第六章 类和对象的使用进阶

模块6.1：构造函数进阶

主讲教师：同济大学计算机科学与技术学院 陈宇飞
同济大学计算机科学与技术学院 龚晓亮

目录

- 构造函数初始值列表
- 委托构造函数
- 默认构造函数的作用
- 隐式的类类型转换
- 复制构造函数的调用时机

1.1 构造函数初始值列表

- 初始化：直接初始化数据成员；赋值：先初始化再赋值
- 构造函数的初始值有时必不可少

```
class ConstRef {
public:
      ConstRef(int ii);
private:
      int i;
      const int ci;
      int &ri;
};
```

```
// 错误：ci和ri必须被初始化
ConstRef::ConstRef(int ii)
{ // 赋值
      i = ii;   // 正确
      ci = ii;  // 错误：不能给const赋值
      ri = i;   // 错误：ri没被初始化
}
```

1.1 构造函数初始值列表

- 初始化：直接初始化数据成员；赋值：先初始化再赋值
- 构造函数的初始值有时必不可少

```
class ConstRef {
public:
      ConstRef(int ii);
private:
      int i;
      const int ci;
      int &ri;
};
```

```
// 正确：显式的初始化引用和const成员
ConstRef::ConstRef(int ii): i(ii),
ci(i), ri(ii) { }
```

结论：如果成员是const、引用，或者属于某种未提供默认构造函数的类类型，必须通过构造函数初始值列表为这些成员提供初值。

1.1 构造函数初始值列表

- 成员初始化的顺序：与类定义中出现的顺序一致，跟初始化列表中的顺序无关
  - 建议1：构造函数初始值的顺序与成员声明的顺序保持一致
  - 建议2：尽量避免使用某些成员初始化其它成员

```
class X {
   int i;
   int j;
public:
   //实际上是i先被初始化！
   X(int val): j(val),i(j){};
};
```

```
class X {
   int i;
   int j;
public:
   //正确
   X(int val): i(val),j(val){};
};
```

1.1 构造函数初始值列表

- 默认实参和构造函数：

➢若构造函数为所有参数都提供了默认实参，则相当于定义了默认构造函数

```
class Sales_data{
      Sales_data (string s = ""):bookNo(s){ }
};
```

上例：当没有给定实参或者给定了一个string实参时，类创建的对象相同。

（因为不提供实参也能调用上述构造函数，所以该构造函数实际就为类提供了默认构造函数）

1.1 构造函数初始值列表

- 默认实参和构造函数：

➢不能为构造函数的全部形参都提供默认实参

```
//接受string的构造函数
class Sales_data{
      Sales_data (string s = ""):bookNo(s){ }
};
//接受istream&参数的构造函数
class Sales_data{
      Sales_data (istream &is = cin) { is >> *this; }
};
```

错误

上例：不提供任何实参的创建类对象时，产生二义性

目录

- 构造函数初始值列表
- 委托构造函数
- 默认构造函数的作用
- 隐式的类类型转换
- 复制构造函数的调用时机

2.1 委托构造函数

- 委托构造函数使用它所属类的其他构造函数执行自己的初始化过程，或者说它把自己的一些(或全部)职责委托给了其他的构造函数
- 委托构造函数也有一个成员初始值的列表和一个函数体。在委托构造函数内，成员初始值列表只有唯一的一个入口，就是类名本身。和其他成员初始值一样，类名后面紧跟圆括号括起来的参数列表，参数列表必须与类中另外一个构造函数匹配

```
class Sales_data {
public:
    //非委托构造函数接收三个实参，使用这些实参初始化数据成员，然后结束工作
    Sales_data(string s,unsigned cnt,double price):
        bookNo(s),units_sold(cnt),revenue(cnt*price){}

    //其余构造函数全都委托给另一个构造函数
    Sales_data():Sales_data(" ", 0, 0) {}
        定义默认构造函数令其使用三参数的构造函数完成初始化过程
    Sales_data(string s):Sales_data(s,0,0){}
        定义接收一个string的构造函数，同样委托给了三参数版本
    Sales_data(istream &is):Sales_data() { read(is, *this); }
        定义接收istream &的构造函数，它委托给了默认构造函数，默认构造函数接
        着委托给三参数的构造函数。当接受委托的构造函数执行完后，接着执行
        istream &构造函数体的内容，即调用read函数读取给定的istream
private:
    string bookNo; unsigned units_sold; double revenue;
};
```

目录

- 构造函数初始值列表
- 委托构造函数
- 默认构造函数的作用
- 隐式的类类型转换
- 复制构造函数的调用时机

3.1 默认构造函数的作用

- 默认初始化发生的情况：
  - 在块作用域内不使用任何初始值定义一个非静态变量或数组
  - 类本身含有类类型的成员并且使用合成的默认构造函数
  - 当类类型的成员没有在构造函数初始值列表中显式的初始化
- 值初始化发生的情况：
  - 数组初始化的过程中如果提供的初始值少于数组的大小
  - 不使用初始值定义一个局部静态变量
  - 通过书写形如 T() 的表达式显式地请求值初始化时，T是类型名

➡ 类必须包含一个默认构造函数以便在上述情况下使用

```
//例：类的数据成员缺少默认构造函数
class NoDefault {
public:
        NoDefault(const string&){…};
};
```

注意：使用默认构造函数：NoDefault() {}; 才可以正常编译通过！！

```
struct A {
        NoDefault my_mem; //默认public
};

A a;  //错误，不能为A合成构造函数

struct B {
        B(){} //错误，b_member没有初始值
        NoDefault b_member;
};
```

合成构造函数：如果用户定义的类中没有显式的定义任何构造函数，编译器才会自动为该类型生成默认构造函数，称为合成的构造函数

```
#include <iostream>
#include <string>
using namespace std;
class NoDefault {
public:
      NoDefault(const string&) {…};
      NoDefault() {};
};
NoDefault obj1()
{     cout << "helloworld" << endl;
      NoDefault obj;
      return obj;
}
int main()
{     NoDefault obj2 = obj1();
      return 0;
}
```

```
//注意区分：
NoDefault obj1() //定义了一个obj1函数
NoDefault obj2   //定义了一个obj2对象
```

目录

- 构造函数初始值列表
- 委托构造函数
- 默认构造函数的作用
- 隐式的类类型转换
- 复制构造函数的调用时机

4.1 隐式的类类型转换

- 转换构造函数（converting constructor）：
  - 当一个构造函数只有一个参数，而且该参数又不是本类的const引用时，这种构造函数称为转换构造函数
  - 转换构造函数的作用是将一个其他类型的数据转换成一个类的对象

注意：

转换构造函数只能有一个参数。如果有多个参数，就不是转换构造函数

```
class Sales_data {
private:
    string book_no;
    unsigned units_sold = 1;
    double revenue = 1.0;

public:
    Sales_data() = default;//不接受任何实参，默认构造函数
    Sales_data(const string& s) : book_no(s) {}   //类型转换构造函数
    Sales_data(const string& s, unsigned n, double p) :
                book_no(s), units_sold(n), revenue(p* n) {}
    Sales_data(istream&) {};

    Sales_data& combine(const Sales_data&);
    //其他成员函数…
};
```

```
string null_book = "9-999-999-9";
//构造一个临时的Sales_data对象item
item.combine(null_book);
```

4.1 隐式的类类型转换

- 只允许一步类类型转换：

```
item.combine("9-999-999-9");
//错误：需要用户定义的两种转换："9-999-999-9"到string到Sales_data

item.combine(string("9-999-999-9"));
//正确：显式地转换成string，隐式地转换成Sales_data

item.combie(Sales_data("9-999-999-9"));
//正确：隐式地转换成string，显式地转换成Sales_data
```

```
Sales_data(const string& s) : book_no(s) {}   //类型转换构造函数
Sales_data& combine(const Sales_data&);
```

4.1 隐式的类类型转换

- 类类型转换不是总有效：

item.combine(cin);//隐式地将 cin 转换成 Sales_data，这个转换执行接受一个 istream 的 Sales_data 构造函数，该构造函数通过读取标准输入创建了一个临时的 Sales_data 对象，随后将得到的对象传递给 combine。该对象是一个临时量，一旦 combine 完成就不能再访问它了

```
...
Sales_data(istream&) {};
Sales_data& combine(const Sales_data&);
```

4.1 隐式的类类型转换

➢ 抑制构造函数定义的隐式类型转换：

- 将构造函数声明为 explicit 可以阻止构造函数的隐式类型转换

```
explicit Sales_data(const std::string &s):bookNo(s){ }
explicit Sales_data(std::istream&);
```

此时：

```
item.combine(null_book);    //错误
item.combine(cin);          //错误
```

4.1 隐式的类类型转换

```
explicit Sales_data(const std::string &s):bookNo(s){ }
explicit Sales_data(std::istream&);
```

- 关键字explicit只对一个实参的构造函数有效，需要多个实参的构造函数不能用于隐式转换，所以无须将这些构造函数指定为explicit
- 只能在类内声明构造函数时使用explicit，类外定义时不应重复
- 当使用explicit声明构造函数时，将只能以直接初始化的形式使用

```
Sales_data item1(null_book); //正确，直接初始化
Sales_data item2 = null_book;
        //错误，不能将explicit构造函数用于拷贝形式的初始化过程
```

4.1 隐式的类类型转换

```
explicit Sales_data(const std::string &s):bookNo(s){ }
explicit Sales_data(std::istream&);
Sales_data& combine(const Sales_data&);
```

- 为转换显式地使用构造函数：

item.combine(Sales_data(null_book));

//直接使用 Sales_data 的构造函数，该调用通过接受 string 的构造函数创建一个临时的 Sales_data 对象

item.combine(static_cast<Sales_data>(cin));

//使用_cast 执行了显式的转换：使用 istream 构造函数创建了一个临时static的 Sales_data 对象

目录

- 构造函数初始值列表
- 委托构造函数
- 默认构造函数的作用
- 隐式的类类型转换
- 复制构造函数的调用

对象复制的基本概念（复习回顾）

- 含义：建立一个新对象,其值与某个已有对象完全相同
- 使用：

  类 对象名(已有对象名)

  类 对象名=已有对象名

  Time t1(14,15,23), t2(t1), t3=t1;
- 与对象赋值的区别：

  Time t1(14,15,23),t2, t3=t1; //复制：定义语句中

  t2 = t1; //赋值：执行语句中
- 对象复制的实现：建立新对象时自动调用复制构造函数（也称为拷贝构造函数）

5.1 复制构造函数基本概念

- 形式：

  类名(const 类名 &引用名)

  - 用一个对象的值去初始化另一个对象
  - 若不定义复制构造函数，则系统自动定义一个，参数为const型引用，函数体为对应成员内存拷贝
  - 若定义了复制构造函数，则系统缺省定义的消失
  - 允许体内实现或体外实现
  - 复制构造函数和普通构造函数（可能多个）的地位平等，调用其中一个后就不再调用其它构造函数

5.2 复制构造函数调用

- 复制构造函数和普通构造函数(可能多个)的地位平等，调用其中一个后就不再调用其它构造函数：

```
class Time {
        ...
    public:
      Time(int h=0);
      Time(int h, int m, int s=0);
      Time(const Time &t);
};
```

```
int main()
{
      Time t1;
      Time t2(10);
      Time t3(1,2,3);
      Time t4(4,5);
      Time t5(t2);
      Time t6 = t4;
}
```

```
#include <iostream>
using namespace std;
int tcount = 0; //全局变量，计数器
class Time {
  private:
    int hour, minute, sec;
 public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time() {cout<<"tcount="<<--tcount<<endl;}
    void display()
    {cout<<hour<<":"<<minute<<":"<<sec<<endl;}
};
Time::Time(int h, int m, int s)
{   hour = h;
    minute = m;
    sec = s;
    ++tcount; //计数器+1
    cout << "普通构造" << endl;
}
```

```
Time::Time(const Time &t)
{  hour = t.hour - 1;
   minute = t.minute - 1;
   sec = t.sec - 1;
   ++tcount; //计数器+1
   cout << "复制构造" << endl;
}

int main()
{  //用对象初始化新对象
   Time t1(14,15,23),t2(t1);
   t2.display();
}
```

```
普通构造
复制构造
13:14:22
tcount=1
tcount=0
```

```
#include <iostream>
using namespace std;
int tcount = 0; //全局变量，计数器
class Time {
  private:
    int hour, minute, sec;
 public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time() {cout<<"tcount="<<--tcount<<endl;}
    void display()
    {cout<<hour<<":"<<minute<<":"<<sec<<endl;}
};
Time::Time(int h, int m, int s)
{   hour = h;
    minute = m;
    sec = s;
    ++tcount; //计数器+1
    cout << "普通构造" << endl;
}
```

```
Time::Time(const Time &t)
{  hour = t.hour - 1;
   minute = t.minute - 1;
   sec = t.sec - 1;
   ++tcount; //计数器+1
   cout << "复制构造" << endl;
}

void fun(Time t)
{  //函数形参为对象
   t.display();
}
int main()
{  Time t1(14,15,23);
   fun(t1);
}
```


```
普通构造
复制构造
13:14:22
tcount=1
tcount=0
```

```
#include <iostream>
```

在这个代码中，没有触发拷贝构造函数的原因是**编译器优化**，特别是**返回值优化（Return Value Optimization, RVO）** 或 **移动语义（Move Semantics）**。

具体分析如下：

1. 在函数 `fun()` 中创建了一个局部变量 `t1`，它是一个 `Time` 对象。按照传统的 C++ 构造与拷贝规则，当 `t1` 作为返回值返回时，应该调用拷贝构造函数将 `t1` 拷贝给 `t2`。
2. 然而，现代的 C++ 编译器会使用一种叫做返回值优化（RVO）的技术。编译器会直接在 `t2` 中构造 `t1` 的值，跳过临时对象的创建和拷贝构造。因此，`t1` 不会被拷贝，拷贝构造函数不会被调用。
3. 此外，C++11 及更高版本中引入了移动语义和 `std::move`，使得在某些情况下会触发移动构造函数，而不是拷贝构造函数。不过，返回值优化已经发生时，移动语义也不会生效。

优化过程：

- 当 `t1` 在 `fun()` 中返回时，编译器知道 `t1` 不会再被使用，因此直接将 `t1` 的内存返回给 `t2`。这就是返回值优化的一部分，目的是避免不必要的对象拷贝，提升性能。

因此，在你的程序中，由于 RVO 的存在，拷贝构造函数没有被调用。

```
Time::Time(int h, int m, int s)
{   hour = h;
    minute = m;
    sec = s;
    ++tcount; //计数器+1
    cout << "普通构造" << endl;
}
```

```
Time::Time(const Time &t)
{   hour = t.hour - 1;
    minute = t.minute - 1;
    sec = t.sec - 1;
    ++tcount; //计数器+1
    cout << "复制构造" << endl;
}
```

```
Time fun() //函数返回值为对象
{   Time t1(14,15,23);
    return t1;
}
int main()
{   Time t2 = fun();
    t2.display();
}
```

```
普通构造
14:15:23
tcount=0
```

➢ 变量定义时赋初值与使用赋值语句赋初值的区别：

```
Time fun()
{
    Time t1(14,15,23);
    return t1;
}
int main()
{
    Time t2 = fun();//定义时赋初值
    t2.display();
}
```

普通构造
14:15:23
tcount=0

```
Time fun()
{
    Time t1(14,15,23);
    return t1;
}
int main()
{   Time t2;
    t2 = fun();//赋值语句赋初值
    t2.display();
}
```

普通构造
普通构造
tcount=1
14:15:23
tcount=0

均针对拷贝构造函数进行过编译器优化，具体以实际编译器运行结果为准

总结

- 构造函数初始值列表（熟悉）
- 委托构造函数（了解）
- 默认构造函数的作用（熟悉）
- 隐式的类类型转换（熟悉）
- 复制构造函数的调用时机（掌握）