第5章

类与对象

◇ **引言**

面向对象的程序设计方法就是运用面向对象的观点对现实世界中的各种问题进行抽象，然后用计算机程序来描述并解决该问题，这种描述和处理是通过类与对象实现的。类与对象是C++程序设计中最重要的概念，在C++中如何描述类与对象？如何通过建立类与对象解决现实世界的问题？这些将是本章讨论的内容。

◇ **学习目标**

(1) 掌握类的定义，会根据需求设计类；
(2) 会根据类创建各种对象；
(3) 掌握对象的各种成员的使用方法；
(4) 会设计构造函数与拷贝构造函数初始化对象，理解其调用过程与顺序；
(5) 理解浅拷贝与深拷贝的概念；
(6) 掌握动态对象以及动态对象数组的建立与释放；
(7) 理解类的静态成员的概念；
(8) 理解友元函数与友元类的概念；
(9) 掌握常对象与常成员的使用；
(10) 了解对象在内存中的分布情况。

5.1 类与对象的概念

5.1.1 从面向过程到面向对象

在面向过程的结构化程序设计中，程序模块由函数构成，函数将进行数据处理的语句放在函数体内，完成特定的功能，数据则通过函数参数传递进入函数体。在前面的章节中使用C++编写的实际上是面向过程的程序。在面向对象的程序设计中，程序模块是由类构成的。类是对逻辑上相关的函数与数据的封装，它是对问题的抽象描述。为了对面向对象的程序设计方法有一个初步的认识，我们先举一个例子。

【例5-1】 模拟时钟。

分析：不管什么样的时钟，也不管各种时钟是如何运行的，它们都具有时、分、秒3个属性。除了运行、显示时间的基本功能外，还有设置(调整)时间、设置闹钟等功能。将时钟的这些属性和功能抽象出来，分别给出面向过程的程序与面向对象的程序实现对时钟的模拟。

	时钟程序 A	时钟程序 B
1	/**********	/**********
2	* 程序名：p5_1_a.cpp *	* 程序名：p5_1_b.cpp *
3	* 功 能：面向过程的时钟程序 *	* 功 能：面向对象的时钟程序 *
4	**********/	**********/
5	#include <iostream>	#include <iostream>
6	using namespace std;	using namespace std;
7	***struct Clock*** {	***class Clock*** {
8	int H,M,S;	private:
9	};	int H,M,S;
10	Clock MyClock;	public:
11	**void SetTime(int H,int M,int S)**	**void SetTime(int h,int m,int s)**
12	{	{
13	MyClock.H=(H>=0&&H<24)?H:0;	H=(h>=0&&h<24)?h:0;
14	MyClock.M=(M>=0&&M<60)?M:0;	M=(m>=0&&m<60)?m:0;
15	MyClock.S=(S>=0&&S<60)?S:0;	S=(s>=0&&s<60)?s:0;
16	}	}
17	**void ShowTime()**	**void ShowTime()**
18	{	{
19	cout<<MyClock.H<<":";	cout<<H<<":"<<M<<":"<<S<<endl;
20	cout<<MyClock.M<<":";	}
21	cout<<MyClock.S<<endl;	};
22	}	**int main()**
23	**int main()**	{ Clock MyClock;
24	{ ShowTime();	MyClock.ShowTime();
25	SetTime(8,30,30);	MyClock.SetTime(8,30,30);
26	ShowTime();	MyClock.ShowTime();
27	return 0;	return 0;
28	}	}

运行结果：

0:0:0	-85893460:-85893460:-85893460
8:30:30	8:30:30

程序解释：

通过对上述两种方案的程序进行简单的观察，可以发现它们存在下面几点不同：

（1）在程序 A 中，时钟数据用一个结构型的变量存储，对时钟数据的存取通过函数实现。由于存储时钟数据的是一个全局变量，在任何地方都可见，可以不通过函数单独存取时钟数据。在程序 B 中，只能通过类提供的函数操作时钟。

（2）在程序 A 中，数据与对数据的操作相互独立，数据作为参数传递给函数。在程序 B

中，数据与对数据的操作构成一个整体。

(3) 程序A与程序B运行的初始结果不同，这是因为在程序A中，变量是全局的；在程序B中，对象(变量)MyClock是函数main()中的局部对象。全局变量与局部变量在没有初始化时取初值方式不同，这样造成了运行结果不同。将第23行移出main()外，使之变成全局对象后，两程序的结果完全相同。

在程序B中发现，一个以class开头的类似结构体的结构将时钟的数据与对数据进行处理的函数包括在一起，这就是用C++实现的类。

5.1.2　类的定义

简单地讲，类是一个包含函数的结构体，因此类的定义与结构类型的定义相似，其格式如下：

```
class 类名
    {
      public:
            公有数据成员或公有函数成员的定义;
      protected:
            保护数据成员或保护函数成员的定义;
      private:
            私有数据成员或私有函数成员的定义;
};
```

其中：

- 关键字class表明定义的是一个类。
- 类名是类的名称，应该是一个合法的标识符。
- public、protected、private为**存取控制属性**(访问权限)，用来控制对类的成员的存取，如果前面没有标明访问权限，默认访问权限为private。
- 类的成员有数据成员和函数成员两类，类的数据成员和函数成员统称为类的成员，类的**数据成员**一般用来描述该类对象的属性，称为**属性**；函数成员用来描述类行为，称为方法。函数成员由函数构成，这些作为类成员的函数因此也称为**成员函数**。

☆**注意**：

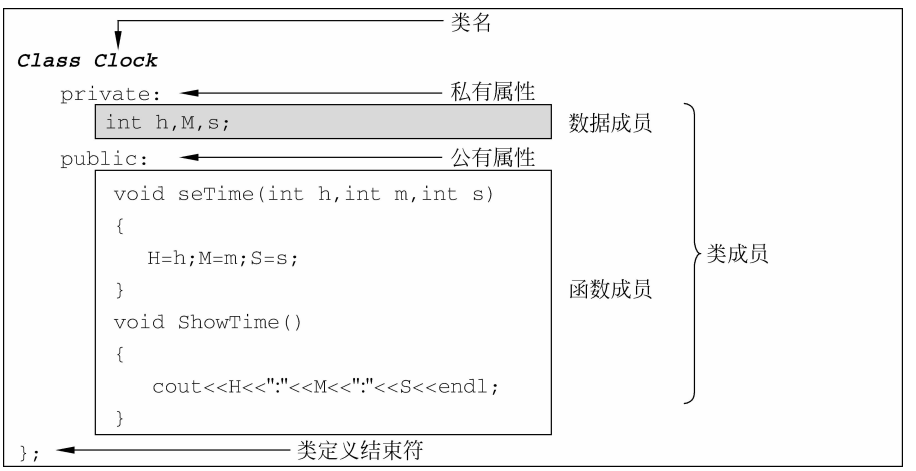
(1) 在C++中，class和struct都可以有成员函数，包括各类构造函数、析构函数、成员函数，并且都可以有public、private、protected修饰符。

(2) class的成员默认是private权限，struct默认是public权限。

(3) 从编程习惯的角度而言，C++的struct中只包含数据成员不包含成员函数，是一种用户自定义的数据结构，体现的是面向过程的编程思想，而class既包含数据成员也包含函数成员，是数据和对数据的操作行为的封装，体现的是面向对象的编程思想。

(4) C语言中没有class。

例如，例5-1中定义了一个时钟类Clock。



1. 数据成员

类定义中的数据成员描述了类对象所包含的数据类型，数据成员的类型可以是 C++基本数据类型，也可以是构造数据类型。

例①：

```
struct Record
{
    char name[20];
    int score;
};
class Team {
    private:
        int num;                          //基本数据类型
        Record * p;                       //构造数据类型
};
```

定义的类 Team 既包含基本数据类型，也包含构造数据类型。

☆**注意**：

（1）因为类只是一种类型，类中的数据成员不占内存空间，因此在定义数据成员时不能给数据成员赋初值。

（2）类的数据成员除了可以使用前面讲述的 C++类型外，还可以使用已定义的类类型。

（3）在正在定义的类中，由于该类型没有定义完，所以不能定义该类类型的变量，只能定义该类类型的指针成员以及该类类型的引用成员。

例②：

```
class Team;                               //已定义的类
class Grade {
    Team a;                               //使用了已定义的类类型
    Grade * p;                            //使用正在定义的类类型定义指针成员
    Grade &r;                             //使用正在定义的类类型定义引用成员
    Grade b;                              //错误! 使用了未定义完的类 Record 定义变量
};
```

2. 成员函数

作为类成员的成员函数描述了对类中的数据成员实施的操作。成员函数的定义、声明格

式与非成员函数(全局函数)的格式相同。成员函数可以放在类中定义,也可以放在类外定义,放在类中定义的成员函数为内联(inline)函数。Clock 类中的成员函数就是放在类中定义的。

C++可以在类中声明成员函数的原型,在类外定义函数体。这样做的好处是相当于在类中列了一个函数功能表,使我们对类的成员函数的功能一目了然,避免了在各个函数实现的一大堆代码中查找函数的定义。在类中声明函数原型的方法与一般函数原型的声明一样,在类外定义函数体的格式如下:

```
返回值类型 类名 :: 成员函数名(形参表)
{
    函数体;
}
```

其中,::是类的作用域分辨符,用在此处,放在类名后、成员函数前,表明后面的成员函数属于前面的那个类。

Clock 类中的成员函数可以在类中声明:

```
class Clock {
        private:
            int H,M,S;
        public:
          void SetTime(int h,int m,int s);     //声明成员函数
          void ShowTime();                     //声明成员函数
      };
```

在类外实现成员函数如下:

```
void Clock::SetTime(int h,int m,int s)
      {
        H = h,M = m,S = s;
       }
      void Clock::ShowTime()
      {
       cout << H <<":"<< M <<":"<< S << endl;
      }
```

如果要将在类外定义的成员函数编译成内联函数,可以在类外定义函数时,在函数的返回类型前加上 inline。下面将 ShowTime()定义成内联函数,与在类中定义成员函数的效果相同。

```
inline void Clock::ShowTime()
      {
        cout << H <<":"<< M <<":"<< S << endl;
       }
```

☆**注意**:

在成员函数中不仅可以自由使用类的成员,还可以使用该类定义变量(对象),通过变量使用成员。其原因是函数在调用时才在栈内存中建立函数体中的变量(包括实参),这时类已经定义完毕,当然可以使用已定义完的类类型的变量。

例如,在 Clock 类中可以定义成员函数 AddTime():

```
class Clock {
      private:
          int H,M,S;
      public:
          Clock AddTime(Clock C2){          //形参为 Clock 类型的变量
             Clock T;                       //在函数体中定义了 Clock 类型的变量
             …
             return T;                      //返回类型为 Clock 类型
          }
};
```

3. 类作用域

类是一组数据成员和函数成员的集合，类作用域作用于类中定义的特定的成员，包括数据成员与成员函数，类中的每一个成员都具有**类作用域**。实际上，类的封装作用也就是限制类的成员的访问范围局限于类的作用域之内。

5.1.3 对象的建立与使用

类相当于一种包含函数的自定义数据类型，它不占内存，是一个抽象的“虚”体，使用已定义的类建立对象就像用数据类型定义变量一样。对象在建立后，对象占据内存，变成了一个“实”体，类与对象的关系就像数据类型与变量的关系一样。其实，一个变量就是一个简单的不含成员函数的数据对象。

建立对象的格式如下：

```
类名  对象名;
```

其中，对象名可以是简单的标识符，也可以是数组。

在例 5-1 中，使用

```
Clock MyClock;
```

建立了一个 Clock 型的对象 MyClock。

在建立对象后，就可以通过对象存取对象中的数据成员，调用成员函数。其存取格式如下：

```
对象名.属性
对象名.成员函数名(实参 1,实参 2,…,)
```

例如，通过对象 MyClock 使用成员函数 SetTime()的方式如下：

```
MyClock.SetTime(8,30,30);
```

至于对数据成员 H、M、S 的存取，因其存取权限为 private 而被保护，所以不能直接进行存取。

值得注意的是，为了节省内存，编译器在创建对象时只为各对象分配用于保存各对象数据成员初始化的值，并不为各对象的成员函数分配单独的内存空间，而是共享类的成员函数定义，即类中成员函数的定义为该类的所有对象所共享，这是 C++编译器创建对象的一种方法。

在实际应用中，我们仍要将对象理解为由数据成员和函数成员两部分组成。

5.1.4　成员的存取控制

通过设置成员的存取控制属性使对类成员的存取得到控制，从而达到了隐藏信息的目的。C++的存取控制属性有公有类型(public)、私有类型(private)和保护类型(protected)，它们的意义如表5-1所示。

表5-1　存取控制属性表

存取属性	意　义	可存取对象
public	公开(公有)级	该类成员以及所有对象
protected	保护级	该类及其子类成员
private	私有级	该类的成员

类中定义为public等级的成员可以被该类的任何对象存取，适用于完全公开的数据。而private等级的成员只可被类内的成员存取，适用于不公开的数据。至于protected等级，属于半公开性质的数据，定义为protected等级的成员可以被该类及其子类存取。关于子类的概念，将在以后的章节中讲述。

在Clock类中，H、M、S的存取控制属性为private。这样，这些数据不能在类外存取而被保护，下列存取方法是错误的：

```
MyClock.M = 30;
```

而成员函数SetTime()、ShowTime()的存取控制属性为public，因此在类外可以通过对象存取。

由于private成员被隐藏起来，不能直接在类外存取，为了取得这些被隐藏的数据，通常在类内定义一个public的成员函数，通过该成员函数存取private成员，而public的成员函数又能在类外被调用。这样通过调用public型的成员函数，可以间接存取private成员。这样的函数起到了为private成员提供供外界访问的接口作用。类Clock中的成员函数SetTime()、ShowTime()就是存取private数据成员H、M、S的接口。

通过接口访问类的数据成员，一方面有效地保护了数据成员，另一方面又保证了数据的合理性。在程序p5_1_b.cpp中通过SetTime()设置一个合理的时钟值，但在p5_1_a.cpp中可以不通过SetTime()单独设置H、M、S的值，将H设置为70，这样的数据是不合理的。

☆**注意**：

(1) 即使是具有public存取控制属性的成员，在类外对其存取还必须通过该类的对象进行。对于Clock中的public成员，下列存取方法是错误的：

```
ShowTime();                          //错误,未通过对象
Clock::ShowTime()                    //错误,未通过对象
```

(2) 在类内声明在类外定义的成员函数，其函数体属于类内，因此在函数体中可以存取类的任意存取控制属性的成员。

(3) 成员函数还可以通过对象存取形参中与函数体中本类对象的所有成员。例如Clock类中的成员函数AddTime()通过形参对象C2、函数体中的对象T存取了访问控制属性为private的数据成员H、M、S：

```
Clock AddTime(Clock C2){
        Clock T;
        T.S = (S + C2.S) % 60;                    //通过对象 C2、T 存取了 private 数据成员 S
        ...
        return T;
}
```

5.2 构造函数与析构函数

在定义一个对象的同时，希望能给它的数据成员赋初值——对象的初始化。在特定对象使用结束时，还经常需要进行一些清理工作。C++程序中的初始化和清理工作分别由两个特殊的成员函数完成，它们就是构造函数和析构函数。

5.2.1 构造函数

首先看一个简单变量的建立过程：每一个变量在程序运行时都要占据一定的内存空间，如果在定义一个变量时对变量进行初始化，那么在为该变量分配内存空间的同时就向新分配的内存单元写入了变量的初始值。

例如：

```
int i(2);
int *p = &i;
double a[] = {1.0,3.4,5.7,8.0};
```

对象的建立过程也是类似的：在程序载入执行的过程中，当遇到对象定义语句时，系统会申请一定的内存空间用于存放新建的对象。如果在定义对象的同时允许指定数据成员的初始值，我们希望程序能像对待普通变量那样在分配内存空间后立即将指定的初始值写入。

在例 5-1 的程序 B 中，因为对象 MyClock 在建立时没有被初始化，所以最初显示的时钟是杂乱的数字。现在，我们试图在建立 Clock 类对象 MyClock 时对其初始化：

```
Clock MyClock(0,0,0);
```

但是很不幸，这将引起编译时的语法错误。对于类对象来说，如此初始化是不行的，因为与普通变量相比，类的对象比较复杂，而且由于类的封装性，它不允许在类的非成员函数中直接访问类对象的私有和保护数据成员。这样，对类对象数据成员的初始化工作自然就落到了类对象的成员函数身上，因为它们可以访问类对象的私有和保护数据成员。C++为用户提供了专门用于对象初始化的函数——**构造函数**。

构造函数(**constructor**)是与类名相同的在建立对象时自动调用的函数。如果在定义类时没有为类定义构造函数，编译系统会生成一个**默认形式**的**隐含**的构造函数，这个构造函数的函数体是空的，因此默认构造函数不具备任何功能。

如果用户至少为类定义了一个构造函数，C++就不会生成任何默认的构造函数，而是根据对象的参数类型和个数从用户定义的构造函数中选择最合适的构造函数完成对该对象的初始化。

作为类的成员函数，构造函数可以直接访问类的所有数据成员，可以是内联函数，可以不带任何参数，可以带有参数表以及默认形参值，还可以重载，用户可以根据不同问题的具体需

要有针对性地设计合适的构造函数将对象初始化为特定的状态。

构造函数是一种特殊的函数，主要用来在创建对象时初始化对象，即为对象的数据成员赋初始值。在需要为对象数据成员动态地分配内存时，构造函数总是与 new 运算符一起用在创建对象的语句中。一个类可以有多个构造函数，用户可根据其参数个数的不同或参数类型的不同来区分它们，即构造函数的重载。

构造函数与其他成员函数的区别如下：

(1) 构造函数的命名必须和类名完全相同，而一般成员函数不能和类名相同。

(2) 构造函数的功能主要用于在创建类的对象时定义初始化的状态，它没有返回值，也不能用 void 修饰，这就保证了它什么也不用返回，而其他成员函数可以有返回值，如果没有返回值，则必须用 void 予以说明。

(3) 构造函数不能被直接调用，必须在创建对象时由编译器自动调用，一般成员函数在程序执行到它的时候被调用。

(4) 在定义一个类的时候，如果用户没有定义构造函数，编译器会提供一个默认的构造函数，而成员函数不存在这一特点。

例如，在例 5-1 程序的 p5-1_b.cpp 中的 Clock 类中添加带有默认形参值的构造函数：

```
Clock(int h = 0, int m = 0, int s = 0)
{
      H = (h >= 0&&h < 24)?h:0;
      M = (m >= 0&&m < 60)?m:0;
      S = (s >= 0&&s < 60)?s:0;
}
```

执行：

```
Clock MyClock;
MyClock.ShowTime();
```

显示结果为：

```
0:0:0
```

这是因为建立对象时调用了 Clock()，各个形参被设成了默认值。

当执行：

```
Clock MyClock(9,30,45);
MyClock.ShowTime();
```

显示结果为：

```
9:30:45
```

这是因为建立对象时调用了 Clock(9,30,45)。

利用构造函数初始化新建立对象的方式还有：

```
Clock MyClock = Clock(9,30,45);
```

Clock(9,30,45)可看作是一个 Clock 类型的常量。

构造函数是类的一个成员函数，除了具有一般成员函数的特征外，还具有以下特殊性质：

(1) 构造函数的函数名必须与定义它的类同名。

（2）构造函数没有返回值，如果在构造函数前加 void 是错误的。

（3）构造函数被声明定义为公有函数。

（4）构造函数在建立对象时由系统自动调用。

☆**注意**：

由于构造函数可以重载，可以定义多个构造函数，在建立对象时根据参数来调用相应的构造函数。如果相应的构造函数没有定义，则出错。例如，若将例 5-1 中的构造函数定义成不带默认形参值的构造函数：

```
Clock(int h, int m,int s)
{
    H = (h >= 0&&h < 24)?h:0;
    M = (m >= 0&&m < 60)?m:0;
    S = (s >= 0&&s < 60)?s:0;
}
```

在定义对象 Clock MyClock 时调用 Clock()，而 Clock 类没有 Clock()函数，因此出错。

5.2.2 析构函数

自然界万物都是有生有灭，程序中的对象也是一样。对象在定义时诞生，不同生存期的对象在不同的时期消失。在对象要消失时，通常有一些善后工作需要做，例如构造对象时，通过构造函数动态申请了一些内存单元，在对象消失之前就要释放这些内存单元。C++用什么来保证这些善后清除工作的执行呢？答案是**析构函数**。

析构函数(**destructor**)也译作拆构函数，是在对象消失之前的瞬间自动调用的函数，其形式如下：

```
~构造函数名();
```

析构函数与构造函数的作用几乎相反，相当于“逆构造函数”。析构函数也是类的一个特殊的公有函数成员，它具有以下特点：

（1）析构函数没有任何参数，不能被重载，但可以是虚函数，一个类只有一个析构函数。

（2）析构函数没有返回值。

（3）析构函数名在类名前加上一个逻辑非运算符“～”，以与构造函数相区别。

（4）析构函数一般由用户自己定义，在对象消失时由系统自动调用，如果用户没有定义析构函数，系统将自动生成一个不做任何事的默认析构函数。

☆**注意**：

对象消失时的清理工作并不是由析构函数完成，而是靠用户在析构函数中添加清理语句完成。

【例 5-2】 构造函数与析构函数。

为 Clock 类重新定义一个析构函数如下：

```
1  /*************************************
2   *   程序名: p5_2.cpp                  *
3   *   功   能: 构造函数与析构函数       *
4   *************************************/
5  #include <iostream>
```

```
6   using namespace std;
7   class Clock {
8       private:
9         int H,M,S;
10      public:
11      Clock(int h = 0,int m = 0,int s = 0)
12      {
13          H = h,M = m,S = s;
14          cout <<"constructor:"<< H <<":"<< M <<":"<< S << endl;
15      }
16      ~Clock()
17      {
18          cout <<"destructor:"<< H <<":"<< M <<":"<< S << endl;
19      }
20  };
21  int main()
22  {
23      Clock C3(10,0,0);
24      Clock C4(11,0,0);
25  return 0;
26  }
27  Clock C1(8,0,0);
28  Clock C2(9,0,0);
```

运行结果：

```
constructor:8:0:0
constructor:9:0:0
constructor:10:0:0
constructor:11:0:0
destructor:11:0:0
destructor:10:0:0
destructor:9:0:0
destructor:8:0:0
```

程序解释：

(1) 从运行结果可以看出，构造函数执行的顺序为 C1::Clock()→C2::Clock()→C3::Clock()→C4::Clock()，由此看出对象建立的顺序为 C1→C2→C3→C4。即先建立全局对象，再建立局部对象。与普通变量建立的顺序相同。

(2) 析构函数调用的顺序为 C4::～Clock()→C3::～Clock()，由此看出对象消失的顺序为 C4→C3。这是因为局部对象在栈中建立，因此消失的顺序与建立的顺序相反。

(3) 对象 C1、C2 在什么时候消失呢？因为 C1、C2 是全局对象，像全局变量一样，在程序结束时消失，析构函数在程序结束时调用。

例 5-2 只是演示了构造函数与析构函数调用的顺序，并没有在析构函数中加入必要的功能。

【例 5-3】 字符串类与对象。

分析：字符串通常用字符数组来表示，C++提供了对字符数组操作的函数；同时，对字符

数组又可以直接进行操作，这样对字符数组的操作显得混乱。如果将一个字符串设计成一个类，将对字符数组的操作封装在类中，通过使用类对象的方法对字符串进行操作则显得直观且不易出错。C++标准库中提供了字符串类 string，为了与之相区别，将定义的字符串类名取为 String。

```
/**********************************
*   程序名：p5_3.cpp                *
*   功   能：基本的字符串类         *
**********************************/
#include <iostream>
using namespace std;
class String {
    private:
      char *Str;
      int len;
    public:
    void ShowStr()
    {

        cout <<"string:"<< Str <<",length:"<< len << endl;
    }
    String()
    {
        len = 0;
        Str = NULL;
    }
    String(const char *p)
    {
        len = strlen(p);
        Str = new char[len + 1];
        strcpy(Str,p);
    }
    ~String()
    {
        if (Str!= NULL)
        {
            delete [] Str;
            Str = NULL;
        }

    }
};
int main()
{   char s[] = "ABCDE";
    String s1(s);
    String s2("123456");
    s1.ShowStr();
    s2.ShowStr();
    return 0;
}
```

运行结果：

```
string:ABCDE, length:5
string:123456, length:6
```

程序解释：

(1) 第 16～26 行定义了两个构造函数 String()、String(char * *p*)。String()在建立空串时调用，String(char * *p*)在建立字符串时使用已有的字符数组做初值时调用。

(2) 由于建立字符串时，构造函数在堆里申请了空间，将字符串保存。因此，当字符串对象消失时，必须释放堆空间。所以，在析构函数中要加上释放堆空间的语句，以释放空间。程序的第 27～35 行为析构函数。

5.2.3　拷贝构造函数

拷贝构造函数是一种特殊的构造函数，C++提供的复制构造函数用于在建立新对象时将已存在对象的数据成员值复制给新对象，即用一个已存在的对象初始化一个新建立的对象。拷贝构造函数与类名相同，其形参是本类的对象的引用。

类的拷贝构造函数一般由用户定义，如果用户没有定义构造函数，系统就会自动生成一个默认函数来进行对象之间的位拷贝(bitwise copy)，这个默认拷贝构造函数的功能是把初始值对象的每个数据成员的值依次复制到新建立的对象中。因此，也可以说是完成了同类对象的克隆(clone)，这样得到的对象和原对象具有完全相同的数据成员，即完全相同的属性。事实上，拷贝构造函数是由普通构造函数和赋值操作符共同实现的。

用户也可以根据实际问题的需要定义特定的拷贝构造函数来改变默认拷贝构造函数的行为，以实现同类对象之间数据成员的传递。如果用户自定义了拷贝构造函数，则在用一个类的对象初始化该类的另外一个对象时将自动调用自定义的拷贝构造函数。

定义一个拷贝构造函数的一般形式如下：

```
类名(类名 & 对象名)
  {
    …
  }
```

拷贝构造函数在用类的一个对象初始化该类的另一个对象时调用，以下 3 种情况相当于用一个已存在的对象初始化新建立的对象，此时调用拷贝构造函数：

(1) 当用类的一个对象初始化该类的另一个对象时。

(2) 如果函数的形参是类的对象，在调用函数将对象作为函数实参传递给函数的形参时。

(3) 如果函数的返回值是类的对象，函数执行完成时将返回值返回。

【例 5-4】 带拷贝构造函数的时钟类。

```
1  /**********************************
2   *  程序名: p5_4.cpp                *
3   *  功  能: 构造拷贝构造函数        *
4   **********************************/
```

```
#include <iostream>
using namespace std;
class Clock {
    private:
        int H,M,S;
    public:
    Clock(int h = 0,int m = 0,int s = 0)
    {
            H = h,M = m,S = s;
            cout <<"constructor:"<< H <<":"<< M <<":"<< S << endl;

    }
    ~Clock()
    {
        cout <<"destructor:"<< H <<":"<< M <<":"<< S << endl;
    }
    Clock(Clock & p)
    {
    cout <<"copy constructor,before call:"<< H <<":"<< M <<":"<< S << endl;
        H = p.H;
        M = p.M;
        S = p.S;
    }
    void ShowTime()
    {
    cout << H <<":"<< M <<":"<< S << endl;
    }
};
Clock fun(Clock C)
{
    return C;
}
int main( )
{
    Clock C1(8,0,0);
    Clock C2(9,0,0);
    Clock C3(C1);
    fun(C2);
    Clock C4;
    C4 = C2;
    return 0;
}
```

运行结果：

```
a  constructor:8:0:0
b  constructor:9:0:0
c  copy constructor,before call: -858993460:-858993460:-858993460
d  copy constructor,before call: 1310592:4200534:1310568
e  copy constructor,before call: -858993460:-858993460:-858993460
```

```
f    destructor:9:0:0
g    destructor:9:0:0
h    constructor:0:0:0
i    destructor:9:0:0
j    destructor:8:0:0
k    destructor:9:0:0
l    destructor:8:0:0
```

程序解释：

(1) 第38、39行定义了两个对象C1、C2，调用构造函数Clock(int,int,int)，产生运行结果的a、b两行。

(2) 第40行首先建立了一个新对象C3，然后将已初始化的对象C1作为初值去初始化尚未初始化的对象C3，因此调用拷贝构造函数。由于拷贝构造函数事实上由构造函数和赋值操作构成，因此在拷贝构造函数中首先显示 H、M、S 的值，然而由于对象C3在初建立时 H、M、S 没有初始化，因此其值是一个无意义的随机数。程序运行结果的第c行就是调用对象C3的拷贝构造函数所产生的结果。

(3) 第41行调用函数fun(C2)时，在栈中为形参建立了一个临时形参对象，实参与形参结合的过程相当于用实参对象初始化新建立的该临时形参对象。临时形参对象的拷贝构造函数被调用，而且临时形参对象在初建立时 H、M、S 没有初始化，程序运行结果为函数fun(C2)将对象作为函数实参传递给函数的形参时拷贝构造函数被调用的结果。

(4) 函数fun(C2)的返回值是一个Clock类的对象，因此在返回前要建立一个临时的返回对象用来存储返回值，在执行return C时将对象C的值复制给临时返回对象，返回过程相当于用已存在的对象C初始化新建立的临时返回对象，临时返回对象的拷贝构造函数被调用。程序运行结果的第e行为临时返回对象的拷贝构造函数被调用的结果。

(5) 函数fun(C2)调用结束时，临时形参对象消失，析构函数被调用，产生结果第f行；临时返回对象消失，析构函数被调用，产生结果第g行。

(6) 第42行建立了一个新对象C4，构造函数被调用，产生运行结果的第h行。

(7) 程序运行结果的第i、j、k、l行分别是对象C4、C3、C2、C1依次调用析构函数的结果。

(8) 第43行是C2对C4的赋值，对象的赋值是当两个对象都已存在，用一个对象的值去覆盖另一个对象的值，被覆盖对象在内存中的原有内容将消失，拷贝构造函数不被调用。

如果将41行改为：

```
fun(Clock(9,0,0));
```

函数调用时，用一个Clock常量(9,0,0)初始化实参，而不是用一个已存在的对象初始化实参。此时调用构造函数，建立一个值为(9,0,0)的实参对象。结果的第d行为：

```
constructor:9:0:0
```

同样，如果将第34行“return C;”改为：

```
return Clock(7,7,7);
```

此时同样调用构造函数产生一个临时返回对象，以常量(7,7,7)为初值，而不是调用拷贝

构造函数将一个已存在的对象初始化返回对象。

在学习拷贝构造函数时，用户要注意拷贝构造函数和对象赋值的区别：拷贝构造函数是用一个存在的对象去构造另一个不存在的对象；对象赋值是两个对象都已存在，用一个对象的值去覆盖另一个对象的值。

☆**注意**：

（1）拷贝构造函数只是在用一个已存在的对象去初始化新建立的对象时调用，在已存在对象间赋值时，拷贝构造函数将不被调用。

（2）用一个常量初始化新建立的对象时，调用构造函数，不调用拷贝构造函数。

（3）建立对象时，构造函数与拷贝构造函数有且只有一个被调用。

（4）当对象作为函数的返回值时需要调用拷贝构造函数，此时C++将从堆中动态建立一个临时对象，将函数返回的对象复制给该临时对象，并把该临时对象的地址存储到寄存器里，从而由该临时对象完成函数返回值的传递。

5.2.4 浅拷贝与深拷贝

在默认的拷贝构造函数中，复制的策略是直接将原对象的数据成员值依次复制给新对象中对应的数据成员，如前面示例p5_4.cpp中定义的拷贝函数，那么我们为何不直接使用系统默认的拷贝构造函数，而自己定义一个拷贝构造函数呢？这是因为，有些情况下使用默认的拷贝构造函数会出现意想不到的问题。

例如，使用程序p5_3.cpp中定义的String类执行下列程序系统会出错：

```
int main()
{
    String s1("123456");
    String s2 = s1;
    return 0;
}
```

为什么会出错呢？程序中首先创建对象s1，为对象s1分配相应的内存资源，调用构造函数初始化该对象，然后调用系统默认的拷贝构造函数将对象s1复制给对象s2，这一切看来似乎很正常，但程序的运行却出现了异常！原因在于默认的拷贝构造函数实现的只能是浅拷贝，即直接将原对象的数据成员值依次复制给新对象中对应的数据成员，并没有为新对象另外分配内存资源。这样，如果对象的数据成员是指针，两个指针对象实际上指向的是同一块内存空间。

当执行String s2＝s1时，默认的浅拷贝构造函数进行的是下列操作：

```
s2.len = s1.len;
s2.Str = s1.Str;
```

实际上是将s1.Str的地址赋给了s2.Str，并没有为s2.Str分配内存，执行“String s2＝s1;”后，对象s2析构，释放内存，然后对象s1析构，由于s1.Str和s2.Str所占用的是同一块内存，而同一块内存不可能释放两次，所以当对象s1析构时，程序出现异常，无法正常执行和结束。由此可知，在某些情况下，浅拷贝会带来数据安全方面的隐患。

当类的数据成员中有指针类型时，我们必须定义一个特定的拷贝构造函数，该拷贝构造函数不仅可以实现原对象和新对象之间数据成员的复制，而且可以为新的对象分配单独的内存资源，这就是**深拷贝构造函数**。

【例 5-5】 带深拷贝构造函数的字符串类。

在程序 p5_3 的 String 类中加入下列拷贝构造函数，构成带深拷贝函数的字符串类。

```
String(String & r)
{
    len = r.len;
    if(len!= 0)
    {
      Str = new char[len + 1];
      strcpy(Str,r.Str);
    }
}
```

下列程序能正常运行(见图 5-1)。

```
int main( ){
    String s1("123456");
    String s2 = s1;
    s1.ShowStr();
    s2.ShowStr();
    return 0;
}
```

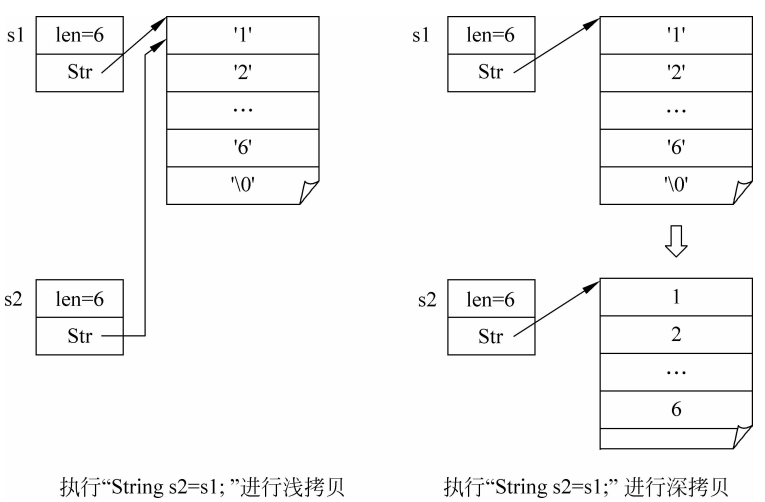


图 5-1　浅拷贝与深拷贝示意图

运行结果：

```
string:123456,length:6
string:123456,length:6
```

☆注意：

(1) 在重新定义拷贝构造函数后，默认拷贝构造函数与默认构造函数就不存在了，如果在此时调用默认构造函数就会出错。

(2) 在重新定义构造函数后，默认构造函数就不存在了，但默认拷贝构造函数还存在。

(3) 在对象进行赋值时，拷贝构造函数不被调用，此时进行的是结构式的拷贝。

5.3 对象的使用

5.3.1 对象指针

对象如同一般变量，占用一块连续的内存区域，因此可以使用一个指向对象的指针来访问对象，即**对象指针**，它指向存放该对象的地址。

用类创建（定义）对象就像使用一般数据类型来定义变量。同样，可以用类来定义对象指针变量，通过对象指针来访问对象的成员。对象指针遵循一般变量指针的各种规则，其定义形式如下：

```
类名  *对象指针名;
```

例如：

```
Clock *Cp;
```

建立了一个指向 Clock 对象的指针。

☆**注意**：

建立了一个对象指针，并没有建立对象，所以此时不调用构造函数。

如同通过对象名访问对象的成员一样，使用对象指针也只能访问该类的公有数据成员和函数成员，但与前者使用“.”运算符不同，对象指针采用“->”运算符访问公有数据成员和成员函数。其语法形式如下：

```
对象指针名->数据成员名
或
对象指针名->成员函数名(参数表)
```

例如：

```
Clock C1(8,0,0);
Clock *Cp;
Cp=&C1;
Cp->ShowTime();
```

在 C++ 中，对象指针可以作为成员函数的形参，一般而言，使用对象指针作为函数的参数要比使用对象作为函数的参数更普遍一些，因为使用对象指针作为函数的参数有以下两点好处。

（1）实现地址传递：通过在调用函数时将实参对象的地址传递给形参指针对象，使形参指针对象和实参对象指向同一内存地址，这样对象指针所指向对象的改变也将同样影响实参对象，从而实现信息的双向传递。

（2）使用对象指针效率高：使用对象指针传递的仅仅是对应实参对象的地址，并不需要实现对象之间的副本复制，这样就会减小时空开销，提高运行效率。

【例 5-6】　时间加法。

时间加法有两种，一种是时钟加秒数，另一种是时钟加时、分、秒，采用重载函数实现这两种加法。

```
/*************************************
*   程序名：p5_6.cpp                  *
*   功　能：带时间加法的时钟类        *
*************************************/
#include <iostream>
using namespace std;
class Clock {
    private:
        int H,M,S;
    public:
        void SetTime(int h,int m,int s)
    {
        H = h,M = m,S = s;
    }
    void ShowTime()
    {
        cout << H <<":"<< M <<":"<< S << endl;
    }
    Clock(int h = 0,int m = 0,int s = 0)
    {
        H = h,M = m,S = s;
    }
    Clock(Clock &p)
    {
         H = p.H, M = p.M, S = p.S;
    }
    void TimeAdd(Clock *Cp);
    void TimeAdd(int h,int m,int s);
    void TimeAdd(int s);
};
void Clock::TimeAdd(Clock *Cp)
{
    H = (Cp->H + H + (Cp->M + M + (Cp->S + S)/60)/60) % 24;
    M = (Cp->M + M + (Cp->S + S)/60) % 60;
    S = (Cp->S + S) % 60;
}
void Clock::TimeAdd(int h,int m,int s)
{
    H = (h + H + (m + M + (s + S)/60)/60) % 24;
    M = (m + M + (s + S)/60) % 60;
    S = (s + S) % 60;
}
void Clock::TimeAdd(int s)
{
    H = (H + (M + (S + s)/60)/60) % 24;
    M = (M + (S + s)/60) % 60;
    S = (S + s) % 60;
```

```
48  }
49  int main()
50  {
51      Clock C1;
52      Clock C2(8,20,20);
53      C1.TimeAdd(4000);
54      C1.ShowTime();
55      C2.TimeAdd(&C1);
56      C2.ShowTime();
57      return 0;
58  }
```

运行结果：

```
1:6:40
9:27:0
```

5.3.2 对象引用

对象引用就是对某类对象定义一个引用，其实质是通过将被引用对象的地址赋给引用对象，使二者指向同一个内存空间，这样引用对象就成为了被引用对象的“别名”。

对象引用的定义方法与基本数据类型变量引用的定义是一样的。定义一个对象引用，并同时指向一个对象的格式如下：

```
类名  &  对象引用名 = 被引用对象;
```

☆注意：

（1）对象引用与被引用对象必须是同类型的。

（2）除非是作为函数参数与函数返回值，对象引用在定义时必须要初始化。

（3）定义一个对象引用并没有定义一个对象，所以不分配任何内存空间，不调用构造函数。

对象引用的使用格式如下：

```
对象引用名.数据成员名
或
对象引用名.成员函数名(参数表)
```

例如：

```
Clock C1(8,20,20);
Clock& Cr = C1;                    //定义了 C1 的对象引用 Cr
Cr.ShowTime();                     //通过对象引用使用对象的成员
```

运行结果为：

```
8:20:20
```

对象引用通常用作函数的参数，它不仅具有对象指针的优点，而且比对象指针更简洁、更方便、更直观。在 p5_6.cpp 中添加以下函数：

```
void Clock::TimeAdd(Clock & Cr)
{
    H = (Cr.H + H + (Cr.M + M + (Cr.S + S)/60)/60) % 24;
    M = (Cr.M + M + (Cr.S + S)/60) % 60;
    S = (Cr.S + S) % 60;
}
```

将“C2. TimeAdd(&C1);”替换为“C2. TimeAdd(C1);”，运行结果与 p5_6.cpp 一样。

5.3.3　对象数组

对象数组是以对象为元素的数组。对象数组的定义、赋值、引用与普通数组一样，只是数组元素与普通数组的数组元素不同。对象数组的定义格式如下：

```
类名　对象数组名[常量表达式 n],…,[常量表达式 2][常量表达式 1];
```

其中，类名指出该数组元素所属的类，常量表达式给出某一维元素的个数。

与结构数组不同，对象数组的初始化需要使用构造函数完成，以一个大小为 n 的一维数组为例，对象数组的初始化格式如下：

```
数组名[n] = {类名(数据成员 1 初值,数据成员 2 初值,…),
            类名(数据成员 1 初值,数据成员 2 初值,…),
            …                                          } n
            类名(数据成员 1 初值,数据成员 2 初值,…)};
```

不带初始化表的对象数组，其初始化靠调用不带参数的构造函数完成。

以一个 m 维数组为例，对象数组元素的存取格式如下：

```
对象数组名[下标表达式 1][下标表达式 2]…[下标表达式 m].数据成员名
或
对象数组名[下标表达式 1][下标表达式 2]…[下标表达式 m].成员函数名(参数表)
```

【例 5-7】 计算一个班学生某门功课的总评成绩。

分析：首先设计一个类 Score，这个类的数据成员为一个学生的学号、姓名、平时成绩、期末考试成绩，成员函数有求总评成绩、显示成绩，然后定义一个对象数组存储一个班学生的成绩，最后通过逐一调用数组元素的成员函数求每个学生的总评成绩。

```
1  /**************************************************
2   *   程序名: p5_7.cpp                              *
3   *   功   能: 求一个班学生某门功课的总评成绩       *
4   **************************************************/
5  #include <iostream>
6  using namespace std;
7  const int MaxN = 100;
```

```
const double Rate = 0.6;                                          //平时成绩比例
class Score {
      private:
          long No;                                                //学号
          char * Name;                                            //姓名
          int Peace;                                              //平时成绩
          int Final;                                              //期末考试成绩
          int Total;                                              //总评成绩
      public:
          Score(long = 0,char * = NULL,int = 0,int = 0,int = 0);  //构造函数
          void Count();                                           //计算总评成绩
          void ShowScore();                                       //显示成绩
};
Score::Score(long no,char * name,int peace,int final,int total)
      {                                                           //构造函数
           No = no;
           Name = name;
           Peace = peace;
           Final = final;
           Total = total;
      }
void Score::Count()
{
        Total = Peace * Rate + Final * (1 - Rate) + 0.5;
}
void Score::ShowScore()
{
  cout << No <<"\t"<< Name <<"\t"<< Peace <<"\t"<< Final <<"\t"<< Total << endl;
}

int main()
{
    Score ClassScore1[3];
    Score ClassScore2[3] = { Score(200607001,"LiuNa",80,79),
                             Score(200607002,"CuiPeng",90,85),
                             Score(200607003,"ZhouJun",70,55)};
    for(int i = 0;i < 3;i++)
         ClassScore2[i].Count();
    for(i = 0;i < 3;i++)
         ClassScore2[i].ShowScore();
    return 0;
}
```

运行结果：

```
200607001       LiuNa   80      79      80
200607002       CuiPeng 90      85      88
200607003       ZhouJun 70      55      64
```

程序解释：

(1) 程序第 17 行定义了带默认形参值的构造函数。

(2) 程序第 39 行建立了 3 个元素的对象数组，相当于建立了 3 个对象，即 ClassScore1[0]、ClassScore1[1]、ClassScore1[2]，建立每个对象时调用一次不带参数的构造函数，从构造函数的默认形参值中获得数据成员的初值。

(3) 第 40 行建立了 3 个元素的对象数组，在初值表中给出了构造函数的调用形式，每个数组元素(对象)通过调用构造函数从构造函数的实参中获得数据成员的初值。因此，定义一个对象数组，相当于定义了若干个对象。每个数组元素都是一个对象。数组元素的个数为 n，定义数组时调用构造函数的次数为 n。

5.3.4　动态对象

函数体内的局部对象在调用函数时建立，在函数调用完后消失；全局对象则在程序执行时建立，要执行完成后消失；这些对象在何时建立，何时消失是 C++ 规定好的，不是编程者能控制的。**动态对象**是指编程者随时动态建立并可随时消失的对象。

建立动态对象采用动态申请内存的语句 new，删除动态对象使用 delete 语句。建立一个动态对象的格式如下：

```
对象指针 = new 类名(初值表);
```

其中：

- 对象指针的类型应该与类名一致。
- 动态对象存储在 new 语句从堆中申请的空间中。
- 建立动态对象时要调用构造函数，当初值表省略时调用默认的构造函数。

例如：

```
Clock * Cp;                          //建立对象指针
Cp = new Clock;                      //建立动态对象,调用默认构造函数 Clock()
Cp -> ShowTime();                    //结果为 0:0:0
Cp = new Clock(8,0,0);               //建立动态对象,调用构造函数 Clock(int,int,int)
Cp -> ShowTime();                    //结果为 8:0:0
```

在堆中建立的动态对象不能自动消失，需要使用 delete 语句删除对象，格式如下：

```
delete 对象指针;
```

在删除动态对象时，释放堆中的内存空间，在对象消失时，调用析构函数。例如：

```
delete Cp;                           //删除 Cp 指向的动态对象
```

动态对象的一个重要的使用方面是用动态对象组成动态对象数组。建立一个一维动态对象数组的格式如下：

```
对象指针 = new 类名[数组大小];
```

删除一个动态对象数组的格式如下：

```
delete[]  对象指针;
```

在建立动态对象数组时，要调用构造函数，调用的次数与数组的大小相同；在删除对象数组时，要调用析构函数，调用次数与数组的大小相同。

将 p5_7.cpp 改为用动态对象数组实现，代码如下：

```
Score::SetScore(long no,char *name,int peace,int final,int total)
    {
        No = no;
        Name = name;
        Peace = peace;
        Final = final;
        Total = total;
    }
```

SetScore()函数为动态数组设置初值。

```
int main()
{
    Score *ClassScore;
    ClassScore = new Score [3];
    ClassScore[0].SetScore(200607001,"LiuNa",80,79),
    ClassScore[1].SetScore(200607002,"CuiPeng",90,85),
    ClassScore[2].SetScore(200607003,"ZhouJun",70,55);
    for(int i = 0;i<3;i++)
        ClassScore[i].Count();
    for(i = 0;i<3;i++)
        ClassScore[i].ShowScore();
    delete [] ClassScore;
    return 0;
}
```

5.3.5 this 指针

在一个类的成员函数中，有时希望引用调用它的对象，对此，C++采用隐含的 this 指针来实现。**this** 指针是一个系统预定义的特殊指针，指向当前对象，表示当前对象的地址。例如：

```
void Clock::SetTime(int h, int m, int s)
{
    H = h, M = m, S = s;
    this->H = h, this->M = m, this->S = s;
    (*this).H = h, (*this).M = m, (*this).S = s;
}
```

此 3 句是等效的

起初，为了与类的数据成员 H、M、S 相区别，将 SetTime 的形参名设为 h、m、s。如果使用 this 指针，就可以凭 this 指针区分本对象的数据成员与其他变量。使用 this 指针重新设计的 SetTime()成员函数如下：

```
void Clock::SetTime (int H, int M, int S)
{
    this->H = H, this->M = M, this->S = S;
}
```

系统利用 this 指针明确指出成员函数当前操作的数据成员所属的对象。实际上，当一个对象调用其成员函数时，编译器先将该对象的地址赋给 this 指针，然后调用成员函数，这样成员函数对对象的数据成员进行操作时就隐含使用了 this 指针。

一般而言，通常不直接使用 this 指针来引用对象成员，但在某些少数情况下，可以使用 this 指针，例如重载某些运算符以实现对象的连续赋值等。

☆**注意**：

(1) this 指针不是调用对象的名称，而是指向调用对象的指针的名称。

(2) this 的值不能改变，它总是指向当前调用对象。

5.3.6　组合对象

在现实生活中有很多“组合”的例子，例如，如果你想根据自己工作或学习的需要配置一台合适的计算机，首先要根据自己的资金预算设计符合自己需要的个性化装机方案，然后从市场上选购 CPU、主板、内存、硬盘、显示器、光驱、机箱、键盘、鼠标等硬件和软件，最后将这些硬件和软件按规范的方式“组合”。这样，你就能得到一台为自己“量身定制”的计算机，这就是组合的概念。

组合概念体现的是一种包含与被包含的关系，在语义上表现为“is a part of”的关系，即在逻辑上 A 是 B 的一部分，例如眼(eye)、鼻(nose)、口(mouth)、耳(ear)是头(head)的一部分，如果将 head、eye、nose、mouth、ear 定义成类，类 head 应该由类 eye、nose、mouth、ear 组合而成。同样，CPU、主板、内存等都是计算机的一个组成部 分，相对于计算机这个复杂类而言，它们是成员类，所以我们把这种组合关系称为“is part of”。

在 C++程序设计中，类的组合用来描述一类复杂的对象，在类的定义中，它的某些属性往往是另一个类的对象，而不是像整型、浮点型之类的简单数据类型，也就是“一个类内嵌其他类的对象作为成员”，对于将对象嵌入到类中的这样一种描述复杂类的方法，我们称之为“类的组合”，一个含有其他类对象的类称为**组合类**，组合类的对象称为**组合对象**。

1. 组合类的定义

组合类定义的步骤为先定义成员类，再定义组合类。

【例 5-8】 计算某次火车的旅途时间。

分析：某次火车有车次、起点站、终点站、出发时间、到达时间。前面定义的 Clock 类具有时间特性，因此可以利用 Clock 对象组合成一个火车旅途类 TrainTrip。假定火车均在 24 小时内到达，旅途时间为到达时间减去出发时间。

若用空方框表示类，用灰框表示对象，组合类可以表示为空框包含灰框。设计 TrainTrip 类的示意图与成员构成图如图 5-2 所示。

类名	成员名	
TrainTrip	Clock StartTime	*H*、*M*、*S*
		SetTime()、ShowTime()
	Clock EndTime	*H*、*M*、*S*
		SetTime()，ShowTime()
	char * TrainNO	
	clock TripTime()	

图 5-2　类 TrainTrip 的构成与成员

```
/*****************************************
*  程序名: p5_8.cpp                      *
*  功  能: 计算火车旅途时间的组合类      *
*****************************************/
#include <iostream>

using namespace std;
class Clock {
    private:
      int H,M,S;
    public:
      void ShowTime()
      {
         cout<<H<<":"<<M<<":"<<S<<endl;
      }
      void SetTime(int H=0,int M=0,int S=0)
      {
         this->H=H, this->M=M, this->S=S;
      }
      Clock(int H=0,int M=0,int S=0)
      {
         this->H=H, this->M=M, this->S=S;
      }
      int GetH()
      {
          return H;
      }
      int GetM()
      {
          return M;
      }
      int GetS()
      {
          return S;
      }
  };
class TrainTrip {
    private:

      char *TrainNo;                          //车次
      Clock StartTime;                        //出发时间
      Clock EndTime;                          //到达时间
    public:
      TrainTrip(char * TrainNo, Clock S, Clock E)
      {
          this->TrainNo=TrainNo;
          StartTime=S;
          EndTime=E;
      }
```

```
49        Clock TripTime()

50        {
51            int tH,tM,tS;                          //临时存储小时、分、秒数
52            int carry = 0;                         //借位
53            Clock tTime;                           //临时存储时间
54            (tS = EndTime.GetS() - StartTime.GetS())> 0?carry = 0:tS += 60, carry = 1;
55      (tM = EndTime.GetM() - StartTime.GetM() - carry)> 0?carry = 0:tM += 60, carry = 1;
56            (tH = EndTime.GetH() - StartTime.GetH() - carry)> 0?carry = 0: tH += 24;
57          tTime.SetTime(tH,tM,tS);
58          return tTime;
59      }
60  };
61  int main()
62  {
63     Clock C1(8,10,10), C2(6,1,2);                 //定义 Clock 类的对象
64     Clock C3;                                     //定义 Clock 类对象,存储结果
65     TrainTrip T1("K16",C1,C2);                    //定义 TrainTrip 对象
66     C3 = T1.TripTime();
67     C3.ShowTime();
68     return 0;
69  }
```

运行结果：

```
21:50:52
```

程序解释：

(1) 时、分、秒值 H、M、S 是 Clock 类的私有成员，在 Clock 类外无法存取；而在 TrainTrip 类中需要 H、M、S 值，因此在 Clock 类中提供了公有的存取 H、M、S 值的接口函数 GetH()、GetM()、GetS()供 TrainTrip 类读取 H、M、S。程序第 23～34 行 GetH()、GetM()、GetS()为读取 H、M、S 值的函数。

(2) 旅途时间的结果是一个时间值，因此将函数 TripTime()返回值的类型设置成 Clock 类型。

(3) TrainTrip 是组合类，为了给各个成员提供初值，建立了两个 Clock 对象 C1、C2，并设计了一个构造函数 TrainTrip，将 C1、C2 的值赋给组合类的成员对象。程序第 65 行在建立对象 T1 时利用构造函数将初值赋给 T1。

2. 组合对象的初始化

在程序 p5_8.cpp 中，为了初始化成员对象，建立了两个对象 C1、C2，这两个对象除了用于初始化外，没有其他用处，而且在程序运行期间一直占据内存，造成额外的内存开销。

C++为组合对象提供了初始化机制：在定义组合类的构造函数时可以携带初始化表。其格式如下：

```
组合类名(形参表): 成员对象 1(子形参表 1),成员对象 2(子形参表 2),…
                  └──────────成员对象初始化表──────────┘
```

其中：

- “成员对象 1(子形参表 1)，成员对象 2(子形参表 2)，…”称为**初始化列表**，该表放在构造函数的头部，各参数之间用逗号隔开。成员对象可以是类的普通数据成员。
- 形参表中的形参为该类的所有数据成员提供初值，子形参表中的形参为形参表中形参的一部分，子形参表 *n* 为成员对象 *n* 提供初值。

在为组合类定义了带初始化表的构造函数后，在建立组合对象时为对象提供初值的格式如下：

```
类名 对象名(实参表);
```

在建立对象时，调用组合类的构造函数；在调用组合类的构造函数时，先调用各个成员对象的构造函数，成员对象的初值从初始化列表中取得。这样，实际上是通过成员类的构造函数对成员对象进行初始化，初始化值在初始化表中提供。

使用初始化列表，将 p5_8.cpp 修改成 p5_8a.cpp，将其中的构造函数修改如下：

```
TrainTrip(char * TrainNo, int SH,int SM, int SS, int EH, int EM,int ES):
                EndTime(EH,EM,ES), StartTime(SH,SM,SS)
    {
        this->TrainNo = TrainNo;
    }
```

将定义组合对象的程序修改如下：

```
int main()
{
    Clock C3;                                          //定义 Clock 类对象,存储结果
    TrainTrip T1("K16",8,10,10,6,1,2);                 //定义 TrainTrip 对象
    C3 = T1.TripTime();
    C3.ShowTime();
    return 0;
}
```

程序运行结果与 p5_8.cpp 完全相同。

在建立对象 T1 时调用构造函数 TrainTrip()，由于建立对象 T1 要先建立 StartTime、EndTime 两个成员对象，因此分别通过 StartTime、EndTime 调用构造函数 Clock()，构造函数的参数从初始化列表中取得。在 StartTime、EndTime 构造完毕后，再执行 TrainTrip()余下的部分。在构造 StartTime、EndTime 时如果没有初始化列表，则分别调用默认形式的构造函数 Clock()。

☆**注意**：

初始化列表既不能决定是否调用成员对象的构造函数，也不能决定调用构造函数的顺序，成员对象的调用顺序由成员对象定义的顺序决定，初始化列表只是提供调用成员对象构造函数的参数。

对于 C++语言而言，类中的成员以其在类中声明的先后顺序依次构造，而不是根据构造函数初始化列表中成员对象构造函数声明的先后顺序来决定。如果构造函数中指定了一个特殊的构造顺序，那么析构函数将不得不去查询构造函数的定义，以获得如何对成员进行析构的顺序。由于构造函数和析构函数可以在不同的文件中定义，这就给编译器的实现者造成一个难

题，而且由于构造函数可以重载，因此一个类可以有两个或者更多的构造函数。对于这些构造函数的定义，程序设计者并不能保证它们中的所有成员对象声明的先后顺序都是一致的，所以C++语言只能依据类中成员对象的声明顺序决定成员对象的构造和析构顺序。

在上述初始化列表中，初始化 StartTime 与 EndTime 采用的是用 H、M、S 分别初始化成员对象的方法，也可以采用 Clock 对象整体初始化成员对象。将 p5_8a.cpp 修改成 p5_8b.cpp，对其中的构造函数修改如下：

```
TrainTrip(char * TrainNo, Clock S, Clock E): StartTime(S),EndTime(E)
      {
         this -> TrainNo = TrainNo;
      }
```

可采用与 p5_8.cpp 完全相同的方式定义组合对象，也可以将定义方式修改如下：

```
int main()
{
    Clock C3;                                                    //定义 Clock 类对象,存储结果
    TrainTrip T1("K16",Clock(8,10,10),Clock(6,1,2));     //定义 TrainTrip 对象
    C3 = T1.TripTime();
    C3.ShowTime();
    return 0;
}
```

与 p5_8.cpp 相比，发现分别使用常量 Clock(8,10,10)、Clock(6,1,2)取代了 C1、C2 作为成员对象的初值。

3. 组合对象的构造函数

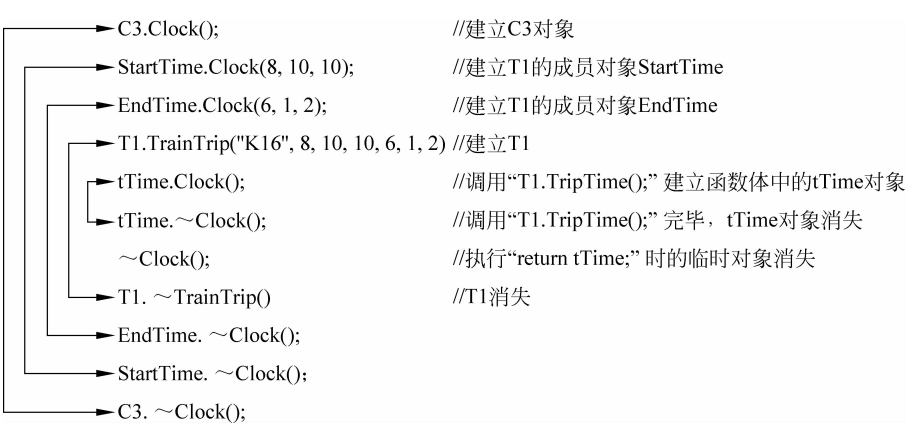
在定义一个组合类的对象时，不仅它自身的构造函数将被调用，还将调用其成员对象的构造函数，调用的先后顺序如下：

(1) 成员对象按照在其组合类的声明中出现的次序依次调用各自的构造函数，而不是按初始化列表中的顺序。如果建立组合类的成员对象时没有指定对象的初始值，则自动调用默认的构造函数。

(2) 组合类对象调用组合类构造函数。

(3) 调用析构函数，析构函数的调用顺序与构造函数正好相反。

p5_8a.cpp 的构造函数的调用顺序如下：



用对象初始化成员对象，在实参对象与形参结合时，函数返回对象时还会调用拷贝构造函数。程序 p5_8.cpp、p5_8a.cpp 和 p5_8b.cpp 的构造函数、拷贝构造函数、析构函数的详细调用过程留给读者作为练习。

4. 组合类成员对象的访问权限

组合类成员对象的成员有存取控制权限，成员对象作为组合类的成员后，又有存取控制权限。成员对象中的成员在组合类中的存取权限经过了两次限制，其最终权限见表 5-2。

表 5-2　各种成员对象在组合类中的访问控制属性

组合类 \ 成员对象	public	protected	private
public	public	不可访问	不可访问
protected	protected	不可访问	不可访问
private	private	不可访问	不可访问

例如：

```
1   class Clock {
2       protected:
3         int H,M,S;
4
5   };
6   class TrainTrip {
7       public:
8          Clock StartTime;
9          Clock EndTime;
10         void ShowEndTime()
11          {
12             cout << EndTime.H;   //错误
13         }
14  };
15  int main()
16  {
17    TrainTrip T;
18    cout << T.StartTime.H << endl;   //错误
19    return 0;
20  }
```

```
class Clock {
    public:
      int H,M,S;

};
class TrainTrip {
    public:
       Clock StartTime;
       Clock EndTime;
       void ShowEndTime()
        {
          cout << EndTime.H;
       }
};
int main()
{
  TrainTrip T;
  cout << T.StartTime.H << endl;
  return 0;
}
```

（1）程序第 12 行，在类外存取成员对象 EndTime 的 protected 成员，因此出错。

（2）程序第 18 行，在类外存取访问控制为 protected 的非本类成员，因此出错。将类 Clock 中的存取控制权限由 protected 改为 public，第 12 行、第 18 行均能通过编译，正确运行。

5.4 静态成员

在一所大学中，每个学生都有自己独立拥有的私人生活空间，但在校园中，总有许多活动空间和场所，它们不属于某个学生所独享，而为所有学生所共享，例如大学生活动中心、操场、体育馆等公共设施。在C++类的定义中，如何描述不为某个对象独享，而是为类的所有对象共享的数据？

一个类对象的public成员可被本类的其他对象存取，即可供所有对象使用，但是此类的每个对象各自拥有一份，不存在真正意义上的共享成员。所以，C++提供了静态成员，用于解决同一个类的不同对象之间数据成员和函数的共享问题。

静态成员的特点是不管这个类创建了多少个对象，其静态成员在内存中只保留一个副本，这个副本为该类的所有对象所共享。

类的静态成员有两种，即静态数据成员和静态函数成员，下面分别对它们进行讨论。

5.4.1 静态数据成员

在C++程序中，类与对象的关系如同模具与铸件之间的关系，每个对象都称为类的实例，实例又意味着每个对象需要自己的存储空间，以保存它们自己的属性值，我们说同类的对象具有相同的属性和行为，是指各个对象属性的名称、个数、数据类型相同，而不是指每个对象的属性值都相同。这种属性在面向对象方法中称为**实例属性**(instance attribute)。

在面向对象方法中还有**类属性**(class attribute)的概念，类属性是描述类的所有对象的共同特征的一个数据项，对于任何对象实例，它的属性值是相同的，C++通过静态数据成员来实现类属性。

类的普通数据成员在类的每一个对象中都拥有一个副本，也就是说每个对象的同名数据成员可以分别存储不同的数值，这也是保证对象拥有自身区别于其他对象的特征的需要，属于实例属性。**静态数据成员**是类的数据成员的一种特例，采用static关键字来定义，属于类属性，每个类只有一个副本，由该类的所有对象共同维护和使用，从而实现了同类的不同对象之间的数据共享。

静态数据成员的定义分为两个必不可少的部分，即类内声明、类外初始化。

在类内，声明静态数据成员的格式如下：

```
static  数据类型  静态数据成员名;
```

在类外初始化的形式如下：

```
数据类型  类名::静态数据成员名 = 初始值;
```

在定义与使用静态数据成员时应注意以下几点：

(1) 静态数据成员的访问属性和普通数据成员一样，可以为public、private和protected。

(2) 静态数据成员脱离具体对象而独立存在，其存储空间是独立分配的，不是任何对象存储空间的一部分，但逻辑上所有的对象都共享这一存储单元，所以对静态数据成员的任何操作都将影响共享这一存储单元的所有对象。

（3）静态数据成员是一种特殊的数据成员，它表示类属性，而不是某个对象单独的属性，它在程序开始时产生，在程序结束时消失。静态数据成员具有静态生存期。

（4）由于在类的定义中仅仅是对静态数据成员进行了引用性声明，因此必须在文件作用域的某个地方对静态数据成员进行定义并初始化，即应该在类体外对静态数据成员进行初始化（静态数据成员的初始化与它的访问控制权限无关）。

（5）在对静态数据成员初始化时前面不加 static 关键字，以免与一般静态变量或对象混淆。

（6）由于静态数据成员是类的成员，因此在初始化时必须使用类作用域运算符::限定它所属的类。

例如，我们对某学校不同学生的特性进行抽象，找出共性设计一个学生类 Student。如果需要统计学生的总人数，可以在类外定义一个全局变量，但是类外的全局变量不受类存取控制的保护。因此可以将学生人数定义为静态成员，即学生类的类属性。设计的 Student 类如下：

```
class Student {
    private:
        char * Name;
        int No;
        static int countS;
    };
```

在类外对静态成员初始化如下：

```
int Student::countS = 0;
```

除了在初始化时可以在类外通过类对静态成员赋初值外，在其他情况下对静态成员的存取规则与一般成员相同，即在类内可以任意存取，在类外通过类名与对象只能访问存取属性为 public 的成员。例如：

```
Student S1;
S1.countS++;                                    //错误,countS 存取属性为 private
Student: : countS++;                            //错误
```

对静态数据成员的访问还可以通过类的成员函数进行。

5.4.2 静态成员函数

静态数据成员为类属性，在定义类后、建立对象前就存在。因此，在建立对象前不能通过成员函数存取静态数据成员。C++提供了静态成员函数，用来存取类的静态成员。**静态成员函数**是用关键字 static 声明的成员函数，它属于整个类而不属于类中的某个对象，是该类的所有对象共享的成员函数。

静态成员函数可以在类体内定义，也可以在类内声明为 static，在类外定义。当在类外定义时，不能再使用 static 关键字作为前缀。

静态函数成员的调用形式有以下两种。

1. 通过类名调用静态成员函数

静态成员函数为类的全体对象而不是部分对象服务，与类相联系而不与类的对象联系，因此访问静态成员函数时可以直接使用类名。其格式如下：

```
类名::静态成员函数;
```

通过类名访问静态成员函数同样受静态成员函数访问权限的控制。

2. 通过对象调用静态成员函数

其格式如下：

```
对象.静态成员函数
```

在通过对象调用静态成员函数时应注意下面几点：

(1) 通过对象访问静态成员函数的前提条件为对象已经建立。

(2) 静态成员函数的访问权限和普通成员函数一样。

(3) 静态成员函数也可以省略参数、使用默认形参值以及进行重载。

静态成员函数和普通成员函数在使用上还有以下区别：

(1) 由于静态成员函数在类中只有一个拷贝(副本)，因此它访问对象的成员时要受到一些限制，即静态成员函数可以直接访问类中说明的静态成员，但不能直接访问类中的非静态成员。若要访问非静态成员，必须通过参数传递的方式得到相应的对象，再通过对象访问。

(2) 由于静态成员是独立于类对象存在的，因此静态成员没有 this 指针。

【例 5-9】 使用静态成员维护内存中 Student 类对象的个数(对象计数器)。

分析：为了维护内存中 Student 类对象的个数，除了定义一个静态数据成员存储类对象个数外，还要在所有可能建立对象、删除对象的场合记载对对象个数的修改。

不管是以什么方式建立对象，都要调用构造函数或拷贝构造函数，因此在构造函数和拷贝构造函数中将对象计数器加 1。对象消失、删除对象时要调用析构函数，在析构函数中将对象计数器减 1。

```
1   /*****************************************
2    *   程序名: p5_9.cpp                    *
3    *   功   能: 含有对象计数器的学生类      *
4    *****************************************/
5   #include<iostream>
6   using namespace std;
7   class Student {
8       private:
9           char *Name;
10          int No;
11          static int countS;
12      public:
13           static int GetCount()
14        {
15            return countS;
16        }
17        Student(char* ="", int=0);
18        Student(Student &);
19        ~Student();
20  };
```

```
Student::Student(char * Name, int No)
{
   this->Name = new char [strlen(Name) + 1];
   strcpy(this->Name, Name);
   this->No = No;
   ++countS;
   cout <<"constructing:"<< Name << endl;
}
Student::Student(Student &r)
{
   Name = new char [strlen(r.Name) + 1];
   strcpy(Name, r.Name);
   No = r.No;
   ++countS;
   cout <<"copy constructing:"<< r.Name << endl;
}
Student::~Student()
{
   cout <<"destructing:"<< Name << endl;
   delete [] Name;
   --countS;
}
int Student::countS = 0;
int main()
{
      cout << Student::GetCount()<< endl;          //使用类调用静态成员函数
      Student s1("Antony");                        //建立一个新对象
      cout << s1.GetCount()<< endl;                //通过对象调用静态成员函数
      Student s2(s1);                              //利用已有对象建立一个新对象
      cout << s1.GetCount()<< endl;
      Student S3[2];                               //建立一个对象数组
      cout << Student::GetCount()<< endl;
      Student * s4 = new Student[3];               //建立一个动态对象数组
      cout << Student::GetCount()<< endl;
      delete [] s4;                                //删除动态对象数组
      cout << Student::GetCount()<< endl;
      return 0;
}
```

运行结果：

```
0
constructing:Antony
1
copy constructing:Antony
2
constructing:
constructing:
4
constructing:
constructing:
constructing:
```

```
7
destructing:
destructing:
destructing:
4
destructing:
destructing:
destructing:Antony
destructing:Antony
```

程序解释：

(1) 第46行，此时还没有任何对象建立，因此只能通过类调用静态成员函数返回对象数目。

(2) 第49行，用已存在的对象初始化新建立的对象，此时调用拷贝构造函数。

(3) 第51行，建立对象数组，元素个数为2，调用两次构造函数，对象计数器值加2。

(4) 第53行，建立动态对象数组，元素个数为3，调用3次构造函数，对象计数器值加3。

(5) 第55行，释放动态对象数组，调用3次析构函数，对象计数器值减3。

5.5　友　　元

类具有封装性和隐蔽性，只有类的函数成员才能访问类的私有成员，程序中的其他函数是无法访问类的私有成员的，对象的这种数据封装和数据隐藏使得类的对象和外界像被一堵不透明的墙隔开，有些时候，类的这种特性给程序设计增加了负担，它要求程序设计者必须确保每个类都能够提供足够的成员函数对所有可能遇到的访问请求进行处理。而且，某些成员函数频繁调用时，由于函数参数的传递、C++严格的类型检查和安全性检查将带来时间上的开销，从而影响程序的运行效率。

但是，将类的成员访问控制属性设计为public类型，这样类的非成员函数就可以访问类的成员，如此一来，就破坏了类封装性和隐蔽性，失去了C++最基本的优点。

由此可见，数据隐藏给不同类和对象的成员函数之间、类的成员函数和类外的一般函数之间进行属性共享带来障碍，必须寻求一种方法使得类外的对象能够访问类中的私有成员，提高程序的效率。为了解决这个问题，C++提出了使用友元作为实现这一要求的辅助手段，C++中的友元为类的封装、隐藏这堵不透明的墙开了一个小小的孔，外界可以通过这个小孔窥视类内部的一些属性，只要将外界的某个对象说明为某一个类的友元，那么这个外界对象就可以访问这个类对象中的私有成员。

友元不是类的成员，但它可以访问类的任何成员。声明为友元的外界对象既可以是另一个类的成员函数，也可以是不属于任何类的一般函数，称为**友元函数**。友元也可以是整个的一个类，称为友元类。

5.5.1　友元函数

友元函数是在类定义中由关键字friend修饰的非成员函数。其格式如下：

```
friend 返回类型 函数名(形参表)
{
…  //函数体
}
```

从定义语法上看，友元函数的定义与成员函数一样，只是在类中用关键字 friend 予以说明。但友元函数是一个普通的函数，它不是本类的成员函数，因此在调用时不能通过对象调用。

友元函数也可以在类内声明，在类外定义，但在类外定义时不得指出函数所属的类。

友元函数对类成员的存取和成员函数一样，可以直接存取类的任何存取控制属性的成员；可通过对象存取形参、函数体中该类类型对象的所有成员。

private、protected、public 访问权限与友员函数的声明无关，因此原则上友元函数声明可以放在类体中的任意部分，但为了使程序清晰，一般放在类体的后面。

下面举例说明友元函数的应用。

【例 5-10】 使用友元函数计算某次火车的旅途时间。

分析：为了简化问题，分别用两个 Clock 类对象表示某次火车的出发时间、到达时间。假定火车均在 24 小时内到达，旅途时间为到达时间减去出发时间。

```
1  /*******************************************
2   *   程序名: p5_10.cpp                      *
3   *   功   能: 计算火车旅途时间的友元函数    *
4   *******************************************/
5  #include <iostream>
6  using namespace std;
7  class Clock {
8      private:
9        int H,M,S;
10     public:
11       void ShowTime()
12       {
13          cout<<H<<":"<<M<<":"<<S<<endl;
14       }
15       void SetTime(int H=0,int M=0,int S=0)
16       {
17           this->H=H, this->M=M, this->S=S;
18       }
19       Clock(int H=0,int M=0,int S=0)
20       {
21          this->H=H, this->M=M, this->S=S;
22       }
23       friend Clock TripTime(Clock &StartTime, Clock &EndTime);
24 };
25 Clock TripTime(Clock &StartTime, Clock &EndTime)
26 {
27     int tH,tM,tS;                          //临时存储小时、分、秒数
28     int carry=0;                           //借位
29     Clock tTime;                           //临时存储时间
```

```
30      (tS = EndTime.S - StartTime.S)> 0?carry = 0:tS += 60,carry = 1;
31      (tM = EndTime.M - StartTime.M - carry)> 0?carry = 0:tM += 60,carry = 1;
32      (tH = EndTime.H - StartTime.H - carry)> 0?carry = 0:tH += 24;
33      tTime.SetTime(tH,tM,tS);
34      return tTime;
35  }
36  int main()
37  {
38      Clock C1(8,10,10), C2(6,1,2);                    //定义 Clock 类的对象
39      Clock C3;                                        //定义 Clock 类对象,存储结果
40      C3 = TripTime(C1,C2);
41      C3.ShowTime();
42      return 0;
43  }
```

运行结果：

```
21:50:52
```

程序解释：

(1) 第 25 行，定义友元函数，如果前面加上 Clock::是错误的。

(2) 第 40 行，调用友元函数，如果通过对象调用 C1.TripTime()是错误的。

在本例中，在 Clock 类体中设计了一个友元函数 TripTime()，它不是类的成员函数。但是，我们可以看到友元函数中通过对象名 StartTime 和 EndTime 直接访问了它们的私有数据成员 StartTime.H、StartTime.M、StartTime.S，这就是友元的关键所在。

使用友元成员的好处是两个类可以以某种方式相互合作、协同工作，共同完成某一个任务。

5.5.2　友元类

友元除了可以是函数外，还可以是类，如果一个类声明为另一个类的友元，则该类称为另一个类的**友元类**。若 A 类为 B 类的友元类，则 A 类的所有成员函数都是 B 类的友元函数，都可以访问 B 类的任何数据成员。

友元类的声明是在类名之前加上关键字 friend 实现的。声明 A 类为 B 类的友元类的格式如下：

```
class B {
…
friend class A;
};
```

这里要注意一个问题，在声明 A 类为 B 类的友元类时，A 类必须已经存在，但是如果 A 类又将 B 类声明为自己的友元类，又会出现 B 类不存在的错误。

显然，当遇到两个类相互引用的情况时，必然有一个类在定义之前就被引用，那么怎么办呢？对此，C++专门规定了**前向引用声明**，用于解决这类问题。前向引用声明是在引用未定义的类之前对该类进行声明，它只是为程序引入一个代表该类的标识符，类的具体定义可以在程

序的其他地方进行。

例如下面的情况，类 A 的成员函数 funA()的形式参数是类 B 的对象，同时类 B 的成员函数 funB()以类 A 的对象为形参，这时就必须使用前向引用声明：

```
class  B;                                          //前向引用声明
class  A  {                                        //A 类的定义
    public:                                        //外部接口
      void funA(B  b);                             //以 B 类对象 b 为形参的成员函数
};
class  B  {                                        //B 类的定义
    public:                                        //外部接口
      void funB(A  a);                             //以 A 类对象 a 为形参的成员函数
};
```

【例 5-11】 使用友元类计算某次火车的旅途时间。

分析： 在 p5_8.cpp 中，定义了一个组合类 TrainTrip，组合了 Clock 类对象表示某次火车的出发时间、到达时间。但是，TrainTrip 中的成员函数无法直接存取出发时间、到达时间中的访问控制为 private 的 H、M、S。如果将 TrainTrip 定义为 Clock 的友元类，则 TrainTrip 中的成员函数可以直接存取出发时间、到达时间中的数据成员。

```
1   /*********************************************
2    *   程序名：p5_11.cpp                        *
3    *   功   能：计算火车旅途时间的友元类        *
4    *********************************************/
5   #include <iostream>
6   using namespace std;
7   class TrainTrip;                               //前向引用声明
8   class Clock {
9       private:
10        int H,M,S;
11      public:
12        void ShowTime()
13        {
14           cout<<H<<":"<<M<<":"<<S<<endl;
15        }
16        void SetTime(int H=0,int M=0,int S=0)
17        {
18            this->H=H, this->M=M, this->S=S;
19        }

20        Clock(int H=0,int M=0,int S=0)
21        {
22           this->H=H, this->M=M, this->S=S;
23        }
24        friend class TrainTrip;
25    };
26  class TrainTrip {
27      private:
28         char *TrainNo;                            //车次
29         Clock StartTime;                          //出发时间
30         Clock EndTime;                            //到达时间
31      public:
```

```
32          TrainTrip(char * TrainNo, Clock S, Clock E)
33          {
34              this->TrainNo = TrainNo;
35              StartTime = S;
36              EndTime = E;
37          }
38          Clock TripTime()
39          {
40              int tH,tM,tS;                                    //临时存储小时、分、秒数
41              int carry = 0;                                   //借位
42              Clock tTime;                                     //临时存储时间
43             (tS = EndTime.S - StartTime.S)> 0?carry = 0:tS += 60,carry = 1;
44             (tM = EndTime.M - StartTime.M - carry)> 0?carry = 0:tM += 60,carry = 1;
45             (tH = EndTime.H - StartTime.H - carry)> 0?carry = 0:tH += 24;
46             tTime.SetTime(tH,tM,tS);
47             return tTime;
48         }
49  };
50  int main()
51  {
52      Clock C1(8,10,10), C2(6,1,2);                            //定义 Clock 类的对象
53      Clock C3;                                                //定义 Clock 类对象,存储结果
54      TrainTrip T1("K16",C1,C2);                               //定义 TrainTrip 对象
55      C3 = T1.TripTime();
56      C3.ShowTime();
57      return 0;
58  }
```

运行结果：

```
21:50:52
```

友元关系具有以下性质：

(1) 友元关系是不能传递的，B 类是 A 类的友元，C 类是 B 类的友元，在 C 类和 A 类之间，如果没有声明，就没有任何友元关系，就不能进行数据共享。

(2) 友元关系是单向的，如果声明 B 类是 A 类的友元，B 类的成员函数就可以访问 A 类的私有和保护数据，但 A 类的成员函数不能访问 B 类的私有和保护数据。

友元概念的引入提高了数据的共享性，加强了函数与函数之间、类与类之间的相互联系，大大提高了程序的效率，这是友元的优点，但友元也破坏了数据隐蔽和数据封装，导致程序的可维护性变差，给程序的重用和扩充埋下了深深的隐患，这是友元的缺点。

因此，在使用友元时必须慎重，要具体问题具体分析，要在提高效率和增加共享之间把握好一个“度”，要在共享和封装之间进行恰当的平衡。

5.6　常成员与常对象

由于常量是不可改变的，因此我们将“常”广泛用在 C++ 中表示不可改变的量，例如前面讲的常变量。不仅变量可以定义为常变量，函数的形参、类的成员、对象也可以使用 const 定义为“常”，以便对实参、类成员、类进行保护。

5.6.1 函数实参的保护

函数的形参如果用 const 修饰，在函数体中该形参为只读变量，在函数体中不能对该形参变量进行修改。

如果形参变量是一个基本类型的变量，实参采用传值方式进行参数传递，无论在函数体中是否对形参变量进行了修改，都不会影响到实参。此时，const 修饰形参的意义仅仅表明该形参是一个读入值，在函数体内没有对其进行修改。

但是，如果形参变量是指针型或引用型，参数传递是传地址与“传名”方式，函数体对实参的修改会影响到实参，为了对实参进行保护，需要用 const 对实参进行修饰。C++中大量系统函数的形参均用 const 进行修饰。例如

```
int atoi( const char * str );                                    //将数字串转换成整数
char * strcpy( char * to, const char * from );
                                                  //将源字符串 from 复制到目标串 to 中,保护源串 from
string& insert( size_type index, const string& str );
                                                    //在当前串的 index 位置插入串 str, 保护串 str
```

在自己设计的程序中，要养成将只读形参用 const 修饰的习惯。一方面保护实参，避免错误，使程序健壮；另一方面，增强程序的可读性。

5.6.2 常对象

在程序中，我们有时候不允许修改某些特定的对象。如果某个对象不允许被修改，则称该对象为**常对象**。在 C++中用关键字 const 来定义常对象。

C++编译器对常对象(const 对象)的使用是极为苛刻的，它不允许常对象调用任何类的成员函数，而且常对象一经定义，在其生存期内不允许改变，否则将导致编译错误。

常对象的定义格式如下：

```
    类型  const  对象名;
或
    const  类型  对象名;
```

例如：

```
Clock const C1(9,9,9);
```

这里，我们就建立类 Clock 的一个 const 对象，并初始化该对象，该对象在程序运行过程中不能被修改。

既然 const 对象不能被任何对象修改，那么它又能否被其他对象访问呢？

C++规定只有类的常成员函数(const 成员函数)才能访问该类的常对象，当然，const 成员函数依然不允许修改常对象。

下列程序段演示了常对象的使用。

```
int main()
{
      const  Clock  C1(9,9,9);                    //定义常对象 C1
      Clock  const  C2(10,10,10);                 //定义常对象 C2
```

```
    Clock  C3(11,11,11);
    //C1 = C3;                                    //错误!C1 为常对象, 不能被赋值
    //C1.ShowTime();                              //错误!C1 为常对象, 不能访问非常成员函数
    C3. ShowTime();
    //C1.SetTime(0,0,0);                          //错误!C1 为常对象, 不能被更新
    return 0;
}
```

程序运行说明：

(1) const 对象不能被赋值,所以必须在定义时由构造函数初始化；

(2) const 对象不能访问非常成员函数,只能访问常成员函数。

若将成员函数改为常成员函数,如下所示：

```
void ShowTime() const
{  cout << H <<":"<< M <<":"<< S << endl;}
```

则 const 对象 C1 能访问常成员函数 ShowTime()。

5.6.3　常数据成员

const 也可以用来限定类的数据成员和成员函数,分别称为类的**常数据成员**和**常成员函数**。在 C++中,常对象、常数据成员、常成员函数的访问和调用各有特别之处。

使用 const 说明的数据成员称为常数据成员。常数据成员的定义与一般常变量的定义方式相同,只是它的定义必须出现在类体中。

常数据成员同样必须进行初始化,并且不能被更新,但常数据成员的初始化只能通过构造函数的初始化列表进行。

常数据成员定义的格式如下：

```
    数据类型  const  数据成员名;
或
    const 数据类型  数据成员名;
```

【例 5-12】 演示常数据成员的使用。

```
1   /*********************************************
2    *   程序名: p5_12.cpp                        *
3    *   功   能: 演示常数据成员的使用            *
4    *********************************************/
5   #include<iostream>
6   using namespace std;
7   class A {
8   private:
9       const int& r;                       //常引用数据成员
10      const int a;                        //常数据成员
11      static const int b;                 //静态常数据成员
12  public:
13      A(int i):a(i),r(a)                  //常数据成员只能通过初始化列表获得初值
14      {
15          cout<<"constructor!"<<endl;
```

```
16      }
17      void display()
18      {
19          cout << a <<","<< b <<","<< r << endl;
20      }
21  };
22  const int A::b = 3;                          //静态常数据成员在类外说明和初始化
23  int main()
24  {   A a1(1);
25      a1.display();
26      A a2(2);
27      a2.display();
28      return 0;
29  }
```

运行结果：

```
constructor!
1,3,1
constructor!
2,3,2
```

程序分析：

程序中定义了 3 个常数据成员，分别为常数据成员 a、静态常数据成员 b 以及常引用数据成员 r，对于静态常数据成员 b 在类外初始化其值为 3，对于常数据成员 a 和常引用数据成员 r 则通过构造函数采用初始化列表予以初始化其值。

5.6.4 常成员函数

在定义时使用 const 关键字修饰的用于访问类的常对象的函数称为**常成员函数**。常成员函数的说明格式如下：

```
返回类型  成员函数名  (参数表) const;
```

在定义和使用常成员函数时要注意下面几点：

(1) const 是函数类型的一个组成部分，因此在函数实现部分也要带有 const 关键字。

(2) 常成员函数不能更新对象的数据成员，也不能调用该类中没有用 const 修饰的成员函数。

(3) 常对象只能调用它的常成员函数，不能调用其他成员函数，这是 C++从语法机制上对常对象的保护，也是常对象唯一的对外接口方式。

成员函数与对象之间的操作关系如表 5-3 所示。

表 5-3 成员函数与对象之间的操作关系

对象 / 函数	常 对 象	一 般 对 象
常成员函数	√	√
一般成员函数	×	√

(4) const 关键字可以用于参与重载函数的区分。例如：

```
void Print();
void Print() const;
```

这两个函数可以用于重载。重载的原则是，常对象调用常成员函数，一般对象调用一般成员函数。

例如：我们可以定义一个日期类 Date，通过常成员函数读出年、月、日。

```
class Date
{
    private:
      int  Y, M, D;
    public:
      int year() const;
      int month() const;
      int day() const    {return D;    };
      int day(){return D++;}
      int AddYear(int i)  {return Y + i;  };
};
```

有下列容易出现的错误：

```
① int Date::month()                          //错误：常成员函数的实现不能缺少 const
  {
      return M;
  }

② int Date::year()const
  {
      //return Y++;                            //错误：常成员函数不能更新类的数据成员
      return Y;
  }

③ Date const d1;
  //int j = d1.AddYear(10);                    //错误：常对象不能调用非常成员函数
  int j =  d1. year();                         //正确

④ Date d2;
  int i = d2.year();                           //正确，非常对象可以调用常成员函数
  d2.day();                                    //正确，非常对象可以调用非常成员函数
```

常对象和常成员概念的建立明确规定了程序中各种对象的变与不变的界线，从而进一步增强了 C++程序的安全性和可控性。

5.7　对象的内存分布

类只是一个型，除了静态数据成员外，在没有实例化成对象前不占任何内存。类的静态数据成员与全局对象(变量)一样，在数据段中分配内存。

5.7.1　对象的内存空间的分配

当类被实例化成对象后，不同类别的对象占据不同类型的内存，其规律与普通变量相同：

（1）建立的全局对象占有数据段的内存。

（2）建立的局部对象内存分配在栈中。

（3）函数调用时为实参建立的临时对象内存分配在栈中。

（4）使用动态内存分配语句 new 建立的动态对象内存在堆中分配。

虽然我们说类（对象）是由数据成员和成员函数组成的，但是程序运行时，系统只为各对象的数据成员分配单独的内存空间，该类的所有对象共享类的成员函数定义以及为成员函数分配的空间。对象的内存空间分配有下列规则：

（1）对象的数据成员与成员函数占据不同的内存空间，数据成员的内存空间与对象的存储类别相关，成员函数的内存空间在代码段中。

（2）一个类所有对象的数据成员拥有各自的内存空间。

（3）一个类所有对象的成员函数为该类的所有对象共享，在内存中只有一个副本。

5.7.2 对象的内存空间的释放

随着对象的生命周期的结束，对象所占的空间会释放，各类对象内存空间释放的时间和方法如下：

（1）全局对象数据成员占有的内存空间在程序结束时释放。

（2）局部对象与实参对象数据成员的内存空间在函数调用结束时释放。

（3）动态对象数据成员的内存空间要使用 delete 语句释放。

（4）对象的成员函数的内存空间在该类的所有对象生命周期结束时自动释放。

5.8 本章小结

（1）在面向对象的程序设计中，程序模块是由类构成的。类是对逻辑上相关的函数与数据的封装，它是对问题的抽象描述。

（2）类中有数据成员与成员函数，成员的访问控制属性有 private、protected、public。在类内可以访问所有控制属性的成员，在类外通过对象只能访问控制属性为 public 的成员。

（3）类是“型”，是“虚”的，不占内存，在使用类建立对象后，对象是“实”的，占有内存空间。

（4）在建立对象时调用构造函数初始化对象的数据成员，一个类提供默认的构造函数与默认的拷贝构造函数。默认的构造函数是空的，默认的拷贝构造函数的内容为浅拷贝语句。在对象消失时会调用析构函数。构造函数与析构函数都可以重新定义。

（5）在默认的拷贝构造函数中，拷贝的策略是直接将原对象的数据成员值依次拷贝给新对象中对应的数据成员，这种方式为浅拷贝。深拷贝能将原对象指针指向的内容拷贝给新对象中。

（6）建立对象指针、对象引用均没有建立对象，所以此时不调用构造函数。通过对象指针使用对象的成员要用操作符->，通过对象引用使用对象的成员要用操作符。

（7）对象数组是以对象为元素的数组，对象数组的定义、赋值、引用与普通数组一样，建立一个对象数组相当于建立了多个对象，因此多次调用构造函数。对象数组的初始化需要使用构造函数完成。

（8）建立动态对象使用语句 new，动态对象一定要用语句 delete 删除。建立动态对象数组使用语句 new[]，删除动态对象数组使用语句 delete[]。

(9) this 指针是一个系统预定义的指向当前对象的指针，通过 this 指针可以访问对象的成员。

(10) 组合类是含有类对象的类，组合类对象称为组合对象。在定义组合对象时调用构造函数的顺序为类中成员对象定义的顺序，子对象在构造时初始值通过组合类构造函数的成员对象初始化表提供。

(11) 静态数据成员是类的数据成员，独立于类存在，在类内定义，在类外初始化。静态成员函数属于整个类，是该类的所有对象共享的成员函数，可通过类名、对象调用静态成员函数访问静态函数成员。

(12) 友元函数不是类的成员，但它可以访问类的任何成员。一个类的友元类可以访问该类的任何成员。

(13) 使用关键字 const 定义的对象称为常对象，常对象的成员不允许被修改。使用 const 定义的数据成员称为常数据成员，常数据成员不能被更新。在定义时使用 const 关键字修饰的成员函数称为常成员函数，用于访问类的常对象。

(14) 各类对象在内存中的分布以及生命周期与普通变量一样，一个类的所有对象共有该类的成员函数，独享各自的数据成员。

习　题　5

1. 填空题

(1) 类的_____只能被该类的成员函数或友元函数访问。

(2) 类的数据成员不能在定义的时候初始化，而应该通过_____初始化。

(3) 类成员默认的访问方式是_____。

(4) 类的_____是该类给外界提供的接口。

(5) 类的_____可以被类作用域内的任何对象访问。

(6) 为了能够访问某个类的私有成员，必须在该类中声明该类的_____。

(7) _____为该类的所有对象共享。

(8) 每个对象都有一个指向自身的指针，称为_____指针，通过使用它来确定其自身的地址。

(9) 运算符_____自动建立一个大小合适的对象并返回一个具有正确类型的指针。

(10) C++禁止_____访问 const 对象。

(11) 在定义类的动态对象数组时，系统自动调用该类的_____函数对其进行初始化。

(12) C++中语句“_____ p="hello";”所定义的指针 *p* 和它所指的内容都不能被改变。

(13) 假定 AB 为一个类，则语句“_____”为该类拷贝构造函数的原型说明。

(14) 在 C++中，访问一个对象的成员所用的运算符是_____，访问一个指针所指向对象的成员所用的运算符是_____。

(15) 析构函数在对象的_____时被自动调用，全局对象和静态对象的析构函数在_____调用。

(16) 设 *p* 是指向一个类动态对象的指针变量，则执行“delete p;”语句时将自动调用该类的_____。

2. 选择题

(1) 数据封装就是将一组数据和与这组数据有关的操作组装在一起，形成一个实体，这个实体也就是(　　)。

A. 类　　B. 对象　　C. 函数体　　D. 数据块

(2) 类的实例化是指(　　)。

A. 定义类　　B. 创建类的对象　　C. 指明具体类　　D. 调用类的成员

(3) 已知 p 是一个指向类 Sample 数据成员 m 的指针，s 是类 Sample 中的一个对象。如果要给 m 赋值为 5，正确的是(　　)。

A. S.P=5;　　B. S->P=5;　　C. S.*P=5;　　D. *S.P=5;

(4) 关于类和对象的说法不正确的是(　　)。

A. 对象是类的一个实例

B. 一个类只能有一个对象

C. 一个对象只能属于一个具体的类

D. 类与对象的关系和数据类型与变量的关系是相似的

(5) 下列说法错误的是(　　)。

A. 封装是将一组数据和这组数据有关的操作组装在一起

B. 封装使对象之间不需要确定的接口

C. 封装要求对象具有明确的功能

D. 封装使得一个对象可以像一个部件一样用在各种程序中

(6) 下面说法正确的是(　　)。

A. 内联函数在运行时是将该函数的目标代码插入每个调用该函数的地方

B. 内联函数在编译时是将该函数的目标代码插入每个调用该函数的地方

C. 类的内联函数只能在类体内定义

D. 类的内联函数只能在类体外通过加关键字 inline 定义

(7) 下列说法正确的是(　　)。

A. 类定义中只能说明函数成员的函数头，不能定义函数体

B. 类中的函数成员可以在类体中定义，也可以在类体之外定义

C. 类中的函数成员在类体之外定义时必须要与类声明在同一个文件中

D. 在类体之外定义的函数成员不能操作该类的私有数据成员

(8) 下面关于对象概念的描述错误的是(　　)。

A. 对象就是 C 语言中的结构体变量

B. 对象代表着正在创建的系统中的一个实体

C. 对象是一个状态和操作(或方法)的封装体

D. 对象之间的信息传递是通过消息进行的

(9) 在建立类的对象时(　　)。

A. 为每个对象分配用于保存数据成员的内存

B. 为每个对象分配用于保存函数成员的内存

C. 为所有对象的数据成员和函数成员分配一个共享的内存

D. 为每个对象的数据成员和函数成员同时分配不同内存

(10) 有以下类定义：

```
class SAMPLE
{
    int n;
  public:
    SAMPLE(int i = 0):n(i){}
    void setValue(int n0);
};
```

下列关于 getValue 成员函数的实现正确的是(　　)。

A. SAMPLE::setValue(int n0){n=n0;}

B. void SAMPLE::setValue(int n0){n=n0;}

C. void setValue(int n0){n=n0;}

D. setValue(int n0){n=n0;}

(11) 在下面的类定义中,错误的语句是(　　)。

```
class sample
{
  public:
      sample(int val);          //①
      ~sample();                //②
  private:
      int a = 2.5;              //③
      sample();                 //④
}
```

A. ①②③④　　B. ②　　C. ③　　D. ①②③

(12) 对于任意一个类,析构函数的个数最多为(　　)。

A. 0　　B. 1　　C. 2　　D. 3

(13) 类的构造函数被自动调用执行的情况是在定义该类的(　　)时。

A. 成员函数　　B. 数据成员　　C. 对象　　D. 友元函数

(14) 有关构造函数的说法不正确的是(　　)。

A. 构造函数的名字和类的名字一样

B. 构造函数在声明类变量时自动执行

C. 构造函数无任何函数类型

D. 构造函数有且只有一个

(15) (　　)是析构函数的特征。

A. 一个类中只能定义一个析构函数

B. 析构函数名与类名没任何关系

C. 析构函数的定义只能在类体内

D. 析构函数可以有一个或多个参数

(16) 下列(　　)不是构造函数的特征。

A. 构造函数的函数名和类名相同

B. 构造函数可以重载

C. 构造函数可以设置默认参数

D. 构造函数必须指定类型说明

(17) 在下列函数原型中，可以作为类 AA 构造函数的是(　　)。

A. void AA(int)；　B. int AA( )；　　C. AA(int)const；　D. AA(int)

(18) 下列关于成员函数特征的描述，(　　)是错误的。

A. 成员函数一定是内联函数

B. 成员函数可以重载

C. 成员函数可以设置参数的默认值

D. 成员函数可以是静态的

(19) 不属于成员函数的是(　　)。

A. 静态成员函数　B. 友元函数　　C. 构造函数　　D. 析构函数

(20) 已知类 A 是类 B 的友元，类 B 是类 C 的友元，则(　　)。

A. 类 A 一定是类 C 的友元

B. 类 C 一定是类 A 的友元

C. 类 C 的成员函数可以访问类 B 的对象的任何成员

D. 类 A 的成员函数可以访问类 B 的对象的任何成员

(21) 关于动态存储分配，下列说法正确的是(　　)。

A. new 和 delete 是 C++中用于动态内存分配和释放的函数

B. 动态分配的内存空间也可以被初始化

C. 当系统内存不够时，会自动回收不再使用的内存单元，因此程序中不必用 delete 释放内存空间

D. 当动态分配内存失败时，系统会立刻崩溃，因此一定用慎用 new

(22) 静态成员函数没有(　　)。

A. 返回值　　B. this 指针　　C. 指针参数　　D. 返回类型

(23) 有以下类定义：

```
class Foo { int bar; };
```

则 Foo 类的成员 bar 是(　　)。

A. 公有数据成员　B. 公有成员函数　C. 私有数据成员　D. 私有成员函数

(24) 下列关于 this 指针的叙述正确的是(　　)。

A. 任何与类相关的函数都有 this 指针

B. 类的成员函数都有 this 指针

C. 类的友元函数都有 this 指针

D. 类的非静态成员函数才有 this 指针

(25) 下列程序的执行结果是(　　)。

```
#include <iostream>
using namespace std;
class Test {
public:
    Test()     { n+=2; }
    ~Test()    { n-=3; }
    static int getNum() { return n; }
private:
```

```
    static int n;
};
int Test::n = 1;
int main()
{
    Test * p = new Test;
    delete p;
    cout << "n = " << Test::getNum() << endl;
 return 0;
}
```

A. $n=0$　　B. $n=1$　　C. $n=2$　　D. $n=3$

(26) 下列程序执行后的输出结果是(　　)。

```
#include <iostream>
using namespace std;
class AA{
    int n;
public:
    AA(int k):n(k){}
    int get(){ return n;}
    int get()const{ return n + 1;}
};
int main()
{
    AA a(5);
    const AA b(6);
    cout << a.get()<< b.get();
    return 0;
}
```

A. 55　　B. 57　　C. 75　　D. 77

(27) 由于常对象不能被更新,因此(　　)。

A. 通过常对象只能调用它的常成员函数

B. 通过常对象只能调用静态成员函数

C. 常对象的成员都是常成员

D. 通过常对象可以调用任何不改变对象值的成员函数

(28) 有以下类定义:

```
class AA
{
    int a;
public:
    int getRef()const{return &a;}                    //①
    int getValue()const{return a;}                   //②
    void set(int n)const{a = n;}                     //③
    friend void show(AA aa)const{cout << a;}         //④
};
```

其中的 4 个函数定义中正确的是(　　)。

A. ①　　B. ②　　C. ③　　D. ④

(29) 有以下类定义：

```
class Point
      {
    int x_,y_;
public:
    Point():x_(0),y_(0){}
    Point(int x,int y = 0):x_(x),y_(y){}
};
```

若执行语句：

```
Point a(2.,b[3],*c[4];
```

则 Point 类的构造函数被调用的次数是(　　)。

A. 两次　　B. 3 次　　C. 4 次　　D. 5 次

(30) 有以下类定义：

```
class Test
{
  public:
      Test(){a = 0;c = 0;}                          //①
      int f(int a)const{ this->a = a;}              //②
      static int g(){ return a;}                    //③
      void h(int b){Test::b = b;};                  //④
  private:
          int a;
          static int b;
          const int c;
};
int Test::b = 0;
```

在标注号码的行中，能被正确编译的是(　　)。

A. ①　　B. ②　　C. ③　　D. ④

(31) 若有以下类声明：

```
class MyClass
{
 public:
    MyClass(){   cout<<1;     }
};
```

执行下列语句：

```
MyClass a,b[2],*P[2];
```

程序的输出结果是(　　)。

A. 11　　B. 111　　C. 1111　　D. 11111

(32) 有以下程序：

```
#include<iostream>
using namespace std;
class A
{
```

```
  public:
      static int a;
      void init(){a = 1;}
      A(int a = 2)
      {
        init();
        a++;
    }
};
int A::a = 0;
A obj;
int main()
{
   cout << obj.a;
   return 0;
}
```

运行时输出的结果是(　　)。

A. 0　　B. 1　　C. 2　　D. 3

(33) 有以下程序：

```
#include <iostream>
using namespace std;
class MyClass
{
  public:
    MyClass(){cout <<"A";}
    MyClass(char c)
    { cout << c; }
~MyClass(){cout <<"B";}
};
int main()
{
    MyClass p1, * p2;
    p2 = new MyClass('X');
    delete p2;
    return 0;
}
```

执行这个程序，计算机屏幕上将显示输出(　　)。

A. ABX　　B. ABXB　　C. AXB　　D. AXBB

3. 简答题

(1) C++中的空类默认产生哪些类成员函数？

(2) 类和数据类型有何关联？

(3) 类和对象的内存分配关系如何？

(4) 什么是浅拷贝？什么是深拷贝？二者有何异同？

(5) 什么是 this 指针？它的作用是什么？

(6) C++中的静态成员有何作用？它有何特点？

(7) 友元关系有何性质？

（8）在 C++程序设计中，友元关系有什么优点和缺点？

（9）如何实现不同对象的内存空间的分配和释放？

4. 程序填空题

（1）在下面横线处填上适当字句，完成类中成员函数的定义。

```
class A{
      int * a;
     public:
      A(int aa = 0) {
            a = __①__;                    //用 aa 初始化 a 指向的动态对象
      }
      ~A(){ __②__;};                      //释放动态存储空间
```

（2）

```
class A{
    __①__
    int n;
  public:
    A(int nn = 0):n(nn){
      if(n == 0)a = 0;
      else a = new int[n];
    }
    __②__                                 //定义析构函数，释放动态数组空间
};
```

（3）

```
class Location {
      private:
        int X,Y;
      public:
        void init(int initX,int initY) {
          X = initX,Y = initY;
      }
    int  GetX() {
        reutrn X;
    }
    int  GetY(){
     reutrn Y;
    }
};
int main()
{
    Location A1; A1.init(20,90);
    __①__                                 //定义一个指向 A1 的引用 rA1
    __②__                                 //用 rA1 在屏幕上输出对象 A1 的数据成员 X 和 Y 的值
          return 0;
}
```

5. 程序分析题(写出运行结果)

(1)

```
#include<iostream>
using namespace std;
class MyClass {
    public:
          int number;
        void set(int i);
    };
    int number=3;
    void MyClass::set (int i)
    {
        number=i;
    }
    int main()
    {
        MyClass my1;
        int number=10;
        my1.set(5);
        cout<<my1.number<<endl;
        my1.set(number);
        cout<<my1.number<<endl;
        my1.set(::number);
        cout<<my1.number;
    }
```

(2)

```
#include<iostream>
using namespace std;
class Location{
      public:
          int X,Y;
    void init (int initX,int initY)
     {
      X=initX, Y=initY;
     }
    int GetX()
     {
       return X;
     }
    int GetY()
     {
       return Y;
    }
};
void display(Location& rL)
{
    cout<<rL. GetX()<<"   "<<rL.GetY()<<"\n";
}
int main()
{
```

```
    Location A[5] = {{0,0},{1,1},{2,2},{3,3},{4,4}};
    Location * rA = A;
    A[3].init(5,3);
    rA -> init(7,8);
    for (int i = 0;i < 5;i++)
         display( * (rA++));
    return 0;
}
```

(3)

```
# include < iostream >
using namespace std;
 class Test{
    private:
         static int val;
         int a;
    public:
         static int func();
         void sfunc(Test &r);
};
int Test::val = 200;
int Test::func()
{
    return val++;
}
void Test::sfunc(Test &r)
{
    r.a = 125;
    cout <<"            Result3 = "<< r.a;
}
int main()
{
    cout <<"Result1 = "<< Test::func()<< endl;
    Test a;
    cout <<"Result2 = "<< a.func();
    a.sfunc(a);
    return 0;
}
```

(4)

```
# include < iostream >
using namespace std;
class Con
{
    char ID;
public:
    char getID()const{return ID;}
    Con():ID('A') {cout << 1;}
    Con(char ID):ID(ID){cout << 2;}
    Con(Con& c):ID(c.getID()) {cout << 3;}
```

```
};
void show(Con c)
{cout << c.getID();}
int main()
{
    Con c1;
    show(c1);
    Con c2('B');
    show(c2);
    return 0;
}
```

6. 改错题

(1) 下面的程序段有多处错误,说明错误原因并改正错误。

```
Class A {
    int a(0),b(0);
public:
    A(int aa,int bb) {a = aa;b = bb;}
}
A   x(2,3), y(4);
```

(2) 下面的程序有一处错误,请用下横线标出错误所在行并改正错误。

```
class Test{
    public;
      static int x;
};
int x = 20;                                    //对类成员初始化
int main()
{
    cout << Test::x;
    return 0;
}
```

(3) 用下横线标出下面程序 main()函数中的错误行,并说明错误原因。

```
#include <iostream>
using namespace std;
class Location{
    private:
        int X,Y;
    public:
        void init(int initX,int initY) {
            X = initX;
            Y = initY;
            }
        int sumXY() {
            return X + Y;
        }
};
int main()
{
      Location A1;
```

```
    int x,y;
    A1.init(5,3);
    x = A1.X;y = A1.Y;
    cout << x + y <<"   "<< A1.sumXY()<< endl;
    return 0;
}
```

(4) 指出下面程序中的错误,并说明出错原因。

```
#include <iostream>
using namespace std;
class ConstFun{
        public:
            void ConstFun(){}
            const int f5()const{return 5;}
            int Obj() {return 45;}
            int val;
            int f8();
    };
    int ConstFun::f8(){return val;}
    int main()
    {
            const ConstFun s;
            int i = s.f5();
            cout <<"Value = "<< endl;
            return 0;
    }
```

7. 编程题

(1) 定义一个三角形类 Ctriangle,求三角形的面积和周长。

(2) 定义一个点类 Point,并定义成员函数 double Distance(const& Point)求两点的距离。

(3) 定义一个日期类 CData,它能表示年、月、日。设计一个 NewDay()成员函数,增加一天日期。

(4) 定义一个时钟类 Clock,设计成员函数 SetAlarm (int hour, int minute, int second)设置响铃时间；用 run()成员函数模拟时钟运行,当运行到响铃时间时提示响铃。

(5) 设计一个学生类,包含学生学号、姓名、课程、成绩等基本信息,计算学生的平均成绩。

(6) 有一个信息管理系统,要求检查每一个登录系统的用户(User)的用户名和口令,系统检查合格以后方可登录系统,用 C++程序予以描述。

(7) 定义一个字符串类 String,增加下列成员函数。

- bool IsSubstring(const char * str)：判断 str 是否为当前串的子串；
- bool IsSubstring(const String & Str)：判断 str 是否为当前串的子串；
- int str2num()：把数字串转换成数；
- void toUppercase()：把字符串转换成大写字母。

(8) 定义一个元素类型为 int、元素个数不受限制的集合类 Set。除了定义一些必要的函数外,还定义具有下列功能的成员函数。

- bool IsEmpt()：判断集合是否为空；
- int size()：返回元素个数；

- bool IsElement(int e) const：判断 e 是否属于集合；
- bool IsSubset(const Set& s) const：判断 s 是否包含于集合；
- bool IsEqual(const Set& s) const：判断集合是否相等；
- Set& insert(int e)：将 e 加入到集合中；
- Set union(const Set& s) const：求集合的并；
- Set intersection(const Set& s) const：求集合的交；
- Set difference(const Set& s) const：求集合的差。