第7章 自定义数据类型

C++提供了许多种基本的数据类型(如int、float、double、char等)供用户使用。但是由于程序需要处理的问题往往比较复杂，而且呈多样化，已有的数据类型显得不能满足使用要求。因此C++允许用户根据需要自己声明一些类型,例如第5章介绍的数组就是用户自己声明的数据类型。此外，用户可以自己声明的类型还有结构体(structure)类型、共用体(union)类型、枚举(enumeration)类型、类(class)类型等，这些统称为用户自定义类型(user-defined type,UDT)。 本章介绍结构体类型、共用体类型和枚举类型，第8章将介绍类类型。

7.1 结构体类型

7.1.1 结构体概述

有时需要将不同类型的数据组合成一个有机的整体，以供用户方便地使用。这些组合在一个整体中的数据是互相联系的。例如，一个学生的学号、姓名、性别、年龄、成绩、家庭地址等项，都是这个学生的属性。见图**7.1**。

num	name	sex	age	score	addr
10010	Li Fun	M	18	87.5	Beijing

图**7.1**

在一个组合项中包含若干个类型不同（当然也可以相同）的数据项。C和**C++**允许用户自己指定这样一种数据类型，它称为结构体。它相当于其他高级语言中的记录**(record)**。

例如，可以通过下面的声明来建立如图**7.1**所示的数据类型。

```
struct Student          //声明一个结构体类型Student
{ int num;              //包括一个整型变量num
char name[20];          //包括一个字符数组name，可以容纳20个字符
char sex;               //包括一个字符变量sex
int age;                //包括一个整型变量age
float score;            //包括一个单精度型变量
char addr[30];          //包括一个字符数组addr，可以容纳30个字符
};                      //最后有一个分号
```

这样，程序设计者就声明了一个新的结构体类型**Student(struct**是声明结构体类型时所必须使用的关键字，不能省略)，它向编译系统声明： 这是一种结构体类型，它包括**num, name, sex, age, score, addr**等不同类型的数据项。应当说明**Student**是一个类型名，它和系统提供的标准类型（如**int**、**char**、**float**、**double**等）一样，都可以用来定义变量，只不过结构体类型需要事先由用户自己声明而已。

声明一个结构体类型的一般形式为

struct 结构体类型名

｛成员表列｝；

结构体类型名用来作结构体类型的标志。上面的声明中**Student**就是结构体类型名。大括号内是该结构体中的全部成员**(member)**，由它们组成一个特定的结构体。上例中的**num,name,sex,score**等都是结构体中的成员。在声明一个结构体类型时必须对各成员都进行类型声明，即

类型名 成员名；

每一个成员也称为结构体中的一个域**(field)**。成员表列又称为域表。成员名的定名规则与变量名的定名规则相同。

声明结构体类型的位置一般在文件的开头，在所有函数(包括**main**函数)之前，以便本文件中所有的函数都能利用它来定义变量。当然也可以在函数中声明结构体类型。

在C语言中，结构体的成员只能是数据(如上面例子中所表示的那样)。C++对此加以扩充，结构体的成员既可以包括数据(即数据成员)，又可以包括函数(即函数成员)，以适应面向对象的程序设计。但是由于C++提供了类(class)类型，一般情况下，不必使用带函数的结构体，因此在本章中只介绍只含数据成员的结构体，有关包含函数成员的结构体将在第8章介绍类类型时一并介绍。

7.1.2 结构体类型变量的定义方法及其初始化

前面只是指定了一种结构体类型，它相当于一个模型，但其中并无具体数据，系统也不为之分配实际的内存单元。为了能在程序中使用结构体类型的数据，应当定义结构体类型的变量，并在其中存放具体的数据。

1. 定义结构体类型变量的方法

可以采取以下3种方法定义结构体类型的变量。

(1） 先声明结构体类型再定义变量名

如上面已定义了一个结构体类型Student，可以用它来定义结构体变量。如

```
Student  student1, student2;
```

以上定义了**student1**和**student2**为结构体类型**Student**的变量，即它们具有**Student**类型的结构。如图**7.2**所示。

student1：	10001	Zhang Xin	M	19	90.5	Shanghai
student2：	10002	Wang Li	F	20	98	Beijing

图**7.2**

在定义了结构体变量后，系统会为之分配内存单元。例如**student1**和**student2**在内存中各占**63**个字节（**4+20+1+4+4+30=63**）。

(2) 在声明类型的同时定义变量

例如：

```
struct Student              //声明结构体类型Student
{ int num;
char name[20];
char sex;
int age;
float score;
char addr[30];
} student1,student2;  //定义两个结构体类型Student的变量student1,student2
```

这种形式的定义的一般形式为

struct 结构体名

{

成员表列

}变量名表列;

(3) 直接定义结构体类型变量

其一般形式为

```
struct                  //注意没有结构体类型名
{
成员表列
} 变量名表列;
```

这种方法虽然合法，但很少使用。提倡先定义类型后定义变量的第**(1)**种方法。在程序比较简单，结构体类型只在本文件中使用的情况下，也可以用第**(2)**种方法。

关于结构体类型，有几点要说明：

(1) 不要误认为凡是结构体类型都有相同的结构。实际上，每一种结构体类型都有自己的结构，可以定义出许多种具体的结构体类型。

(2) 类型与变量是不同的概念，不要混淆。只能对结构体变量中的成员赋值，而不能对结构体类型赋值。在编译时，是不会为类型分配空间的，只为变量分配空间。

(3) 对结构体中的成员（即“域”），可以单独使用，它的作用与地位相当于普通变量。关于对成员的引用方法见**7.3**节。

(4) 成员也可以是一个结构体变量。

如

```
struct Date        //声明一个结构体类型Date
{ int month;
int day;
int year;
} ;
struct Student              //声明一个结构体类型Student
{ int num;
char name[20];
char sex;
int age;
Date birthday;          //Date是结构体类型，birthday是Date类型的成员
char addr[30];
}student1,student2; //定义student1和student2为结构体类型Student的变量
```

Student的结构见图7.3所示。

<table>
<tr><td rowspan="2">num</td><td rowspan="2">name</td><td rowspan="2">sex</td><td rowspan="2">age</td><td colspan="3">birthday</td><td rowspan="2">addr</td></tr>
<tr><td>month</td><td>day</td><td>year</td></tr>
</table>

图7.3

(5) 结构体中的成员名可以与程序中的变量名相同,但二者没有关系。例如,程序中可以另定义一个整型变量**num**,它与**student**中的**num**是两回事,互不影响。

2. 结构体变量的初始化

和其他类型变量一样，对结构体变量可以在定义时指定初始值。如

```
struct Student
{ int num;
char name[20];
char sex;
int age;
float score;
char addr[30];
} student1={10001，"Zhang Xin",'M',19,90.5,"Shanghai"};
```

这样，变量**student1**中的数据如图**7.2**中所示。

也可以采取声明类型与定义变量分开的形式，在定义变量时进行初始化：

```
Student student2={10002，"Wang Li",'F',20,98,"Beijing";
```

//Student是已声明的结构体类型

7.1.3 结构体变量的引用

在定义了结构体变量以后,当然可以引用这个变量。

(1) 可以将一个结构体变量的值赋给另一个具有相同结构的结构体变量。如上面的**student1**和**student2**都是**student**类型的变量，可以这样赋值：

```
student1= student2;
```

(2) 可以引用一个结构体变量中的一个成员的值。例如， **student1.num**表示结构体变量**student1**中的成员的值，如果**student1**的值如图**7.2**所示，则**student1.num**的值为**10001**。

引用结构体变量中成员的一般方式为

结构体变量名.成员名

例如可以这样对变量的成员赋值:

student1.num=10010;

(3) 如果成员本身也是一个结构体类型,则要用若干个成员运算符,一级一级地找到最低一级的成员。例如,对上面定义的结构体变量**student1,** 可以这样访问各成员:

student1.num (引用结构体变量**student1**中的**num**成员)

如果想引用**student1**变量中的**birthday**成员中的**month**成员，不能写成**student1.month**，必须逐级引用，即

student1.birthday.month (引用结构体变量**student1**中的**birthday**成员中的**month**成员)

(4) 不能将一个结构体变量作为一个整体进行输入和输出。例如,已定义**student1**和**student2**为结构体变量,并且它们已有值。不能企图这样输出结构体变量中的各成员的值:

```
cout<<student1;
```

只能对结构体变量中的各个成员分别进行输入和输出。

(5) 对结构体变量的成员可以像普通变量一样进行各种运算（根据其类型决定可以进行的运算种类）。例如

```
student2.score＝student1.score;
sum＝student1.score＋student2.score;
student1.age++;
++student1.age;
```

由于“.”运算符的优先级最高，因此**student1.age++**相当于**(student1.age)++**。**++**是对**student1.age**进行自加运算，而不是先对**age**进行自加运算。

(6) 可以引用结构体变量成员的地址，也可以引用结构体变量的地址。如

```
cout<<&student1;          //输出student1的首地址
cout<<&student1.age;      //输出student1.age的地址
```

结构体变量的地址主要用作函数参数，将结构体变量的地址传递给形参。

例7.1 引用结构体变量中的成员。

```
#include <iostream>
using namespace std;
struct Date                    //声明结构体类型Date
{int month;
int day;
int year;
};
struct Student                 //声明结构体类型Student
{int num;
char name[20];
char sex;
Date birthday;                 //声明birthday为Date类型的成员
float score;
}student1,student2={10002,"Wang Li",'f',5,23,1982,89.5};
 //定义Student 类型的变量student1,student2，并对student2初始化
int main( )
{ student1=student2;                  //将student2各成员的值赋予student1的相应成员
```

```
cout<<student1.num<<endl;        //输出student1中的num成员的值
cout<<student1.name<<endl;       //输出student1中的name成员的值
cout<<student1.sex<<endl;        //输出student1中的sex成员的值
cout<<student1.birthday.month<<'/'<<student1.birthday.day<<'/'
<<student1.birthday.year<<endl;  //输出student1中的birthday各成员的值
cout<<student1.score<<endl;
return 0;
}
```

运行结果如下：

```
10002
Wang Li
f
5/23/1982
89.5
```

7.1.4 结构体数组

一个结构体变量中可以存放一组数据（如一个学生的学号、姓名、成绩等数据）。如果有10个学生的数据需要参加运算，显然应该用数组，这就是结构体数组。结构体数组与以前介绍过的数值型数组的不同之处在于： 每个数组元素都是一个结构体类型的数据，它们都分别包括各个成员项。

1. 定义结构体数组

和定义结构体变量的方法相仿，定义结构体数组时只需声明其为数组即可。如

```
struct Student                  //声明结构体类型Student
{ int num;
char name[20];
char sex;
int age;
float score;
char addr[30];
};
```

Student stu[3];　　//定义Student类型的数组stu

也可以直接定义一个结构体数组，如

```
struct Student
{ int num;
char name[20];
char sex;
int age;
float score;
char addr[30];
}stu[3];
```

或

```
struct
{ int num;
char name[20];
char sex;
int age;
float score;
char addr[30];
}stu[3];
```

见图**7.4**。数组各元素在内存中连续存放，见图**7.5**示意。

	num	name	sex	age	score	addr
stu[0]	10101	Li Lin	M	18	87.5	103 Beijing Road
stu[1]	10102	Zhang Fun	M	19	99	130 Shanghai Road
stu[2]	10104	Wang Min	F	20	78.5	1010 Zhongshan Road

图**7.** 4

图**7.5**

2. 结构体数组的初始化

与其他类型的数组一样，对结构体数组可以初始化。如

```
struct Student
{ int num;
char name[20];
char sex;
int age;
float score;
char addr[30];
}sty[3]={{10101,"Li Lin",'M',18,87.5,"103 Beijing Road"},
{10102,"Zhang Fun",'M'，19,99,"130 Shanghai Road"},
{10104,"Wang Min",'F',20,78.5,"1010,Zhongshan Road"}};
```

定义数组**stu**时，也可以不指定元素个数，即写成以下形式：

stu[ ]={{…}, {…} , {…} };

编译时，系统会根据给出初值的结构体常量的个数来确定数组元素的个数。一个结构体常量应包括结构体中全部成员的值。

当然，数组的初始化也可以用以下形式：

Student stu[ ]={{…},{…},{…}}; //已事先声明了结构体类型**Student**

由上可以看到，结构体数组初始化的一般形式是在所定义的数组名的后面加上

={初值表列};

3. 结构体数组应用举例

下面举一个简单的例子来说明结构体数组的定义和引用。

例7.2 对候选人得票的统计程序。设有3个候选人，最终只能有1人当选为领导。今有10个人参加投票，从键盘先后输入这10个人所投的候选人的名字，要求最后输出这3个候选人的得票结果。

可以定义一个候选人结构体数组，包括3个元素，在每个元素中存放有关的数据。

程序如下：

```
#include <iostream>
struct Person                          //声明结构体类型Person
{ char name[20];
int count;
```

```
};
int main( )
{ Person  leader[3]={"Li",0,"Zhang",0,"Fun",0};
//定义Person类型的数组，内容为3个候选人的姓名和当前的得票数
int i,j;
char leader_name[20];          //leader_name为投票人所选的人的姓名
for(i=0;i<10;i++)
{cin>>leader_name;           //先后输入10张票上所写的姓名
 for(j=0;j<3;j++)           //将票上姓名与3个候选人的姓名比较
if(strcmp(leader_name,leader[j].name)==0) leader[j].count++;
//如果与某一候选人的姓名相同，就给他加一票
 }
cout<<endl;
for(i=0;i<3;i++)         //输出3个候选人的姓名与最后得票数
{cout<<leader[i].name<<":"<<leader[i].count<<endl;}
return 0;
}
```

运行情况如下：

Zhang↙ (每次输入一个候选人的姓名)

Li↙

Fun↙

Li↙

Zhang↙

Li↙

Zhang↙

Li↙

Fun↙

Wang↙

Li:4 (输出3个候选人的姓名与最后得票数)

Zhang:3

Fun:2

程序定义一个全局的结构体数组**leader,**它有**3**个元素，每一元素包含两个成员，即**name**（姓名）和**count(**得票数**)**。在定义数组时使之初始化，使**3**位候选人的票数都先置零。见图**7.6**。

name	count
Li	0
Zhang	0
Fun	0

图**7.6**

在这个例子中，也可以不用字符数组而用**string**方法的字符串变量来存放姓名数据，程序可修改如下：

```
#include <iostream>
#include <string>
using namespace std;
struct Person
{ string name;                    //成员name为字符串变量
int count;
};
int main( )
{ Person leader［3］={"Li",0,"Zhang",0,"Fun",0};
int i,j;
string leader_name;               // leader_name为字符串变量
for(i=0;i<10;i++)
{cin>>leader_name;
 for(j=0;j<3;j++)
if(leader_name==leader[j].name) leader[j].count++;  //用“==”进行比较
 }
cout<<endl;
for(i=0;i<3;i++)
{cout<<leader[i].name<<":"<<leader[i].count<<endl;}
return 0;
}
```

7.1.5 指向结构体变量的指针

一个结构体变量的指针就是该变量所占据的内存段的起始地址。可以设一个指针变量，用来指向一个结构体变量，此时该指针变量的值是结构体变量的起始地址。指针变量也可以用来指向结构体数组中的元素。

1. 通过指向结构体变量的指针引用结构体变量中的成员

下面通过一个简单例子来说明指向结构体变量的指针变量的应用。

例7.3 指向结构体变量的指针的应用。

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{struct Student          //声明结构体类型student
{ int num;
string name;
char sex;
float score;
};
Student stu;            //定义Student类型的变量stu
Student *p=&stu;        //定义p为指向Student类型数据的指针变量并指向stu
stu.num=10301;          //对stu中的成员赋值
stu.name="Wang Fun";    //对string变量可以直接赋值
stu.sex='f';
stu.score=89.5;
cot<<stu. num<<" "<<stu.name<<" "<<stu.sex<<" "<<stu.score<<endl;
cout<<(*p)>num<<" "<<(*p).name<<" "<<(*p).sex<<" "<<(*p).score<<endl;
return 0;
}
```

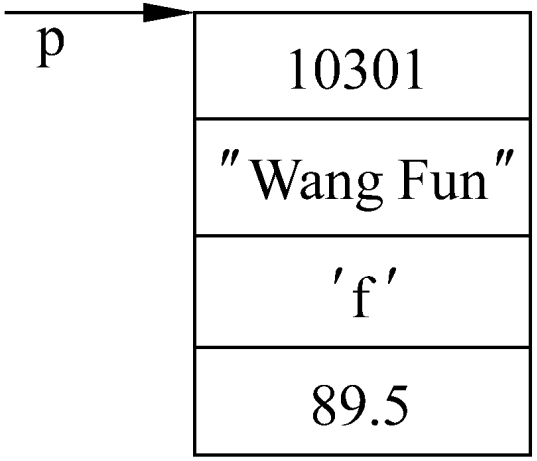


图7.7

程序运行结果如下：

10301 Wang Fun f 89.5 (通过结构体变量名引用成员)

10301 Wang Fun f 89.5 (通过指针引用结构体变量中的成员)

两个**cout**语句输出的结果是相同的。

为了使用方便和使之直观，**C++**提供了指向结构体变量的运算符**->**，例如**p->num**表示指针**p**当前指向的结构体变量中的成员**num**。**p->num** 和**(*p).num**等价。同样，**p->name**等价于**(*p).name**。

也就是说，以下**3**种形式等价：

① 结构体变量**.**成员名。如**stu.num**。

② **(*p).**成员名。如**(*p).num**。

③ **p->**成员名。如**p->num**。“**->**”称为指向运算符。

请分析以下几种运算：

p->n 得到**p**指向的结构体变量中的成员**n**的值。

p->n++ 得到**p**指向的结构体变量中的成员**n**的值，用完该值后使它加**1**。

++p->n 得到**p**指向的结构体变量中的成员**n**的值，并使之加**1**，然后再使用它。

2. 用结构体变量和指向结构体变量的指针构成链表

链表是一种常见的重要的数据结构。图**7.8**表示最简单的一种链表（单向链表）的结构。

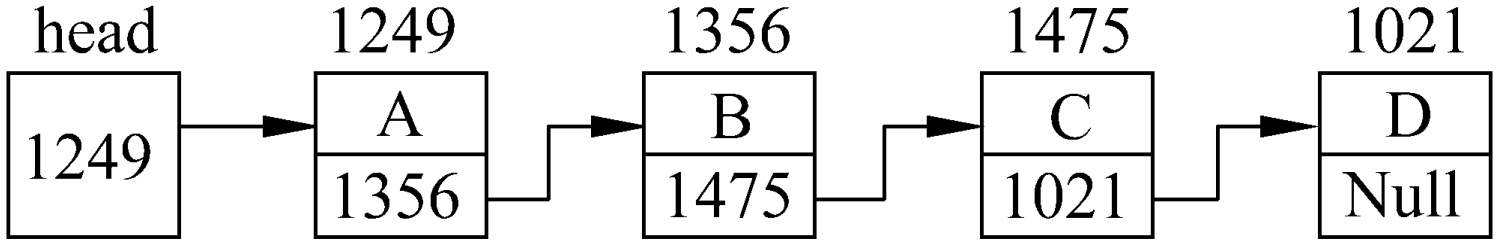


图**7.8**

链表有一个“头指针”变量，图中以**head**表示，它存放一个地址。该地址指向一个元素。链表中的每一个元素称为“结点”，每个结点都应包括两个部分：一是用户需要用的实际数据，二是下一个结点的地址。

可以看到链表中各元素在内存中的存储单元可以是不连续的。要找某一元素，可以先找到上一个元素，根据它提供的下一元素地址找到下一个元素。

可以看到，这种链表的数据结构，必须利用结构体变量和指针才能实现。可以声明一个结构体类型，包含两种成员，一种是用户需要用的实际数据，另一种是用来存放下一结点地址的指针变量。例如，可以设计这样一个结构体类型：

```
struct Student
{ int num;
float score;
Student *next;          //next指向Student结构体变量
};
```

其中成员**num**和**score**是用户需要用到的数据，相当于图**7.8**结点中的**A,B,C,D**。**next**是指针类型的成员，它指向**Student**类型数据（就是**next**所在的结构体类型）。用这种方法就可以建立链表。见图**7.9**。

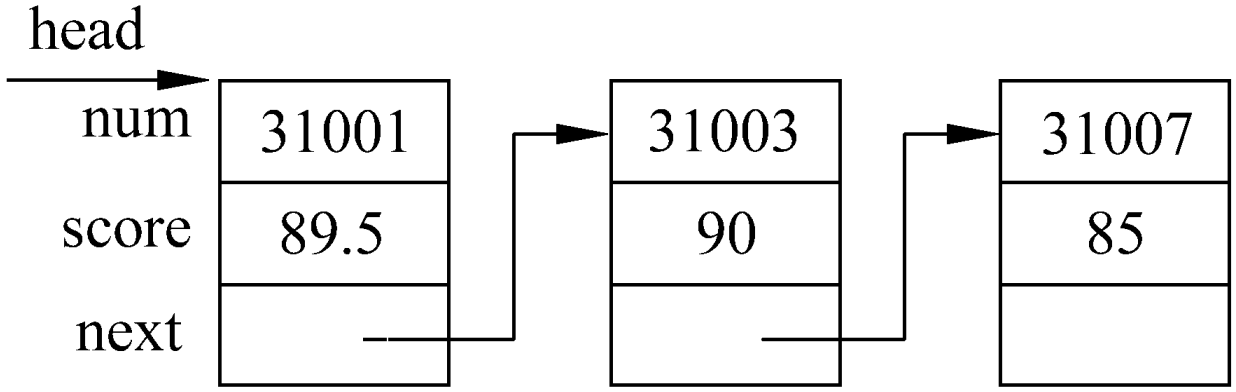


图**7.9**

图中每一个结点都属于**Student**类型，在它的成员**next**中存放下一个结点的地址，程序设计者不必知道各结点的具体地址，只要保证能将下一个结点的地址放到前一结点的成员**next**中即可。

下面通过一个例子来说明如何建立和输出一个简单链表。

例**7.4** 建立一个如图**7.9**所示的简单链表，它由**3**个学生数据的结点组成。输出各结点中的数据。

```
#define NULL 0
#include <iostream>
struct Student
{ long num;
float score;
struct Student *next;
};
int main( )
{ Student a,b,c,*head,*p;
a. num=31001; a.score=89.5;          //对结点a的num和score成员赋值
b. num=31003; b.score=90;            //对结点b的num和score成员赋值
c. num=31007; c.score=85;            //对结点c的num和score成员赋值
head=&a;                        //将结点a的起始地址赋给头指针head
```

```
a.next=&b;                          //将结点b的起始地址赋给a结点的next成员
b.next=&c;                          //将结点c的起始地址赋给b结点的next成员
c.next=NULL;                        //结点的next成员不存放其他结点地址
p=head;                             //使p指针指向a结点
do
{cout<<p->num<<" "<<p->score<<endl;    //输出p指向的结点的数据
p=p->next;                          //使p指向下一个结点
} while(p!=NULL);                   //输出完c结点后p的值为NULL
return 0;
}
```

请读者考虑：①各个结点是怎样构成链表的。②p起什么作用？

本例是比较简单的，所有结点(结构体变量)都是在程序中定义的，不是临时开辟的，也不能用完后释放，这种链表称为静态链表。对各结点既可以通过上一个结点的**next**指针去访问，也可以直接通过结构体变量名**a,b,c**去访问。

动态链表则是指各结点是可以随时插入和删除的，这些结点并没有变量名，只能先找到上一个结点，才能根据它提供的下一结点的地址找到下一个结点。只有提供第一个结点的地址，即头指针**head**，才能访问整个链表。如同一条铁链一样，一环扣一环，中间是不能断开的。建立动态链表，要用到下面**7.1.7**小节介绍的动态分配内存的运算符**new**和动态撤销内存的运算符**delete**。

7.1.6 结构体类型数据作为函数参数

将一个结构体变量中的数据传递给另一个函数，有下列3种方法：

(1) 用结构体变量名作参数。一般较少用这种方法。

(2) 用指向结构体变量的指针作实参，将结构体变量的地址传给形参。

(3) 用结构体变量的引用变量作函数参数。

下面通过一个简单的例子来说明，并对它们进行比较。

例7.5 有一个结构体变量stu，内含学生学号、姓名和3门课的成绩。要求在main函数中为各成员赋值，在另一函数print中将它们的值输出。

(1) 用结构体变量作函数参数

```
#include <iostream>
#include <string>
using namespace std;
struct Student                          //声明结构体类型Student
{ int num;
char name[20];
float score[3];
};

int main( )
{ void print(Student);                  //函数声明，形参类型为结构体Student
Student stu;                            //定义结构体变量
stu.num=12345;                          //以下5行对结构体变量各成员赋值
stu.name="Li Fung";
stu.score[0]=67.5;
stu.score[1]=89;
stu.score[2]=78.5;
print(stu);                             //调用print函数，输出stu各成员的值
return 0;
}
```

```
void print(Student stu)
{cout<<stu.num<<" "<<stu.name<<" "<<stu.score[0]<<" "
<<stu.score[1]<<" "<<stu.score[2]<<endl;
}
```

运行结果为

12345 Li Fung 67.5 89 78.5

(2) 用指向结构体变量的指针作实参

在上面程序的基础上稍作修改即可。

```
#include <iostream>
#include <string>
using namespace std;
struct Student
{ int num;
string name;                    //用string类型定义字符串变量
float score[3];
}stu={12345,"Li Fung",67.5,89,78.5};   //定义结构体student变量stu并赋初值

int main( )
{ void print(Student *);        //函数声明，形参为指向Student类型数据的指针变量
```

```
Student *pt=&stu;              //定义基类型为Student的指针变量pt，并指向stu
print(pt);                     //实参为指向Student类数据的指针变量
return 0;
}
void print(Student *p)         //定义函数，形参p是基类型为Student的指针变量
{ cout<<p->num<<" "<<p->name<<" "<<p->score[0]<<" "
<<p->score[1]<<" "<<p->score[2]<<endl;
}
```

调用**print**函数时，实参指针变量**pt**将**stu**的起始地址传送给形参**p**（**p**也是基类型为**student**的指针变量）。这样形参**p**也就指向**stu**，见图**7.10**。在**print**函数中输出 p 所指向的结构体变量的各个成员值，它们也就是**stu**的成员值。

在**main**函数中也可以不定义指针变量**pt**，而在调用**print**函数时以**&stu**作为实参，把**stu**的起始地址传给实参**p**。

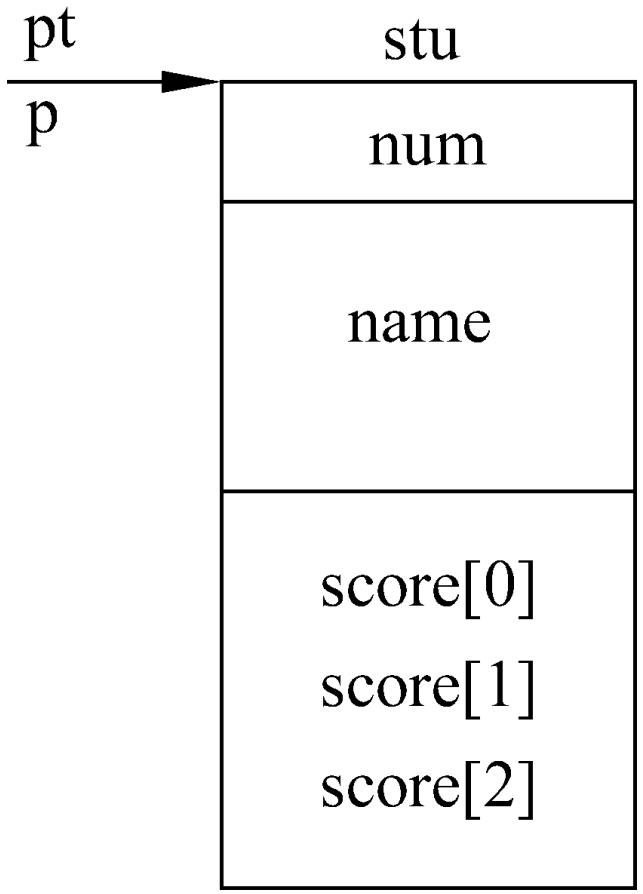


图7.10

(3) 用结构体变量的引用作函数参数

```
#include <iostream>
#include <string>
using namespace std;
struct Student
{ int num;
string name;
float score[3];
}stu={12345,"Li Li",67.5,89,78.5};

int main( )
{ void print(Student &);          //函数声明，形参为Student类型变量的引用
print(stu);                       //实参为结构体Student变量
return 0;
}

void print(Student &stud)         //函数定义，形参为结构体Student变量的引用
{cout<<stud.num<<" "<<stud.name<<" "<<stud.score[0]<<" "
<<stud.score[1]<<" "<<stud.score[2]<<endl;
}
```

程序(1)用结构体变量作实参和形参，程序直观易懂，效率是不高的。

程序(2)采用指针变量作为实参和形参，空间和时间的开销都很小，效率较高。但程序（2）不如程序(1)那样直观。

程序(3)的实参是结构体Student类型变量，而形参用Student类型的引用，虚实结合时传递的是stu的地址，因而效率较高。它兼有(1)和(2)的优点。

引用变量主要用作函数参数，它可以提高效率，而且保持程序良好的可读性。

在本例中用了string方法定义字符串变量，在某些C++系统中目前不能运行这些程序，读者可以修改程序，使之能在自己所用的系统中运行。

*7.1.7 动态分配和撤销内存的运算符new和delete

在软件开发过程中，常常需要动态地分配和撤销内存空间，例如对动态链表中结点的插入与删除。在C语言中是利用库函数malloc和free来分配和撤销内存空间的。C++提供了较简便而功能较强的运算符new和delete来取代malloc和free函数。注意： new和delete是运算符，不是函数，因此执行效率高。虽然为了与C语言兼容，C++仍保留malloc和free函数，但建议用户不用malloc和free函数，而用new和delete运算符。

new运算符的例子：

```
new int;        //开辟一个存放整数的存储空间，返回一个指向该存储空间的地址(即指针)
```

new int(100);

//开辟一个存放整数的空间，并指定该整数的初值为**100**，返回一个指向该存储空间的地址

new char[10]; //开辟一个存放字符数组(包括**10**个元素)的空间，返回首元素的地址

new int[5][4]; //开辟一个存放二维整型数组(大小为**5*4**)的空间，返回首元素的地址

float *p=new float(3.14159); //开辟一个存放单精度数的空间，并指定该实数的初值为

//**3.14159**，将返回的该空间的地址赋给指针变量**p**

new运算符使用的一般格式为

new 类型 **[**初值**]**

用**new**分配数组空间时不能指定初值。如果由于内存不足等原因而无法正常分配空间，则**new**会返回一个空指针**NULL**，用户可以根据该指针的值判断分配空间是否成功。

delete运算符使用的一般格式为

delete [ ] 指针变量

例如要撤销上面用**new**开辟的存放单精度数的空间**(**上面第**5**个例子**)**，应该用

delete p;

前面用**"new char[10];"**开辟的字符数组空间，如果把**new**返回的指针赋给了指针变量**pt**，则应该用以下形式的**delete**运算符撤销该空间：

delete [] pt; //在指针变量前面加一对方括号，表示是对数组空间的操作

例**7.6** 开辟空间以存放一个结构体变量。

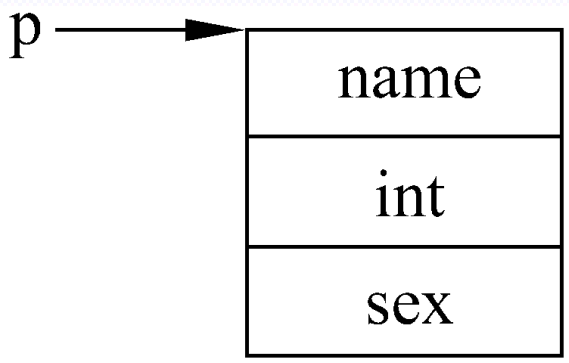
```
#include <iostream>
#include <string>
using namespace std;
struct Student            //声明结构体类型Student
{ string name;
int num;
char sex;
};
int main( )
{ Student *p;          //定义指向结构体类型Student的数据的指针变量
p=new Student;         //用new运算符开辟一个存放Student型数据的空间
p->name="Wang Fun";    //向结构体变量的成员赋值
p->num=10123;
p->sex='m';
cout<<p->name<<endl<<p->num<<endl<<p->sex<<endl; //输出各成员的值
delete p;          //撤销该空间
return 0;
}
```

运行结果为

Wang Fun

10123

m



用 new student
开辟的空间

图**7.11**

在动态分配/撤销空间时，往往将这两个运算符和结构体结合使用，是很有效的。

可以看到： 要访问用**new**所开辟的结构体空间，无法直接通过变量名进行，只能通过指针**p**进行访问。如果要建立一个动态链表，必须从第一个结点开始，逐个地开辟结点并输入各结点数据，通过指针建立起前后相链的关系。

7.2 共用体

7.2.1 共用体的概念

有时需要使几种不同类型的变量存放到同一段内存单元中。例如，可把一个整型变量、一个字符型变量、一个双精度型变量放在同一个地址开始的内存单元中（见图**7.12**）。

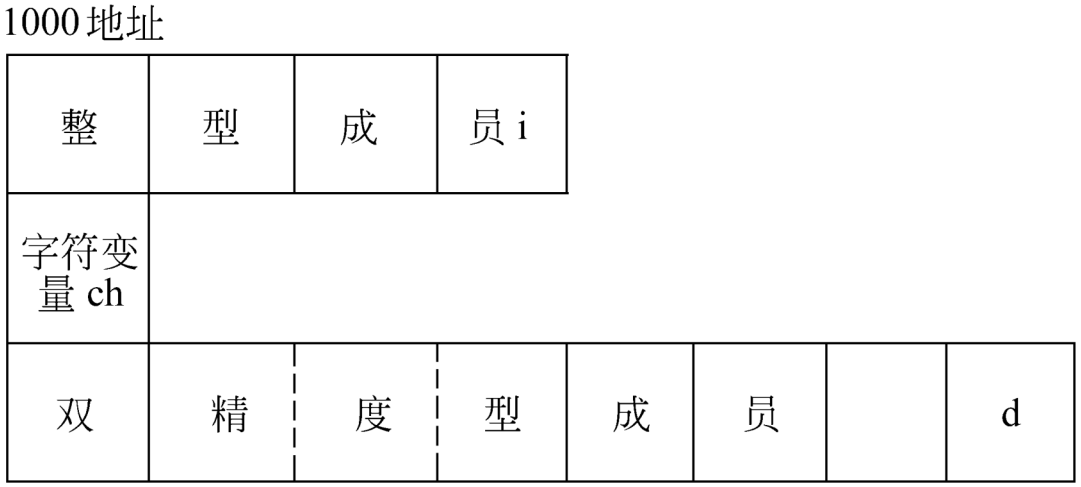


图7.12

以上3个变量在内存中占的字节数不同，但都从同一地址开始存放。也就是使用覆盖技术，几个变量互相覆盖。这种使几个不同的变量共占同一段内存的结构，称为共用体(union)类型的结构(有些书译为联合)。

声明共用体类型的一般形式为

union 共用体类型名

｛成员表列

｝；

定义共用体变量的一般形式为

共用体类型名 共用体变量名;

当然也可在声明共用体类型的同时定义共用体变量，也可没有共用体类型名而直接定义共用体变量。

例如

```
union data
{ int i;
char ch;
double d;
} a,b,c;
```
(有共用体类型名)

```
union
{ int i;
char ch;
double d;
}a,b,c;
```
(无共用体类型名)

可以看到，“共用体”与“结构体”的定义形式相似。但它们的含义是不同的。结构体变量所占内存长度是各成员占的内存长度之和。每个成员分别占有其自己的内存单元。共用体变量所占的内存长度等于最长的成员的长度。

7.2.2 对共用体变量的访问方式

不能引用共用体变量，而只能引用共用体变量中的成员。例如，下面的引用方式是正确的：

a.i（引用共用体变量中的整型成员**i**）

a.ch（引用共用体变量中的字符型成员**ch**）

a.f（引用共用体变量中的双精度型成员**d**）

不能只引用共用体变量，例如

```
cout<<a;
```

是错误的，应该写成**cout<<a.i;**或**cout<<a.ch;**等。

7.2.3 共用体类型数据的特点

(1) 使用共用体变量的目的是希望用同一个内存段存放几种不同类型的数据。但请注意： 在每一瞬时只能存放其中一种，而不是同时存放几种。

(2) 能够访问的是共用体变量中最后一次被赋值的成员，在对一个新的成员赋值后原有的成员就失去作用。

(3) 共用体变量的地址和它的各成员的地址都是同一地址。

(4) 不能对共用体变量名赋值；不能企图引用变量名来得到一个值；不能在定义共用体变量时对它初始化；不能用共用体变量名作为函数参数。

例**7.7** 设有若干个人员的数据，其中有学生和教师。学生的数据中包括： 姓名、号码、性别、职业、年级。教师的数据包括： 姓名、号码、性别、职业、职务。可以看出，学生和教师所包含的数据是不同的。现要求把它们放在同一表格中，见图**7.13**。

name	num	sex	job	class(班) position(职务)
Li	1011	f	s	501
wang	2085	m	t	prof

图**7.13**

如果**job**项为**s**（学生），则第**5**项为**grade**（年级）。即**Li**是**3**年级的。如果**job**项是**t**（教师），则第**5**项为**position**（职务）。**Wang**是**prof(**教授**)**。显然对第**5**项可以用共用体来处理（将**class**和**position**放在同一段内存中）。

要求输入人员的数据，然后再输出。为简化起见，只设两个人（一个学生、一个教师）。

程序如下：

```
#include <iostream>
#include <string>
#include <iomanip>          //因为在输出流中使用了控制符setw
using namespace std;
struct
{ int num;
char name[10];
```

```
char sex;
char job;
union P                 //声明共用体类型
{ int grade;            //年级
char position[10];      //职务
}category;              //成员category 为共用体变量
}person[2];             //定义共用体数组person，含两个元素

int main( )
{ int i;
for(i=0;i<2;i++)                                  //输入两个学生的数据
{cin>>person[i].num>>person[i].name>>person[i].sex>>person[i].job;
if(person[i].job=='s') cin>>person[i].category.grade; //若是学生则输入年级
else if (person[i].job=='t') cin>>person[i].category.position;//若是教师则输入职
务
}
cout<<endl<<"No.  Name   sex  job  grade/position"<<endl;
for(i=0;i<2;i++)
```

```
{if (person[i].job=='s')
cout<<person[i].num<<setw(6)<<person[i].name<<"   "<<person[i].sex
<<"   "<<person[i].job<<setw(10)<<person[i].category.grade<<endl;
else
cout<<person[i].num<<setw(6)<<person[i].name<<"   "<<person[i].sex
<<"   "<<person[i].job<<setw(10)<<person[i].category.position<<endl;
}
return 0;
}
```

运行情况如下：

101 Li fs 3↙ (注意在输入的字母**f**和**s**之间无空格)

102 Wang mt prof↙ (注意在输入的字母**m**和**t**之间无空格)

No.	Name	sex	job	grade/position
101	Li	f	s	3
102	Wang	m	t	prof

为了使输出结果上下对齐，在**cout**语句中用了**setw**控制符和插入空格。往往需要试验多次。

7.3 枚举类型

如果一个变量只有几种可能的值，可以定义为枚举**(enumeration)**类型。所谓“枚举”是指将变量的值一一列举出来，变量的值只能在列举出来的值的范围内。

声明枚举类型用**enum**开头。例如

```
enum weekday{sun，mon，tue，wed，thu，fri，sat};
```

上面声明了一个枚举类型**weekday**，花括号中**sun,mon,…,sat**等称为枚举元素或枚举常量。表示这个类型的变量的值只能是以上**7**个值之一。它们是用户自己定义的标识符。

声明枚举类型的一般形式为

enum 枚举类型名 **{**枚举常量表列**};**

在声明了枚举类型之后，可以用它来定义变量。如

weekday workday,week_end;

这样，**workday**和**week_end**被定义为枚举类型**weekday**的变量。

在**C**语言中，枚举类型名包括关键字**enum**，以上的定义可以写为

enum weekday workday,week_end;

在**C++**中允许不写**enum**，一般也不写**enum**，但保留了**C**的用法。

根据以上对枚举类型**weekday**的声明，枚举变量的值只能是**sun**到**sat**之一。例如

workday=mon;

week_end=sun;

是正确的。也可以直接定义枚举变量，如

enum{sun，mon，tue，wed，thu，fri，sat} workday,week_end；

这些标识符并不自动地代表什么含义。

说明:

(1) 对枚举元素按常量处理，故称枚举常量。

(2) 枚举元素作为常量，它们是有值的，**C++**编译按定义时的顺序对它们赋值为**0,1,2,3,…**。也可以在声明枚举类型时另行指定枚举元素的值。

(3) 枚举值可以用来做判断比较。

(4) 一个整数不能直接赋给一个枚举变量。

例7.8 口袋中有红、黄、蓝、白、黑5种颜色的球若干个。每次从口袋中任意取出3个球，问得到3种不同颜色的球的可能取法，输出每种排列的情况。

```
#include <iostream>
#include <iomanip>                    //在输出时要用到setw控制符
using namespace std;
int main( )
{ enum color {red,yellow,blue,white,black};     //声明枚举类型color
color pri;                            /定义color类型的变量pri
int i,j,k,n=0,loop;                     //n是累计不同颜色的组合数
for (i=red;i<=black;i++)                  //当i为某一颜色时
for (j=red;j<=black;j++)                  //当j为某一颜色时
if (i!=j)                           /若前两个球的颜色不同
{ for (k=red;k<=black;k++) //只有前两个球的颜色不同，才需要检查第3个球的颜色
if ((k!=i) && (k!=j))   //3个球的颜色都不同
{n=n+1;              //使累计值n加1
cout<<setw(3)<<n;       //输出当前的n值，字段宽度为3
for (loop=1;loop<=3;loop++)   //先后对3个球作处理
{switch (loop)           //loop的值先后为1,2,3
```

```
{case 1: pri=color(i);break;  //color(i)是强制类型转换，使pri的值为i
case 2: pri=color(j);break;  //使pri的值为j
case 3: pri=color(k);break;  //使pri的值为k
default:break;
}
switch (pri)        //判断pri的值，输出相应的“颜色”
{case red:    cout<<setw(8)<<"red"; break;
case yellow: cout<<setw(8)<<"yellow"; break;
case blue:   cout<<setw(8)<<"blue"; break;
case white:  cout<<setw(8)<<"white"; break;
case black:  cout<<setw(8)<<"black"; break;
default   :                  break;
}
}
cout<<endl;
}
}
cout<<"total:"<<n<<endl;       //输出符合条件的组合的个数
return 0;
}
```

运行结果如下：

```
1       red       yellow    blue
2        red      yellow    white
3        red      yellow    black
         ¦         ¦         ¦
58      black     white     red
59      black     white     yellow
60      black     white     blue
total:60
```

不用枚举常量,而用常数0代表“红”，1代表“黄”……也可以。但显然用枚举变量更直观，因为枚举元素都选用了令人“见名知意”的标识符，而且枚举变量的值限制在定义时规定的几个枚举元素范围内，如果赋予它一个其他的值，就会出现出错信息，便于检查。

7.4 用typedef声明类型

除了用以上方法声明结构体、共用体、枚举等类型外，还可以用**typedef**声明一个新的类型名来代替已有的类型名。如

```
typedef int INTEGER;          //指定用标识符INTEGER代表int类型
typedef float REAL;           //指定用REAL代表float类型
```

这样，以下两行等价：

```
① int     i,j;  float a,b;
② INTEGER i,j;   REAL  a,b;
```

这样可以使熟悉**FORTRAN**的人能用**INTEGER**和**REAL**定义变量，以适应他们的习惯。

如果在一个程序中，整型变量是专门用来计数的，可以用**COUNT**来作为整型类型名：

```
typedef int COUNT;      //指定用COUNT代表int型
COUNT  i,j;             //将变量i,j定义为COUNT类型，即int类型
```

在程序中将变量**i,j**定义为**COUNT**类型，可以使人更一目了然地知道它们是用于计数的。

也可以声明结构体类型：

```
typedef struct          //注意在struct之前用了关键字typedef，表示是声明新名
{ int month;
int day;
int year;
} DATE;                 //注意DATE是新类型名，而不是结构体变量名
```

所声明的新类型名**DATE**代表上面指定的一个结构体类型。这样就可以用**DATE**定义变量：

```
DATE birthday;
DATE  *p;               //p为指向此结构体类型数据的指针
```

还可以进一步：

① **typedef int NUM[100]；** //声明**NUM**为整型数组类型，包含**100**个元素

NUM n； //定义**n**为包含**100**个整型元素的数组

② **typedef char *STRING；** //声明**STRING**为字符指针类型

STRING p,s[10]； //**p**为字符指针变量，**s**为指针数组(有**10**个元素)

③ **typedef int (*POINTER)( )** //声明**POINTER**为指向函数的指针类型，函数返回整型值

POINTER p1，p2; // **p1，p2**为**POINTER**类型的指针变量

归纳起来，声明一个新的类型名的方法是：

① 先按定义变量的方法写出定义语句（如**int i;**）。

② 将变量名换成新类型名（如将**i**换成**COUNT**）。

③ 在最前面加**typedef**(如**typedef int COUNT**)。

④ 然后可以用新类型名去定义变量。

再以声明上述的数组类型为例来说明：

① 先按定义数组形式书写： **int n[100];**

② 将变量名**n**换成自己指定的类型名： **int NUM[100];**

③ 在前面加上**typedef**，得到**typedef int NUM[100];**

④ 用来定义变量： **NUM n;(n**是包含**100**个整型元素的数组**)**。

习惯上常把用**typedef**声明的类型名用大写字母表示，以便与系统提供的标准类型标识符相区别。

说明：

(1) typedef可以声明各种类型名，但不能用来定义变量。用**typedef**可以声明数组类型、字符串类型，使用比较方便。

(2) 用**typedef**只是对已经存在的类型增加一个类型名，而没有创造新的类型。

(3) 当在不同源文件中用到同一类型数据（尤其是像数组、指针、结构体、共用体等类型数据）时，常用**typedef**声明一些数据类型，把它们单独放在一个头文件中，然后在需要用到它们的文件中用#**include**命令把它们包含进来，以提高编程效率。

(4) 使用**typedef**有利于程序的通用与移植。有时程序会依赖于硬件特性，用**typedef**便于移植。