



## § 7. 结构体、枚举及typedef声明新类型

### 7.1. 用户自定义类型的引入

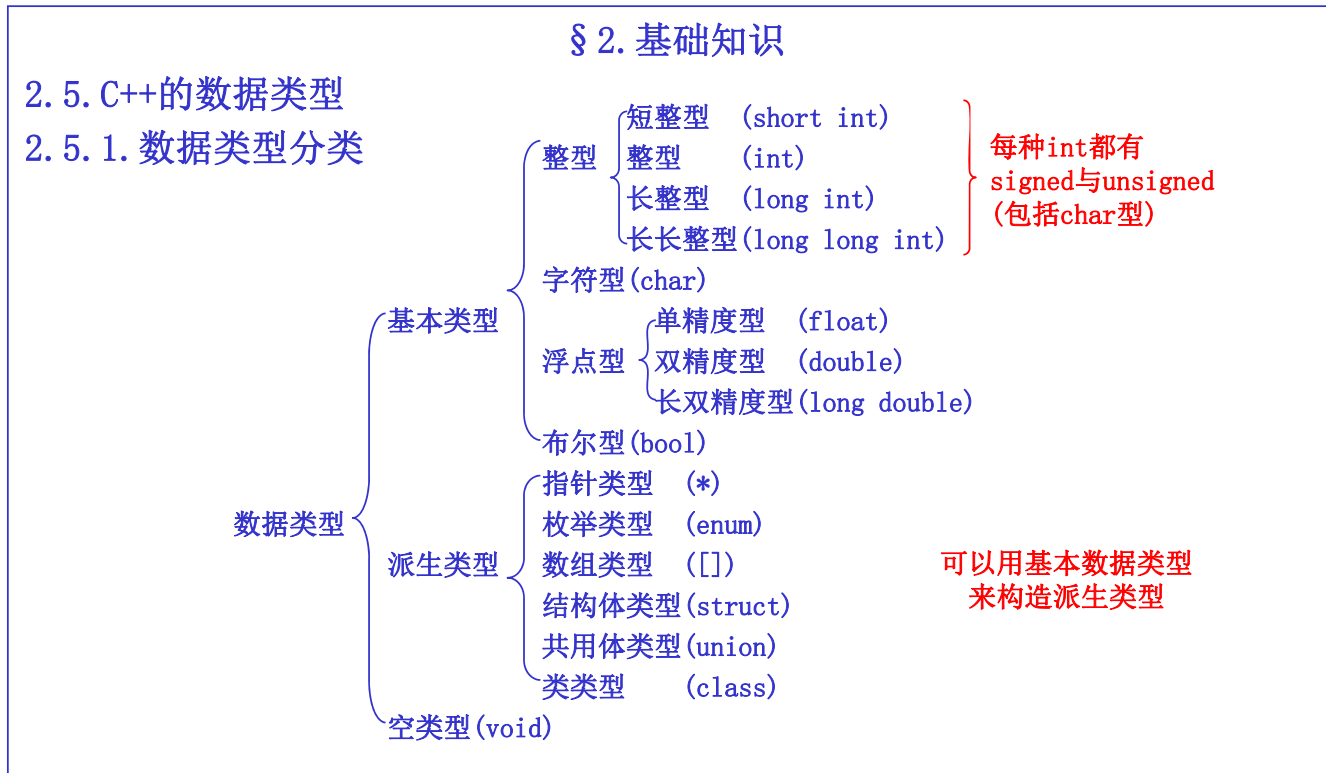
#### 7.1.1. 用户自定义类型(派生类型)的含义

用**基本数据类型**以及**已存在的自定义数据类型**组合而成的新数据类型

#### 7.1.2. 自定义数据类型的分类

元素同类型的自定义数据类型：**数组**

元素不同类型的自定义数据类型：**结构体、共用体、类**





## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.1. 引入

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（简称结构体）

★ 将描述一个事物的各方面特征的数据组合成一个有机的整体，说明数据之间的内在关系

例1：键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出

```
int main()
{
    int num, age;
    char sex, name[20], addr[30];
    float score;
    cin >> num ... ;
    ...
    cout << sex ... ;
    return 0;
}
```

1个学生的6方面信息：  
用6个彼此完全独立的  
不同类型的变量来表达

缺点：访问时无整体性

例2：键盘输入100个学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出

```
const int N=100;
int main()
{
    int num[N], age[N], i;
    char sex[N], name[N][20], addr[N][30];
    float score[N];
    for(i=0; i<N; i++) {
        cin >> num[i] ... ;
        ...
        cout << sex[i] ... ;
    }
}
```

100个学生的6方面信息：  
用6个彼此完全独立的不同类型  
的数组变量来表达  
缺点：1. 访问时无整体性  
2. 访问同一个人时，不同数组  
的下标必须对应

说明：

这3个例子都是  
非结构体方式，主要  
看缺点

例3：键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出，要求以指针方式操作

```
int main()
{
    int num, age, *p_num=&num, *p_age=&age;
    char sex, name[20], addr[30];
    char *p_sex=&sex, *p_name=name, *p_addr=addr;
    float score, *p_score=&score;
    cin >> *p_num ... ;
    ...
    cout << *p_sex ... ;
    return 0;
}
```

学号	姓名	性别	年龄	课程成绩	家庭住址
1001	张三	男	18	80.5	上海市杨浦区***
1002	李四	女	18	76	黑龙江省齐齐哈尔市***
1003	王五	女	19	90.5	四川省宜宾市***
1004	赵六	男	17	88	陕西省汉中市***
...	...	...	...	...	...



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.2. 结构体类型的声明

##### 7.2.2.1. 结构体类型声明的形式

```
struct 结构体名 {  
    结构体成员1 (类型名 成员名)  
    ...  
    结构体成员n (类型名 成员名)  
}; (带分号)
```

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

★ 结构体成员也称为结构体的数据成员

★ 结构体名, 成员名命名规则同变量

★ 同一结构体的成员名不能同名, 但可与其它名称 (其它结构体的成员名, 其它变量名等) 相同

```
struct x1 {      struct x2 {      struct x3 {      int main()      int fun()      void num()  
    int num;      int num;      float num; {      {      {  
    ...           ...           ...           long num;      int num[5];      ...  
};               };               }           }           }
```

★ 每个成员的类型可以相同, 也可以不同



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.2. 结构体类型的声明

##### 7.2.2.1. 结构体类型声明的形式

★ 每个成员的类型既可以是基本数据类型，也可以是**已存在**的自定义数据类型

```
struct date {  
    int year;  
    int month;  
    int day;  
};  
  
struct student {  
    int num;  
    char name[20];  
    char sex;  
    struct date birthday;  
    float score;  
    char addr[30];  
};
```

struct date必须在struct student  
的前面定义, 否则无法知道birthday  
占多少字节

```
struct student {  
    int num;  
    char name[20];  
    char sex;  
    struct student monitor;  
    float score;  
    char addr[30];  
};
```

★ 每个成员的类型不允许是自身的结构体类型  
原因: 无法判断 monitor 占多少个字节

★ 每个成员的类型不允许是自身的结构体类型

★ 结构体类型的定义既可以放在函数外部, 也可以放在函数内部(具体见后)

★ 结构体类型的大小为所有成员的大小的总和, 可用sizeof(struct 结构体名) 计算, 但不占用具体的内存空间  
(结构体类型不占空间, 结构体变量占用一段连续的空间)

★ C的结构体只能包含数据成员, C++还可以包含函数(后续模块)

```
int i; sizeof(int)得4  
但int型不占空间, i占4字节
```



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.2. 结构体类型的声明

##### 7.2.2.2. 结构体类型声明与字节对齐

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**

例：1、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

2、64位编译系统下，指针变量的起始地址是8的倍数

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**



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.2. 结构体类型的声明

##### 7.2.2.2. 结构体类型声明与字节对齐

内存对齐的基本概念：

结构体的成员对齐：

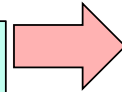
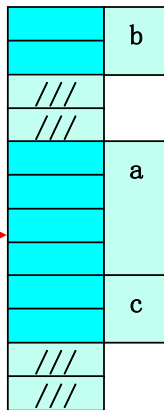
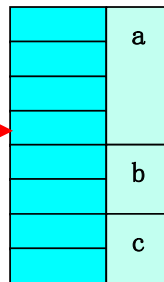
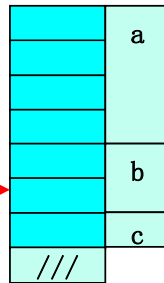
//例1: 结构体声明与字节对齐

```
#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;
};
int main()
{
    cout << sizeof(s1) << endl;
    cout << sizeof(s2) << endl;
    cout << sizeof(s3) << endl;
}
```

理论：7字节  
实际：8字节

理论：8字节  
实际：12字节

理论：8字节  
实际：8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

- 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1/2/4的倍数

Microsoft Visual Studio 调试控制台

```
8
12
8
```



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.2. 结构体类型的声明

##### 7.2.2.2. 结构体类型声明与字节对齐

内存对齐的基本概念：

结构体的成员对齐：

//例2：结构体声明与字节对齐

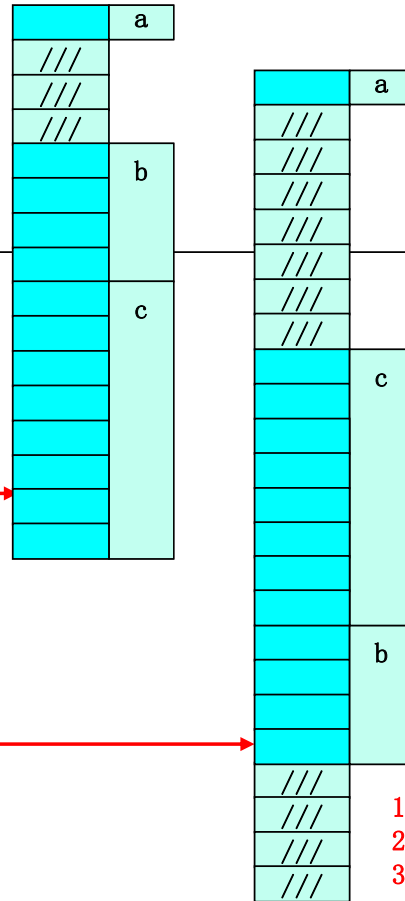
```
#include <iostream>
using namespace std;
struct s1 {
    char  a;
    int   b;
    double c;
};
struct s2 {
    char  a;
    double c;
    int   b;
};
int main()
{
    cout << sizeof(s1) << endl;
    cout << sizeof(s2) << endl;
}
```

理论：13字节  
实际：16字节

理论：13字节  
实际：24字节

Microsoft Visual Studio 调试控制台

```
16
24
```



- 1、起始地址必须是8的倍数
- 2、总大小必须是8的倍数
- 3、每个成员的起始地址必须是1/4/8的倍数



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.2. 结构体类型的声明

##### 7.2.2.2. 结构体类型声明与字节对齐

内存对齐的基本概念：

结构体的成员对齐：

//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];
```

```
};
```

```
int main()
```

```
{
```

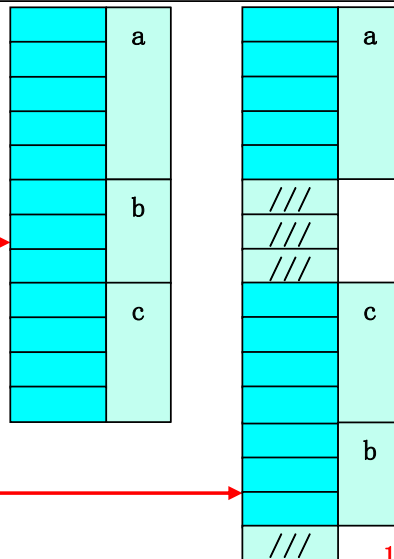
```
    cout << sizeof(s1) << endl;
```

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

```
}
```

理论：12字节  
实际：12字节

理论：12字节  
实际：16字节



推论：

1、s2中，a数组越界3字节/b数组越界1字节，不会导致系统错误

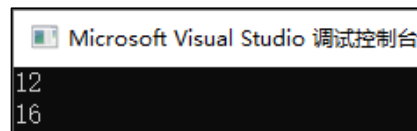
2、s2中，a数组越界4 ~ 11字节，会影响c/b的取值，但不会导致系统错误

=> 越界错误的后果不可预料  
(包括不体现出错误)

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1/4的倍数







## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.3. 结构体变量的定义及初始化

##### 7.2.3.1. 先定义结构体类型，再定义变量

```
struct student {  
    ...  
};  
struct student s1;  
struct student s2[10];  
struct student *s3;
```

★ 关键字struct(阴影部分)在C中不能省，在C++中可省略

★ 结构体变量占用实际的内存空间，根据变量的不同类型(静态/动态/全局/局部)在不同区域进行分配，遵守各自的初始化规则

##### 7.2.3.2. 在定义结构体类型的同时定义变量

```
struct student {  
    ...  
} s1, s2[10], *s3;  
struct student s4;
```

★ 可以再次用7.2.3.1的方法定义新的变量



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.3. 结构体变量的定义及初始化

##### 7.2.3.3. 直接定义结构体类型的变量(结构体无名)

```
struct {  
    ...  
} s1, s2[10], *s3;
```

- ★ 因为结构体无名，因此无法再用7.2.3.1的方法进行新的变量定义  
(适用于仅需要一次性定义的地方)

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

#### 7.2.3.4. 结构体变量定义时初始化

```
student s1={1, "张三", 'M', 20, 78.5, "上海"};
```

- ★ 按各成员依次列出

- ★ 若嵌套使用，要列出最低级成员

```
student s1={1, "张三", 'M', {1982, 5, 9}, 78.5};
```

内{}可省  
但不建议

```
struct date {  
    int year;  
    int month;  
    int day;  
};
```

```
struct student {  
    int num;  
    char name[20];  
    char sex;  
    struct date birthday;  
    float score;  
};
```

- ★ 可用一个同类型变量初始化另一个变量

```
student s1={1, "张三", 'M', {1982, 5, 9}, 78.5};  
student s2=s1;
```

内{}可省  
但不建议



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.4. 结构体变量的使用

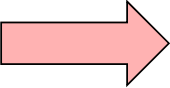
##### 7.2.4.1. 形式

变量名. 成员名

★ . 称为成员运算符（附录D 优先级第2组，左结合）

★ C中最高，C++中次高

```
struct student {  
    int    num;  
    char   name[20];  
    char   sex;  
    int    age;  
    float  score;  
    char   addr[30];  
} s1;  
  
s1.num = 1;  
strcpy(s1.name, "张三");  
s1.sex = 'M';  
s1.age = 20;  
s1.score = 76.5;  
strcpy(s1.addr, "上海");
```





## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.4. 结构体变量的使用

##### 7.2.4.1. 形式

##### 7.2.4.2. 使用

★ 结构体变量允许进行整体赋值操作

student s1={...}, s2;      用一个同类型变量初始化另一个变量:  
s2=s1;    //赋值语句      student s1={...}, s2=s1;    //定义时初始化

★ 在所有基本类型变量出现的地方，均可以使用该基本类型的结构体变量的成员

int i, *p;	student s1; 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



## § 7. 结构体、枚举及typedef声明新类型

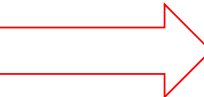
### 7.2. 结构体类型

#### 7.2.4. 结构体变量的使用

##### 7.2.4.2. 使用

- ★ 结构体变量允许进行整体赋值操作
- ★ 在所有基本类型变量出现的地方，均可以使用该基本类型的结构体变量的成员
- ★ 若嵌套使用，只能对最低级成员操作

```
struct date {  
    int year;  
    int month;  
    int day;  
};  
struct student {  
    int num;  
    char name[9];  
    char sex;  
    struct date birthday ;  
    float score;  
};
```



```
s1.birthday.year=1980;  
cin >> s1.birthday.month;  
cout << s1.birthday.day;
```

- ★ 结构体变量不能进行整体的输入和输出操作

```
student s1={...};  
cin >> s1;    ✗  
cout << s1;   ✗
```



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.4. 结构体变量的使用

##### 7.2.4.2. 使用

例1：键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出 (前面例子的对比)

```
int main()
{
    int num;
    int age;
    char sex;
    char name[20];
    char addr[30];
    float score;

    cin >> num ... ;
    ...
    cout << sex ... ;
    return 0;
}
```

6个独立变量



```
struct student {
    ...;
};

int main()
{
    struct student s1;

    cin >> s1.num ... ;
    ...
    cout << s1.sex ... ;
    return 0;
}
```

1个变量的  
6个成员



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.5. 结构体变量数组

##### 7.2.5.1. 含义

一个数组，数组中的元素是结构体类型

##### 7.2.5.2. 定义

`struct 结构体名 数组名[正整型常量表达式]`

`struct 结构体名 数组名[正整型常量表达式1][正整型常量表达式2]`

★ 包括整型常量、整型符号常量和整型只读变量

```
struct student s2[10];
```

```
struct student s4[10][20];
```

内{}可省  
但不建议

##### 7.2.5.3. 定义时初始化

```
struct student s2[10] = { {1, "张三", 'M', 20, 78.5, "上海"},  
                          {2, "李四", 'F', 19, 82, "北京"},  
                          {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...} };
```

★ 其它同基本数据类型数组的初始化

(占用空间、存放、下标范围、初始化时省略大小)



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.5. 结构体变量数组

##### 7.2.5.4. 使用

数组名[下标].成员名

```
s2[0].num=1;
```

```
cin >> s2[0].age >> s2[0].name;
```

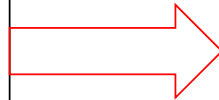
```
cout << s2[1].age << s2[1].name;
```

```
s2[2].name[0] = 'A'; //注意两个[]的位置
```

例2: 键盘输入100个学生的学号、姓名、性别、年龄、成绩和家庭住址, 再依次输出 (前面例子对比)

```
const int N=100;
int main()
{
    int num[N], age[N], i;
    char sex[N];
    char name[N][20];
    char addr[N][30];
    float score[N];
    for(i=0; i<N; i++) {
        cin >> num[i] ... ;
        ...
        cout << sex[i] ... ;
    }
}
```

6个独立的  
大小为100  
的数组变量



```
const int N=100;
struct student {
    ...;
};
int main()
{
    int i;
    struct student s2[N];
    for(i=0; i<N; i++) {
        cin >> s2[i].num ... ;
        ...
        cout << s2[i].sex ... ;
    }
    return 0;
}
```

1个大小为100  
的数组, 每个  
元素有6个成员





## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.5. 结构体变量数组

##### 7.2.5.4. 使用

例：现有Li/Zhang/Sun三个候选人，键盘输入10个候选人的名字，统计每个人的得票

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

struct Person {
    char name[20];
    int count;
};

int main()
{
    struct Person leader[3]={"Li", 0, "Zhang", 0, "Sun", 0};
    int i, j;
    char leader_name[20];
    for(i=0; i<10; i++) {
        cin >> leader_name; //一维数组不带下标，表示串方式输入(≤19)
        for(j=0; j<3; j++)
            if (!strcmp(leader_name, leader[j].name)) //严格大小写
                leader[j].count++;
    } //end of for(i)
    cout << endl;
    for(i=0; i<3; i++)
        cout << leader[i].name << ":" << leader[i].count << endl;
    return 0;
}
```

可改进的地方：

1、3/10/20应该用宏定义或常量变量

2、下面的语句可优化效率

```
if(!strcmp(leader_name, leader[j].name)){
    leader[j].count++;
    break;
}
```

运行效率高，避免比较成功后再做不必要的比较

Microsoft

```
Zhang
Li
Li
Sun
Zhang
Sun
Sun
Li
Sun
Sun
Li:3
Zhang:2
Sun:5
```

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

struct Person {
    string name; //变化，用string替代一维字符数组
    int count;
};

int main()
{
    Person leader[3]={"Li", 0, "Zhang", 0, "Sun", 0};
    int i, j;
    string leader_name;
    for(i=0; i<10; i++) {
        cin >> leader_name; //可输入任意长度字符串
        for(j=0; j<3; j++)
            if (leader_name == leader[j].name) //直接用==进行比较
                leader[j].count++;
    } //end of for(i)
    cout << endl;
    for(i=0; i<3; i++)
        cout << leader[i].name << ":" << leader[i].count << endl;
    return 0;
}
```

换string后：

长度不受限、比较运算较简单，但不影响整个程序的处理逻辑



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.6. 指向结构体变量的指针

含义：存放结构体变量的地址

##### 7.2.6.1. 结构体变量的地址与结构体变量中成员地址

student s1;

&s1 : 结构体变量的地址

(基类型是结构体变量, +1表示一个结构体长度)

&s1.age : 结构体变量中某个成员地址

(基类型是该成员的类型, +1表示一个成员长度)

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

&s1.age => 2028  
(&s1.age)+1 => 2032

有填充(实际)

&s1 => 2000  
&s1+1 => 2068

s1	2000	num
	2003	
	2004	name
	...	
	2023	sex
	2024	
	2025	age
	2027	
	2028	score
	2031	
	2032	addr
	2035	
	2036	...
	...	
	2065	addr
	2066	
	2067	...
	...	

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

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

```
int main()  
{
```

```
    student s1;
```

```
    cout << sizeof(s1)      << endl;  
    cout << &s1             << endl;  
    cout << &s1.num          << endl;  
    cout << (void *)&s1.name << endl;  
    cout << (void *)&s1.sex << endl;  
    cout << &s1.age          << endl;  
    cout << &s1.score        << endl;  
    cout << (void *)&s1.addr << endl;
```

```
    return 0;  
}
```

在多编译器下运行本程序, 观察:

- 1、s1的大小是否是4的倍数
- 2、s1的起始地址是否4的倍数
- 3、s1中每个数据成员地址是否其自身类型的整数倍

Microsoft

```
68  
0053FD24  
0053FD24  
0053FD28  
0053FD3C  
0053FD40  
0053FD44  
0053FD48
```



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.6. 指向结构体变量的指针

含义：存放结构体变量的地址

##### 7.2.6.1. 结构体变量的地址与结构体变量中成员地址

student s1;

&s1 : 结构体变量的地址

(基类型是结构体变量, +1表示一个结构体长度)

&s1.age : 结构体变量中某个成员地址

(基类型是该成员的类型, +1表示一个成员长度)

##### 7.2.6.2. 结构体指针变量的定义

struct 结构体名 \*指针变量名

struct student s1, \*s3;

int \*p;

s3=&s1; 结构体变量的指针

s3的值为2000, ++s3后值为2068

p=&s1.age; 结构体变量成员的指针

p的值为2028, ++p后值为2032

(注:不要说指向score, 应该说不再指向age)

s1	2000	num
	2003	
	2004	name
	...	
	2023	
	2024	sex
	2025	/// (3)
	2027	
	2028	age
	2031	
	2032	score
	2035	
	2036	addr
	...	
	2065	
	2066	/// (2)
	2067	

```
#include <iostream>
#include <iomanip>
using namespace std;
```

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

```
int main()
{
```

```
    struct student s1;
    cout << &s1 << endl;
    cout << &s1+1 << endl;
```

地址X  
地址X + 68

```
    cout << &s1.num << endl;
    cout << &s1.num+1 << endl;
```

地址X  
地址X + 4

```
    cout << (void *)(&s1.sex) << endl;
    cout << (void *)(&s1.sex+1) << endl;
```

地址Y (X+24)  
地址Y + 1

```
    cout << &s1.age << endl;
    cout << &s1.age+1 << endl;
    return 0;
}
```

地址Z (Y+1+3)  
地址Z + 4

Microsoft
010FFC50
010FFC94
010FFC50
010FFC54
010FFC68
010FFC69
010FFC6C
010FFC70



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.6. 指向结构体变量的指针

##### 7.2.6.1. 结构体变量的地址与结构体变量中成员地址

##### 7.2.6.2. 结构体指针变量的定义

##### 7.2.6.3. 使用

(\*指针变量名). 成员名

指针变量名->成员名 ⇔ (\*指针变量名). 成员名

★ -> 称为间接成员运算符（附录D 优先级第2组，左结合）

```
struct student s1, *s3=&s1;
cout << s1.num    << s1.name    << s1.sex;
cout << (*s3).num << (*s3).name << (*s3).sex;
cout << s3->num   << s3->name    << s3->sex;
```

s3->age++; 值后缀++

++s3->age; 值前缀++



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

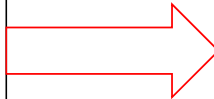
#### 7.2.6. 指向结构体变量的指针

##### 7.2.6.3. 使用

例3：键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出，要求以指针方式操作 (前例)

```
int main()
{
    int num, age;
    char sex, name[20], addr[30];
    float score;
    int *p_num=&num;
    int *p_age=&age;
    char *p_sex=&sex;
    char *p_name=name;
    char *p_addr=addr;
    float *p_score=&score;
    cin >> *p_num ... ;
    ...
    cout << *p_sex ... ;
    return 0;
}
```

6个值变量  
6个指针变量  
分别指向



```
struct student {
    ...;
};

int main()
{
    struct student s1;
    struct student *s3;
    s3 = &s1;
    cin >> s3->num ... ;
    ...
    cout << s3->sex ... ;
    return 0;
}
```

1个值变量  
1个指针变量  
指向6个成员



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.6. 指向结构体变量的指针

##### 7.2.6.4. 指向结构体数组的指针

```
struct student s2[10], *p;
```

```
p = s2; ✓
```

```
p = &s2[0]; ✓
```

```
p = &s2[0].num; ✗ 指针的基类型不匹配
```

```
p = (struct student *)&s2[0].num; ✓ 强制类型转换
```

各种表示形式:

---

`(*p).num` : 取p所指元素中成员num的**值**

`p->num` : ...

`p[0].num` : ...

---

`p+1` : 取p指元素的下一个元素的**地址**

---

`(*p+1).num`: 取p指向的元素的下一个元素的num**值**

`(p+1)->num` : ...

`p[1].num` : ...

---

`(p++)->num` : 保留p的旧值到临时变量中, p++后指向下一元素, 再取p旧值所指元素的成员num的值

---

`(++p)->num` : p先指向下一个元素, 再取p所指元素的成员num的值

---

`p->num++` : 取p所指元素中成员num的值, 值++

---



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.7. 结构体数据类型作为函数参数

##### 7.2.7.1. 形参为结构体简单变量

★ 对应实参为结构体简单变量/数组元素

```
void fun(struct student s)
{
    ...;
}
int main()
{
    struct student s1, s2[10];
    struct student s3[3][4];
    ...
    fun(s1);
    fun(s2[4]);
    fun(s3[1][2]);
    return 0;
}
```



```
void fun(int s)
{
    ...;
}
int main()
{
    int s1, s2[10];
    int s3[3][4];
    ...
    fun(s1);
    fun(s2[4]);
    fun(s3[1][2]);
    return 0;
}
```

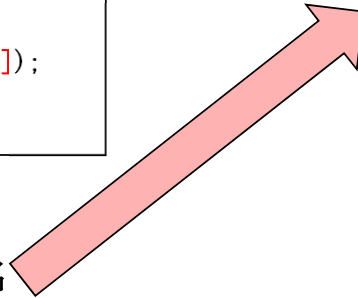
```
void fun(struct student *s)
{
    struct student s[]
    ...;
}
int main()
{
    struct student s1, s2[10];
    ...
    fun(&s1);
    ...
    fun(s2);
    return 0;
}
```



```
void fun(int *s)
{
    int s[]
    ...;
}
int main()
{
    int s1, s2[10];
    ...
    fun(&s1);
    ...
    fun(s2);
    return 0;
}
```

##### 7.2.7.2. 形参为结构体变量的指针

★ 对应实参为结构体简单变量的地址/一维数组名



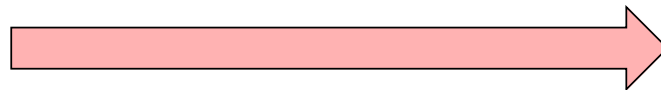
```
void fun(struct student &s)
{
    ...;
}
int main()
{
    struct student s1;
    ...
    fun(s1);
    ...
    return 0;
}
```



```
void fun(int &s)
{
    ...;
}
int main()
{
    int s1;
    ...
    fun(s1);
    ...
    return 0;
}
```

##### 7.2.7.3. 形参为结构体的引用声明

★ 对应实参为结构体简单变量



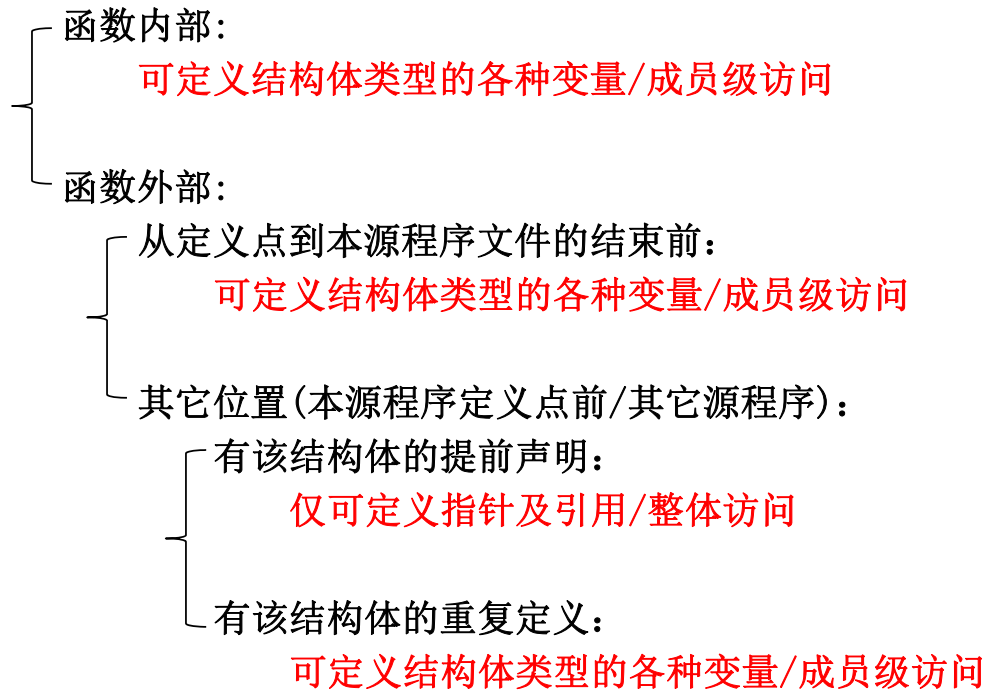


## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.8. 结构体类型在不同位置定义时的使用 (续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)



类似外部全局变量概念,但不完全相同





## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.8. 结构体类型在不同位置定义时的使用 (续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部 (具体见后)

情况一: 定义在函数内部

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

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;
}
```

正确

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

    return 0;
}
```

不正确



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.8. 结构体类型在不同位置定义时的使用 (续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部 (具体见后)

情况二: 定义在函数外部,从定义点到本源程序结束前

```
#include <iostream>
using namespace std;
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;
}
```

都正确

```
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();
    return 0;
}
```



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.8. 结构体类型在不同位置定义时的使用 (续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

情况三: ex1.cpp和ex2.cpp构成一个程序, 无提前声明

<pre>/* ex1.cpp */ #include &lt;iostream&gt; using namespace std;  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    ✓ }</pre>	<pre>/* ex2.cpp */ #include &lt;iostream&gt; using namespace std;  int f2() {     不可定义/使用student型各种变量    × }</pre>
---	--



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.8. 结构体类型在不同位置定义时的使用 (续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

情况四: ex1.cpp和ex2.cpp构成一个程序, 有提前声明

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

允许

不允许

```
/* ex2.cpp */
#include <iostream>
using namespace std;
struct student; //结构体声明

void f2()
{
    struct student *s1;
    struct student s3, &s2=s3;
    s1.age = 15;
}
```

允许

不允许

虽可定义指针/引用,但不能  
进行成员级访问, 无意义



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.8. 结构体类型在不同位置定义时的使用 (续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

情况四(变化1): ex1.cpp和ex2.cpp构成一个程序, 有提前声明

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

允许

不允许

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

void f2()
{
    struct student *s1;
}

void f3()
{
    struct student; //结构体声明

    struct student *s1;
    s1->age = 15;
}
```

不允许

允许

不允许

虽可定义指针/引用,但不能  
进行成员级访问, 无意义



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.8. 结构体类型在不同位置定义时的使用 (续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部 (具体见后)

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



## § 7. 结构体、枚举及typedef声明新类型

### 7.2. 结构体类型

#### 7.2.8. 结构体类型在不同位置定义时的使用 (续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

问题: 如何在其它位置访问定义和使用结构体?

<pre>/* ex.h */ struct student { //结构体定义     ...; };</pre>	<pre>/* ex2.cpp */ #include &lt;iostream&gt; #include "ex.h" ← using namespace std;  int f2() {     可定义/使用student型各种变量 ✓ }</pre>
<pre>/* ex1.cpp */ #include &lt;iostream&gt; #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 ✓ }</pre>	<div>解决方法: 在头文件中定义</div>



## § 7. 结构体、枚举及typedef声明新类型

### 7.3. 枚举类型

#### 7.3.1. 含义

当变量的取值在**有限范围**内且**离散**时(可一一列出), 称为枚举类型

**例: 性别、星期、月份、血型**

#### 7.3.2. 枚举类型的定义

```
enum 枚举类型名 {枚举元素1, ..., 枚举元素n};
```

```
enum sex {male, female};
```

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

```
enum month {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
```

```
enum blood_type {A, B, O, AB};
```

★ 枚举类型和元素的命名同变量

★ 枚举元素也称**枚举常量**, 不是字符串, 不加",", 作为整型常量处理, 值从0开始顺序递增, 也可自行指定, 在程序执行中不可变

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

0 1 2 3 4 5 6

```
enum week {sun=7, mon=1, tue, wed, thu, fri, sat};
```

7 1 2 3 4 5 6

```
enum week {sun=7, mon, tue, wed, thu, fri, sat};
```

7 8 9 10 11 12 13

★ 如果执行的常量值出现重叠, 不算错误 (**但会造成误解**)

```
enum week {sun=3, mon=1, tue, wed, thu, fri, sat};
```

3 1 2 3 4 5 6





## § 7. 结构体、枚举及typedef声明新类型

### 7.3. 枚举类型

#### 7.3.3. 枚举类型变量的定义

**enum** 枚举类型名 变量名 (红色阴影: 在C++下定义时, enum关键字可省略)

**enum** week w1, w2;

#### 7.3.4. 枚举类型变量的使用

赋值: w1=mon w2=fri w2=w1

★ 不能直接赋整型量, 需要进行强制类型转换

w1 = 3;

w1 = week(3);

w1 = (week)3;

错误

正确

正确

error C2440: “=”: 无法从“int”转换为“week”  
message : 强制转换为枚举类型要求显式强制转换(static\_cast、C 样式强制转换或函数样式强制转换)

In function 'int main()':  
[Error] invalid conversion from 'int' to 'week' [-fpermissive]

比较: w1==mon w2>sat w2>=w1

★ 按枚举常量对应的值进行比较

输出: 直接输出: (按整型输出)

w1 = wed;

cout << w1 << endl; 3

间接输出: (可以是自定义的任意格式)

输入: 直接输入: 只能用C方式, 且不检查范围

week w1;

scanf("%d", &w1);

cin >> w1; 不允许

error C2678: 二进制“>>”: 没有找到接受“std::istream”类型的左操作数的运算符(或没有可接受的转换)

```
switch(w1) {  
    case sun:  
        cout << "Sunday";  
        break;  
    ...  
}
```

```
if (w1 == sun)  
    cout << "星期天";  
else if (...)  
    ...
```

间接输入: 可以多种格式

char s[80];

cin >> s;

if (!strcmp(s, "sun"))

w1=sun;

else if (...)

```
int w;  
cin >> w;  
w1=(week)w;
```



## § 7. 结构体、枚举及typedef声明新类型

### 7.3. 枚举类型

#### 7.3.4. 枚举类型变量的使用

```
#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++)
        cout << *p << " ";
    cout << endl;

    return 0;
}
```

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

int main()
{
    int i, m[13], *p;

    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;

    for (p=&m[1]; p<m+13; p++)
        cout << *p << " ";
    cout << endl;

    return 0;
}
```

#### 特别说明:

- ★ enum枚举类型不属于一定要使用的概念，左右例子分别是使用/未使用enum的情况，功能相同
- ★ 请大家自行评估可读性并选择适合自己的风格，不强求



## § 7. 结构体、枚举及typedef声明新类型

### 7.4. 用typedef声明类型

#### 7.4.1. 含义

用新的名称来等价代替已有的数据类型

★ 不产生新类型，仅使原有类型有新的名称

★ 建议声明的新类型名称为大写，与系统类型区分

#### 7.4.2. 使用

声明名称：

typedef 已有类型 新名称；

typedef int INTEGER；

typedef struct student STUDENT； //C++无此需求

typedef int ARRAY[10]；

typedef char \* STRING；

用新名称定义变量及等价对应关系：

INTEGER i, j；

STUDENT s1, s2[10], \*s3；

ARRAY a, b[5]；

STRING p, x[10]；

等价方式

int i, j；

student s1, s2[10], \*s3；

int a[10], b[5][10]；

char \*p, \*x[10]；

```
struct student {  
    ...  
};  
struct student s1; //C方式  
STUDENT s1; //C方式  
student s1; //C++方式
```

C方式的另一种小技巧，将  
无名结构体声明为student

```
typedef struct {  
    ...  
} student;  
student s1; //C方式
```



## § 7. 结构体、枚举及typedef声明新类型

### 7.5. 用typedef声明类型

#### 7.5.3. 声明新类型的一般步骤

- ① 以现有类型定义一个变量
- ② 将变量名替换为新类型名
- ③ 加typedef
- ④ 完成, 可定义新类型的变量

特别说明:

- ★ typedef声明新类型不属于必须使用的方法
- ★ 请大家自行评估可读性并选择适合自己的风格, 不强求

```
① int i;  
② int INTEGER;  
③ typedef int INTEGER;  
④ INTEGER i, j;
```

```
① int a[10];  
② int ARRAY[10];  
③ typedef int ARRAY[10];  
④ ARRAY a, b[5];
```

```
① char *s;  
② char *STRING;  
③ typedef char *STRING;  
④ STRING p, x[10];
```

```
#include <iostream>  
#include <cstring>  
using namespace std;  
  
typedef const char* STRING;  
  
int main()  
{  
    const char *p1="house";  
    STRING p2="horse";  
  
    if (strcmp(p1, p2)>0)  
        cout<<"大于"<<endl;  
    else  
        cout<<"不大于"<<endl;  
  
    return 0;  
}
```



### § 4. 函数

#### 4.8. 函数的重载

使用:

- ★ 要求同名函数的参数个数、参数类型不能完全相同
- ★ 返回类型及参数名不做检查(仅这两个不同认为错)
- ★ 若参数类型是由typedef定义的不同名称的相同类型, 则会产生二义性 (typedef具体见后续07模块)

```
typedef int INTEGER;  
    相当于给int起个别名叫INTEGER, 具体见后续  
int fun(int a);      错误  
INTEGER fun(INTEGER a);
```

★ 使用方法与原来的类型一致, 与原类型可直接混用, 不需要进行强制类型转换