

动态内存

动态分配的对象, 只有被显示释放时, 才会被销毁。否则会造成内存泄漏或者被操作系统负责收回。

- 栈内存保存定义在函数内的非static对象
- 静态内存保存局部static对象, 类static数据成员, 以及定义在任何函数外的变量
- 堆保存动态分配的对象
- 分配在栈内存的 (如局部变量) 或静态内存的由编译器自动创建和销毁。
 - 栈内对象, 生命周期最短, 仅在所在程序库运行时才存在
- static对象在使用之前分配, 在程序结束时销毁。

智能指针

```
#include <memory>
```

由于动态分配内存, 其管理较为不便。因此C++引入了**智能指针**来管理动态对象。

1. 智能指针是一个模板类, 通过封装方法, 使其使用起来和指针类似, 但是其具有构造函数和析构函数

智能指针有三类shared_ptr、unique_ptr、weak_ptr智能指针, 区别就在于资源所有权的区别。

- shared_ptr允许多个指针指向同一个对象, 即共享同一资源。
- unique_ptr独占所指的对象。
- weak_ptr是一种弱的引用, 指向shared_ptr管理的对象。

shared_ptr类

shared_ptr将new、delete等操作符封装成一个模板类, 因为可以多个shared_ptr对象共享资源, 因此可以拷贝和复制。一般来说, shared_ptr指向的对象, 应当是调用一个名为make_shared的标准库函数。

```
shared_ptr<int> p3=make_shared<int> (42);    // 指向一个new出来的值为42的int的shared_ptr对象
shared_ptr<string> p4=make_shared<string> (10, '9');
shared_ptr<int> p5=make_shared<int> ();
```

因为shared_ptr不是一般的指针, 因此下面的调用是非法的。

```
shared_ptr<int> p=new int(4);// error

shared_ptr<int> p=static_cast<shared_ptr<int>> (new int (4));    // ok, 显示转换
```

每个shared_ptr都有一个引用计数。引用计数在拷贝或赋值时会发生变化, 如

```
auto r=make_shared<int> (42);    // r指向的int只有一个引用者, 即该动态分配的对象的引用计数为1
r=q;    // 给r赋值, 令r指向另一个地址
        // 递增q所指向对象的引用计数
        // 递减r原来指向的对象的引用计数
        // 如果r原来指向的对象的引用计数为0, 自动释放该对象资源
```

使用什么数据结构来记录共享对象个数, 取决于标准库的实现。

1. shared_ptr类通过析构函数完成销毁工作。
2. shared_ptr会自动释放其关联的内存

释放过程

一个shared_ptr对象, 在其生命周期结束时, 调用其析构函数, 然后递减其引用计数, 如果递减后为0, 则自动销毁指向对象

```
shared_ptr<Foo> foo(T arg){
    //
    return make_shared<Foo> (arg);
}
void use_foo(T arg){
    shared_ptr<Foo> p=foo(arg);
    // 使用p
} // 此时p离开了作用域, 生命周期结束, 调用其析构函数, 递减其引用计数, 如果递减后为0, 则销毁指向对象
```

使用动态生存期的资源的类, 程序使用动态内存有以下三个原因:

1. 程序不知道需要使用多少对象, 如vector, string等容器类
2. 程序不知道所需对象的准确类型
3. 程序需要多个对象间共享数据 (资源)

现在先关注第3个原因, 如vector, 不过的对象, 其资源是独立的。

```
vector<string > v1;
{// 新的作用域
    vector<string> v2={"a", "an"};
    v1=v2;    // v2拷贝赋值到v1, v1和v2的资源是相互独立的
} // v2的生命周期结束, 其中的资源被销毁
// v1的生命周期还存在, 保留者被拷贝的元素。
```

因此可以通过shared_ptr设计一个类, 共享同一资源。

new动态分配

`new`和`delete`是两个可以直接管理内存的运算符。

1. 使用`new`, 默认情况, 分配的对象是默认初始化的 (和类一样), 这就意味着内置类型或组合类型的值将是未定义的, 而类类型对象将用默认构造函数进行初始化。

```
string *ps=new string; // 使用string的默认构造函数, 初始化为空string
int *pi=new int;      // pi指向的一个未初始化的int
```

2. 可以使用值初始化为动态分配的对象赋值,

```
string *ps1=new string; //默认初始化为空string
string *ps= new string(); // 值初始化为空string
int *pi1=new int;      // 默认初始化, *pi1的值未定义
int *pi2=new int();    // 值初始化, *pi2的值为0
```

delete释放动态内存

1. `delete`一个局部变量, 而不是指向动态内存的指针, 其结果是未定义的

```
int i,*p=&i;
delete p; // 未定义的
```

2. `delete`一个已经释放过的资源是未定义的, 如一个指向动态内存的指针被`delete`多次。
3. `delete`一个空指针是合法的。
4. `const`对象不可改变, 但本身可以被销毁。

```
const int *pci=new const int(1024);
delete pci; // ok, 释放一个const对象
```

5. `delete`一个指针后, 但是有可能只是释放掉该指针指向对象的资源, 但是该指针保存的地址依然存在, 这就可能造成空悬指针 (**dangling pointer**), 即常说的野指针。这种行为也是极其危险的, 如果`delete`一个指针后, 再使用这个指针的行为是十分不安全的。因此因在`delete`之后, 设置为`nullptr`。
6. `new`和`delete`无法很好的处理共享资源的多个对象的释放问题。

shared_ptr和new结合使用

之前说过, `shared_ptr`不是常规指针, 是一个模板类, 因此不可以直接将`new`的对象赋值给一个`shared_ptr`对象。但由于`shared_ptr`类包含接受常规指针的构造函数, 并且该构造函数是`explicit`的。

```
shared_ptr<int> p1=new int(1024); // error, 必须使用直接初始化的形式, 因为该构造函数是explicit
的
shared_ptr<int> p2(new int(1024)); // ok, 使用直接初始化形式
```

如前所述, 不可以进行常规指针到智能指针的隐式转换, 但是可以通过强制转换, 显示绑定到一个常规指针。

```
shared_ptr<int> clone(int p){
    // ok, 显示地用int *创建shared_ptr<int> 对象
    return shared_ptr<int>(new(int (p));
}

shared_ptr<int> p1= new int (1024);    // error
shared_ptr<int> p1=static_cast<shared_ptr<int>>(new int (1024));    // 强制转换, 显示创建...
```

这里的强制转换和普通的有些区别, 该static_cast是调用的shared_ptr类的构造函数, 显示构建一个对象。

unique_ptr指针

我们不可以使用make_shared来返回一个unique_ptr, 必须使用直接初始化形式定义一个unique_ptr对象。即

```
unique_ptr<double> p1; // 可以指向一个double的unique_ptr, 默认指向nullptr
unique_ptr<int> p2(new int(42)); // p2指向一个值为42的int
```

因为, unique_ptr独占资源所有权, 因此

- 不支持拷贝操作
- 不支持赋值操作

虽然不支持拷贝和赋值, 但是可以通过release或reset将指针所有权从一个 (非const) unique_ptr转移给另一个unique_ptr:

```
// 所有权从p1转移到p2
unique_ptr<string > p2(p1.release()); // release将p1置为空
unique_ptr<string > p3(new string("Trex"));
// 将所有权从p3转移到p2
p2.reset(p3.release()); // release将p3置为空, reset将p2原来拥有的资源释放, 并重新指向p3指向的资源
```

unique_ptr操作

```
unique_ptr<T> u1; // 空unique_ptr, 可以指向类型为T的对象, u1会使用delete释放资源
unique_ptr<T, D> u2; // u2是使用类型为D的可调用对象来释放它的指针, D默认为delete
unique_ptr<T, D> u(d); // 空unique_ptr, 指向类型为T的对象, 用类型为D的对象d代替delete

// 由于unique_ptr是独占资源的, 因此将其置为nullptr, 就可释放其指向的资源
u=nullptr; // 释放u指向的资源, 将u置为nullptr

u.release(); // u放弃对指针的控制权, 返回指针, 将u置为nullptr

u.reset(); // 释放u指向的对象, u置为nullptr
```

```
u.reset(q);           // 释放u指向的对象, 并将其指向q
u.reset(nullptr);
```

! 这里应当注意release()操作, 如

```
p2.release();         // ok, p2不会释放其原本指向的资源, p2只是放弃了该资源的控制权, 并置为nullptr
                      // 返回指针, 但是我们没有保存该指针
auto p=p2.release();  // ok, 但是我们应当自己delete p;
```

返回unique_ptr和传递unique_ptr

我们不能拷贝和赋值unique_ptr, 这是因为其独占所有权。或者利用reset和release进行所有权的转移。但是考虑到如果一个unique_ptr的生命周期将要终止, 即将要被销毁的unique_ptr对象, 此时就可以"拷贝和赋值"操作。如,

```
unique_ptr<int> clone(int p){
    // ok
    return unique_ptr<int> (new int(p));
} // 临时创建的unique_ptr对象将要被销毁, 因此可以被拷贝
```

或者, 返回一个局部unique_ptr对象, 当然, 本质上和上面的例子是一样的。

```
unique_ptr<int> clone(int p){
    unique_ptr<int> ret(new int(p));
    // ...
    return ret;
}
```

这种拷贝是编译器执行的一种特殊的拷贝, 即移动拷贝, move语义的拷贝。使用move即可拷贝或赋值操作, 因为move也是一种资源控制权转移的操作, 不影响unique_ptr独占资源的概念。如

```
unique_ptr<int> p;
unique_ptr<int> p2(new int(1024));
p=p2;    // error
p=std::move(p2);    // ok, 移动赋值
// unique_ptr<int> p3(std::move(p2));    // ok, 移动拷贝
```

向unique_ptr传递删除器

如前所述, unique_ptr默认的deleter (删除器) 是delete, 但是我们可以传递其一个自定的deleter, 行为和shared_ptr有些许差异:

- `shared_ptr`不是将`deleter`直接保存为一个成员, 因为`deleter`到运行时才会知道, 而且在`shared_ptr`的生命周期内, 我们可以随时改变其`deleter`。通常, 类的成员类型在运行时是不能改变的。因此, `shared_ptr`不能直接保存为一个成员, 而是间接保存的, 如

```
del? del(p):delete p;    // del(p)需要在运行时跳转到del的地址
```

运行时可以随时改变`del`指向的`deleter`。

- `unique_ptr`的`deleter`是类的一部分, 即`unique_ptr`有两个模板参数, 一个表示管理的指针, 一个表示`deleter`的类型。由于`deleter`是`unique_ptr`的一部分, 因此`deleter`在编译期就明确了。

```
// del编译时绑定, 直接调用实例化的deleter
// del默认为delete
del(p);
```

`unique_ptr`对象创建, 指定保存的指针和`deleter`

```
// p指向一个类型为objT的对象, 并使用一个类型为delT的对象释放objT对象
// delT类型的对象被绑定到fcn上
unique_ptr<objT,delT> p(new objT,fcn);
```

weak_ptr

C++ Primer: `weak_ptr`是一种不控制所指向对象生存期的智能指针, 它指向一个由`shared_ptr`管理的对象。

`weak_ptr`是一个弱引用指针。为什么要引入弱引用指针呢?

弱引用指针就是没有所有权的指针, 有时候我们只是想找个指向这块内存的指针, 但我不能把这块内存的生命周期与这个指针相关联。即这种情况下, 弱引用指针仅代表我可以访问该内存, 但是该内存的释放与我无关。

当然, 原生指针也是一种弱引用指针, 只是原生指针的含义不够明确, 万一该指针被`delete`, 则就会不符合弱引用的原则, 从而出现安全问题。下面看一下, 弱引用指针如何做到这种安全。

弱引用的作用之一是可以消除引用环问题。

`weak_ptr`只能由`shared_ptr`或者其他的`weak_ptr`构造, `weak_ptr`和`shared_ptr`共享一个引用计数, 在引用计数对象上增加一个`weak_count`, 但不增加`ref_count`。引用计数减为0则会销毁管理的资源, `weak_ptr`通过`ref_count`是否为0来判断所指向的资源是否可用。当`ref_count`和`weak_count`都为0时引用计数对象会销毁其自身。

```
weak_ptr<T> w;           // 空weak_ptr, 可以指向类型为T的对象
weak_ptr<T> w(sp);       // 指向shared_ptr sp指向的对象, T必须能转换为sp指向的类型

w=p;                    // p可以是一个shared_ptr, 也可以是一个weak_ptr。赋值后w与p共享对象
```

```

w.reset();           // 将w置为空

w.use_count();       // 与w共享对象的shared_ptr的数量, 其实就是w指向的shared_ptr对象内的引用
计数
                    // weak_ptr对象不会影响引用计数, 只会改变weak_count

w.expired();         // 若w.use_count()为0, 返回true, 否则false。即w指向的内存对象被释放了已经

w.lock();            // 如果w.expired()为true, 返回一个空shared_ptr, 否则返回一个指向w的对象
的shared_ptr

```

weak_ptr必须和shared_ptr搭配使用

创建一个weak_ptr, 需要用shared_ptr来初始化它, 即

```

auto p=make_shared<int> (42);
weak_ptr<int> wp(p);    // p的引用计数未改变, weak_count自增

```

由于weak_ptr指向的内存资源可能不存在, 因此我们不能直接访问, 必须调用lock。如

```

if(shared_ptr<int> np=wp.lock()){ // 如果np不为空则条件成立
    // if中np和p共享对象
}

```

环形引用

```

class TB;
class TA{
public:TA(){...};
void refB(std::shared_ptr<TB> ptr)this->ptr=ptr;

~TA(){...}
private:
std::shared_ptr<TB> ptr;
};
class TB{
public:TB(){...};
void refA(std::shared_ptr<TA> ptr)this->ptr=ptr;
~TB(){...};
private:
std::shared_ptr<TA> ptr;
};
int main(){
std::shared_ptr<TA> pa=std::make_shared<TA>();
std::shared_ptr<TB> pb=std::make_shared<TB>();//new一个TB类型的对象, 调用TB();

```

```

    pa->refB(pb);
    pb->refA(pa);
    return 0;
} //    pa, pb结束生命周期, 调用shared_ptr的析构函数, pa.use_count()减1, pb.use_count()减1,
//    自减之后, pb和pb指向对象的引用计数仍然是1, 因此无法销毁该生成对象, 因此没有调用该对象的析构函数

```

输出结果:

```

TA()
TB()

```

即以上example说明只有TA()和TB()被调用。而pb和pa在程序结束后, 销毁, 同时pb和pa引用对象的引用计数减一。但是该生成(new)对象的引用计数此时还是为1, 因此并没有销毁该生成对象, 也就是没有调用该对象的析构函数。因此造成了内存泄漏。

new和数组

new和delete在动态分配一个对象数组时, 写法有些不同, 即

```

int *pia=new int[get_size()];    // pia指向生成的数组的第一个int

```

动态一个数组会得到一个元素类型的指针

new T[]分配的内存是连续的, 称之为“动态数组”, 但是这和数组是有区别的。类似于malloc分配的资源。这意味着, 我们应当这样遍历它,

```

int *p=new int(N);
for(int i=0;i<N;++i){
    // p[i]
}
// 或者
for(int *q=p;q!=p+N;++q){}
// 而不是使用begin, 范围for语句

```

可以分配大小为0的数组, new会返回一个合法的非空指针。对于零长度的动态分配的数组, 该指针类似于尾后指针, 我们可以使用该指针进行比较操作, 但是不可以解引用该指针。

释放动态数组

为了区别普通的delete, 释放动态数组为,

```

delete p;
delete [] pa;    // 表明将要释放的是一个动态数组

```


delete和delete[]的区别

delete只会调用一次析构函数, 而**delete[]**会调用每个成员的析构函数。