第 2 篇　C++面向对象编程

20 世纪 50 年代末端倪初显的第一次软件危机，给了计算机领域的精英们大展才华的机会。结构化程序设计思想提出之后，先是出现了用函数（或子程序）进行代码封装的模式。但是，函数（子程序）是基于功能的程序代码封装，其粒度很小，在设计大程序系统时，还显得十分烦琐。于是人们开始寻找更大粒度的代码封装体。1967 年 5 月 20 日，在挪威奥斯陆郊外的小镇莉沙布举行的 IFIP TC-2 工作会议上，挪威科学家 Ole-Johan Dahl 和 Kristen Nygaard 正式发布了 Simula 67 语言。这一语言为程序设计带来一股新风。它采用类（class）作为程序代码的封装体。类是对系统中具有共同特征的实体的抽象。也就是说，采用这种语言进行程序设计，首先要分析系统涉及哪些对象（实体），并且要分析这些对象可以抽象为哪几种类型。编码针对类进行。把具体对象看作是类的实例。所以，这是一种分析式编程思想，也是一种基于类（尽管多称为面向对象）的程序设计思想。

类是一种抽象数据结构（Abstract Data Type，ADT），它封装了描述一类事物的属性以及行为。属性用数据描述，行为用函数（称为方法）描述。所以它还是基于命令式编程的，只不过封装的粒度大了，适合于构造大型程序。

面向对象编程的另外特点是通过继承与多态实现代码复用。继承允许在已有类的基础上生成新的类，多态可以赋予一个名字或符号不同的意义。从这一点上来提高程序设计的效率和可靠性。

本篇具体介绍 C++中如何应用这些机制。

第5章　类 与 对 象

C++是一种强类型语言。命令式编程中以数据为核心，强调所有的数据（包括字面量、常量、变量、表达式、函数返回等）都要属于某一种类型。在面向对象的程序设计中，它强调一切皆对象（objects），所有对象也要属于特定的类型。这些“类型”都是抽象数据类型，封装了用数据描述的对象相关属性种类和用函数表示的对象相关行为，并将这种抽象数据类型称为类（class）。类是一种将抽象类型转换为用户定义类型的载体。要使用对象，必须用类进行声明（构造），所声明的变量就是该类的对象。因此，在面向对象的编程中，要使用对象，就要先定义类。有了类，才能由类构造对象。所以，面向对象的编程实际上主要的工作是设计类。

5.1　类 的 设 计

5.1.1　类的声明与实现

1. 类的声明

面向对象程序设计认为，世界是由对象（objects）组成的。面向对象的程序就是对世界的某个子结构的描述。要对一个问题进行求解，首先要分析这个问题中有哪些对象，并进一步将这些对象抽象为几种类型，然后用类来描述这几个类。

任何一类对象，都可以从两个方面进行描述：属性（attributes）和行为（behaviors）。然后将这个现实世界中的对象用程序设计语言描述，就粗略地得到了程序世界中对象的类型——class。图 5.1 为将现实世界中的职员类型转化为 C++程序世界中的 Employee 类的示意图。

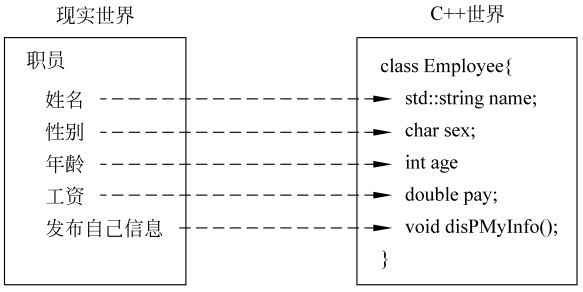

图 5.1　从现实世界到 C++世界

由此可以得到一个简单的 C++类声明的粗略框架。

（1）一个 C++类声明由类头和类体两部分组成。

（2）类头由关键字 class 引出，后面是一个类名。类名应当符合 C++标识符命名规则。为了与变量相区别，通常类名第一个字符大写。

（3）类体是由括在花括号中的类成员组成。类成员有两种：数据成员和成员函数。数据成员用于描述类的属性，在类声明中以变量声明的形式表示。成员函数用于描述类的行为，在类声明中以原型表示。

（4）类声明要以分号（;）结尾。

（5）在 C++程序中，类具有外部作用域，即它定义在所有函数的外部。

这还是个粗略的框架，意思是这还不是可用的类。要可用，还需要进行以下完善。

2. 成员函数的实现

在图 5.1 中的 C++世界中，定义了一个 Employee 类。在这个类中声明了一个函数——成员函数 dispMyInfo()，但是没有给出其函数体，即没有函数的实现部分——函数定义。那么，这个成员函数被调用时，到哪里去找其函数体呢？C++允许将成员函数的定义在类体中直接写出，也允许其在类体外写出。这里主要介绍在类体外定义的方法。

由于函数具有外部作用域，当一个类的成员函数定义在类外部时，像普通函数那样定义成员函数，就会出现一个问题：若有几个类，那么如何知道哪个函数是哪个类的成员呢？

为消除这种混乱，C++提供了一个域解析运算符（::）来指定哪个函数是哪个类的成员函数。这样，就可以写成员函数的定义了。

【代码 5-1】 在 Employee 类外定义其成员函数。

```
#include <iostream>
using namespace std;
void Employee :: dispMyInfo()
{cout << name << ","<< sex << "," << age << "," << pay << endl;}
```

注意，这里有一个问题：为什么在这个成员函数中使用的变量 name、sex、age 和 pay 没有在函数中声明就可以直接使用呢？因为在类的声明中已经声明了，它们就具有类作用域，可以供类的每个成员函数访问。

5.1.2 信息隐藏原则与类成员的访问控制

在图 5.1 的 C++世界中定义了一个 Employee 类，并在类体中声明了几个数据成员和一个成员函数。那么，Employee 类中定义的这些成员是给谁用的呢？答案有二：一是给自己用，如在代码 5-1 中定义成员函数 dispMyInfo()中，就使用了各个数据成员；另一个是在其外部使用。但是，图 5.1 中定义的这个 Employee 类的成员是不能被外部使用的。为什么呢？这要从信息隐藏原则说起。

在模块化程序设计的实践中，人们又总结出了一条经验：一个好的模块是内互动最少的模块，即内部的元素尽量不与外界联系、互动；如果非要联系，就“光明正大”地通过规定接口联系，并且让外部具有最少的操作权限。1972 年，David Parnas 将这些经验总结为信息隐蔽原则。

在面向对象的程序设计中，类是一组属性和行为的抽象和封装体。这种封装性

（encapsulation）的好处就是可以将这些成员作为一个整体，按照信息隐蔽原则把操作分为两类：一部分局限在这个封装体内部；一部分对外开放，形成一个公开的接口。于是，在设计类时将成员分为两类：公开（public）成员和私密（private）成员。它们的区别在于，公开成员可以被外部（类的定义域之外）的对象访问，而私密成员不可以被外部的对象直接访问。也就是说，类及其对象，要以公开成员作为接口。C++严格遵循了信息隐藏原则，它把所有类成员都默认为是 private 的，如果要把某些成员作为接口，就需要用关键词 public 去声明它。这样，就清楚了在图 5.1 中定义的类 Employee 不能被外部使用的原因了，因为它还没有用 public 声明的成员，即没有可访问的接口。

那么，在一个类中，哪些成员应当声明为 public 的，哪些成员应当声明为 private 的呢？一个基本的考量是：数据成员一般应声明为 private 的，成员函数可以声明为 public 的。因为，一个类的所有对象都具有相同的成员函数，没有保密的必要；而一个类的各对象具有不同的数据成员值，所以才具有隐私性质。public 的成员函数，可以被外部访问，也可以对内进行操作，是理想的接口元素。

关键词 public 和 private 是 C++保护成员的两个访问控制关键词，它们有两种用法：一种是逐成员地声明；另一种是在一个成员处声明，后面的成员只要不是变更访问控制属性，就认为还是这种访问属性。

【代码 5-2】 带有访问控制约束的 Employee 类声明。

```
#include <string>
using std :: string;
class Employee
{
    private:
       string name;
       int age;
       char sex;
       double pay;
    public:
      void dispMyInfo();
};
```

在 C++类中，私密成员是不需要特殊声明的。但是，从可移植的角度明显地用关键词 private 对私密成员加以标记，也是有好处的。

到此为止，一个类的声明和定义才基本可用。例如使用语句

```
Employee emplZgang;
```

就可以创建一个名为 emplZhang 的对象。不过，虽然如此，却还不完全。因为，创建对象时的初始化问题还没有交代清楚。

5.1.3 构造函数与析构函数

1. 对象初始化

创建一个对象，可能是空的，也可能是有特定值的。对象初始化就是创建一个有特定

值的对象。这时要涉及两个问题。

（1）在创建对象的同时，如何为对象开辟一个对象存储区。当一个 C++程序开始运行时，编译器就把其函数，包括类的成员函数保存到内存代码区。这些代码可以被任何符合语法规则和访问条件的表达式调用。而存放到栈区的类的数据成员要在执行对象创建语句时分配。

（2）另外一个问题是给对象的各数据成员赋初值。这一项工作常常可以由构造函数代劳。但是，这不是必需的。因为程序员完全可以设立 set 类型的函数来完成这个任务。例如，对于 Employee 类，可以设置下面的 set 函数进行初值设置

```
void Employee::setName(){cin >> name;}
```

但是，若让构造函数一身兼两职不是可以减少函数调用的次数吗？这何乐而不为呢？

2. 有参构造函数、无参构造函数与构造函数重载

担负对象初始化任务的是一种特殊的成员函数——构造函数（constructor）。构造函数具有如下特点。

（1）构造函数与类同名。例如 Employee 类的构造函数为 Employee()。

（2）构造函数是在声明对象时，由对象名激活。

（3）构造函数不需要写返回类型，因为其返回类型是不言而喻的。

（4）构造函数有两种形式：有参构造函数——兼有存储分配与数据成员初始化二职；无参构造函数——只有存储分配职责而不管数据成员初始化。

【代码 5-3】 Employee 类的有参构造函数。

```
#include <iostream>
#include <string>
using namespace std;

Employee::Employee(const string & nm, int ag, char sx, double py)
{
   name = nm;
   age = ag;
   sex = sx;
   pay = py;
}
```

说明：

（1）在这个构造函数中，参数 nm 使用了引用，因为字符串比较长，不希望为它开辟其他的存储空间。而且名字是不会轻易被修改的，所以用了 const 保护。

（2）当然，也可以从键盘输入这些数据成员的值。这时，参数就没有用处了。

（3）C++允许构造函数重载。重载时用参数进行区别绑定。

【代码 5-4】 Employee 类的构造函数重载测试。

（1）将类声明存储为头文件 ex0504.hpp。

```
//文件名：ex0504.hpp
#include <string>
using namespace std;
class Employee
{
    private:
       string name;
       int age;
       char sex;
       float pay;
    public:
       Employee();                                        //无参构造函数
       Employee(string nm, int ag, char sx);              //部分参数构造函数
       Employee(string nm, int ag, char sx, float py); //完全参数构造函数
      void dispMyInfo();
};
```

（2）将成员函数实现和主函数存储为程序文件 ex0504.cpp。

```
#include <iostream>
#include "ex0504.hpp"                                     //将头文件包含到当前文件
Employee::Employee()
{
    cout << "执行无参构造函数。\n";
}

Employee::Employee(string nm, int ag, char sx)
{
    cout << "执行部分参数构造函数。\n";
    name = nm;
    age = ag;
    sex = sx;
}

Employee::Employee(string nm, int ag, char sx, float pay)
{
    cout << "执行全部参数构造函数。\n";
    name = nm;
    age = ag;
    sex = sx;
    pay = py;
}

void Employee::dispMyInfo()
{
    cout << name << "," << age << "," << sex << "," << pay << endl;
}

int main()
{
```

```
    Employee empl1;
    empl1.dispMyInfo();

    Employee empl2("zhangsan",28,'f');
    empl2.dispMyInfo();

    Employee empl3("Lisi",25,'m',2345.67);
    empl3.dispMyInfo();

    return 0;
}
```

测试结果如下。

```
执行无参构造函数。
,1, ,nan
执行部分参数构造函数。
zhangsan,28,f,5.96155e-039
执行全部参数构造函数。
Lisi,25,m,2345.67
```

说明：

（1）在编译预处理命令中，头文件名要用一对尖括号（<>）括起来。这对尖括号表明这个头文件被保存在系统给出的特定位置（文件夹）中，编译器可以直接去那里找到。而自定义的头文件名要用一对双撇号（""）引起来，因为自定义头文件不在系统给出的特定位置（文件夹）中，需要编译器搜索才能找到。

（2）表达式 empl1.dispMyInfo()、empl2.dispMyInfo()和 empl3.dispMyInfo()表示分别引用 empl1、empl2 和 empl3 的 dispMyInfo()。圆点（.）运算符称为分量运算符或成员运算符，用于指定所引用的成员是哪个对象的。此外，还可以用别名（引用）或指针引用。例如下面的语句等效于 empl1.dispMyInfo()。

```
Employee *pEmpl1 = &empl1;
pEmpl1 -> dispMyInfo();
```

这里的箭头（->）称为指向分量运算符。也可以使用指针的递引用，例如

```
Employee *pEmplZhang = &emplZhang;
（* pEmplZhang）.dispMyInfo();
```

（3）成员变量初始化时，其值是不可预料的。

3. 类的默认构造函数

如果声明一个类时，没有显式地定义一个构造函数，编译器会自动生成一个默认构造函数。这个构造函数是无参的。但是，如果类中已经定义了一个构造函数，不管是有参的，还是无参的，编译器就不再生成构造函数了。

应当注意，无参构造函数与默认构造函数还是有所不同的。在无参构造函数中除了分配存储空间外，还可以做一些别的事情，如通过键盘输入对数据成员初始化，而默认构造

函数就不会做任何多余的工作。

4. 对象的生命周期与析构函数

一个对象的生命周期是在其构造，即调用构造函数开始的。这时，编译器将会为该对象的各数据成员分配需要的存储空间。与变量一样，一个对象的生命周期，视其创建时的存储属性，可以是自动的，也可以是静态的，还可以是动态的。当这个对象的生命周期结束时，编译器还会调用另一个特殊的成员函数——析构函数，将该对象的数据成员占用的存储空间回收。

析构函数有如下特点。

（1）析构函数名是类名号前加一个波浪号（~）。如 Employee 类的析构函数名为~Employee。

（2）“三无”特点：无返回类型、无参数、在一个类中独一无二。

（3）它可以是程序员自己定义的。当程序员没有定义时，在该对象生命结束时，编译器会自动为其生成一个个默认的析构函数进行善后工作。自定义析构函数与默认析构函数的不同之处在于，它可以在函数体中搞点花样。例如可以在函数体中用输出语句输出其被调用的信息。

【代码 5-5】 显示有显式构造函数和析构函数的 Employee 类对象生命周期状况示例。

（1）将类声明定义成头文件。

```
//头文件：ex0504.hpp
#include <string>
using namespace std;
class Employee
{
    private:
        string name;
        int age;
        char sex;
        double pay;
    public:
        Employee(const string & nm, int ag, char sx, double py);
        ~Employee( );
        void dispMyInfo();
};
```

（2）将类的实现和主函数存储为程序文件。

```
//程序文件：ex0504.cpp
#include <iostream>
#include "ex0504.hpp"

Employee::Employee(const string & nm, int ag, char sx, double py)
{
   name = nm;
   age = ag;
```

```
   sex = sx;
   pay = py;
   cout << "构造" << name << "\n";
}
Employee::~Employee(){cout << "析构" << name <<"\n"; }
void Employee :: dispMyInfo(){cout << name << ","<< sex << "," << age << "," << pay << endl;}

int main()
{
  Employee emplWang("Wang1",26,'f',3333.33) ;
  {
    cout << "-------------------\n";
    Employee emplZhang("Zhang1",28,'m',5555.55) ;
    emplZhang.dispMyInfo();                 //引用 emplZhang 的 dispMyInfo()
  }
  cout << "-------------------\n";
  emplWang.dispMyInfo();                    //引用 emplWang 的 dispMyInfo()
}
```

（3）程序运行情况如下。

```
构造Wang1
-------------------
构造Zhang1
Zhang1,m,28,5555.55
析构Zhang1
-------------------
Wang1,f,26,3333.33
析构Wang1
```

说明：析构函数确实是在对象寿终正寝之时来料理后事。由于对象名也是变量，所以，对于自由对象，在哪个作用域内被创建，其生命周期就会在这个作用域尾部因自动调用析构函数而结束；若对象是静态的，则其生命周期就会在程序结束时才调用析构函数而结束。

5.1.4 对象的动态存储分配

1. 对象的堆空间分配与回收

一般来说类对象都比基本类型数据占有较多的存储空间。因此，对对象进行动态内存分配，即进行堆空间的分配，比较有意义。

【代码 5-6】 Employee 对象的动态分配示例。

```
void fun(){
    pTime *pEmployee;               //创建一个指向 Employee 类的指针
    pEmployee = new Employee;       //将 pEmployee 指向 new 在堆空间分配的一个 Employee 对象空间
    ...//其他操作
    delete pTime;                   //释放堆空间
}
```

说明：

（1）从字面上看，语句“pEmployee = new Employee;”执行时，new 先分配一个空堆空