

## C++ 动态内存

了解动态内存存在 C++ 中是如何工作的是成为一名合格的 C++ 程序员必不可少的。C++ 程序中的内存分为两个部分：

- **栈**：在函数内部声明的所有变量都将占用栈内存。
- **堆**：这是程序中未使用的内存，在程序运行时可用于动态分配内存。

很多时候，您无法提前预知需要多少内存来存储某个定义变量中的特定信息，所需内存的大小需要在运行时才能确定。

在 C++ 中，您可以使用特殊的运算符为给定类型的变量在运行时分配堆内的内存，这会返回所分配的空间地址。这种运算符即 **new** 运算符。

如果您不再需要动态分配的内存空间，可以使用 **delete** 运算符，删除之前由 new 运算符分配的内存。

### new 和 delete 运算符

下面是使用 new 运算符来为任意的数据类型动态分配内存的通用语法：

```
new data-type;
```

在这里，**data-type** 可以是包括数组在内的任意内置的数据类型，也可以是包括类或结构在内的用户自定义的任何数据类型。

让我们先来看下内置的数据类型。例如，我们可以定义一个指向 double 类型的指针，然后请求内存，该内存存在执行时被分配。我们可以按照下面的语句使用 **new** 运算符来完成这点：

```
double* pvalue = NULL; // 初始化为 null 的指针
pvalue = new double; // 为变量请求内存
```

如果自由存储区已被用完，可能无法成功分配内存。所以建议检查 new 运算符是否返回 NULL 指针，并采取以下适当的操作：

```
double* pvalue = NULL;
if( !(pvalue = new double ))
{
    cout << "Error: out of memory." << endl;
    exit(1);
}
```

**malloc()** 函数在 C 语言中就出现了，在 C++ 中仍然存在，但建议尽量不要使用 malloc() 函数。new 与 malloc() 函数相比，其主要的优点是，new 不只是分配了内存，它还创建了对象。

在任何时候，当您觉得某个已经动态分配内存的变量不再需要使用时，您可以使用 delete 操作符释放它所占用的内存，如下所示：

```
delete pvalue; // 释放 pvalue 所指向的内存
```

下面的实例中使用了上面的概念，演示了如何使用 new 和 delete 运算符：

## 实例

```
#include <iostream>
using namespace std;
int main ()
{
    double* pvalue = NULL; // 初始化为 null 的指针
    pvalue = new double; // 为变量请求内存
    *pvalue = 29494.99; // 在分配的地址存储值
    cout << "Value of pvalue : " << *pvalue << endl;
    delete pvalue; // 释放内存
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of pvalue : 29495
```

## 数组的动态内存分配

假设我们要为一个字符数组（一个有 20 个字符的字符串）分配内存，我们可以使用上面实例中的语法来为数组动态地分配内存，如下所示：

```
char* pvalue = NULL; // 初始化为 null 的指针
pvalue = new char[20]; // 为变量请求内存
```

要删除我们刚才创建的数组，语句如下：

```
delete [] pvalue; // 删除 pvalue 所指向的数组
```

下面是 new 操作符的通用语法，可以为多维数组分配内存，如下所示：

### 一维数组

```
// 动态分配, 数组长度为 m
int *array=new int [m];
//释放内存
delete [] array;
```

### 二维数组

```
int **array
// 假定数组第一维长度为 m， 第二维长度为 n
// 动态分配空间
array = new int *[m];
for( int i=0; i<m; i++ )
{
    array[i] = new int [n] ;
}
```

```
//释放
for( int i=0; i<m; i++ )
{
    delete [] arrar[i];
}
delete [] array;
```

二维数组实例测试：

### 实例

```
#include <iostream>
using namespace std;
int main()
{
    int **p;
    int i,j; //p[4][8]
    //开始分配4行8列的二维数据
    p = new int *[4];
    for(i=0;i<4;i++){
        p[i]=new int [8];
    }
    for(i=0; i<4; i++){
        for(j=0; j<8; j++){
            p[i][j] = j*i;
        }
    }
    //打印数据
    for(i=0; i<4; i++){
        for(j=0; j<8; j++){
            {
                if(j==0) cout<<endl;
                cout<<p[i][j]<<"\t";
            }
        }
    }
    //开始释放申请的堆
    for(i=0; i<4; i++){
        delete [] p[i];
    }
    delete [] p;
    return 0;
}
```

### 三维数组

```
int ***array;
// 假定数组第一维为 m， 第二维为 n， 第三维为h
// 动态分配空间
array = new int **[m];
for( int i=0; i<m; i++ )
{
    array[i] = new int *[n];
    for( int j=0; j<n; j++ )
    {
```

```
array[i][j] = new int [h];
}
}
//释放
for( int i=0; i<m; i++ )
{
for( int j=0; j<n; j++ )
{
delete array[i][j];
}
delete array[i];
}
delete [] array;
```

三维数组测试实例：

### 实例

```
#include <iostream>
using namespace std;
int main()
{
int i,j,k; // p[2][3][4]
int ***p;
p = new int **[2];
for(i=0; i<2; i++)
{
p[i]=new int *[3];
for(j=0; j<3; j++)
p[i][j]=new int[4];
}
//输出 p[i][j][k] 三维数据
for(i=0; i<2; i++)
{
for(j=0; j<3; j++)
{
for(k=0;k<4;k++)
{
p[i][j][k]=i+j+k;
cout<<p[i][j][k]<<" ";
}
cout<<endl;
}
cout<<endl;
}
// 释放内存
for(i=0; i<2; i++)
{
for(j=0; j<3; j++)
{
delete [] p[i][j];
}
}
for(i=0; i<2; i++)
```

```
{
delete [] p[i];
}
delete [] p;
return 0;
}
```

## 对象的动态内存分配

对象与简单的数据类型没有什么不同。例如，请看下面的代码，我们将使用一个对象数组来理清这一概念：


### 实例

```
#include <iostream>
using namespace std;
class Box
{
public:
Box() {
cout << "调用构造函数! " <<endl;
}
~Box() {
cout << "调用析构函数! " <<endl;
}
};
int main( )
{
Box* myBoxArray = new Box[4];
delete [] myBoxArray; // 删除数组
return 0;
}
```


如果要为一个包含四个 Box 对象的数组分配内存，构造函数将被调用 4 次，同样地，当删除这些对象时，析构函数也将被调用相同的次数（4次）。

当上面的代码被编译和执行时，它会产生下列结果：

```
调用构造函数!
调用构造函数!
调用构造函数!
调用构造函数!
调用析构函数!
调用析构函数!
调用析构函数!
调用析构函数!
```



3 篇笔记

 写笔记