课程目标

- C++基本类型转换
- 静态转换，动态转换，常量转换，重新解释转换
- 智能指针的介绍和使用
- 智能指针的原理
- 智能指针的实现(扩展)

课程实验

课堂引入

智能指针是C++编程中的重要环节，它带来了许多优势，包括内存管理的改进、避免内存泄漏、减少野指针问题等。智能指针提供了一种内存自动管理的机制，不需要手动调用`new`和`delete`分配和释放内存。这有助于减少内存泄漏和重复释放内存的风险。智能指针可以在底层对象被删除后自动将指针置为`nullptr`，从而避免了悬空指针(野指针)问题。使用智能指针可以有效地减少由于使用已被删除的对象而导致的运行时错误。

通过智能指针，不需要手动管理资源，这包括内存释放、文件关闭等，使得代码更加清晰、简洁，并减少了出错的机会。智能指针可以提供更安全的代码，减少了手动内存管理所带来的错误。程序员不再需要关心何时释放内存，从而降低了错误的发生概率。在现代`C++`中，使用智能指针被视为一种良好的实践。许多`C++`编程规范和库都鼓励使用智能指针，因此学习它是跟随最佳实践的一部分。

授课进程

一 类型转换

类型转换 （cast） 是将一种数据类型转换成另一种数据类型。如果将一个整型值赋给一个浮点类型的变量，编译器会默认将其转换成浮点类型。转换是非常有用的，但是它也会带来一些问题，比如在转换指针时，我们很可能将其转换成一个比它更大的类型，但这可能会破坏其他的数据。应该小心类型转换，因为转换也就相当于对编译器说：忘记类型检查，把它看做其他的类型。一般情况下，尽量少的去使用类型转换，除非用来解决非常特殊的问题。无论什么原因，任何一个程序如果使用很多类型转换都值得怀疑。

标准 `C++` 提供了一个显示的转换的语法，来替代旧的C风格的类型转换。

使用 C 风格的强制转换可以把想要的任何东西转换成我们需要的类型。那为什么还需要一个新的 `C++` 类型的强制转换呢?

新类型的强制转换可以提供更好的控制强制转换过程，允许控制各种不同种类的强制转换。`C++` 风格的强制转换其他的好处是，它们能更清晰的表明它们要干什么。程序员只要扫一眼这样的代码，就能立即知道一个强制转换的目的。

1.1 静态转换(static_cast)

主要应用于基本数据类型的转换和类层次结构中基类（父类）和 派生类（子类）之间指针或引用的转换。

> - 进行上行转换（把派生类的指针或引用转换成基类表示）是安全的；
> - 进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的。
> - 用于基本数据类型之间的转换，如把 int 转换成 char，把 char 转换成 int。这种转换的安全性也要开发人员来保证。

```
class Animal {};
class Dog : public Animal {};
class Other {};

// 基础数据类型转换
void test_01() {
    char a = 'a';
    double b = static_cast<double>(a);
}

// 继承关系指针互相转换
void test_02() {
    // 继承关系指针转换
    Animal *animal01 = nullptr;
    Dog *dog01 = nullptr;
    // 子类指针转成父类指针,安全
    Animal *animal02 = static_cast<Animal*>(dog01);
    // 父类指针转成子类指针，不安全
    Dog *dog02 = static_cast<Dog*>(animal01);
}

// 继承关系引用相互转换
void test_03() {
    Animal ani_ref;
    Dog dog_ref;
    // 继承关系指针转换
    Animal& animal01 = ani_ref;
    Dog& dog01 = dog_ref;
    // 子类指针转成父类指针,安全
    Animal& animal02 = static_cast<Animal&>(dog01);
    // 父类指针转成子类指针，不安全
    Dog& dog02 = static_cast<Dog&>(animal01);
}

// 无继承关系指针转换
void test_04() {
    Animal* animal01 = NULL;
    Other* other01 = NULL;

    // 转换失败
    // Animal *animal02 = static_cast<Animal*>(other01);
}
```

1.2 动态转换(dynamic_cast)

> - dynamic_cast 主要用于类层次间的上行转换和下行转换;
> - 在类层次间进行上行转换时，dynamic_cast 和 static_cast 的效果是一样的;
> - 在进行下行转换时，dynamic_cast 具有类型检查的功能，比 static_cast 更安全;

```
class Animal {
public:
    virtual void ShowName() = 0;
};

class Dog : public Animal {
public:
    virtual void ShowName() override {
        cout << "I am a dog!" << endl;
    }
};

class Other {
public:
    void PrintSomething(){
        cout << "我是其他类!" << endl;
    }
};

// 普通类型转换
void test_01() {
    int a = 10;
    // 不支持基础数据类型
    // double a = dynamic_cast<double>(a);
}

// 继承关系指针
void test_02() {
    Animal* animal01 = NULL;
    Dog* dog01 = new Dog;
    // 子类指针转换成父类指针 可以
    Animal* animal02 = dynamic_cast<Animal*>(dog01);
    animal02->ShowName();
    // 父类指针转换成子类指针 不可以
    // Dog *dog02 = dynamic_cast<Dog*>(animal01);
}

// 继承关系引用
void test_03() {
    Dog dog_ref;
    Dog& dog01 = dog_ref;

    // 子类引用转换成父类引用 可以
    Animal& animal02 = dynamic_cast<Animal&>(dog01);
    animal02.ShowName();
}

// 无继承关系指针转换
void test_04() {
```

```
    Animal* animal01 = NULL;
    Other* other = NULL;

    // 不可以
    // Animal* animal02 = dynamic_cast<Animal*>(other);
}
```

1.3 常量转换(const_cast)

该运算符用来修改类型的const属性。

> - 常量指针、引用被转化成非常量指针和引用，并且仍然指向原来的对象；

注意： 不能直接对非指针和非引用的变量使用const_cast操作符去直接移除它的const。

```
// 常量指针转换成非常量指针
void test_01() {
    const int* p = nullptr;
    int* np = const_cast<int*>(p);
    int* pp = NULL;
    const int* npp = const_cast<const int*>(pp);
    const int a = 10;  // 不能对非指针或非引用进行转换
    // int b = const_cast<int>(a);

// 常量引用转换成非常量引用
void test_02() {
    int num = 10;
    int &refNum = num;
    const int& refNum2 = const_cast<const int&>(refNum);
}
```

1.4 重新解释转换(reinterpret_cast)

这是最不安全的一种转换机制，最有可能出问题。

主要用于将一种数据类型从一种类型转换为另一种类型。它可以将一个指针转换成一个整数，也可以将一个整数转换成一个指针。

二 智能指针

2.1 智能指针介绍

C++里面的四个智能指针：

```
auto_ptr        // C++11中已经被弃用

unique_ptr

shared_ptr

weak_ptr
```

其中后三个是C++11支持，并且第一个已经被C++11弃用。

2.2 为什么要使用智能指针

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

2.3 智能指针的原理

智能指针主要用于管理在堆上分配的内存，它将普通的指针封装为一个栈对象。当栈对象的生存周期结束后，会在析构函数中释放掉申请的内存，从而防止内存泄漏。C++中最常用的智能指针类型为shared_ptr,它采用引用计数的方法，记录当前内存资源被多少个智能指针引用。该引用计数的内存在堆上分配。当新增一个时引用计数加1，当过期时引用计数减一。只有引用计数为0时，智能指针才会自动释放引用的内存资源。对shared_ptr进行初始化时不能将一个普通指针直接赋值给智能指针，因为一个是指针，一个是类。可以通过make_shared函数或者通过构造函数传入普通指针。并可以通过get函数获得普通指针。

2.3.1 auto_ptr

（C++98的方案，C++11已经抛弃）采用所有权模式。

```
auto_ptr<string> p1 (new string ("I reigned lonely as a cloud."));
auto_ptr<string> p2;
p2 = p1;     // auto_ptr不会报错.
```

此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto_ptr的缺点是：存在潜在的内存崩溃问题！

2.3.2 unique_ptr

unique_ptr实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如以"new创建对象后因为发生异常而忘记调用delete")特别有用。

采用所有权模式，还是上面那个例子

```
unique_ptr<string> p3(new string("auto"));     // #4
unique_ptr<string> p4;                         // #5
p4 = p3;     // 此时会报错！！
```

编译器认为p4 = p3非法，避免了p3不再指向有效数据的问题。尝试复制p3时会编译器出错。

另外unique_ptr还有更聪明的地方：当程序试图将一个unique_ptr赋值给另一个时，如果源unique_ptr是个临时右值，编译器允许这么做；如果源unique_ptr将存在一段时间，编译器将禁止这么做，比如：

```
unique_ptr<string> pu1(new string("hello world"));
unique_ptr<string> pu2;
pu2 = pu1;                                          // #1 不允许
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string("You"));    // #2 允许
```

其中#1留下悬挂的unique_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique_ptr，因为它调用 unique_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。

注意：如果确实想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move()，让你能够将一个unique_ptr赋给另一个。尽管转移所有权后 还是有可能出现原有指针调用（调用就崩溃）的情况。但是这个语法能强调你是在转移所有权，让你清晰的知道自己在做什么，从而不乱调用原有指针。

另外：boost库的boost::scoped_ptr也是一个独占性智能指针，但是它不允许转移所有权，从始而终都只对一个资源负责，它更安全谨慎，但应用的范围更狭窄。

例如：

```
unique_ptr<string> ps1, ps2;
ps1 = demo("hello");
ps2 = std::move(ps1);
ps1 = demo("alexia");
cout << *ps2 << *ps1 << endl;
```

2.3.3 shared_ptr

shared_ptr实现共享式拥有概念。多个智能指针可以指向相同对象，该对象和其相关资源会在"最后一个引用被销毁"时候释放。从名字就可以看出资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数use_count()来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入auto_ptr,unique_ptr,weak_ptr来构造。当我们调用release()时，当前指针会释放资源所有权，计数减一。当计数等于0时，资源会被释放。

shared_ptr 是为了解决 auto_ptr 在对象所有权上的局限性(auto_ptr 是独占的)，在使用引用计数的机制上提供了可以共享所有权的智能指针。

成员函数：

> - use_count 返回引用计数的个数
> - unique 返回是否是独占所有权(use_count 为 1)
> - swap 交换两个 shared_ptr 对象(即交换所拥有的对象)
> - reset 放弃内部对象的所有权或拥有对象的变更，会引起原有对象的引用计数的减少
> - get 返回内部对象(指针)，由于已经重载了()方法，因此和直接使用对象是一样的.

```
shared_ptr<int> sp(new int(1));
```

sp 与 sp.get() 是等价的。

share_ptr 的简单例子：

```
int main(int argc, char *argv[]) {
```

```
    string *s1 = new string("s1");
    shared_ptr<string> ps1(s1);
    shared_ptr<string> ps2;
    ps2 = ps1;

    cout << ps1.use_count() << endl;    // 2
    cout << ps2.use_count() << endl;    // 2
    cout << ps1.unique() << endl;       // 0

    string *s3 = new string("s3");
    shared_ptr<string> ps3(s3);

    cout << ps1.get() << endl;      // 033AEB48
    cout << ps3.get() << endl;      // 033B2C50
    swap(ps1, ps3);                 // 交换所拥有的对象
    cout << ps1.get() << endl;      // 033B2C50
    cout << ps3.get() << endl;      // 033AEB48

    cout << ps1.use_count() << endl;        // 1
    cout << ps2.use_count() << endl;        // 2
    ps2 = ps1;
    cout << ps1.use_count() << endl;        // 2
    cout << ps2.use_count() << endl;        // 2
    ps1.reset();    // 放弃ps1的拥有权，引用计数的减少
    cout << ps1.use_count() << endl;    // 0
    cout << ps2.use_count() << endl;    // 1
}
```

2.3.4 weak_ptr

share_ptr虽然已经很好用了，但是有一点share_ptr智能指针还是有内存泄露的情况，当两个对象相互使用一个shared_ptr成员变量指向对方，会造成循环引用，使引用计数失效，从而导致内存泄漏。

weak_ptr是一种不控制对象生命周期的智能指针，它指向一个shared_ptr管理的对象。进行该对象内存管理的是那个强引用shared_ptr，weak_ptr只是提供对管理对象的访问手段。weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作，它只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造，它的构造和析构不会引起引用记数的增加或减少。weak_ptr是用来解决shared_ptr相互引用时的死锁问题,如果说两个shared_ptr相互引用,那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和shared_ptr之间可以相互转化，

shared_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared_ptr。

```
class B;    // 声明
class A {
public:
    shared_ptr<B> pb_;
    ~A() { cout << "A delete\n"; }
};

class B {
public:
```

```
    shared_ptr<A> pa_;
    ~B() { cout << "B delete\n"; }
};

void fun() {
    shared_ptr<B> pb(new B());
    shared_ptr<A> pa(new A());
    cout << pb.use_count() << endl;     // 1
    cout << pa.use_count() << endl;     // 1
    pb->pa_ = pa;
    pa->pb_ = pb;
    cout << pb.use_count() << endl;     // 2
    cout << pa.use_count() << endl;     // 2
}

int main(int argc, char *argv[]) {
    fun();
    return 0;
}
```

可以看到fun函数中pa，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减1，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A、B的析构函数没有被调用）运行结果没有输出析构函数的内容，造成内存泄露。如果把其中一个改为weak_ptr 就可以了，我们把类A里面的shared_ptr pb_，改为weak_ptr pb，运行结果如下：

```
1
1
1
2
B delete
A delete
```

这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减1，同时pa析构时使A的计数减1，那么A的计数为0，A得到释放。

注意：我们不能通过weak_ptr直接访问对象的方法，比如B对象中有一个方法print()，我们不能这样访问，pa→pb→print()，因为pb_是一个weak_ptr，应该先把它转化为shared_ptr，如：

```
shared_ptr<B> p = pa->pb_.lock();
p->print();
```

> - weak_ptr 没有重载*和→但可以使用 lock 获得一个可用的 shared_ptr 对象. 注意，weak_ptr 在使用前需要检查合法性。
> - expired 用于检测所管理的对象是否已经释放，如果已经释放，返回 true；否则返回 false。
> - lock 用于获取所管理的对象的强引用(shared_ptr). 如果 expired 为 true，返回一个空的 shared_ptr；否则返回一个 shared_ptr，其内部对象指向与 weak_ptr 相同。
> - use_count 返回与 shared_ptr 共享的对象的引用计数。
> - reset 将 weak_ptr 置空。
> - weak_ptr 支持拷贝或赋值，但不会影响对应的 shared_ptr 内部对象的计数。

weak_ptr 一般通过 share_ptr 来构造，通过 expired 函数检查原始指针是否为空，lock 来转化为 share_ptr。

课堂小结

- 静态转换，动态转换，常量转换，重新解释转换的使用
- 智能指针的介绍和使用
- 智能指针的原理

随堂作业