第14讲 C++新特性

目录 CONTENTS

14.1 智能指针

- 在C++中，使用new申请内存后，一定要使用delete释放这块空间，否则会造成内存泄漏。

```
void memoryLeak1() {
    string *str = new string("动态分配内存！");
    return; //没有调用delete
}
void memoryLeak2() {
    string *str = new string("内存泄露！");
    if (true) {
        return;
    }
    delete str; //虽然有delete，但是没有执行
}
```

14.1 智能指针

- 如果指针有一个析构函数，在指针过期时能自动释放它指向的内存。
- 这正是智能指针的思想，C++98使用模板auto_ptr解决这一问题。
- C++11又增加了unique_ptr和shared_ptr两种智能指针。
- 要创建智能指针对象，必须包含头文件memory。

14.1.1 auto_ptr

- auto_ptr是这样的一种指针: 它是"其所指向的对象"的拥有者。
- 所以，当身为对象拥有者的auto_ptr被销毁时，该对象也将被销毁。
- auto_ptr的使用格式：

auto_ptr<类型> 指针名(new 类型)

14.1.1 auto_ptr

- auto_ptr的构造函数是explicit，阻止了一般指针隐式转换为 auto_ptr的构造，所以不能直接将一般类型的指针赋值给auto_ptr类型的对象，必须用auto_ptr的构造函数创建对象。
- 由于auto_ptr对象析构时会删除它所拥有的指针，所以使用时避免多个auto_ptr对象管理同一个指针。

```
class Data
{
        int m_data;
public:
        Data(int data): m_data(data)
        {   }
        ~Data()
        {
            cout<<"Data is deleted"<<endl;
        }
        void ShowData()
        {
            cout<<"Data:"<<m_data<<endl;
        }
};
int main()
{
        auto_ptr<Data> p1(new Data(10));
        p1->ShowData();
        return 0;
}
```

智能指针可以像普通指针一样访问成员

14.1.1 auto_ptr

- get()：获取智能指针托管的指针地址，必须通过“.”访问。

```
Data* p2 = p1.get();
p2->ShowData();
```

- release()：取消智能指针对动态内存的托管针，改由程序员进行管理。

```
Data* p3 = p1.release();
delete p3;
```

14.1.1 auto_ptr

- reset()：重置智能指针托管的内存地址，如果地址不一致，原来的会被析构掉。

```
auto_ptr<Data> p1(new Data(10));
p1.reset();              //释放，变为NULL
Data* data = new Data(20);
p1.reset(data);      //托管新的内存
```

14.1.1 auto_ptr被淘汰的原因

- （1）复制会改变资源的所有权。

```
auto_ptr<Data> p4(p1);
p1->ShowData(); // p1变为NULL
p4->ShowData();
```

- （2）赋值会改变资源的所有权。

```
auto_ptr<Data> p5(new Data(20));
p5->ShowData();
p1 = p5; //p1放弃原来托管的对象，p5变为NULL
cout<<p1.get()<<endl;
cout<<p5.get()<<endl;
```

14.1.1 auto_ptr被淘汰的原因

- （3）不支持数组的指针管理。

```
auto_ptr<int []> p6(new int [5]);
```

14.1.2 unique_ptr

- unique_ptr 和 auto_ptr用法几乎一样，除了一些特性：
- （1）两个指针不能指向同一个资源。

```
unique_ptr<Data> p1(new Data(10));
p1->ShowData();
unique_ptr<Data> p2(new Data(20));
p2->ShowData();
p1 = p2; //不允许
cout<<p1.get()<<endl;
cout<<p2.get()<<endl;
```

14.1.2 unique_ptr

- unique_ptr 和 auto_ptr用法几乎一样，除了一些特性：
- （2）无法进行左值unique_ptr复制构造。

```
unique_ptr<Data> p1(new Data(10));
p1->ShowData();
unique_ptr<Data> p2(p1); //不允许
```

14.1.2 unique_ptr

- unique_ptr 和 auto_ptr用法几乎一样，除了一些特性：
- （3）支持数组的内存管理。

```
unique_ptr<int[]> p3(new int[5]);
p3[2] = 7;
```

14.1.3 shared_ptr

- unique_ptr 和 auto_ptr存在的问题

```
auto_ptr<Data> p1(new Data(10));
p1.reset();
Data* data = new Data(20);
p1.reset(data);
{
        auto_ptr<Data> p2;
        p2.reset(data); //p2申请管理data，但p1未知
}
p1->ShowData();
```

- 原因：排他，不能共享指针

14.1.3 shared_ptr

- shared_ptr 采用的策略：记录引用特定内存对象的智能指针数量，当复制或拷贝时，引用计数加1，当智能指针析构时，引用计数减1，如果计数为零，代表已经没有指针指向这块内存，那么就释放它！

14.1.3 shared_ptr

```
shared_ptr<Data> p1(new Data(10));
shared_ptr<Data> p2;
cout<<"p1 is used count:"<<p1.use_count()<<endl;
cout<<"p2 is used count:"<<p2.use_count()<<endl;

p2 = p1;
cout<<"p1 is used count:"<<p1.use_count()<<endl;
cout<<"p2 is used count:"<<p2.use_count()<<endl;

shared_ptr<Data> p3(p1);
cout<<"p1 is used count:"<<p1.use_count()<<endl;
cout<<"p2 is used count:"<<p2.use_count()<<endl;
```

14.1.3 shared_ptr

```
p1.reset();
cout<<"p1 is used count:"<<p1.use_count()<<endl;
cout<<"p2 is used count:"<<p2.use_count()<<endl;

p2.swap(p3); //对象的引用计数不变
cout<<"p3 is used count:"<<p3.use_count()<<endl;
cout<<"p2 is used count:"<<p2.use_count()<<endl;
```

14.1.3 shared_ptr

- shared_ptr 存在的问题：要注意避免对象交叉使用智能指针的情况。

```
class B;
class A
{
        shared_ptr<B> m_b;
public:
        A()
        {
            cout<<"A的构造函数"<<endl;
        }
        ~A()
        {
            cout<<"A的析构函数"<<endl;
        }
        void SetPTRB(shared_ptr<B> b)
        {
            m_b = b;
        }
};
```

14.1.3 shared_ptr

```
class B
{
    shared_ptr<A> m_a;
public:
    B()
    {
        cout<<"B的构造函数"<<endl;
    }
    ~B()
    {
        cout<<"B的析构函数"<<endl;
    }
    void SetPTRA(shared_ptr<A> a)
    {
        m_a = a;
    }
};
```

14.1.3 shared_ptr

```
int main()
{
        {
            shared_ptr<A> pa(new A);
            shared_ptr<B> pb(new B);
            pa->SetPTRB(pb);
            pb->SetPTRA(pa);
            cout<<"A的计数"<<pa.use_count()<<endl;
            cout<<"B的计数"<<pb.use_count()<<endl;
        }
        return 0;
}
```

不能释放内存

14.2 类型转换

- 强制类型转换有一定风险，有的转换并不一定安全。
  - 把int整形数值转换成一个指针类型
  - 把基类指针转换成派生类指针
  - 把一种函数指针转换成另一种函数指针
  - 把常量指针转换成非常量指针

14.2 类型转换

- C语言强制类型转换的以下三个缺点：
  - 没有从形式上体现转换功能和风险的不同
  - 将多态基类指针转换成派生类指针时不检查安全性
  - 难以在程序中寻找到底什么地方进行了强制类型转换

14.2 类型转换

- C++ 引入了四种功能不同的强制类型转换运算符以进行强制类型转换：
  - static_cast
  - reinterpret_cast
  - const_cast
  - dynamic_cast

强制类型转换运算符 <要转换到的类型> (待转换的表达式)

14.2.1 static_cast

- static_cast，编译时期的静态类型转换。
  - 完成基础数据类型，整型和浮点型、字符型之间的互相转换
  - 同一个继承体系中类型的转换
  - 任意类型与空指针类型void*之间的转换
  - 不能用于在不同类型的指针之间互相转换
  - 不能用于整型和指针之间的互相转换

14.2.1 static_cast

```
int main()
{
	int a = 10;
	double pi = 3.14;
	int* p = &a;
	int b = static_cast<int>(pi);
	//错误，int c = static_cast<int>(p);
	//错误，int* q = static_cast<int*>(a);
	return 0;
}
```

14.2.2 reinterpret_cast

- reinterpret_cast，<>中必须是一个指针、引用、算术类型、函数指针或者成员指针。
  - 进行各种不同类型的指针之间
  - 不同类型的引用之间
  - 指针和能容纳指针的整数类型之间
  - 最不安全的类型转换

14.2.2 reinterpret_cast

```
int main()
{
	int a = 10;
	double pi = 3.14;
	int* pa = &a;
	double* ppi = &pi;
	Data data(20);
	Data* pdata = reinterpret_cast<Data*>(pa);
	ppi = reinterpret_cast<double*>(&data);
	int& b = reinterpret_cast<int&>(pi);
	return 0;
}
```

14.2.3 const_cast

- const_cast，仅用于进行去除引用，转换为同类型的非 const 属性的转换。
  - 将 const 引用转换为同类型的非 const 引用
  - 将 const 指针转换为同类型的非 const 指针

14.2.3 const_cast

```
class C
{
public:
    const int m_data;
public:
    C(int data): m_data(data)
    {  }
};
```

简单类型的变量，即使使用了const_cast其值不能修改，类中的成员变量则可以修改。

```
int main()
{
    C c1(100);
    cout<<c1.m_data<<endl;
    //c1.m_data = 200;
    int& rdata = const_cast<int&>(c1.m_data);
        cout<<c1.mrdata = 200;
_data<<endl;

    const int a = 10;
    int* p = const_cast<int*>(&a); // int* p = &a;错误
    *p = 9;
    cout<<*p<<endl;
    int& r = const_cast<int&>(a); // int& r = a;错误
    return 0;
}
```

14.2.4 dynamic_cast

- dynamic_cast，通过“运行时类型检查”来保证安全性的换。
  - 只能用于具有继承关系，且包含多态性的类型转换

14.2.4 dynamic_cast

```
class B
{
public:
    virtual void f(){}
};

class D: public B
{
public:
    void f(){}
};
```

```
int main()
{
    B b;
    D d;
    D* pd;
    pd = reinterpret_cast<D*>(&b);
    if(pd != NULL) // 转换成功
        cout<<"reinterpret cast succeeded"<<endl;
    pd = dynamic_cast<D*>(&b);
    if(pd == NULL) //转换失败
        cout<<"dynamic cast failed"<<endl;
    return 0;
}
```