第3章 CHAPTER 3 简易C++基础

Qt是一个基于C++的开发库，在跨系统、跨硬件平台程序开发方面有着持久的积累和得天独厚的优势。学习Qt需要具有一定的C++基础。如果只有C语言基础，直接学习Qt难免会事倍功半。由于C++的知识浩如烟海，本章只能根据后续章节的需要讲解一些C++的基础知识，如面向对象编程的概念、基本输入输出、名称空间、函数重载、类的继承和派生等。通过学习本章的内容，可以打下继续学习C++和Qt的基础。

3.1 C和C++


视频讲解

3.1.1 C++简史

C++是从C语言发展而来的。要梳理C++的历史，就不得不提C语言。

C语言是一种面向过程的编程语言。1972年，贝尔实验室的Dennis Ritchie(1941.09.09—2011.10.12)以B语言为基础，在一台DEC PDP-11计算机上实现了最初的C语言。C语言不但语法简洁灵活、运算符和数据结构丰富，而且具有结构化控制语句，程序执行效率高。与同时期的其他高级语言相比，C语言可以直接访问物理地址；与汇编语言相比，C语言具有良好的可读性和可移植性。

1979年10月，贝尔实验室的Bjarne Stroustrup博士为C语言增加了类和一些其他特性，并将这种新的编程语言命名为C with class。1983年，C with class更名为C++。在C语言中，++运算符的作用是对一个变量进行递增操作，由此也可以看出C++的定位。在这一时期，C++引入了许多重要的特性，例如虚函数、函数重载、引用、const关键字、双斜线引导的单行注释等。

1985年，Bjarne Stroustrup博士的著作*C++Programming Language*出版。同年，C++的商业版本问世。1990年，*Annotated C++Reference Manual*发布，Borland公司的Turbo C++编译器问世。1998年，C++标准委员会发布了C++的第一个国际标准——ISO/IEC 14882:1998。该标准即为广泛应用的C++98。随后C++进入了高速发展时期。

C++在C语言的基础上增加了面向对象以及泛型编程机制，大大提高了大中型程序开

发的效率。由于C++是C语言的扩展，因此C语言代码几乎可以不加修改地用于C++。如果不使用C++的特性，那么C++的执行效率和C语言几乎相同。

3.1.2 面向过程编程和面向对象编程

C++最早称为C with class。可见class(类)是C++的最重要特征。那么为什么要在C语言的基础上引入类？引入类又有什么优点？要回答这两个问题，就要从面向过程编程(Procedure Oriented Programming，POP)和面向对象编程(Object Oriented Programming，OOP)说起。

面向过程编程和面向对象编程是解决问题的两种思路。使用面向过程编程的思路解决问题时，会把任务拆分成一个个函数和数据(见图3.1)，然后按照一定的顺序执行函数。C语言就是一种典型的面向过程的语言。

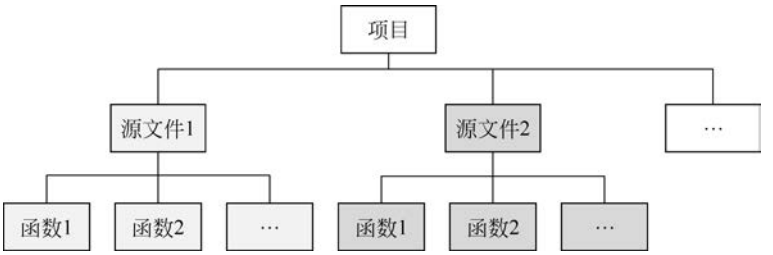

图3.1 面向过程编程的程序结构示意图

例如，要通过边长计算三角形的面积。如果使用面向过程的思路，会自然地将这个问题分解为数据和算法两部分。数据是三角形的3个边长(a、b、c)，算法则是海伦公式：

$$S=\sqrt{p(p-a)(p-b)(p-c)}$$

其中，

$$p=\frac{a+b+c}{2}$$

有了数据和算法，只要用代码将算法一步一步实现，问题就解决了。

面向对象编程的思路则不同。使用这一思路解决问题时会先把事物进行抽象，得到描述事物的信息(数据)和事物的行为(功能)并组成类。具体问题反而成了类的特例。按照面向对象的思路编写的程序结构示意图如图3.2所示。

还是以计算三角形面积的问题为例。既然三角形是二维图形的一种，那么是不是可以编写一个通用的类来计算各种二维图形的面积？计算时是否可以使用不同的算法？能否可以提供不同精度的结果？将这些内容放到一起，就可以组成一个类。当然，也可以只围绕三角形做文章，比如在计算面积的基础上增加计算周长、计算正弦和余弦(只针对直角三角形)的功能，或者计算外接圆、内切圆半径的功能等。当然，具体如何设计这个类需要根据实际需求进行。

如果说面向过程编程是使问题满足编程语言的语法要求，那么面向对象编程就是试图

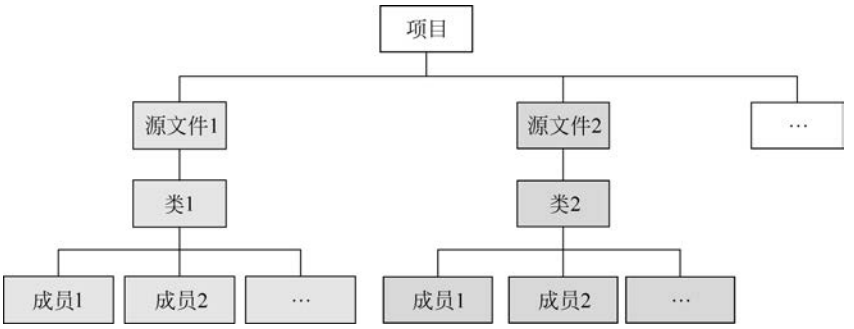

图 3.2　面向对象编程的程序结构示意图

让语言来满足问题的要求。面向对象编程本质是通过建立模型来体现抽象思维过程。因为模型不可能反映客观事物的一切具体特征，所以必须要对事物特征和变化规律进行抽象，得到一个能普遍、深刻地描述问题特征的模型，也就是类。通过这一过程，不仅加深了对问题的认识，还可以把原本分散开来的数据与功能整合到类中，使程序的扩展性得以提高。

3.1.3　面向对象编程的特征

面向对象编程有 4 个特征，即抽象、封装、继承、多态。由于本书并非专门的 C++教程，因此只简单介绍一下这 4 个特征的含义。

(1) 抽象，就是对同一类事物的共有的属性特征和功能行为进行提取、归纳、总结的过程。如汽车都有轮子、发动机等部件，这些是汽车的属性。汽车能前进后退、能载客载货，这些是汽车的功能。通过类似的抽象可以把汽车的属性与功能提取出来，供描述汽车的程序使用。

(2) 封装，就是隐藏对象的属性和实现细节，仅公开接口，从而控制程序对数据的访问。例如，在驾驶汽车时，驾驶人员只能接触到方向盘、油门、刹车等接口设备，而汽车复杂的机械原理则隐藏在接口之后。

(3) 继承，即从已有的类产生出新的类。新的类能接收已有类的数据和功能，并能扩展新的数据和功能。通过继承可以有效地提高代码的重用性。例如，在上述的汽车类的基础上，还可以派生出轿车、SUV、MPV 等不同车型的类。

(4) 多态，指同一种事物在不同的情况下有多种表现形式。例如，在同一个类中，可以有多个名称相同但参数不同的函数。程序运行时可以根据需要调用对应的函数。

3.2　Hello，C++！


视频讲解

3.2.1　一个简单的 C++程序

虽然 C++是一门面向对象的编程语言，但是 C 代码仍然可以在 C++环境下运行。作为第一个例子，不妨先看一个 C++风格的面向过程程序，其代码如下：

```
1   // 示例代码\ch3\ch3 - 1CPPDemo\main.cpp
2   #include < iostream >
3   namespace mySpace
4   {
5       int nNum = 321;
6   }
7   using namespace std;
8   int nNum = 0;
9   int main()
10  {
11      char cStr[] = "Please enter a number:";
12      cout << "Hello!" << endl << cStr;
13      cin >> nNum;
14      cout << "The number you entered is: " << nNum << endl;
15      cout << "The number in mySpace is: " << mySpace::nNum << endl;
16      return 0;
17  }
```

在这段代码中，第 1 行是 C++风格的注释。编译器会忽略从双斜杠（“//”）到行尾的所有内容。在 C++中，双斜杠注释和 C 风格的/*…*/注释均可使用。这段代码在本书配套资源中的路径和文件名为“配套代码\ch3\ch3-1CPPDemo\main.cpp”。

第 2 行为预处理内容。iostream 是头文件，定义了标准输入/输出流对象。该头文件使得 C++程序能够从键盘输入信息并通过屏幕上输出信息。关于输入/输出流的具体内容请见 3.2.2 节。

第 3～6 行定义了名称空间 mySpace 和属于 mySpace 的变量 nNum，并将 nNum 初始化为 321。关于名称空间的具体内容请见 3.2.3 节。

第 7 行指明了后续代码使用的默认名称空间为 std。

第 8 行（在默认名称空间 std 中）声明了全局变量 nNum。

第 9 行声明了主函数 main()，这与 C 语言相同。

第 11 行定义了字符串 cStr 并初始化。

第 12 行调用 cout 在屏幕上显示“Hello!”和字符串 cStr 的内容。cout 是 C++的标准输出流对象，endl 是 C++风格的换行符。

第 13 行调用 cin 读取用户输入的数字并保存到变量 nNum 中。cin 是 C++的标准输入流对象。

第 14 行和第 15 行输出了名称空间 mySpace 中的变量 nNum 和名称空间 std 中的变量 nNum。

第 16 行程序结束，返回 0。

要运行该程序，可以使用 Qt Creator、VS Code、Code::Blocks 等工具，也可以使用在线 C++编译器 OnlineGDB 等。如果使用 Qt Creator，可以直接打开该项目的 pro 文件并运行。下面是该程序的运行结果，其中符号↙代表按 Enter 键：

```
Hello!
Please enter a number:12↙
The number you entered is: 12
The number in mySpace is: 321
```

3.2.2　C++的基本输入/输出

1. 输入/输出流

C++将数据从一个对象到另一个对象的流动抽象为“流”。流在使用前要建立,使用后要删除。从流中获取数据的操作称为提取操作,向流中添加数据的操作称为插入操作。C++的头文件 iostream 负责实现输入/输出流。iostream 由输入流 istream、输出流 ostream、文件流 fstream、缓冲流 streambuf 等多个库组合而成的。开发者也可以单独引用 istream 或 ostream 等头文件,从而减小程序的大小。

2. cin 和 cout

在 C 语言中使用 scanf()和 printf()实现输入和输出。在 C++中,可以使用预定的流对象 cin 和 cout。cin 是一个输入流对象,用来处理标准输入(即从键盘读取的信息)。在使用 cin 时需要紧跟≫运算符。cout 是一个输出流对象,用来处理标准输出(即通过屏幕输出的信息)。在使用 cout 时需要紧跟≪运算符。这两个运算符可以自动分析数据类型,无须像 scanf()和 printf()一样给出格式控制字符串。在使用 cout 时,可以使用 C++风格的换行符 endl(即 end of line 的缩写)。endl 与 C 语言中的'\n'作用相同,在 C++中可以相互替代,例如:

```
// 示例代码\ch3\ch3-2cincout\main.cpp
#include <iostream>
using namespace std;
int main ()
{
    int nNum;
    float fNum;
    cout << "Please input an int number and a float number:" << endl;
    cin >> nNum >> fNum;
    cout << "The int number is " << nNum << '\n';
    cout << "The float number is " << fNum << endl;
    return 0;
}
```

运行该程序,结果如下:

```
Please input an int number and a float number:
8 7.4↙
The int number is 8
The float number is 7.4
```

运行程序后，用户首先要通过键盘输入两个数字（如 8 和 7.4），并按 Enter 键。这两个数字分别存储到 nNum、fNum 两个变量中。然后程序将这两个变量的值输出在了屏幕上。

3.2.3 名称空间

一个中大型软件往往由多名程序员共同开发。不同程序员使用的变量名或函数名难免会出现冲突。为了解决这一问题，C++引入了名称空间（Namespace，也称为命名空间）的概念。通过名称空间，开发者可以将自己的代码封装在私有的空间中，从而避免与其他代码出现冲突。

名称空间的关键字为 namespace。定义名称空间的方法为：

```
namespace spacename
{
   //定义变量、函数、类等
}
```

其中，spacename 是名称空间的名字，花括号表明了名称空间的范围。在花括号内部可以定义变量、函数、类，也可以进行 typedef、#define 等操作。C++默认的名称空间是 std。C++的标准函数或者对象都是在 std 中定义的，如 cin 和 cout。

例如，小赵与小钱各自定义了自己的名称空间，并在各自的名称空间里定义了变量 dSum（见示例代码\ch3\ch3-3Namespace）：

```
namespace Zhao
{
   double dSum = 1000;
}

namespace Qian
{
   double dSum = 100;
}
```

虽然这两个变量名称相同，但是分别处于不同的名称空间，所以可以共存。在使用时，可以通过域解析操作符::来指明变量（或函数、类等）的归属，例如，

```
Zhao::dSum = 123;          //使用小赵定义的变量 dSum
Qian::dSum = 456;          //使用小钱定义的变量 dSum
```

要声明默认使用的名称空间，可以使用 using namespace 关键字，例如，

```
using namespace Zhao;

int main()
{
   std::cout << dSum << " " << Qian::dSum << std::endl;
```

```
    dSum = 1001;
    Qian::dSum = 101;
    std::cout << dSum << " " << Qian::dSum << std::endl;

    return 0;
}
```

在这段代码中，首先声明了默认使用的名称空间 Zhao，然后在主函数中进行了变量输出和赋值操作。需要注意的是，cout、endl 均是在名称空间 std 中定义的。由于默认的名称空间改为了 Zhao，因此在使用 cout、endl 时需要加上 std::前缀。这段代码的运行结果为：

```
1000  100
1001  101
```

3.3　函数和 new 运算符


视频讲解

3.3.1　函数的默认参数

C++允许在定义函数时给形参指定默认的值。在调用函数时，如果没有给形参赋值，那么就使用默认值；如果给出了形参的值，则忽略默认值。

在下面的代码中，定义了用于输出两个数最大值的函数 printMax()。函数的两个参数分别具有默认值 2 和 3。printMax()函数在主函数中被重复调用。每次调用时形参的数量均不相同。

```
// 示例代码\ch3\ch3-4DefaultArguments\main.cpp
#include <iostream>
using namespace std;

void printMax(int nArg1 = 2, int nArg2 = 3)
{
    cout << "Max of " << nArg1 << " and " << nArg2 << " is " << (nArg1 > nArg2 ? nArg1 : nArg2) <<
endl;
}

int main()
{
    printMax(10, 6);          //形参 nArg1 = 10,nArg2 = 6
    printMax(5);              //形参 nArg1 = 5,nArg2 为默认值 3
    printMax();               //形参 nArg1 为默认值 2,nArg2 为默认值 3
    return 0;
}
```

程序的运行结果为：

```
Max of 10 and 6 is 10
Max of 5 and 3 is 5
Max of 2 and 3 is 5
```

C++规定带默认参数的形参只能放在形参列表的末尾。一旦为某个形参指定了默认值，那么后面的所有形参都必须有默认值。以下定义和调用函数的方法都是错误的：

```
void printMax(int nArg1 = 2, int nArg2);          //错误的定义方法
printMax( , 6);                                   //错误的调用方法
```

3.3.2 函数重载

在实际开发中，有时候需要实现几个功能类似的函数，例如，需要分别求两个 int 型变量、两个 float 型变量、两个 char 型变量的最大值。在 C 语言中需要编写不同的函数来实现这一功能，如：

```
int maxInt(int nArg1, int nArg2);                 //求两个 int 型变量的最大值
float maxFloat(float fArg1, float fArg2);         //求两个 float 型变量的最大值
char maxChar(char cArg1, char cArg2);             //求两个 char 型变量的最大值
```

但是在 C++中，借助函数重载(Function Overloading)功能可以编写多个函数名相同但参数不同的函数，例如：

```
int max(int nArg1, int nArg2)                     //求两个 int 型变量的最大值
float max(float fArg1, float fArg2);              //求两个 float 型变量的最大值
char max(char cArg1, char cArg2);                 //求两个 char 型变量的最大值
```

在调用 max()函数时，系统会根据参数的类型自动选择合适版本的 max()函数。例如：

```
// 示例代码\ch3\ch3 - 5FunctionOverloading\main.cpp
# include < iostream >
using namespace std;
int max(int nArg1, int nArg2)                     // int 型
{
   return nArg1 > nArg2 ? nArg1 : nArg2;
}

float max(float fArg1, float fArg2)               // float 型
{
   return fArg1 > fArg2 ? fArg1 : fArg2;
}

char max(char cArg1, char cArg2)                  // char 型
```

```
{
    return cArg1 > cArg2 ? cArg1 : cArg2;
}

int main()
{
    int n1 = 3, n2 = 4;
    cout << "Max of n1 and n2 is: " << max(n1, n2) << endl;
    float f1 = 2.72, f2 = 3.14;
    cout << "Max of f1 and f2 is: " << max(f1, f2) << endl;
    char c1 = 'A', c2 = 'B';
    cout << "Max of c1 and c2 is: " << max(c1, c2) << endl;
    return 0;
}
```

程序的运行结果为：

```
Max of n1 and n2 is: 4
Max of f1 and f2 is: 3.14
Max of c1 and c2 is: B
```

在实现函数重载时，函数的名称必须相同，但是函数的参数列表必须不同。这里的参数列表涉及参数的类型、参数的个数和参数的顺序。只要其中有一个不同就叫作参数列表不同。函数的返回值不能作为函数重载的判定依据。

3.3.3　new 和 delete 运算符

动态内存分配是编程中经常用到的方法。在 C 语言中，常使用 malloc()函数动态申请内存。当这部分内存使用完毕后，需要用 free()函数释放内存。例如：

```
int *p = (int*) malloc( sizeof(int) * 10 );        //申请内存
p[1] = 1;                                          //使用内存
free(p);                                           //释放内存
```

C++在 malloc()函数和 free()函数的基础上增加了 new 和 delete 两个运算符。new 用来动态申请内存，delete 用于释放 new 申请的内存，例如：

```
int *p2 = new int;              //申请 1 个 int 型的内存空间
int *p3 = new int[10];          //申请 10 个 int 型的内存空间
delete p2;                      //释放 1 个 int 型的内存空间
delete[] p3;                    //释放整个 int 型数组的内存空间
```

new 运算符会根据后面的数据类型来自动推断所需空间的大小，无须使用 sizeof()计算。通过 new 申请的内存必须调用 delete 手动释放，否则只能等到程序运行结束由操作系统回收。值得注意的是，当使用 new 运算符为类对象指针申请内存时，会自动调用类的构造函数(见 3.4.3 节)，但是使用 malloc()函数为类对象指针申请内存则不会调用构造函数。


视频讲解

3.4 类和对象

3.4.1 抽象、类和对象

1. 抽象

抽象的过程是对问题(也就是研究对象)进行分析和认识的过程,也是类和对象的基础。一般来说,对研究对象的抽象应该包括两个方面：数据抽象和行为抽象。前者描述研究对象的属性或状态,也就是研究对象区别于其他对象的特征；后者也叫作功能抽象、代码抽象,描述的是研究对象的共同行为或功能特征。

例如,如果要对日常生活中常用的时钟进行抽象,则需要 3 个 int 型变量来存储时、分和秒,这就是对时钟所具有的数据进行抽象。时钟要有显示时间、设置时间等功能,这就是对时钟的行为进行抽象。用变量和函数可以将抽象后的时钟属性描述如下：

```
变量 int nHour, int nMinute, int nSecond
函数 showTime(), setTime()
```

2. 类和对象

在学习 C 语言时我们了解了结构体(Struct)的知识。结构体是一种构造类型,可以包含若干不同类型的成员变量(也称为元素)。C++中的类也是一种构造类型,但是其功能远比结构体强大得多。类的成员不但可以是变量,还可以是函数,也可以是另外一个类。通过类定义出来的变量也有特定的称呼,叫作“对象”(Object)。

类规定了可以使用哪些数据来表示对象,以及可以对这些数据执行哪些操作。例如,上面分析了一个时钟具有的数据和功能。在这个基础上,就可以定义一个通用的时钟类。要实现时钟程序,只需要定义一个时钟类的对象。对象包含了时钟所有的数据和使用时钟所有的操作。如果需要在程序中实现多个时钟,则可以定义多个类对象。

3.4.2 定义类和类对象

在了解了类的含义以后,下面来学习如何用代码定义、实现一个类,以及如何定义、使用类对象。

1. 类的定义和实现

仍以上面的时钟类为例来讲解类的定义和实现。此处为时钟类额外增加了闹钟功能。

(1) 定义类。在定义类时,要在头文件(.h 文件)中按照 C++语法对类的成员变量和成员函数进行定义。这非常类似于变量声明和函数声明。定义类的通用格式为：

```
class 类名称
{
```

```
public:
   公有成员/接口
protected:
   受保护的成员
private:
   私有成员
};
```

上述代码第1行的class是C++的关键词，代表这段代码定义了一个类。这与C语言的结构体关键词struct非常类似。代码中的花括号{}限定了类的范围。public(公有的)、protected(受保护的)、private(私有的)都是C++的关键词，用于定义成员函数和成员变量的访问类型。一般也将它们称为访问权限限定符。这一部分内容将在访问控制部分讲解。类的每个成员变量和成员函数都要有自己的访问类型。例如，如果希望成员变量是私有的，那么只要将变量定义写在private下方即可。

假如将时钟类命名为ClassClock，那么可以按照上述格式在头文件ClassClock.h中完成时钟类的定义：

```
// 示例代码\ch3\ch3-6ClassClock\ClassClock.h
class ClassClock
{
private:
   int m_nHour;                                    //当前时间的小时部分
   int m_nMinute;                                  //当前时间的分钟部分
   int m_nSecond;                                  //当前时间的秒部分

public:
   int m_nHourAlarm;                               //闹钟时间的小时部分
   int m_nMinuteAlarm;                             //闹钟时间的分钟部分

   void setTime(int h, int m, int s);              //设置当前时间
   void setAlarm(int h, int m);                    //设置闹钟时间
   void getTime(int *h, int *m, int *s);           //读取当前时间
   void getAlarm(int *h, int *m);                  //读取闹钟时间
};
```

在上述代码中，当前时间(m_nHour、m_nMinute、m_nSecond)是私有成员，而闹钟时间(m_nHourAlarm、m_nMinuteAlarm)和所有的成员函数(setTime()、setAlarm()、getTime()、getAlarm())都是公有成员。代码中没有受保护的成员。

在类定义的过程中，private、protected、public的出现顺序没有要求，出现的次数也没有限制。为了使程序清晰，可以使每种访问限定符在类定义中只出现一次。如果没有显式地声明访问权限，则默认为private。例如，在上面ClassClock类的定义中，如果删去private一行，那么所有类成员的访问权限保持不变。

(2) 成员函数实现。头文件ClassClock.h用于保存类的定义，而同名的.cpp文件则保

存成员函数的具体代码。在本例中，时钟类的代码实现如下：

```
// 示例代码\ch3\ch3-6ClassClock\ClassClock.cpp
#include "ClassClock.h"

void ClassClock::setTime(int h, int m, int s)
{
    m_nHour = h;
    m_nMinute = m;
    m_nSecond = s;
}

void ClassClock::setAlarm(int h, int m)
{
    m_nHourAlarm = h;
    m_nMinuteAlarm = m;
}

void ClassClock::getTime(int *h, int *m, int *s)
{
    *h = m_nHour;
    *m = m_nMinute;
    *s = m_nSecond;
}

void ClassClock::getAlarm(int *h, int *m)
{
    *h = m_nHourAlarm;
    *m = m_nMinuteAlarm;
}
```

在 ClassClock.cpp 中，每个成员函数的函数名前面都需要增加前缀 ClassClock::，从而指明该函数所属的类。如果不指明所属的类，那么编译器会报告错误。

2. 类对象的创建和使用

类的使用和结构体的使用十分类似。可以通过以下两种方法创建 ClassClock 类的对象：

```
ClassClock clock1;                              //方法 1
ClassClock * clock2 = new ClassClock();         //方法 2
```

方法 1 直接创建了一个类对象 clock1；方法 2 则先创建了一个类对象指针 clock2，然后通过 new 运算符为该指针申请了内存。从内存管理的角度看，方法 1 是将变量存放在内存的栈中，方法 2 是将变量存放在内存的堆中。由于方法 2 使用了 new 运算符，因此在类对象完成任务后需要调用 delete 运算符释放内存，即

```
delete clock2;
```

对于类对象 clock1，可以使用“.”运算符（即英文句号）访问其成员，例如：

```
clock1.setTime(12, 34, 56);
clock1.m_nHourAlarm = 11;
```

对于类对象指针 clock2，可以使用“->”运算符访问其成员，例如：

```
clock2->setTime(12, 34, 56);
clock2->m_nHourAlarm = 11;
```

3. 类成员的访问控制

在面向对象编程中，一个核心原则就是数据和数据的操作相分离。在理想的情况下，数据是隐藏的（如汽车的发动机），而数据的接口是公开的（如汽车的油门、方向盘）。数据是无法直接访问的，但是可以通过数据接口来间接访问或者操作数据。类的访问控制机制可以有效地控制谁能访问类成员、谁不能访问类成员。

在 C++ 中，类的成员有 3 种访问权限，分别是 public（公有的）、protected（受保护的）、private（私有的）。所谓 public 权限，就是该成员可以被任何代码访问（类似于汽车内的所有人都可以接触方向盘）。private 指该成员只能被类中的代码访问，不能被外界直接访问（例如只能通过油门间接控制发动机的工作状态）。protected 权限主要用于类的继承和派生，具体在 3.5 节进行讲解。在不考虑类的继承和派生的情况下，可以简单地认为 protected 权限和 private 权限相同。

下面通过一个例子来体会类成员访问权限的作用。下面的代码中定义一个时钟类 ClassClock 的对象，并访问了类对象的各个成员变量和成员函数：

```
1   // 示例代码\ch3\ch3-6ClassClock\main.cpp
2   #include<iostream>
3   #include "ClassClock.h"          //引用类定义文件
4   using namespace std;
5   int main()
6   {
7      ClassClock myClock;           //定义类对象
8      int h, m, s;
9      myClock.setTime(12, 34, 56); //调用公有成员函数
10     myClock.m_nHourAlarm = 3;    //直接访问公有成员变量
11     myClock.m_nMinuteAlarm = 30;//直接访问公有成员变量
12     // myClock.m_nHour = 1;      //直接访问私有成员变量,会引发错误!
13     myClock.getTime(&h, &m, &s); //调用公有成员函数
14     cout << "Current time is " << h << ":" << m << ":" << s << endl;
15     cout << "Alarm time is " << myClock.m_nHourAlarm << ":" << myClock.m_nMinuteAlarm <<
       endl;                        //直接访问公有成员变量
16     return 0;
17  }
```

因为类的公有成员可以任意访问，所以可以在主函数中直接调用公有成员函数（见代码

第 9 行、第 13 行），也可以在主函数中直接操作公有成员变量（见代码第 10 行、第 11 行、第 15 行）。而类的私有成员是不能直接访问的，因此在主函数代码的第 12 行无法为私有成员变量 m_nHour 赋值。但是在类的内部，所有成员变量和成员函数都可以互相访问，不受权限限制。所以公有成员函数 setTime()可以访问类的私有成员变量来设置时间。

由于成员变量一般设置为私有，不能在类外部直接访问，所以常常设计一组公有的 set()函数和 get()函数来访问私有的成员变量。set()函数用于修改私有成员变量的值；get()函数用于读取、输出、打印私有成员变量的值。因为 set()和 get()函数可以为外界提供接口来访问类内部的成员，所以将它们统称为接口函数。使用接口函数体现了面向对象编程中的封装特性。

3.4.3 构造函数和析构函数

1. 构造函数

通过前面的学习可以看到，创建类对象和创建 int 型变量并没有本质上的区别。但是类是一种复杂的构造类型，内部包含了不同的成员。在创建 int 型变量时，可以为变量提供初始值。但是创建类对象时，成员变量的初始值是多少呢？要回答这一问题，就不得不提到类的构造函数（Constructor）。

构造函数是类的一种特殊的公有成员函数，会在创建类对象时自动执行，完成类对象初始化操作。构造函数的函数名和类名相同，但是没有返回值（返回值是 void 也不可以），也不能含有 return 语句。用户不能手动调用构造函数。在设计类的过程中，如果开发者没有显式地定义构造函数，那么编译器会自动生成一个默认的构造函数。默认构造函数的函数体是空的，也没有形参。下面仍以闹钟类 ClassClock 为例讲解，默认的构造函数为：

```
ClassClock()
{}
```

下面为闹钟类 ClassClock 显式地添加构造函数，并在这个构造函数中将成员变量初始化为 0。添加构造函数的具体步骤如下。

（1）在类定义中增加一个没有返回值的公有函数 ClassClock()，函数名与类名相同：

```
// 示例代码\ch3\ch3-7Constructor\ClassClock.h
public:
   ClassClock();
```

（2）在 cpp 文件中实现构造函数：

```
// 示例代码\ch3\ch3-7Constructor\ClassClock.cpp
ClassClock::ClassClock()
{
   m_nHour = 0;
   m_nMinute = 0;
```

```
    m_nSecond = 0;
    m_nHourAlarm = 0;
    m_nMinuteAlarm = 0;
}
```

这样在创建类对象时会自动地调用构造函数，将成员变量初始化为0。

构造函数是允许重载的。一个类可以有多个重载的构造函数。创建对象时，系统会根据传递的实参类型和数量来调用相应的构造函数。下面继续为ClassClock类增加一个重载的带参数的构造函数，从而在创建类对象的时候指定当前时间和闹钟时间。该构造函数的定义为：

```
// 示例代码\ch3\ch3-7Constructor\ClassClock.h
public:
    ClassClock(int h, int m, int s, int ha, int ma);
```

该构造函数的代码为：

```
// 示例代码\ch3\ch3-7Constructor\ClassClock.cpp
ClassClock::ClassClock(int h, int m, int s, int ha, int ma)
{
    m_nHour = h;
    m_nMinute = m;
    m_nSecond = s;
    m_nHourAlarm = ha;
    m_nMinuteAlarm = ma;
}
```

在创建类对象时，可以通过构造函数的参数决定调用哪个构造函数，例如，

```
// 示例代码\ch3\ch3-7Constructor\main.cpp
#include <iostream>
#include "ClassClock.h"
using namespace std;

int main()
{
    ClassClock clock1;                          //调用不带参数的构造函数
    ClassClock clock2(0, 0, 0, 3, 30);          //调用带参数的构造函数
    return 0;
}
```

构造函数的调用是强制性的。一旦在类中定义了构造函数，那么创建对象时就一定会调用它。如果有多个重载的构造函数，那么创建对象时提供的实参必须和其中的一个构造函数匹配。

2. 析构函数

创建类对象时系统会自动调用构造函数进行初始化工作，销毁对象时系统也会自动调

用一个函数来进行清理工作，如调用 delete 释放内存、关闭打开的文件等。这个负责清理工作的函数就是析构函数(Destructor)。

析构函数的函数名是在类名前面加一个波浪线“～”，如：～ClassClock()。析构函数没有返回值、没有参数，不能显式地调用，也不能重载。如果用户没有定义析构函数，那么编译器会自动生成一个默认的析构函数，例如，

```
～ClassClock()
{}
```

3.4.4 this 指针

一个类可以有多个类对象。这些类对象有着相同的成员变量和成员函数。对于一个特定的类对象而言，应该如何区分自己的成员和别的对象的成员？换言之，不同类对象之间的边界如何界定？要解决这一问题，就需要用到 this 指针。

this 是 C++中的关键字。this 指针是一个常(const)指针，指向当前对象本身。通过 this 指针可以访问当前对象的所有成员。所谓当前对象，也就是调用 this 指针的对象。使用 this 指针可以明确变量的归属，从而解决重名问题。

例如，在为 ClassClock 类编写构造函数时，一位初学者是这样编写的：

```
ClassClock(int m_nHour, int m_nMinute, int m_nSecond, int m_nHourAlarm, int m_nMinuteAlarm)
{
   m_nHour = m_nHour;
   m_nMinute = m_nMinute;
   m_nSecond = m_nSecond;
   m_nHourAlarm = m_nHourAlarm;
   m_nMinuteAlarm = m_nMinuteAlarm;
}
```

从这段代码看，该初学者的思路是正确的，只是出现了重名的变量。在 Visual Studio 和 Qt 中，这段代码可以通过编译，但是运行结果不正确。这是由于编译器无法区分哪个是形参、哪个是类成员变量。要解决这一问题，可以适当修改形参的名称，也可以使用 this 指针。例如，只要在类成员变量前面加上 this 指针作为限定，就能轻松地区分出变量的身份：

```
ClassClock(int m_nHour, int m_nMinute, int m_nSecond, int m_nHourAlarm, int m_nMinuteAlarm)
{
   this->m_nHour = m_nHour;
   this->m_nMinute = m_nMinute;
   this->m_nSecond = m_nSecond;
   this->m_nHourAlarm = m_nHourAlarm;
   this->m_nMinuteAlarm = m_nMinuteAlarm;
}
```

由于 this 指针指向对象本身，所以只有在创建对象后才会为 this 指针赋值。这个赋值

的过程是编译器自动完成的，不需要用户干预，用户也不能给 this 指针赋值。this 指针只能在类成员函数内部使用，但不能在 3.4.5 节即将讲到的静态成员函数中使用。

3.4.5 静态成员

1. 静态成员变量

同一个类的不同对象之间是相互隔离的，占用着不同的内存空间。如果需要在类的多个对象之间共享同一份数据（例如，统计类对象的个数），就可以使用类的静态成员变量。

静态成员变量是一种特殊的成员变量，不属于某个具体的对象，而是属于整个类。它在所有类对象之外开辟内存，所有的类对象都共用这份内存中的数据。即使从未创建过类对象，静态成员变量也是可以访问的。

要定义静态成员变量，需要使用关键字 static，例如，

```
// 示例代码\ch3\ch3-8StaticMembers\ClassClock.h
public:
   static int ms_nCount;
```

静态成员变量必须在类的外部初始化。没有初始化的静态成员变量不能使用。静态成员变量的初始化语法类似于全局变量的声明，即

```
type classname::name = value;
```

其中，type 是变量的类型，classname 是类名，name 是静态成员变量名，value 是初始值。下面是示例代码中静态成员的初始化代码。在这段代码中，静态成员变量 ms_nCount 的初始化位于主函数前：

```
// 示例代码\ch3\ch3-8StaticMembers\main.cpp
#include <iostream>
#include "ClassClock.h"
using namespace std;
int ClassClock::ms_nCount = 0;        //初始化静态成员变量
int main()
{
   //…
}
```

静态成员变量既可以通过类对象访问，也可以通过类名访问，例如，

```
clock1.ms_nCount = 2;                 //通过类对象访问静态成员变量
ClassClock::ms_nCount = 1;            //通过类名访问静态成员变量
```

需要注意的是，静态成员变量的访问权限可以是 private、protected 或 public。但是私有的静态成员变量不能通过类对象访问，也不能通过类名访问，只能通过静态成员函数访问。

2. 静态成员函数

static 除了可以声明静态成员变量，还可以声明静态成员函数。静态成员函数与普通成员函数的区别在于：普通成员函数可以访问类中的任意成员，但是静态成员函数只能访问静态成员（包括静态成员变量和静态成员函数），不能访问非静态成员。这是因为编译器在编译一个普通成员函数时，会隐式地增加一个形参 this，并把当前对象的地址赋值给 this。所以普通成员函数只能在创建对象后通过对象调用。但是静态成员函数可以通过类名直接调用，编译器不会增加形参 this。因为静态成员函数不需要当前对象的地址，所以不管有没有创建对象都可以调用。在 Qt 编程中经常会用到类的静态成员函数。

作为例子，下面为 ClassClock 类增加一个静态成员函数 getCount()，用于获取静态成员变量 ms_nCount 的值。具体步骤如下：

（1）在类定义中增加静态成员函数 getCount()。

```
// 示例代码\ch3\ch3 - 8StaticMembers\ClassClock.h
public:
   static int ms_nCount;
   static int getCount();           //静态成员函数
```

（2）在 cpp 文件中完成该函数。

```
// 示例代码\ch3\ch3 - 8StaticMembers\ClassClock.cpp
int ClassClock::getCount()
{
   return ms_nCount;
}
```

（3）在主函数中调用该函数。

```
1   // 示例代码\ch3\ch3 - 8StaticMembers\main.cpp
2   #include < iostream >
3   #include "ClassClock.h"
4   using namespace std;
5   int ClassClock:: ms_nCount = 11;                                    //初始化静态变量
6   int main()
7   {
8      ClassClock clock1, clock2;
9      clock1.ms_nCount = 11;                                           //通过类对象访问
10     cout << "clock1.ms_nCount: " << clock1.ms_nCount << endl;  //通过类对象访问
11     cout << "clock2.getCount(): " << clock1.getCount() << endl;//通过静态成员函数访问
12     ClassClock::ms_nCount = 22;                                      //通过类名访问
13     cout << "ClassClock:: ms_nCount: " << ClassClock::ms_nCount << endl;
                                                                        //通过类名访问
14     return 0;
15  }
```

在代码中，先定义了两个类对象（第 8 行），并通过类对象 clock1 访问静态成员变量（第

9行和第10行)。然后通过类对象clock2的静态成员函数访问该静态变量(第11行)。最后通过类名实现了静态成员变量的赋值和访问(第12行和第13行)。程序的运行结果为:

```
clock1.ms_nCount: 11
clock2.getCount(): 11
ClassClock::ms_nCount: 22
```

3.5　类的继承和派生


视频讲解

3.5.1　继承和派生的概念

在C++中,继承(Inheritance)可以理解为一个类从另一个类获取成员变量和成员函数的过程。例如,类B继承于类A,那么B就拥有A的成员变量和成员函数。A被B继承,所以A称为B的"父类"或"基类"。B继承了A,所以B称为A的"子类"或"派生类"。子类除了拥有父类的成员,还可以再增加自己的成员,从而增强类的功能。

派生(Derive)和继承是同一个概念的两个方面。如果说继承是儿子(子类)接受父亲(父类)的"产业"(成员变量和成员函数),那么派生就是父亲(父类)把"产业"(成员变量和成员函数)传承给儿子(子类)。所以"子类"和"父类"相对应,而"基类"和"派生类"相对应。

使用继承的典型场景有:

(1) 当新的类与现有的类相似,只是多出若干成员变量或成员函数时,可以使用继承。这样不但会减少代码量,而且派生类会拥有基类的所有功能。

(2) 当需要创建多个类,且这些类拥有很多相似的成员变量或成员函数时,可以将这些类的共同成员提取出来定义为基类,然后从基类继承。

3.5.2　类的3种继承方式

前面介绍了C++类成员的3种访问权限,即public、private、protected。在继承时,也有3种继承方法,即public(公有继承)、private(私有继承,默认的继承方式)、protected(受保护的继承)。虽然3种继承方法和3种访问权限都使用了相同的关键字,但是含义完全不同。3种访问权限控制着谁能够访问类成员,谁不能访问类成员,而3种继承方法则控制着基类成员在派生类中的存在状态。

类的3种继承方式和类成员的3种访问控制权限是相辅相成的。在类的继承过程中,继承方式和访问控制权限会相互影响,共同决定派生类中各个成员的访问权限。表3.1汇总了不同继承方式和不同访问权限的类成员相互作用的结果。例如,当采用private继承时,基类中public成员的访问权限会收紧到private。如果采用public继承,那么基类public成员的权限保持不变。类似地,当采用private继承时,基类private成员在派生类对象中不能直接访问,但是可以调用基类的公有函数来间接访问。

表 3.1 不同继承方式对不同属性的成员的影响

继 承 方 式	public 成员	protected 成员	private 成员
public 继承	public	protected	不可见
protected 继承	protected	protected	不可见
private 继承	private	private	不可见

通过表 3.1 可以总结出如下结论：

(1) 基类成员在派生类中的访问权限不高于继承方式指定的权限。例如，当继承方式为 protected 时，那么基类成员在派生类中的访问权限最高为 protected——高于 protected 的会降级为 protected，低于 protected 保持不变。

(2) 不管继承方式如何，基类中的 private 成员在派生类中始终不可见。需要注意的是，不可见不代表不存在。在派生类中，基类的 private 成员仍然能被继承下来，也会占用内存空间。

(3) 如果希望基类的成员能够被派生类继承并且使用，那么只能将基类成员声明为 public 或 protected。如果希望基类的成员不向外暴露(即不能通过派生类对象访问)，但又能在派生类中使用，则要将基类成员声明为 protected 权限。

3.5.3 继承和派生的实现

在 C++中，定义派生类的语法为：

```
class 派生类名 : 继承方式 基类名 1, 继承方式 基类名 2, …, 继承方式 基类名 n
{
   派生类成员声明
}
```

一个派生类只有一个基类的情况称为单继承。一个派生类有多个基类的情况称为多继承。单继承可以看作是多继承的特例，多继承可以看作是多个单继承的组合。

下面来看一个单继承的例子。假设将闹钟类 ClassClock 作为基类，希望派生出秒表类 ClassStopwatch。秒表类额外提供了开始计时 startTiming()、停止计时 stopTiming()、清除计时结果 clear()等成员函数，同时还增加了 m_nTiming(保存计时时长)、m_nRunningFlag(秒表运行状态)这两个成员变量。

要实现上述功能，首先要在闹钟类 ClassClock 的基础上新建头文件 ClassStopwatch.h，并在其中完成类的继承。此处采用公有继承，从而保持基类成员的权限不变：

```
// 示例代码\ch3\ch3-9Inheritance\ClassStopwatch.h
#include "ClassClock.h"
class ClassStopwatch : public ClassClock
{
private:
```

```
    int m_nTiming;                  //计时时间
    int m_nRunningFlag;             //秒表运行状态

public:
    void startTiming();             //开始计时
    void stopTiming();              //停止计时
    void clear();                   //归零
};
```

同时还要在 ClassStopwatch.cpp 中实现秒表类的成员函数：

```
// 示例代码\ch3\ch3-9Inheritance\ClassStopwatch.cpp
#include "ClassStopwatch.h"
void ClassStopwatch::startTiming()
{
    m_nRunningFlag = 1;
}

void ClassStopwatch::stopTiming()
{
    m_nRunningFlag = 0;
}

void ClassStopwatch::clear()
{
    m_nTiming = 0;
}
```

3.5.4　派生类的使用

派生类的使用方法与非派生类的使用方法相同，都是引用头文件并定义类对象，然后对类对象进行操作，例如：

```
1   // 示例代码\ch3\ch3-9Inheritance\main.cpp
2   #include <iostream>
3   #include "ClassStopwatch.h"                    //引用头文件
4   using namespace std;
5   int ClassClock::ms_nCount = 0;                 //初始化基类的静态成员变量
6   int main()
7   {
8       ClassStopwatch myWatch;                    //定义派生类对象
9       myWatch.setTime(12, 0, 0);                 //调用基类的公有成员函数
10      myWatch.startTiming();                     //调用派生类的公有成员函数
11      ClassClock::ms_nCount = 2;                 //通过基类名访问基类静态成员变量
12      cout << "myWatch.ms_nCount:" << myWatch.ms_nCount << endl;
                                                   //访问基类静态成员变量
13      ClassStopwatch::ms_nCount = 5;             //通过派生类名访问基类静态成员变量
```

```
14      cout << "myWatch.getCount():" << myWatch.getCount() << endl; //访问基类静态成员函数
15      return 0;
16  }
```

程序的运行结果为：

```
myWatch.ms_nCount:2
myWatch.getCount():5
```

从这个例子可以看到，基类的公有成员函数在经过公有继承以后，仍可以通过子类访问，并且使用起来与子类的成员函数没有任何差异。但是如果将继承方式改为私有继承，则基类所有的私有成员都不能访问了。在这种情况下，代码的第 9 行、第 12～14 行都无法运行。

3.6 本章小结

本章根据后续章节的需要介绍了 C++相对于 C 的一些新功能和新特性，方便没有 C++基础的读者学习后面的内容。受篇幅限制，本章只讨论了 C++的基础内容。在学习完本章的内容后，可以继续学习 C++的其他功能和特性，如运算符重载、友元、引用、异常、Lambda 表达式等，也可以将本章介绍的 C++和自己熟悉的编程语言进行对比，找出异同点，思考背后的原因。