

看完這篇你還能不懂C語言/C++內存管理？

C語言Plus 今天

以下文章來源於C語言與CPP編程，作者自稱一派123



C語言與CPP編程

分享C語言/C++，數據結構與算法，計算機基礎，操作系統等





C 語言內存管理指對系統內存的分配、創建、使用這一系列操作。在內存管理中，由於是操作系統內存，使用不當會造成畢竟麻煩的結果。本文將從系統內存的分配、創建出發，並且使用例子來舉例說明內存管理不當會出現的情況及解決辦法。

一、內存

在計算機中，每個應用程序之間的內存是相互獨立的，通常情況下應用程序A並不能訪問應用程序B，當然一些特殊技巧可以訪問，但此文並不詳細進行說明。例如在計算機中，一個視頻播放程序與一個瀏覽器程序，它們的內存並不能訪問，每個程序所擁有的內存是分區進行管理的。

在計算機系統中，運行程序A將會在內存中開闢程序A的內存區域1，運行程序B將會在內存中開闢程序B的內存區域2，內存區域1與內存區域2之間邏輯分隔。



1.1 內存四區

在程序A開闢的內存區域1會被分為幾個區域，這就是**內存四區**，內存四區分為棧區、堆區、數據區與代碼區。





棧區指的是存儲一些臨時變量的區域，臨時變量包括了局部變量、返回值、參數、返回地址等，當這些變量超出了當前作用域時將會自動彈出。該棧的最大存儲是有大小的，該值固定，超過該大小將會造成棧溢出。

堆區指的是一個比較大的內存空間，主要用於對動態內存的分配；在程序開發中一般是開發人員進行分配與釋放，若在程序結束時都未釋放，系統將會自動進行回收。

數據區指的是主要存放全局變量、常量和靜態變量的區域，數據區又可以進行劃分，分為全局區與靜態區。全局變量與靜態變量將會存放至該區域。

代碼區就比較好理解了，主要是存儲可執行代碼，該區域的屬性是只讀的。

1.2 使用代碼證實內存四區的底層結構

由於棧區與堆區的底層結構比較直觀的表現，在此使用代碼只演示這兩個概念。首先查看代碼觀察棧區的內存地址分配情況：



```
#include<stdio.h>

int main()
{
```

```
int a = 0;
int b = 0;
char c = '0';
printf("变量a的地址是：%d\n变量b的地址是：%d\n变量c的地址是：%d\n", &a, &b, &c);
}
```

運行結果為：

```
变量a的地址是：2293324
变量b的地址是：2293320
变量c的地址是：2293319
```

我們可以觀察到變量a的地址是2293324 變量b的地址是2293320，由於int的數據大小為4 所以兩者之間間隔為4；再查看變量c，我們發現變量c的地址為2293319，與變量b的地址2293324 間隔1，因為c的數據類型為char，類型大小為1。在此我們觀察發現，明明我創建變量的時候順序是a 到b 再到c，為什麼它們之間的地址不是增加而是減少呢？那是因為棧區的一種數據存儲結構為先進後出，如圖：



首先棧的頂部為地址的“最小”索引，隨後往下依次增大，但是由於堆棧的特殊存儲結構，我們將變量a 先進行存儲，那麼它的一個索引地址將會是最大的，隨後依次減少；第二次存儲的值是b，該值的地址索引比a 小，由於int 的數據大小為4，所以在a 地址為2293324 的基礎上往上減少4 為2293320，在存儲c 的時候為char，大小為1，則地址為2293319。由於a、b、c 三個變量同屬於一個棧內，所以它們地址的索引是連續性的，那如果我創建一個靜態變量將會如何？在以上內容中說明了靜態變量存儲在靜態區內，我們現在就來證實一下：

```
#include<stdio.h>

int main()
{

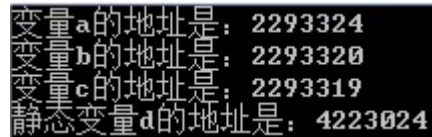
    int a = 0;
    int b = 0;
    char c='0';
    static int d = 0;

    printf("变量a的地址是：%d\n变量b的地址是：%d\n变量c的地址是：%d\n", &a, &b, &c);

    printf("静态变量d的地址是：%d\n", &d);

}
```

運行結果如下：



```
变量a的地址是：2293324
变量b的地址是：2293320
变量c的地址是：2293319
静态变量d的地址是：4223024
```

以上代碼中創建了一個變量d，變量d 為靜態變量，運行代碼後從結果上得知，靜態變量d 的地址與一般變量a、b、c 的地址並不存在連續，他們兩個的內存地址是分開的。那接下來在此建一個全局變量，通過上述內容得知，全局變量與靜態變量都應該存儲在靜態區，代碼如下：

```
#include<stdio.h>

int e = 0;

int main()
{

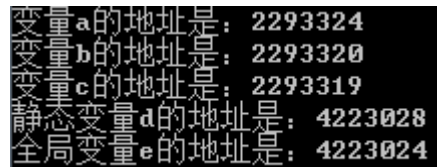
    int a = 0;
    int b = 0;
    char c='0';
    static int d = 0;

    printf("变量a的地址是：%d\n变量b的地址是：%d\n变量c的地址是：%d\n", &a, &b, &c);

    printf("静态变量d的地址是：%d\n", &d);
    printf("全局变量e的地址是：%d\n", &e);

}
```

運行結果如下：



```
变量a的地址是：2293324
变量b的地址是：2293320
变量c的地址是：2293319
静态变量d的地址是：4223028
全局变量e的地址是：4223024
```

从以上运行结果中证实了上述内容的真实性，并且也得到了一个知识点，栈区、数据区都是使用栈结构对数据进行存储。

在以上内容中还说明了一点栈的特性，就是容量具有固定大小，超过最大容量将会造成栈溢出。查看如下代码：

```
#include<stdio.h>

int main()
{
    char arr_char[1024*1000000];
    arr_char[0] = '0';
}
```

以上代码定义了一个字符数组 `arr_char`，并且设置了大小为 `1024*1000000`，设置该数据是方便查看大小；随后在数组头部进行赋值。运行结果如下：



这是程序运行出错，原因是造成了栈的溢出。在平常开发中若需要大容量的内存，需要使用堆。

堆并没有栈一样的结构，也没有栈一样的先进后出。需要人为的对内存进行分配使用。代码如下：

```
#include<stdio.h>
#include<string.h>
#include <malloc.h>

int main()
{
    char *p1 = (char *)malloc(1024*1000000);
}
```



```
strcpy(p1, "这里是堆区");  
printf("%s\n", p1);  
}
```

以上代码中使用了strcpy 往手动开辟的内存空间 p1 中传数据“这里是堆区”，手动开辟空间使用 malloc，传入申请开辟的空间大小 1024*1000000，在栈中那么大的空间必定会造成栈溢出，而堆本身就是大容量，则不会出现该情况。随后输出开辟的内存中内容，运行结果如下：



在此要注意p1是表示开辟的内存空间地址。

二、malloc 和 free

在 C 语言 (不是 C++) 中，malloc 和 free 是系统提供的函数，成对使用，用于从堆中分配和释放内存。malloc 的全称是 memory allocation 译为“动态内存分配”。

2.1 malloc 和 free 的使用

在开辟堆空间时我们使用的函数为 malloc，malloc 在 C 语言中是用于申请内存空间，malloc 函数的原型如下：

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

在 `malloc` 函数中，`size` 是表示需要申请的内存空间大小，申请成功将会返回该内存空间的地址；申请失败则会返回 `NULL`，并且申请成功也不会自动进行初始化。

细心的同学可能会发现，该函数的返回值说明为 `void *`，在这里 `void *` 并不指代某一种特定的类型，而是说明该类型不确定，通过接收的指针变量从而进行类型的转换。在分配内存时需要注意，即时在程序关闭时系统会自动回收该手动申请的内存，但也要进行手动的释放，保证内存能够在不需要时返回至堆空间，使内存能够合理的分配使用。

释放空间使用 `free` 函数，函数原型如下：

```
void free(void *ptr);
```

`free` 函数的返回值为 `void`，没有返回值，接收的参数为使用 `malloc` 分配的内存空间指针。一个完整的堆内存申请与释放的例子如下：

```
#include<stdio.h>
#include<string.h>
#include <malloc.h>

int main() {
    int n, *p, i;
    printf("请输入一个任意长度的数字来分配空间:");
    scanf("%d", &n);

    p = (int *)malloc(n * sizeof(int));
    if(p==NULL){
        printf("申请失败\n");
    }
}
```

```
    return 0;
}
else{
    printf("申請成功\n");
}

memset(p, 0, n * sizeof(int)); //填充0

//查看
for (i = 0; i < n; i++)
    printf("%d ", p[i]);
printf("\n");

free(p);
p = NULL;
return 0;
}
```

以上代碼中使用了 `malloc` 創建了一個由用戶輸入創建指定大小的內存，判斷了內存地址是否創建成功，且使用了 `memset` 函數對該內存空間進行了填充值，隨後使用 `for` 循環進行了查看。最後使用了 `free` 釋放了內存，並且將 `p` 賦值 `NULL`，這點需要主要，不能使指針指向未知的地址，要置於 `NULL`；否則在之後的開發者會誤以為是個正常的指針，就有可能再通過指針去訪問一些操作，但是在這時該指針已經無用，指向的內存也不知此時被如何使用，這時若出現意外將會造成無法預估的後果，甚至導致系統崩潰，在 `malloc` 的使用中更需要需要。

2.2 內存洩漏與安全使用實例與講解

內存洩漏是指在動態分配的內存中，並沒有釋放內存或者一些原因造成了內存無法釋放，輕度則造成系統的內存資源浪費，嚴重的導致整個系統崩潰等情況的發生。

！内存泄漏

内存泄漏通常比较隐蔽，且少量的内存泄漏发生不一定会发生无法承受的后果，但由于该错误的积累将会造成整体系统的性能下降或系统崩溃。特别是在较为大型的系统中，如何有效的防止内存泄漏等问题的出现变得尤为重要。例如一些长时间的程序，若在运行之初有少量的内存泄漏的问题产生可能并未呈现，但随着运行时间的增长、系统业务处理的增加将会累积出现内存泄漏这种情况；这时极大的会造成不可预知的后果，如整个系统的崩溃，造成的损失将会难以承受。由此防止内存泄漏对于底层开发人员来说尤为重要。

C 程序员在开发过程中，不可避免的面对内存操作的问题，特别是频繁的申请动态内存时会及其容易造成内存泄漏事故的发生。如申请了一块内存空间后，未初始化便读其中的内容、间接申请动态内存但并没有进行释放、释放完一块动态申请的内存后继续引用该内存内容；如上所述这种问题都是出现内存泄漏的原因，往往这些原因由于过于隐蔽在测试时不一定会完全清楚，将会导致在项目上线后的长时间运行下，导致灾难性的后果发生。

如下是一个在子函数中进行了内存空间的申请，但是并未对其进行释放：

```
#include<stdio.h>
#include<string.h>
#include <malloc.h>

void m() {
    char *p1;

    p1 = malloc(100);

    printf("开始对内存进行泄漏...");
}

int main() {
    m();
}
```

```
    return 0;
}
```

如上代碼中，使用 `malloc` 申請了 100 個單位的內存空間後，並沒有進行釋放。假設該 `m` 函數在當前系統中調用頻繁，那將會每次使用都將會造成 100 個單位的內存空間不會釋放，久而久之就會造成嚴重的後果。理應在 `p1` 使用完後添加 `free` 進行釋放：

```
free(p1);
```

以下示範一個讀取文件時不規範的操作：

```
#include<stdio.h>
#include<string.h>
#include <malloc.h>

int m(char *filename) {
    FILE* f;
    int key;
    f = fopen(filename, "r");
    fscanf(f, "%d", &key);
    return key;
}

int main() {
    m("number.txt");
    return 0;
}
```

以上文件在读取时并没有进行 `fclose`，这时将会产生多余的内存，可能一次还好，多次会增加成倍的内存，可以使用循环进行调用，之后在任务管理器中可查看该程序运行时所占的内存大小，代码为：

```
● ● ●  
  
#include<stdio.h>  
#include<string.h>  
#include <malloc.h>  
  
int m(char *filename) {  
    FILE* f;  
    int key;  
    f = fopen(filename, "r");  
    fscanf(f, "%d", &key);  
    return key;  
}  
  
int main() {  
    int i;  
    for(i=0;i<500;i++) {  
        m("number.txt");  
    }  
    return 0;  
}
```

可查看添加循环后的程序与添加循环前的程序做内存占用的对比，就可以发现两者之间添加了循环的代码将会成本增加占用容量。

未被初始化的指针也会有可能造成内存泄漏的情况，因为指针未初始化所指向不可控，如：



```
int *p;  
*p = val;
```

包括错误的释放内存空间：

```
pp=p;  
free(p);  
free(pp);
```

释放后使用，产生悬空指针。在申请了动态内存后，使用指针指向了该内存，使用完毕后我们通过 `free` 函数释放了申请的内存，该内存将会允许其它程序进行申请；但是我们使用过后的动态内存指针依旧指向着该地址，假设其它程序下一秒申请了该区域内的内存地址，并且进行了操作。当我依旧使用已 `free` 释放后的指针进行下一步的操作时，或者所进行了一个计算，那么将会造成的结果天差地别，或者是其它灾难性后果。所以对于这些指针在生存期结束之后也要置为 `null`。查看一个示例，由于 `free` 释放后依旧使用该指针，造成的计算结果天差地别：

```
#include<stdio.h>  
#include<string.h>  
#include <malloc.h>  
int m(char *freep) {  
    int val=freep[0];  
    printf("2*freep=%d\n",val*2);  
    free(freep);  
    val=freep[0];  
    printf("2*freep=%d\n",val*2);  
}
```

```
int main() {  
    int *freep = (int *) malloc(sizeof (int));  
    freep[0]=1;  
    m(freep);  
    return 0;  
}
```

以上代码使用 malloc 申请了一个内存后，传值为 1；在函数中首先使用 val 值接收 freep 的值，将 val 乘 2，之后释放 free，重新赋值给 val，最后使用 val 再次乘 2，此时造成的结果出现了极大的改变，而且最恐怖的是该错误很难发现，隐蔽性很强，但是造成的后顾难以承受。运行结果如下：

```
2*freep=:2  
2*freep=: -224
```

三、new 和 delete

C++ 中使用 new 和 delete 从堆中分配和释放内存，new 和 delete 是运算符，不是函数，两者成对使用(后面说明为什么成对使用)。

new/delete 除了分配内存和释放内存（与 malloc/free），还做更多的事情，所有在 C++ 中不再使用 malloc/free 而使用 new/delete。

3.1 new 和 delete 使用

new 一般使用格式如下：


指针变量名 = new 类型标识符;

指针变量名 = new 类型标识符(初始值);

指针变量名 = new 类型标识符[内存单元个数];

在C++中new的三種用法包括：plain new，nothrow new 和 placement new。

plain new 就是我們最常使用的 new 的方式，在 C++ 中的定義如下：



```
void* operator new(std::size_t) throw(std::bad_alloc);
void operator delete( void *) throw();
```

plain new 在分配失敗的情況下，拋出異常 std::bad_alloc 而不是返回 NULL，因此通過判斷返回值是否為 NULL 是徒勞的。



```
char *getMemory(unsigned long size)
{
    char * p = new char[size];
    return p;
}

void main(void)
{
    try{
        char * p = getMemory(1000000);    // 可能發生異常
        // ...
        delete [] p;
    }
    catch(const std::bad_alloc & ex)
    {
        cout <<< ex.what();
    }
}
```

nothrow new 是不拋出異常的運算符new的形式。nothrow new在失敗時，返回NULL。定義如下：



```
void * operator new(std::size_t, const std::nothrow_t&) throw();
void operator delete(void*) throw();
```



```
void func(unsigned long length)
{
    unsigned char * p = new(nothrow) unsigned char[length];
    // 在使用这种new时要加(nothrow) ，表示不使用异常处理 。


    if (p == NULL) // 不拋異常，一定要檢查
        cout << "alloc failed !";
    // ...
    delete [] p;
}
```

placement new 意即“放置”，这种new允许在一块已经分配成功的内存上重新构造对象或对象数组。**placement new**不用担心内存分配失败，因为它根本不分配内存，它做的唯一一件事情就是调用对象的构造函数。定义如下：



```
void* operator new(size_t, void*);
void operator delete(void*, void*);
```

placement new 的主要用途就是反复使用一块较大的动态分配的内存来构造不同类型的对象或者他们的数组。**placement new**构造起来的对象或其数组，要显示的调用他们的析构函数来销毁，千万不要使用delete。



```
void main()
{
    using namespace std;
```


```

char * p = new(nothrow) char [4];
if (p == NULL)
{
    cout << "allocete failed" << endl;
    exit( -1 );
}
// ...
long * q = new (p) long(1000);
delete []p;    // 只释放 p，不要用q释放。
}

```

p 和 q 仅仅是首址相同，所构建的对象可以类型不同。所“放置”的空间应小于原空间，以防不测。当“放置new”超过了申请的范围，Debug 版下会崩溃，但 Release 能运行而不会出现崩溃！

该运算符的作用是：只要第一次分配成功，不再担心分配失败。



```

void main()
{
    using namespace std;
    char * p = new(nothrow) char [100];
    if (p == NULL)
    {
        cout << "allocete failed" << endl;
        exit(-1);
    }
    long * q1 = new (p) long(100);
    // 使用q1 ...
    int * q2 = new (p) int[100/sizeof(int)];
    // 使用q2 ...
    ADT * q3 = new (p) ADT[100/sizeof(ADT)];
    // 使用q3 然后释放对象 ...
}

```

```
delete [] p;    // 只释放空间，不再析构对象。  
}
```

注意：使用该运算符构造的对象或数组，一定要显式调用析构函数，不可用 `delete` 代替析构，因为 `placement new` 的对象的大小不再与原空间相同。

```
void main()  
{  
    using namespace std;  
    char * p = new(nothrow) char [sizeof(ADT)+2];  
    if (p == NULL)  
    {  
        cout << "allocete failed" &lt;&lt; endl;  
        exit(-1);  
    }  
    // ...  
    ADT * q = new (p) ADT;  
    // ...  
    // delete q; // 错误  
    q->ADT::~~ADT(); // 显式调用析构函数，仅释放对象  
    delete [] p;    // 最后，再用原指针来释放内存  
}
```

`placement new` 的主要用途就是可以反复使用一块已申请成功的内存空间。这样可以避免申请失败的徒劳，又可以避免使用后的释放。

特别要注意的是对于 `placement new` 绝不可以调用的 `delete`，因为该 `new` 只是使用别人替它申请的地方。释放内存是 `nothrow new` 的事，即要使用原来的指针释放内存。`free/delete` 不要重复调用，被系统立即回收后再利用，再一次 `free/delete` 很可能把不是自己的内存释放掉，导致异常甚至崩溃。

上面提到 `new/delete` 比 `malloc/free` 多做了一些事情，`new` 相对于 `malloc` 会额外的做一些初始化工作，`delete` 相对于 `free` 多做一些清理工作。

```
class A
{
public:
    A()
    {
        cout<<"A() 构造函数被调用"<<endl;
    }
    ~A()
    {
        cout<<"~A() 构造函数被调用"<<endl;
    }
}
```

在 `main` 主函数中，加入如下代码：

```
A* pa = new A(); //类 A 的构造函数被调用
delete pa;        //类 A 的析构函数被调用
```

可以看出：使用 `new` 生成一个类对象时系统会调用该类的构造函数，使用 `delete` 删除一个类对象时，系统会调用该类的析构函数。可以调用构造函数/析构函数就意味着 `new` 和 `delete` 具备针对堆所分配的内存进行初始化和释放的能力，而 `malloc` 和 `free` 不具备。


2.2 `delete` 与 `delete[]` 的区别

c++ 中对 new 申请的内存的释放方式有 delete 和 delete[] 两种方式，到底这两者有什么区别呢？

我们通常从教科书上看到这样的说明：

delete 释放 new 分配的单个对象指针指向的内存

delete[] 释放 new 分配的对象数组指针指向的内存 那么，按照教科书的理解，我们看下下面的代码：



```
int *a = new int[10];
delete a;          //方式1
delete[] a;        //方式2
```

针对简单类型 使用 new 分配后的不管是数组还是非数组形式内存空间用两种方式均可 如：



```
int *a = new int[10];
delete a;
delete[] a;
```

此种情况中的释放效果相同，原因在于：分配简单类型内存时，内存大小已经确定，系统可以记忆并且进行管理，在析构时，系统并不会调用析构函数。

它直接通过指针可以获取实际分配的内存空间，哪怕是一个数组内存空间(在分配过程中 系统会记录分配内存的大小等信息，此信息保存在结构体 _CrtMemBlockHeader 中，具体情况可参看 VC 安装目录下 CRTSRDBGDEL.cpp)。

针对类 Class，两种方式体现出具体差异

当你通过下列方式分配一个类对象数组：



```
class A
{
private:
    char *m_cBuffer;
    int m_nLen;

public:
    A(){ m_cBuffer = new char[m_nLen]; }

    ~A() { delete [] m_cBuffer; }
};

A *a = new A[10];
delete a;           //仅释放了a指针指向的全部内存空间 但是只调用了a[0]对象的析构函数 剩下的从a[1]到a[9]这9个用户自行分配的m_cBuffer对应内存空间将不能释放
delete[] a;         //调用使用类对象的析构函数释放用户自己分配内存空间并且 释放了a指针指向的全部内存空间
```

所以总结下就是，如果 **ptr** 代表一个用**new**申请的内存返回的内存空间地址，即所谓的指针，那么：

delete ptr 代表用来释放内存，且只用来释放 **ptr** 指向的内存。**delete[] rg** 用来释放**rg**指向的内存，！！还逐一调用数组中每个对象的 destructor ！！


对于像 **int/char/long/int*/struct** 等等简单数据类型，由于对象没有 **destructor**，所以用 **delete** 和 **delete []**是一样的！但是如果是 **C++** 对象数组就不同了！

关于 **new[]** 和 **delete[]**，其中又分为两种情况：

- (1) 为基本数据类型分配和回收空间；
- (2) 为自定义类型分配和回收空间；

对于 (1)，上面提供的程序已经证明了 **delete[]** 和 **delete** 是等同的。但是对于 (2)，情况就发生了变化。

我们来看下面的例子，通过例子的学习了解 **C++** 中的 **delete** 和 **delete[]** 的使用方法



```
#include <iostream>
using namespace std;

class Babe
{
public:
    Babe()
    {
        cout << \"Create a Babe to talk with me\" << endl;
    }

    ~Babe()
    {
        cout << \"Babe don't Go away,listen to me\" << endl;
    }
};

int main()
{
    Babe* pbabe = new Babe[3];
    delete pbabe;
    pbabe = new Babe[3];
    delete[] pbabe;
    return 0;
}
```

结果是:




```
Create a babe to talk with me
Create a babe to talk with me
Create a babe to talk with me
Babe don't go away,listen to me

Create a babe to talk with me
Create a babe to talk with me
Create a babe to talk with me
Babe don't go away,listen to me
Babe don't go away,listen to me
Babe don't go away,listen to me
```

大家都看到了，只使用 `delete` 的时候只出现一个 `Babe don't go away,listen to me`，而使用 `delete[]` 的时候出现 3 个 `Babe don't go away,listen to me`。不过不管使用 `delete` 还是 `delete[]` 那三个对象的在内存中都被删除，既存储位置都标记为可写，但是使用 `delete` 的时候只调用了 `pbabe[0]` 的析构函数，而使用了 `delete[]` 则调用了 3 个 `Babe` 对象的析构函数。

你一定会问，反正不管怎样都是把存储空间释放了，有什么区别。

答：关键在于调用析构函数上。此程序的类没有使用操作系统的系统资源（比如：`Socket`、`File`、`Thread`等），所以不会造成明显恶果。如果你的类使用了操作系统资源，单纯把类的对象从内存中删除是不妥当的，因为没有调用对象的析构函数会导致系统资源不被释放，这些资源的释放必须依靠这些类的析构函数。所以，在用这些类生成对象数组的时候，用 `delete[]` 来释放它们才是王道。而用 `delete` 来释放也许不会出问题，也许后果很严重，具体要看类的代码了。



最后祝各位保持良好的代码编写规范降低严重错误的产生。



C语言Plus

C/C++开发、最新资讯，打造C/C++程序员最喜爱的交流平台！吐槽、吹水、开车，我们无“恶”不做！学习进阶、分享经验、招聘内推我...
150篇原创内容



公众号

喜欢此内容的人还喜欢

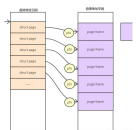
C语言指针详解

C语言Plus



萬字整理，肝翻Linux內存管理所有知識點

Linux內核之旅



C語言| 文件位置標記

C語言入門到精通

