动态存储管理机制

重载 new 和 delete 操作符，创建自己的内存管理机制。

动态存储结构的最大缺陷在于所申请的内存在使用完毕后经常不会合理释放，以及容易造成指针访问错误。这里面有程序员的原因（很多程序员对指针使用方法一知半解，该释放时不知道释放，不该释放时却盲目释放），也有系统的原因（像广义表这样的数据结构很难确定内存释放的时机）。此时，设计一类灵活的动态内存策略就是非常有必要的了。

作为一般原则，可以在程序开始时分配一片足够大的存储空间，并使用 pool 指针指向它。其后所有内存分配都使用重载的 new 操作符在 pool 指向的存储空间中进行，即将 pool 指向的存储空间作为后续存储管理的存储池。

所谓存储池（storage pool），是指一片连续的存储区，其中包括多个连续的、不相交的、更小的、可进一步分配的存储区，它可用来管理动态内存分配，即系统一次分配整个存储池的存储空间，而将进一步的存储分配保留在存储池中进行。存储池保存在进程地址空间中的某个位置，通过使用特殊方法创生数据对象，程序就有可能在某个固定的存储池中为该对象分配大小合适的存储空间，当释放该存储空间时再将其返回给存储池。

这种创生数据对象的特殊方法就是 new 操作符的第三种使用形式——定位创生表达式（placement new expression）。该形式的 operator new() 函数原型为：

```
void* operator new( std::size_t size, void* p );
```

调用方法为：

```
char* base_addr = new char[4096];  // 分配 4096 个字节的存储空间
int* p = new(base_addr) int(16);  // 在该存储空间中分配整数匿名数据对象
```

带有定位参数的 new 操作符不会在堆中分配存储空间，它只是使用后面的初始体（如果存在）初始化已分配好的堆存储空间（由参数 base_addr 指定），函数的返回值是转型后的基地址值 base_addr。

定位创生表达式在 base_addr 处只能“分配”单个匿名数据对象，两次分配只以最后一次为准。例如对于下述代码，p、q 均指向 base_addr，因而只有最后一个匿名数据对象有效：

```
using namespace std;
char* base_addr = new char[4096];
int* p = new(base_addr) int(16);
cout << hex << showbase << setw(8) << int(p) << " " << dec << *p << endl;
int* q = new(base_addr) int(32);
cout << hex << showbase << setw(8) << int(q) << " " << dec << *q << endl;
delete[] base_addr;
```

使用定位创生表达式创生的匿名数据对象不能使用 delete 操作符销毁。事实上，也没有这样的 delete 操作符。

定位创生表达式的最大意义在于可用来设计一种应用程序独有的动态内存管理策略，以

部分解决标准内存管理策略下频繁分配小数据对象时的低效率问题（当然只有在应用程序独有的动态内存管理策略好于标准内存管理策略时才有效）。

分析存储池可以发现，它至少应支持下述两个操作：

1. Acquire()　用于从存储池中获取内存并返回指向该内存的指针；此过程称为获取（acquirement）或保留（reservation），并称该片存储空间已保留（reserved）。

2. Reclaim()　将某个指针指向的内存返回给存储池，此过程称为回收（reclaim）或释放（liberation）；返回给存储池的存储空间仍维持空闲（free）。

注意，因为定位创生表达式只能在指定位置重新解释已分配的内存，所以简单使用定位创生表达式不能完成 Acquire() 任务。为完成 Acquire() 与 Reclaim() 操作，必须重载 operator new() 与 operator delete() 操作符函数。

技术上，operator new() 与 operator delete() 的重载相当特殊——既可以重载为全局操作符函数，也可以重载为类的成员函数，部分编译器（Borland C++）甚至允许将它们重载到某个独立的名空间。另外，operator new() 与 operator delete() 既可以重载为多参数版本，也可以重载为单参数版本，后者将覆盖（override）全局 operator new() 与 operator delete() 操作符函数。

重载的 operator new() 与 operator delete() 应与 C++ 标准库中的形式相同或类似：operator new() 必须接受一个 std::size_t 型式的参数以表示分配空间的大小，必须返回一个 void* 型式的指针；operator delete() 必须接受一个 void* 型式的指针。例如，假设存在存储池 pool，则可以这样重载 operator new() 与 operator delete() 操作符函数：

```
void* operator new( std::size_t size, STORAGEPOOL* pool )
{
  return Acquire( pool, size );
}

void operator delete( void* ptr )
{
  Reclaim( ptr );
}
```

一旦实现了上述函数与 Acquire()、Reclaim() 函数，则可以在程序中使用下述代码在存储池中进行内存分配：

```
int* p = new(pool) int;  // 假设 pool 指向的存储池数据对象已分配
delete p;  // 使用完毕后通过 delete 操作符调用重载的 operator delete() 回收存储空间
```

其中，存储池 pool 表示进一步内存分配的源，通过重载的 operator new() 分配的所有内存都从存储池 pool 中获得，通过重载的 operator delete() 释放的所有内存都返回给存储池 pool。当调用重载的 operator new() 时，程序显式给出了 pool 参数，该参数指示了从哪个存储池获取 p 所指向的匿名数据对象的存储空间，然而在调用重载的 operator delete() 时却没有指定将该片存储空间返回给哪个存储池。当程序中存在两个存储池时，上述程序代码就会导致问题。

仔细研究存储管理策略，兼顾存储管理的效率与性能。典型的存储管理策略有伙伴系统（buddy system）。

特别说明：本题相对较难，程序只要能够完成最简单的功能就是巨大的胜利！