk-throughs.** As you make these design decisions, it's important for you to be able to determine their implications. For this purpose, you should walk through your system after each step. Choose a set of typical inputs to your system, and hand-simulate its behavior, given these inputs. In this way, you can more easily determine the implications of your decisions. Feel free to revise previous decisions as you go, and walk through your new configuration. The point of this stage of your design process is, after all, to explore as many different possibilities as seems reasonable. Walk-throughs can help you determine the implications of these various possibilities.

Let's look at what happens when a cell is asked for its value. Cells maintain their values indirectly by storing an instance of a subclass of class Value. Therefore, cells must retrieve their values when requested by sending a message to a Value. The Value may represent the value directly, as with a number or text, or it may know the expression by which the value can be computed. In the latter case, the Value must ask the expression to evaluate itself.

Expressions evaluate themselves differently depending on which type of expression they are. Constant Expressions evaluate themselves by returning the constant they represent. Cell Reference Expressions evaluate themselves by asking for the value of the cell they reference. Binary Expressions evaluate themselves by applying their operator to the values of their two arguments. So far, it seems the system works the way it was intended to.

## THE ANALYSIS PHASE

The analysis phase of object-based design also consists of three steps:

1. Building optimal inheritance hierarchies.
2. Streamlining the collaborations between classes.
3. Defining the signatures for each responsibility.

Let's look at each of these steps individually.

### BUILDING HIERARCHIES

A carefully considered and crafted inheritance hierarchy provides the maximum amount of reusable code. Carefully assigning responsibilities ensures that the resulting hierarchies of classes are easily reused, maintained, and extended.

**Record existing hierarchies.** First, examine the present class hierarchies in the design. Draw hierarchy graphs of your application. The hierarchy graph is rather simple. Classes are represented by rectangles, labeled with the class names. Inheritance is indicated by a line from superclass to subclass, and by position on the page—superclasses are above their subclasses.

Analyze the responsibilities assigned to each class to determine whether each class is abstract or concrete.

- Abstract classes are designed to be inherited. Instances of abstract classes are never created as the system executes.
- Concrete classes are designed to be instantiated. They are designed primarily so that their instances may be generally useful, and secondarily so that they may also be usefully inherited.

Go through your inheritance hierarchies, labeling each class as abstract or concrete on the cards and by filling in the upper-left corner of hierarchy graphs for abstract classes. If you have trouble deciding whether a given class is abstract or concrete, think about your working system. Will an instance of this class be used during execution? If so, the class is concrete.

Our spreadsheet program includes two abstract classes: Expression and Value. All other classes in this design are concrete. The hierarchy graphs for hierarchies containing more than one class appear as shown in Figure 2.

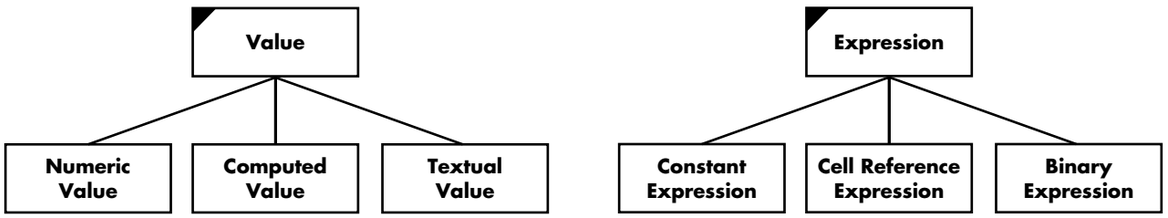



**Figure 2.**
**Hierarchy Graphs for Our Abstract Classes**

**Restructure hierarchies.** Because our spreadsheet example is so small, there is nothing we can show you here to exemplify optimizing the hierarchy. Nevertheless, the following guidelines can help you build better hierarchies:

- When you have determined how many abstract classes are presently in your design, speculate on abstract classes that might encapsulate behavior that could be reused by existing and future subclasses. In general, the more abstract classes an application has, the more code in the application can be reused. Therefore, define as many abstract classes as seems reasonable to capture the abstractions present in your design, or that you reasonably suspect you might have future use for.
- Factor responsibilities as high as possible in the hierarchy. If a set of classes all support the same set of responsibilities, all the classes should inherit those responsibilities from a common superclass. If a common superclass does not exist, create one, and move the common responsibilities to it. After all, such a class is demonstrably useful—you have already shown that the responsibilities will be inherited by some classes.
- Factor implementation details as low as possible in the hierarchy. If a superclass supports its responsibilities in only the most generic possible way, (providing only templates, as it were, for the desired behavior), then implementation details cannot impede a new subclass from inheriting those responsibilities. Each subclass is free to implement the responsibilities in the way most appropriate for it. This can include subclasses unforeseen by the original design.
- Ensure that each class encapsulates a single integral set of responsibilities. Each class should have a single, overarching purpose; each class should serve one main function in the system of which it is a part.

These observations of what enhances or detracts from the reusability of a class lead to the principle that the appropriate use of inheritance is to model a type hierarchy: every class should be a particular kind of its superclasses. Subclasses should add responsibilities to their superclasses; they should not cancel inherited responsibilities, or override them to become errors, or no behavior at all.

When you have modified your design, redo your graphs and cards to correspond to the new state of your design. Then recheck your system. For each responsibility, make sure there is a corresponding collaboration, and vice versa. Once again, walk through the design to ensure that every object is still communicating with the rest of the system in the appropriate manner.

**Group responsibilities into contracts.** Once the responsibilities have been properly factored in the hierarchies, they need to be grouped into the contracts supported by each class. This is usually straightforward because classes usually support a small and cohesive set of responsibilities. If the responsibilities of a class are not cohesive, it should have more than one contract. Not all responsibilities will be public behavior for the class. Only public behavior should be grouped into contracts. Number the contracts so that they can be referenced.

Here are the contracts for the classes in the spreadsheet design:

Cell
1. Maintain the value and format
2. Compute the value

Cell Group
3. Know the cells contained in the group

Expression
4. Compute the value of the expression

File
5. Input and output to disk

Spreadsheet
6. Know the group of cells within it

Value
7. Compute the value represented

## STREAMLINING COLLABORATIONS

We are now going to streamline the collaborations between classes—each communication path that can occur as information and execution flows through the system. We analyze these collaborations to attain an overall perspective, to identify natural ways to divide responsibilities between groups of classes, and thereby to simplify the various ways in which communication can flow. Simplifying the potential communication flow simplifies the application: the application becomes easier for others to understand, maintain, reuse, refine, and extend.

Earlier, we discussed performing a walk-through of your system, trying out various scenarios, simulating the results of various typical inputs. Each such scenario brings to light one possible path along which information and control can flow.

To do a good job of analyzing collaborations between objects, you must first collect an exhaustive description of all the paths along which control and information can flow. You can then analyze the collaborations between classes in order to simplify them.

A collaborations graph is a tool for accomplishing this analysis. A collaborations graph allows you to examine the collaborations between classes in graphical form, so that you can better identify areas of unnecessary complexity or other design flaws. Collaborations graphs represent four distinct elements: classes, subsystems, contracts, and collaborations.

Classes are shown as labeled rectangles. Subsystems are shown by drawing a rounded rectangle around the classes that comprise them. Contracts are shown as small semicircles inside the edges of the class or subsystem to which they belong. Draw one semicircle per contract, labeled by the contract number. Collaborations between classes or subsystems are represented by an arrow from the client to a contract supported by the server. If two objects both collaborate with a class by means of the same contract, draw the arrows to the same semicircle. Otherwise, draw the arrows to different semicircles.

In addition, collaborations graphs show superclass/subclass relationships, such as that between the class Value and the specific kinds of values, or between Expression and the different kinds of expressions. A superclass represents the contracts supported by all of its subclasses; because of polymorphism, we can focus on the abstract contract. We need not consider whether the superclass, or one of its subclasses, will be the object actually providing the service during execution. This is represented in the collaborations graph by nesting subclasses within the bounds of their superclasses.

Figure 3 shows the collaborations graph of the spreadsheet application as we have so far designed it.

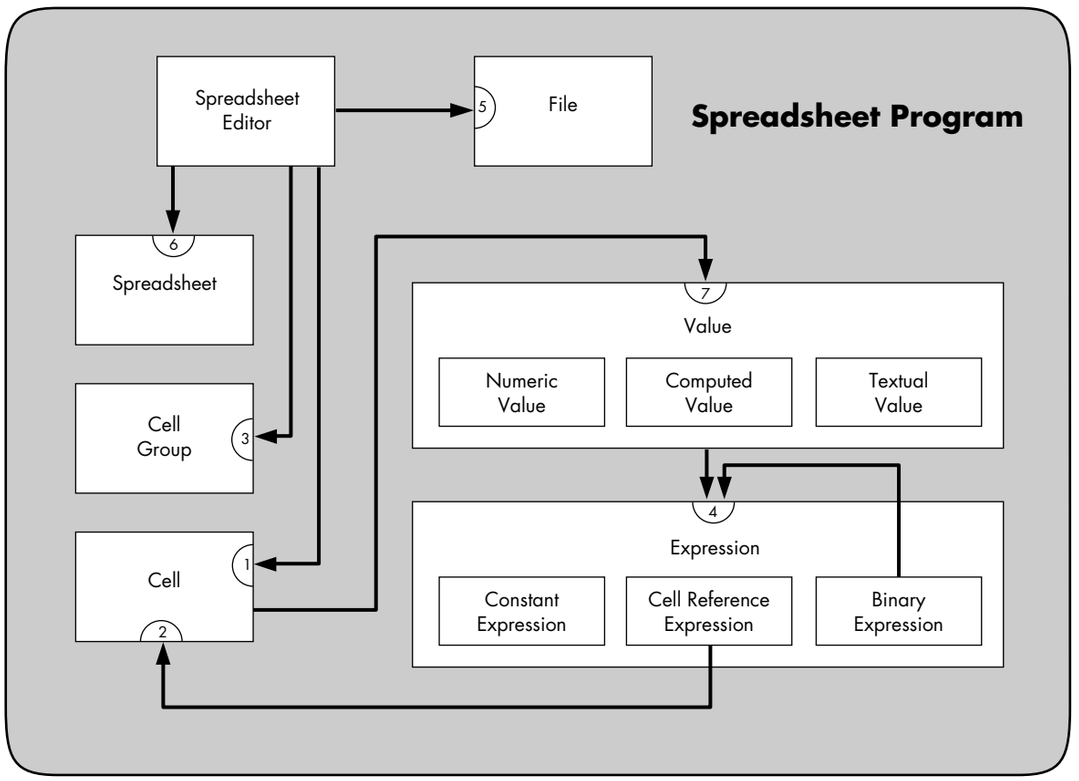



**Figure 3.**
**Initial Collaborations Graph for Our Spreadsheet Program**

The goal of this step in the design process is to simplify the patterns of collaboration. Without such simplification, the communication paths could flow from nearly any class to any other, with only the slenderest of justifications and no coherent structuring. Such anarchic flow leads to spaghetti code—the same problem that eliminating "go to" statements was designed to avoid.

Because such applications are impossible to maintain or sensibly modify, we aim to simplify the patterns of collaboration. Successfully doing so translates into a simplification of the graph. The technique we will use, at least in part, is to work backward: we shall simplify the graph in order to simplify the collaborations. What criteria should you use to accomplish this simplification?

- Minimize the number of different contracts supported by each class and subsystem. Too many contracts for one class or subsystem can be a sign that too much of the application's intelligence is concentrated in that class or subsystem.
- Each contract supported by a subsystem should be handled by only one class or subsystem. If the contract representing the external interface of a subsystem mediates direct collaborations with two or more classes, it can be a sign that a level of indirection is missing, or that the contract is really two or more contracts.
- Minimize the number of classes and subsystems within a subsystem that are collaborated with by classes or subsystems outside the subsystem. Otherwise, your subsystem does not truly encapsulate its component entities. It does not provide the desired level of abstraction.

Three basic mechanisms can be used to simplify your graph, and hence to streamline the collaborations between your classes and subsystems.

- Build clean subsystems by centralizing communications to a subsystem or introducing an intermediary to a subsystem.
- Coalesce classes whose responsibilities overlap.
- Split classes with too many contracts.

Our spreadsheet application can be cleanly divided into two large pieces: the editing capabilities and the structure being edited. For this reason, it makes sense to create a subsystem representing the structure of a spreadsheet, which we will call the Spreadsheet Subsystem. The Spreadsheet Subsystem is responsible for creating spreadsheets, and maintaining their structure.

It may well be that the Spreadsheet Editor is itself really a subsystem rather than a single class, but in the interests of simplicity let's presume that it is a class.

Having created the Spreadsheet Subsystem, we need to clean up the way in which the Spreadsheet Editor collaborates with it. In particular, the Spreadsheet Editor should not collaborate with so many of the classes inside the subsystem. We can simplify the paths of collaboration by forcing all accesses to other classes to go through the Spreadsheet. This implies that Spreadsheets must be able to understand and pass along all messages to the cells or cell groups that compose them. Spreadsheets therefore now collaborate with Cell Groups, which in turn collaborate with Cells. Two new collaborations therefore appear in the graph.

This set of changes results in the graph of the application shown in Figure 4.

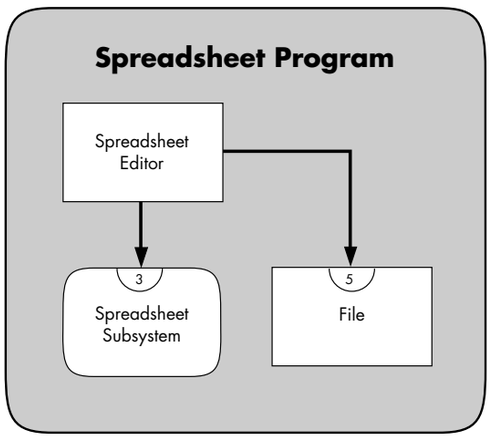



**Figure 4.**
**Simplified Collaborations Graph for Our Spreadsheet Program**

The collaborations within the Spreadsheet Subsystem would then look like Figure 5.

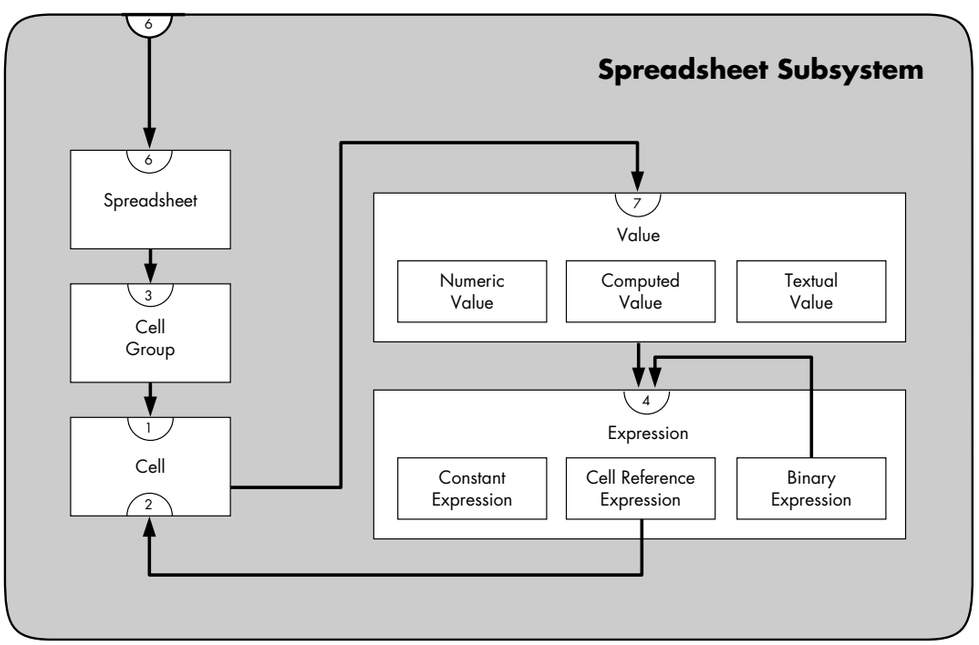



**Figure 5.**
**Collaborations Graph for the Spreadhseet Subsystem**

## DEFINING SIGNATURES

Once the responsibilities have been assigned to classes, and changes are unlikely, the final stage of the design process is to refine the responsibilities into protocols. A *protocol* is a set of messages to which an object will respond.

The goal of this part of the process is to make the classes in your application, particularly their instances, as generally useful as possible. This is accomplished in two ways:

- Maximize polymorphism. Polymorphism, as you recall, is the ability of instances of different classes to respond to the same message, each in its own appropriate way. Polymorphism has already been maximized by moving responsibilities as high in the hierarchy as they can reasonably go. By moving a responsibility from a class to its superclass, you increase the number of classes that can support that responsibility, and hence respond to that message.

  Polymorphism can also be maximized by carefully selecting message names, so that it makes sense for instances of many classes to respond to messages by those names. Use a single message name for each conceptual operation, wherever in the system it is found. Likewise, associate a single conceptual operation with each message name.

- Make the protocol as generally useful as possible. Instances will be more reusable if the protocols used to make requests of them have been designed in anticipation of as many different uses as possible. Think about what might change if the system were modified or extended. Think about what related systems might wish to use.

First, define the most general message, one that allows clients to supply all possibly required parameters. Next, provide reasonable default behavior for as many parameters as possible. Finally, analyze how each client uses (or is likely in the future to use) this general message. From that analysis, define a useful set of messages that allows clients to specify only some of the parameters, while relying on the defaults for the others.

List the contracts of each class or subsystem in your application, and turn each contract into a set of signatures. Each contract will have one or several messages associated with it. Name these messages thoughtfully, bearing in mind the considerations just described. Along with the message names, specify the types of all arguments required, and the type of object returned by the method, if any.

Here is an example set of signatures for the class Spreadsheet:

Class: Spreadsheet

6. Know the group of cells within it
   cells() returns Cell Group
   cells(Cell Group) returns void
   row(Integer) returns Cell Group
   column(Integer) returns Cell Group
   rows(Integer, Integer) returns Cell Group
   columns(Integer, Integer) returns Cell Group
   rowsAndColumns(Integer, Integer, Integer, Integer) returns Cell Group
   positionOfCell(Cell) returns String
   cellAt(Integer, Integer) returns Cell
   cellNamed(String) returns Cell

You are now ready to write a formal specification for each class. The specification will state the name of the class and its overall purpose, whether it is abstract or concrete, its position in its inheritance hierarchy and the collaborations graph, and its contracts and their associated signatures. Each signature should be followed by a description of the behavior captured by the signature. In addition, include any notes on special implementation considerations, such as algorithms, behavioral constraints, or error conditions.

As a result of this design process, you now have one or more collaborations graphs, one or more hierarchy graphs, a specification for each class, and a set of formal contracts for each class.

You are now ready to implement your application.

## CONCLUSION

The result of this process is a design based on objects. The responsibilities of each object become messages to which the object will respond by providing the services requested. Collaborations represent classes from which an object must request operations or information in order to fulfill its own responsibilities.

The design therefore supports the basic concepts of object-based technology—it encapsulates operations and information within objects, it hides details of the state of an object, and it uses inheritance to incrementally refine the definitions of objects, maximizing the amount of reusable code.

Classes and subsystems can be tested before they are connected to the entire application. Because the paths of communication are mapped out and rigorously controlled, maintenance can be performed without risking unpredictable side-effects. Finally, because the software has been designed from the start with future

extensions in mind, functionality can be added to the application with a minimum of difficulty.

Applications implemented from such a design can therefore reap the benefits of object-based technology.

## FURTHER READING ON OBJECT-BASED DESIGN


Beck, Kent, and Ward Cunningham, "A Laboratory for Teaching object-based Thinking," *OOPSLA'89 Conference Proceedings*, SIGPLAN Notices, October 1989, pp. 1-6.

Cox, Brad, "Message/Object Programming: An Evolutionary Change in Programming Technology," IEEE Software, January 1984, pp. 50-61.

Halbert, Daniel, and Patrick O'Brien, "Using Types and Inheritance inobject-based Languages," IEEE Software, September 1987, pp. 71-79.

Johnson, Ralph, and Brian Foote, "Designing Reusable Classes," *Journal of* object-based Programming, June/July 1988, pp. 22-35.

LaLonde, Wilf, "Designing Families of Data Types Using Exemplars," *ACM Transactions on Programming Languages and Systems,* April 1989, pp. 212-248.

Meyer, Bertrand, "Reusability: The Case for object-based Design," *IEEE* Software, March 1987, pp. 50-64.

Snyder, Alan, "Encapsulation and Inheritance in object-based Programming Languages," *OOPSLA'86 Conference Proceedings*, SIGPLAN Notices, November 1986, pp. 38-45.

Wirfs-Brock, Allen, and Brian Wilkerson, "Variables Limit Reusability," *Journal of object-based Programming,* May/June 1989, pp. 34-40.

Wirfs-Brock, Rebecca, and Brian Wilkerson, "object-based Design: A Responsibility-Driven Approach," *OOPSLA'89 Conference Proceedings*, SIGPLAN Notices, October 1989, pp. 71-76.

# UNOFFICIAL C++ STYLE GUIDE

*C++ is fast emerging as the premier object-based language of the 1990s. Its expressive power combined with its down-and-dirty C heritage makes it a natural choice for writing Macintosh applications. But beware: all is not sweetness and light. Harnessing C++'s power without getting tripped up by some of its less savory features isn't all that easy. Here are guidelines that make it easier to write, debug, and change C++ programs.*

**DAVID GOLDSMITH**
**JACK PALEVICH**

When we first started using C++ several years ago, we learned the hard way—by trial and error—a lot of what we're about to tell you. After thousands of hours of C++ programming, we've formed strong opinions about the best way to use C++. In this unofficial style guide we tell you which features to use and which ones to avoid. Following our techniques will lead to programs that are easier to write, debug, and change. You may not agree with all of our guidelines—some are more a matter of taste than of science—but we hope you'll find them useful and enlightening.

## PART 1: STYLE

In this part you'll find fairly simple advice on formatting source files. Even if you prefer to use a different style in your own work, you may be interested to see how we handle these standard matters of style.

### SOURCE FILE CONVENTIONS

Use the following conventions to keep your source files easy to read, easy to use, and legally protected.

**Include a copyright notice.** To protect your intellectual property rights, include the following line at the front of every file you create:

```
// Copyright © 1990 ~~~Your name or company~~~.  All rights
reserved.
```



**DAVID GOLDSMITH** has been at Apple for four years, and now focuses his energies on future system software. He has also worked on MacApp and the Macintosh Toolbox (during the inception of System 6.0). He received an MA in physics from Harvard in 1980, and since has worked for Wang, Mosaic Technologies, and as a contractor for Lotus. For him, the thrill of the job is "making computers available to people whose lives don't revolve around computers." His main interest outside the office is his family, with its new member, eight-month-old Jane. He also enjoys reading and listening to classical music.

**"HackerJack" Palevich** is our local meerkat handler (look that one up in National Geographic!). He has a degree in computer science from MIT, where his "Thesis of Terror"

You can make the © by typing Option-g. The "All rights reserved" is specifically for our foreign friends. (We bet you thought all you had to say was "Copyright." Ha!) In addition, any binary files you ship should contain a copyright notice somewhere.

If you modify a file in more than one calendar year, you must list every year in which you modified it. For example,

```
// Copyright © 1988-1990 ~~~Your name or company~~~.  All rights
reserved.
```

**Add helpful comments.** We won't go into a lengthy discussion of comment style here. We'll only say it's good to have them. Also, comments that describe something subtle about the source are more helpful than comments like `assign a to b`.

**Be careful about omitting argument names in function prototypes.** It's OK to omit dummy argument names in function prototypes, but only if the meaning is clear without them. It's almost always necessary to include argument names when you have more than one argument of the same type, as it's impossible to figure out which one is which otherwise.

```
double cosine(double angle);      // Prototype with argument name.
double cosine(double);            // Reasonable omission of arg.
name.
TPoint::TPoint(short h, short v); // Easy to understand.
TPoint::TPoint(short, short);     // Impossible to figure out.
```

If you are getting compiler warnings of the form "warning: foo not used" where `foo` is an argument to a function, leaving the argument name out of the function header for the function's implementation will stop the warning. Whether or not an argument name appears in the function's declaration has no bearing on the warning.

```
void f(short foo, long bar);      // Prototype.
void f(short /* foo */, long bar) // Implementation.
      {
      bar = 7;
      }
```

**Put only related classes in one file.** To keep your class definitions under control and to make life easier for those trying to decipher them, follow the lead of the MPW {CIncludes} files. Limit each header file to a single class definition or a set of related class definitions. MPW C has always followed this convention, for example Windows.h, Controls.h versus Toolbox.h.

On the implementation side, put only one class implementation in a given source file (classes private to the implementation of the class may be declared and implemented



was a prelude to his video game days at Atari. Two years at Apple have landed him in video conferencing and screen sharing. As the son of an American diplomat, he's lived in some pretty interesting places (Berlin, Poland, Laos, Greece, Maryland, and Cupertino). He didn't buy an Apple II in the eleventh grade, and has regretted it since. His interests include filmmaking, animation, and aerobics. He claims to be halfway a nerd, but only his wife knows for sure. He also likes to tell stories, so we're not sure that any of this is really true.

in the same source file). Name the file after the class, but without the initial *T*—for example, put the class **`TMyView`** in MyView.c.

**Make it easy to use your header files.** Trying to figure out whether you've included all the necessary antecedents for a header file is a pain. To save your clients this pain, enclose the definitions in your header with code that looks like the following:

```
#ifndef __MYCLASS__
#define __MYCLASS__
#include "prerequisite1.h"
#include "prerequisite2.h"
… definitions for MyClass
#endif
```

Now you can include your header's prerequisites without caring whether they've already been included elsewhere (assuming that everyone follows this convention). The name of the preprocessor variable should be all upper case and consist of the file name (without *.h*) surrounded by two underlines on either side.

To speed up compilation, you can even do this in your files that use foo.h:

```
#ifndef __FOO__
#include "foo.h"
#endif
```

which skips the overhead of reading and parsing foo.h.

**Store files in Projector.** As soon as a file is published for use by others (for example, you stick it on a file server so others can use it), you should start storing it in Projector. Projector is part of MPW 3.0. It consists of a collection of built-in MPW commands and windows that help programmers (both individuals and teams) control and account for changes to all the files (documentation, source, applications, and so forth) associated with a software project. This lets you recreate old versions if necessary, and makes sure things don't get lost.

Since the whole point of using Projector is to make it easier for those who follow you in the great chain of software being, please use the features that will make their lives easier. Try to maintain a proper set of versions (for example, don't remove your file from Projector and then add it again—that loses all the revisions), and use the comment features when you check things in and out.

## NAMING CONVENTIONS

To make C++ even more readable, you should adopt a consistent set of naming conventions. Here at Apple we use the following conventions:



**This** is an enhanced version of an internal, informal Apple style guide written by David Goldsmith. Jack Palevich edited the original document to bring it into its current form. The opinions expressed in this document are David's and/or Jack's. They are not necessarily those of Apple Computer, Incorporated. 

**Type names** *All* type names begin with a capital letter. In addition, class names begin with a *T* for base classes, and an *M* for mix-in classes. See "Multiple Inheritance" in Part 2. Examples: **`Boolean`**, **`TView`**, **`MAdjustable`**. Never use C types directly; see below.

**Member names** Member names should begin with an *f*, for "field." Member function names need only begin with a capital letter. Examples: **`fVisible`**, **`Draw.`**

**Global names** Names of global variables (*including* static members of classes) should begin with a *g*. Examples: **`gApplication`**, **`TGame::gPlayingField.`**

**Local and parameter names** Names of local variables and function arguments should begin with a word whose initial letter is lower case. Examples: **`i, thePort`**, **`aRegion.`**

**Constant names** Names of constants should begin with a *k*. Example: **`kSaveDialogResID.`**

**Abbreviations** It's best to avoid abbreviations, especially ad hoc ones. Inconsistent use of abbreviations makes it hard for clients to remember the correct name of a function or variable. Abbreviations are OK as long as they are consistent and universal. For example, don't use **`VisibleRegion`** some places and **`VisRgn`** others; use one or the other throughout.

**Multiple-word names** In any name that contains more than one word, the first word should follow the convention for the type of the name, and subsequent words should immediately follow, with the first letter of each word capitalized. Do not use underscores in names. Here are multiple-word examples of each type:

| | |
|---|---|
| **`TSortedList`** | class name |
| **`fSubViews`** | data member of class |
| **`DrawContents`** | function member of class |
| **`gDeviceList`** | global or static data member |
| **`theCurrentSize`** | local or parameter |
| **`kMaxStringLength`** | constant |

**Names with global scope** Any name with global scope (for example, class names, typedefs, constants, globals) should have a distinctive and unique name. This will help avoid name conflicts. Names like **`Short`** and **`Number`** are fairly nondescript and likely to wind up conflicting with identifiers from other header files accidentally (this is a big problem with MPW and the ROM interfaces today). Better in these cases would be **`kShortLived`** (to follow our advice on constant names) or **`StringLength`** (more descriptive of the function). When you name something with global scope, think about the fact that it's in a global name space and someone may have to figure out what it is without context. Use more specific names rather than more general ones.

C++ relieves this problem somewhat by adding enumerations with class scope and static members. Enumerations declared inside classes are accessible using qualification, as in

```
class TFoo {
public:
       enum {kFred, kBarney};
…
};

i = TFoo::kFred;
```

This lets you put constants associated with different classes into different name spaces, somewhat like when C changed a few years back so that structure members from different structs were in different name spaces.

Static members let you put ordinary functions and global variables associated with a class into the scope of the class. For example:

```
class TView {
public:
       static void Initialize();
       static const TView kWhizzyView;
       static const long kMagicNumber;
…
};
TView::Initialize();
…TView::kWhizzyView…
i = TView::kMagicNumber;
```

Putting such global functions and variables into the scope of the class helps avoid name collisions. In fact, we frown on the use of ordinary globals: most global functions and variables should be static members of some class. The same with constants; they should be made members of an enumeration inside a class, if possible. Of course, global variables that are not constants of the sort shown above shouldn't be public at all; instead, access should be through static or normal member functions:

```
class TFoo {
public:
       static Boolean gWhoopeeFlag;      // BAD!
}
TFoo::gWhoopeeFlag = TRUE;               // BAD!
```

## THE PREPROCESSOR

One of the most powerful features of the C and C++ languages is the C preprocessor.

Don't use it.

Except for include files and conditional compilation, C++ has features that supersede most of the techniques that used the preprocessor. Sometimes you need to use the preprocessor to accomplish things you can't with C++, but far less often than with straight C.

**Use const for constants.** Don't use `#define` for symbolic constants. Instead, C++ defines the const storage class. As with `#define` symbols, these are evaluated at compile time. Unlike `#define` symbols, they follow the C scope rules and have types associated with them. You can also use enums. For example:

```
#define kGreen 1                        // No no
const int kGreen = 1;                   // Better
enum Color {kRed, kGreen, kBlue};       // Best
```

This prevents a host of problems. With `#define` symbols, for example, if you accidentally redefine a name, the compiler will silently change the meaning of your program. With const or enums, you'll get an error message. Even better, with enums you can put the identifiers in the scope of an enclosing class; see "Naming Conventions," earlier. As an extra bonus, each enumeration defined is treated as a separate type for purposes of type checking (much like the way scalars are handled in Pascal).

Unlike in ANSI C, objects in C++ that are declared const and initialized with compile-time expressions are themselves compile-time constants (but only if they are of integral type). Thus, they can be used as case labels and such.

**Use enum for a set of constants.** If your constants define a related set, don't use separate const definitions. Instead, make your constants an enumerated type. For example:

```
// Bleah.
const int kRed = 0;
const int kBlue = 1;
const in kGreen = 2;
// Allll Riiiight!
enum ColorComponent {kRed, kBlue, kGreen};
```

This causes `ColorComponent` to become a distinct type that is type-checked by the compiler. Values of type `ColorComponent` will be automatically converted to int as needed, but integers cannot be changed to `ColorComponents` without a cast. If you need to assign particular numerical values, you can do that too:

```
enum ColorComponent {kRed = 0x10, kGreenk = 0x20, kBlue = 0x40};
```

Where possible, the type declaration should occur within the scope of a class. Then, references to the constants outside of the class's member functions must be qualified:

```
class TColor {
public:
       ColorComponent enum {kRed, kGreen, kBlue};
...
}
foo = TColor::kRed;
```

Note that the enum type name is not local to the class; only the actual constants. The enum type name should not be qualified.

**Use inline for macro functions.** Function macros are another source of fun problems in C programs, like this classic example:

```
#define SQUARE(x) ((x)*(x))
SQUARE(y++);
```

C++ allows functions to be declared inline (see also "Inline Functions" in Part 2), which completely obviates the need for function macros. Like const, inline functions follow the C++ scope rules and allow argument type-checking. Both member functions and nonmember functions can be declared inline. So the preceding example becomes

```
inline int Square(int x)
{
       return x*x;
};
Square(y++);
```

which does the right thing, and is actually more efficient than the macro version (as well as being correct).

**Use the preprocessor only in these cases.** As stated earlier, the preprocessor is necessary for `#include` files, and preprocessor symbols are necessary for conditional compilation.

### USE OF CONST

Both ANSI C and C++ add a new modifier to declarations, **`const`**. You use this modifier to declare that the specified object cannot be changed. The compiler can then optimize code, and also warn you if you do something that doesn't match the declaration. Here are some examples of const declarations:

```
const int *foo;
```

This is a modifiable pointer to constant integers. **`foo`** can be changed, but what it points to cannot be.

```
int *const foo;
```

This is a constant pointer to modifiable integers. The pointer cannot be changed (once initialized), but the integers it points to can be changed at will.

```
const int *const foo;
```

This is a constant pointer to a constant integer. Neither the pointer nor the integer it points to can be changed.

Note that const objects can be assigned to non-const objects (thereby making a copy), and the modifiable copy can of course be changed. However, *pointers* to const objects *cannot* be assigned to pointers to non-const objects, although the converse is allowed. Both of these assignments are legal:

```
(const int *) = (int *);
(int *) = (int *const);
```

Both of these assignments are illegal:

```
(int *) = (const int *);
(int *const) = (int *);
```

When **`const`** is used in an argument list, it means that the argument will not be modified. This is especially useful when you want to pass an argument by reference, but you don't want the argument to be modified. For example:

```
void BlockMove(const void* source, void* destination, size_t
length);
```

Here we are explicitly stating that the source data will not be modified, but that the destination data will be modified. (Of course, if the length is 0, then the destination won't actually be modified.)

All of these rules apply to class objects as well; you can declare something (**`const TView *`**). There used to be a hole in the language, however: you could call any member function of an object using a const pointer to it, and that member function could modify the object (since there was no way to declare which member functions modify the object). For example, this was legal:

```
const TView *aView;
…
aView->ModifySomething();
```

To plug this hole, member functions that will be called for const objects must now be declared const; see the 1985-1989 paper discussed in the sidebar "Background Reading" for details. The syntax looks like this:

```
class TFoo {
public:
      void Bar1() const;
      void Bar2();
};
…
const TFoo *fp;
fp->Bar1();  // legal
fp->Bar2();  // illegal (actually, just a warning for now)
```

Note that inside a const member function, the pointer has type **`const TFoo *`**, so you really can't change the object. You could cast the pointer to be just a **`TFoo *`**, but then you may be surprising your clients. Even though you think that your change to the object is not externally visible (that is, it doesn't change the state of the object as far as clients are concerned—one example is an internal cache), it can have an impact. If your object is being used by an interrupt routine that "reads" it, your client may assume that it's OK to call a const member function, since he or she thinks the object isn't going to change. However, if the internal state of the object changes anyway, access by multiple "readers" will cause its state to become corrupted.

Another example is an object placed in ROM. The client thinks it's all right to call a const member function of the object, and then gets a bus error because the attempted write access fails.

The bottom line is that if you attempt to cast your **`this`** pointer to a non-const version inside a const member function, you had better think through the implications of this for your clients, and you had better document it.

## PART 2: USING LANGUAGE FEATURES

In this part you'll find advice on using particular features of the C++ language. The topics are arranged roughly in order of increasing difficulty.

### GLOBAL VARIABLES (!)

**Static class members are the same as global variables (actually, better).** Static class members do have one major advantage over regular globals: scope. Regular globals (that is, **`static extern`** variables) have global scope. That means there are potential name collisions with globals from any other **`include`** file the developer may use. Static class members, however, have full scoping: they're qualified by the name of their class, and don't conflict with any identifier outside the class. They can also be protected. If you were going to have a simple global, consider a static member instead.

**Be careful about static initialization.** If you design a class that depends on some other facility in its constructor, be careful about order dependencies in static initialization. The order in which static constructors (that is, the constructors of objects with `static` storage class) get called is *undefined*. You cannot count on one object being initialized before another. Therefore, if you have such a dependency, you must either document that your class cannot be used for static objects, or you must use "lazy evaluation" to defer the dependency until later.

### INLINE FUNCTIONS

We mentioned inline functions under "The Preprocessor," earlier. Never use them. Well, hardly ever. The main reason is that they get compiled into your caller's code. This makes them a tad difficult to override. Also, you have to ship their source code to everyone. There are, however, a few times when it's OK to use them.

**Use an inline function if it expands to call to something else.** If your inline function just calls something else that isn't inline, that's fine, as long as the other function *has identical semantics*. An example: You might have a class that defines a virtual function `IsEqual`, which compares two objects for equality. It also has an inline definition for `operator ==`, as a notational convenience. Since `operator==` just turns around and calls the `IsEqual` function, it's OK for it to be inline and not virtual. This does *not* apply if your function just happens to have a one-line implementation.

**Use an inline function if efficiency is very, very important and you'll never change it.** Of course, the other time it's OK to use inline is if efficiency is extremely important. Note that code size may increase due to duplication of code. *You may actually decrease system performance by making something inline*, since you're increasing the amount of code that must fit in memory. Also, once a function is more than a couple of lines long, the function call overhead is a very small fraction of the total time, and you are not buying much by making it inline.

An example is addition for a complex number type; here, the efficiency consideration together with the low probability of a future change makes an inline implementation a good idea. The complex number implementation shipped with C++ makes addition and subtraction inlines (fairly short), but makes multiplication and division regular functions (they are longer, so the overhead for a call is less important and the code size issue is more important).

If you don't *know* (that is, God has told you in person) that your implementation must be inline, *don't make it inline*. Build it normally and then *measure* the performance. Experience has shown again and again that programmers spend lots of time optimizing code that hardly ever gets executed, while totally missing the real bottlenecks. The empirical approach is much more reliable. Experience has also shown that a better algorithm or smarter data structures will buy you a lot more performance than code tweaking.

**Don't write inlines in declarations.** C++ has two ways of declaring an inline member function (of course). One is to declare the member function normally and then supply an inline function definition later in the same header file. The other is to write the function definition directly in the class declaration. Never use this latter form; always declare the function normally and then put an inline definition at the end of the file. That way, it's much easier to change between inline and regular implementations of a function, and it's no less efficient. The fact that something is inline should not be made obvious in the class declaration, since clients may start counting on it.

```
class TFoo {
public:
      int TweedleDee() { return 1; };   // Bad!
      int TweedleDum();                 // Good!
};

inline int TFoo::TweedleDum()
{
      return 2;
};
```

### UNSPECIFIED AND DEFAULT ARGUMENTS

It's possible to partially circumvent the strong type checking C++ imposes on function arguments. You should avoid doing this if at all possible.

**Don't use unspecified arguments.** C++, like ANSI C, allows you to cling to C's wild and woolly past, by declaring functions that take unspecified numbers and types of arguments, the classic example being

```
void printf(char *, ...);
```

This is a cheesy leftover from the Cretaceous era. There are very few functions indeed that need to have an interface like this. If you want to be able to omit arguments, for example, you can use default arguments or function overloading (both defined below). Unspecified arguments come from hell.

**Do use default arguments, but cautiously.** A better technique than unspecified arguments is default arguments. You can specify default values for arguments that are only used sometimes. This is especially handy in constructors. For example,

```
TView::TView(TVPoint itsSize, TVPoint itsLocation, TView
*itsSuperView = NIL);
```

which can be called either with three arguments or with two. This can help you avoid that agonizing decision as to whether to include an option or not. However, be sparing; clearly, long strings of defaults can make it hard to figure out what's going on. Furthermore, you can only leave off arguments at the end, not the middle, so if

you have ten defaults and someone wants to specify the last one, they must specify the preceding nine as well. This sort of defeats the idea. Having more than two default arguments is a bad idea, and even two is questionable.

Also remember that for both default arguments and function overloading, having too many versions of the same function decreases the safety provided by type checking, and makes it more likely that you will accidentally call a different version from the one you intended to call.

### FUNCTION NAME OVERLOADING

C++ also lets you overload function names, by letting two functions (member or nonmember) have the same name as long as the types of their arguments differ. This feature is useful when you want to have different versions of the same function; they should all be related. For example, you may want to have a constructor that takes lots of options, as well as one that is simple to use. Also, you may want to make functions that take different types. Examples include:

```
Rectangle::Rectangle(Point leftTop, Point rightBottom)
Rectangle::Rectangle(short top, short left, short bottom, short
right)
void TPort::MoveTo(short, short)
void TPort::MoveTo(Point)
TComplex TComplex::Add(TComplex)
TComplex TComplex::Add(int)
```

This can be very useful but also can cause problems.

**Don't unintentionally use the wrong argument type.** The biggest problem is the unintentional use of the wrong argument type when overloading, which defines a new function. If `TView` has a member function

```
TView::Print(const TPrintRecord *)
```

and you define a subclass where, intending to override this function, you declare

```
TMyView::Print(const TStdPrintRecord *)
```

or

```
TMyView::Print(TPrintRecord *)
```

or even

```
TMyView::print(const TPrintRecord *)
```

because you forgot the type of the original argument (or misspelled the name), C++ assumes you don't want to overload the original function and simply declares it as a

new function. You have *not* overridden the original; it's still available. The latest version of CFront will warn you about the first two cases, but not the last.

Also remember that, as mentioned earlier, the more variants a function has, the easier it is to call the wrong one unintentionally because the arguments you supply just happen to match another variant.

**Watch your overrides.** If you have an overloaded member function (whether virtual or not), and you override it in a derived class, then your override hides all overloaded variants of that member function, *not just the one you overrode*. Thus, if you want to override an overloaded member function, you must override *all* of the overloaded variants. C++ treats the overloaded function as a single entity; the scope resolution rule for C++ is to find the first class that has any function with that name defined, then look for a match based on argument type. The C++ team at AT&T believes this is the correct rule; their reasoning is that an overloaded set of functions is really just one function with a bunch of variants, and that you should not be naming functions with the same name unless they are really the same function.

An example that illustrates this behavior follows:

```
class A {
public:
       void Foo(long);
       void Foo(double);
};

class B: public A {
public:
       void Foo(double);
};

B bar;
bar.Foo(2);
```

The call actually winds up calling `B::Foo(double)` after coercing the integer argument to double.

On a positive note, CFront will warn you if you override some but not all of a set of overloaded member functions. For details, see the 1985-1989 paper discussed in the "Background Reading" sidebar, and the reference manual.

**Don't abuse function overloading.** Function overloading can be abused. Functions should not have the same name unless they basically perform the same operation, as in the preceding examples. If that is the case, then having the functions identically named can be a great help in reducing the number of things a programmer must remember.

## OPERATOR OVERLOADING

Another fun C++ feature is the ability to define operators for your own classes. If you define a fixed-point data type, C++ lets you define the standard arithmetic operators for it, which makes code a lot easier to read.

**Use operator overloading only where appropriate and clear.** Operator overloading also has tremendous potential for abuse. Defining the + operator for fixed-point numbers helps clarify code. Defining it as set union is also fairly clear. Defining => to mean "send a message" is crazy. Operator redefinition only helps when the new function is similar to the standard meaning of the operator; otherwise, it just confuses people. An example is C++'s streams facility, which redefines `< and >` as output and input operators. This confuses a lot of people.

**If you like, use functional syntax to call a base class's operator.** C++ occasionally delivers a pleasant surprise. One example is the syntax for calling overloaded operators. Of course, you can use the usual inline operator syntax; that's why C++ has operator overloading. The surprise is that you can also use functional syntax, which is sometimes essential, especially for calling a base class operator. Here is an example of a subclass operator using the base class's operator:

```
const TWindow& TWindow::operator=(const TView &v)
{
       this->TView::operator=(v);
       return *this;
}
```

(The explicit `this->` is actually unnecessary here but was included for clarity.) In this example, `TWindow` has a base class `TView`, and we want to be able to assign a `TView` to a `TWindow` by just copying the `TView` part and leaving the rest of `TWindow` alone. To do this, we want to use `TView`'s assignment operator. The functional notation here is the only way to do it. The pleasant surprise was that this notation is allowed.

## TYPE COERCION

**Use type coercion selectively.** Like so many C++ features, type coercion can either clarify or obfuscate your code. If a type coercion seems "natural," like the coercion between reals and integers, then providing a coercion function seems like a good idea. If the coercion is unusual or nonsensical, then the existence of a coercion function can make it very hard to figure out what's going on. In the latter case, you should define a conversion function that must be called explicitly.

In general, coercion operators are useful in a way similar to operator overloading, and the same guidelines make sense.

**Define type coercion rules for C++ to use.** C++ will automatically coerce one type to another, but only if there is a direct way of doing so. In other words, if a

coercion is defined from type A to type B, C++ will use it automatically where appropriate. It will *not* concatenate coercion operators if there is not a direct coercion. So even though a coercion may be defined from type A to type B, and type B to type C, C++ will not automatically coercion type A to type C (you can do it explicitly via casts, though).

There are two ways of defining type coercions for C++ to use: constructors and type coercion operators. They are appropriate under different circumstances. Note that since all coercion operators must be member functions of some class, it is not possible to define a coercion from one nonclass type to another nonclass type. Also note that if more than one function is defined to perform the same coercion, they cannot be used implicitly or via a cast, since an ambiguity exists as to which to call. They can still be invoked explicitly.

If you have a constructor with a prototype that looks like any of the following:

```
TargetClass::TargetClass(SourceType)
TargetClass::TargetClass(SourceType &)
TargetClass::TargetClass(const SourceType &)
```

then C++ will use it to convert from **`SourceType`** to **`TargetClass`** where appropriate. This form is useful when (1) the target type is a class (it can't be used for a primitive target type), and (2) the author of the target type wants to provide a coercion. If either of these conditions don't hold, you can use the second form of type coercion.

If the source type is a class, you can define a member operator function to perform the coercion. These operators have prototypes that look like

```
SourceClass::operator TargetType()
```

TargetType may be either a primitive type or a class. It need not be the name of a type; it can be any type specifier (as long as it does not contain **`array of []`** or **`function ()`** forms. Those must be handled via a typedef). This form is appropriate when the target type is not a class, or the source code for the target type is not available (that is, the coercion is being provided by someone else).

### ENCAPSULATION AND DATA-HIDING

**Make explicit use of public, private, protected.** C++ thoughtfully allows you to leave out the private keyword in several places. Don't: it decreases C++'s well-known clarity. Class definitions should always explicitly state the visibility of their members and/or base classes. Write it like this:

```
class TFoo: public TBar, private MBaz {
public:
       public members;

protected:
       protected members;

private:
       private members;
};
```

Your public interface should come before your protected interface, and since your private interface is only necessary to make the compiler happy, it should be last.

**Use no public or protected members that aren't functions.** Always make *all* member variables private. (It's OK to make functions protected or public. In fact, classes that don't are very boring.) You can provide access functions to get and set your variables if you want (although you should think about exporting a more abstract interface instead). If you're really concerned with performance, you can make those functions inline (but see "Inline Functions," earlier). Remember, don't compromise for the sake of performance *until you have numbers* to base your decision on!

**Understand what "protected" really means.** What the "protected" access mode means is not completely clear from Bjarne's various books and papers, so we will attempt to clarify the issue.

When a member of a class is declared protected, to clients of the class it is as if the member were private. Subclasses, however, can access the member as if it were declared private to them. This means that a subclass can access the member, but only as one of its own private fields. Specifically, it *cannot* access a protected field of its parent class via a pointer to the parent class, only via a pointer to itself (or a descendant). Here are some examples:

```
class A {
protected:
       void Bar();
};

class B: public A {
       void Foo();
};

class C: public B {
…
};

void B::Foo()
```

```
{
        A *pa;
        B *pb;
        C *pc;

        pa->Bar();          // Illegal: A::Bar() is "private"
        Bar();              // OK: "this" is of type B*
        pb->Bar();          // Also OK
        pc->Bar();          // Also OK
};
```

**Protect constructors for abstract base classes.** It's frequently useful to have a class that is not meant to be instantiated as an actual object, but only to be used as a base class for other classes. Examples include classes such as **`TApplication`** or **`TView`**. Such classes are called *abstract base classes.* If you want to enforce this status, you can make it impossible to instantiate such a class by making all of its constructors protected. In that case, the object cannot be created by itself, but only as part of a derived class.

In addition, there is a way to declare a member function abstract (that is, to require that it be overridden in descendants); this is called a *pure virtual function*. A class with such a member function cannot be instantiated, nor can any descendant class, unless all such functions are overridden. The syntax for this is as follows:

```
class Foo {
public:
        virtual void Bar() = 0;
};
```

You can also declare some (but not all) of the constructors for a class protected if you want those constructors to be used only by derived classes. The class can still be instantiated using the constructors that are public.

**As a shortcut, declare private base classes.** When you declare a base class private in C++, the derived class inherits all of its members as private members. This means that the derived class is *not* considered a subtype of the base class, even though it is a subclass. You cannot pass a pointer to an object of the derived class when a pointer to the base is expected. This lets you inherit the behavior of a class without inheriting its type signature.

Since the derived class is not a subtype, it doesn't have an "is-a" relationship with the base class. Why not just make it a member, then? This is what you would normally do. However, if you need to reexport most of the functionality of the base class, you would have to write wrapper functions in your derived class that turned around and called the member class functions. Instead, you can use this slimy shortcut.

By making the class a private base class, you don't inherit the type signature, but you do inherit the functionality, which can be made selectively visible without having to write wrapper functions. If A is a private base class of B, and B wants to make `A::Foo()` visible, then write the following in the declaration of B:

```
…
public:
        void A::Foo();

which makes Foo() visible to clients of B.
```

**Use friends sparingly.** Friend classes and functions are another C++ feature. Needless to say, they are a breach in the safety of types and in the integrity of the data abstraction provided by classes. If you have friends, you're probably doing something wrong (like taking frequent showers).

About the only time this feature should be used is when implementing binary operators that can't be member functions. Another circumstance is a set of tightly related classes (an example from Bjarne's book is matrices and vectors). Generally, however, avoid friend classes and functions.

**Hide implementation classes by declaring them incomplete.** Sometimes a public class (one that you export to clients) must refer to a class that is only used in your implementation. How do you avoid exposing the implementation class? If the only reference is a pointer, then you can declare the implementation class as an incomplete class:

```
class TImplementation;

class TInterface {
...
private:
        TImplementation *fHidden;
};
```

This also works if your member functions have arguments of type `TImplementation *` or `TImplementation &`. If you have actual `TImplementation` objects as fields, though, you must include the full declaration of `TImplementation`.

**Don't expose yourself.** The most important thing to remember is not to expose your implementation to either your clients or your subclasses (which are really just another kind of client). If you do so, you are tying the hands of those who must enhance your code.

It is very important to make sure that your class acts like a black box. The interface you export to clients and subclasses should reflect precisely what they need to know and nothing more. You should ask yourself, for every member function you export (remember, you're not exporting any data members, right?), "Does my client (or subclass) really need to know this, or could I recast the interface to reveal less?"

If you find that the interface to your class consists mostly of functions to get and set your private data members, you should ask yourself whether your object is really defining an abstract enough interface. The key is to think about the abstraction that your object represents and how clients view and use that abstraction, not how it is implemented. This is possibly *the* hardest thing to do in object-based design, but is also one of the biggest advantages and has the biggest long-term payoff.

**See Alan Snyder's paper.** For an excellent discussion of the issues involved in data abstraction, encapsulation, and typing, see Alan Snyder's paper "Encapsulation and Inheritance in object-based Programming Languages" in the 1986 OOPSLA proceedings.

**VIRTUAL FUNCTIONS**

**(Almost) all member functions should be virtual.** Virtual functions are pretty inexpensive. Because of this, any class that is intended to be used in a polymorphic fashion (that is, a pointer to a subclass may be passed where a pointer to the class is expected) should have all of its functions virtual. It's hard to guess in advance which functions may be overridden in the future (although private functions can't be, and so need not be virtual).

You should only use nonvirtual functions where you are very sure that the class (or this particular aspect of it) will *never have a subclass*. An example is a fixed-point data type, which is self-contained, or a graphics point, or other similar classes.

The assignment operator is also a special case. Assignment isn't inherited like other operators. If you do not define an assignment operator, one is automatically defined for you; it consists of calls to the assignment operators of all of your base classes and members (this is discussed in the 1985-1989 paper mentioned in the sidebar "Background Reading"). It's OK to make your assignment operator virtual, but it's only useful under rather specialized circumstances. A virtual function call will be generated for a virtual assignment operator only when the left-hand side of an assignment is a reference or a dereferenced pointer.

**(Almost) all destructors should be virtual.** What the !@&#%! is a virtual destructor, you ask? And well you might, because this is actually something you should never have had to worry about. What do you think happens here?

```
class A {
      ~A();
};

class B: A {
      ~B();
};

A *foo = new B;
delete foo;
```

**`B::~B`** gets called, then **`A::~A`**, right? Wrong! Only **`A::~A`** gets called! Isn't that special? For the right thing to happen, you must declare your destructors virtual. For example,

```
virtual ~A();
virtual ~B();
```

If you do this, the right destructors will get called. Needless to say, *any* class that has a virtual member function, inherits one, or is used in a polymorphic fashion *must* have its destructor declared virtual, or horrible things will happen.

**Be careful when you call virtual functions in constructors and destructors.** If you call a virtual function from a constructor, be aware that the version of the function that corresponds to the constructor will be called, *not* the version that would normally be called. For example, if A has a method **`foo`**, B is a subclass of A, and B overrides **`foo`**, a call to **`foo`** from A's constructor calls **`A::foo`**, not **`B::foo`**. A call to **`foo`** from B's constructor does call **`B::foo`**. This is sufficiently confusing that it is best not to call a virtual function from a constructor at all. Naturally, this only applies to virtual functions of the object whose constructor is running; virtual functions of other objects (including those of the same class) are perfectly fine (unless they in turn call one of your virtual functions).

If you think about it, it doesn't make sense to call virtual functions from constructors and destructors. Since base class constructors are called before derived class constructors, and base class destructors are called after derived class destructors, the object is in a partially valid state. If a virtual function overridden in a derived class is called from the base class constructor, it may access derived class features that have not been initialized. Similarly, if it is called from the destructor, it may access features that have already been destroyed.

**Use virtual functions the right way.** If you are coming from the non-object-based programming world, a word about use of virtual functions might be in order. Virtual functions should be used whenever you want to have more than one implementation of the same abstract class. They allow the system to defer the decision on which function to execute until run time.

The right way to use virtual functions is to structure them around well-defined abstractions. If someone is to override a virtual function, they must have a clear definition of what the function does, even if they only call the inherited version after a little bit of processing.

The *wrong* way to use virtual functions is via a "come-from" mechanism like some Macintosh trap patches. Don't override a virtual function because "I know it's called from over here with these parameters." Needless to say, this wreaks havoc with the data abstractions that are one of the major benefits of object-based programming. This is why the function must have a definition that is clear in terms of the object it belongs to, without any reference to its possible callers. If you override a function, the override must make sense in terms of the definition of the function itself.

## MULTIPLE INHERITANCE

Multiple inheritance is a fairly new feature in object-based languages. To understand it better, look at Figure 1. In the single inheritance class hierarchy on the left, each class has only one parent. By contrast, in the multiple inheritance class hierarchy on the right, a class can have more than one parent. Note, for instance, that `TAirplane` inherits from both `MFlyingObject` and `MVehicle`.

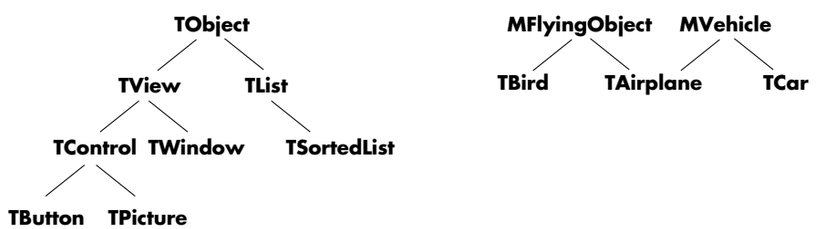



**Figure 1.**
**Single Inheritance vs. Multiple Inheritance**

**Use in a controlled fashion.** With multiple inheritance, there is great potential for designing a confusing class hierarchy that resembles a spaghetti bowl. Here are some guidelines for use of multiple inheritance.

We start by artificially partitioning classes into two categories: base classes and mix-in classes. To distinguish the two, base class names begin with *T* (for example, **`TView`**), and mix-in class names begin with *M* (for example, **`MEditable`**). Base classes represent fundamental functional objects (like a car); mix-ins represent optional functionality (like power steering).

The first guideline is: A class can inherit from *zero or one* base classes, plus *zero or more* mix-in classes. If a class does not inherit from a base class, it probably should be a mix-in class (though not always, especially if it is at the root of a hierarchy).

The second guideline is: A class that inherits from a base class is itself a base class; it cannot be a mix-in class. Mix-in classes can only inherit from other mix-in classes.

The net effect of these two rules is that the base classes form a conventional, tree-structured inheritance hierarchy rather than an arbitrary acyclic graph. This makes the base class hierarchy much easier to understand. Mix-ins then become add-in "options" that do not fundamentally alter the inheritance hierarchy.

Like all guidelines, this one is not meant to be hard and fast. Multiple inheritance can and should be used in other ways as well if it makes sense. The fundamental thing to keep in mind is that people (including programmers) are better at understanding regular structures than arbitrary acyclic graphs.

As part of multiple inheritance, C++ contains a new feature called virtual base classes. The trouble is, if both B and C are subclasses of A, and D has both B and C as base classes, then D will have two A's if A is not virtual, but only one A if it is. This is a very confusing situation, and no matter which alternative you choose, programmers will have a hard time understanding it. To avoid getting into a situation like this, follow the preceding guidelines for use of multiple inheritance. And incidentally, using virtual bases presents another problem: once you have a pointer to a virtual base, there's no way to convert it back into a pointer to its enclosing class.

**It's OK to have multiple occurrences of a base.** Sometimes the same base class (it should be a mix–in) will occur more than once as an ancestor of a class. It doesn't hurt to have a base class twice (aside from wasting space because of multiple pointers to the virtual function table) and if you need to cast back from the base class pointer to something else you may not have a choice, but it's really only useful to have a base class twice if data members are associated with it.

## PART 3: DESIGN ISSUES

In this part you'll find a discussion of general problems of programming in C++.

### WORKING IN A VALUE-BASED LANGUAGE

C++ has a different object model from Object Pascal or Smalltalk. The most fundamental difference is that whereas Object Pascal and Smalltalk are reference based (that is, like Lisp, assignment means copying a pointer), C++ is value based. By this we mean that classes in C++ are treated just like primitive types, whereas in Object Pascal objects are treated very differently from primitive types. This is actually a benefit, since all types in a C++ program are handled in the same style, as opposed to the multiple styles in Object Pascal. (Smalltalk, like C++, is also self-consistent.) There are some implications for your C++ programming style, however.

**Don't use pointers unless you mean it.** Pointers should be used in C++ in the same way you would use them in plain C or plain Pascal; that is, when you really want multiple references to the same object, or a dynamic data structure. If you really just want to pass something by reference to avoid copying, then you can use a reference rather than a pointer (see below). In fact, you can even pass a class by value if the copying overhead isn't too high and you don't care about polymorphism (for example, the class has no virtual functions).

**Don't allocate storage unless you must.** In a reference-based language like Object Pascal or Smalltalk, all objects must be heap allocated. In C++, it's better to treat values the same way you would in C. For example, instead of defining a `Clone` operator, overload the assignment operator; instead of allocating and returning an object, have the caller pass one in by reference and set it. This allows your classes to be treated just like primitive types, and in the same style. In general, leave storage allocation up to the class client.

By doing so, you can make use of one of C++'s unique features: the ability to have automatic and static objects, and objects as members of classes. No matter how clever or efficient a storage allocator we have, it can never be as fast as allocating an object on the stack, or as part of another object. If an object can be local to a function, there is no storage allocation overhead. Many objects have very localized scope and do not need to be allocated on the heap.

There is one exception to the rule about allocating an object and returning a pointer: you must do this when the type of the returned object may vary. Anytime a function must choose what type of object to return, the function must allocate the object, not the caller.

It's still all right for the caller to allocate storage even when the type of object being passed in may vary, since references, like pointers, can be used polymorphically (that is, you can specify a `TSubFoo&` to an argument of type `TFoo&`). The key question is

whether the caller or the function must determine the type. In the former case, leave allocation to the client; in the latter, the function must allocate the object on the heap and return it.

**Summary: pretend everything is a primitive.** In summary, you should design your classes so that using them is just like using a primitive type in C. This will let the client use them in a style that is "natural" for C. In cases where you wish to avoid copying, pass arguments by reference. Use pointers only when you want a truly dynamic data structure, or when polymorphism demands it (note that references allow for polymorphism also, since they are really just a different kind of pointer).

## BACKGROUND READING

**Bring yourself up to date.** You've just finished reading Bjarne Stroustrup's book *The C++ Programming Language* (Prentice-Hall, 1987) and you're feeling pretty smug. You've finally got C++ nailed.

Wrong.

You still have one more reference to read: the paper "The Evolution of C++: 1985 to 1989." This paper is included with the AT&T C++ *Selected Readings* manual, which is available in conjunction with MPW C++.

At least one statement made in earlier versions of the paper (which were titled "The Evolution of C++: 1985 to 1987") is wrong. The order of execution of base class and member constructors and destructors is determined by their declaration order, not by the order in which calls are made to such constructors in your constructor's header. This is a change to the language that was made after the 1985-1987 paper was written; see the 1985-1989 version for a full discussion.

Other books that give a good discussion of features that are new in C++ 2.0 are *The C++ Primer* by Lippman (Addison-Wesley, 1989) and *The C++ Answer Book* by Hansen (Addison-Wesley, 1990). The latter book not only discusses 2.0 but also gives solutions to all of the problems in Stroustrup's book.

Finally, look for Bjarne's own updated manual, *The Annotated C++ Reference Manual* (with Ellis, Addison-Wesley) to be published later this year.

**Read up on ANSI C**. If you were whelped on good ole Kernighan and Ritchie C, you may have a few surprises in store for you. There have been several changes to the language as part of the ANSI standardization process. If you learned C a while back, it might be a good idea to brush up on ANSI C. We highly recommend the second edition of Kernighan and Ritchie (*The C Programming Language* , Prentice-Hall, 1989), which has appendixes that detail the differences between the original language and the ANSI version. Another good book is C: *A Reference Manual*, 2nd ed., by Harbison and Steele (Prentice-Hall, 1987). For purists, the ANSI C draft and rationale are available from ANSI itself (*Draft Proposed American National Standard for Information Systems—Programming Language C*, 1988, ANSI Doc No X3J11/88-159).

**Study object-based design**. Doing a good job of object-based software design requires more than just learning an object-based language. The whole point of object-based languages is to permit a different approach to software design. This approach takes time and energy to learn. Without spending that time and energy, it's not possible to gain the full benefits of the approach.

That's why you should read *Abstraction and Specification in Program Development* by Liskov (McGraw-Hill, 1987) and *Object-Oriented Software Construction* by Meyer (Prentice-Hall, 1988). The first book does not discuss object-based design per se, but it does cover the topic of data abstraction, an important component of object-based design, very well. The second book is a little pedantic in

places, but has many, many good suggestions and ideas in it. Reading both is hard work (especially since both are based on obscure languages), but will help you a great deal.

One example of an issue that *Object-Oriented Software Construction* covers quite well is the question of whether to use a class as a base (inherit from it) or a member (include it as a field). As Bertrand Meyer notes, the distinction is whether the new class can act as an instance of its base class (that is, it "is-a" object of that type) or uses an instance of the class (it "has-a" object of that type). For example, an automobile "is-a" vehicle, but it "has-a" engine. For good discussions of this and other issues, read the book (this particular discussion starts on page 333).

Brian Wilkerson's "How to Design an Object-Based Application," in this issue of develop, is a good language-independent introduction to object-based design. You should find the design techniques presented in his article useful in your own work.

Taking the time to learn how to design with objects may be painful (after all, we're all working as hard as we can already), but it can make a big difference in the quality of the system when it's done. Every extra minute you take to improve the design now will pay off in easier maintenance and enhancements later.

## POINTERS VERSUS REFERENCES

C++ provides two very similar mechanisms for passing references to entities. One is the familiar pointer from classic C; the other is a new concept, the reference. Pointers are declared as follows:

```
TFoo *fooPtr;
```

But references are declared like this:

```
TFoo &fooRef;
```

A pointer must be dereferenced to access what it points to, but a reference can be used as is, and acts as a synonym for the object it refers to, both for fetching and storing. This makes it similar to other highly refined mechanisms, such as FORTRAN's **`equivalence`** statement (or **`VAR`** parameters in Pascal). The entity to which a reference refers may only be set when the reference is created; in this respect it is somewhat like a const pointer that gets a **`virtual *`** put in front of it wherever it is used, and puts a **`virtual &`** in front of the expression from which it is initialized. Here are two illustrative examples:

```
void Bump(int *ip)
{
        *ip += 1;
```

```
}
Bump(&j);

void BumpR(int &i)
{
       i += 1;
}
BumpR(j);
```

There are certain circumstances where references are mandatory—for example, overloading the assignment operator. In other cases, either a pointer or a reference can be used. The question is, which should be used when?

References should be used when a parameter is to be passed "by reference," as in Pascal. This means that the called function is going to forget about the argument as soon as it returns. A regular reference should be used if you are going to modify the argument (**`TFoo &`**), and a const reference should be used if you are not going to modify it but don't want the overhead of call by value (**`const TFoo &`**).

Pointers should be used when the function you are calling is going to retain a reference (an alias) to the object you are passing in, such as when you are constructing a dynamic data structure. An example is putting an object into a MacApp **`TList`**: the **`TList`** retains a pointer to your copy of the object. The explicit use of pointers lets the reader know that aliasing is occurring.

By using pointers and references appropriately, you can increase the readability of your code by giving the reader hints as to what is going on.

**PORTABILITY**

**Don't make assumptions.** The Macintosh is the best personal computer in the world, but there are times when you'll want to run your code on a different machine. For example, you might have access to a CRAY or a VAX. Don't make assumptions that are only valid for the 680x0 family of processors. For example:

- Don't assume that **`int`** and **`long`** are the same size.
- Don't assume that **`longs`**, **`floats`**, **`doubles`**, or **`long doubles`** can be at any even address.
- Don't assume you know the memory layout of a data type.
- Especially don't assume you know how structs or classes are laid out in memory, or that they can be written to a data file as is.
- Don't assume pointers and integers are interchangeable. Use **`void *`** if you want an untyped pointer, not **`char *`**.
- Don't assume you know how the calling conventions are implemented, or indeed any detail of the language implementation or run time.

ANSI specifies the following about C's built in types. *This is all you can safely assume:*

- **`unsigned chars`** can hold at least **`0`** to **`255.`** They may hold more.
- **`signed chars`** can hold **`–127`** to **`+127.`** They may hold more.
- **`chars`** may be either **`unsigned chars`** or **`signed chars.`** You can't assume either. Therefore, don't use **`char`** unless you don't care about sign extension.
- **`shorts`** can hold at least **`–32,767`** to **`32,767 (signed)`** or **`0`** to **`65,535 (unsigned).`**
- **`longs`** can hold at least **`–2,147,483,647`** to **`2,147,483,647 (signed)`** or **`0`** to **`4,294,967,295 (unsigned).`**
- **`ints`** can hold at least **`–32,767`** to **`32,767 (signed)`** or **`0`** to **`65,535 (unsigned).`** In other words, **`ints`** *cannot be counted on to hold any more than a* **`short`**. **`int`** is an appropriate type to use if a **`short`** would be big enough but you would like to use the processor's "natural" word size to improve efficiency (on some machines, a 32-bit operation is more efficient than a 16-bit operation because there is no need to do masking). *If you need something larger than a* **`short`** *can hold, you must specify* **`long`**.

If you need exact information, you can use the symbols defined in limits.h. or float.h. Remember, though, that the values of these symbols can change from processor to processor or compiler to compiler, within the limits just defined (for more information, see the ANSI C specification).

It's very easy to write nonportable code, and it takes some vigilance to avoid it. It's well worth the effort, however, the first time you port to a different processor, or try to use a different compiler.

**Pick a canonical format for messages and data files.** Remember that AppleTalk networks connect to non-Apple computers such as the Intel-8x86 based MS-DOS machines. Thus, if you write or read any data in a context where it might go to or come from a different CPU, you have to worry about formats. Such situations include reading or writing disk files, or sending data over a network (or even over NuBus). The other CPU might even have a different byte order! The only solution to this problem is to pick a canonical format for your messages or data files.

Just because you have a canonical format doesn't mean you must pay a big overhead every time you access your data. An alternative is to perform the translation to or from canonical format at a predetermined time. For example, outline fonts might have a certain canonical format, which may not be convenient for a particular processor to deal with. However, they can certainly be converted to a convenient local format when the font is installed.

**Don't use (gasp!) naked C types.** Another way to make your life miserable is to use primitive C data types in your declarations. This is a bad idea, since if the implementation ever changes you have to do a lot of editing by hand. It's much better to declare a type (via class definition or typedef) that represents the abstract concept you want to represent, then phrase your declarations that way. This lets you change your implementation by simply editing the original type definition. Think of these types as giving your data physical units, like kilograms or meters/second. This prevents you from accidentally assigning a length to a variable with type **`Kilogram`**, catching more errors at compile time.

So instead of

```
long time;
short mouseX;
char *menuName;
```

use (for example)

```
typedef long TimeStamp;
typedef short Coordinate;
class TString { … };
.
.
.
TimeStamp time;
Coordinate mouseX;
TString menuName;
```

It's OK to use a raw C type under certain circumstances, such as when the quantity is machine dependent, or when it can be characterized as (for example) a small integer. Otherwise, though, it's best to give yourself flexibility down the road.

Two ANSI C header files, StdDef.h and Limits.h, contain useful definitions. Here are two of the more useful ones:

**`size_t`**

The type returned by the built-in C **`sizeof`** function. This is useful for representing the sizes of things.

**`ptrdiff_t`**

A type that can represent the difference between any two pointers.

The astute reader has noticed that these names do not conform to our guidelines. In the interest of clarity, however, we deem it better to use the names as defined by the ANSI C committee.

Another item worthy of note: if a data type is unsigned, declare it unsigned; this helps avoid nasty bugs down the road.

### ERROR REPORTING

Returned error codes are (ironically) a very error-prone technique. MacApp's standard is to use exceptions: a structured technique for reporting exceptions back to a function's callers.

Unfortunately, the exception scheme does not handle an important case: an exception that occurs in a constructor. Handling this properly requires compiler support, since any base class and/or member constructors that have already been called must have their corresponding destructors called; only the compiler can know this. Until we get an official C++ exception scheme, you must handle this problem on a case-by-case basis.

Signaling an exception in a destructor is not a good idea, since any destructive behavior that has already taken place probably cannot be reversed. Don't do it.

## CONCLUSION

We've covered a lot of ground in this article. Don't feel bad if you didn't understand every point we tried to make. It took us years of working with objects before we figured out what virtual base classes were good for! As you use C++ in your own work, think about the rules we've laid down and see if they aren't applicable to your own situation. Come back and read these guidelines every so often, just to refresh your memory. And finally, don't be shy about formulating some guidelines of your own!

# DEMYSTIFYING THE GS/OS CACHE

*GS/OS has given Apple IIgs users an important capability: caching. To the newly initiated, the mysteries of the GS/OS cache may still seem profound. This article clarifies the basics of the caching algorithm and offers useful pointers for working safely and efficiently.*

MATT DEATHERAGE

In the past couple of years, the Apple IIgs system software has grown by leaps and bounds. With the introduction of thousands of new features and capabilities, one very important one often goes unnoticed. GS/OS is the first Apple II operating system of any kind to provide a comprehensive caching implementation.

Although the cache is an important part of GS/OS, its purpose and nature are sometimes misunderstood. Some people create huge caches in the hope that peripherals will start behaving like RAM disks. Others set the cache size to zero because they think it's wasting time and memory. Some developers mistakenly believe their program has absolute control over caching, which is only true if they're writing device drivers. This article reaches into the murky depths of your IIgs's memory and allows you to examine the cache in the light you would normally use, say, to read develop by.

## CACHE FUNDAMENTALS

Let's start at the beginning, with the basics. What's a cache? How is it managed? How big is it? How does the GS/OS cache differ from that of the Macintosh? Read on.

### WHAT'S A CACHE?

A cache is a part of memory in which the operating system can keep a spare copy of information read from and written to a device, in an attempt to decrease disk access time.

Many people use RAM disks to decrease disk access time. Because the operating system doesn't actually have to physically manipulate any media to access information on a RAM disk, the information is saved and retrieved at a very nice clip, exceeded only by the speed at which the operating system can read information already in main memory. In a caching algorithm, that is exactly what happens.



**MATT DEATHERAGE** says, "You always want what you can't get," and the thing he wants most is sleep. This desire probably started in his college days while studying industrial engineering at the University of Oklahoma. In his far too many waking hours he's an Apple II DTS engineer, a self-proclaimed nonhardware person. "Eat lots of toast" is his motto and the ideal to which he aspires (thanks to his friend Robert Thurman). When he's not gobbling toast or snoozing, he can be found writing music (on the Apple IIGS, naturally) or playing the piano, the clarinet, or the bass clarinet. He isn't married, as far as he can tell. •

When information is read from a device, the operating system reads from the physical media and returns the information to the caller who asked for it. Additionally, the operating system keeps a spare copy of the information in the cache. The next time a caller requests information, the operating system first looks in the cache to see if it already has it from a previous media access. If it's there, the operating system simply moves the data into the caller's buffer, completing the call without having to actually read anything from disk.

Information to be written can also be cached. The operating system writes the information to the disk and also writes it to the cache. This method, called a write-through cache, ensures that the information in the cache is never more recent than the information on the disk. GS/OS uses a write-through cache except in one special situation—during a write-deferral session.

### THE CACHE MANAGER AND WHAT IT CACHES

In GS/OS, several distinct managers handle different parts of the operating system or environment. Many of these managers (the Loader and the Device Manager, for example) are familiar to application and driver authors. Another, more obscure manager takes care of the cache; plainly enough, it's called the Cache Manager. The Cache Manager handles the storage of the cache and all requests involving it.

The GS/OS Cache Manager only caches one thing: blocks. Blocks are what both generated and loaded drivers read from block devices. They can be traditional ProDOS-sized 512-byte blocks from traditional ProDOS devices; they can be 512-byte blocks from nontraditional ProDOS devices, such as network volumes or nondisk devices; or they can be odd-sized blocks from odd peripherals. If you had a GS/OS loaded driver to read 981-byte blocks from a 20-gigahertz, 35-terabyte RAM disk hooked to your Apple IIgs through a slot-based card, GS/OS could cache those blocks. (Whether or not any file system translators in the system could use the device is another story.) The Cache Manager does not cache anything having to do with printers, modems, or other character devices.

### THE SIZE OF THE CACHE

The size of the cache is determined by a battery RAM (BRAM) parameter. When the Cache Manager is initialized as part of the GS/OS boot process, it retrieves this value from the battery backed-up RAM and adjusts so that the given value is taken as the *maximum* size to which the cache can grow. The BRAM parameter is currently number $81 and represents the cache size in 32K increments (a value of 2 would indicate 64K). You should know that the location and interpretation of the cache size parameter are not guaranteed at this point, and relying on the location or interpretation of the parameter might get you into trouble. When they are

guaranteed, Apple II Developer Technical Support will release a Technical Note detailing this point, or the *GS/OS Reference* will be revised to document it.

The size of the cache is not an exact number but an exact maximum, and it is that only when nonzero. If the user has set the cache size to nonzero, that value (which will always be an increment of 32K) is the maximum cache size. If the user has set the cache size to zero, GS/OS allocates a 16K cache for system purposes—so that system components such as file system translators (FSTs) and drivers (both generated and loaded) can take advantage of the speed increase the cache provides. There is no way to completely turn off caching under GS/OS.

When the cache is initialized, it is empty and has zero size. As blocks are added to it, it grows as necessary to accommodate the increased use until the maximum size is reached. If no one ever asks for blocks to be cached, the cache remains empty and occupies no memory.

### WHAT HAPPENS WHEN THE CACHE IS FULL

Having a maximum cache size implies that there can be a problem — what happens when nothing else can be written to the cache because there is no room in cache memory for more information? How does GS/OS behave when there's no room in the Cache Inn?

The operating system, through the cache algorithm, has a few options for handling the possibility that the cache will be full. The most obvious option is for the algorithm to shrug its bit-encoded shoulders and say, "Well, the cache is full, so nothing else will be written to it." Information that comes knocking at the door thereafter is written to disk only and read from disk each time it's needed.

Most caching algorithms, including the one used by GS/OS, are a little bolder. They attempt to identify which blocks in the cache have actually been used and to keep those blocks in the cache, simultaneously removing from the cache the blocks that have not been used. The rationale for this is that if a block hasn't been read from the cache in a long time, it's probable that no one's going to want to read it again for a while, and the cache can be better used by a new block. For example, when GS/OS is booted, the file START.GS.OS is read from the disk. Suppose, for the sake of argument, that this file's information was placed in the cache. This file isn't likely to be read again by the operating system or by any application unless GS/OS has to be reloaded, which under version 5.0 and later never happens. This file could, therefore, be sitting in the cache, taking up valuable cache space that subsequently can't be used by directory blocks, bitmap blocks, or even program files that will be needed repeatedly, such as the Finder or APW commands. The strategy used by most caching algorithms tries to keep space in the cache free for use by directory blocks, bitmap blocks, and program files that will be needed repeatedly.

One popular caching algorithm keeps all the blocks in a chain with the most-recently used block at the beginning of the chain and the block not used for the longest time at the end. When there's no more room in the cache, the blocks at the end of the chain are removed, making room for new blocks. This algorithm, straightforwardly called a least-recently used (LRU) caching algorithm, is what GS/OS uses.

Some blocks, however, can't be kicked out of the cache by the LRU algorithm. These are the blocks placed in the cache by a write-deferral session. When such a session is in progress, some or all of the information written to files is write-deferred, which means GS/OS keeps the information around in the cache instead of taking the time to write it all to physical media. This is accomplished in different ways for different file systems; each file system translator behaves in the way it can achieve the best performance during a session. For example, the ProDOS FST writes actual data to the disk but not system-level information such as directories or bitmap blocks.

If any write-deferred blocks are in the cache, the Cache Manager will not purge them to make room for new blocks until they have been written to disk. If the cache fills up with nonpurgeable blocks and another nonpurgeable block must be added, a feature known as AutoFlush, which was new to System Software 5.0, takes over, stops the session, and flushes all write-deferred blocks to disk so that new ones can be added. This has the effect of breaking a session into lots of mini-sessions, each exactly long enough to get the best possible use from the cache.

Blocks in the cache can also be deleted from it by another method besides the LRU algorithm. If a driver detects a disk-switched condition, which normally means that an on-line volume has been taken off-line, it makes the **`SET_DISKSW`**.System Service call, which enables the file system translators to remove all blocks belonging to the switched disk from the cache so that no blocks are in the cache for volumes that aren't currently on-line. This ensures that no one accidentally reads from the device and gets an old block from the cache, or that no blocks are in the cache for volumes that aren't currently on-line.

### DIFFERENCES FROM THE MACINTOSH

The GS/OS Cache Manager is different from the caching implementation on the Macintosh in two important ways. The first difference is in the way memory is allocated. On the Macintosh, if the user sets the disk cache size to 128K, a 128K block of memory stays allocated for the disk cache unless someone resizes the cache. Under GS/OS, memory for cached blocks is allocated as needed up to the maximum set by the user. The other difference between GS/OS and Macintosh is in the size of the blocks that can be cached. The GS/OS cache can handle a block of any size; if a device deals in 2048-byte blocks—as some CD-ROM discs do—GS/OS will cache a 2048-byte block. The Macintosh Cache Manager, on the other hand, can only cache 512-byte blocks.

### HOW APPLICATIONS REQUEST TO USE THE CACHE

Applications—or desk accessories, inits, or anyone who makes GS/OS system calls—request to use the cache through cache-related parameters to the system calls. Specifically, applications can ask that data read from files be cached by using the **`cachePriority`** field of class one (and only class one) **`Read`** and **`Write`** calls.

### CACHING CAN BE REQUESTED BY CLASS ONE CALLS

An application requests caching by setting the **`cachePriority`** field in the GS/OS parameter block of **`Read`** and **`Write`** calls. (These are the only two GS/OS system calls with this field in the parameter block. The system calls **`DRead`** and **`DWrite`** do not have a **`cachePriority`** field, as the Device Manager always disables caching of blocks read or written through these calls.)

A value of $0000 for the word-length parameter in the **`cachePriority`** field is the norm (and the default if this parameter is omitted) and indicates that blocks involved in this call should not be cached. A value of $0001 identifies the blocks involved as candidates for caching. Only a value of $0001 in this field will cause files read at the application level to be considered for caching.

The fact that a caching request has been made by an application doesn't mean that it will be fulfilled. Applications do not call the Cache Manager; other system components do. Those components (file system translators and drivers) may deny the request when it doesn't make sense or is dangerous for the file's blocks to be cached.

### CACHING CANNOT BE REQUESTED BY CLASS ZERO CALLS

Applications written using class zero calls (including older ProDOS 16 applications) cannot request that their files be placed in the cache. To make such a request, the application must be changed to use class one calls. At first this seems a little harsh and arbitrary, but it's quite the opposite.

GS/OS could treat class zero file calls one of two ways—it could automatically cache everything, or automatically cache nothing. If it cached everything, reading any file larger than the maximum cache size would flush all the cached blocks, no matter how frequently they were used. This would force them all to be reread from disk the next time they were needed. Such a method is normally a grand waste of time, since most files on the Apple IIgs are typically read once. Most programs still follow the old Apple II method of "read the file, modify it, and write it back," and writing such files to the cache only serves to slow things down while the flushed blocks are reread from disk. Better methods for file manipulation exist these days, but the system was designed for maximum performance using the methods of the time.

An application knows best which files it will be reading from disk often enough to benefit from caching. Such decisions are often reached after long performance studies of cached reads vs. purgeable handles in memory. For more pointers on application-level caching, see GS/OS Technical Note #3, Pointers on Caching.

### WRITE-DEFERRAL SESSIONS AND THE CACHE

Applications can also speed up disk-intensive operations through the use of write-deferral sessions. An application begins such a session by giving the GS/OS system call **BeginSession**. This places all subsequent writes in a mode where some or all of the blocks written to disk are placed only in the cache and not on the media.

Deferred blocks are then written to the media when the **EndSession** call is made. If the **EndSession** call is not made or if it's made before the files being written to are closed, some of the blocks for the files written may be on the disk while others are in the cache only. This damages disks in most file systems as fast as fingerprints on the media. Be sure to *always* issue **EndSession** calls on every exit path after a **BeginSession** call to prevent blocks written from being only in the cache and not on the disk. And be sure to close all open files before calling **EndSession**, since the operating system can get caught with write-deferred blocks in open files if the session is ended and the disk is ejected before the files are closed.

## HOW CACHE REQUESTS ARE FULFILLED

GS/OS attempts to make sure that although an application can cache blocks if it so chooses, by default the most intelligent use possible of the caching capabilities will be made. It does this by filtering out possibly bogus requests for caching at several levels. When an application makes a request to use the cache, the request is filtered through drivers and/or file system translators. These agents of the Cache Manager evaluate requests to make sure that the most intelligent possible use of the caching capability is made. If an application's request to use the cache is found to be valid, one of these system components calls a System Service routine to add blocks to the cache.

### THE SYSTEM SERVICE CALLS

The ultimate caching authority is at the System Service call level. System Service calls are used by drivers and file system translators to access the routines that act on cache requests. These calls are available only to drivers and file system translators; they are not available to applications. System Service calls are accessed through vectors in the $01/FC00 page and are described in the *GS/OS Reference*, volume 2, chapter 12.

The following four System Service calls—to add blocks, find blocks in the cache, move blocks in and out of the cache, and remove all purgeable blocks in the cache belonging to a switched disk—are the only calls that can be made by drivers. Other cache-related System Service calls enable the system to delete blocks and volumes from the cache, but these are not documented in the *GS/OS Reference* and can only be used by file system translators and other Apple-supplied system components.

**CACHE_ADD_BLK** ($01/FC08) is the System Service call for a routine that adds blocks to the cache. This call takes several parameters on GS/OS direct page (which is available to drivers and file system translators but not to applications), including

information to identify the block by volume, device number, size and block number, and whether or not a write-deferral session is in progress. The Cache Manager is called, and it adds the block to the cache with no filtering. If necessary, the entire purgeable cache (the cache size minus all nonpurgeable write-deferred blocks) will be purged to add the block. The call will return with an error if a block could not be added to the cache, most likely because the entire purgeable cache was smaller than the block to be cached.

**`CACHE_FIND_BLK`** ($01/FC04) is the System Service call for a routine that finds blocks in the cache. It will search the cache for a specified block, returning a pointer to it if it is found. The cache can be searched by device number, so a device driver can find all blocks it has cached, or by volume ID, so that a file system translator can find all blocks it has cached (when a write-deferral session is in progress).

**`MOVE_INFO`** is the System Service call for a memory-moving routine. This routine is called by drivers and file system translators to move data in and out of the cache.

**`SET_DISKSW`** is the System Service call for a routine that kicks out all purgeable blocks in the cache belonging to a switched disk. If **`SET_DISKSW`** is called while a write-deferral session is in progress involving closed files on that device, GS/OS puts up a dialog box warning that the disk was prematurely ejected and that the disk's structure may be damaged.

There are two main parts of GS/OS that can use these System Service calls — drivers and file system translators.

### THE ROLE PLAYED BY DRIVERS

Drivers filter cache requests passed on from the application and file system translator levels. With every **`Driver_Read`** and **`Driver_Write`** command a driver gets from an application, it is passed instructions to take one of three possible caching actions:

- If the **`cachePriority`** word (on GS/OS direct page) is $0000, the block being read or written should not be cached.
- If **`cachePriority`** is nonzero with the high bit clear ($0001—$7FFF), the block should be cached as a normal, purgeable block.
- If **`cachePriority`** has the high bit set ($8000-$FFFF), the block should be cached as a deferred nonpurgeable block. This means a write-deferral session is in progress. In this case, which is only valid for **`Driver_Write`** calls (there are no read-deferral sessions in GS/OS), the driver should write the requested block only to the cache and not to the physical media. The end of the session will result in the driver being called again to write all the cached blocks to the physical media.

There are instances, such as identifying volumes, in which a file system translator might wish to force a read from physical media rather than from the cache (if the block is in it). In these cases, the FST ID number on GS/OS direct page has the high bit set, telling drivers not to read the block from the cache.

A driver is not obligated to cache blocks when requested to, but instead can decide to disable caching completely or selectively. In some cases, the driver should refuse to cache *any* blocks. For example, this would be appropriate if the driver is for a device that cannot identify a disk-switched condition with any degree of reliability, as with a 5.25" disk. Since the driver can't call **`SET_DISKSW`** until it notices the disk has been switched—which could be well after the fact—a block in the cache for such a device might not be deleted when it should be, and thus the driver should refuse to cache any blocks in the first place. Or for example, if a driver can actually read from the media faster than a block can be returned from the cache, it should refuse to cache blocks.

### THE ROLE PLAYED BY FILE SYSTEM TRANSLATORS

File system translators can initiate cache requests themselves, and can filter cache requests passed on from the application level.

When a file is opened and read or written, not all of the requests to a driver for information from the physical media are requests for data blocks from the file in question. Many of the read and write requests are for directories, file-system-specific data structures—such as key blocks and index blocks in ProDOS—and bitmap blocks. File system translators need these data structures repeatedly during file operations, and may ask that the blocks involved be cached. The ProDOS FST does this, caching all blocks it reads and writes that aren't passed on to the application. If the caching can be done, the file system translator gets much faster response time the next time it needs those blocks. On ProDOS disks, the caching of the volume directory and volume bitmap give tremendous speed increases since every file opening causes a read of the volume directory, and every write operation causes a read and write of the volume bitmap.

The file system translator may also cause caching of a different variety. When a write-deferral session is enabled, the file system translator changes the **`cachePriority`** field it passes to the drivers so that blocks are marked as write-deferred. This places them in the cache in a nonpurgeable state but not on the disk. The ProDOS FST asks that all of its ProDOS-specific directory, index, and bitmap blocks be placed in the cache write-deferred, but that all pure data blocks go straight out to disk without caching. This enables most devices to write data to contiguous areas of the disk, so that the head doesn't need to move back and forth writing directories and index blocks and updating bitmaps. All of that is done in one burst at the end of the session. Other file system translators may use the cache during sessions in ways that make sense for that particular file system. For example, the AppleShare FST handles caching in a completely different way, not involving the

GS/OS cache at all, since the media could change on the server from another workstation. Most file system translators, however, use the GS/OS cache in ways similar to the way the ProDOS FST uses it.

A file system translator can deny an application's caching request if it interferes with optimal system performance. For example, caching a ProDOS file being copied to a ProDOS disk during a session would slow things down, since cached file blocks would be continually removed from a full cache to make room for more write-deferred system blocks from the ProDOS FST. Write-deferral sessions are usually used to copy large numbers of files or create them from memory. In either case, the files in question aren't likely to be read again, so the ProDOS FST eliminates possible overhead by denying requests to cache files during write-deferral sessions.

### RESETTING THE CACHE FROM THE APPLICATION

Only one application-level call acts only on the cache—the **`ResetCache`** call (class one only, call number $2026). This call forces the cache to be reinitialized, purging all blocks that are in it and resizing if necessary. Do not issue this call while a write-deferral session is in progress; you can use **`SessionStatus`** to see if a write-deferral session is currently active.

If you're writing a utility program and suspect that an file system translator has cached something you don't really want around, calling **`ResetCache`** will ensure the cache is flushed. The cache is also resized from the battery RAM parameter. As discussed earlier, this parameter is currently $81 and represents the cache size in 32K increments, but this is not guaranteed. **`ResetCache`** is called by the RAM CDev to change the size of the cache.

To make the call, simply issue it with a parameter block pointer to a word of $0000. There are no parameters.

### WHERE TO GO FROM HERE

This article has described the basics of the GS/OS cache, and has given you an idea of how an application requests caching and how such a request is fulfilled (or not fulfilled, if the request turns out not to be in the best interests of the system).

If you want to experiment with the effects of GS/OS calls that request caching, play around with these calls in the Exerciser that comes on the GSBug disk. That's why it's there. It's come a long way from the ProDOS 16 Exerciser. It makes all the calls (inline or stack-based, any class, with the exception of ResetCache), lists all devices, and catalogs directories to 255 levels. It also lets you choose any number of parameters for any class one call, except ResetCache, and allows you to modify memory through a built-in editor—you can visit the Monitor and return if you so choose. (Incidentally, the new calls for System Software 5.0 are coming in a revision soon.)

As always, help to Apple Partners on all matters, including GS/OS, is available on AppleLink and MCIMail from Apple II Developer Technical Support at the address AIIDTS. (Macintosh developers can contact Macintosh Developer Technical Support at AppleLink address MACDTS or MCIMail address MACTECH.) If you're not an Apple Partner, you can often get help from knowledgeable programmers on third-party online services, usually in the "Developers" or "Development" forum.

## Q

*I'm having trouble detecting clicks on picture and static text controls, even though I'm doing exactly what the Apple IIgs Toolbox Reference, Volume 3 says to do.*

## A

There is an error in the Toolbox Reference regarding static text and picture controls. The definition procedures for these control types do not include code to test for clicks within the control rectangle; therefore, routines like TestControl will always return FALSE. If you want to test for mouse clicks within a static text or picture control, retrieve the control rectangle from the control record and use a routine like PtInRect.

## Q

*How can I put extended controls in a dialog?*

## A

The Dialog Manager does not support the new extended controls, and probably will not be revised to do so. Inserting extended controls as user items generally won't work because the Dialog Manager will insert an invisible custom control over your extended control, preventing FindControl from finding it. If you want a pop-up menu in a dialog, you can create a user item with code that contains a non-control pop-up menu. When you get a hit on the item, call PopUpMenuSelect to handle the selection of a menu item. You must handle the drawing of the menu title, selected item and drop-shadowed rectangle yourself.

## Q

*Can my DA have a resource fork?*

## A

A new desk accessory may use the Resource Manager if it requires System Software 5.0 or later, as the Resource Manager will always be present. However, New Desk Accessory (NDA) authors must be aware that any application doing a Close with reference number zero will close the NDA's resource fork. The IIgs Finder™ can do this under some circumstances. Changing the file level will not solve this problem, as the level belongs to the current application and should not be permanently changed by desk accessories.

Classic Desk Accessories may also have resource forks. The same warnings apply, and the author must also be aware that the Resource Manager is not available to the CDA when the current operating system is ProDOS 8.

## Q

*How do I use TaskMasterDA?*

## A

TaskMasterDA is a call provided in System Software 5.0 and later so that new desk accessory authors may use the features of TaskMaster even when the current application doesn't use TaskMaster. The Desk Manager will pass an event record to the NDA, which should then copy it into an extended task record. TaskMasterDA will return values in the extended task record and will use values passed in that record, so be sure that the non-input fields are

zeroed before your first call to TaskMasterDA. The parameters to the call are listed in Volume 3 of the *Apple IIgs Toolbox Reference*.

## Q

*How come AppleTalk won't use the modem port on my ROM 01 IIgs?*

## A

If Slot 1 is set to "Your Card" in the Control Panel on a ROM 01 IIgs, AppleTalk will always use the printer port. AppleTalk will only use the modem port if Slot 1 is set to "Printer Port". This means that if you have a physical peripheral in slot 1 (such as a hard drive), AppleTalk will always use the printer port. ROM 3 IIgs machines require Slot 1 or Slot 2 to be set to "AppleTalk" to determine the network port, eliminating this confusion.

## Q

*What is the format of the IIgs Finder Data Files? There is no File Type Note describing the format.*

## A

The format of Finder data files is internal to the Finder. It is version-dependent and the information in the files can't be reliably used at this point, so the file format is not available.

## Q

*How can I control the speed of the Apple IIc Plus?*

## A

Firmware routines exist to allow the programming of the accelerator in the Apple IIc Plus. These routines are documented in the Apple IIc Technical Reference, Second Edition available from APDA. Apple encourages all developers with products that run on the Apple IIc to replace previous IIc documentation with this book.

## Q

*How can I get the Print Manager to print to the parallel card I choose when more than one is in my system?*

## A

The port driver provided with the system (the file Parallel.Card) searches for a card with which it can communicate in the PrDevIsItSafe routine, as documented in Apple IIgs Technical Note #36. The search routine scans upward starting with slot one, stopping when it finds a suitable card. The driver will always print to the card in the lower-numbered slot; it has no facilities to let the user choose among all suitable interfaces.

## Q

*How come the Memory Manager says I can't allocate a 300K handle when the Control Panel shows I have 312K free?*

## A

Just because there's 312K free doesn't mean that 300K of it is in one allocatable block. Memory may be fragmented so that small amounts of memory are available througout the memory map. Several public-domain and shareware utilities are available to show you exactly what memory is free and what is allocated. The ROM contains a Classic Desk Accessory to show memory allocation; the CDA may be installed by executing the "#" command from the system monitor prompt.

## Q

*How can I use the "extra" keys on the Apple Extended Keyboard on my IIgs? What values do they return?*

## A

The following chart shows what the "extra" keys and other extended keys of the Apple Extended Keyboard return on the Apple IIgs. All of the keys turn on bit 4 of the Modifier Key register at $C025 (see page 124 of the *Apple IIgs Hardware Reference*) to distinguish these keys from the "regular" keys.

| Key | ASCII char |
|---|---|
| F1 | z |
| F2 | x |
| F3 | c |
| F4 | v |
| F5 | ` ($E0) |
| F6 | a |
| F7 | b |
| F8 | d |
| F9 | e |
| F10 | m |
| F11 | g |
| F12 | o |
| F13 | i |
| F14 | k |
| F15 | q |
| help | r |
| home | s |
| page up | t |
| page down | u |
| end | w |
| del \|X> | y |

## Q

*Can I tell if a TextEdit record has changed?*

## A

TextEdit maintains a "dirty flag" in bit six ($40) of the flags field (not the moreFlags or the textFlags fields). This bit is cleared when a TextEdit record is created and is set when the TextEdit record is changed. The flags field is byte $0010 in the TextEdit record (or the control record if using TextEdit controls).

## Q

*When I'm juggling, everything starts out fine and then I find that I'm throwing everything forward. What's wrong?*

## A

What you are doing is called "joggling," it's good for your cardio-vascular system, but not so great for your juggling. Joggling is usually caused by throwing your second object a little late. Since you've given yourself less time to throw, you're using an abbreviated arm motion and you're forcing the object forward. To get back to juggling, try to make all of your throws just a tad higher (and make sure you initiate all tosses when the object being causht is at it's apex).

## Q

*How can I stop getting junk mail sent to my house?*

## A

Americans receive about 2 million tons of junk mail a year, about 44% of which is thrown out unopened. To keep your name from being sold to large mailing- list companies, write:

Mail Preference Service
Direct Marketing Association
11West 42nd Street
P.O. Box 3861
New York, NY 10017

# INDEX

This is a cumulative index; all entries for articles in this issue are in red and all entries for articles in previous issues are in black. The first issue ran from pages 1-112, and this issue runs from pages 113-256

## G



## H



## I-J-K



## L



## M

## N



## O



## P

## Q



## R



## S



## T

## U-V-W



## X-Y-Z