new与malloc的区别

1. 申请的内存所在位置

new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。

那么自由存储区是否能够是堆（问题等价于new是否能在堆上动态分配内存），这取决于operator new 的实现细节。自由存储区不仅可以是堆，还可以是静态存储区，这都看operator new在哪里为对象分配内存。

特别的，new甚至可以不为对象分配内存！布局new的功能可以办到这一点：new (place\_address) type，place\_address为一个指针，代表一块内存的地址。当使用上面这种仅以一个地址调用new操作符时，需要包含头文件new，同时new操作符调用特殊的operator new，也就是下面这个版本：

void \* operatornew (size\_t,void \*) //不允许重定义这个版本的operator new

这个operator new不分配任何的内存，它只是简单地返回指针实参，然后右new表达式负责在place\_address指定的地址进行对象的初始化工作。

1.参数

使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算。而malloc则需要显式地指出所需内存的尺寸。

2.返回类型

new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故new是符合类型安全性的操作符。而malloc内存分配成功则是返回void \* ，需要通过强制类型转换将void\*指针转换成我们需要的类型。

3.分配失败

new内存分配失败时，会抛出bac\_alloc异常，它不会返回NULL。malloc分配内存失败时返回NULL。

在使用C语言时，我们习惯在malloc分配内存后判断分配是否成功：

int \*a = (int \*)malloc ( sizeof (int ));

if(NULL == a)

{

...

}

else

{

...

}

从C语言走入C++阵营的新手可能会把这个习惯带入C++：

int \* a = new int();

if(NULL == a)

{

...

}

else

{

...

}

实际上这样做一点意义也没有，因为new根本不会返回NULL，而且程序能够执行到if语句已经说明内存分配成功了，如果失败早就抛异常了。正确的做法应该是使用异常机制：

try

{

int \*a = new int();

}

catch (bad\_alloc)

{

...

}

5.自定义类型

使用new操作符来分配对象内存时会经历三个步骤：

第一步：调用operator new 函数（对于数组是operator new[]）分配一块足够大的，原始的，未命名的内存空间以便存储特定类型的对象。

第二步：编译器运行相应的构造函数以构造对象，并为其传入初值。

第三步：对象构造完成后，返回一个指向该对象的指针。

使用delete操作符来释放对象内存时会经历两个步骤：

第一步：调用对象的析构函数。

第二步：编译器调用operator delete(或operator delete[])函数释放内存空间。

总之来说，new/delete会调用对象的构造函数/析构函数以完成对象的构造/析构。而malloc则不会。

8.重载

opeartor new /operator delete可以被重载。标准库是定义了operator new函数和operator delete函数的8个重载版本：

//这些版本可能抛出异常

void \* operatornew(size\_t);

void \* operator new[](size\_t);

void \* operatordelete (void \* )noexcept;

void \* operator delete[](void \*0）noexcept;

//这些版本承诺不抛出异常

void \* operatornew(size\_t ,nothrow\_t&) noexcept;

void \* operator new[](size\_t, nothrow\_t& );

void \* operatordelete (void \*,nothrow\_t& )noexcept;

void \* operator delete[](void \*0,nothrow\_t& ）noexcept;

我们可以自定义上面函数版本中的任意一个，前提是自定义版本必须位于全局作用域或者类作用域中。太细节的东西不在这里讲述，总之，我们知道我们有足够的自由去重载operator new /operator delete ,以决定我们的new与delete如何为对象分配内存，如何回收对象。

而malloc/free并不允许重载。

9.重新分配内存

使用malloc分配的内存后，如果在使用过程中发现内存不足，可以使用realloc函数进行内存重新分配实现内存的扩充。realloc先判断当前的指针所指内存是否有足够的连续空间，如果有，原地扩大可分配的内存地址，并且返回原来的地址指针；如果空间不够，先按照新指定的大小分配空间，将原有数据从头到尾拷贝到新分配的内存区域，而后释放原来的内存区域。

new没有这样直观的配套设施来扩充内存。

10. 客户处理内存分配不足

在operator new抛出异常以反映一个未获得满足的需求之前，它会先调用一个用户指定的错误处理函数，这就是new-handler。new\_handler是一个指针类型：

namespace std

{

typedefvoid (\*new\_handler)();

}

指向了一个没有参数没有返回值的函数,即为错误处理函数。为了指定错误处理函数，客户需要调用set\_new\_handler，这是一个声明于的一个标准库函数:

namespace std

{

new\_handler set\_new\_handler(new\_handler p ) throw();

}

set\_new\_handler的参数为new\_handler指针，指向了operator new 无法分配足够内存时该调用的函数。其返回值也是个指针，指向set\_new\_handler被调用前正在执行（但马上就要发生替换）的那个new\_handler函数。

对于malloc，客户并不能够去编程决定内存不足以分配时要干什么事，只能看着malloc返回NULL。

总结

将上面所述的10点差别整理成表格：

| 特征 | new/delete | malloc/free |
| --- | --- | --- |
| 分配内存的位置 | 自由存储区 | 堆 |
| 内存分配失败返回值 | 完整类型指针 | void\* |
| 内存分配失败返回值 | 默认抛出异常 | 返回NULL |
| 分配内存的大小 | 由编译器根据类型计算得出 | 必须显式指定字节数 |
| 处理数组 | 有处理数组的new版本new[] | 需要用户计算数组的大小后进行内存分配 |
| 已分配内存的扩充 | 无法直观地处理 | 使用realloc简单完成 |
| 是否相互调用 | 可以，看具体的operator new/delete实现 | 不可调用new |
| 分配内存时内存不足 | 客户能够指定处理函数或重新制定分配器 | 无法通过用户代码进行处理 |
| 函数重载 | 允许 | 不允许 |
| 构造函数与析构函数 | 调用 | 不调用 |