



# 第九章 程序设计复合类型-结构体

主讲教师：同济大学电子与信息工程学院 陈宇飞  
同济大学电子与信息工程学院 龚晓亮



# 目录

- 结构体类型
- 结构体变量的定义与初始化
- 结构体变量的使用
- 枚举
- 用typedef声明新类型



# 目录

- 结构体类型
  - 概念引入
  - 结构体类型



# 1.1 概念引入

- ✓ C++语言有丰富的系统预定义的基本数据类型
  - 基本类型：整型、实型、字符型、指针……



# 衍生问题

学号	姓名	年龄	课程成绩
1	张一	18	80.5
2	李二	18	76
3	王三	19	90.5
4	陈四	17	88

➤ 每列的数据类型不同，如何表示该二维数据？

1. 逐一定义基本变量？



• 学生1:

```
int num_1;  
char name_1[20];  
int age_1;  
float score_1;
```

• 学生2:

```
int num_2;  
char name_2[20];  
int age_2;  
float score_2;
```

.....

• 学生n:

```
int num_n;  
char name_n[20];  
int age_n;float score_n;
```

## 1. 逐一定义基本变量:



1. 工作量巨大, 不现实

2. n个单变量, 修改不便, 无法循环处理



# 衍生问题

学号	姓名	年龄	课程成绩
1	张一	18	80.5
2	李二	18	76
3	王三	19	90.5
4	陈四	17	88

➤ 每列的数据类型不同，如何表示该二维数据？

2. 用数组表示？



## 2. 用数组表示:

```
int num [4];  
char name [4][20];  
int age [4];  
float score [4];
```



1. 数组表现的数据类型必须相同，无法体现每一个学生（独立个体）的信息
2. 操作（输入输出）时容易信息错位





# 1.1 概念引入

- ✓ C++语言不仅有丰富的系统预定义的基本数据类型
  - 基本类型：整型、实型、字符型、指针……
- ✓ 而且允许用户在现有数据类型的基础上进行数据类型的自定义
  - 自定义数据类型：数组、结构体（struct）、联合体（union）、枚举（enum）



## 1.2 结构体类型

✓ 结构体：将不同性质类型但是互相有关联的数据放在一起，组合成一种新的复合型数据类型，称为结构体类型（结构体）

	储存数据	数据类型	格式灵活
数组	多个	相同	
结构体	多个	不同	✓



# 1.2 结构体类型

✓ 结构体类型声明的一般形式:

关键字

**struct** **结构体类型名**  $\rightarrow$  由用户自定义 (见名知义)

类型名 成员名1;

类型名 成员名2;

类型名 成员名3;

.....

结构体成员列表

举例

**};** 结束结构体声明

```
struct student
{
    int num;
    char name[20];
    int age;
    float score;
};
```



## 1.2 结构体类型

### 注意：

- 结构体类型是一种构造（自定义）数据类型，它和整型int、字符型char等系统定义的基本数据类型具有相同地位，不同的是它是由用户自定义的
- 结构体类型的声明仅仅是声明了一个类型，系统并不为之分配内存，只有当使用这个类型定义了变量时，系统才会为变量分配内存（等同于int类型的声明，不是变量定义）



# 目录

- 结构体变量的定义与初始化
  - 结构体变量的定义
  - 结构体变量的初始化



## 2.1 结构体变量的定义

✓ 方法一：先声明结构体类型再定义变量名

```
struct 结构体类型名  
{
```

类型名 成员名1;

类型名 成员名2;

类型名 成员名3;

}; //先声明结构体类型

举例



```
struct student  
{  
    int num;  
    char name [20];  
    int age;  
    float score;  
};  
struct student st1, st2;
```

struct 结构体类型名 变量1, 变量2; //再定义结构体变量



## 2.1 结构体变量的定义

✓ 方法一：先声明结构体类型再定义变量名

```
struct 结构体类型名  
{
```

类型名 成员名1;

类型名 成员名2;

类型名 成员名3;

}; //先定义结构体类型

举例



```
struct student  
{  
    int num;  
    char name [20];  
    int age;  
    float score;  
};  
student st1, st2;
```

**struct** 结构体类型名 变量1, 变量2; //再定义结构体变量



在C语言中，要在结构体类型前加关键字struct，C++中则可以省略关键字，较为方便



## 2.1 结构体变量的定义

✓ 方法二：在声明结构体类型的同时定义变量名

```
struct 结构体类型名  
{
```

类型标识符 成员名1;

类型标识符 成员名2;

类型标识符 成员名3;

} 变量1, 变量2; //紧接着定义结构体变量

举例



```
struct student  
{  
    int num;  
    char name [20];  
    int age;  
    float score;  
} st1, st2;
```

可以继续用方法1定义st3、st4





## 2.1 结构体变量的定义

✓ 方法三：直接定义结构体变量名（不出现结构体名称）

struct //此处不出现结构体类型名

{

类型标识符 成员名1;

类型标识符 成员名2;

类型标识符 成员名3;

} 变量1, 变量2;

举例



```
struct
{
    int num;
    char name [20];
    int age;
    float score;
} st1, st2;
```

无法继续用方法1定义st3、st4



## 2.1 结构体变量的定义

关于三种方法的使用：

- 第三种方法虽然合法，但很少使用：如果只需要 st1、st2 两个变量，后面不需要再使用结构体名定义其他变量，可以使用方法三
- 如果程序比较简单，结构体类型只在本程序中使用的情况下，可以使用第二种方法
- 第一种方法将声明结构体类型和定义结构体变量分开，便于不同的函数甚至不同的程序文件都可以使用声明的结构体类型。（**推荐使用第一种方法**）



## 2.1 结构体变量的定义

- ✓ 结构体**类型**和结构体**变量**是不同的概念，只能对变量进行操作
- ✓ 结构体变量所占内存空间是其全体成员所占内存**总和**

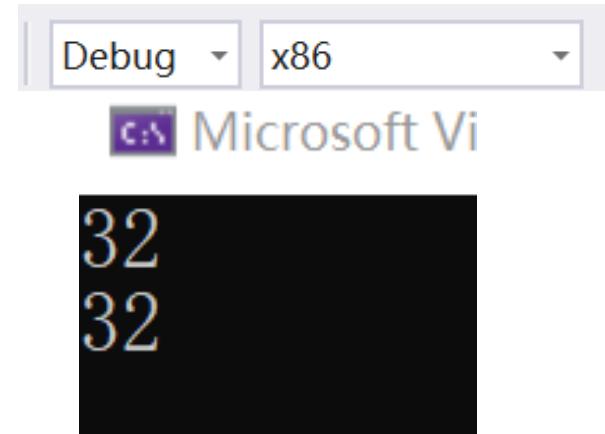
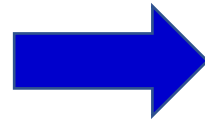
```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
};
student st1, st2;
```

**st1和st2在内存中各占32个字节  
(32=4+20+4+4)**



✓ 结构体变量所占内存空间是其全体成员所占内存**总和**

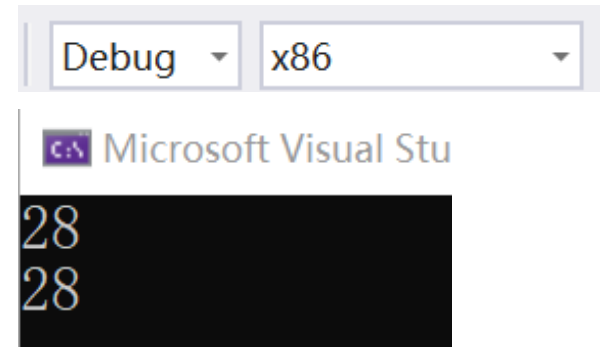
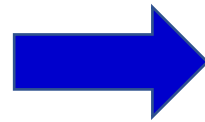
```
#include <iostream>
using namespace std;
struct student
{
    int num;
    char name[20];
    int age;
    float score;
};
int main()
{
    struct student st1, st2;
    cout << sizeof (st1) << endl;
    cout << sizeof (st2) << endl;
    return 0;
}
```





✓ 结构体变量所占内存空间是其全体成员所占内存**总和**

```
#include <iostream>
using namespace std;
struct student
{
    short num;
    char name[20];
    char age;
    float score;
};
int main()
{
    struct student st1, st2;
    cout << sizeof (st1) << endl;
    cout << sizeof (st2) << endl;
    return 0;
}
```



Q: 为什么不是 $2+20+1+4=27$ ?

A: 内存对齐!



# \*补充内容：结构体类型声明与字节对齐

1. 内存对齐的基本概念：为保证CPU的运算稳定和效率，要求基本数据类型在内存中的存储地址必须**对齐**，即基本数据类型的变量不能简单的存储于内存中的任意地址处，该变量的起始地址必须是该类型大小的**整数倍**
  - 32位编译系统下，int型数据的起始地址是4的倍数，short型数据的起始地址是2的倍数，double型数据的起始地址是8的倍数，指针变量的起始地址是4的倍数
  - 64位编译系统下，指针变量的起始地址是8的倍数

# \*补充内容：结构体类型声明与字节对齐



## 2. 结构体的成员对齐：

- 结构体类型的**起始地址**，必须是所有数据成员中占**最大字节**的基本数据类型的**整数倍**
- 结构体类型的**所有数据成员的大小总和**，必须是所有数据成员中占**最大字节**的基本数据类型的**整数倍**，因此可能会有**填充字节**
- 结构体类型中**各数据成员**的起始地址，必须是该类型大小的**整数倍**，因此可能会有**填充字节**



//例1: 结构体声明与字节对齐

```
#include <iostream>
```

```
using namespace std;
```

```
struct s1 {
```

```
    int    a;
```

```
    short b;
```

```
    char  c;
```

```
};
```

```
struct s2 {
```

```
    short b;
```

```
    int   a;
```

```
    short c;
```

```
};
```

```
struct s3 {
```

```
    int a;
```

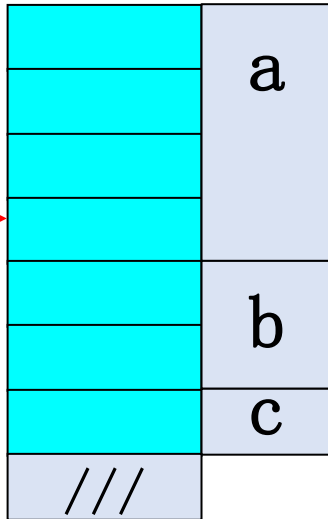
```
    short b;
```

```
    short c;
```

```
};
```

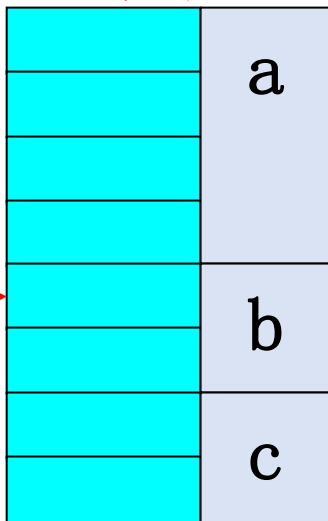
8字节

7



8

8字节



8

```
int main()
```

```
{
```

```
    cout << sizeof(s1) << endl;
```

```
    cout << sizeof(s2) << endl;
```

```
    cout << sizeof(s3) << endl;
```

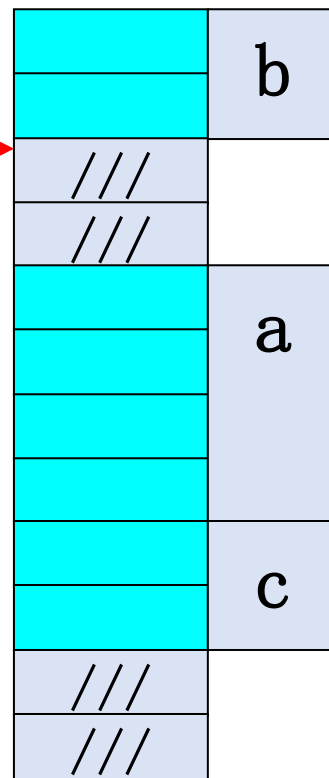
```
}
```

8

12

8

12字节



- 1、起始地址必须是4的倍数
- 2、总大小必须是4的倍数
- 3、每个成员的起始地址必须是4/2/1的倍数





Microsoft Visual Studio 调试控制台

8  
12  
8

//例1: 结构体声明与字节对齐

```
#include <iostream>
```

```
using namespace std;
```

```
struct s1 {
```

```
    int    a;
```

```
    short b;
```

```
    char  c;
```

7

```
};
```

```
struct s2 {
```

```
    short b;
```

```
    int   a;
```

```
    short c;
```

8

```
};
```

```
struct s3 {
```

```
    int a;
```

```
    short b;
```

```
    short c;
```

8

```
};
```

```
int main()
```

```
{
```

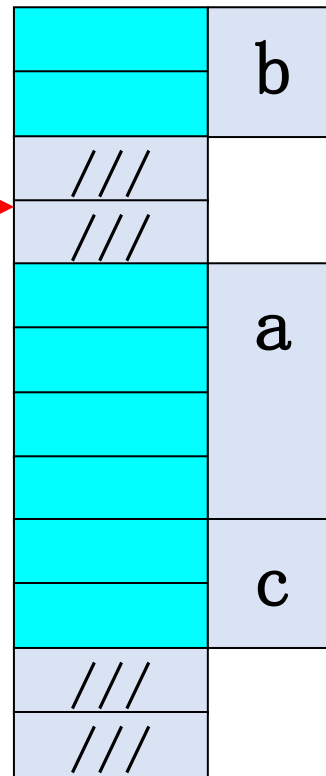
```
    cout << sizeof(s1) << endl;
```

```
    cout << sizeof(s2) << endl;
```

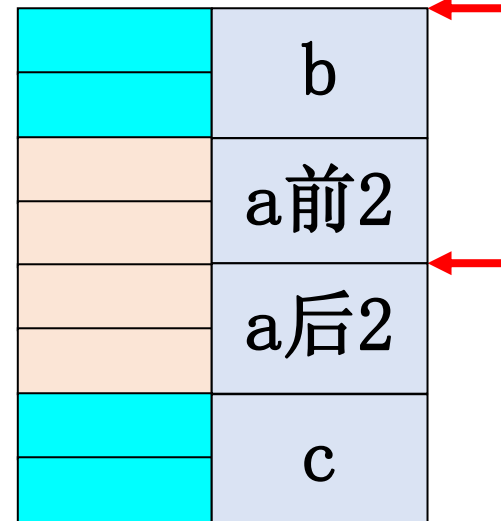
```
    cout << sizeof(s3) << endl;
```

```
}
```

12字节



8字节



假设无填充字节，  
则读a需要两个CPU  
时钟周期

=>节约4字节，  
速度慢一半



## //例2：结构体声明与字节对齐

```
#include <iostream>
```

```
using namespace std;
```

```
struct s1 {
```

```
    char    a;
```

```
    int     b;
```

```
    double  c;
```

```
};
```

```
struct s2 {
```

```
    char    a;
```

```
    double  c;
```

```
    int     b;
```

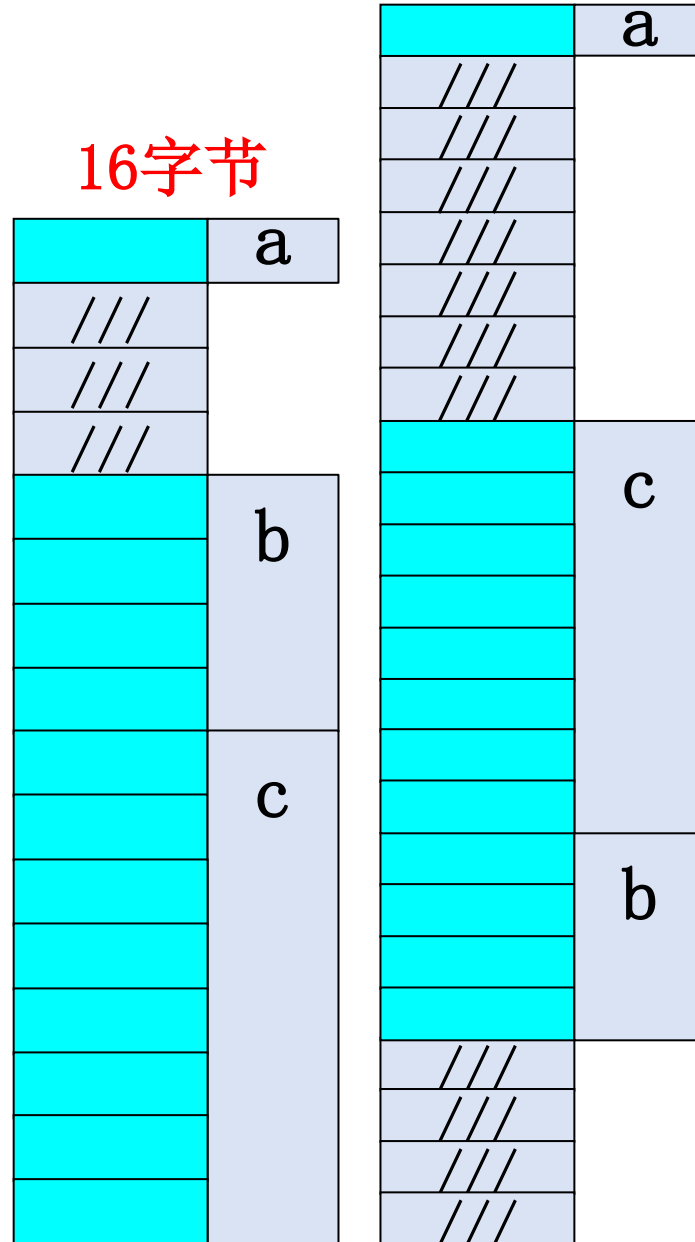
```
};
```

16字节

13

13

24字节



```
int main()
```

```
{
```

```
    cout << sizeof(s1) << endl;
```

```
    cout << sizeof(s2) << endl;
```

```
}
```

Mic

16

24



### //例3：结构体声明与字节对齐

```
#include <iostream>
```

```
using namespace std;
```

```
struct s1 {
```

```
    char a[5];
```

```
    char b[3];
```

```
    int c;
```

```
};
```

```
struct s2 {
```

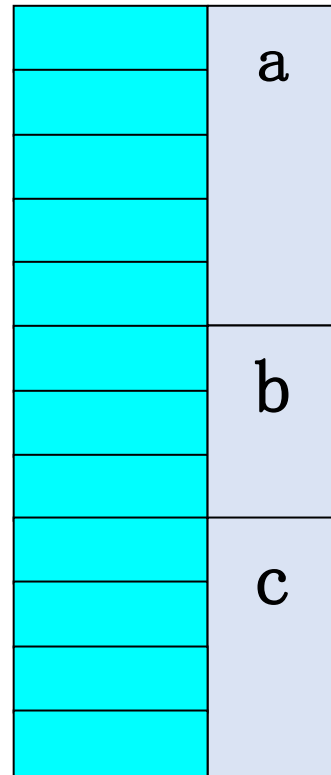
```
    char a[5];
```

```
    int c;
```

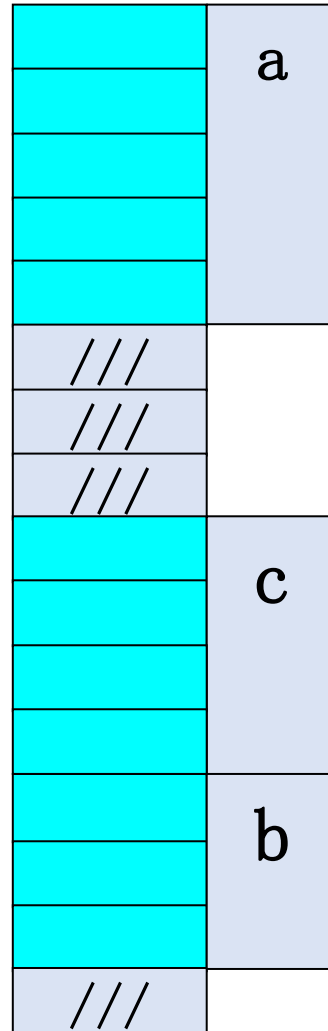
```
    char b[3];
```

```
};
```

12字节



16字节



```
int main()
```

```
{
```

```
    cout << sizeof(s1) << endl;
```

```
    cout << sizeof(s2) << endl;
```

```
}
```

12

16



## ✓ 结构体声明的位置很重要

```
#include <iostream>
using namespace std;
int main()
{
    struct student
    {
        int num;
        char name[20];
        int age;
        float score;
    };
    struct student st1, st2;
    cout << sizeof (st1) << endl;
    cout << sizeof (st2) << endl;
    return 0;
}
```

内部声明：只能被该声明所属的函数使用

```
#include <iostream>
using namespace std;
struct student
{
    int num;
    char name[20];
    int age;
    float score;
};
int main()
{
    struct student st1, st2;
    cout << sizeof (st1) << endl;
    cout << sizeof (st2) << endl;
    return 0;
}
```

外部声明：可以被其后面所有的函数使用



## ✓ 结构体声明的位置很重要

1. 常规建议放在函数外部
2. 如果多源文件，把结构体声明放在头文件中，然后在需要用这个结构体的源文件中包含对应的头文件

```
#include <iostream>
using namespace std;
struct student 外部声明：可以被其
                后面所有的函数使用
{
    int num;
    char name[20];
    int age;
    float score;
};
int main()
{
    struct student st1, st2;
    cout << sizeof (st1) << endl;
    cout << sizeof(st2) << endl;
    return 0;
}
```



✓ 结构体成员本身也可以是另一个结构体（结构体嵌套）

例如：先声明结构体date，再使用结构体date

```
struct date
{
    int year;
    int month;
    int day;
};
struct student
{
    int num;
    char name[20];
    date birthday; //date是结构体类型，birthday是date类型的成员
};
```



✓ 结构体成员本身也可以是另一个结构体（结构体嵌套）

再如：不事先声明，直接嵌套

```
struct student
{
    int num;
    char name[20];
    struct date
    {
        int year;
        int month;
        int day;
    } birthday;
};
```



## ✓ 不允许对结构体本身递归定义

例如:

```
struct student
{
    int age;
    struct student stu; ✗
} st1, st2;
```

结构体变量是有大小的，不能定义  
“一个结构体变量的大小等于一个  
整型变量的大小加上它本身的大小”  
(无法递归求自身大小!)

- ✓ 可以定义成员为结构体自身的指针 `struct student *next;` 因为指针所占内存大小是确定的 (后续课程的动态申请、链表都会用到)





## 2.2 结构体变量的初始化

✓ 方法一：先声明结构体类型，再定义结构体变量并赋初值

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
}; //先声明结构体类型
student st1={1001, "AA", 18, 82}; //再定义变量并初始化
```

## ➤例1:



```
struct student
```

```
{
```

```
    int num;
```

```
    char name[20];
```

```
    struct date
```

```
    {
```

```
        int year;
```

```
        int month;
```

```
        int day;
```

```
    } birthday;
```

```
} st1;
```

```
st1={1001, "AA", {2000, 1, 19}}; //嵌套型结构体变量的初始化
```

此处大括号可以不加，  
但是加上可读性更好



## 2.2 结构体变量的初始化

✓ 方法二：在声明结构体类型的同时定义结构体变量并赋初值

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
} st1={1001, "AA", 18, 82}; //定义结构体变量的同时初始化
```



## ➤例2:

```
#include <iostream>
using namespace std;
struct student
{
    int num;
    char name[20];
    int age;
    float score;
} st1 = { 1001, "AA", 18, 82 };
int main()
{
    cout << st1.num << ' ' << st1.name << ' ' << st1.age << ' ' << st1.score << endl;
    return 0;
}
```

1. 一一对应赋值
2. 不允许跳跃赋值



## ➤例2:

```
#include <iostream>
using namespace std;
struct student
```

```
{
```

```
    int num;
```

```
    char name[20];
```

```
    int age;
```

```
    float score;
```

```
} st1 = { 1001, "AA" };
```

```
int main()
```

```
{
```

```
    cout <<st1.num<<' ' <<st1.name<<' ' <<st1.age<<' ' <<st1.score<<endl;
```

```
    return 0;
```

```
}
```

Microsoft Visual Studio 调试控制台

1001 AA 0 0

1. 一一对应赋值
2. 不允许跳跃赋值
3. 可以给成员部分赋值, 其他未赋值成员被设置为0



# 目录

- 结构体变量的使用
  - 结构体成员的访问
  - 结构体数组变量
  - 结构体指针变量
  - 将引用用于结构

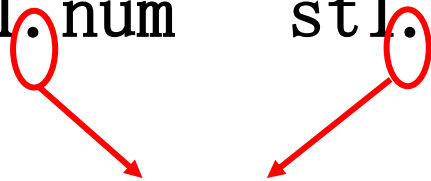


## 3.1 结构体成员的访问

- ✓ 结构体中的成员可以像基本变量那样赋值、输入输出、参与表达式计算等操作，这些操作统称为对结构体成员的访问
- ✓ 结构体变量成员表现形式：

结构体变量名. 成员名

例：st1.num    st1.age...



直接成员运算符，直接调用结构体中的某个成员（运算符优先级为2）

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
};
student st1, st2;
```



- ✓ 结构体嵌套中，成员本身是结构体时，结构体变量必须逐层引用：

```
struct student
{
    int num;
    char name[20];
    struct date
    {
        int year;
        int month;
        int day;
    } birthday;
} st1;
```

逐层找到最低一级成员名进行访问

st1.birthday.year

→ st1.birthday.month

st1.birthday.day

只能对最内层的成员进行运算、输入输出操作





- ✓ 不能整体输入输出一个结构体变量，只能对结构体变量中的各个成员分别进行输入输出：

```
struct student  
{
```

```
    int num;
```

```
    char name [20];
```

```
    int age;
```

```
    float score;
```

```
};
```

```
student st1 = {1001, "AA", 18, 82}, st2;
```

`cin>>st1; cout<<st1; ✗`

错误列表

整个解决方案

✗ 错误 2

⚠ 警告 0

i 27消息的 0

✗

生成 + IntelliSense

	代码	说明
▷ abc	E0349	没有与这些操作数匹配的 "<<" 运算符
✗	C2679	二进制"<<": 没有找到接受"student"类型的右操作数的运算符(或没有可接受的转换)

`cin>>st1.num; cout<<st1.num; ✓`



- ✓ 相同类型的结构体变量可以相互赋值，这个变量所有成员的值将全部赋值给另外一个变量的成员（整体赋值）

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
};
student st1, st2;
```

st1=st2



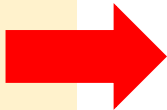
等价于

```
st1.num=st2.num
strcpy(st1.name, st2.name)
st1.age=st2.age
st1.score=st2.score
```



✓ 结构体变量的成员可以像普通变量一样进行各种操作

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
};
student st1, st2;
```



int i, *p;	s1.num; int *p;	
i++;	s1.num++;	自增/减
... + i*10 +...;	... + s1.num*10 +...;	各种 表达式
if (i>=10)	if (s1.num>=10)	
p = &i;	p = &s1.num;	取地址
scanf("%d", &i);	scanf("%d", &s1.num);	输入
cout << i;	cout << s1.num;	输出
fun(i);	fun(s1.num);	函数实参
return i;	return s1.num;	返回值

```
//例1:
#include <iostream>
using namespace std;
struct student //声明结构体类型student
{
    int num;
    char name[20];
    int age;
    float score;
};
int main()
{
    student st1 = { 1001, "AA", 18, 82 }, st2;
    //定义结构体变量st1, st2, 并对st1初始化
    st1.age++; //对结构体变量成员进行自增运算
    st2 = st1; //对结构体变量进行相互赋值（可以进行整体赋值）
    cout<<st2.num<<endl<<st2.name<<endl<<st2.age<<endl<<st2.score<<endl;
    //对结构体变量成员分别进行输出（不能进行整体输出）
    return 0;
}
```

```
1001
AA
19
82
```

//例2:

```
int main()
{
```

```
    inflatable guest = { "Glorious Gloria", 1.88, 29.99 };
```

//定义结构体变量guest并初始化

```
    inflatable pal = { "Audacious Arthur", 3.12, 32.99 };
```

//定义结构体变量pal并初始化

```
    cout << "Expand your guest list with " << guest.name;
```

```
    cout << " and " << pal.name << "!" << endl;
```

```
    cout << "You can have both for $" ;
```

```
    cout << guest.price + pal.price << "!" << endl;
```

```
    guest = pal; //结构体变量的相互赋值
```

```
    cout << guest.name << ' ' << guest.volume << ' ' << guest.price << endl;
```

```
    guest.price++; pal.price++;
```

```
    cout << guest.price + pal.price << endl; //对结构体变量成员的操作
```

```
    return 0;
```

```
}
```

```
Microsoft Visual Studio 调试控制台
Expand your guest list with Glorious Gloria and Audacious Arthur!
You can have both for $62.98!
Audacious Arthur 3.12 32.99
67.98
```

```
#include <iostream>
using namespace std;
struct inflatable
{
    char name[20];
    float volume;
    double price;
}; //声明结构体类型
```



## 3.2 结构体数组变量

✓ 结构体数组:

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
};
student st1, st2;
```

如果有100个学生信息录入，定义100个结构体变量？ st1, st2……st100

解决方法:

student st[100];

结构体数组：结构体数组中每个元素都是一个结构体类型，它们分别包括各个成员项



## 3.2 结构体数组变量

✓ 结构体数组的定义（参照结构体变量定义方法）

//方法1:

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
};
student st[n];
```

//方法2:

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
} st[n];
```

//方法3:

```
struct
{
    int num;
    char name [20];
    int age;
    float score;
} st[n];
```



## 3.2 结构体数组变量

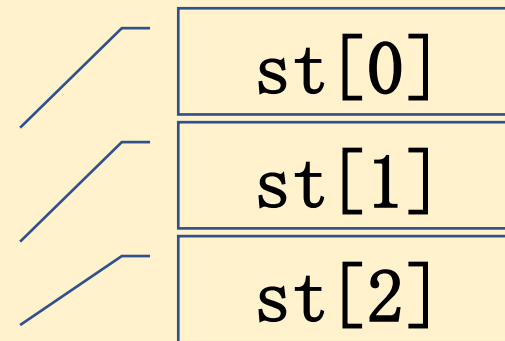
✓ 结构体数组的初始化

//方法1

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
};
```

```
student st[3] = {{1001, "AA", 18, 82},
                 {1002, "BB", 18, 76},
                 {1003, "CC", 19, 80}};
```

初始化写三行，可读性更好



//用逗号分隔每个元素的值，并将这些值用花括号括起来





## 3.2 结构体数组变量

✓ 结构体数组的初始化

//方法2

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
};
```

不推荐此用法

```
student st[3]={1001, "AA", 18, 82, 1002, "BB", 18, 76,
1003, "CC", 19, 80};
```

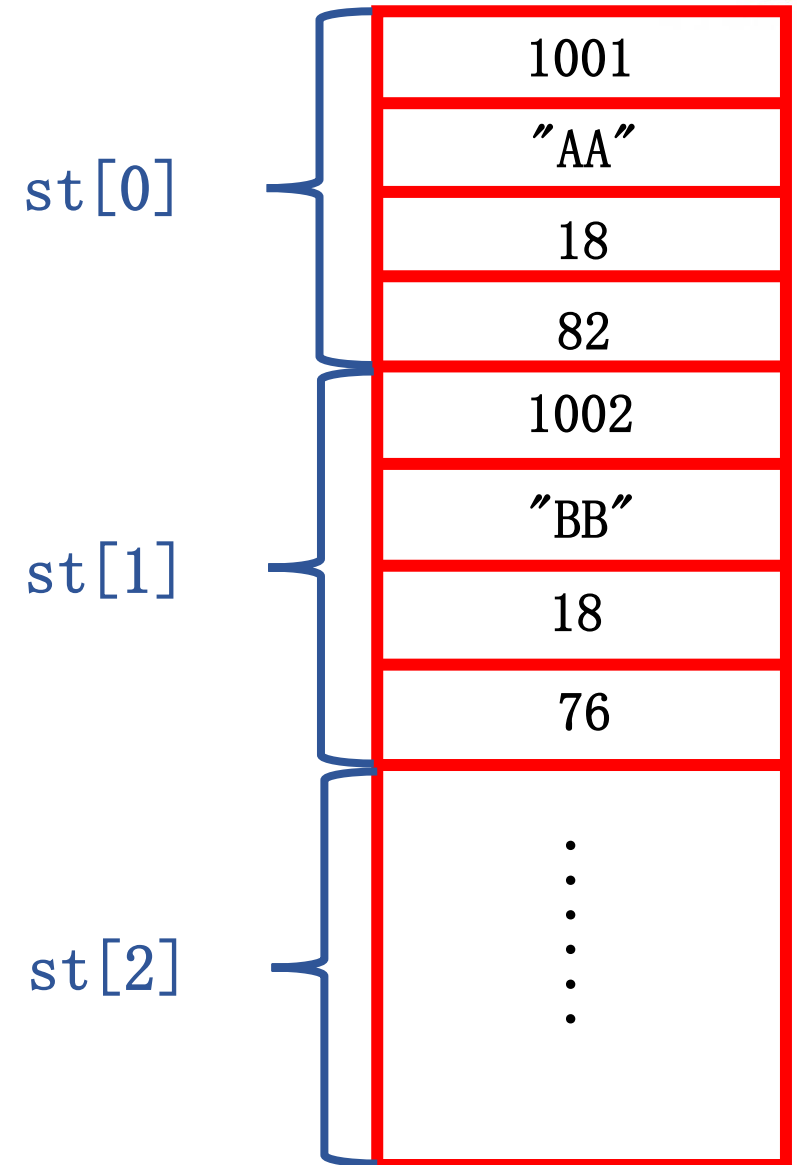
//用逗号分隔每个成员的值，并将这些值用花括号括起来



## 3.2 结构体数组变量

- ✓ 结构体数组在内存中是连续存放的

```
struct student
{
    int num;
    char name [20];
    int age;
    float score;
};
student st[3] = {{1001, "AA", 18, 82},
                 {1002, "BB", 18, 76},
                 {1003, "CC", 19, 80}};
```





## 3.2 结构体数组变量

### ✓ 访问结构体数组成员

```
struct student
{
    int num;
    char name [20];
    int age;
    float score[3];
};
struct student st[3]={
    {1001, "AA", 18, 82},
    {1002, "BB", 18, 76},
    {1003, "CC", 19, 80}};
```

st[0].score -结构体数组第一个元素的score成员

st[1].age -结构体数组第二个元素的age成员

st[2].num-结构体数组第三个元素的num成员

st[0].score[0]-结构体数组第一个元素的第一门成绩（逐个访问）

浮点型

结构体类型

只给第一门成绩赋初值，其余成绩默认值为0

```
//例1:
#include <iostream>
using namespace std;
struct student //声明结构体类型student
{
    int num;
    char name[20];
    int age;
    float score;
};
int main()
{
    struct student st[3] ;
    for (int i=0; i < 3; i++) //输入所有学生信息
        cin >> st[i].num >> st[i].name >> st[i].age >> st[i].score;
    for (int i=0; i < 3; i++) //输出所有学生信息
        cout << "学号:"<<st[i].num << ' ' << "姓名:"<<st[i].name << ' '
            << "年龄:" <<st[i].age << ' ' << "分数:"<< st[i].score << endl;
    return 0;
}
```

 Microsoft Visual Studio 调试控制台

```
200156 AA 18 82
200157 BB 18 76
200158 CC 19 80
学号:200156 姓名:AA 年龄:18 分数:82
学号:200157 姓名:BB 年龄:18 分数:76
学号:200158 姓名:CC 年龄:19 分数:80
```



- 例2:

一个小组中有3个学生，每个学生分别有3门成绩，编写程序依次输入学生的学号、姓名、年龄和三门课程的成绩信息，计算学生的平均成绩并输出每个学生的学号、姓名、年龄和三门课程的平均成绩

```
struct student
{
    int num;
    char name[20];
    int age;
    float score[3];
    float ave;
};
```

```
#include <iostream>
using namespace std;
int main()
{   int i, j, k; student st[3];
    for (i = 0; i < 3; i++)
    {   cout << "请输入学生的学号、姓名和年龄信息"<<endl;
        cin >> st[i].num >> st[i].name >> st[i].age;
        float sum = 0;  //sum要清零
        for (j = 0; j < 3; j++)
        {   cout << "请输入学生的成绩信息" << endl;
            cin >> st[i].score[j];
            sum += st[i].score[j];
        }
        st[i].ave = sum / 3;
    }
    for (k = 0; k < 3; k++)
        cout <<"学号: " <<st[k].num <<' ' << "姓名: " << st[k].name <<' '
            <<"年龄: " << st[k].age<<' ' <<"平均分: " <<st[k].ave << endl;
    return 0;
}
```



```
#include <iostream>
using namespace std;
int main()
{
    int i, j, k; student st[3];
    for (i = 0; i < 3; i++)
    {
        cout << "请输入学生的学号: ";
        cin >> st[i].num >> st[i].name >> st[i].age;
        float sum = 0; //sum要
        for (j = 0; j < 3; j++)
        {
            cout << "请输入学生" + st[i].name + "的第" + j + "次成绩: ";
            cin >> st[i].score[j];
            sum += st[i].score[j];
        }
        st[i].ave = sum / 3;
    }
    for (k = 0; k < 3; k++)
        cout << "学号: " << st[k].num << " 姓名: " << st[k].name << " 年龄: " << st[k].age << " 平均分: " << st[k].ave << endl;
    return 0;
}
```

```
请输入学生的学号、姓名和年龄信息
1001 AA 18
请输入学生的成绩信息
90
请输入学生的成绩信息
85
请输入学生的成绩信息
80
请输入学生的学号、姓名和年龄信息
1002 BB 18
请输入学生的成绩信息
90
请输入学生的成绩信息
90
请输入学生的成绩信息
90
请输入学生的学号、姓名和年龄信息
1003 CC 18
请输入学生的成绩信息
95
请输入学生的成绩信息
95
请输入学生的成绩信息
95
学号: 1001 姓名: AA 年龄: 18 平均分: 85
学号: 1002 姓名: BB 年龄: 18 平均分: 90
学号: 1003 姓名: CC 年龄: 18 平均分: 95
```

d1;



- 例3: 改写上述程序, 将程序中的部分功能定义为函数, 实现相同的目的

```
#define N 3
#define M 3
struct student{
    int num;
    char name[20];
    int age;
    float score[M];
    float ave;
}stu[N] = {{1001, "AA", 18, 90, 85, 80},
           {1002, "BB", 18, 90, 90, 90},
           {1003, "CC", 18, 95, 95, 95}};
void aver(struct student arr[], int n); //形参为结构体数组
```



## 更好的实现方式

```
#include <iostream>
using namespace std;
int main()
{
    ...
    aver (stu, N); //实参为结构体数组名
    for (int k = 0; k < N; k++)
        cout << "学号: " << stu[k].num << "姓名: " << stu[k].name
            << "年龄: " << stu[k].age << "平均分: " << stu[k].ave << endl;
    return 0;
}

void aver(student arr[], int n)
{
    float sum;
    for (int i = 0; i < N; i++)
    {
        sum = 0;
        for (int j = 0; j < M; j++)
            sum += stu[i].score[j];
        stu[i].ave = sum / 3.0;
    }
}
```

 Microsoft Visual Studio 调试控制台

```
学号: 1001姓名: AA年龄: 18平均分: 85
学号: 1002姓名: BB年龄: 18平均分: 90
学号: 1003姓名: CC年龄: 18平均分: 95
```



## 3.2 结构体数组变量

### 结构体数组作函数参数

- ✓ 数组名代表数组首元素地址
- ✓ 用数组名做函数参数，实际上传递的是数组首元素的地址
- ✓ 系统编译时将数组名**按指针处理**
- ✓ 形参为数组时，从实参数组那里获得起始地址，因此形参数组和实参数组**共同占用一段内存单元**，调用函数时**形参的改变将影响实参**的值
- ✓ 形参 `struct student arr[]` 中方括号的数值并无实际作用（不分配存储单元），只是用 `arr[]` 这样的形式表明一个一维数组，接受实参的地址，编译时对括号内的数字不做理会，因此形参中的元素个数可以写也可以不写



## 3.3 结构体指针变量

- ✓ 定义一个指针变量，用来指向一个结构体变量，该指针的值就是这个结构体变量的起始地址
- ✓ 引入结构体指针的目的是为了实现函数之间的数据传递
- ✓ 指向结构体变量的指针变量的定义形式：

**结构体类型名 \*指针名**

- ✓ 注意和指向结构体成员中某个变量的指针进行区分

//例1:

```
struct student
{
```

```
    int num;
```

```
    char name[20];
```

```
    int age;
```

```
    float score;
```

```
} st={1001, "AA", 18, 82};
```

```
struct student st, *p;
```

```
int *p1;
```

```
float *p2;
```

```
p = &st;           //指针指向结构体变量st(++ ⇔ +32)
```

```
p1 = &st.age;      //指针指向结构体变量的age成员(++ ⇔ +4)
```

```
p2 = &st.score;    //指针指向结构体变量的score成员(++ ⇔ +4)
```

```
struct student
{
    int num;
    char name[20];
    int age;
    float score;
} st={1001, "AA", 18, 82};
```

```
struct student st, *p;
```

```
p=&st; ✓
```

```
p=&st.num; ✗ （左右地址的基类型不匹配）
```

: error C2440: “=”: 无法从“int\*”转换为“main::student\*”

message: 与指向的类型无关; 强制转换要求 reinterpret\_cast、C 样式强制转换或函数样式强制转换

```
p=(struct student *)&st.num; ✓ 进行强制类型转换后正确
```



## 3.3 结构体指针变量

✓ 指向结构体变量的指针变量的定义形式:

结构体类型名 \*指针名

✓ 利用指向结构体变量的指针引用其成员的方式

1. (\*结构体指针). 成员名

例: (\*p). num: 括号不能省略, 括号作用是优先执行取内容操作

(\*p). num  $\xleftrightarrow[\text{结合性}]{\text{优先级}}$  st. num ✓

(\*p). num  $\xleftrightarrow[\text{不加括号?}]{\text{结合性}}$  \*p. num  $\xleftrightarrow{\text{优先级}}$  \*(p. num) ✗



## 3.3 结构体指针变量

✓ 指向结构体变量的指针变量的定义形式:

结构体类型名 \*指针名

✓ 利用指向结构体变量的指针引用其成员的方式

2. 结构体指针->成员名 (->为指向运算符)

例: p->num 指针变量p指向结构体变量st中的num成员

p->num 最终代表的就是num这个成员的内容

```
struct student
{
    int num;
    char name[20];
    int age;
    float score;
} st={1001, "AA", 18, 82};
student *p=&st; //定义p为指向student类型数据的指针变量并指向stu
```

利用指向结构体变量的指针引用其成员的方式:

1. `(*p).num`

2. `p->num`





//例2:

```
#include <iostream>
```

```
using namespace std;
```

```
struct student
```

```
{    int num;
```

```
    char name[20];
```

```
    int age;
```

```
    float score;
```

```
} st = { 1001, "AA", 18, 82 };
```

```
int main()
```

```
{    student* p = &st; //定义p为指向student类型的指针变量并指向st
```

//引用结构体变量的成员的三种方式

```
cout<<st.num<<' ' <<st.name<<' ' <<st.age<<' ' <<st.score<<endl;
```

```
cout<<(*p).num<<' ' <<(*p).name<<' ' <<(*p).age<<' ' <<(*p).score<<endl;
```

```
cout<<p->num<<' ' <<p->name<<' ' <<p->age<<' ' <<p->score<<endl;
```

```
return 0;
```

```
}
```





## 3.3 结构体指针变量

✓ 利用指针变量引用结构体数组元素成员的方式:

```
struct student st[4];
```

```
struct student *p;
```

```
p=st;           // p=&st[0];
```

```
p->num;          // (*p).num;
```

```
(p+i)->num;     // (*(p+i)).num
```



```
st[i].num;
```

//定义结构体数组并初始化

```
struct student {  
    int num;  
    char name[20];  
    int age;  
    float score;  
} stu[2] = { {1001, "AA", 18, 90}, {1002, "BB", 18, 80} };
```

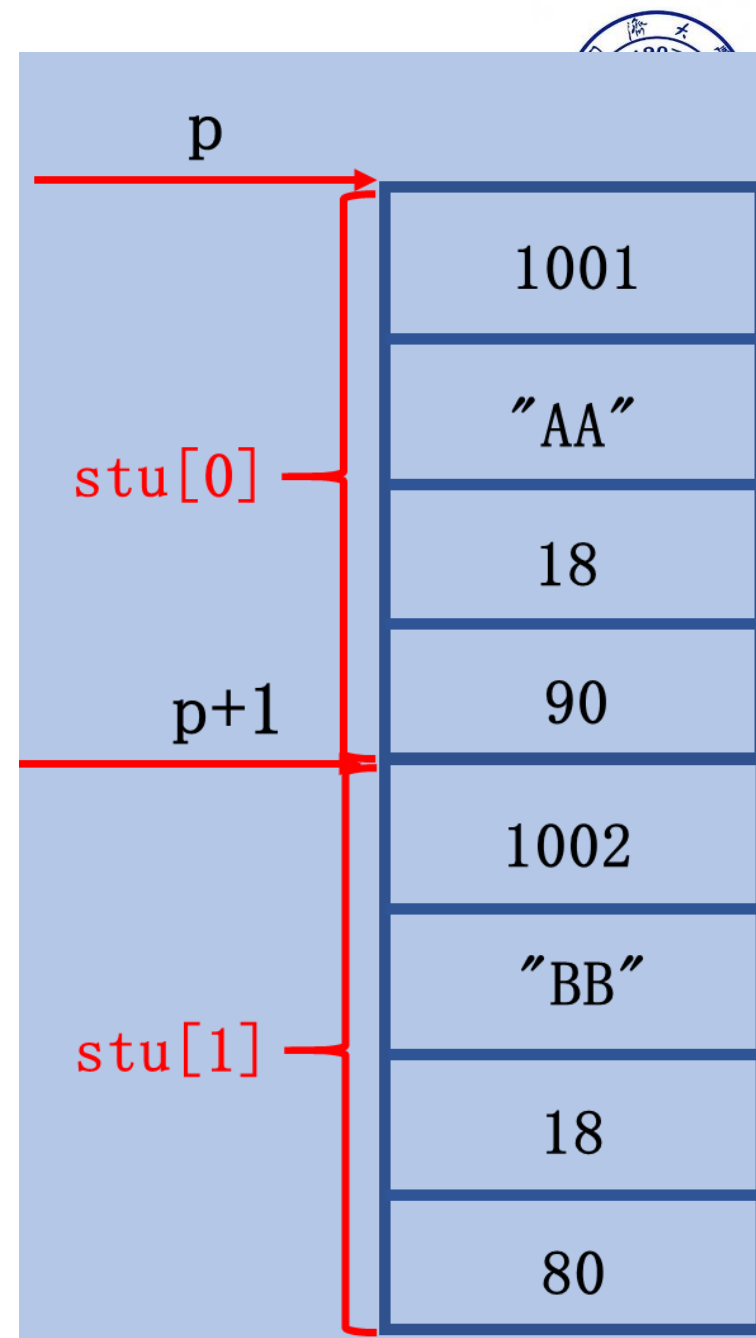
```
struct student* p = stu;
```

//利用指针指向结构体数组第一个元素的成员

```
p->num      p->name      p->age      p->score
```

//利用指针指向结构体数组第二个元素的成员

```
(p+1)->num  (p+1)->name  (p+1)->age  (p+1)->score
```



```
学号: 1001姓名: AA年龄: 18成绩: 90
学号: 1002姓名: BB年龄: 18成绩: 80
学号: 1001姓名: AA年龄: 18成绩: 90
学号: 1002姓名: BB年龄: 18成绩: 80
```

//例3:

```
#include <iostream>
```

```
using namespace std;
```

//同前定义结构体数组stu[2]并初始化

```
int main() {
```

```
    struct student* p = stu;
```

//利用指针指向结构体数组第一个元素的成员

```
    cout << "学号: " << p->num << "姓名: " << p->name << "年龄: " << p->age
        << "成绩: " << p->score<<endl;
```

//利用指针指向结构体数组第二个元素的成员

```
    cout << "学号: " << (p+1)->num << "姓名: " << (p+1)->name << "年龄: "
        << (p+1)->age << "成绩: " << (p+1)->score << endl;
```

//利用指针遍历结构体数组的所有成员

```
    for(int i=0;i<2;i++)
```

```
        cout << "学号: " << (p+i)->num << "姓名: " << (p+i)->name << "年龄: "
            << (p+i)->age << "成绩: " << (p+i)->score << endl;
```

```
    return 0;
```

```
}
```



## 3.3 结构体指针变量

### 结构体类型数据做函数参数

- ✓ 结构体变量名作函数参数
- ✓ 用指向结构体变量的指针作函数参数，将结构体变量的地址传给形参
- ✓ 用结构体数组作函数参数，将结构体数组的首地址传给形参

//例4:

```
#include<iostream>
```

```
using namespace std;
```

```
struct stud{
```

```
    long int num; float score;
```

```
};
```

```
void funvr(struct stud t) //注意结构体变量成员的引用形式
```

```
{    t.num = 2000101;
```

```
    t.score = 71.0;
```

```
}
```

```
void funar(struct stud t[]) //注意结构体数组元素的成员的引用形式
```

```
{    t[0].num = 3000101; t[0].score = 81.0;
```

```
    t[1].num = 3000102; t[1].score = 82.0;
```

```
}
```

```
void funpr(struct stud* t) //注意通过结构体指针变量引用成员的具体形式
```

```
{    t->num = 4000101;
```

```
    (*t).score = 92.0;
```

```
}
```

//例4续1:

```
int main()
```

```
{
```

```
    struct stud a[2] = {{1000101, 61.0}, {1000102, 62.0}};
```

```
    struct stud b = a[0];
```

//输出结构体变量b的成员的原有值

```
    cout << "old b:" << b.num << ' ' << b.score << endl;
```

```
    funvr(b); //将结构体变量作为函数的实参
```

```
    cout << "call funvr() new b: " << b.num << ' ' << b.score << endl;
```

```
    return 0;
```

```
}
```

```
old b:1000101 61
```

```
call funvr() new b: 1000101 61
```

```
void funvr(struct stud t)
{
    t.num = 2000101;
    t.score = 71.0;
}
```

//例4续2:

```
int main()
```

```
{
```

```
    struct stud a[2] = {{1000101, 61.0}, {1000102, 62.0}};
```

```
    struct stud b = a[0];
```

```
    funpr(&b); //将结构体变量的地址作为函数的实参
```

```
    cout<<"call funpr() new b: "<< b.num << ' ' <<b.score << endl;
```

```
    return 0;
```

```
}
```

```
void funpr(struct stud* t)
{
    t->num = 4000101;
    (*t).score = 92.0;
}
```

```
call funpr() new b: 4000101 92
```



```
//例4续3:  
int main()  
{
```

```
    struct stud a[2] = {{1000101, 61.0}, {1000102, 62.0}};
```

```
    /*输出结构体数组a元素的原来的成员值*/
```

```
    cout<<"old a[0]:"<< a[0].num<<' ' << a[0].score << endl;
```

```
    cout<<"old a[1]:"<< a[1].num << ' ' << a[1].score << endl;
```

```
    funar(a); //将结构体数组a作为函数的实参，然后再输出其元素的成员的值
```

```
    cout<<"new a[0]:"<< a[0].num << ' ' << a[0].score << endl;
```

```
    cout<<"new a[1]:"<< a[1].num << ' ' << a[1].score << endl;
```

```
    return 0;
```

```
}
```

```
old a[0]:1000101 61
```

```
old a[1]:1000102 62
```

```
new a[0]:3000101 81
```

```
new a[1]:3000102 82
```

```
void funar(struct stud t[])  
{  
    t[0].num = 3000101; t[0].score = 81.0;  
    t[1].num = 3000102; t[1].score = 82.0;  
}
```



# 结构体参数调用归纳（例4）

## 1. 结构体变量作为函数形参：

形参结构体变量成员值的改变**不影响**对应的实参结构体变量成员值(单向传值)

实参：结构体变量b

b.num=1000101  
b.score=61.0

单向传值

形参：结构体变量t

t.num=1000101  
t.score=61.0

重新赋值

t.num=2000101  
t.score=71.0

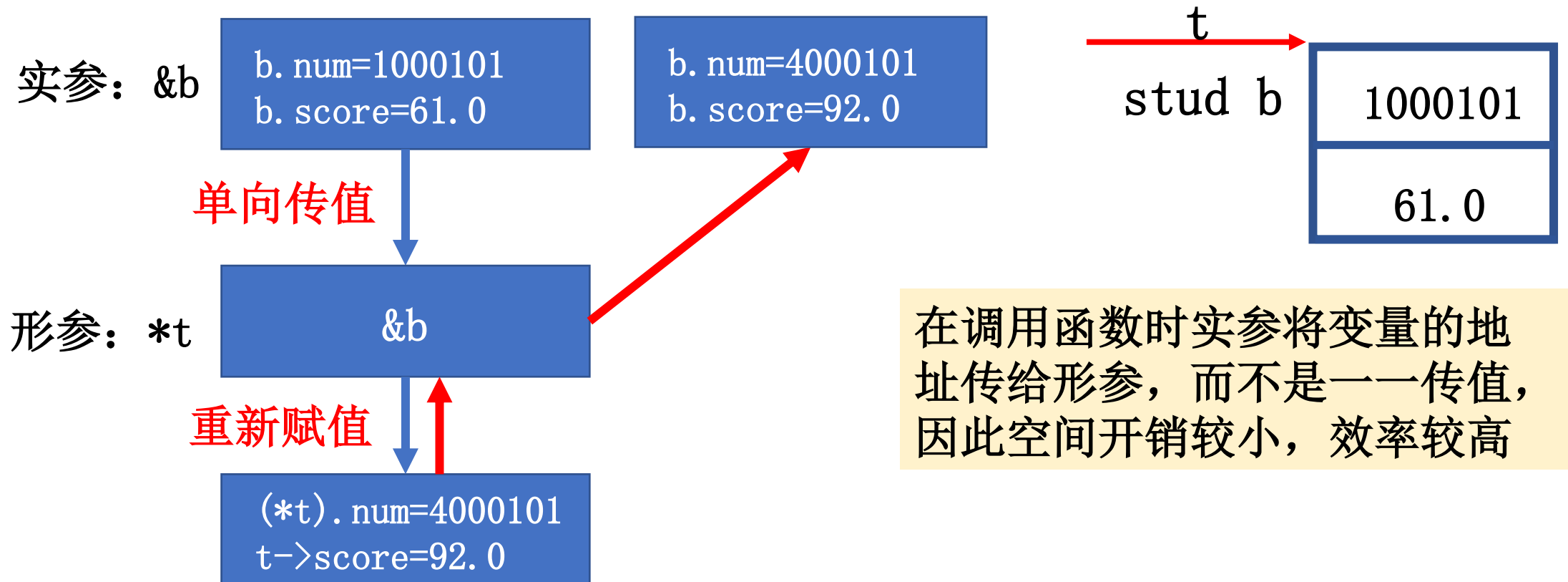
程序简单易懂，但是如果结构体变量占的内存空间很大，在实参形参传递过程中空间的开销较大，效率不高



# 结构体参数调用归纳（例4）

## 2. 结构体指针变量作为函数形参：

形参所指结构体数组元素成员值的改变会影响对应的实参结构体数组成员值(单向传值)

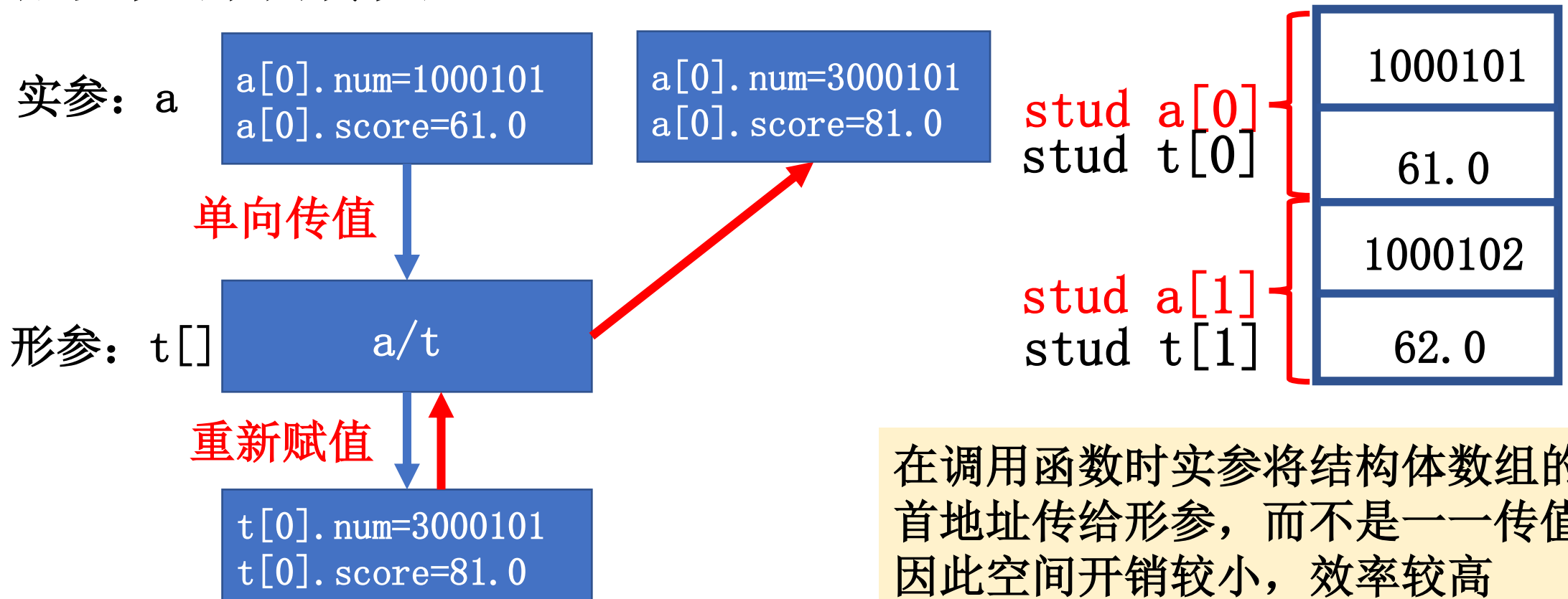




# 结构体参数调用归纳（例4）

## 3. 结构体数组作为函数形参：

形参结构体数组元素成员值的改变会影响对应的实参结构体数组成员值(单向传值)





# \*补充内容:

## 结构体类型在不同位置定义时的使用

### ✓ 函数内部:

可定义结构体类型的各种变量/成员级访问

### ✓ 函数外部:

#### ➤ 从定义点到本源程序文件的结束前:

可定义结构体类型的各种变量/成员级访问

#### ➤ 其它位置(本源程序定义点前/其它源程序):

##### ❖ 有该结构体的提前声明:

仅可定义指针及引用/整体访问


##### ❖ 有该结构体的重复定义:

可定义结构体类型的各种变量/成员级访问

## 情况一：定义在函数内部




```
void fun(void)
{ struct student {
    int num;
    char name[20];
    char sex;
    int age;
    float score;
};
  struct student s1, s2[10], *s3;
  s1.num = 10;
  s2[4].age = 15;
  s3 = &s1;
  s3->score = 75;
  s3 = s2;
  (s3+3)->age = 15;
}
```



```
int main()
{
    struct student s;
    s.age = 15;

    return 0;
}
```



## 情况二：定义在函数外部, 从定义点到本源程序结束前



```
struct student {
    int num;
    char name[20];
    char sex;
    int age;
    float score;
};
void f1(void)
{
    struct student s1, s2[10], *s3;
    s1.num = 10;
    s2[4].age = 15;
    s3 = &s1;
    s3->score = 75;
    s3 = s2;
    (s3+3)->age = 15;
}
```



```
void f2(struct student *s)
{
    s->age = 15;
}
struct student f3(void)
{
    struct student s;
    ...
    return s;
}
int main()
{
    struct student s1, s2;
    f1();
    f2(&s1);
    s2 = f3();
}
```



### 情况三：ex1.cpp和ex2.cpp构成一个程序, 无提前声明



```
/* ex1.cpp */
```

```
void f1()
```

```
{
```

✗

不可定义/使用student型各种变量

```
}
```

```
struct student {
```

```
    ...;
```

```
};
```

```
int fun()
```

```
{
```

✓

可定义student型各种变量, 访问成员

```
}
```

```
int main()
```

```
{
```

✓

可定义student型各种变量, 访问成员

```
}
```

```
/* ex2.cpp */
```

```
int f2()
```

```
{
```

✗

不可定义/使用student型  
各种变量

```
}
```





## 情况四：ex1.cpp和ex2.cpp构成一个程序, 有提前声明

```
/* ex1.cpp */
struct student; //结构体声明
void f1(struct student *s1)
{
    s1->age;
}
void f2(struct student &s2)
{
    s2.score;
}
struct student {
    ...;
};
int main()
{
    可定义student型各种变量, 访问成员
}
```

Diagram annotations for ex1.cpp:

- Box "允许" (Allowed) with arrows pointing to `s1->age;` and `struct student *s1`.
- Box "不允许" (Not Allowed) with arrows pointing to `s2.score;` and `struct student &s2`.

```
/* ex2.cpp */
struct student; //结构体声明

void f2()
{
    struct student *s1;

    struct student s3, &s2=s3;

    s1.age = 15;
}
```

Diagram annotations for ex2.cpp:

- Box "允许" (Allowed) with an arrow pointing to `struct student *s1;`.
- Box "不允许" (Not Allowed) with arrows pointing to `s1.age = 15;` and `struct student s3, &s2=s3;`.

虽可定义指针/引用, 但不能  
进行成员级访问, 无意义

## 情况四：ex1.cpp和ex2.cpp构成一个程序, 有提前声明



```
/* ex1.cpp */
struct student; //结构体声明
void f1(struct student *s1)
{
    s1->age;
}
void f2(struct student &s2)
{
    s2.score;
}
struct student {
    ...;
};
int main()
{
    可定义student型各种变量, 访问成员
}
```

Diagram annotations for ex1.cpp:

- Box "允许" (Allowed) with arrows pointing to `s1->age;` and `struct student *s1`.
- Box "不允许" (Not Allowed) with an arrow pointing to `s2.score;`.

```
/* ex2.cpp */
void f2()
{
    struct student *s1;
}
void f3()
{
    struct student; //结构体声明
    struct student *s1;
    s1->age = 15;
}
```

Diagram annotations for ex2.cpp:

- Box "不允许" (Not Allowed) with an arrow pointing to `struct student *s1;` in `f2()`.
- Box "允许" (Allowed) with an arrow pointing to `struct student *s1;` in `f3()`.
- Box "不允许" (Not Allowed) with an arrow pointing to `s1->age = 15;`.

虽可定义指针/引用, 但不能进行成员级访问, 无意义

## 情况五: ex1.cpp和ex2.cpp构成一个程序, 有重复定义



```
/* ex1.cpp */  
#include <iostream>  
using namespace std;
```

```
struct student { //结构体定义  
    ...;  
};
```

```
int fun()  
{  
    可定义/使用student型各种变量 ✓  
}
```

```
int main()  
{  
    可定义/使用student型各种变量 ✓  
}
```

```
/* ex2.cpp */  
#include <iostream>  
using namespace std;
```

```
struct student { //结构体定义  
    ...;  
};
```

```
int f2()  
{  
    可定义/使用student型  
    各种变量 ✓
```

本质上是两个不同的结构体  
struct student, 因此即使  
不完全相同也能正确, 这样  
会带来理解上的偏差



## 问题：如何在其它位置访问定义和使用结构体？

```
/* ex.h */  
struct student { //结构体定义  
    ...;  
};
```

```
/* ex1.cpp */  
#include <iostream>  
#include "ex.h" ←  
using namespace std;  
int fun()  
{  
    可定义/使用student型各种变量  
}  
int main()  
{  
    可定义/使用student型各种变量  
}
```

```
/* ex2.cpp */  
#include <iostream>  
#include "ex.h" ←  
using namespace std;
```

```
int f2()  
{  
    可定义/使用student型  
    各种变量  
}
```

解决方法：在头文件中定义



## 3.4 将引用用于结构

- ✓ 将引用用于结构的应用
- ✓ 为何要返回引用
- ✓ 返回引用时需要注意的问题
- ✓ 为何将const用于引用返回类型



```
//strc_ref.cpp -- using structure references
```

```
#include <iostream>
```

```
#include <string>
```

```
struct free_throws
```

```
{
```

```
    std::string name;
```

```
    int made;
```

```
    int attempts;
```

```
    float percent;
```

```
};
```

```
void display(const free_throws& ft);
```

```
void set_pc(free_throws& ft);
```

```
free_throws& accumulate(free_throws& target, const free_throws& source);
```

- ❖ 使用结构引用参数的方式与使用基本变量引用相同，只需在声明结构参数时使用引用运算符&即可
- ❖ 在函数中将指向该结构的引用作为参数
- ❖ 如果不希望函数修改传入的结构，可以使用const
- ❖ 还可以让函数返回指向结构的引用

//strc\_ref.cpp -- using structure references 续1

int main()

```
{ free_throws one = { "Ifelsa Branch", 13, 14 }; //partial initializations-remaining members set to 0
  free_throws two = { "Andor Knott", 10, 16 };
  free_throws three = { "Minnie Max", 7, 9 };
  free_throws four = { "Whily Looper", 5, 9 };
  free_throws five = { "Long Long", 6, 14 };
  free_throws team = { "Throwgoods", 0, 0 };
```

```
  free_throws dup; //no initialization
```

```
  set_pc(one);
```

```
  display(one);
```

```
  accumulate(team, one);
```

```
  display(team);
```

```
  // use return value as argument
```

```
  display(accumulate(team, two));
```

```
  accumulate(accumulate(team, three), four);
```

```
  display(team);
```

```
  // use return value in assignment
```

```
  dup = accumulate(team, five);
```

```
  std::cout << "Displaying team:\n";
```

```
  display(team);
```

```
  std::cout << "Displaying dup after assignment:\n";
```

```
Name: Ifelsa Branch
Made: 13    Attempts: 14    Percent: 92.8571
Name: Throwgoods
Made: 13    Attempts: 14    Percent: 92.8571
Name: Throwgoods
Made: 23    Attempts: 30    Percent: 76.6667
Name: Throwgoods
Made: 35    Attempts: 48    Percent: 72.9167
Displaying team:
Name: Throwgoods
Made: 41    Attempts: 62    Percent: 66.129
Displaying dup after assignment:
Name: Throwgoods
Made: 41    Attempts: 62    Percent: 66.129
Displaying dup after ill-advised assignment:
Name: Whily Looper
Made: 5     Attempts: 9     Percent: 55.5556
```

```
  display(dup);
```

```
  set_pc(four);
```

```
  // ill-advised assignment
```

```
  accumulate(dup, five) = four;
```

```
  std::cout << "Displaying dup
after ill-advised assignment:\n";
```

```
  display(dup);
```

```
  // std::cin.get();
```

```
  return 0;
```

```
}
```

//strc\_ref.cpp -- using structure references 续2

```
void display(const free_throws& ft)
```

```
{    using std::cout;
    cout << "Name: " << ft.name << '\n';
    cout << "    Made: " << ft.made << '\t';
    cout << "Attempts: " << ft.attempts << '\t';
    cout << "Percent: " << ft.percent << '\n';
}
```

```
void set_pc(free_throws& ft)
```

```
{
    if (ft.attempts != 0)
        ft.percent = 100.0f * float(ft.made) / float(ft.attempts);
    else
        ft.percent = 0;
}
```

```
free_throws& accumulate(free_throws& target, const free_throws& source)
```

```
{
    target.attempts += source.attempts;
    target.made += source.made;
    set_pc(target);
    return target;
}
```

```
Name: Ifelsa Branch
    Made: 13      Attempts: 14      Percent: 92.8571
Name: Throwgoods
    Made: 13      Attempts: 14      Percent: 92.8571
Name: Throwgoods
    Made: 23      Attempts: 30      Percent: 76.6667
Name: Throwgoods
    Made: 35      Attempts: 48      Percent: 72.9167
Displaying team:
Name: Throwgoods
    Made: 41      Attempts: 62      Percent: 66.129
Displaying dup after assignment:
Name: Throwgoods
    Made: 41      Attempts: 62      Percent: 66.129
Displaying dup after ill-advised assignment:
Name: Whily Looper
    Made: 5       Attempts: 9       Percent: 55.5556
```





## 3.4 将引用用于结构

### ✓ 将引用用于结构的应用

```
void display(const free_throws& ft);  
void set_pc(free_throws& ft);  
free_throws& accumulate(free_throws& target, const free_throws& source);
```

```
set_pc(one);  
display(one);  
accumulate(team, one);  
display(team)
```

```
display(accumulate(team, two));
```



## 3.4 将引用用于结构

### ✓ 为何要返回引用

```
free_throws& accumulate(free_throws& target, const free_throws& source);  
dup = accumulate(team, five);
```

- 传统返回机制与按值传递函数参数类似：计算关键字return后面的表达式，并将结果返回给调用函数。从概念上说，这个值被赋值到一个临时位置，而调用程序将使用这个值
- 如果accumulate()返回一个结构，将把整个复制到一个临时位置，再将拷贝复制给dup
- 但在返回值为引用时，将直接把team复制到dup，其效率更高

**注意：**返回引用的函数实际上是被引用的变量的别名



## 3.4 将引用用于结构

### ✓ 返回引用时需要注意的问题

#### ➤ 应避免返回函数终止时不再存在的内存单元引用

```
const free_throws& clone2(free_throws& ft) {  
    free_throws newguy; // × first step to big error  
    newguy = ft;         // copy all members  
    return newguy;       // return reference to copy  
}
```

#### ➤ 该函数返回一个指向临时变量（newguy）的引用，函数运行完毕后它将不再存在

#### ➤ **避免的方法一：** 返回一个作为参数传递给函数的引用。作为参数的引用将指向调用函数使用的数据，因此返回的引用也将指向这些数据，参考accumulate函数的做法



## 3.4 将引用用于结构

### ✓ 返回引用时需要注意的问题

➤ 应避免返回函数终止时不再存在的内存单元引用

➤ **避免的方法二：** 用new来分配新的存储空间，并返回指向该内存空间的指针

```
ft对应的存储空间已经在之前通过new申请过
const free_throws& clone(free_throws& ft) {
    free_throws * pt;
    *pt = ft;           // copy info
    return *pt;         // return reference to copy
}
free_throws & jolly = clone(three);
```


➤ 但该方法存在一个问题：在不再需要new分配的内存时，应使用delete来释放它们。调用clone()隐藏了对new的调用，使得以后很容易忘记使用delete来释放内存



## 3.4 将引用用于结构

### ✓ 为何将const用于引用返回类型

```
free_throws& accumulate(free_throws& target, const free_throws& source);  
accumulate(dup, five) = four;
```



- 上句可以通过编译，因在赋值语句中，左边必须是可修改的左值（标识一个可修改的内存块）。此函数的返回值是一个引用，可行。如果函数accumulate()按值返回，则不能通过编译
- 常规（非引用）函数返回应该是右值（不能通过地址访问的值）
- 要使用引用返回值，但又不允许执行像给accumulat()赋值这样操作，只需将返回类型声明为const引用：

```
const free_throws& accumulate(free_throws& target, const free_throws& source);  
accumulate(dup, five) = four; // ✕ not allowed for const reference return  
display(accumulate(team, two)); // ✓ void display(const free_throws& ft);
```



# 目录

- 枚举



# 4.1 枚举

✓含义：如果一个变量的取值有限且离散，那么将这个变量所有可能的取值一一列举出来，并限定在某个集合内（将变量的值一一列举）

✓声明形式：

enum 枚举类型名 {枚举（元素）常量取值表};

↑  
关键字

↑  
以标识符形式表示的整型量，  
表示枚举类型的取值集合



## 4.1 枚举

✓ 声明形式示例：

1. 定义一周内的天数

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

↑  
关键字

↑  
枚举类型名

↑  
枚举常量（元素）列表（可能取值）

2. 定义光谱的分类

```
enum spectrum {red, orange, yellow, green, blue, indigo,  
violet, ultraviolet} ;
```





## 4.1 枚举

- ✓声明枚举类型时，系统给每一个枚举元素一个指定的整数值（序号），默认从0开始，依次加1

```
enum weekday {sun, mon, tue, wed, thu, fri, sat};  
              0,    1,    2,    3,    4,    5,    6
```

- ✓可以在声明时另行指定枚举元素的序号值（部分指定）

```
enum weekday {sun=7, mon=1, tue, wed, thu, fri, sat};  
              7,      1,    2,    3,    4,    5,    6
```

```
enum weekday {sun, mon=4, tue, wed, thu, fri, sat};  
              0,      4,    5,    6,    7,    8,    9
```



## 4.1 枚举

- ✓ 如果定义的常量重复，编译器不报错，但后续使用中会有问题

```
enum weekday {sun=3, mon=1, tue, wed, thu, fri, sat};  
              3,      1,      2,      3,      4,      5,      6
```

- ✓ 声明枚举类型时系统已初始化，声明后不允许再对枚举元素赋值

```
int sun; 错误
```

```
error C2365: “main::sun”: 重定义; 以前的定义是“枚举数”
```

```
sun=10; 错误
```

```
error C2440: “=”: 无法从“int”转换为“main::weekday”
```



# 4.1 枚举

✓ 枚举变量的定义形式:

1. 声明枚举类型的同时定义变量

```
enum 枚举类型名 {枚举（元素）常量取值表} 变量;
```

2. 先声明枚举类型，再用已声明的枚举类型定义变量

```
enum 枚举类型名 {枚举（元素）常量取值表};
```

```
enum 枚举类型名 变量; //该变量的值只能在枚举常量中取值
```



## 4.1 枚举

✓ 枚举变量的定义示例:

➤ 例: 定义一周内的天数

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

```
enum weekday w1, w2;
```

定义枚举类型变量w1和w2, 且枚举变量的值只能是sun到sat中间之一



## 4.1 枚举

✓ 枚举变量只能参与赋值、比较和输入输出操作，运算时用其本身的整数值

0      1      2      3      4      5      6

```
enum weekday {sun, mon, tue, wed, thu, fri, sat} w1, w2;
```

(1) 赋值:

```
w1 = mon; //将枚举常量值1赋给枚举变量
```

```
w2 = fri; //将枚举常量值5赋给枚举变量
```

```
w2 = w1; //将枚举变量w1的值1赋值给同类型枚举变量w2
```



## 4.1 枚举

✓ 枚举变量只能参与赋值、比较和输入输出操作，运算时用其本身的整数值

0      1      2      3      4      5      6

```
enum weekday {sun, mon, tue, wed, thu, fri, sat} w1, w2;
```

(2) 比较:

```
w1 == mon    //将枚举变量w1的值与枚举常量mon比较，返回true/false  
w2 > wed     //将枚举变量w2的值与枚举常量sat比较，返回true/false  
w2 >= w1     //比较枚举变量w1和w2的值，返回true/false（按各自的  
              值比较）
```



## 4.1 枚举

✓ 枚举变量只能参与赋值、比较和输入输出操作，运算时用其本身的整数值

0      1      2      3      4      5      6

```
enum weekday {sun, mon, tue, wed, thu, fri, sat} w1, w2;
```

(3) 输出:

➤ 直接输出: (按整型输出)

```
w1 = wed;
```

```
cout << w1 << endl; //3
```

```
printf("w1=%d\n", w1); //w1=3
```



## 4.1 枚举

✓ 枚举变量只能参与赋值、比较和输入输出操作，运算时用其本身的整数值

```
enum weekday {0    1    2    3    4    5    6  
sun, mon, tue, wed, thu, fri, sat} w1, w2;
```

(3) 输出:

➤ 间接输出:

```
switch(w1) //括号内表达式为枚举类型  
{  
    case wed: //wed是整型常量3，不加双引号  
        cout << "wed"; //判断变量w1的值，输出对应字符串  
        break;  
    ... //其它case  
}
```





## 4.1 枚举

✓ 枚举变量只能参与赋值、比较和输入输出操作，运算时用其本身的整数值

0      1      2      3      4      5      6

```
enum weekday {sun, mon, tue, wed, thu, fri, sat} w1, w2;
```

(4) 输入:

➤直接输入:

```
weekday w1;
```

```
scanf("%d", &w1); //人为保证输入0-6之间
```

```
cin >> w1; //不允许(后续学运算符重载后才允许)
```



## 4.1 枚举

✓ 枚举变量只能参与赋值、比较和输入输出操作，运算时用其本身的整数值

```
enum weekday {0    1    2    3    4    5    6  
sun, mon, tue, wed, thu, fri, sat} w1, w2;
```

(4) 输入:

➤ 间接输入:

```
char s[80];  
cin >> s; // 键盘输入 sun  
if (!strcmp(s, "sun"))  
    w1=sun;  
else if (...)
```

```
int w;  
cin >> w;  
w1=(weekday)w; // 强制类型转换
```



//例1: 打印12个月的天数

```
#include <iostream>
using namespace std;
enum month { Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
int main()
{
    int i, m[13], *p;
    for (i=Jan; i<=Dec; i++)
        if (i==Feb)
            m[i] = 28;
        else if (i==Apr || i==Jun || i==Sep || i==Nov)
            m[i] = 30;
        else
            m[i] = 31;
    for (p=&m[1]; p<m+13; p++) //指针法访问m数组, [0]舍弃不用
        cout << *p << " ";
    return 0;
}
```

Microsoft Visual Studio 调试控制台

31 28 31 30 31 30 31 31 30 31 30 31



Sep, Oct, Nov, Dec };

```
//例
#include <iostream>
using namespace std;
enum month {
    Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};

int main() {
    for (i=1; i<=12; i++)
        if (i==2)
            m[i] = 28;
        else if (i==4 || i==6 || i==9 || i==11)
            m[i] = 30;
        else
            m[i] = 31;

    if (i==Feb)
        m[i] = 28;
    else if (i==Apr || i==Jun || i==Sep || i==Nov)
        m[i] = 30;
    else
        m[i] = 31;

    for (p=&m[1]; p<m+13; p++) //指针法访问m数组, [0]舍弃不用
        cout << *p << " ";
    return 0;
}
```

对比可读性  
自行选择

Microsoft Visual Studio 调试控制台

31 28 31 30 31 30 31 31 30 31 30 31



# 目录

- 用typedef声明新类型



## 5.1 用typedef声明类型

✓定义：为一种数据类型声明一个新名字。这里的数据类型包括基本数据类型（int，char等）和自定义的数据类型（struct等），使得代码可读性更高，意义更明确

✓形式：

typedef 原有类型名 **新类型名（大写）**

**通常**将typedef声明的类型名用大写字母表示，以便于系统提供的标准类型标识符区别



## 5.1 用typedef声明类型

形式:

typedef 原有类型名 新类型名 (大写)

例如:

typedef int COUNTER; //用指定标识符COUNTER代表int类型

COUNTER i, j;  int i, j; //相当于定义i, j为COUNTER型, 即整型

//例1:

```
struct Point
{
    float x;
    float y;
}; // 声明结构体类型

typedef struct Point POINT;
POINT pt; //新类型名定义结构体变量
pt.x = 10.0f;
pt.y = 20.0f;
```

```
//在struct前加了typedef, 表明是声明新类型名
typedef struct Point
{
    float x;
    float y;
} POINT; //POINT是新类型名, 不是变量名
POINT pt; // 用新类型名定义结构体变量
pt.x = 10.0f;
pt.y = 20.0f;
```

- 上述两种方式都是为struct Point结构体类型声明了一个新名字POINT
- 所以POINT pt 相当于 struct Point pt //pt为结构体类型变量
- POINT \*p 相当于 struct Point \*p //p为指向此结构体类型的指针





## 5.1 用typedef声明类型

例如:

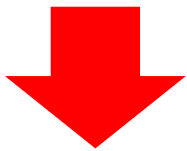
`int a[10], b[10], c[10], d[10];` //四个一维数组, 大小相同

`typedef int ARR[10];` //将大小为10的一维数组声明为新类型ARR

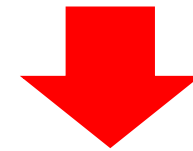
`ARR a, b, c, d;` //用ARR定义数组变量

1. `ARR a;`  $\Leftrightarrow$  `int a[10];`

2. `ARR b[5]`  $\Leftrightarrow$  `int b[5][10];`



相当于a, b, c, d都被定义为一维整型数组, 含有10个元素



相当于数组b中每一个元素被定义为一个含有10个元素的一维整型数组, 数组b即二维数组



# 5.1 用typedef声明类型

声明类型的方法步骤:

1. 先按定义变量的方法写出定义语句 (`int i;`)
2. 将变量名换成新类型名 (`int INTEGER;`)
3. 在最前面加typedef (`typedef int INTEGER;`)
4. 用新类型名去定义变量 (`INTEGER i, j;`)



有了上面的typedef声明, `INTEGER i, j;` 相当于定义了两个  
INTEGER类型变量*i*, *j*, 即整型



## 5.1 用typedef声明类型

声明类型的方法步骤（再例）：

1. 先按定义变量的方法写出定义语句 (`int a[10];`)
2. 将变量名换成新类型名 (`int ARR[10];`)
3. 在最前面加typedef (`typedef int ARR[10];`)
4. 用新类型名去定义变量 (`ARR a, b[5];`)



有了上面的typedef声明：

1. 相当于a被定义为一维整型数组，含有10个元素
2. 相当于b被定义为一个二维整型数组



# 5.1 用typedef声明类型

注意：

1. typedef 没有创造新类型，只是将原有数据类型声明新名字（“大名和昵称”）
2. typedef是声明类型，不是定义变量
3. typedef和#define的差异

#define是宏定义（用一个标识符来表示一个常量），发生在预处理阶段，也就是编译之前，它只进行简单机械的字符串替换，不进行任何检查



# 5.1 用typedef声明类型

✓ typedef和#define的差异

例如:

```
typedef (char*) PIONT; //声明PIONT为字符型指针类型
```

```
PIONT a, b的效果同char* a; char* b; //表示定义了两个字符型指针变量
```

```
#define PIONT char*
```

```
//预处理指令，仅将char*替换为标识符PIONT，替换完毕再编译
```

```
PIONT a, b的效果同char* a, b;
```

```
//表示定义了一个字符型指针变量a和字符型变量b
```



Microsoft Visual Studio

4441

//例2:

```
#include <iostream>
```

```
using namespace std;
```

```
typedef (char*) pint; //注意: 两者有行末分号的區別
```

```
#define PINT char *
```

```
pint s1, s2;
```

```
PINT s3, s4;
```

```
int main()
```

```
{
```

```
    cout << sizeof(s1) << sizeof(s2) << sizeof(s3) << sizeof(s4);
```

```
    return 0;
```

```
}
```

字符型指针变量

字符型变量



# 总结

- 结构体类型
- 结构体变量的定义与初始化
- 结构体变量的使用
- 枚举
- 用typedef声明新类型