

Memory

This chapter provides a brief introduction to memory management on Macintosh computers. It describes the organization of the partition of memory assigned to your application when it is launched and explains the basic data types used by the Macintosh Toolbox and Operating System. This chapter also describes how you can allocate portions of that memory partition for specific purposes and how the Memory Manager helps to maintain an orderly partition.

This chapter provides only the minimum information about memory that you'll need to understand the rest of this book and to begin reading other *Inside Macintosh* books. For a more detailed description of basic memory management strategies, see the chapter "Introduction to Memory Management" in the book *Inside Macintosh: Memory*.

About Memory

In the cooperative multitasking environment provided by the Macintosh Operating System, your application can use only part of the total amount of RAM available on a computer. Some of the available RAM is reserved for use by the Operating System itself, and the remainder of the available memory is shared among all open applications.

When the Operating System starts up, it divides the available RAM into two broad sections. It reserves for itself a zone or *partition* of memory known as the **system partition**. The system partition always begins at the lowest addressable byte of memory (memory address 0) and extends upward. The system partition consists of two main parts:

- a system heap
- a set of global variables

In general, the memory in the system partition is for use by the Operating System alone. Your application probably won't need to read or write that memory.

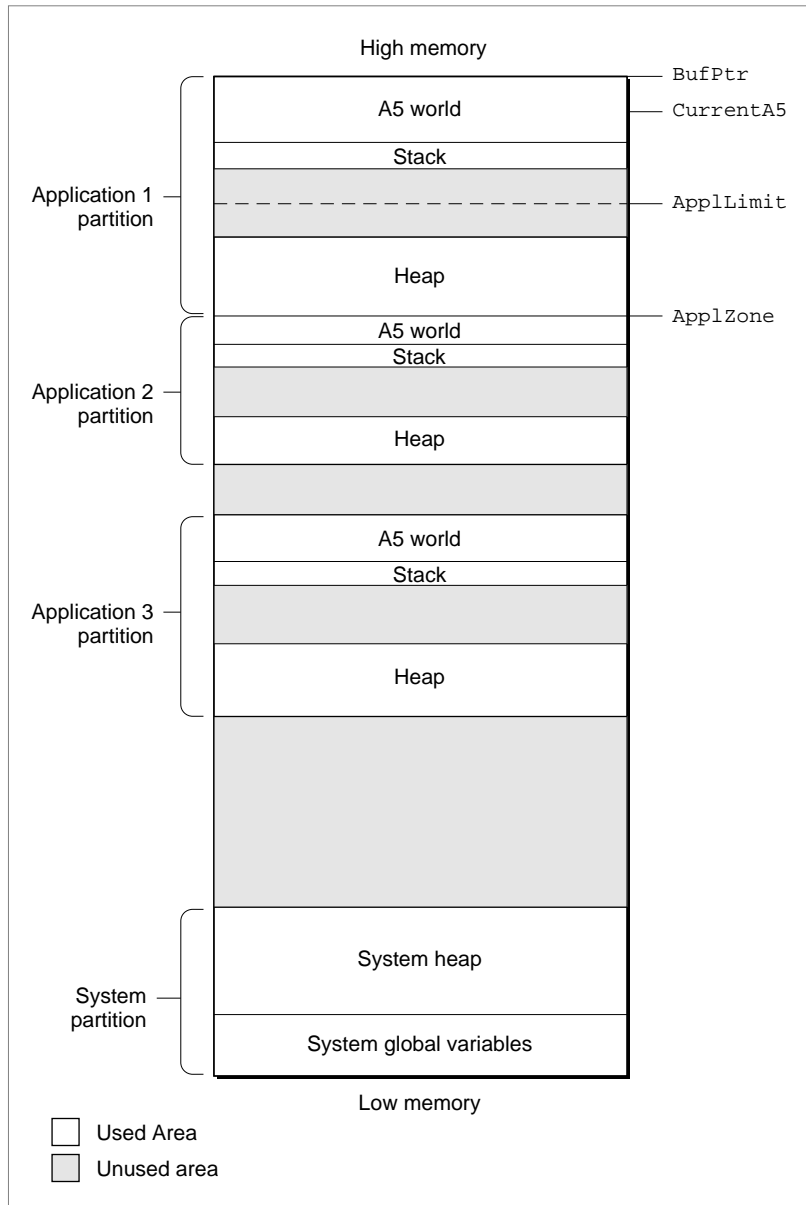
All memory outside the system partition is available for allocation to applications or other software components. In the cooperative multitasking environment, the user can have multiple applications open at once. When an application is launched, the Operating System assigns it a section of memory known as its application partition. In general, an application uses only the memory contained in its own application partition.

Figure 2-1 illustrates the organization of memory when several applications are open at the same time. The system partition occupies the lowest position in memory. Application partitions occupy some or all of the remaining space. Note that application partitions are loaded into the top part of memory first. An application partition consists of three main parts:

- an application heap
- a stack
- an A5 world, which includes the application's global variables

Memory

Figure 2-1 Memory organization in the cooperative multitasking environment



The System Heap

The main part of the system partition is an area of memory known as the *system heap*. In general, the system heap is reserved for exclusive use by the Operating System and other system software components, which load into it various items such as system resources, system code segments, and system data structures. All system buffers and queues, for example, are allocated in the system heap.

The system heap is also used for code and other resources that do not belong to specific applications, such as code resources that add features to the Operating System or that provide control of special-purpose peripheral equipment. System patches and system extensions (stored as code resources of type 'INIT') are loaded into the system heap during the system startup process. Hardware device drivers (stored as code resources of type 'DRVR') are loaded into the system heap when the driver is opened.

The System Global Variables

The lowest part of memory is occupied by a collection of global variables called *system global variables* (or *low-memory system global variables*). The Operating System uses these variables to maintain different kinds of information about the operating environment. For example, the `Ticks` global variable contains the number of ticks (sixtieths of a second) that have elapsed since the system was most recently started up. Similar variables contain, for example, the height of the menu bar (`MBarHeight`) and pointers to the heads of various operating-system queues (`DTQueue`, `FSQHdr`, `VBLQueue`, and so forth). Most low-memory global variables are of this variety: they contain information that is generally useful only to the Operating System or other system software components.

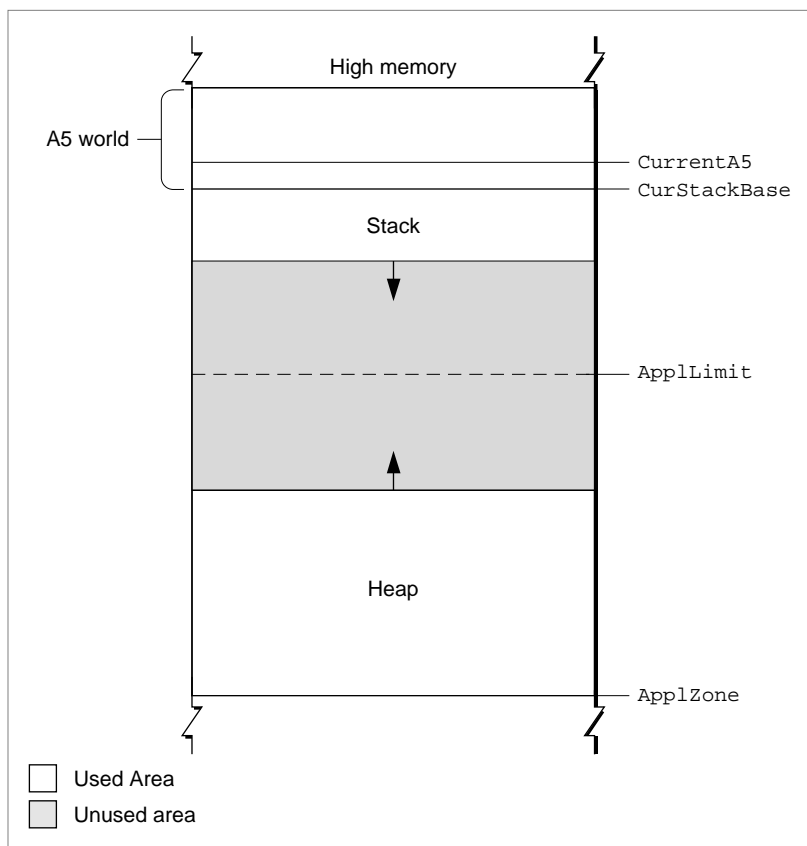
Other low-memory global variables contain information about the current application. For example, the `App1Zone` global variable contains the address of the first byte of the active application's partition. The `App1Limit` global variable contains the address of the last byte the active application's heap can expand to include. The `CurrentA5` global variable contains the address of the boundary between the active application's global variables and its application parameters. Because these global variables contain information about the active application, the Operating System changes the values of these variables whenever a context switch occurs (that is, whenever an application takes control of the CPU from another application).

In general, it is best to avoid reading or writing low-memory system global variables. Most of these variables are undocumented, and the results of changing their values can be unpredictable. Usually, when the value of a low-memory global variable is likely to be useful to applications, the system software provides a routine that you can use to read or write that value. For example, you can get the current value of the `Ticks` global variable by calling the `TickCount` function.

Application Partitions

When your application is launched, the Operating System allocates for it a partition of memory called its *application partition*. That partition contains required segments of the application's code as well as other data associated with the application. Figure 2-2 illustrates the general organization of an application partition.

Figure 2-2 Organization of an application partition



Your application partition is divided into three major parts:

- the application stack
- the application heap
- the application global variables and A5 world

The heap is located at the low-memory end of your application partition and always expands (when necessary) toward high memory. The A5 world is located at the

Memory

high-memory end of your application partition and is of fixed size. The stack begins at the high-memory end of the A5 world and expands downward, toward the top of the heap.

As you can see in Figure 2-2, there is usually an unused area of memory between the stack and the heap. This unused area provides space for the stack to grow without encroaching upon the space assigned to the application heap. In some cases, however, the stack might grow into space reserved for the application heap. If this happens, it is very likely that data in the heap will become corrupted.

The `ApplLimit` global variable marks the upper limit to which your heap can grow. If you call the `MaxApplZone` procedure at the beginning of your program, the heap immediately extends all the way up to this limit. If you were to use all of the heap's free space, the Memory Manager would not allow you to allocate additional blocks above `ApplLimit`. If you do not call `MaxApplZone`, the heap grows toward `ApplLimit` whenever the Memory Manager finds that there is not enough memory in the heap to fill a request. However, once the heap grows up to `ApplLimit`, it can grow no further. Thus, whether you maximize your application heap or not, you can use only the space between the bottom of the heap and `ApplLimit`.

Unlike the heap, the stack is not bounded by `ApplLimit`. If your application uses heavily nested procedures with many local variables or uses extensive recursion, the stack could grow downward beyond `ApplLimit`. Because you do not use Memory Manager routines to allocate memory on the stack, the Memory Manager cannot stop your stack from growing beyond `ApplLimit` and possibly encroaching upon space reserved for the heap. However, an Operating System task checks approximately 60 times each second to see if the stack has moved into the heap. If it has, the task, known as the "stack sniffer," generates a system error.

The Application Stack

The *stack* is an area of memory in your application partition that can grow or shrink at one end while the other end remains fixed. This means that space on the stack is always allocated and released in LIFO (last-in, first-out) order. The last item allocated is always the first to be released. It also means that the allocated area of the stack is always contiguous. Space is released only at the top of the stack, never in the middle, so there can never be any unallocated "holes" in the stack.

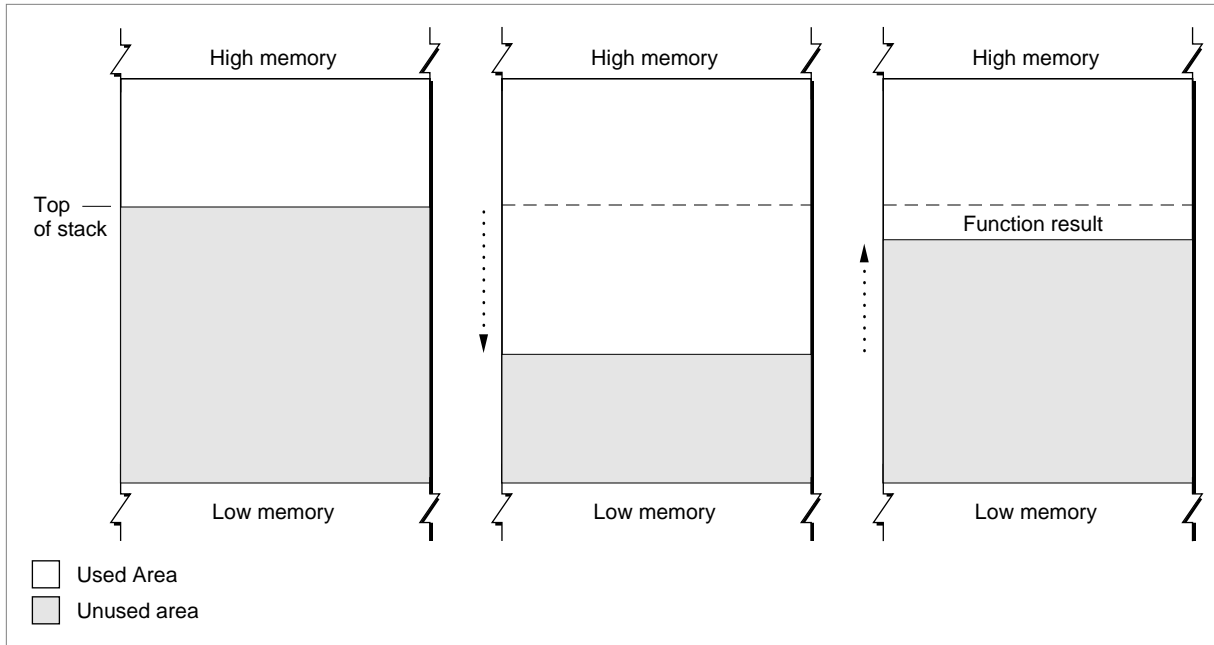
By convention, the stack grows from high-memory addresses toward low-memory addresses. The end of the stack that grows or shrinks is usually referred to as the "top" of the stack, even though it's actually at the lower end of memory occupied by the stack.

Because of its LIFO nature, the stack is especially useful for memory allocation connected with the execution of functions or procedures. When your application calls a routine, space is automatically allocated on the stack for a stack frame. A *stack frame* contains the routine's parameters, local variables, and return address. Figure 2-3 illustrates how the stack expands and shrinks during a function call. The leftmost diagram shows the stack just before the function is called. The middle diagram shows the stack expanded to hold the stack frame. Once the function is executed, the local

Memory

variables and function parameters are popped off the stack. If the function is a Pascal function, all that remains is the previous stack with the function result on top.

Figure 2-3 The application stack

**Note**

Dynamic memory allocation on the stack is usually handled automatically if you are using a high-level development language such as Pascal. The compiler generates the code that creates and deletes stack frames for each function or procedure call. ♦

The Application Heap

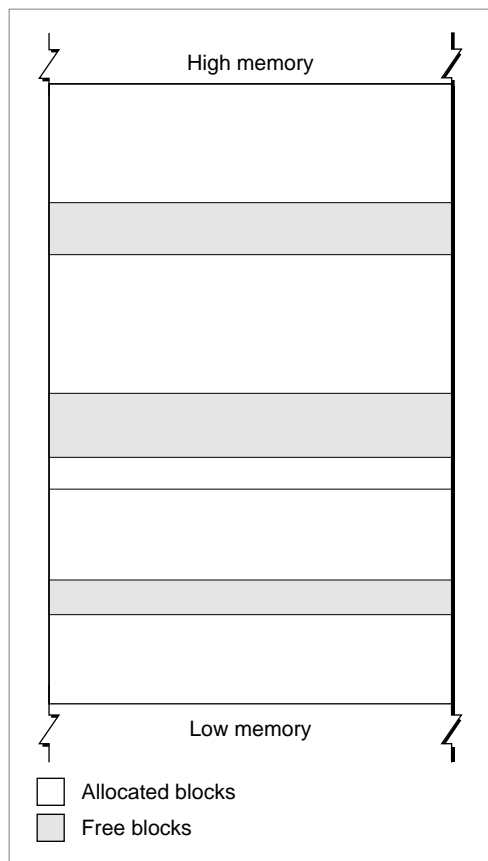
An *application heap* is the area of memory in your application partition in which space is dynamically allocated and released on demand. The heap begins at the low-memory end of your application partition and extends upward in memory. The heap contains virtually all items that are not allocated on the stack. For instance, your application heap contains the application's code segments and resources that are currently loaded into memory. The heap also contains other dynamically allocated items such as window records, dialog records, document data, and so forth.

Memory

You allocate space within your application's heap by making calls to the Memory Manager, either directly (for instance, using the `NewHandle` function) or indirectly (for instance, using a routine such as the Window Manager's `NewWindow`, which in turn calls Memory Manager routines). Space in the heap is allocated in **blocks**, which can be of any size needed for a particular object.

The Memory Manager does all the necessary housekeeping to keep track of blocks in the heap as they are allocated and released. Because these operations can occur in any order, the heap doesn't usually grow and shrink in an orderly way, as the stack does. Instead, after your application has been running for a while, the heap can tend to become fragmented into a patchwork of allocated and free blocks, as shown in Figure 2-4. This fragmentation is known as *heap fragmentation*.

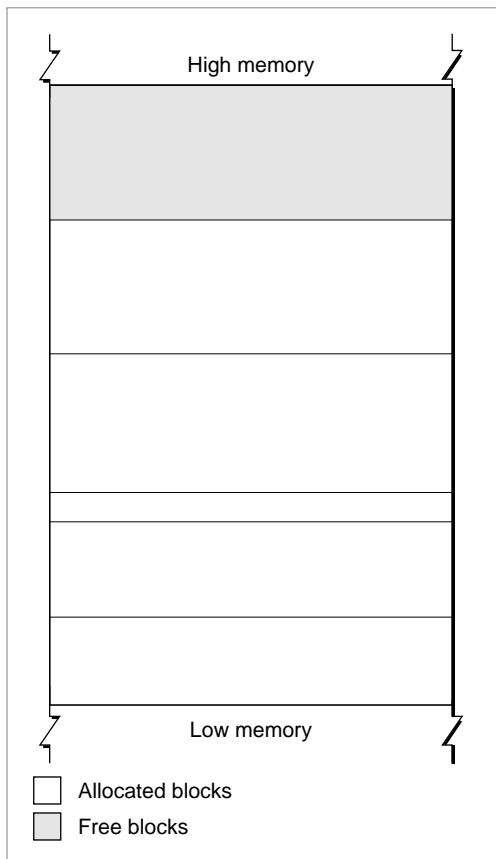
Figure 2-4 A fragmented heap



Memory

One result of heap fragmentation is that the Memory Manager might not be able to satisfy your application's request to allocate a block of a particular size. Even though there is enough free space available, the space is broken up into blocks smaller than the requested size. When this happens, the Memory Manager tries to create the needed space by moving allocated blocks together, thus collecting the free space in a single larger block. This operation is known as *heap compaction*. Figure 2-5 shows the results of compacting the fragmented heap shown in Figure 2-4.

Figure 2-5 A compacted heap



Heap fragmentation is generally not a problem as long as the blocks of memory you allocate are free to move during heap compaction. There are, however, two situations in which a block is not free to move: when it is a nonrelocatable block, and when it is a relocatable block that is temporarily locked in place. To minimize heap fragmentation, you should use nonrelocatable blocks sparingly, and you should lock relocatable blocks only when absolutely necessary. See "Memory Blocks" starting on page 38 for a description of relocatable and nonrelocatable blocks.

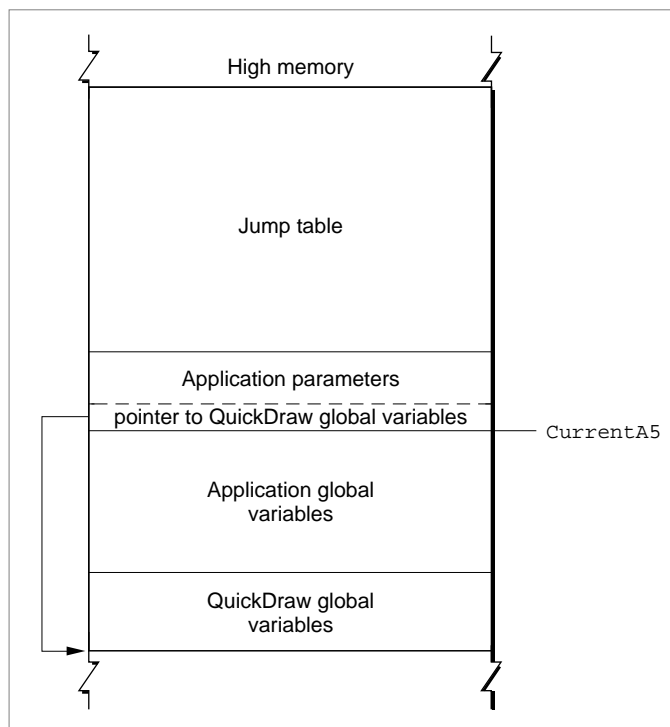
The Application Global Variables and A5 World

Your application's global variables are stored in an area of memory near the top of your application partition known as the application *A5 world*. The A5 world contains four kinds of data:

- application global variables
- application QuickDraw global variables
- application parameters
- the application's jump table

Each of these items is of fixed size, although the sizes of the global variables and of the jump table vary from application to application. Figure 2-6 shows the standard organization of the A5 world.

Figure 2-6 Organization of an application's A5 world



Note

An application's global variables may appear either above or below the QuickDraw global variables. The relative locations of these two items are determined by your development system's linker. In addition, part of the jump table might appear below the boundary pointed to by CurrentA5. ♦

Memory

The system global variable `CurrentA5` points to the boundary between the current application's global variables and its application parameters. For this reason, the *application's global variables* are found as negative offsets from the value of `CurrentA5`. This boundary is important because the Operating System uses it to access the following information from your application: its global variables, its QuickDraw global variables, the application parameters, and the jump table. This information is known collectively as the A5 world because the Operating System uses the microprocessor's A5 register to point to that boundary.

Your application's *QuickDraw global variables* contain information about its drawing environment. For example, among these variables is a pointer to the current graphics port.

Your application's *jump table* contains an entry for each of your application's routines that is called by code in another segment. The Segment Manager uses the jump table to determine the address of any externally referenced routines called by a code segment. For more information on jump tables, see the chapter "Segment Manager" in *Inside Macintosh: Processes*.

The *application parameters* are 32 bytes of memory located above the application global variables; they're reserved for use by the Operating System. The first long word of those parameters is a pointer to your application's QuickDraw global variables.

Memory Blocks

You can use the Memory Manager to allocate two different types of blocks in your heap: nonrelocatable blocks and relocatable blocks. A *nonrelocatable block* is a block of memory whose location in the heap is fixed. In contrast, a *relocatable block* is a block of memory that can be moved within the heap (perhaps during heap compaction). The Memory Manager sometimes moves relocatable blocks during memory operations so that it can use the space in the heap optimally.

The Memory Manager provides data types that reference both relocatable and nonrelocatable blocks. It also provides routines that allow you to allocate and release blocks of both types.

Memory

Nonrelocatable Blocks

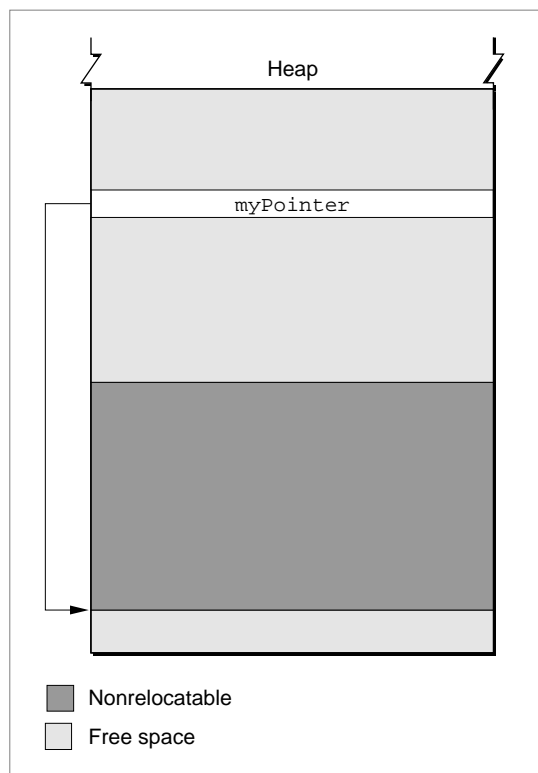
To reference a nonrelocatable block, you can use a *pointer* variable, defined by the `Ptr` data type.

```

TYPE
    SignedByte    = -128..127;
    Ptr           = ^SignedByte;
  
```

A pointer is simply the address of an arbitrary byte in memory, and a pointer to a nonrelocatable block of memory is simply the address of the first byte in the block, as illustrated in Figure 2-7. After you allocate a nonrelocatable block, you can make copies of the pointer variable. Because a pointer is the address of a block of memory that cannot be moved, all copies of the pointer correctly reference the block as long as you don't dispose of it.

Figure 2-7 A pointer to a nonrelocatable block



Memory

You can allocate a nonrelocatable block of memory by calling the Memory Manager function `NewPtr`. The Venn Diagrammer application uses the following line of code to allocate a new window record each time the user creates a new document window:

```
myPointer := NewPtr(sizeof(WindowRecord));
```

Here, `myPointer` is of type `Ptr`. (To see this line of code in context, look at Listing 6-6 on page 117.)

Relocatable Blocks

To reference relocatable blocks, the Memory Manager uses a scheme known as *double indirection*. The Memory Manager keeps track of a relocatable block internally with a *master pointer*, which itself is part of a nonrelocatable *master pointer block* in your application heap.

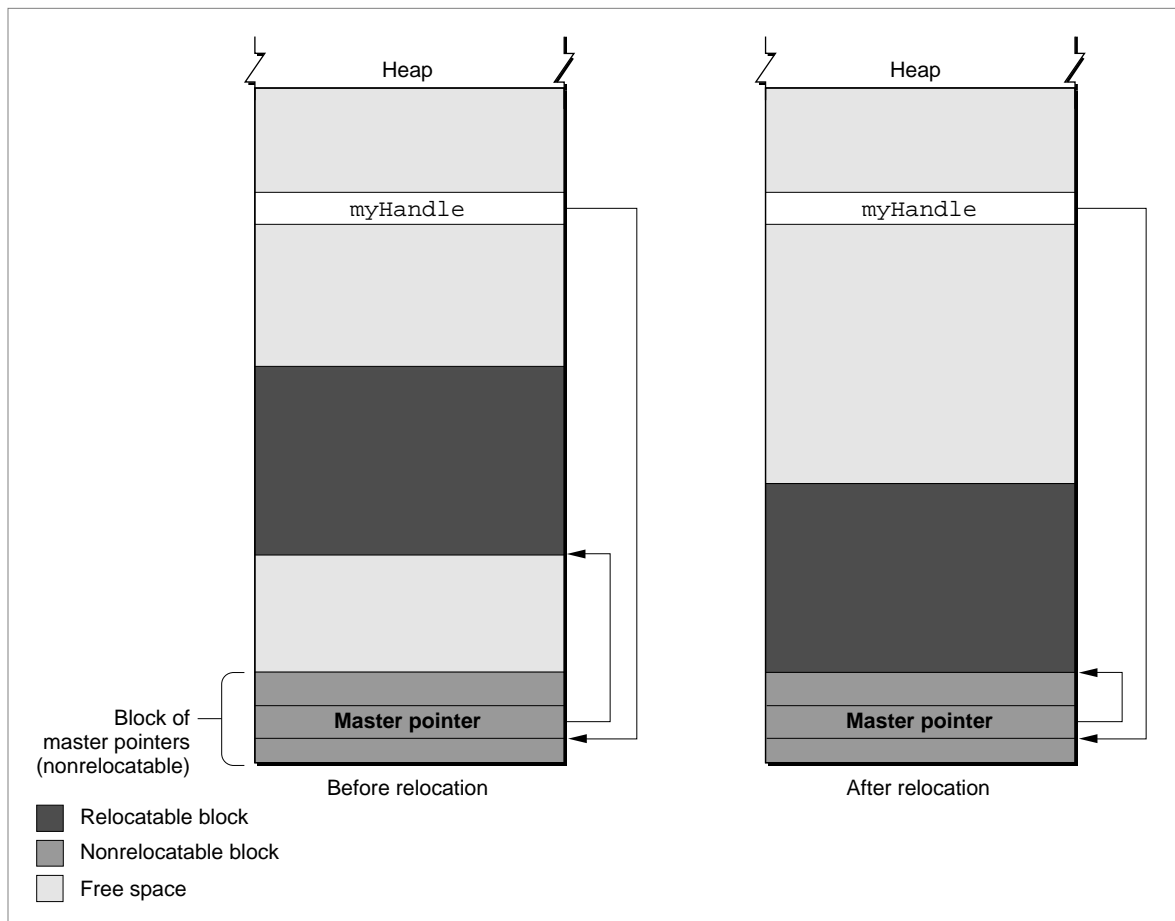
Note

The Memory Manager allocates one master pointer block (containing 64 master pointers) for your application at launch time, and you can call the `MoreMasters` procedure to request that additional master pointer blocks be allocated. ♦

When the Memory Manager moves a relocatable block, it updates the master pointer so that it always contains the address of the relocatable block. You reference the block with a *handle*, defined by the `Handle` data type.

```
TYPE
    Handle          = ^Ptr;
```

A handle contains the address of a master pointer. The left side of Figure 2-8 shows a handle to a relocatable block of memory located in the middle of the application heap. If necessary (perhaps to make room for another block of memory), the Memory Manager can move that block down in the heap, as shown in the right side of Figure 2-8.

Figure 2-8 A handle to a relocatable block

Master pointers for relocatable objects in your heap are always allocated in your application heap. Because the blocks of master pointers are nonrelocatable, it is best to allocate them as low in your heap as possible. You can do this by calling the `MoreMasters` procedure when your application starts up.

Memory

You can allocate a relocatable block of memory by calling the Memory Manager function `NewHandle`. The Venn Diagrammer application uses the following line of code to allocate a new document record each time the user creates a new document window:

```
myHandle := MyDocRecHnd(NewHandleClear(sizeof(MyDocRec)));
```

Here, `myHandle` is of type `MyDocRecHnd`. The `NewHandleClear` function is a variant of `NewHandle` that clears all bytes in the new block to 0. (To see this line of code in context, look at Listing 6-6 on page 117.)

Whenever possible, you should allocate memory in relocatable blocks. This gives the Memory Manager the greatest freedom when rearranging the blocks in your application heap to create a new block of free memory. In some cases, however, you may be forced to allocate a nonrelocatable block of memory. When you call the Window Manager function `NewWindow`, for example, the Window Manager internally calls the `NewPtr` function to allocate a new nonrelocatable block in your application partition. You need to exercise care when calling Toolbox routines that allocate such blocks, lest your application heap become overly fragmented.

Using relocatable blocks makes the Memory Manager more efficient at managing available space, but it does carry some overhead. As you have seen, the Memory Manager must allocate extra memory to hold master pointers for relocatable blocks. It groups these master pointers into nonrelocatable blocks. For large relocatable blocks, this extra space is negligible, but if you allocate many very small relocatable blocks, the cost can be considerable. For this reason, you should avoid allocating a very large number of handles to small blocks; instead, allocate a single large block and use it as an array to hold the data you need.

As you have seen, a heap block can be either relocatable or nonrelocatable. The designation of a block as relocatable or nonrelocatable is a permanent property of that block. If relocatable, a block can be either locked or unlocked; if it's unlocked, a block can be either purgeable or unpurgeable. These attributes of relocatable blocks can be set and changed as necessary. The following sections explain how to lock and unlock blocks, and how to mark them as purgeable or unpurgeable.

Locking and Unlocking Relocatable Blocks

Occasionally, you might need a relocatable block of memory to stay in one place. To prevent a block from moving, you can **lock** it, using the `HLock` procedure. Once you have locked a block, it won't move. Later, you can **unlock** it, using the `HUnlock` procedure, allowing it to move again.

In general, you need to lock a relocatable block only if there is some danger that it might be moved during the time that you read or write the data in that block. This might happen, for instance, if you dereference a handle to obtain a pointer to the data and (for increased speed) use the pointer within a loop that calls routines that might cause memory to be moved. If, within the loop, the block whose data you are accessing is in fact moved, then the pointer no longer points to that data; this pointer is said to **dangle**.

Memory

Using locked relocatable blocks can, however, hinder the Memory Manager as much as using nonrelocatable blocks. The Memory Manager can't move locked blocks. In addition, except when you allocate memory and resize relocatable blocks, it can't move relocatable blocks around locked relocatable blocks (just as it can't move them around nonrelocatable blocks). Thus, locking a block in the middle of the heap for long periods can increase heap fragmentation.

Locking and unlocking blocks every time you want to prevent a block from moving can become troublesome. Fortunately, the Memory Manager moves unlocked, relocatable blocks only at well-defined, predictable times. In general, each routine description in *Inside Macintosh* indicates whether the routine could move or purge memory. If you do not call any of those routines in a section of code, you can rely on all blocks to remain stationary while that code executes.

Purging and Reallocating Relocatable Blocks

One advantage of relocatable blocks is that you can use them to store information that you would like to keep in memory to make your application more efficient, but that you don't really need if available memory space becomes low. For example, your application might, at the beginning of its execution, load user preferences from a preferences file into a relocatable block. As long as the block remains in memory, your application can access information from the preferences file without actually reopening the file. However, reopening the file probably wouldn't take enough time to justify keeping the block in memory if memory space were scarce.

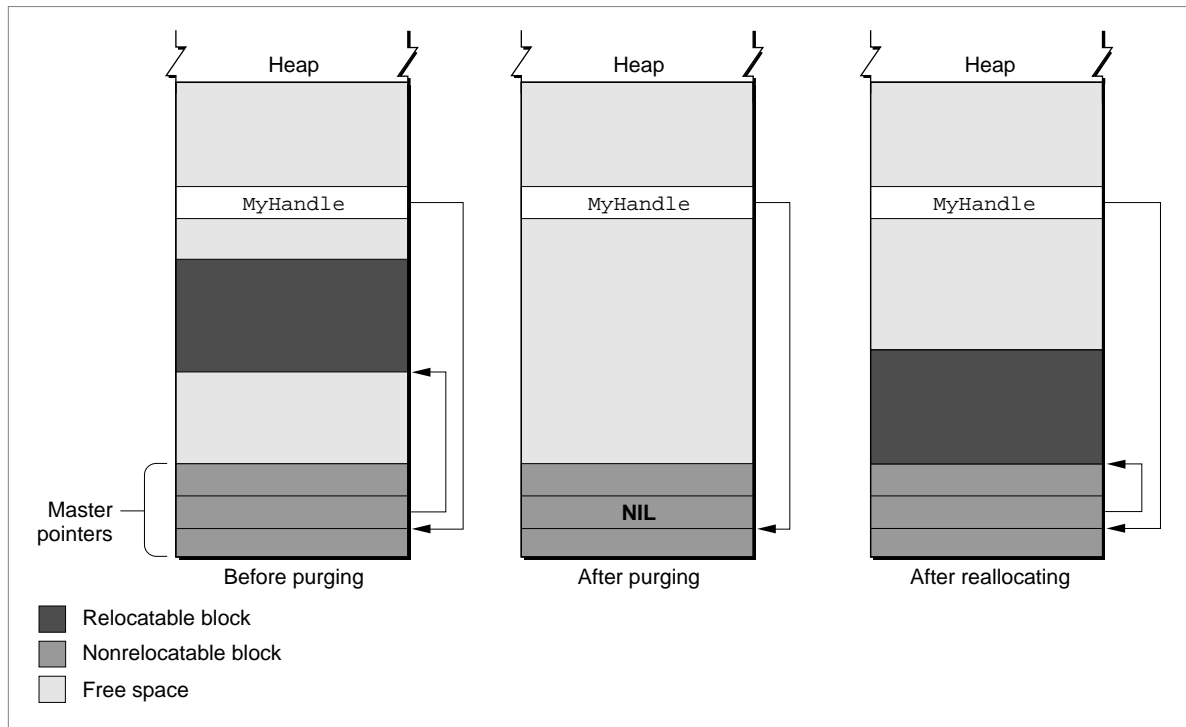
By making a relocatable block *purgeable*, you allow the Memory Manager to free the space it occupies if necessary. If you later want to prohibit the Memory Manager from freeing the space occupied by a relocatable block, you can make the block *unpurgeable*. You can use the `HPurge` and `HNoPurge` procedures to change back and forth between these two states.

IMPORTANT

A block you create by calling `NewHandle` is initially unlocked and unpurgeable. As a result, you don't have to worry about the block being purged unless you make the block purgeable. ▲

Once you make a relocatable block purgeable, you should subsequently check handles to that block before using them if you call any of the routines that could move or purge memory. If a handle's master pointer is set to `NIL`, then the Operating System has purged its block. To use the information formerly in the block, you must reallocate space for it (perhaps by calling the `ReallocateHandle` procedure) and then reconstruct its contents (for example, by rereading the preferences file). Figure 2-9 illustrates the purging and reallocating of a relocatable block. When the block is purged, its master pointer is set to `NIL`. When it is reallocated, the handle correctly references a new block, but that block's contents are initially undefined.

Memory

Figure 2-9 Purging and reallocating a relocatable block

Data Types

This section describes some of the general-purpose data types that the Memory Manager defines. These data types are used throughout the Macintosh Toolbox and Operating System.

Pointers and Handles

As you've seen, the Memory Manager uses pointers and handles to reference nonrelocatable and relocatable blocks, respectively. The data types `Ptr` and `Handle` define pointers and handles as follows:

TYPE

<code>SignedByte</code>	<code>= -128..127;</code>	{any byte in memory}
<code>Byte</code>	<code>= 0..255;</code>	{an unsigned byte}
<code>Ptr</code>	<code>= ^SignedByte;</code>	{address of a signed byte}
<code>Handle</code>	<code>= ^Ptr;</code>	{address of a master pointer}

Memory

The `SignedByte` data type stands for an arbitrary byte in memory, just to give `Ptr` and `Handle` something to point to. The `Byte` data type is an alternative definition that treats byte-length data as an unsigned rather than a signed quantity.

The Pascal language defines the special symbol `NIL`, which can be the value of any pointer type. You can assign `NIL` to any pointer (and hence to any handle) to indicate that the pointer has a defined value but does not point anywhere useful. Some system software routines return `NIL` as the value of a pointer or handle if the routine fails to perform the requested action. For example, the `NewHandle` routine returns `NIL` if the requested amount of memory is not available in the application heap.

For C, the type declarations look like this:

```
typedef char SignedByte;      /*any byte in memory*/
typedef unsigned char Byte;   /*an unsigned byte*/
typedef char *Ptr;            /*address of a signed byte*/
typedef Ptr *Handle;          /*address of a master pointer*/
```

Unlike Pascal, the C language does not contain a reserved symbol for a nil pointer. Most development systems, however, include definitions of both `nil` and `NULL`:

```
#define NULL 0
#define nil 0
```

Because of C's loose type conventions, you can assign the values `nil` and `NULL` to data types other than pointers and handles. In Pascal, the compiler generates an error if you try to assign the value `NIL` to an object whose data type is not defined as a pointer to some data type.

Strings

The Macintosh system software uses strings in arrays of up to 255 characters, with the first byte of the array storing the length of the string. Some Toolbox routines allow you to pass such a string directly; others require that you pass a pointer or a handle to a string. The Memory Manager provides the following type definitions that define character strings in terms of the Pascal `String` data type:

```
TYPE
    Str15          = String[15];
    Str27          = String[27];
    Str31          = String[31];
    Str63          = String[63];
    Str255         = String[255];
    StringPtr      = ^Str255;
    StringHandle   = ^StringPtr;
```

Memory

The C language treats strings differently than Pascal does. In C, strings are of variable length, with the end of the string marked by a special delimiter, usually the null character (ASCII 0). If you are using C, you must make certain to pass Pascal-style strings to Toolbox routines or to use special versions of the Toolbox routines that accept C strings. Check the documentation for your development environment for complete details.

Procedure Pointers

For treating procedures and functions as data objects, the Memory Manager defines the `ProcPtr` data type:

```
TYPE
    ProcPtr      = Ptr;           {pointer to a procedure}
```

For example, after the declarations

```
VAR
    myProcPtr:    ProcPtr;

PROCEDURE MyProc;
BEGIN
    ...
END;
```

you can make `myProcPtr` reference the `MyProc` procedure by using Pascal's `@` operator, as follows:

```
myProcPtr := @MyProc;
```

With the `@` operator, you can assign procedures and functions to variables of type `ProcPtr`, embed them in data structures, and pass them as arguments to other routines. Notice, however, that the data type `ProcPtr` technically points to an arbitrary byte, not an actual routine. As a result, there's no way in Pascal to access the underlying routine via this pointer in order to call it. Only routines written in assembly language can actually call routines designated by pointers of type `ProcPtr`.

Note

You can't use the `@` operator to reference procedures or functions whose declarations are nested within other routines. ♦

Type Coercion

Because of Pascal's strong typing rules, you can't directly assign a pointer value to a variable of some other pointer type, or pass a pointer variable to a routine requesting some other pointer type. Instead, you have to coerce the pointer from one type to another.

For example, you can call the `HLock` procedure to lock a relocatable block of memory. The `HLock` procedure requires a parameter of type `Handle`. If the block you want to lock isn't referenced by a variable of type `Handle`, you must coerce the variable to the required type. Here's an example:

```
HLock(Handle(myData));
```

Similarly, the `GetDialogItem` procedure returns in a `VAR` parameter a handle to an item in a dialog box. If you were to use the procedure to obtain the handle to a button in the variable `itemHand` of type `Handle`, you might need to access the button as a control. For example, you could access the button's enclosing rectangle with the code:

```
ControlHandle(itemHand)^^.contrlRect;
```

You can use this same syntax to equate any two variables of the same length. For example:

```
VAR
    myChar:      Char;
    myByte:      Byte;

myByte := Byte(myChar);
```

You can also use the functions `ORD`, `ORD4`, and `POINTER` to coerce variables of different length from one type to another. For example:

```
VAR
    myInteger:   Integer;
    myLongInt:   LongInt;
    myPointer:   Ptr;

myInteger := ORD(myLongInt);      {two low-order bytes only}
myInteger := ORD(myPointer);      {two low-order bytes only}
myLongInt := ORD(myInteger);      {packed into high-order bytes}
myLongInt := ORD4(myInteger);     {packed into low-order bytes}
myLongInt := ORD(myPointer);
myPointer := POINTER(myInteger);
myPointer := POINTER(myLongInt);
```

Note

Assembly-language and C language programmers don't need to bother with type coercion. ♦

