Best Practices - Memory Management in C and C++

By Shilpi Gupta Agarwal

Abstract

C and C++ programmers very often allocate and de-allocate memory on the heap without the proper understanding on how these low-level facilities work and what happens underneath.

"Pay no attention to the man behind the curtain." —The Wizard of Oz

But these memory related problems becomes a great concern in the systems with shortage of almost all the resources including memory, like embedded real-time systems. This dynamic behavior tends to be non-deterministic and the failure is hard to contain.

Similarly memory allocation failure on such systems can be fatal. Unlike a desktop application, most embedded systems do not have the opportunity to pop up a dialog and discuss options with the user. Often, resetting is the only option, which is unattractive.

Hence, this paper attempts to discuss the strategies to achieve clean code and appropriate memory management.

Introduction

Memory management is the process of recognizing when allocated objects are no longer needed, de-allocating (freeing) the memory used by such objects, and making it available for subsequent allocations.

In C and C++ programming languages, memory management is programmer’s responsibility. The complexity of this task leads to many common errors that can cause unexpected or erroneous program behavior and crashes. As a result, a large proportion of developer time is often spent debugging and trying to correct such errors.

The newer languages (Java, Python, and Perl) have automated memory management. This ranges from Python’s simple reference-counted model to Java’s sophisticated garbage collector. A garbage collector automatically analyzes the program’s data and decides whether memory is in use and when to return it to the system. When it identifies memory that is no longer in use, it frees that memory.

This automatic memory management enables increased abstraction of interfaces and more reliable code. To the programmer, however, the final result is nearly always the same: no more worrying about memory errors.

In order to ease up C/C++ programmer’s life, many memory errors detection tools are available in the market such as Valgrind, Electric Fence, and Purify. These tools can be used to detect some memory defects, but these cannot guarantee their absence.

For the ones who do not have such a tool or don't want to use, there are two strategies available to the programmer. The first strategy must always be to architect memory errors out of your system. The second is the old-school way: code inspection. To restrict the search in order to speed up the debugging process, one should first review the pieces of code with a dense use of pointers and casts.
Memory Layout

The basic process memory layout is the result of "linking" (combining the various .o and .a (or .so) files into an executable program format) plus arranging the disk file’s data into main memory (allocating data storage and such).

**Text Segment:** The text segment contains the actual code to be executed. It's usually sharable, so multiple instances of a program can share the text segment to lower memory requirements. This segment is usually marked read-only so a program can't modify its own instructions.

**Initialized Data Segment:** This segment contains global variables which are initialized by the programmer. For example, the C declaration

```
int maxcount = 99;
```

**Uninitialized Data Segment:** Also named "bss" (block started by symbol) which was an operator used by an old assembler. This segment contains uninitialized global variables. All variables in this segment are initialized to 0 or NULL pointers before the program begins to execute. The C declaration
appearing outside any function causes this variable to be stored in the uninitialized data segment.

The stack: The stack is a collection of stack frames. Every time a function is called, an area of memory is set aside, called a stack frame, for the new function call. This area of memory holds some crucial information, like:

1. Storage space for all the automatic variables for the newly called function.
2. The line number of the calling function to return to when the called function returns.
3. The arguments, or parameters, of the called function.

The heap: Most dynamic memory, whether requested via C’s malloc() and friends or C++’s new is doled out to the program from the heap. The C library also gets dynamic memory for its own personal workspace from the heap as well. As more memory is requested "on the fly", the heap grows as per requirement.

Size() function

The GNU `size' utility lists the section sizes(text, data, and bss segments) for each object files to be examined.

For example:

```
$ size /usr/bin/cc /bin/sh
  text  data  bss  dec   hex   filename
4096  1536  916  82058  1408a  /usr/bin/cc
619234 21120 18260 658614 a0cb6 /bin/sh
```

The fourth and fifth columns are the total of the three sizes, displayed in decimal and hexadecimal, respectively.
Memory Problems Taxonomy

Typically, the following four kinds of memory problems are encountered by developers:

1. **Memory leaks**
   Memory leaks occur when a block of memory is allocated, but it is never released back to the system after it is no longer needed.

   If enough leaks occur, they can keep consuming memory until all available memory is exhausted and when memory leaks are serious, applications can be terminated due to a shortage of memory. This type of problem is hard to locate.

   ```c
   int f(void)
   {
       char* p;
       p = (char *) malloc(8 * sizeof(char));
       ...
       return 0;
   }
   int main(void) {
      f();
      ...
   }
   ```

   An easy way to find out if your code is leaking memory is by executing it and examining its memory usage either using Task Manager on Windows or top on Linux.

**Best Practices:**

- If you intend others to inherit from your class, you must declare its destructor virtual.
- Make sure that the default methods that C++ generates work well with your design.
- A good approach to protect from memory leaks is to use a garbage collector and replace new with its allocator. There are a lot of libraries which can provide good garbage collectors, or one can implement their own.
- Another choice can be to use private heaps, that is, you first allocate a big chunk of memory and manage it in a Heap class (whenever you need new memory to be allocated, a request is being made to the Heap class). Instead of deleting every chunk you allocate for every object, you can delete the whole heap when the objects using it are no longer needed. This can also be used just for backup. In case you forget to delete some object you can be sure that by deleting the Heap there will be no memory leaks remaining.
- Smart pointers are another efficient way of fighting memory leaks. They are often used to manage the lifetimes of other dynamically allocated objects.
A smart pointer type is defined as any class type that overloads operator->, operator*, or operator->* and overloading these operators allows a smart pointer to behave much like a built-in pointer.

One smart pointer provided by the standard library (STL) is called auto_ptr and it is declared in the <memory> header. The following piece of code illustrates its usage:

```cpp
void f()
{
    Dialog *dl = new Dialog();
    auto_ptr pdl = dl;
}
```

At the end of the function the auto_ptr object will go out of scope and will delete automatically the memory allocated in the first statement. This above technique is used when there are some exceptions thrown between allocating and deallocating memory and you want to make sure there aren't going to be any memory leaks.

2. **Duplicate release or Double Free of memory**

When a block of memory is deleted or deallocated and the pointer to that block of memory is not modified, now if this same pointer is used to delete the memory again then the result of the second deletion is unpredictable. Sometimes this silent corruption of memory data can lead to the bugs that are extremely hard to solve.

```cpp
int* p;
p = new int;
...
delete p;
```

Another kind of problem also exists

```cpp
class Contact
{
    char* name;
    int age;

public:
    Contact( const char* inName, inAge )
    {
        name = new char[strlen( inName ) + 1];
        strcpy( name, inName );
        age = inAge;
    }

    ~Contact()
    {
        delete[] name;
    }
};
```
Contact c1("Fred", 40);
Contact c2 = c1;

The problem here is- c1 and c2 will have the same pointer value for the "name" field. When c2 goes out of scope, its destructor will get called and delete the memory that was allocated when c1 was constructed (because the name field of both objects have the same pointer value). Then, when c1 destructs, it will attempt to delete the pointer value, and a "double-free" occur. At best, the heap will catch the problem and report an error. At worst, the same pointer value may, by then, be allocated to another object, the delete will free the wrong memory, and this will introduce a difficult-to-find bug in the code.

Best Practices:

- Assign NULL to a pointer after freeing (or deleting) it. This prevents the program from crashing should the pointer be accidentally freed again. Calling free or delete on NULL pointers is guaranteed not to cause a problem.
- Write your own copy constructors and assignment operators that will work correctly with your classes.

3. Accessing memory incorrectly

Incorrect memory access includes accessing

- a memory address that has been released (also known as dangling reference)

   It is possible to deallocate the space used by an object to which some other object still has a reference. If the object with that (dangling) reference tries to access the original object, but the space has been reallocated to a new object, the result is unpredictable and not what was intended.

```c
char *a = malloc(128*sizeof(char));
char *b = malloc(128*sizeof(char));
b = a;
free(a);
free(b); // will not free the pointer to the original allocated memory
```

- a local pointer variable not initialized (i.e., it then points at some random location), or by setting the pointer to an incorrect value., or

- Reading/writing to memory out of the bounds of a static array.

```c
char *f(void)
{
    char p[8];
    ... return p;
}
int main(void) {
```
Best Practices:

- Add an explicit copy constructor and an assignment operator to the class or struct which contains pointer fields.
- Disable copying and assignment altogether by making the copy constructor and assignment operator private.
- Modify your classes so that the default copy constructor and assignment operator are correct (by using member objects instead of dynamic allocation, or certain types of smart pointers such as the shared_ptr from Boost.org).

4. Memory allocation failure

If there is no memory available or if the program has exceeded the amount of memory it is allowed to reference, `malloc` function call will return a null pointer, which should always be checked for after allocation. Many programs do not check for `malloc` failure. Such a program would attempt to use the null pointer returned by `malloc` as if it pointed to allocated memory, and the program would crash.

Memory allocation may also fail when the requested memory block is not available in contigous address space. This situation is often referred as "fragmentation".

The best way to understand memory fragmentation is to look at an example. For this example, it is assumed that there is a 10K heap. First, an area of 3K is requested, thus:

```c
#define K (1024)
char *p1;
p1 = malloc(3*K);
```

Then, a further 4K is requested:
```
p2 = malloc(4*K);
```

The resulting situation is shown in Figure 2 - 3K of memory is now free.

Sometime later, the first memory allocation, pointed to by `p1`, is de-allocated:
```
free(p1);
```

This leaves 6K of memory free in two 3K chunks, as illustrated in Figure 3. A further request for a 4K allocation is issued:
```
p1 = malloc(4*K);
```
This results in a **failure** - NULL is returned into `p1` - because, even though 6K of memory is available, there is not a 4K contiguous block available. This is memory fragmentation.

It would seem that an obvious solution would be to de-fragment the memory, merging the two 3K blocks to make a single one of 6K. However, this is not possible because it would entail moving the 4K block to which `p2` points. Moving it would change its address, so any code that has taken a copy of the pointer would then be broken. In other languages (such as Visual Basic, Java and C#), there are de-fragmentation (or “garbage collection”) facilities. This is only possible because these languages do not support direct pointers, so moving the data has no adverse effect upon application code. This de-fragmentation may occur when a memory allocation fails or there may be a periodic garbage collection process that is run. In either case, this would severely compromise real time performance and determinism.

**Best practices:**

- When using `new`, enclose it within a `try-catch` block. The `new` operator throws an exception and does not return a value. To force the `new` operator to return a value, use the `nothrow` qualifier as shown below:

```cpp
int * pt = new (std::nothrow) int[100];
```

- Using an array member will be cleaner (more succinct, less error prone) and faster as there is no need to call allocation and deallocation functions.

```cpp
float x[6];
vs.
float *x;  // in class definition
x = new float[6]; // in class constructor
delete [] x;     // in class destructor
```

**Other Generic Best practices in C/C++**

- When storing secrets such as passwords in memory, overwrite them with random data before deleting them. Need to note that `free` and `delete` merely make previously allocated memory unavailable, they don't really 'delete' data contained in that memory.

- Do not allocate and deallocate memory in a loop as this may slow down the program and may sometime cause security malfunctions.
• Use **unsigned integer** types to hold the number of bytes to be allocated, when allocating memory dynamically. This weeds out negative numbers. Also check the length of memory allocated against a maximum value.

• When using C and C++ code together, if `new` has been used to allocate memory, use `delete` to free it. Do not use `free`. Likewise, if `malloc` or `calloc` has been used to allocate memory, use `free` when deallocating. Do not use `delete`.

**Common Memory Problems**

Here are some memory problems that frequently occur in every developer’s life and what does each problem imply.

<table>
<thead>
<tr>
<th>Symptom</th>
<th>Implies</th>
</tr>
</thead>
<tbody>
<tr>
<td>“random” failures, especially on call return</td>
<td>Corrupted the stack frame return info</td>
</tr>
<tr>
<td>calls to <code>malloc/free</code> fail</td>
<td>Corrupted the malloc bookkeeping data</td>
</tr>
<tr>
<td>program magically works if printf inserted</td>
<td>Corrupted storage space in stack frame</td>
</tr>
</tbody>
</table>

**Debugging Memory Techniques**

There are some simple techniques that can be used to debug crash issues:

1. **Obtain Stack Dump**

Make sure that every embedded processor in the system supports dumping of the stack at the time of crash. The crash dump should be saved in non volatile memory so that it can be retrieved by tools on processor reboot. In fact attempt should be made to save as much as possible of processor state and key data structures at the time of crash.

2. **Using assert**

An ounce of prevention is better than a pound of cure. Detecting crash causing conditions by using assert macro can be a very useful tool in detecting problems much before they lead to a crash. Basically assert macros check for a condition which the function assumes to be true. For example, the code below shows an assertion which checks that the message to be processed is non NULL. During initial debugging of the system this assert condition might help you detect the condition, before it leads to a crash.

Note that asserts do not have any overhead in the shipped system as in the release builds asserts are defined to a NULL macro, effectively removing all the assert conditions.

```c
void function(const char *pMsg)
{
    assert(pMsg);
    . . .
}
```
3. Defensive Checks and Tracing

Similar to asserts, use of defensive checks can many times save the system from a crash even when an invalid condition is detected. The main difference here is that unlike asserts, defensive checks remain in the shipped system.

Tracing and maintaining event history can also be very useful in debugging crashes in the early phase of development. However tracing of limited use in debugging systems when the system has been shipped.

Conclusion

This paper has focused on understanding the most difficult aspect: memory management. C++ memory management is an enormously useful tool for creating elegant software. The common memory-related errors discussed in this paper can be used as a handy reference, to find and avoid such errors in your own code.