

第十一章 内存管理

欢迎进入内存这片雷区。伟大的 Bill Gates 曾经失言：

640K ought to be enough for everybody

— Bill Gates 1981

程序员们经常编写内存管理程序，往往提心吊胆。如果不想触雷，唯一的解决办法就是发现所有潜伏的地雷并且排除它们，躲是躲不了的。本章的内容比一般教科书的要深入得多，读者需细心阅读，做到真正地通晓内存管理。

11.1 内存分配方式

内存分配方式有三种：

- (1) 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，`static` 变量。
- (2) 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
- (3) 从堆上分配，亦称动态内存分配。程序在运行的时候用 `malloc` 或 `new` 申请任意多少的内存，程序员自己负责在何时用 `free` 或 `delete` 释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

11.2 常见的内存错误及其对策

发生内存错误是件非常麻烦的事情。编译器不能自动发现这些错误，通常是在程序运行时才能捕捉到。而这些错误大多没有明显的症状，时隐时现，增加了改错的难度。有时用户怒气冲冲地把你找来，程序却没有发生任何问题，你一走，错误又发作了。

常见的内存错误及其对策如下：

- ◆ 内存分配未成功，却使用了它。

编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检查指针是否为 `NULL`。如果指针 `p` 是函数的参数，那么在函数的入口处用 `assert(p!=NULL)` 进行检查。如果是用 `malloc` 或 `new` 来申请内存，应该用 `if(p==NULL)` 或 `if(p!=NULL)` 进行防错处理。

- ◆ 内存分配虽然成功，但是尚未初始化就引用它。

犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。

内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。

◆ 内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多 1”或者“少 1”的操作。特别是在 for 循环语句中，循环次数很容易搞错，导致数组操作越界。

◆ 忘记了释放内存，造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然死掉，系统出现提示：内存耗尽。

动态内存的申请与释放必须配对，程序中 malloc 与 free 的使用次数一定要相同，否则肯定有错误（new/delete 同理）。

◆ 释放了内存却继续使用它。

有三种情况：

（1）程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。

（2）函数的 return 语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存存在函数体结束时被自动销毁。

（3）使用 free 或 delete 释放了内存后，没有将指针设置为 NULL。导致产生“野指针”。

● **【规则 11-2-1】** 用 malloc 或 new 申请内存之后，应该立即检查指针值是否为 NULL。防止使用指针值为 NULL 的内存。

● **【规则 11-2-2】** 不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。

● **【规则 11-2-3】** 避免数组或指针的下标越界，特别要当心发生“多 1”或者“少 1”操作。

● **【规则 11-2-4】** 动态内存的申请与释放必须配对，防止内存泄漏。

● **【规则 11-2-5】** 用 free 或 delete 释放了内存之后，立即将指针设置为 NULL，防止产生“野指针”。

11.3 指针与数组的对比

C++/C 程序中，指针和数组在不少地方可以相互替换着用，让人产生一种错觉，以

为两者是等价的。

数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。数组名对应着（而不是指向）一块内存，其地址与容量在生命期内保持不变，只有数组的内容可以改变。

指针可以随时指向任意类型的内存块，它的特征是“可变”，所以我们常用指针来操作动态内存。指针远比数组灵活，但也更危险。

下面以字符串为例比较指针与数组的特性。

11.3.1 修改内容

示例 11-3-1 中，字符数组 `a` 的容量是 6 个字符，其内容为 `hello\0`。`a` 的内容可以改变，如 `a[0]='X'`。指针 `p` 指向常量字符串“world”（位于静态存储区，内容为 `world\0`），常量字符串的内容是不可以被修改的。从语法上看，编译器并不觉得语句 `p[0]='X'` 有什么不妥，但是该语句企图修改常量字符串的内容而导致运行错误。

```
char a[] = "hello";
a[0] = 'X';
cout << a << endl;
char *p = "world";    // 注意 p 指向常量字符串
p[0] = 'X';           // 编译器不能发现该错误
cout << p << endl;
```

示例 11-3-1 修改数组和指针的内容

11.3.2 内容复制与比较

不能对数组名进行直接复制与比较。示例 11-3-2 中，若想把数组 `a` 的内容复制给数组 `b`，不能用语句 `b = a`，否则将产生编译错误。应该用标准库函数 `strcpy` 进行复制。同理，比较 `b` 和 `a` 的内容是否相同，不能用 `if(b==a)` 来判断，应该用标准库函数 `strcmp` 进行比较。

语句 `p = a` 并不能把 `a` 的内容复制指针 `p`，而是把 `a` 的地址赋给了 `p`。要想复制 `a` 的内容，可以先用库函数 `malloc` 为 `p` 申请一块容量为 `strlen(a)+1` 个字符的内存，再用 `strcpy` 进行字符串复制。同理，语句 `if(p==a)` 比较的不是内容而是地址，应该用库函数 `strcmp` 来比较。

```
// 数组...
char a[] = "hello";
char b[10];
strcpy(b, a);          // 不能用 b = a;
if(strcmp(b, a) == 0)  // 不能用 if (b == a)
```

```

...
// 指针...
int len = strlen(a);
char *p = (char *)malloc(sizeof(char)*(len+1));
strcpy(p, a);           // 不要用 p = a;
if(strcmp(p, a) == 0)   // 不要用 if (p == a)
...

```

示例 11-3-2 数组和指针的内容复制与比较

11.3.3 计算内存容量

用运算符 `sizeof` 可以计算出数组的容量（字节数）。示例 11-3-3 (a) 中，`sizeof(a)` 的值是 12（注意别忘了 `'\0'`）。指针 `p` 指向 `a`，但是 `sizeof(p)` 的值却是 4。这是因为 `sizeof(p)` 得到的是一个指针变量的字节数，相当于 `sizeof(char*)`，而不是 `p` 所指的内存容量。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。

注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。示例 11-3-3 (b) 中，不论数组 `a` 的容量是多少，`sizeof(a)` 始终等于 `sizeof(char *)`。

```

char a[] = "hello world";
char *p = a;
cout<< sizeof(a) << endl; // 12 字节
cout<< sizeof(p) << endl; // 4 字节

```

示例 11-3-3 (a) 计算数组和指针的内存容量

```

void Func(char a[100])
{
    cout<< sizeof(a) << endl; // 4 字节而不是 100 字节
}

```

示例 11-3-3 (b) 数组退化为指针

11.4 指针参数是如何传递内存的

如果函数的参数是一个指针，不要指望用该指针去申请动态内存。示例 11-4-1 中，`Test` 函数的语句 `GetMemory(str, 200)` 并没有使 `str` 获得期望的内存，`str` 依旧是 `NULL`，为什么？

```
void GetMemory(char *p, int num)
{
    p = (char *)malloc(sizeof(char) * num);
}
```

```
void Test(void)
{
    char *str = NULL;
    GetMemory(str, 100); // str 仍然为 NULL
    strcpy(str, "hello"); // 运行错误
}
```

示例 11-4-1 试图用指针参数申请动态内存

毛病出在函数 `GetMemory` 中。编译器总是要为函数的每个参数制作临时副本，指针参数 `p` 的副本是 `_p`，编译器使 `_p = p`。如果函数体内的程序修改了 `_p` 的内容，就导致参数 `p` 的内容作相应的修改。这就是指针可以用作输出参数的原因。在本例中，`_p` 申请了新的内存，只是把 `_p` 所指的内存地址改变了，但是 `p` 丝毫未变。所以函数 `GetMemory` 并不能输出任何东西。事实上，每执行一次 `GetMemory` 就会泄露一块内存，因为没有用 `free` 释放内存。

如果非得要用指针参数去申请内存，那么应该改用“指向指针的指针”，见示例 11-4-2。

```
void GetMemory2(char **p, int num)
{
    *p = (char *)malloc(sizeof(char) * num);
}
```

```
void Test2(void)
{
    char *str = NULL;
    GetMemory2(&str, 100); // 注意参数是 &str，而不是 str
    strcpy(str, "hello");
    cout<< str << endl;
    free(str);
}
```

示例 11-4-2 用指向指针的指针申请动态内存

由于“指向指针的指针”这个概念不容易理解，我们可以用函数返回值来传递动态内存。这种方法更加简单，见示例 11-4-3。

```
char *GetMemory3(int num)
```

```

{
    char *p = (char *)malloc(sizeof(char) * num);
    return p;
}

```

```

void Test3(void)
{
    char *str = NULL;
    str = GetMemory3(100);
    strcpy(str, "hello");
    cout<< str << endl;
    free(str);
}

```

示例 11-4-3 用函数返回值来传递动态内存

用函数返回值来传递动态内存这种方法虽然好用，但是常常有人把 `return` 语句用错了。这里强调不要用 `return` 语句返回指向“栈内存”的指针，因为该内存在函数结束时自动消亡，见示例 11-4-4。

```

char *GetString(void)
{
    char p[] = "hello world";
    return p; // 编译器将提出警告
}

```

```

void Test4(void)
{
    char *str = NULL;
    str = GetString(); // str 的内容是垃圾
    cout<< str << endl;
}

```

示例 11-4-4 `return` 语句返回指向“栈内存”的指针

用调试器逐步跟踪 `Test4`，发现执行 `str = GetString` 语句后 `str` 不再是 `NULL` 指针，但是 `str` 的内容不是“hello world”而是垃圾。

如果把示例 11-4-4 改写成示例 11-4-5，会怎么样？

```

char *GetString2(void)
{
    char *p = "hello world";
    return p;
}

```

```

}

void Test5(void)
{
    char *str = NULL;
    str = GetString2();
    cout<< str << endl;
}

```

示例 11-4-5 return 语句返回常量字符串

函数 Test5 运行虽然不会出错，但是函数 GetString2 的设计概念却是错误的。因为 GetString2 内的 “hello world” 是常量字符串，位于静态存储区，它在程序生命期内恒定不变。无论什么时候调用 GetString2，它返回的始终是同一个 “只读” 的内存块。

11.5 free 和 delete 把指针怎么啦

别看 free 和 delete 的名字恶狠狠的（尤其是 delete），它们只是把指针所指的内存给释放掉，但并没有把指针本身干掉。

用调试器跟踪示例 11-5，发现指针 p 被 free 以后其地址仍然不变（非 NULL），只是该地址对应的内存是垃圾，p 成了 “野指针”。如果此时不把 p 设置为 NULL，会让人误以为 p 是个合法的指针。

如果程序比较长，我们有时记不住 p 所指的内存是否已经被释放，在继续使用 p 之前，通常会用语句 if (p != NULL) 进行防错处理。很遗憾，此时 if 语句起不到防错作用，因为即便 p 不是 NULL 指针，它也不指向合法的内存块。

```

char *p = (char *) malloc(100);
strcpy(p, "hello");
free(p);          // p 所指的内存被释放，但是 p 所指的地址仍然不变
...
if(p != NULL)    // 没有起到防错作用
{
    strcpy(p, "world"); // 出错
}

```

示例 11-5 p 成为野指针

11.6 动态内存会被自动释放吗？

函数体内的局部变量在函数结束时自动消亡。很多人误以为示例 11-6 是正确的。理

由是 `p` 是局部的指针变量，它消亡的时候会让它所指的动态内存一起完蛋。这是错觉！

```
void Func(void)
{
    char *p = (char *) malloc(100);    // 动态内存会自动释放吗？
}
```

示例 11-6 试图让动态内存自动释放

我们发现指针有一些“似是而非”的特征：

(1) 指针消亡了，并不表示它所指的内存会被自动释放。

(2) 内存被释放了，并不表示指针会消亡或者成了 **NULL** 指针。

这表明释放内存并不是一件可以草率对待的事。也许有人不服气，一定要找出可以草率行事的理由：

如果程序终止了运行，一切指针都会消亡，动态内存会被操作系统回收。既然如此，在程序临终前，就可以不必释放内存、不必将指针设置为 `NULL` 了。终于可以偷懒而不会发生错误了吧？

想得美。如果别人把那段程序取出来用到其它地方怎么办？

11.7 杜绝“野指针”

“野指针”不是 `NULL` 指针，是指向“垃圾”内存的指针。人们一般不会错用 `NULL` 指针，因为用 `if` 语句很容易判断。但是“野指针”是很危险的，`if` 语句对它不起作用。

“野指针”的成因主要有两种：

(1) 指针变量没有被初始化。任何指针变量刚被创建时不会自动成为 `NULL` 指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为 `NULL`，要么让它指向合法的内存。例如

```
char *p = NULL;
char *str = (char *) malloc(100);
```

(2) 指针 `p` 被 `free` 或者 `delete` 之后，没有置为 `NULL`，让人误以为 `p` 是个合法的指针。参见 11.5 节。

(3) 指针操作超越了变量的作用范围。这种情况让人防不胜防，示例程序如下：

```
class A
{
public:
    void Func(void){ cout << "Func of class A" << endl; }
};
```



```

void Test(void)
{
    A *p;
    {
        A a;
        p = &a; // 注意 a 的生命期
    }
    p->Func();    // p 是“野指针”
}

```

函数 Test 在执行语句 p->Func() 时，对象 a 已经消失，而 p 是指向 a 的，所以 p 就成了“野指针”。但奇怪的是我运行这个程序时居然没有出错。

有位读者指出可能的原因：成员函数 Func() 不是虚函数，它的功能是在编译时刻就决定了的。而且，Func() 中没有引用任何数据成员，所以不会出错啦！

11.8 有了 malloc/free 为什么还要 new/delete

malloc 与 free 是 C++/C 语言的标准库函数，new/delete 是 C++ 的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用 malloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。

因此 C++ 语言需要一个能完成动态内存分配和初始化工作的运算符 new，以及一个能完成清理与释放内存工作的运算符 delete。注意 new/delete 不是库函数。

我们先看一看 malloc/free 和 new/delete 如何实现对象的动态内存管理，见示例 11-8。

```

class Obj
{
public :
    Obj(void){ cout << "Initialization" << endl; }
    ~Obj(void){ cout << "Destroy" << endl; }
    void Initialize(void){ cout << "Initialization" << endl; }
    void Destroy(void){ cout << "Destroy" << endl; }
};

void UseMallocFree(void)
{
    Obj *a = (obj *)malloc(sizeof(obj)); // 申请动态内存

```

```

        a->Initialize();                // 初始化
        //...
        a->Destroy(); // 清除工作
        free(a);           // 释放内存
    }

void UseNewDelete(void)
{
    Obj  *a = new Obj; // 申请动态内存并且初始化
    //...
    delete a;          // 清除并且释放内存
}

```

示例 11-8 用 malloc/free 和 new/delete 实现对象的动态内存管理

类 Obj 的函数 Initialize 模拟了构造函数的功能，函数 Destroy 模拟了析构函数的功能。函数 UseMallocFree 中，由于 malloc/free 不能执行构造函数与析构函数，必须调用成员函数 Initialize 和 Destroy 来完成初始化与清除工作。函数 UseNewDelete 则简单得多。

所以我们不要企图用 malloc/free 来完成动态对象的内存管理，应该用 new/delete。由于内部数据类型的“对象”没有构造与析构的过程，对它们而言 malloc/free 和 new/delete 是等价的。

既然 new/delete 的功能完全覆盖了 malloc/free，为什么 C++ 不把 malloc/free 淘汰出局呢？这是因为 C++ 程序经常要调用 C 函数，而 C 程序只能用 malloc/free 管理动态内存。

如果用 free 释放“new 创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用 delete 释放“malloc 申请的动态内存”，理论上讲程序不会出错，但是该程序的可读性很差。所以 new/delete 必须配对使用，malloc/free 也一样。

11.9 内存耗尽怎么办？

如果在申请动态内存时找不到足够大的内存块，malloc 和 new 将返回 NULL 指针，宣告内存申请失败。通常有三种方式处理“内存耗尽”问题。

(1) 判断指针是否为 NULL，如果是则马上用 return 语句终止本函数。例如：

```

void Func(void)
{
    A  *a = new A;
    if(a == NULL)
    {
        return;
    }
    ...
}

```

```
}
```

(2) 判断指针是否为 NULL，如果是则马上用 `exit(1)` 终止整个程序的运行。例如：

```
void Func(void)
{
    A *a = new A;
    if(a == NULL)
    {
        cout << "Memory Exhausted" << endl;
        exit(1);
    }
    ...
}
```

(3) 为 `new` 和 `malloc` 设置异常处理函数。例如 Visual C++ 可以用 `_set_new_handler` 函数为 `new` 设置用户自己定义的异常处理函数，也可以让 `malloc` 享用与 `new` 相同的异常处理函数。详细内容请参考 C++ 使用手册。

上述 (1) (2) 方式使用最普遍。如果一个函数内有多处需要申请动态内存，那么方式 (1) 就显得力不从心（释放内存很麻烦），应该用方式 (2) 来处理。

很多人不忍心用 `exit(1)`，问：“不编写出错处理程序，让操作系统自己解决行不行？”

不行。如果发生“内存耗尽”这样的事情，一般说来应用程序已经无药可救。如果不用 `exit(1)` 把坏程序杀死，它可能会害死操作系统。道理如同：如果不把该死的歹徒击毙，歹徒在老死之前会犯下更多的罪恶。

有一个很重要的现象要告诉大家。对于 32 位以上的应用程序而言，无论怎样使用 `malloc` 与 `new`，几乎不可能导致“内存耗尽”。我在 Windows 98 下用 Visual C++ 编写了测试程序，见示例 11-9。这个程序会无休止地运行下去，根本不会终止。因为 32 位操作系统支持“虚存”，内存用完了，自动用硬盘空间顶替。我只听到硬盘嘎吱嘎吱地响，Window 98 已经累得对键盘、鼠标毫无反应。

我可以得出这么一个结论：对于 32 位以上的应用程序，“内存耗尽”错误处理程序毫无用处。这下可把 Unix 和 Windows 程序员们乐坏了：反正错误处理程序不起作用，我就不写了，省了很多麻烦。

我不想误导读者，必须强调：不加错误处理将导致程序的质量很差，千万不可因小失大。

```
void main(void)
{
```

```

float *p = NULL;
while(TRUE)
{
    p = new float[1000000];
    cout << "eat memory" << endl;
    if(p==NULL)
        exit(1);
}
}

```

示例 11-9 试图耗尽操作系统的内存

11.10 malloc/free 的使用要点

函数 malloc 的原型如下：

```
void * malloc(size_t size);
```

用 malloc 申请一块长度为 length 的整数类型的内存，程序如下：

```
int *p = (int *) malloc(sizeof(int) * length);
```

我们应当把注意力集中在两个要素上：“类型转换”和“sizeof”。

- ◆ malloc 返回值的类型是 void *，所以在调用 malloc 时要显式地进行类型转换，将 void * 转换成所需要的指针类型。
- ◆ malloc 函数本身并不识别要申请的内存是什么类型，它只关心内存的总字节数。我们通常记不住 int, float 等数据类型的变量的确切字节数。例如 int 变量在 16 位系统下是 2 个字节，在 32 位下是 4 个字节；而 float 变量在 16 位系统下是 4 个字节，在 32 位下也是 4 个字节。最好用以下程序作一次测试：

```

cout << sizeof(char) << endl;
cout << sizeof(int) << endl;
cout << sizeof(unsigned int) << endl;
cout << sizeof(long) << endl;
cout << sizeof(unsigned long) << endl;
cout << sizeof(float) << endl;
cout << sizeof(double) << endl;
cout << sizeof(void *) << endl;

```

在 malloc 的“()”中使用 sizeof 运算符是良好的风格，但要当心有时我们会昏了头，写出 p = malloc(sizeof(p)) 这样的程序来。

- ◆ 函数 free 的原型如下：

```
void free( void * memblock );
```

为什么 free 函数不象 malloc 函数那样复杂呢？这是因为指针 p 的类型以及它所指

的内存的容量事先都是知道的，语句 `free(p)` 能正确地释放内存。如果 `p` 是 `NULL` 指针，那么 `free` 对 `p` 无论操作多少次都不会出问题。如果 `p` 不是 `NULL` 指针，那么 `free` 对 `p` 连续操作两次就会导致程序运行错误。

11.11 new/delete 的使用要点

运算符 `new` 使用起来要比函数 `malloc` 简单得多，例如：

```
int *p1 = (int *)malloc(sizeof(int) * length);
int *p2 = new int[length];
```

这是因为 `new` 内置了 `sizeof`、类型转换和类型安全检查功能。对于非内部数据类型的对象而言，`new` 在创建动态对象的同时完成了初始化工作。如果对象有多个构造函数，那么 `new` 的语句也可以有多种形式。例如

```
class Obj
{
public :
    Obj(void);    // 无参数的构造函数
    Obj(int x);   // 带一个参数的构造函数
    ...
}

void Test(void)
{
    Obj *a = new Obj;
    Obj *b = new Obj(1); // 初值为 1
    ...
    delete a;
    delete b;
}
```

如果用 `new` 创建对象数组，那么只能使用对象的无参数构造函数。例如

```
Obj *objects = new Obj[100]; // 创建 100 个动态对象
```

不能写成

```
Obj *objects = new Obj[100](1); // 创建 100 个动态对象的同时赋初值 1
```

在用 `delete` 释放对象数组时，留意不要丢了符号 `[]`。例如

```
delete []objects; // 正确的用法
delete objects; // 错误的用法
```

后者相当于 `delete objects[0]`，漏掉了另外 99 个对象。

11.12 一些心得体会

我认识不少技术不错的 C++/C 程序员，很少有人能拍拍胸脯说通晓指针与内存管理（包括我自己）。我最初学习 C 语言时特别怕指针，导致我开发第一个应用软件（约 1 万行 C 代码）时没有使用一个指针，全用数组来顶替指针，实在蠢笨得过分。躲避指针不是办法，后来我改写了这个软件，代码量缩小到原先的一半。

我的经验教训是：

- （1）越是怕指针，就越要使用指针。不会正确使用指针，肯定算不上是合格的程序员。
- （2）必须养成“使用调试器逐步跟踪程序”的习惯，只有这样才能发现问题的本质。