动态内存

动态内存与智能指针

1. 静态内存用来保存局部static对象、类static数据成员以及定义在任何函数体之外的变量。栈内存用来保存定义在函数内的非static对象。分配在静态内存或栈内存中的对象由编译器自动创建和销毁。对于栈对象，仅在其定义的程序块运行时才存在。static对象在使用之前分配，在程序结束时销毁。
2. 每个程序拥有一个内存池，称为自由空间或者堆。程序用堆来存储动态分配的对象————即程序运行时分配的对象。
3. 新的标准库提供了两种智能指针类型来管理动态对象。智能指针的行为类似常规指针，区别在于**它负责自动释放所指向的对象**。其中shared_ptr允许多个指针指向同一个对象；unique_ptr只允许一个指针指向一个对象。还有一个名为weak_ptr的伴随类，是一种弱引用，指向shared_ptr所管理的对象。这三种类都定义在memory头文件中。
4. 默认初始化的智能指针中保存一个空指针；使用智能指针的方式与普通指针一页，解引用一个智能指针就能返回它指向的对象。
5. shared_ptr和unique_ptr都支持的操作：
   - p.get()：返回p中保存的指针。若智能指针释放了其对象，则返回的指针所指向的对象也消失了。
   - shared_ptr<类型> sp / unique<类型> up：创建智能指针
   - swap(p,q)：交换两个智能指针
6. shared_prt特有的操作：
   - make_shared<类型>(args)：返回一个shared_ptr，指向一个动态分配的对象并使用args进行初始化，args初始化的是<>中指定的类型；（这是一个函数）
   - shared_ptr<类型> p(q)：p是q的拷贝，此操作会递增q中的计数器，q中的指针必须能转换为定义的类型的指针；
   - p=q：要求p和q都是shared_ptr且所保存的指针必须能够互相转换。此操作会递减p的引用计数，递增q的引用计数；若p的引用计数变为0则将其管理的原内存释放。
   - p.use_count()：返回与p共享对象的智能指针的数量
   - p.unique()：若p.use_count()==1则返回true
7. 最安全的分配和使用动态内存的方法是调用make_shared函数，此函数在动态内存中分配一个对象并初始化它，返回指向此对象的shared_ptr。类似顺序容器中的emplace函数，make_shared函数用其参数来构造给定类型的对象。

```
//定义一个指向一个值为“555555”的string的智能指针sp
shared_ptr<string> sp=make_shared<string>(6,'5');
auto p=make_shared<string>(6,'5');  //常用auto更方便
```

8. 当进行拷贝或者赋值操作时，每个shared_ptr都会记录有多少个其他shared_ptr指向相同的对象。可以认为每个shared_ptr都关联一个计数器，通常称为引用计数。无论何时我们拷贝一个shared_ptr（作为函数返回值时也是，需要拷贝），计数器都会递增。**当我们给shared_ptr赋予一个新值或者是shared_ptr被销毁（如一个局部的shared_ptr离开其作用域）时，计数器就会递减**。

```
auto r=make_shared<int>(5);
r=p;     //此操作会递增p指向的对象的引用计数，递减r原来指向的对象的引用计数，若r原来指向的对象
```

9. 当指向一个对象的最后一个shared_ptr被销毁死后，shared_ptr类会通过**析构函数**自动销毁此对象。shared_ptr的析构函数会递减它所指向的对象的引用计数，当引用计数变为0时shared_ptr的析构函数就会销毁对象并释放它所占用的内存。对于一块内存，shared_ptr保证只要有任何shared_ptr对象引用它，它就不会被释放。
10. 使用动态内存的三种原因：
    - 程序不知道自己需要使用多少对象————容器类
    - 程序不知道所需要的对象的准确类型
    - **程序需要在多个对象间共享数据————shared_ptr**

直接管理内存

1. 默认情况下，动态分配的对象是默认初始化的，这意味着内置类型或组合类型的对象的值是**未定义的**，而类类型对象将用默认构造函数进行初始化。

```
int *p1=new int;         //默认初始化，*p1的值未定义
int *p2=new int();       //值初始化为0，*p2=0

auto p=new auto(obj);    //p执行一个与obj类型相同的对象，auto中的初始化器只能有一个
```

2. 默认情况下如果new不能分配所要求的内存空间（内存空间被耗尽），则会抛出一个类型为bad_alloc的异常。我们这样通过**定位new**来防止抛出异常：

```
int *p=new (nothrow) int();
```

   定位new表达式允许我们向new传递额外的参数。这里我们传递的是标准库定义的名为nothrow的对象，表示不能抛出异常。若这种形式的new不能分配所需的内存则它会返回一个空指针。（bad_alloc和nothrow都定义在头文件new中）
3. delete表达式执行两个动作：销毁给定的指针指向的对象；释放对应的内存。传递给delete的指针必须指向动态分配的内存或者是空指针。

shared_ptr和new的结合使用

1. 接受指针参数的智能指针构造函数是explicit的，因此不能将一个内置指针隐式转换为一个智能指针，必须使用直接初始化方式：

```
shared_ptr<int> p1=new int(5);  //错误，int*不能隐式转换为智能指针
shared_ptr<int> p2(new int(5)); //正确，采用了直接初始化的方式
```

2. 默认情况下一个用来初始化智能指针的普通指针必须指向动态内存，因为**智能指针默认使用delete来释放它所关联的对象**。若要将智能指针绑定到一个指向其他类型的资源的指针上，就必须提供自己的操作来代替delete。
3. 定义和改变shared_ptr的方法：
   - shared_ptr<T> p(q)：p管理内置指针q所指向的对象，q必须指向new分配的内存且能够转换为T*
   - shared_ptr<T> p(u)：p从unique_ptr u那里接管了对象的所有权并将u置空
   - shared_ptr<T> p(q,d)：p接管了内置指针q所指向的对象的所有权，并使用对象d来代替delete
   - p.reset(可选参数 q,d)：若p是唯一指向其对象的shared_ptr则reset会释放此对象（即引用计数为1，则递减后等于0，需要释放）。若传递了q作为参数则会令p指向q，否则会将p置空。若还传递了参数d，则会调用d而不是delete来释放。
4. 不要用内置指针显示构造shared_ptr，这样做很可能导致错误。因为显示构造不等于拷贝

```
void func(shared_ptr<int> p){
    cout<<p.use_count()<<endl;
}
int *x=new int(5);
func(shared_ptr<int>(x));   //这里显式构造了智能指针 实际上func里的p.use_count()输出的是1
cout<<*x;   //会输出一个未定义的值
```

5. 智能指针的get函数返回一个内置指针，指向智能指针管理的对象，但我们不能delete这个返回的指针。此外，虽然编译器不会报错，但将另一个智能指针绑定到get返回的指针上也是错误的，因为这样的话会有独立的shared_ptr指向同一个块内存。
6. 函数由于发生异常而退出时会销毁局部对象，函数中的智能指针会在销毁时检查引用计数，若递减后引用计数为0则释放指向的内存。但是函数中直接管理的内存是不会自动释放的，所以会发生内存泄露。

unique_ptr

1. 某个时刻只能有一个unique_ptr指向一个给定对象，因此unique_ptr不支持普通的拷贝或者赋值操作。定义一个unique_ptr时需要将其绑定到一个new返回的指针。
2. unique_prt的相关操作：
   - unique_ptr<T,可选参数 D> p：默认p会使用delete释放它的指针，当声明了D时会用D来释放
   - p=nullptr：释放p指向的对象并将p置空
   - p.release()：p放弃对指针的控制权并返回指针，将p置空，但并没有释放内存

- p.reset(可选参数 q)：释放p指向的对象，若提供了参数q则令p指向q，否则将p置空

3. 不能拷贝unique_ptr的规则有一个例外：可以拷贝或赋值一个将要被销毁的unique_ptr。比如从函数返回一个unique_ptr：

```
unique_ptr<int> clone1(int p){
    return unique_ptr<int>(new int(p));
}
unique_ptr<int> clone2(int p){  //返回局部对象的拷贝
    unique_ptr<int> ptr(new int(p));
    return ptr;
}
```

4. 较早的标准库版本中包含了一个名为auto_ptr的类，其具有unique_ptr的部分特性，但不能在容器中保存auto_ptr，也不能从函数中返回auto_ptr。所以还是用unique_ptr比较好。

weak_ptr

1. weak_ptr是一种不控制所指向对象生存期的智能指针，指向由一个shared_ptr管理的对象。将一个weak_ptr绑定到一个shared_ptr不会改变shared_ptr的引用计数。一旦最后一个指向对象的shared_ptr被销毁则对象被释放，无论是否有weak_ptr指向该对象。
2. 创建一个weak_ptr时要用一个shared_ptr来初始化。weak_ptr的操作如下：
   - w.reset()：将w置空
   - w.use_count()：返回与w共享对象的shared_ptr的数量
   - w.expired()：若w.use_count()为0则返回true，表示“到期”
   - w.lock()：若expired为true则返回一个空的shared_ptr，否则返回一个指向w的对象的shared_ptr。**由于对象可能不存在，所以我们要使用lock来返回对象**。
3. 为什么要使用weak_ptr？首先要明白weak_ptr是用来辅助shared_ptr的，那shared_ptr有什么问题呢？看下面的代码：

```
class ClassB;

class ClassA{
    public:
        ClassA() { cout << "ClassA Constructor..." << endl; }
        ~ClassA() { cout << "ClassA Destructor..." << endl; }
        shared_ptr<ClassB> pb;  // 在A中引用B
};

class ClassB{
    public:
        ClassB() { cout << "ClassB Constructor..." << endl; }
        ~ClassB() { cout << "ClassB Destructor..." << endl; }
        shared_ptr<ClassA> pa;  // 在B中引用A
};

int main() {
    shared_ptr<ClassA> spa = make_shared<ClassA>();
    shared_ptr<ClassB> spb = make_shared<ClassB>();
    spa->pb = spb;
    spb->pa = spa;
}

/* 运行结果如下：
ClassA Constructor...
ClassB Constructor...
*/
```

两个类互相引用，这称为**循环引用**。上面main函数执行完成后，指针所指向的对象没有被释放。
可见shared_ptr指针无法处理这种问题。
因此引入weak_ptr来解决这个问题，因为weak_ptr不影响引用计数

```
        class ClassB;

class ClassA{
    public:
        ClassA() { cout << "ClassA Constructor..." << endl; }
        ~ClassA() { cout << "ClassA Destructor..." << endl; }
        shared_ptr<ClassB> pb;  // 在A中引用B
};

class ClassB{
    public:
        ClassB() { cout << "ClassB Constructor..." << endl; }
        ~ClassB() { cout << "ClassB Destructor..." << endl; }
        shared_ptr<ClassA> pa;  // 在B中引用A
};

int main() {
    shared_ptr<ClassA> spa = make_shared<ClassA>();
    shared_ptr<ClassB> spb = make_shared<ClassB>();
    spa->pb = spb;
    spb->pa = spa;
}

/* 运行结果如下:
ClassA Constructor...
ClassB Constructor...
ClassB Destructor...
ClassA Destructor...
*/
```

动态数组

1. 定义动态数组时可以用空括号对数组中的元素进行值初始化，但不能在括号中给出初始化器（可以用花括号给出），这意味着不能用auto定义动态数组。
2. 当用new分配一个大小为0的数组时，new会返回一个合法的非空指针，保证与new返回的其他任何指针都不相同。对于零长度的数组来说这个指针就像尾后指针一样，可以用这个指针减去自身得到0，但这个指针不能解引用，因为它不指向任何元素。
3. 释放动态数组：delete []p ，数组中的元素按逆序销毁

智能指针与动态数组

1. 可以使用unique_ptr管理动态数组：unique_ptr<int[]> up(new int[10])，此时unique——ptr是指向一个数组，所以不能用点和箭头成员运算符，但可以用下标运算来访问数组中的元素。
2. shared_ptr不直接支持管理动态数组，若要管理则需要自己定义删除器：

```
shared_ptr<int> sp(new int[10],[](int *p){ delete []p;});
```

因为shared_ptr默认是使用delete销毁对象的。
而且shared_ptr未定义下标运算符，所以为了访问数组中的元素必须用get获取一个内置指针然后用它来访问数组元素

allocator类

1. new将内存分配与对象构造组合在一起，delete将对象析构与内存释放组合在一起，但有时候我们想要的是先分配内存，等真正需要的时候再真正构造对象，因此希望**内存分配与对象构造分离**，以避免造成不必要的浪费。（比如对象的多次赋值）
2. 标准库的allocator定义在头文件memory中，支持内存分配与对象构造分离，它类似vector，是一个模板。其方法如下：
   - allocator<T> a：定义了一个名为a的allocator对象，可以为类型为T的对象分配内存。
   - a.allocate(n)：分配一段原始的、未构造的内存，保存n个类型为T的对象
   - a.destroy(p)：p为T*类型的指针，对p所指向的对象执行析构函数
   - a.deallocate(p,n)：释放从指向类型为T的指针p开始的内存，这块内存保存了n个类型为T的对象。p必须是一个先前由allocate返回的指针，且n必须是p创建时所要求的大小。调用deallocate之前必须对每个在这块内存中创建的对象调用destroy
   - a.construct(p,args)：p是一个类型为T*的指针，指向一块元素内存；args被传递给类型为T的构造函数，用来在p指向的内存中构造一个对象。