Module 20: Win32 Memory Management

Program examples compiled using Visual C++ 6.0 (MFC 6.0) compiler on Windows XP Pro machine with Service Pack 2. Topics and sub topics for this Tutorial are listed below. C language for the memory and compiler story can be found in Module W, Module Z and Buffer overflow (quite complete).

Win32 Memory Management

Forget everything you ever knew about Win16 memory management. Some of the Win16 memory management functions, such as `GlobalAlloc()`, were carried forward into Win32, but this was done to enable developers to port source code quickly. Underneath, the original functions work very differently, and many new ones have been added. This module starts out with a dose of Win32 memory management theory, which includes coverage of the fundamental heap management functions. Then you'll see how the C++ new and delete operators connect with the underlying heap functions. Finally, you'll learn how to use the memory-mapped file functions, and you'll get some practical tips on managing dynamic memory. In no way is this module intended to be a definitive description of Win32 memory management. For that, you'll have to read Jeffrey Richter's Advanced Windows (Microsoft Press, 1997). Be sure you have the latest edition, a new version may be in the works that covers Microsoft Windows 98/NT 5.0. At the time this edition was written, both Windows 98 and Windows NT 5.0 (Windows 2000) were in beta and not released. Our examination of these betas indicates that the memory management has not changed significantly.

Processes and Memory Space

Before you learn how Microsoft Windows manages memory, you must first understand what a **process** is. If you already know what a program is, you're on your way. A program is an EXE file that you can launch in various ways in Windows. Once a **program is running**, it's called a **process**. A process owns its memory, file handles, and other system resources. If you launch the same program twice in a row, you have two separate processes running simultaneously. Both the Microsoft Windows NT Task Manager (right-click the taskbar) and the Microsoft Windows 95 **PVIEW95** program give you a detailed list of processes that are currently running, and they allow you to kill processes that are not responding. The **SPY++** program (Microsoft Visual C++ 6.0 Tools) shows the relationships among processes, tasks, and windows. You can also use Windows Task Manager (by pressing Ctrl + Alt + Del).

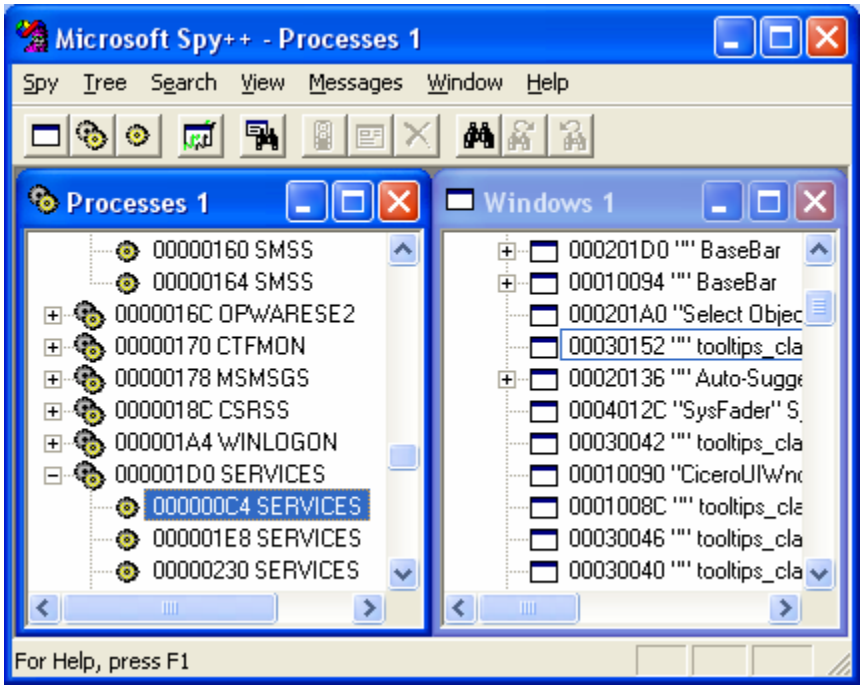

Figure 1: Using SPY++ to see the relationship among processes, tasks and windows.

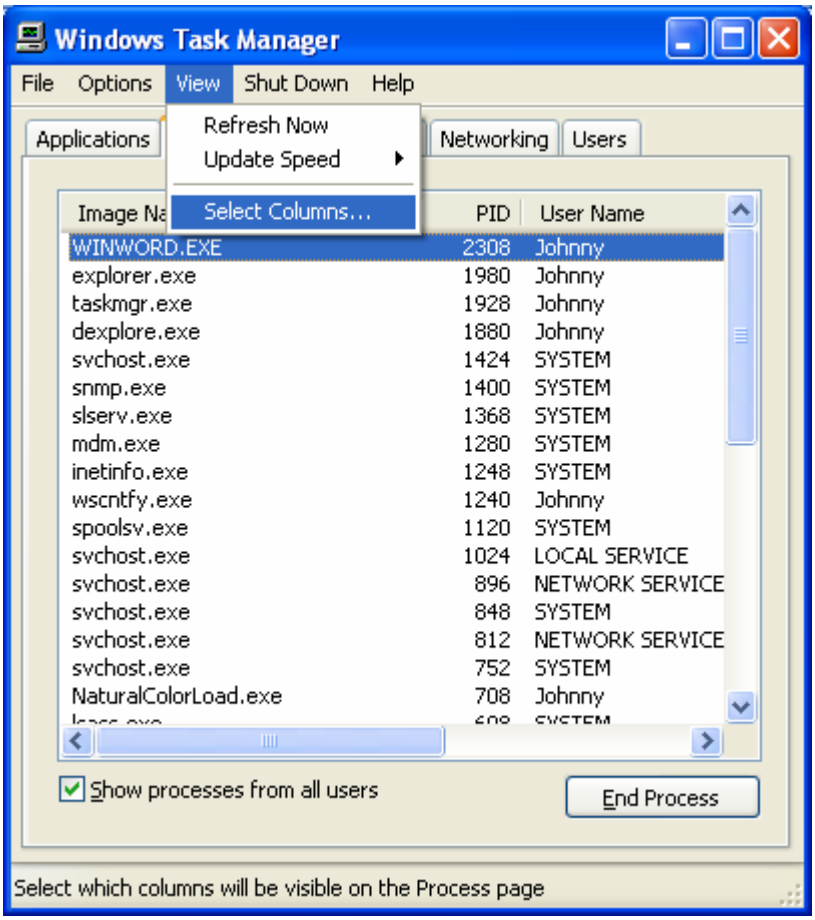

Figure 2: Using Windows Task Manager to see the processes and other windows system objects.

The Windows taskbar shows main windows, not processes. A single process (such as Windows Explorer) might have several main windows, each supported by its own thread, and some processes don't have windows at all. The important thing to know about a process is that it has its own "private" **4-gigabyte (GB) virtual address space** (which will be described in detail in the next section). For now, pretend that your computer has hundreds of gigabytes of RAM and that each process gets 4 GB. Your program can access any byte of this space with a single 32-bit linear address. Each process's memory space contains a variety of items, including the following:

- Your program's EXE image.
- Any non-system DLLs that your program loads, including the MFC DLLs.
- Your program's global data (read-only as well as read/write).
- Your program's stack.
- Dynamically allocated memory, including Windows and C-runtime library (CRT) heaps.
- Memory-mapped files.
- Interprocess shared memory blocks.
- Memory local to specific executing threads.
- All sorts of special system memory blocks, including virtual memory tables.
- The Windows kernel and executive, plus DLLs that are part of Windows.

The Windows 95 Process Address Space

In Windows 95, only the bottom 2 GB (0 to 0x7FFFFFFF) of address space is truly private, and the bottom 4 MB of that is off-limits. The stack, heaps, and read/write global memory are mapped in the bottom 2 GB along with application EXE and DLL files.
The top 2 GB of space is the same for all processes and is shared by all processes. The Windows 95 kernel, executive, virtual device drivers (**VxD**s), and file system code, along with important tables such as page tables, are mapped to the top 1 GB (0xC0000000 to 0xFFFFFFFF) of address space. Windows DLLs and memory-mapped files are located in the range 0x80000000 to 0xBFFFFFFF. Figure 3 shows a memory map of **two processes using the same program**.

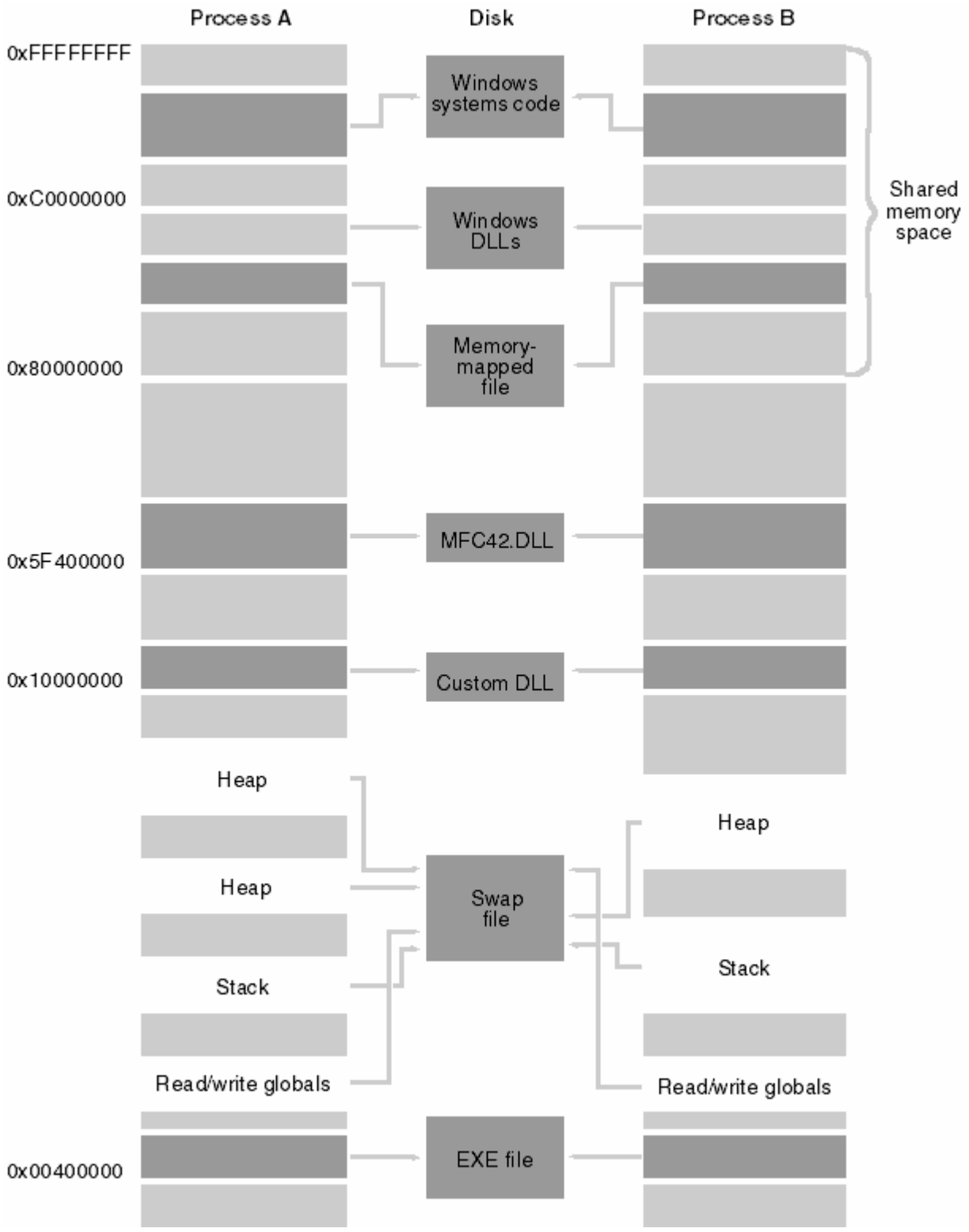

Figure 3: A typical Windows 95 virtual memory map for two processes linked to the same EXE file.

How safe is all this? It's next to impossible for one process to overwrite another process's stack, global, or heap memory because this memory, located in the **bottom 2 GB of virtual address space**, is assigned only to that specific process. All EXE and DLL code is **flagged as read-only**, so there's no problem if the code is mapped in several processes. However, because important Windows read/write data is mapped there, the **top 1 GB of address space is vulnerable**. An errant program could wipe out important system tables located in this region. In addition, one process could mess up

another process's memory-mapped files in the range **0x80000000** through **0xBFFFFFFF** because this region is shared by all processes.

The Windows NT Process Address Space

A process in Windows NT can access only the **bottom 2 GB of its address space** and the lowest and highest 64 KB of that is inaccessible. The EXE, the application's DLLs and Windows DLLs, and memory-mapped files all reside in this space between 0x00010000 and 0x7FFEFFFF. The Windows NT kernel, executive, and device drivers all reside in the **upper 2 GB**, where they are completely protected from any tampering by an errant program. Memory-mapped files are safer, too. One process cannot access another's memory-mapped file without knowing the file's name and explicitly mapping a view.

How Virtual Memory Works

You know that your computer doesn't really have hundreds of gigabytes of RAM. Windows uses some smoke and mirrors here. First of all, a process's 4-GB address space is going to be used sparsely. Various programs and data elements will be scattered throughout the 4-GB address space in 4-KB units starting on 4-KB boundaries. Each 4-KB unit, called a **page**, can hold either code or data. When a page is being used, it **occupies physical memory**, but you never see its physical memory address. The Intel microprocessor chip efficiently maps a **32-bit virtual address** to both a **physical page** and an **offset within the page**, using two levels of **4-KB page tables**, as shown in Figure 4. Note that individual pages can be flagged as either read-only or read/write. Also note that each process has its own set of page tables. The chip's CR3 register (Control Register) holds a pointer to the **directory page**, so when Windows switches from one process to another, it simply updates CR3.

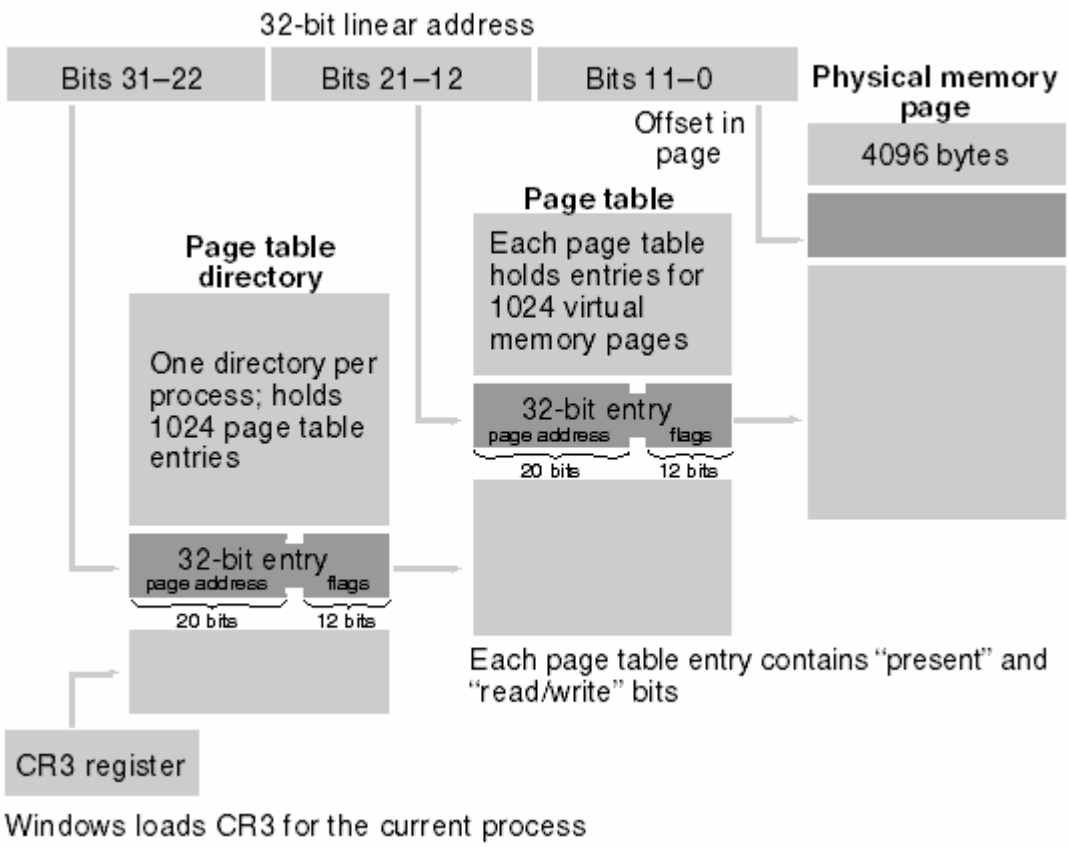

Figure 4: Win32 virtual memory management (Intel).

So now our process is down from 4 GB to maybe 5 MB, a definite improvement. But if we're running several programs, along with Windows itself, we'll still run out of RAM. If you look at Figure 10-2 again, you'll notice that the page table entry has a "present" bit that indicates whether the 4-KB page is currently in RAM. If we try to access a page that's not in RAM, an interrupt fires and Windows analyzes the situation by checking its internal tables. If the memory reference

was bogus, we'll get the dreaded "**page fault**" message and the program will exit. Otherwise, Windows reads the page from a disk file into RAM and updates the page table by loading the physical address and setting the present bit. This is the essence of Win32 virtual memory.

The Windows virtual memory manager figures out how to read and write 4-KB pages so that it optimizes performance. If one process hasn't used a page for a while and another process needs memory, the first page is swapped out or discarded and the RAM is used for the new process's page. Your program isn't normally aware that this is going on. The more disk I/O that happens, however, the worse your program's performance will be, so it stands to reason that more RAM is better.

I mentioned the word "disk," but I haven't talked about files yet. All processes share a big **system wide swap file** that's used for all read/write data and some read-only data. Windows NT supports multiple swap files. Windows determines the swap file size based on available RAM and free disk space, but there are ways to fine-tune the swap file's size and specify its physical location on disk.

The swap file isn't the only file used by the virtual memory manager, however. It wouldn't make sense to write code pages back to the swap file, so instead of using the swap file, Windows maps EXE and DLL files directly to their files on disk. Because the code pages are marked read-only, there's never a need to write them back to disk. If two processes use the same EXE file, that file is mapped into each process's address space. The code and constants never change during program execution, so the same physical memory can be mapped for each process. The two processes cannot share global data, however, and Windows 95 and Windows NT handle this situation differently. Windows 95 maps separate copies of the global data to each process. In Windows NT, both processes use the same copy of each page of global data until one process attempts to write to that page. At that point the page is copied; as a result, each process has its own private copy stored at the same virtual address.

A dynamic link library can be mapped directly to its DLL file only if the DLL can be loaded at its designated **base address**. If a DLL were statically linked to load at, say, 0x10000000 but that address range is already occupied by another DLL, Windows must "fix up" the addresses within the DLL code. Windows NT copies the altered pages to the swap file when the DLL is first loaded, but Windows 95 can do the fix up "on the fly" when the pages are brought into RAM. Needless to say, it's important to build your DLLs with non-overlapping address ranges. If you're using the MFC DLLs, set the base address of your own DLLs outside the range 0x5F400000 through 0x5FFFFFFF. Memory-mapped files, which I'll talk about later, are also mapped directly. These can be flagged as read/write and made available for sharing among processes.

For Win32 Programmers: Segment Registers in Win32

If you've experimented with the debugger in Win32, you may have noticed the segment registers, particularly CS, DS, and SS. These 16-bit relics haven't gone away, but you can mostly ignore them. In 32-bit mode, the Intel microprocessor still uses segment registers, which are 16 bits long, to translate addresses prior to sending them through the virtual memory system. A table in RAM, called the **descriptor table**, has entries that contain the virtual memory base address and block size for code, data, and stack segments. In 32-bit mode, these segments can be up to 4 GB in size and can be flagged as read-only or read/write. For every memory reference, the chip uses the selector, the contents of a segment register, to look up the descriptor table entry for the purpose of translating the address.

Under Win32, each process has two segments, one for code and one for data and the stack. You can assume that both have a base value of 0 and a size of 4 GB, so they overlap. The net result is no translation at all, but Windows uses some tricks that exclude the bottom 16 KB from the data segment. If you try to access memory down there, you get a **protection fault** instead of a **page fault**, which is useful for debugging null pointers. Some future operating system might someday use segments to get around that annoying 4-GB size limitation, but by then we'll have Win64 to worry about!

The `VirtualAlloc()` Function: Committed and Reserved Memory

If your program needs dynamic memory, sooner or later the Win32 `VirtualAlloc()` function will be called. Chances are that your program will never call `VirtualAlloc()`; instead you'll rely on the Windows heap or the C-Run Time (CRT) heap functions to call it directly. Knowing how `VirtualAlloc()` works, however, will help you better understand the functions that call it.

First you must know the meanings of **reserved** and **committed** memory. When memory is reserved, a contiguous virtual address range is set aside. If, for example, you know that your program is going to use a single 5-MB memory block (known as a **region**) but you don't need to use it all right away, you call `VirtualAlloc()` with a `MEM_RESERVE` allocation type parameter and a 5-MB size parameter. Windows rounds the start address of the region to a 64-KB boundary and prevents your process from reserving other memory in the same range. You can specify a start address for

your region, but more often you'll let Windows assign it for you. Nothing else happens. No RAM is allocated, and no swap file space is set aside.
When you get more serious about needing memory, you call `VirtualAlloc()` again to commit the reserved memory, using a `MEM_COMMIT` allocation type parameter. Now the start and end addresses of the region are rounded to 4-KB boundaries, and corresponding swap file pages are set aside together with the required page table. The block is designated either read-only or read/write. Still no RAM is allocated, however; RAM allocation occurs only when you try to access the memory. If the memory was not previously reserved, no problem. If the memory was previously committed, still no problem. The rule is that memory must be committed before you can use it. You call the `VirtualFree()` function to "decommit" committed memory, thereby returning the designated pages back to reserved status. `VirtualFree()` can also free a reserved region of memory, but you have to specify the base address you got from a previous `VirtualAlloc()` reservation call.

The Windows Heap and the `GlobalAlloc()` Function Family

A heap is a **memory pool for a specific process**. When your program needs a block of memory, it calls a heap allocation function, and it calls a companion function to free the memory. There's no assumption about 4-KB page boundaries; the heap manager uses space in existing pages or calls `VirtualAlloc()` to get more pages. First we'll look at Windows heaps. Next we'll consider heaps managed by the CRT library for functions like `malloc()` and `new`.
Windows provides each process with a default heap, and the process can create any number of additional Windows heaps. The `HeapAlloc()` function allocates memory in a Windows heap, and `HeapFree()` releases it.
You might never need to call `HeapAlloc()` yourself, but it will be called for you by the `GlobalAlloc()` function that's left over from Win16. In the ideal 32-bit world, you wouldn't have to use `GlobalAlloc()`, but in this real world, we're stuck with a lot of code ported from Win16 that uses "memory handle" (`HGLOBAL`) parameters instead of 32-bit memory addresses.
`GlobalAlloc()` uses the default Windows heap. It does two different things, depending on its attribute parameter. If you specify `GMEM_FIXED`, `GlobalAlloc()` simply calls `HeapAlloc()` and returns the address cast as a 32-bit `HGLOBAL` value. If you specify `GMEM_MOVEABLE`, the returned `HGLOBAL` value is a pointer to a handle table entry in your process. That entry contains a pointer to the actual memory, which is allocated with `HeapAlloc()`.
Why bother with "moveable" memory if it adds an extra level of indirection? You're looking at an artifact from Win16, in which, once upon a time, the operating system actually moved memory blocks around. In Win32, moveable blocks exist only to support the `GlobalReAlloc()` function, which allocates a new memory block, copies bytes from the old block to the new, frees the old block, and assigns the new block address to the existing handle table entry. If nobody called `GlobalReAlloc()`, we could always use `HeapAlloc()` instead of `GlobalAlloc()`.
Unfortunately, many library functions use `HGLOBAL` return values and parameters instead of memory addresses. If such a function returns an `HGLOBAL` value, you should assume that memory was allocated with the `GMEM_MOVEABLE` attribute, and that means you must call the `GlobalLock()` function to get the memory address. If the memory was fixed, the `GlobalLock()` call just returns the handle as an address. Call `GlobalUnlock()` when you're finished accessing the memory. If you're required to supply an `HGLOBAL` parameter, to be absolutely safe you should generate it with a `GlobalAlloc(GMEM_MOVEABLE, ...)` call in case the called function decides to call `GlobalReAlloc()` and expects the handle value to be unchanged.

The Small-Block Heap, the C++ `new` and `delete` Operators, and `_heapmin()`

You can use the Windows `HeapAlloc()` function in your programs, but you're more likely to use the `malloc()` and `free()` functions supplied by the CRT. If you write C++ code, you won't call these functions directly; instead, you'll use the new and delete operators, which map directly to `malloc()` and `free()`. If you use new to allocate a block larger than a certain threshold (480 bytes is the default), the CRT passes the call straight through to `HeapAlloc()` to allocate memory from a Windows heap created for the CRT. For blocks smaller than the threshold, the CRT manages a small-block heap, calling `VirtualAlloc()` and `VirtualFree()` as necessary. Here is the algorithm:

1. Memory is reserved in 4-MB regions.
2. Memory is committed in 64-KB blocks (16 pages).
3. Memory is decommitted 64 KB at a time. As 128 KB becomes free, the last 64 KB is decommitted.
4. A 4-MB region is released when every page in that region has been decommitted.

As you can see, this small-block heap takes care of its own cleanup. The CRT's Windows heap doesn't automatically decommit and un-reserve pages, however. To clean up the larger blocks, you must call the CRT `_heapmin()` function, which calls the windows `HeapCompact()` function. Unfortunately, the Windows 95 version of `HeapCompact()` doesn't do anything, all the more reason to use Windows NT. Once pages are decommitted, other programs can reuse the corresponding swap file space.
In previous versions of the CRT, the free list pointers were stored inside the heap pages. This strategy required the `malloc()` function to "touch" (read from the swap file) many pages to find free space, and this degraded performance. The current system, which stores the free list in a separate area of memory, is faster and minimizes the need for third-party heap management software. If you want to change or access the block size threshold, use the CRT functions `_set_sbh_threshold()` and `_get_sbh_threshold()`. A special debug version of `malloc()`, `_malloc_dbg()`, adds debugging information inside allocated memory blocks. The new operator calls `_malloc_dbg()` when you build an MFC project with `_DEBUG` defined. Your program can then detect memory blocks that you forgot to free or that you inadvertently overwrote.

Memory-Mapped Files

In case you think you don't have enough memory management options already, I'll toss you another one. Suppose your program needs to read a DIB (device-independent bitmap) file. Your instinct would be to allocate a buffer of the correct size, open the file, and then call a read function to copy the whole disk file into the buffer. The Windows memory-mapped file is a more elegant tool for handling this problem, however. You simply map an address range directly to the file. When the process accesses a memory page, Windows allocates RAM and reads the data from disk. Here's what the code looks like:

```
HANDLE hFile = ::CreateFile(strPathname, GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
ASSERT(hFile != NULL);
HANDLE hMap = ::CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
ASSERT(hMap != NULL);
LPVOID lpvFile = ::MapViewOfFile(hMap, FILE_MAP_READ, 0, 0, 0); // Map whole file
DWORD dwFileSize = ::GetFileSize(hFile, NULL);  // useful info
// Use the file
::UnmapViewOfFile(lpvFile);
::CloseHandle(hMap);
::CloseHandle(hFile);
```

Here you're using virtual memory backed by the DIB file. Windows determines the file size, reserves a corresponding address range, and commits the file's storage as the physical storage for this range. In this case, lpvFile is the start address. The `hMap` variable contains the handle for the file mapping object, which can be shared among processes if desired.
The DIB in the example above is a small file that you could read entirely into a buffer. Imagine a larger file for which you would normally issue seek commands. A memory-mapped file works for such a file, too, because of the underlying virtual memory system. RAM is allocated and pages are read when you access them, and not before. By default, the entire file is committed when you map it, although it's possible to map only part of a file. If two processes share a file mapping object (such as `hMap` in the sample code above), the file itself is, in effect, shared memory, but the virtual addresses returned by `MapViewOfFile()` might be different. Indeed, this is the preferred Win32 method of sharing memory. (Calling the `GlobalAlloc()` function with the `GMEM_SHARE` flag doesn't create shared memory as it did in Win16.) If memory sharing is all you want to do and you don't need a permanent disk file, you can omit the call to `CreateFile()` and pass 0xFFFFFFFF as the `CreateFileMapping()` `hFile` parameter. Now the shared memory will be backed by pages in the **swap file**. Consult Richter for details on memory-mapped files. If you intend to access only a few random pages of a file mapping object that is backed by the swap file, you can use a technique that Jeffrey Richter describes in Advanced Windows under the heading "Sparsely Committed Memory-Mapped Files." In this case, you call `CreateFileMapping()` with a special flag and then you commit specific address ranges later with the `VirtualAlloc()` function. You might want to look carefully at the Windows message `WM_COPYDATA`. This message lets you transfer data between processes in shared memory without having to deal with the file mapping API. You must send this message rather than post it, which means the sending process has to wait while the receiving process copies and processes the data. Unfortunately, there's no direct support for memory-mapped files or shared memory in MFC. The `CSharedFile()` class supports only clipboard memory transfers using `HGLOBAL` handles, so the class isn't as useful as its name implies.

Accessing Resources

Resources are contained inside EXEs and DLLs and thus occupy virtual address space that doesn't change during the life of the process. This fact makes it easy to read a resource directly. If you need to access a bitmap, for example, you can get the DIB address with code like this:

```
LPVOID lpvResource = (LPVOID) ::LoadResource(NULL, ::FindResource(NULL,
MAKEINTRESOURCE(IDB_REDBLOCKS), RT_BITMAP));
```

The `LoadResource()` function returns an `HGLOBAL` value, but you can safely cast it to a pointer.

Some Tips for Managing Dynamic Memory

The more you use the heap, the more fragmented it gets and the more slowly your program runs. If your program is supposed to run for hours or days at a time, you have to be careful. It's better to allocate all the memory you need when your program starts and then free it when the program exits, but that's not always possible. The `CString` class is a nuisance because it's constantly allocating and freeing little bits of memory. Fortunately, MFC developers have recently made some improvements.

Don't forget to call `_heapmin()` every once in a while if your program allocates blocks larger than the small-block heap threshold. And be careful to remember where heap memory comes from. You'd have a big problem, for instance, if you called `HeapFree()` on a small-block pointer you got from new.

Be aware that your stack can be as big as it needs to be. Because you no longer have a 64-KB size limit, you can put large objects on the stack, thereby reducing the need for heap allocations.

As in Win16, your program doesn't run at full speed and then suddenly throw an exception when Windows runs out of swap space. Your program just slowly grinds to a halt, making your customer unhappy. And there's not much you can do except try to figure out which program is eating memory and why. Because the Windows 95 USER and GDI modules still have 16-bit components, there is some possibility of exhausting the 64-KB heaps that hold GDI objects and window structures. This possibility is pretty remote, however, and if it happens, it probably indicates a bug in your program.

Optimizing Storage for Constant Data

Remember that the code in your program is backed not by the swap file but directly by its EXE and DLL files. If several instances of your program are running, the same EXE and DLL files will be mapped to each process's virtual address space. What about **constant data**? You would want that data to be part of the program rather than have it copied to another block of address space that's backed by the swap file. You've got to work a little bit to ensure that constant data gets stored with the program. First consider string constants, which often permeate your programs. You would think that these would be read-only data, but guess again. Because you're allowed to write code like this:

```
char* pch = "test";
*pch = 'x';
```

"`test`" can't possibly be constant data, and it isn't.

If you want "`test`" to be a constant, you must declare it as an initialized const static or global variable. Here's the global definition:

```
const char g_pch[] = "test";
```

Now `g_pch` is stored with the code, but where, specifically? To answer that, you must understand the "data sections" that the Visual C++ linker generates. If you set the link options to generate a **map file**, you'll see a long list of the sections (memory blocks) in your program. Individual sections can be designated for code or data, and they can be read-only or read/write. The important sections and their characteristics are listed here.

Name	Type	Access	Contents
`.text`	Code	Read-only	Program code.
`.rdata`	Data	Read-only	Constant initialized data.
`.data`	Data	Read/write	Non-constant initialized data.
`.bss`	Data	Read/write	Non-constant uninitialized data.

Table 1.

The `.rdata` section is part of the EXE file, and that's where the linker puts the `g_pch` variable. The more stuff you put in the `.rdata` section, the better. The use of the const modifier does the trick. You can put built-in types and even structures in the `.rdata` section, but you can't put C++ objects there if they have constructors. If you write a statement like the following one:

```
const CRect g_rect(0, 0, 100, 100);
```

the linker puts the object into the `.bss` section, and it will be backed separately to the swap file for each process. If you think about it, this makes sense because the compiler must invoke the constructor function after the program is loaded. Now suppose you wanted to do the worst possible thing. You'd declare a `CString` global variable (or static class data member) like this:

```
const CString g_str("this is the worst thing I can do");
```

Now you've got the `CString` object (which is quite small) in the `.bss` section and you've also got a character array in the `.data` section, neither of which can be backed by the EXE file. To make matters worse, when the program starts, the `CString` class must allocate heap memory for a copy of the characters. You would be much better off using a `const` character array instead of a `CString` object.

Further reading and digging:

1. C language for the memory and compiler story can be found in Module W, Module Z and Buffer overflow (quite complete).
2. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
3. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
4. MSDN Library
5. Windows data type.
6. Win32 programming Tutorial.
7. The best of C/C++, MFC, Windows and other related books.
8. Unicode and Multibyte character set: Story and program examples.