第10章　C++入门

90年代以来，面向对象的程序设计（Object-Oriented Programming，简称OOP）异军突起，迅速在全世界流行，一跃成为主流的程序设计技术。在软件市场中，覆盖面大、垄断市场的新一代程序设计语言、软件开发工具和环境以及操作系统大多是面向对象的。

10.1　面向对象的概念

10.1.1　面向对象的程序结构

面向对象的程序设计是一种基于结构分析的、以数据为中心的程序设计方法。在面向对象的程序中，活动的基本单位是对象，向对象发送消息可以激活对象的行为。为此，许多人把面向对象的程序描述为：

程序=对象+消息传递

1. 对象

对象类似于C语言中的变量，可以泛指自然界中的任何事务，包括具体实物和抽象概念。对象具有一些属性、状态和行为。例如，每个人都有姓名、性别、年龄、身高、体重等属性，有工作、学习、吃饭、睡觉等行为。所以，对象一般可以表示为：属性 +行为。在面向对象的程序设计中，对象被表示为：数据 +操作，操作也称为方法。这就是说，面向对象程序设计中的对象是指由一组数据和作用于其上的一组方法组成的实体。

2. 类

在面向对象的程序设计中，会涉及到许多对象，我们无法将所有的对象都描述清楚，如果那样做的话，程序将无限长或无法描述。因此在面向对象的程序设计中引入了类的概念，将同类的对象归于一类，同类对象具有相同的属性和行为，如各种花同属一类，各种草、植物等也分别属于不同的类。

3. 消息

消息就是对对象进行某种操作的信息。当要求对象执行某种特定操作时，就向该对象发送操作消息，对象接收到指定操作的消息后，就调用相应的操作方法，完成有关操作。

消息及其传递机制是面向对象程序设计的一个重要角色。对象的一切活动，都要通过消息来驱动，消息传递是对象间通信的唯一途径。

4. 方法

方法就是对对象进行的某种操作。当对象接收到相应的消息时，就调用对应的方法完成指定的操作，有了消息，就驱动对象工作；有了方法，就能实现消息所要求的操作。

5. 继承

继承是面向对象语言的另一个重要概念。在客观世界中，存在着整体与个体的关系、一般与特殊的关系，继承将后者模型化。

例如，对人的分类我们用图10-1描述如下。

在类的层次结构图中，下层节点都具有上层节点的特性，都具备人的共同特点。但下层节点较之上层节点而言，又具有新的特性，是上层节点所不具有的。这种下层节点对上层节点的特性的保持，就是我们所说的继承。

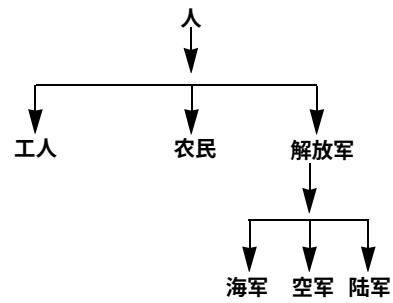

图10-1 对人的分类

在面向对象语言中，类功能支持这种层次结构。除了根结点外，每个类都有它的超类，又称为父类或基类。除了叶结点外，每个类都有它的子类，又称为派生类。一个子类可以从它的基类继承所有的数据和操作，并扩充自己的特殊数据和操作。基类抽象出共同的属性和操作，子类体现其差别。有了类的层次结构和继承性，不同对象的共同特性只需定义一次，用户就可以充分利用已有的类，进行完善和扩充，达到软件可重用的目的。

10.1.2 C++的类

C++语言是一种面向对象的程序设计语言，是对传统 C语言的完善和扩充，并支持面向对象的概念：对象、类、方法、消息和继承，下面给出一个 C++关于类的结构：

[例10-1] 栈操作。栈是一种后进先出的数据结构，我们利用数组这个静态的存储来实现栈、充当栈，完成数据的压栈和出栈。

```
#include  "iostream.h"
#define  SIZE  100
//定义栈类型
class  stack                    /*定义类*/
{
  int  stck[SIZE];              /*数据成员，整型数组做堆栈*/
  int  top;                     /*栈顶指针*/
public:
  void  init(void);             /*初始化成员函数*/
  void  push(int  i);           /*压栈*/
  int  pop(void);               /*出栈*/
};
//栈的初始化
void  stack::init(void)         /*类成员函数的定义*/
{
top=0;                          /*定义栈顶指针指向数组头*/
}
//进栈操作
void  stack::push(int  i)
{
  if  (top= =SIZE)
 {
  cout<<"The  stack  is  full!";    /*栈满*/
  return;
```

```
    }
   stck[top]=i;                /* 压入数据到栈顶*/
   top++;                      /*指针加1*/
}
//出栈操作
int  stack::pop(void)
{
  if  (top= =0)
 {
 cout<<"The  stack  is  underflow!";/*栈空，无数据*/
  return 0;
 }
  top- -;          /*指针减1*/
  return  stck[top];           /*返回栈顶元素*/
}
void  main(void)
{
stack   stack1,  stack2;          //创建对象，
stack1.init();               /*调用成员函数，对栈初始化*/
stack2.init();
stack1.push(1);      //在stack1 栈中，压栈1。
stack2.push(2);      /在stack2 栈中，压栈2。
stack1.push(3);
stack2.push(4);
cout<<stack1.pop()<<"  ";  输出stack1 栈顶数据，即弹出
cout<<stack2.pop()<<"  ";
cout<<stack1.pop()<<"  ";
cout<<stack2.pop()<<"\n ";
}
```

在该程序中，定义了一个对象类型 stack，我们称为类。类 stack由数据成员（变量 stck，top）和操作成员函数（函数 init，push，pop）两部分构成，这种结构类似于 C语言中的结构体类型。不同的是，将C中结构体类型的描述符 struct 替换成了描述符class。stack就是类的类型标识符。类stack将全部信息（数据或变量、方法或函数）都封装在其自身之内，这是 C++语言的最基本的特征。有了类（对象类型） stack，我们就可以定义对象（对象变量），如程序中定义的stack1和stack2，并在对象stack1和stack2之上进行各种不同的操作（初始化、进栈、出栈）。

上述程序运行后输出为： 3 4 1 2

10.2 C++的输入与输出

在C++的程序设计中，除继续使用 C语言中的标准库函数（如 printf，scanf）进行输入输出外，C++还提供自己的输入输出方式。

[例10-2] 显示一行输出。

```
#include <iostream.h>
main()
```

```
{
cout<<"Hello  ，World !";
}
```

C++标准流的输入输出可以在iostream.h文件中找到。cout是C++中与标准输出设备相关的输出流，“ << ” 是运算符，该运算符完成将引号内的字符串写到标准输出流 cout，简单地说，就是将字符串“ Hello，World! ”写到标准输出设备 —— 显示器上，为此，运行程序，我们将在屏幕上看到：

```
Hello ，World!
```

[例10-3] 利用海伦公式，输入三角形的三条边，若满足任意两边之和大于第三边，则计算出给定三角形的面积。

```
#include <iostream.h>
#include <math.h>
main()
{
float s，s1，a1，a2，a3; // a1，a2，a3是三角形的三条边；s 是三角形的面积；
                                                    // s1是二分之一的周长
cout<<"a1=";
cin>>a1;            /* 键盘输入 */
cout<<"a2=";
cin>>a2;
cout<<"a3=";
cin>>a3;
if (((a1+a2)>a3)&&((a1+a3)>a2)&&((a2+a3)>a1))
//任意两边之和应大于第三边
   {
   s1=(a1+a2+a3)/2;
   s=sqrt(s1*(s1-a1)*(s1-a2)*(s1-a3));              //计算面积
   cout<<"area="<<s;
   cout<<"\n";
   }
   else
   cout<<"error!";
   return 0;
   }
```

程序中，cin是C++的标准输入流；“ >> ”是运算符，该运算符完成从标准输入流（通常cin与键盘相连）中读取数据。运行该程序后，可以看到如下的显示信息：

```
a1=3_
a2=4_
a3=5_
area=6
```

10.3 类与对象

客观世界中的事物都包含属性和行为两个方面。在C++程序设计中，对事物的描述分别用数据成员和成员函数来表现，并把它们封装起来，形成一个抽象的数据类型 —— 类。这就是说，类

具有两种成员：数据成员和成员函数，按照面向对象的概念，成员函数又称为方法函数或方法。

10.3.1 类的定义与对象的引用

1. 类的定义

类定义的基本格式如下所示：

```
class 类型名
{
    private:
            私有成员声明；
    protected:
            保护成员声明；
    public:
            公有成员声明；
}
```

类成员分为私有成员和公有成员两部分。外界不能访问一个对象的私有成员，只能与对象的公有成员之间进行信息交换。定义类即是确定选择成员并区分它们的访问权限。

[例10-4] C++程序结构示例。

```
#include <stdio.h>
class exam1      //定义类                                          ┐
{                                                                 │
    private:                        //类的私有成员               │
        int x, y;                   //数据成员                   │
    public:                         //类的公有成员               │ 类的声明部分
         void init();               //成员函数                   │
         float average();                                         │
         void print();                                            │
};                                                                ┘
      void exam1::init()            //类成员函数的定义           ┐
      {                                                          │
          x=3;                                                   │
          y=4;                                                   │
      }                                                          │
   float exam1::average()           //类成员函数的定义           │ 类的实现部分
   {return (x+y)/2.0;                                            │
     }                                                           │
   void exam1::print()              //类成员函数的定义           │
   {                                                            │
    printf("\nx=%d  , y=%d , aver=%7.2f\n"  , x, y, average());  │
    }                                                           ┘
       main()                       //主函数                     ┐
       {                                                        │
            exam1 obj;                      //声明并创建一个对象 │
            obj.init();                     //调用成员函数初始化 │ 类的使用
            obj.print();            //输出运算结果               │
            return 0;                                           │
      }                                                         ┘
```

1) 类在C++中用关键字class来说明，紧跟其后的是类名exam1，类中包含两个私有数据成员x、y和三个公有的成员函数init()、average()、print()。

C++允许隐藏内部状态，由private开始的私有段成员就满足这一特性，它们只能允许该类对象的成员函数来访问。类中的公有段既可以是数据成员，也可以是成员函数，这是类提供给外部的接口。当然，C++还提供另一种保护段符号：protected，下面会介绍到。

运行该程序，得到的输出显示为：

```
x=3 ， y=4 ，      aver=  3.50
```

2) 类的成员在类的定义中出现的顺序可以任意，并且类的实现既可以放在类的外面，又可以内嵌在类内，下面调整类成员的顺序为：

```
class exam1      //定义类
{
   public:
          float average();
          void print();
   private:
        int x, y;
   public:
         void init();
    };
```

3) 若类的实现定义在类的外面，在成员函数的函数头中函数名前，应使用作用域限定符::指明该函数是哪一个类中的成员函数，即有如下格式。

类型 类名::成员函数名（参数表）

```
{
   函数体
}
```

4) 除特殊指明外，成员函数操作的是同一个对象的数据成员。下面的示例将类成员函数的实现内嵌在类中：

```
class exam1       //定义类
{
    public:
          float average()
          {
                return (x+y)/2.0;
          }
           void print()
                          {
                          printf("\nx=%d   , y=%d , aver=%7.2f\n"   , x, y, average());
                          }
   private:
        int x, y;
   public:
         void init()
                          {
                                  x=3;
                                  y=4;
                                  }
    };
```

5) 类定义的最后一个花括号的外面一定要有分号结束。程序中出现的“ //”符号是作为注释开始的标志，与C语言中“ /* ..... */”用法完全相同。

6) 使用public、private和protected关键字

public、private和protected关键字称为访问说明符。

说明为public的类成员可以被任何函数所使用（当它们是数据成员时）或调用（当它们是成员函数时）。调用者不必属于这个类或任何类。

说明为private的类成员只能被同类中的成员函数所使用或调用，也可以被同类的友元使用或调用。在类的成员中，若没有额外声明访问权限，则表明为 private的类成员。

说明为protected的类成员只能被同类中的成员函数或同类的友元类，或派生类的成员函数及友元类所使用或调用。

2. 类与对象

上述类exam1提出了两个概念：类与对象。从形式上看，类与对象的关系类似于 C语言中的数据类型与变量的关系，类是将具有相同属性和行为的事物做一个概括，它是普遍意义上的一般概念，而对象是具有类特征的具体事物。一旦定义了类，那么就有无数的具有该属性和行为的对象与之对应。

类在概念上是一种抽象机制，它抽象了一类对象的存储和操作特性；在系统实现中，类是一种共享机制，它提供了一类对象共享其类的操作实现。

类是对象的模板，对象承袭了类中的数据和方法，只是各对象具有的初始化数据不同，所表示的对象状态也不同。

一个C++文件可以作为一个文件存储，其文件的扩展名为“ .cpp”，也可以作为几个文件存储。若作为几个文件存储，一般说来应把类的声明部分存于“ .h”的头文件中，而把类的实现部分和类的使用部分分别存于扩展名为“ .cpp”的文件中。包含主函数的.cpp文件中应包含.h和其它.cpp文件。规模较大的程序，应采用模块化的程序设计技术。

C++程序的编辑、编译、连接及运行的方法和过程，在 DOS下，与C语言基本一样。Borland C++或是Turbo C++，均有一个集成开发环境，易学易用（与 Turbo C大同小异），操作非常方便。

10.3.2 构造函数与析构函数

C++中，类是一种数据类型，这样的类型总与存储空间相关，即要占用一定的内存资源。当我们定义了对象时，编译系统就会为它分配存储，进行一定的初始化，由于类的结构各不相同，所需的工作量也各不相同，为此， C++提供构造函数来完成上述工作。构造函数是属于某一特定类，可由用户设置，也可用系统缺省设置。与之相对应的是类的析构函数，当类的对象退出作用域时，析构函数负责回收存储空间，并做一些必要的善后处理。析构函数也是属于某一特定类，可由用户设置，也可用系统缺省。

1. 构造函数

当定义一个对象时，我们需要给对象开辟一个存储空间，将对象的数据成员初始化。在使用构造函数以前，首先对构造函数作如下说明：

1) 构造函数具有与类名相同的函数名。

2) 构造函数没有返回类型，即使void也不可以。它的返回值是隐含的，是指向类本身的指针。

3) 构造函数在对象被定义时自动调用，作相应的初始化。

4) 构造函数可以有参数，也可无参数。

5) 构造函数名可以重载。

6) 当类中无与类名相同的构造函数时，C++编译系统为其设置缺省的构造函数。

[例10-5] 构造函数应用举例

```
#include <stdio.h>

class     A{
    int a, b, c;                          //缺省访问权限，为私有数据成员
public:                                   //公有段
     A(int=1 , int=2 , int=3);            //构造函数1
     A(double , double , double);         //构造函数2
     A(long);                             //构造函数3
     A(A&);                               //构造函数4（拷贝构造函数）
     void show()                          //公有成员函数
          {
          printf("%d , %d, %d\n" , a, b, c);
          }
    };
    A::A(int I1, int I2, int I3)          //构造函数1的实现
    {
     a=I1; b=I2;   c=I3;
     }
    A::A(double f1, double f2, double f3)    //构造函数2的实现
    {
     a=(int)f1;    b=(int)f2;      c=(int)f3;
     }
    A::A(long n)                          //构造函数3的实现
    {
    a=b=c=(int)n;
    }
    A::A(A& other)                        //构造函数4的实现
          {
          a=other.a;
          b=other.b;
          c=other.c;
          }
   main()
   {
   A     x1;                              //定义对象x1，调用缺省参数的构造函数1
   x1.show();                                    //调用公有段成员函数show()
   A     x2(3);                           //定义对象x2，调用构造函数1
   x2.show();                                    //调用公有段成员函数show()
   A     x3(3, 1);                        //定义对象x3，调用构造函数1
   x3.show();                                    //调用公有段成员函数show()
   A     x4(3.14, 2.414 , 6.28);          //定义对象x4，调用构造函数2
   x4.show();                                    //调用公有段成员函数show()
   A    x5(53L);                          //定义对象x5，调用构造函数3
   x5.show();                                    //调用公有段成员函数show()
```

```
A  x6=x5;                                   //定义对象x6，调用拷贝构造函数4
x6.show();                                  //调用公有段成员函数show()
return 0;
}
```

运行上述程序，得如下输出：

```
1，2，3
3，2，3
3，1，3
3，2，6
53，53，53
53，53，53
```

可以提供不带参数的构造函数，即是缺省的构造函数。例如：

```
class          A{
               ...
               A();
               ...
               };
A::A       ()
               {
a=0;       b=0;    c=0;
               }
```

但要注意的是，不能将可缺省参数的构造函数与缺省的构造函数一起使用，以免编译系统混淆。例如：

```
class                  A{
                       ...
                       A( );
                       A(int=1 ,   int=2 ,   int=3);
                       ...
                       };
void   main( )
{
 A   obj;                          //编译系统无法区分应调用哪一个构造函数
...
}
```

2. 析构函数

与构造函数对应的是析构函数。C++用析构函数来处理对象的善后工作，在对象撤销时自动调用，并可能要释放一些动态的存储空间等。析构函数具有如下的一些特点：

1) 与类同名，之前冠以波浪线，以区别构造函数。

2) 不指定返回类型。

3) 不能指定参数。

4) 一个类只能有一个析构函数。

析构函数可以这样写：

```
          class  A{
                 ...
```

```
        public:
        ...
        ~A();
        };
    A::A(){...}                          //析构函数定义
```

[例10-6] 我们将构造函数进行编号，在程序的运行中去发现对象被定义后，调用的构造函数及对象被撤销时，调用的析构函数的处理过程。

```
#include <stdio.h>
class    A{
   int a, b, c, number;                  //类的私有成员，number 用于标志构造函数
      public:
     A(int=1 , int=2 , int=3);                  //类的构造函数
     A(double , double , double);
     A(long);
     A(A&);
     ~A(){                               //类的析构函数
       printf("object x destroyed by constr %d created\n",number);
         }
     void show()
          {
          printf("%d  ,  %d,  %d\n" , a, b, c);
          }
    };
    A::A(int i1, int i2, int i3)         //构造函数1
    {
     a=i1; b=i2;   c=i3; number=1;
     }
    A::A(double f1, double f2, double f3)    //构造函数2
    {
     a=(int)f1;    b=(int)f2;      c=(int)f3;    number=2;
     }
    A::A(long n)                         //构造函数3
    {
    a=b=c=(int)n;     number= 3;
    }
    A::A(A& other)                       //拷贝的构造函数4
          {
          a=other.a;
          b=other.b;
          c=other.c;
          number=4;
          }
   main()
   {
   A     x1;                            //定义对象x1调用构造1
   x1.show();
   A     x2(3);                         //定义对象x2调用构造1
   x2.show();
   A     x3(3, 1);                      //定义对象x3调用构造1
   x3.show();
```

```
A    x4(3.14, 2.414 , 6.28);   //定义对象X4调用构造2
x4.show();
A    x5(53L);                  //定义对象X5调用构造3
x5.show();
A  x6=x5;                      //定义对象X 6调用构造4
x6.show();
return 0;                      //各对象调用相应的析构函数。
}
```

程序的输出：

```
1, 2, 3
3, 2, 3
3, 1, 3
3, 2, 6
53, 53, 53
53, 53, 53
object x destroyed by constr 4 created
object x destroyed by constr 3 created
object x destroyed by constr 2 created
object x destroyed by constr 1 created
object x destroyed by constr 1 created
object x destroyed by constr 1 created
```

从上述输出结果来看，对象一旦定义，就必须要调用构造函数，退出时要调用析构函数。先定义的对象最后被毁灭或后定义的对象最先使用析构函数。

10.3.3 函数重载

上述程序中出现的构造函数，在形式上看，参数各不相同。正是由此，被定义的对象根据参数形式的不同，调用不同的构造函数，这个过程我们称为函数的重载。当然，不仅构造函数可以重载，其它的类成员函数也同样可以重载。

[例10-7] 类的成员函数的重载。

```
#include <stdlib.h>
#include <string.h>
#include <iostream.h.>
class string{
          int length;
          char str[256];
          public:
          string(){ length=0;  strcpy(str,"");}
                                    //类的构造函数1，完成字符串的初始化
   string(char *);                  //重载的构造函数2
   char * search(char);             //返回字符指针的成员函数1
   char * search(char *);           //返回字符指针的重载成员函数2
          };
   string::string(char *text)       //构造函数2的定义
   {
    if (strlen(text)<256)           //若串text 的长度小于256
          strcpy(str , text);       //将串text 复制给串str
```

```
        else
         strncpy(str  , text , 255);      //若串text 的长度超过255
                                          //则将串text 的255 个字符复制给串str
         length=strlen(str);
      }
      char *string::search(char arg)//成员函数1的定义
      {
          return strchr(str, arg);      //返回串str 中第一次出现字符arg 的位置
      }
     char *string::search(char *arg)//重载成员函数2的定义
     {
          return strstr(str, arg);      //返回串str 在串arg 中第一次出现的位置
      }
    void main()
    {
     string hellomsg='Hello,there , I'm a string!';
                                          //定义串string 的对象，自动调用构造函数1
     char *found;                         //定义字符串found
     cout<<hellomsg.search('t')<<"\r\n";
                                          //输出对象的成员函数1的返回值
     cout<<hellomsg.search("string")<<"\r\n";
                                          //输出对象的成员函数2的返回值
     }
```

运行程序，输出为：

```
Hello, there, I'm a string!
string!
```

10.3.4 友元

在类成员的介绍中，我们对类的三种成员做过详细的说明，特别是强调了类的私有成员和类保护段成员的隐蔽性，它们只能被类对象的成员函数所访问，这也同样表明，类的成员函数可以访问类中的所有成员。友元是 C++提供给外部的类或函数访问类的私有成员和保护成员的一种途径。

在一个类中，将friend加在某个函数或某个类的前面，则该函数或类将成为所在类的友元。友元不受它们在类中出现次序的影响，而仅表明其是类的友元。

[例10-8] 说明类的友元函数的作用及与成员函数的区别。

```
#include <iostream.h>
#include <string.h>
class stud{
   char *name, *num , *tel;                    //类的私有成员：姓名，学号，电话号码
   public:
     stud(char *na, char *nu,  char *pho )     //类的构造函数
     {
      name=new char[strlen(na)+1];
                                          //使用运算符new 向系统申请name 所需的存
                                            储空间
      strcpy(name  , na);                      //将参数na 的值复制给姓名name
```

```
num=new char[strlen(nu)+1];
                          //使用运算符new向系统申请num所需的存储空间
strcpy(num ,nu);          //将参数nu的值复制给学号num
tel=new char[strlen(pho)+1];
                          //使用运算符new向系统申请tel所需的存储空间
strcpy(tel ,pho);         //将参数pho的值复制给电话号码tel
}
void show(stud&);         //类的公有成员函数
friend void show(stud&);//类的友元函数
~stud()                   //类的析构函数
 { delete name;  delete num; delete tel;}
                          //使用运算符delete 释放类的各数据成员所占存储空间
 };
void show(stud &student);//友元函数的定义
{
 cout<<" 类的友元函数的调用：\n";
 cout<<"student\"s name is:"<<student.name
    <<"\nstudent\"s number:"<<student.num
    <<"\nstudent\"s telephone:"<<student.tel<<"\n";
 cout<<" 友元函数调用类的成员函数：\n";
 student.show(student);              //调用类的成员函数
 }
 void stud::show(stud &student)//类的成员函数的定义
{
 cout<<" 类的成员函数的调用：\n";
 cout<<"student\"s name is:"<<student.name
    <<"\nstudent\"s number:"<<student.num
    <<"\nstudent\"s telephone:"<<student.tel<<"\n";
 }
 void main()
 {
  stud      x("li-ling" ,"j98-10323" ,"0285533123");     //定义类的对象x
  show(x);                          //调用类的友元函数
  x.show(x);                                //调用类的成员函数
 }
```

上述程序中，类的友元函数和类的成员函数完成相同的功能，并且函数名也完全相同，不同的只是说明和定义的形式。让我们先看一下程序运行后的输出：

类的友元函数的调用：

```
student\'s name is: li-ling
student\'s number: j98-10323
student\'s telephone: 0285533123
```

友元函数调用类的成员函数：

```
student\'s name is: li-ling
student\'s number: j98-10323
student\'s telephone: 0285533123
```

类的成员函数的调用：

```
student\'s name is: li-ling
student\'s number: j98-10323
student\'s telephone: 0285533123
```

顺便需要说明的是：类的友元函数的定义既可放在类内，也可在类外，与成员函数是一致的。如放在类内，则不受段访问特性的影响。

友元可以是多个类的友元，即可以跨类访问。

[例10-9] 关于跨类友元的访问，程序能验证其各位数的立方和等于其数本身的任一个三位数，如：$1^3+5^3+3^3=150$等。

```
#include  <iostream.h>
class hundreds;  //类的声明。在类未定义前需使用时，必须先声明。
class cubic             //类cubic 的定义
{
  int sum;
  public:
  void set_sum(int a, int b, int c){ sum=a*a*a+b*b*b+c*c*c;}
                        //类的成员函数用于求其各位数的立方和。
  friend int equal(cubic,chundreds h);     //类的友元函数
    } cub;              //在类定义的同时定义对象cub
  class hundreds{//类hundreds 的定义
    int num;
    public:
    void set_num(int a, int b, int c){
    num=100*a+10*b+c;
   }
                        //类的成员函数用于求其三个数构成的三位数
     friend int equal(cubic,chundreds h);//类的友元函数
     void display(void){cout<<num<<endl;}类的成员函数
     }hun; //在类定义的同时定义对象hun
     int equal(cubic c, hundreds h){ return !(c.sum-h.num);}
          //类cubic 和类hundreds 的友元函数equal 的定义，用于验证由三个数构成的
     main(void)
     {
       int d3, d2, d1;
   for (d3=1;d3<10;d3++)
    for (d2=0;d2<10;d2++)
     for (d1=0;d1<10;d1++)
     {
      cub.set_sum(d3   , d2, d1);
                        //对象cub 调用其成员函数求得三个数的立方和
                        hun.set_num(d3   , d2, d1);
                        //对象hun 调用其成员函数求得构成的三位数
      if (equal(cub, hun)) hun.display();   /相等则显示出三位数
      }
   return 0;
   }
```

该程序的输出是：

```
150
370
371
407
```

一定请注意，友元函数在使用时，与成员函数的使用有所区别，它不能在函数名前加上类名作函数调用的限定，即不能写成cubic::equal或hundreds::equal。同时，我们也可以将类声明为友元。

```
    class Y;      //类的声明
    class X{
    friends       Y;    //说明类Y是类X的友元
    int data1;
    …
    void fun_mem1(…){…}
    …
    };
class Y{
    float data2;
          …
          void fun_mem2(…){ …}
          …
};
```

上述定义形式表明，类Y的全部成员均可访问类X的成员。

10.4 对象指针

当我们需要动态地在存储空间上使用对象时，就需要定义对象指针。正如在C语言中使用指针变量一样，什么时候需要，就使用malloc()为其分配内存，使用完毕，就利用free()释放其占用的内存。

1. 运算符new与delete

在C++中，运算符new和delete可以让用户创建任意持续时间的动态变量，也可以让用户为任意种类的C语言的对象分配内存空间。通常，我们用new来为C或C++的对象动态地分配存储空间，用delete来将占用内存的对象移走或称为释放，其意义与C语言提供的malloc()功能与free()功能类似。

运算符new和delete的用法很简单，它们均为单目运算符：

例：我们定义对象指针如下：

```
  int     *num1;
  float   *num2;
  double  *num3
  char    *str1;
 class X{…}*objx;
```

为对象指针分配内存空间：

```
num1=new    int;
num2=new    float;
num3=new    double;
str1=new    char[80];
objx=new    X;
```

利用new运算符分配空间的对象，使用完毕后应通过delete对所占空间进行释放。

```
delete num1;
delete num2;
delete num3;
delete []str1;
delete objx;
```

2. 动态地创建类对象

生活中常会遇到的实际问题是处理的数据量不固定，数据的个数在动态地改变。链表技术就是为解决这一问题应运而生的，这种特定的数据结构中每一个节点用于存放一组数据，随着数据的不断增加，链表也在不断增长，占用的内存不断增加。当问题得到解决后，数据不再需要保存，占用的内存就释放掉，这个过程就是数据的动态存储。

[例10-10] 动态存放一组字符串在一双向链表，链表结构如图 10-2所示。

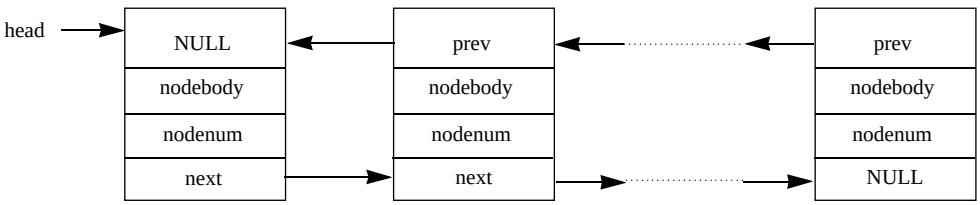

图10-2 链表结构示意

其中，head是头指针，指向链表的头；prev指向链表的前一个节点；next指向链表的下一个节点;nodebody是链表节点存放的字符串；nodenum是链表节点存放的字符串长度。

链表节点的结构为：

```
class tnode{
   tnode *prev;
   tnode *next;
   char * nodebody;
   int   nodenum;
   }
```

双向链表的操作有：

1) 移动指针到链表头。

2) 移动指针到链表尾。

3) 在链表尾部追加一个字符串。

4) 从链表开始，按字母的排列顺序插入一个字符串。

5) 释放链表各节点所占内存。

程序如下：

```
#include "stdlib.h"
#include <stdio.h>
#include <string.h>
#include <conio.h>
class tnode{            //链表节点的数据结构
   public:
   tnode *prev;         //指向前一个节点
```

```
    tnode *next;          //指向下一个节点
    char *nodebody;       //节点所存字符串
    int  nodenum;         //节点字符串的长度
       };
    class dbllist{        //链表结构
     tnode *head;         //链表的头指针

     tnode *base;
     tnode *hold;         //base 与hold 用于跟踪链表增长
     tnode *create(char *);    //为一个节点分配存储空间
     public:
      dbllist();                //链表的构造函数
      ~dbllist();               //链表的析构函数
       void clear();            //释放链表所占空间
       tnode *gohead();         //将指针移到链表头
       tnode *gotail();         //将指针移到链表尾
       tnode *gonext();         //将指针移到链表的下一个节点
       tnode *goprev();         //将指针移到链表的前一个节点
       tnode *append(char *); //追加一个字符串
       tnode *insert(char *); //插入一个字符串
       char* accept(tnode *); //接收指定节点的字符串
       };
       dbllist::dbllist()
       {
        head=base=hold=NULL;        //链表指针初始化
        }
      dbllist::~dbllist()           //析构函数
        {
        clear();
        }
    void dbllist::clear()//释放链表各节点
    {
    base=head;                    //头指针
    while(base)
        {                 //链表非空
     hold=base->next;
//删除节点如
//图10-3 所示
     delete   base;
     base=hold;                   //在跟踪链表的过程中释放各节点
     }
       head=base=hold=NULL;
       }
     tnode * dbllist::gohead() //将指针移到链表头
     {
     base=head;
     if (base) return base;    //返回头指针
     else return NULL;
     }
   tnode * dbllist::gotail()                  //将指针移到链表尾
   {
     if (base)
```