



§ 7. 结构体、类和对象

7.1. 用户自定义类型的引入

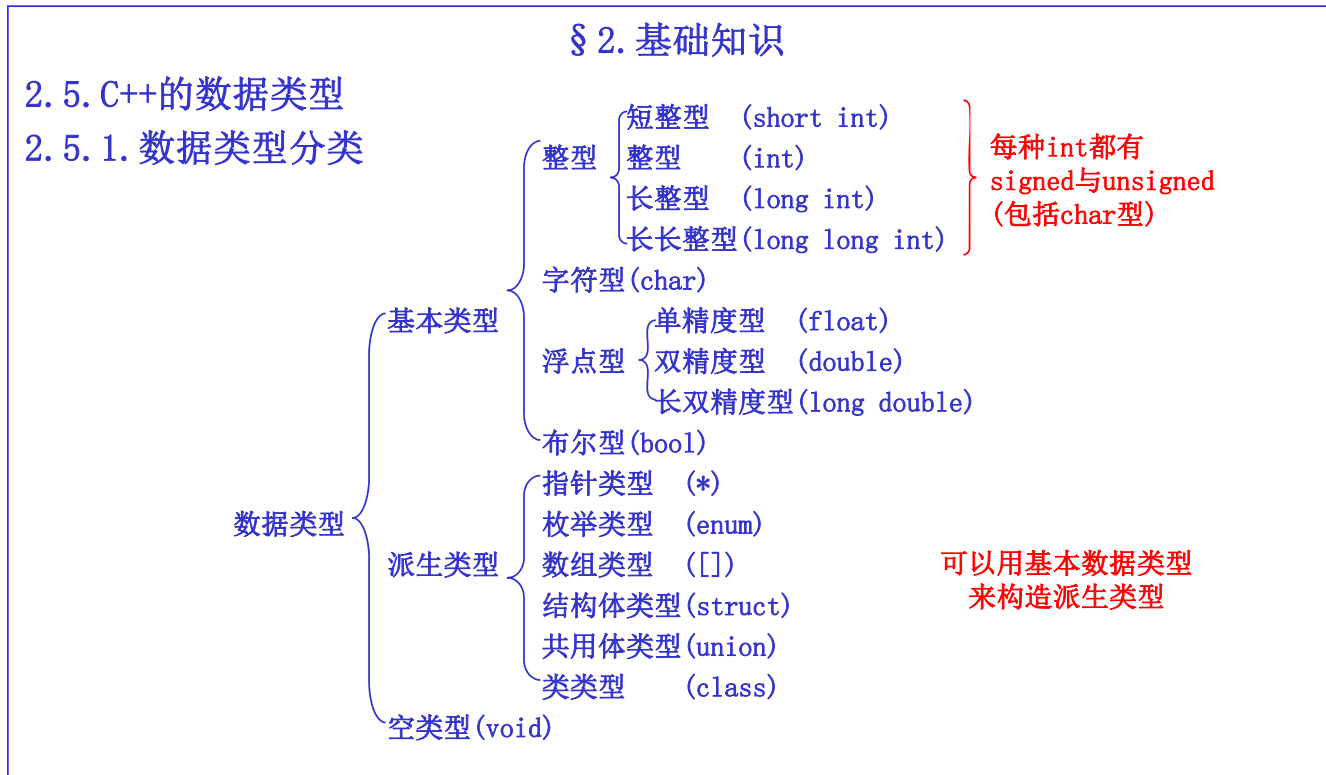
7.1.1. 用户自定义类型(派生类型)的含义

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、类和对象

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、类和对象

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、类和对象

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(编译器思维)

```
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：套娃式定义导致无法确定 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、类和对象

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**结构体类型最后**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**结构体成员之间**可能会有**填充字节**



§ 7. 结构体、类和对象

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：

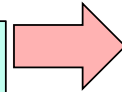
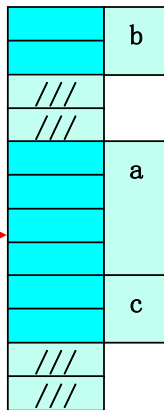
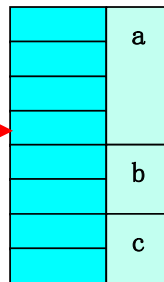
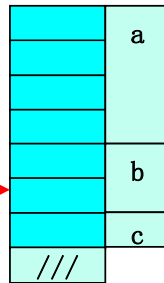
//例1: 结构体声明与字节对齐

```
#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、类和对象

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：

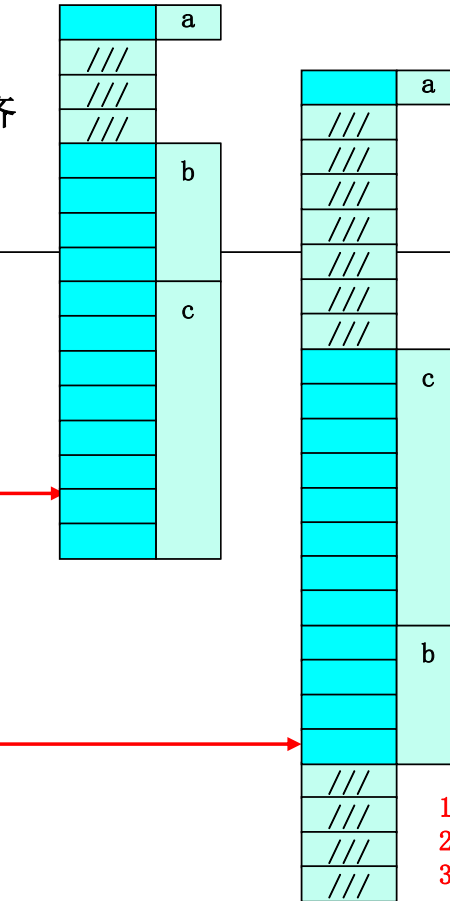
结构体的成员对齐：

//例2：结构体声明与字节对齐

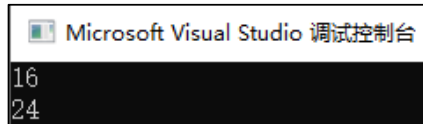
```
#include <iostream>
using namespace std;
struct s1 {
    char  a;
    int   b;
    double c;
};
struct s2 {
    char  a;
    double c;
    int   b;
};
int main()
{
    cout << sizeof(s1) << endl;
    cout << sizeof(s2) << endl;
}
```

理论：13字节
实际：16字节

理论：13字节
实际：24字节



- 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、类和对象

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()
```

```
{
```

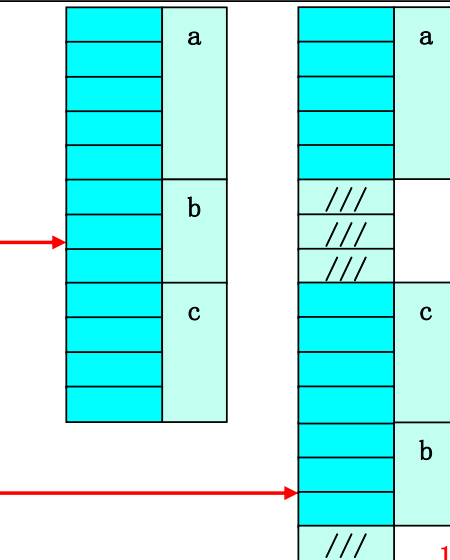
```
    cout << sizeof(s1) << endl;
```

```
    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

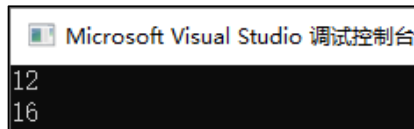
2、s2中，a数组越界4 ~ 11字节，会影响c/b的取值，但不会导致系统错误

=> 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4的倍数





§ 7. 结构体、类和对象

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、类和对象

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、类和对象

7.2. 结构体类型

7.2.4. 结构体变量的使用

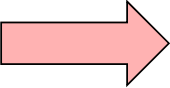
7.2.4.1. 形式

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、类和对象

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、类和对象

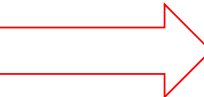
7.2. 结构体类型

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、类和对象

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、类和对象

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、类和对象

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;
```

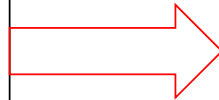
```
cout << s2[1].age << s2[1].name;
```

```
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、类和对象

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、类和对象

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
```

- 1、为什么部分转void
- 2、为什么部分无&



§ 7. 结构体、类和对象

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>
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;
```

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

```
    cout << (void *)(&s1.sex) << endl; 地址Y(X+24)
    cout << (void *)(&s1.sex+1)<<endl; 地址Y + 1
```

```
    cout << &s1.age << endl;           地址Z(Y+1+3)
    cout << &s1.age+1 << endl;         地址Z + 4
    return 0;
}
```

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



§ 7. 结构体、类和对象

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、类和对象

7.2. 结构体类型

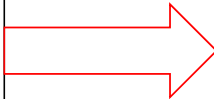
7.2.6. 指向结构体变量的指针

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、类和对象

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、类和对象

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;
}
```

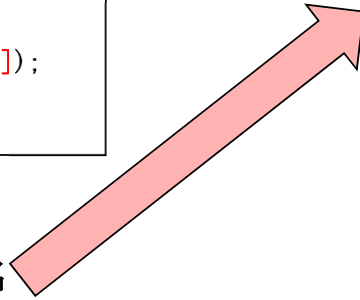
```
void fun(struct student *s)
{
    struct student s[];
    ...;
}
int main()
{
    struct student s1, s2[10];
    ...
    fun(&s1);
    ...
    fun(s2);
    return 0;
}
```



```
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



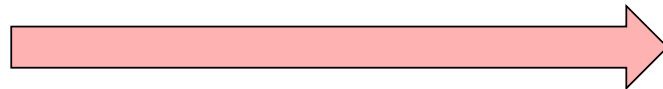
```
void fun(struct student &s)
{
    ...;
}
int main()
{
    struct student s1;
    ...
    fun(s1);
    ...
    return 0;
}
```



```
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



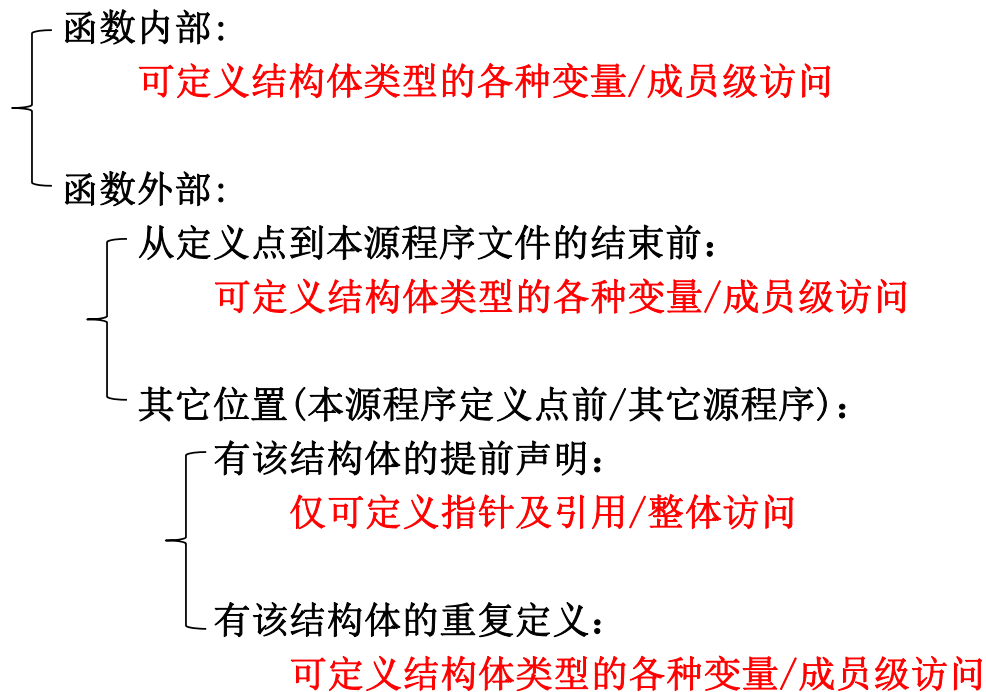


§ 7. 结构体、类和对象

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、类和对象

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、类和对象

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、类和对象

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 <iostream> 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 <iostream> using namespace std; int f2() { 不可定义/使用student型各种变量 × }</pre>
---	--



§ 7. 结构体、类和对象

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、类和对象

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、类和对象

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、类和对象

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



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.1. 类的引入

★ 使用结构体带来的好处

★ 能否使结构体的表达更清晰易懂

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];
    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

第07模块例，从左到右依次是：

- 1、6个独立变量
- 2、1个结构体变量有6个成员，在同一个函数中实现输入/输出/计算等不同功能（用相同下标控制对同一变量的访问）
- 3、1个结构体变量有6个成员，用不同公共函数传不同变量指针/引用方式实现（用相同下标控制对同一变量的访问）
- 4、（期望）1个结构体变量有6个成员，用成员.成员函数()方式实现

```
const int N=100;
struct student {
    ...;
};

//形参类型为引用*2/指针*1
//无特殊含义，仅表示均可用
void input(student &stu)
{
    //输入某学生的6个成员
}

void output(student &stu)
{
    //输出某学生的6个成员
}

void grade(student *stu)
{
    //根据成绩打印等级
}

int main()
{
    struct student s1, s2[N];
    input (s2[17]);
    output(s1);
    grade (&s2[23]);
    grade (&s1)
}
```

公共函数，通过传入
不同元素的值/地址
来访问各元素

```
const int N=100;
struct student {
    ...;
};

//同cin.good()/cout.put()形式
//称这种形式的函数为成员函数
void ...input(...)
{
    //输入某学生的6个成员
}

void ...output(...)
{
    //输出某学生的6个成员
}

void ...grade(...)
{
    //根据成绩打印等级
}

int main()
{
    struct student s1, s2[N];
    s2[17].input(...);
    s1.output(...);
    s2[23].grade(...);
    s1.grade(...);
}
```

改为这种形式，
是否可读性更好？
更容易理解？



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.1. 类的引入

在结构体只包含数据成员的基础上，引入成员函数的概念，使结构体同时拥有数据成员和成员函数

7.3.2. 声明类类型

```
class student {  
    int num;  
    char name[20];  
    char sex;  
    void display()  
    {  
        cout << "num:" << num << endl;  
        cout << "name:" << name << endl;  
        cout << "sex:" << sex << endl;  
    }  
};
```

★ 类类型的使用与结构体的使用方法基本相同

§ 7. 结构体、类和对象

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

所有“结构体”替换为“类”，均有效

- ★ 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
- ★ 每个成员的类型可以相同, 也可以不同
- ★ 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
- ★ 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数据成员的大小的总和, 可以用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)

不含成员函数

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

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, 因此结构体类型最后可能会有填充字节
- ★ 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, 因此结构体成员之间可能会有填充字节



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.2. 声明类类型

★ 用sizeof(类名)计算类的大小时，成员函数不占用空间

★ 通过类的**成员访问限定符**(private/public)，可以指定成员的属性是私有(private)或公有(public)，私有成员不能被外部函数访问，公有成员可被外部函数所访问，具体可由实际应用需求决定

- 内部函数：该class的成员函数
- 外部函数：其它函数
- 类的成员访问限定符是限制“外部函数”的访问，类的“内部函数”不受限定符的限制
- **建议**数据成员private，成员函数public

注意：和函数部分的静态函数/
外部函数的概念有差别!!!

★ 在类的定义中，private/public出现的顺序，次数无限制

```
class student {  
    public:  
        void display()  
        {  
            cout << "num:" << num << endl;  
            cout << "name:" << name << endl;  
            cout << "sex:" << sex << endl;  
        }  
    private:  
        int num;  
        char name[20];  
        char sex;  
};
```

class作为一个整体，
不必考虑对数据成员
的访问在数据成员的
定义之前

```
class student {  
    private:  
        int num;  
    public:  
        void display()  
        {  
            cout << "num:" << num << endl;  
            cout << "name:" << name << endl;  
            cout << "sex:" << sex << endl;  
        }  
    private:  
        char name[20];  
        char sex;  
};
```

```
class student {  
    private:  
        int num;  
        char name[20];  
        char sex;  
    public:  
        void display()  
        {  
            cout << "num:" << num << endl;  
            cout << "name:" << name << endl;  
            cout << "sex:" << sex << endl;  
        }  
};
```

本例：无论三个数据成员的
限定符是什么；无论
display函数的限定
符是什么；都不影响
display函数对三个
数据成员的访问



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.3. 对象的定义和访问

7.3.3.1. 先定义类，再定义对象

结构体类型 $\xrightarrow{\text{实例化}}$ 变量

类 $\xrightarrow{\text{实例化}}$ 对象

★ 含义相同，称呼不同

<pre>class student { ... }; student s1; student s2[10]; student *s3;</pre>	<pre>struct student { ... }; struct student s1; struct student s2[10]; struct student *s3;</pre>
--	--

★ 结构体变量/类对象占用实际的内存空间，根据不同类型(静态/动态/全局/局部)在不同区域进行分配

7.3.3.2. 在定义类的同时定义对象

<pre>class student { ... } s1, s2[10], *s3; student s4;</pre>	<pre>struct student { ... } s1, s2[10], *s3; struct student s4;</pre>
---	---

★ 可以再次用7.3.3.1的方法定义新的变量/类对象

7.3.3.3. 直接定义对象(类无名)

<pre>class { ... } s1, s2[10], *s3;</pre>	<pre>struct { ... } s1, s2[10], *s3;</pre>
---	--

★ 因为结构体/类无名，因此无法再用7.3.3.1的方法进行新的变量/对象定义



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.4. 类与结构体的比较

★ 在C++中，结构体也可以加成员函数，能够实现和类完全一样的功能

```
class student {  
    private:  
        int num;  
        char name[20];  
        char sex;  
    public:  
        void display()  
        {  
            cout << "num:" << num << endl;  
            cout << "name:" << name << endl;  
            cout << "sex:" << sex << endl;  
        }  
};
```

在.cpp中，替换为struct，功能完全相同
在.c中则报语法错误

```
demo.c demo-c (全局范围)  
1 #include <stdio.h>  
2  
3 struct student {  
4     private:  
5         int num;  
6         char* name;  
7     };  
8  
9 int main()  
10 {  
11     return 0;  
12 }
```

```
demo.c(4,1): error C2061: 语法错误: 标识符“private”  
demo.c(7,1): error C2059: 语法错误: “}”
```



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.4. 类与结构体的比较

- ★ 在C++中，结构体也可以加成员函数，能够实现和类完全一样的功能
- ★ 若不指定成员访问限定符，则struct缺省为public，class缺省为private

<pre>#include <iostream> using namespace std; struct student { int num; char name[20]; char sex; void display() { cout << num << endl; cout << name << endl; cout << sex << endl; } }; int main() { student s1; s1.num = 1001; return 0; }</pre>	<pre>#include <iostream> using namespace std; class student { int num; char name[20]; char sex; void display() { cout << num << endl; cout << name << endl; cout << sex << endl; } }; int main() { student s1; s1.num = 1001; return 0; }</pre>	<pre>class student { int num; char name[20]; char sex; void display() { cout << num << endl; cout << name << endl; cout << sex << endl; } }; 全部是private</pre>	<pre>struct student { int num; char name[20]; char sex; void display() { cout << num << endl; cout << name << endl; cout << sex << endl; } }; 全部是public</pre>
<pre>全部public, 外界(main)可访问, 与C相比,多成员函数</pre>	<pre>全部private, 外界(main)不可访问,编译错</pre>	<pre>class student { int num; char name[20]; char sex; public: void display() { ... } }; 私有 公有</pre>	<pre>struct student { int num; char name[20]; char sex; public: void display() { ... } }; 公有 公有</pre>
<pre>(17,7): error C2248: "student::num": 无法访问 private 成员(在 "student" 类中声明) (5): message : 参见 "student::num" 的声明 (4): message : 参见 "student" 的声明</pre>			



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.5. 对象成员的访问

7.3.5.1. 通过对象名访问对象中的成员

7.3.5.2. 通过指向对象的指针访问对象中成员

7.3.5.3. 通过对象的引用来访问对象中的成员

```
class student {  
    private:  
        int name[20];  
        char sex;  
    public:  
        int num;  
        void display()  
        {  
            ...  
        }  
};
```

类定义

```
int main() ★ 通过对象名访问  
对象中的成员  
{  
    student s1, s2[10];  
    s1.sex = 'm'; ✗  
    s1.num=10001; ✓  
    s1.display(); ✓  
    s2[0].sex = 'f'; ✗  
    s2[0].num=10002; ✓  
    s2[3].display(); ✓  
}
```

①

```
int main() ★ 通过指向对象的指针  
来访问对象中的成员  
{  
    student s1, *s3=&s1;  
    s1.num = 10001; ✓  
    (*s3).num = 10001; ✓  
    s3->num = 10001; ✓  
    s1.display(); ✓  
    (*s3).display(); ✓  
    s3->display(); ✓  
}
```

②

```
int main() ★ 通过对象的引用来  
访问对象中的成员  
{  
    student s1, &s3=s1;  
    s1.num = 10001; ✓  
    s3.num = 10001; ✓  
    s1.display(); ✓  
    s3.display(); ✓  
}
```

③

★ 注意访问权限，只能是public



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.5. 对象成员的访问

7.3.5.4. 访问规则

★ 只能访问公有的数据成员和成员函数

★ 数据成员可出现在其基本类型允许出现的任何地方（外部需公有）

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

★ 成员函数的参数传递规则仍为实参单向传值到形参（引用仍为别名）



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.6. 类的成员函数

7.3.6.1. 成员函数的实现

体内实现：class中给出成员函数的定义及实现过程

```
class student {  
    ...  
    public:  
        void display()  
        {  
            cout<<"num:" <<num <<endl;  
            cout<<"name:"<<name<<endl;  
            cout<<"sex:" <<sex <<endl;  
        }  
};
```

体外实现：class中给出成员函数的定义，class外部(class后)给出成员函数的实现

- ★ 函数实现时需要加**类的作用域限定符**
- ★ 即使类成员函数是体外实现方式，仍然算“内部函数”，不受private/public访问限定符限制!!!

```
class student {  
    public:  
        void display();  
};  
  
void student::display()  
{  
    cout << "num:" <<num <<endl;  
    cout << "name:" <<name <<endl;  
    cout << "sex:" <<sex <<endl;  
}
```




§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.6. 类的成员函数

7.3.6.2. 成员函数的性质

- ★ 对应类的成员函数(类函数)，一般的普通函数称为全局函数
- ★ 成员函数的定义、实现及调用时参数传递的语法规则与全局函数相同
- ★ 成员函数也受类的**成员访问限定符**的约束，只有**公有**的成员函数可以被“外部函数”调用
- ★ 私有和公有的成员函数均可以访问/调用本类的所有数据成员/成员函数，不受private/public的限制
(再次强调: private/public是用来限制外部函数对类数据成员/成员函数的访问)

<pre>class test { private: int a; int f1(); public: int b; int f2(); int f3(); };</pre>	<pre>int test::f1() { a=10; ✓ b=15; ✓ f2(); ✓ } int test::f2() { ... } int test::f3() { a=20; ✓ b=25; ✓ f1(); ✓ }</pre>	<pre>int main() { test t1; t1.a=10; ✗ t1.f1(); ✗ t1.b=15; ✓ t1.f2(); ✓ t1.f3(); ✓ }</pre>
---	---	--



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.6. 类的成员函数

7.3.6.2. 成员函数的性质

★ 全局函数与成员函数可以同名，按照低层屏蔽高层的原则进行，也可以通过**全局作用域符 (:: 级别最高)**强制访问高层

<pre>class test { ... public: int fun(); int f1(); }; int fun() 全局函数 { ... }</pre>	<pre>int test::fun() 类函数 { ... } int test::f1() { fun(); 类函数 ::fun(); 全局函数 }</pre> <p>成员函数内部</p>	<pre>int main() { test t1; t1.fun(); 类函数 fun(); 全局函数 }</pre> <p>全局函数中表示不会冲突</p>
---	---	--



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.6. 类的成员函数

7.3.6.2. 成员函数的性质

★ 全局函数与成员函数可以同名，按照低层屏蔽高层的原则进行，也可以通过**全局作用域符(::级别最高)**强制访问高层
(类的数据成员与全局变量也遵循此强制访问规则)

```
int a;    //全局变量
void fun()
{
    int a;    //局部变量
    a=10;    //访问局部变量
    ::a=15;   //访问全局变量(第04模块时说不行)
}
```

```
int a; //全局变量

class test {
    ...
public:
    int a; //类数据成员
    int f1();
};

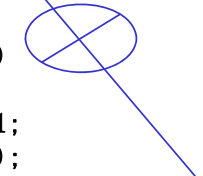
int test::f1()
{
    a=10;    //类数据成员
    ::a=15;  //全局变量
}
```

```
int a; //全局变量
class test {
    ...
public:
    int a;    //类数据成员
    int f1();
};
```

```
int test::f1()
{
    int a;    //成员函数内的自动变量
    a=5;      //自动变量
    test::a=10; //类数据成员
    ::a=15;   //全局变量
}

int main()
{
    test t1;
    t1.f1();

    cout << t1.a << endl; //类数据成员(需public)
    cout << a << endl;   //全局变量
}
```



//外部无法访问f1自动变量

注：极端情况可能出现部分成员无法访问的情况，不建议深究

```
int a; //全局变量

class test {
    public:
        int a; //类数据成员
};

int test::f1()
{
    int a; //成员函数的自动变量
    if (***) {
        long a; //if语句内的自动变量
        for (***) {
            short a; //for循环内的自动变量

            全局a/test类a/函数内a/if中a/for中a,
            共5个，如何在此处访问不同a?
        } //end of for
    } //end of if
}
```



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.6. 类的成员函数

7.3.6.3. 成员函数的存储方式

- ★ 每个类的实例对象仅包含数据成员 ($\text{sizeof}(\text{类}) = \text{所有数据成员之和}$)，根据不同的定义位置占用不同的数据空间 (静态数据区或动态数据区)
- ★ 类的成员函数占用函数(代码)区，每个类的每个成员函数 (包括体内实现和体外实现) 只占用一段空间，所有该类的对象共用成员函数的代码空间
- ★ 当通过对象调用成员函数时，系统会缺省设置一个隐含的 `this` 指针，指向被调用的对象，并以此来区分成员函数对数据成员的访问

```
class student {  
    private:  
        int num;  
    public:  
        void set(int n) {  
            num = n;  
        }  
        void display() {  
            cout << num << endl;  
        }  
};
```

```
int main()  
{  
    student s1, s2;  
    s1.set(10);  
    s2.set(15);  
    s1.display();  
    s2.display();  
    return 0;  
}
```

s1, s2 占用不同的4字节
为什么 s1.set / s2.display 时会指向不同的4字节

```
class student {  
    private:  
        int num;  
    public:  
        void set(student *this, int n) {  
            this->num = n;  
        }  
        void display(student *this) {  
            cout << this->num << endl;  
        }  
};
```

类成员函数中隐含了一个 `this` 指针
调用时会隐含传入调用对象的地址
`s1.set(10) ⇔ s1.set(&s1, 10);`
`s2.display() ⇔ s2.display(&s2);`

注: `this` 不能显式写在函数声明中,
但可显式访问



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

- ★ 公有函数可被外界调用，称为类的公共/对外接口通过**对象. 公用函数(实参表)**的方法进行调用，将函数称为**方法**，将调用过程称为**消息传递**
- ★ 如果允许外界直接改变某个数据成员的值，可直接设置属性为public (**不提倡**)
- ★ 其它不愿公开的数据成员和成员函数可设置为私有, 对外部隐蔽，但仍可通过公有函数进行访问及修改

```
class student {  
    private:  
        int num;  
    public:  
        void set(int n)  
        {  
            num = n;  
        }  
        void display()  
        {  
            cout << num << endl;  
        }  
};
```

```
int main()  
{  
    student s1, s2;  
    s1.set(10);  
    s2.set(15);  
    s1.display(); 10  
    s2.display(); 15  
  
    return 0;  
}
```

set/display函数均间接
访问了私有成员num



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

★ 公有函数的形参称为提供给外部的访问接口，在形参的数量、类型、顺序不变的情况下，私有成员的变化及公有函数实现部分的修改不影响外部的调用

```
class student {  
    private: ①  
        int num;  
    public:  
        void set(int n)  
        {    num = n;  
        }  
        void display()  
        {    cout << num << endl;  
        }  
};
```

```
class student {  
    private: ②  
        int xh;  
    public:  
        void set(int n)  
        {    xh = n;  
        }  
        void display()  
        {    printf("%d\n", xh);  
        }  
};
```

```
class student {  
    private: ③  
        int xh;  
    public:  
        void set(int n)  
        {    xh = (n>=0 ? n:0);  
        }  
        void display()  
        {    printf("%d\n", xh);  
        }  
};
```

```
int main()  
{  
    student s1, s2;  
    s1.set(10);  
    s2.set(15);  
    s1.display(); 10  
    s2.display(); 15  
    return 0;  
}
```

假设class student由乙编写
main函数由甲编写
则：乙用三种方法
甲的程序均不需要变化



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

应用实例1:

谷歌的Android 4.x内核(假装C++)

```
class picture {
```

类的私有数据成员
及成员函数
外界不可见

```
void show(char *图片名)
```

函数实现, 不可见

```
};
```

***公司的游戏软件

```
int main()
```

```
{
```

```
....
```

```
picture p1;
```

```
p1.show(文件名);
```

```
....
```

```
}
```

谷歌公司的Android 12内核

```
class picture {
```

类的私有数据成员
及成员函数
外界不可见
可能已进行过很大调整

```
void show(char *图片名)
```

函数实现, 不可见
实现过程可能与2.3完全不同

```
};
```

谷歌称12.0的显示速度
经优化后比4.x快**%
用户程序不需要变化



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

应用实例2:

A公司的乙团队: V1.0版本

```
class translation {
```

类的私有数据成员
及成员函数
外界不可见

```
void trans(char *英文)
```

函数功能为输出中文
具体实现过程不可见

```
};
```

A公司的甲团队

```
int main()
```

```
{
```

```
....
```

```
translation t1;
```

```
t1.trans("****");
```

```
....
```

```
}
```

A公司的乙团队: V1.1版本

```
class translation {
```

类的私有数据成员
及成员函数
外界不可见
可能已进行过很大调整

```
void trans(char *英文)
```

函数实现, 不可见
实现过程可能与1.0完全不同

```
};
```

V1.1比V1.0的翻译结果
更准确, 更贴切
用户程序不需要变化
两个团队能同时工作

思考:

- 1、甲乙两个团队哪个更不可替代?
- 2、你的职业期望是哪个团队?
- 3、进入不同团队对不同知识的要求?



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

7.3.7.2. 类声明和成员函数定义的分离

★ 将类的声明 (*.h) 与类成员函数的实现 (*.cpp) 分开

例1: 假设程序由ex1.cpp、ex2.cpp和ex.h共同构成

<pre>/* ex.h */ class student { private: 数据成员1; ...; 数据成员n; public: 成员函数1; ...; 成员函数2; }</pre>	<pre>/* ex1.cpp */ #include <iostream> #include "ex.h" using name space std; 返回值 student::成员函数1() { 成员函数1的实现; } ...</pre>
<pre>/* ex2.cpp */ #include <iostream> #include "ex.h" using namespace std; main及其它函数的实现</pre>	<pre>返回值 student::成员函数n() { 成员函数n的实现; }</pre>



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

7.3.7.2. 类声明和成员函数定义的分离

★ 将类的声明 (*.h) 与类成员函数的实现 (*.cpp) 分开

例2: 数学函数sqrt

<pre>/* math.h */ double sqrt(double x);</pre>	<pre>/* 实现sqrt的源码 math.cpp */ sqrt的具体实现过程被隐藏 提供lib/dll (静态/动态库)</pre>
<pre>/* test.cpp */ #include <iostream> #include <math.h> using namespace std; int main() { cout << sqrt(2) << endl; return 0; }</pre>	<pre>h+lib/dll 即可实现相同功能</pre>



§ 7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

7.3.7.2. 类声明和成员函数定义的分离

★ 将类的声明 (*.h) 与类成员函数的实现 (*.cpp) 分开

★ 在需要外部调用的地方，只要提供声明部分即可，类的实现可通过库文件 (*.lib) 或动态链接库 (*.dll) 的方式提供，而不必提供实现的源码

★ 一个程序包含多源程序文件的方法已掌握

★ 建立库文件/动态链接库的方法有兴趣可自学



§ 7. 结构体、类和对象

7. 4. 构造函数与析构函数

7. 4. 1. 引入及作用

- 构造函数: Constructor Function, 用于类对象生成时的初始化
- 析构函数: Destructor Function, 用于类对象消失时的收尾工作

7. 3. 3中

结构体类型 $\xrightarrow{\text{实例化}}$ 变量

类 $\xrightarrow{\text{实例化}}$ 对象

★ 含义相同, 称呼不同



§ 7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.2. 对象的初始化

对象的初值：在静态数据区分配的对象，数据成员初值为0；

在动态数据区分配的对象，数据成员的初值随机

(与普通变量相同)

对象的初始化方法：

(1) 若全部成员都是公有，可按结构体的方式进行初始化

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

class Time {
public:
    int f2(); //函数不占用对象空间
    int hour;
    int minute;
    int sec;
};

int main()
{
    Time t1={14,15,23};
    cout << t1.hour << ':' << t1.minute
         << ':' << t1.sec << endl;
}
```

Microsoft Visual Studio 调试控制台

14:15:23

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

class Time {
public:
    int f2(); //函数不占用对象空间
    int hour;
    int minute;
    int sec;
};

int main()
{
    Time t1=(14,15,23);
    cout << t1.hour << ':' << t1.minute
         << ':' << t1.sec << endl;
}
```

以下两种形式均报错：
Time t1(14,15,23);
Time t1=(14,15,23);

error C2440: “初始化”：无法从“int”转换为“Time”
message : 无构造函数可以接受源类型，或构造函数重载决策不明确

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

class Time {
public:
    int f2(); //函数不占用对象空间
    int hour;
    int minute;
    int sec;
};

int main()
{
    Time t1(14,15,23);
    cout << t1.hour << ':' << t1.minute
         << ':' << t1.sec << endl;
}
```

以下两种形式均报错：
Time t1(14,15,23);
Time t1=(14,15,23);

error C2440: “初始化”：无法从“initializer list”转换为“Time”
message : 无构造函数可以接受源类型，或构造函数重载决策不明确



§ 7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.2. 对象的初始化

对象的初值：在静态数据区分配的对象，数据成员初值为0；

在动态数据区分配的对象，数据成员的初值随机

(与普通变量相同)

对象的初始化方法：

(1) 若全部成员都是公有, 可按结构体的方式进行初始化 (若有私有成员, 不能用此方法)

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

class Time {
public:
    int f2(); //函数不占用对象空间
    int hour;
    int minute;
    int sec;
};

int main()
{
    Time t1={14, 15, 23};
    cout << t1.hour << ':' << t1.minute
         << ':' << t1.sec << endl;
}
```

以下两种形式均报错：
Time t1(14, 15, 23);
Time t1=(14, 15, 23);

Microsoft Visual Studio 调试控制台

14:15:23

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

class Time {
public:
    int hour;
    int minute;
private:
    int sec;
    int f2(); //函数不占空间
};

int main()
{
    Time t1={14, 15, 23};
    cout << t1.hour << ':' << t1.minute
         << ':' << t1.sec << endl;
}
```

error C2440: “初始化”: 无法从“initializer list”转换为“Time”
message : 无构造函数可以接受源类型, 或构造函数重载决策不明确

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

class Time {
    int f1(); //缺省私有
public:
    int hour;
    int minute;
    int sec;
};

int main()
{
    Time t1={14, 15, 23};
    cout << t1.hour << ':' << t1.minute
         << ':' << t1.sec << endl;
}
```

只要所有数据成员均公有即可

Microsoft Visual Studio 调试控制台

14:15:23



§ 7. 结构体、类和对象

7. 4. 构造函数与析构函数

7. 4. 2. 对象的初始化

对象的初始化方法:

- (1) 若全部成员都是公有, 可按结构体的方式进行初始化 (若有私有成员, 不能用此方法)
- (2) 写一个赋初值的公有成员函数, 在其它成员被调用之前进行调用

```
class Time {  
    private:  
        int hour;  
        int minute;  
        int sec;  
    public:  
        void set(int h, int m, int s)  
        {  
            hour=h;  
            minute=m;  
            sec=s;  
        }  
};  
  
int main()  
{  
    Time t;  
  
    t.set(14, 15, 23);  
    t. 其它  
}
```

```
class Time {  
    public:  
        int hour=0;  
        int minute=0;  
        int sec=0;  
};
```

不同对象的值被统一初始化,
无法个性化

- (3) 声明类时对数据成员进行初始化
(C++11标准支持, 目前双编译器均可)



§ 7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

引入：完成对象的初始化工作, 对象建立时被自动调用

形式：与类同名, 无返回类型 (非void, 也不是缺省int)

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

class Time {
private:
    int hour, minute, sec; //相同类型可以写在一行上
public:
    Time()
    {
        hour=0;
        minute=0;
        sec=0;
    }
    void display()
    {
        cout << hour << ':' << minute << ':' << sec << endl;
    }
};

int main()
{
    Time t; //t的三个成员都是0
    t.display();
}
```

体内实现

Microsoft Visual Studio 调试控制台

0:0:0

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

class Time {
private:
    int hour, minute, sec; //相同类型可以写在一行上
public:
    Time();
    void display()
    {
        cout << hour << ':' << minute << ':' << sec << endl;
    }
};

Time::Time()
{
    hour=0;
    minute=0;
    sec=0;
}

int main()
{
    Time t; //t的三个成员都是0
    t.display();
}
```

体外实现

Microsoft Visual Studio 调试控制台

0:0:0



§ 7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

引入：完成对象的初始化工作, 对象建立时被自动调用

形式：与类同名, 无返回类型 (非void, 也不是缺省int)

使用：

- ★ 对象建立时被自动调用
- ★ 构造函数必须公有
- ★ 若不指定构造函数, 则系统缺省生成一个构造函数, 形式为无参空体
- ★ 若用户定义了构造函数, 则缺省构造函数不再存在
- ★ 构造函数既可以体内实现, 也可以体外实现
- ★ 允许定义带参数的构造函数, 以解决无参构造函数初始化各对象的值相同的情况 (个性化初值)

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

class Time {
private:
    int hour;
    int minute;
    int sec;
public:
    Time(int h, int m, int s);
    void display()
    { cout << hour << ':' << minute << ':' << sec << endl;
    }
};

Time::Time(int h, int m, int s)
{
    hour   = h;
    minute = m;
    sec    = s;
}
```

也允许体内实现

```
int main()
{
    Time t1(14, 15);
    Time t2;
}
```

```
cpp-demo.cpp(24, 19): error C2661: "Time::Time": 没有重载函数接受 2 个参数
cpp-demo.cpp(25, 10): error C2512: "Time": 没有合适的默认构造函数可用
cpp-demo.cpp(4, 7): message : 参见 "Time" 的声明
```

```
int main()
{
    Time t1(14, 15, 23);
    Time t2{15, 16, 24};
    Time t3={16, 17, 25};
    t1.display();
    t2.display();
    t3.display();
}
```

三种
形式
均可

Microsoft Visual Studio 调试控制台

```
14:15:23
15:16:24
16:17:25
```



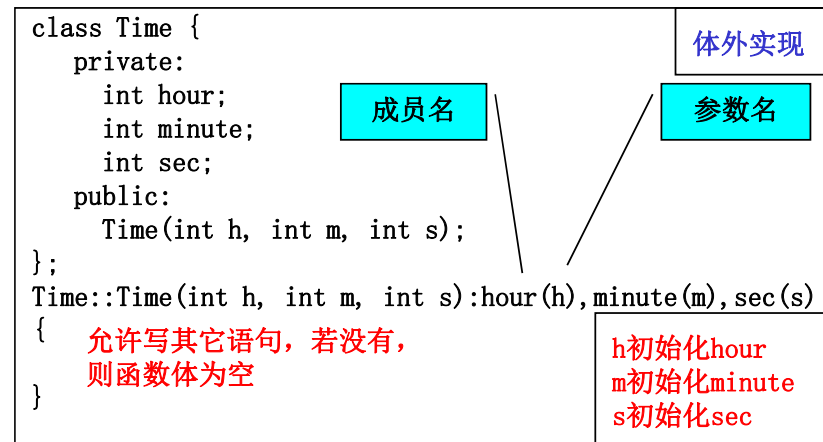
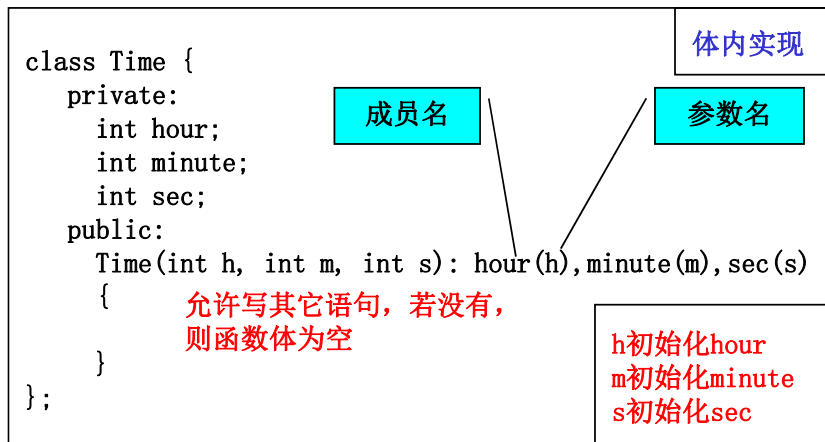
§ 7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

使用:

★ 有参构造函数可以使用参数初始化表来对数据成员进行初始化



★ 有参构造函数可以使用参数初始化表来对数据成员进行初始化 (仅适用于简单的赋值)

```
Time::Time(int h, int m, int s)  
{  
    if (h>=0 && h<=23)  
        hour = h;  
    else  
        hour = 0;  
    ....  
}
```

无法通过参数初始化表实现



§ 7. 结构体、类和对象

7. 4. 构造函数与析构函数

7. 4. 3. 构造函数的引入及使用

使用:

★ 允许定义带参数的构造函数, 以解决无参构造函数初始化各对象的值相同的情况(个性化初值)

引申问题: 构造函数如何做到更自由的个性化?

例:

```
int main()
{
    Time t1(14);           //期望是14:00:00
    Time t2(14, 15);       //期望是14:15:00
    Time t3(14, 15, 23);   //期望是14:15:23
}
```

//方案1: 用三个构造函数实现不同功能
=> 构造函数的名称是类名
=> 三个不同函数名称相同

```
Time(int h)
{
    hour   = h;
    minute = 0;
    sec    = 0;
}
```

```
Time(int h, int m)
{
    hour   = h;
    minute = m;
    sec    = 0;
}
```

```
Time(int h, int m, int s)
{
    hour   = h;
    minute = m;
    sec    = s;
}
```

//方案2: 用一个构造函数实现,
通过形参有默认值的方式来实现

```
Time(int h, int m = 0, int s = 0)
{
    hour   = h;
    minute = m;
    sec    = s;
}
```

调用时, 如果m/s有值, 则用给定值, 否则用默认值



§ 7. 结构体、类和对象

7. 4. 构造函数与析构函数

7. 4. 3. 构造函数的引入及使用

使用:

★ 构造函数允许重载

```
class Time {  
    ...  
    public:  
        Time();  
        Time(int h, int m, int s);  
};
```

```
Time::Time()
```

```
{ hour   = 0;  
  minute = 0;  
  sec    = 0;  
}
```

```
Time::Time(int h, int m, int s)
```

```
{ hour   = h;  
  minute = m;  
  sec    = s;  
}
```

```
int main()
```

```
{ Time t(14, 15, 23); //正确  
  Time t2;           //正确  
  ...  
}
```

也可以体内实现



§ 7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

使用:

★ 构造函数允许带默认参数, 但要注意可能与重载产生二义性冲突

<pre>class Time { ... public: Time(); Time(int h, int m, int s=0); }; Time::Time() { hour = 0; minute = 0; sec = 0; } Time::Time(int h, int m, int s) { hour = h; minute = m; sec = s; } int main() { Time t1(14, 15, 23); //正确 Time t2(14, 15); //正确 Time t3; //正确 }</pre>	无参与带缺省参数的重载, 不冲突 适应带0/2/3个参数的情况
--	---------------------------------------

<pre>class Time { ... public: Time(); Time(int h=0, int m=0, int s=0); }; Time::Time() { hour = 0; minute = 0; sec = 0; } Time::Time(int h, int m, int s) { hour = h; minute = m; sec = s; } int main() { Time t1(14, 15, 23); //正确 Time t2(14, 15); //正确 Time t3(14); //正确 Time t4; //错误 }</pre>	无参与带缺省参数的重载, 冲突!!!
---	-----------------------

```
cpp-demo.cpp(25): error C2668: "Time::Time": 对重载函数的调用不明确  
cpp-demo.cpp(14,5): message : 可能是 "Time::Time(int,int,int)"  
cpp-demo.cpp(8,5): message : 或 "Time::Time(void)"  
cpp-demo.cpp(25,12): message : 尝试匹配参数列表 "()" 时
```



§ 7. 结构体、类和对象

7. 4. 构造函数与析构函数

7. 4. 3. 构造函数的引入及使用

使用:

★ 构造函数也可以显式调用，一般用于带参构造函数

```
class Test {  
    private:  
        int a;  
    public:  
        Test(int x) {  
            a=x;  
        }  
};  
Test fun()  
{ ...  
    return Test(10); //显式  
}  
  
int main()  
{  
    Test t1(10);      //隐式  
    Test t2=Test(10); //显式  
    Test t3=Test{10}; //显式  
}
```



§ 7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.4. 析构函数

引入：在对象被撤销时(生命期结束)时被自动调用，完成一些善后工作(主要是内存清理)，但不是撤销对象本身

形式：

~类名();

★ 无返回值(非void, 也不是int)，无参，不允许重载

使用：

★ 对象撤销时被自动调用，用户不能显式调用

★ 析构函数必须公有

★ 若不指定析构函数，则系统缺省生成一个析构函数，形式为无参空体

★ 若用户定义了析构函数，则缺省析构函数不再存在

★ 析构函数既可以体内实现，也可以体外实现

★ 在数据成员没有动态内存申请需求的情况下，一般不需要定义析构函数(动态内存申请为后续课程内容，此处不再展开)

```
class Time {  
    ...  
    public:  
        Time() //构造体内实现  
        { ...  
        }  
  
        ~Time() //析构体内实现  
        { ...  
        }  
};
```

```
class Time {  
    ...  
    public:  
        Time(); //构造声明  
        ~Time(); //析构声明  
};  
  
Time::Time() //构造体外实现  
{ ...  
}  
  
Time::~~Time() //析构体外实现  
{ ...  
}
```



§ 7. 结构体、类和对象

7. 4. 构造函数与析构函数

7. 4. 5. 构造函数与析构函数的调用时机

构造函数:

- ★ 自动对象(形参) : 函数中变量定义时
- ★ 静态局部对象 : 第一次调用时
- ★ 静态全局/外部全局对象: 程序开始时
- ★ 动态申请的对象: 后续课程课内容(略)

main开始前

析构函数:

- ★ 自动对象(形参) : 函数结束时
- ★ 静态局部对象 : 程序结束时(在全局之前)
- ★ 静态全局/外部全局对象: 程序结束时
- ★ 动态申请的对象: 后续课程内容(略)

main结束后



§ 7. 结构体、类和对象

7.4.5. 构造函数与析构函数的调用时机

```
#include <iostream>
using namespace std;
class Time {
    private:
        int hour;
        int minute;
        int second;
    public:
        Time(int h=0, int m=0, int s=0);
        ~Time();
};
Time::Time(int h, int m, int s)
{
    hour    = h;
    minute  = m;
    second  = s;
    cout << "Time Begin" << endl;
}
Time::~~Time()
{
    cout << "Time End" << endl;
}
```

```
void fun()
{
    Time t1;
    cout << "addr:" << &t1 << endl;
    cout << "fun" << endl;
}
int main()
{
    cout << "main begin" << endl;
    fun();
    cout << "continue" << endl;
    fun();
    cout << "main end" << endl;
}
```

程序的运行结果:

```
main begin
Time Begin
addr:地址a
fun
Time End
continue
Time Begin
addr:地址a(同上)
fun
Time End
main end
```

- 1、函数调用时分配空间
结束时回收空间
- 2、函数多次调用则多次
分配/回收空间

验证了函数模块遗留的什么问题?



§ 7. 结构体、类和对象

7.4.5. 构造函数与析构函数的调用时机

```
#include <iostream>
using namespace std;
class Time {
    private:
        int hour;
        int minute;
        int second;
    public:
        Time(int h=0, int m=0, int s=0);
        ~Time();
};
Time::Time(int h, int m, int s)
{
    hour    = h;
    minute  = m;
    second  = s;
    cout << "Time Begin" << endl;
}
Time::~~Time()
{
    cout << "Time End" << endl;
}
```

```
void fun()
{
    static Time t1;
    cout << "fun" << endl;
}
int main()
{
    cout << "main begin" << endl;
    fun();
    cout << "continue" << endl;
    fun();
    cout << "main end" << endl;
}
```

程序的运行结果:

```
main begin
Time Begin
fun
continue
fun
main end
Time End
```

- 1、函数第1次调用时分配
- 2、后续函数调用不分配
- 3、全部程序结束后回收

验证了函数模块遗留的什么问题?



§ 7. 结构体、类和对象

7.4.5. 构造函数与析构函数的调用时机

```
#include <iostream>
using namespace std;
class Time {
    private:
        int hour;
        int minute;
        int second;
    public:
        Time(int h=0, int m=0, int s=0);
        ~Time();
};
Time::Time(int h, int m, int s)
{
    hour    = h;
    minute  = m;
    second  = s;
    cout << "Time Begin" << endl;
}
Time::~~Time()
{
    cout << "Time End" << endl;
}
```

```
Time t1;
void fun()
{
    cout << "fun begin" << endl;
    cout << "fun end" << endl;
}
int main()
{
    cout << "main begin" << endl;
    fun();
    cout << "main end" << endl;
}
```

程序的运行结果:

```
Time begin
main begin
fun begin
fun end
main end
Time End
```



§ 7. 结构体、类和对象

7.5. 对象数组

7.5.1. 形式

类型 对象名[整型常量表达式] : 一维数组

类型 对象名[整型常量表达式1][整型常量表达式2] : 二维数组

```
Time t[10];
```

```
Time s[3][4];
```

7.5.2. 定义对象时进行初始化

★ 若未定义构造函数或构造函数无参，则按简单对象使用无参构造函数的规则进行

<pre>#include <iostream> using namespace std; class Time { private: int hour, minute, sec; public: void display() { cout << hour << ':' << minute << ':' << sec << endl; } }; int main() { Time t[10]; for (int i=0; i<10; i++) t[i].display(); }</pre>	<pre>Microsoft Visual Studio 调试控制台 -858993460:-858993460:-858993460 -858993460:-858993460:-858993460 -858993460:-858993460:-858993460 -858993460:-858993460:-858993460 -858993460:-858993460:-858993460 -858993460:-858993460:-858993460 -858993460:-858993460:-858993460 -858993460:-858993460:-858993460 -858993460:-858993460:-858993460 -858993460:-858993460:-858993460</pre>
	<p>10个元素的三个成员的值都是随机的，因为调用缺省构造，什么也没做</p>

<pre>#include <iostream> using namespace std; class Time { private: int hour, minute, sec; public: Time() { hour=0; minute=0; sec=0;} void display() { cout << hour << ':' << minute << ':' << sec << endl; } }; int main() { Time t[10]; for (int i=0; i<10; i++) t[i].display(); }</pre>	<pre>Microsoft Visual Studio 调试控制台 0:0:0 0:0:0 0:0:0 0:0:0 0:0:0 0:0:0 0:0:0 0:0:0 0:0:0 0:0:0</pre>
	<p>10个元素的三个成员的值都是0，因为调用无参构造</p>



§ 7. 结构体、类和对象

7.5. 对象数组

7.5.2. 定义对象时进行初始化

★ 若带参构造函数只带一个参数，可用数组定义时初始化的方法进行

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

class Time {
private:
    int hour, minute, sec;
public:
    Time(int h)
    {
        hour = h;
        minute = 0;
        sec = 0;
    }
    void display()
    {
        cout << hour << ':' << minute << ':' << sec << endl;
    }
};

int main()
{
    Time t[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for (int i=0; i<10; i++)
        t[i].display();
}
```

Microsoft Visual Studio 调试控制台

```
1:0:0
2:0:0
3:0:0
4:0:0
5:0:0
6:0:0
7:0:0
8:0:0
9:0:0
10:0:0
```

10个元素的三个成员中
hour=1-10, 其它两个为0
调用一个参数的构造

```
#include <iostream>
using namespace std;
class Time {
private:
    int hour, minute, sec;
public:
    Time()
    {
        hour = 0;
        minute = 0;
        sec = 0;
    }
    Time(int h)
    {
        hour = h;
        minute = 0;
        sec = 0;
    }
    void display()
    {
        cout << hour << ':' << minute << ':' << sec << endl;
    }
};

int main()
{
    Time t[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for (int i=0; i<10; i++)
        t[i].display();
}
```

Microsoft Visual Studio 调试控制台

```
1:0:0
2:0:0
3:0:0
4:0:0
5:0:0
6:0:0
7:0:0
8:0:0
9:0:0
10:0:0
```

10个元素的三个成员中
hour=1-10, 其它两个为0
两个构造用一个参数的



§ 7. 结构体、类和对象

7.5. 对象数组

7.5.2. 定义对象时进行初始化

★ 若带参构造函数有带一个参数和多个参数共存(可以是带默认参数的构造函数), 则可用数组定义时初始化的方法进行, 每个数组元素只传一个参数

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

class Time {
private:
    int hour, minute, sec;
public:
    Time()
    { hour=0; minute=0; sec=0; }
    Time(int h)
    { hour=h; minute=0; sec=0; }
    Time(int h,int m)
    { hour=h; minute=m; sec=0; }
    void display()
    { cout << hour << ':' << minute << ':' << sec << endl; }
};

int main()
{
    Time t[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for (int i=0; i<10; i++)
        t[i].display();
}
```

Microsoft Visual Studio 调试控制台

```
1:0:0
2:0:0
3:0:0
4:0:0
5:0:0
6:0:0
7:0:0
8:0:0
9:0:0
10:0:0
```

无参
1、2
重载

10个元素的三个成员中
hour=1-10, 其它两个为0
多个构造用一个参数的

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

class Time {
private:
    int hour, minute, sec;
public:
    Time(int h=0,int m=0, int s=0)
    {
        hour = h;
        minute = m;
        sec = s;
    }
    void display()
    { cout << hour << ':' << minute << ':' << sec << endl; }
};

int main()
{
    Time t[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for (int i=0; i<10; i++)
        t[i].display();
}
```

Microsoft Visual Studio 调试控制台

```
1:0:0
2:0:0
3:0:0
4:0:0
5:0:0
6:0:0
7:0:0
8:0:0
9:0:0
10:0:0
```

带默认参数的构造函数
可带0/1/2/3个参数

10个元素的三个成员中
hour=1-10, 其它两个为0
调用带一个参数的构造



§ 7. 结构体、类和对象

7.5. 对象数组

7.5.2. 定义对象时进行初始化

★ 如果希望初始化时多于一个参数，则初始化时显式给出构造函数及实参表

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

class Time {
private:
    int hour;
    int minute;
    int sec;
public:
    Time(int h=0, int m=0, int s=0)
    {    hour=h; minute=m; sec=s;
    }
    void display()
    {    cout << hour << ':' << minute << ':' << sec << endl;
    }
};

int main()
{    Time t[10] = {Time(1, 2, 3), Time(4, 5), 6, 7, 8, 9, 10};
    for (int i=0; i<10; i++)
        t[i].display();
}
```

Microsoft Visual Studio 调试控制台

```
1:2:3    ... t[0] 三参构造
4:5:0    ... t[1] 两参构造
6:0:0    ... t[2] 一参构造
7:0:0
8:0:0
9:0:0
10:0:0   ... t[7] 零参构造
0:0:0
0:0:0
0:0:0
```

问：以下两种的差别在哪里？

Time t[10] = { {1, 2, 3}, {4, 5}, 6, 7, 8, 9, 10 };

Time t[10] = { (1, 2, 3), (4, 5), 6, 7, 8, 9, 10 };

★ 初始化的数量不能超过数组大小

★ 定义数组时可不定义大小，有初始化表决定

Time t[10] = {Time(1, 2, 3), Time(4, 5), 6, 7, 8, 9, 10};

不能比7小

Time t[] = {Time(1, 2, 3), Time(4, 5), 6, 7, 8, 9, 10};

自动为7



§ 7. 结构体、类和对象

7.6. 对象指针

7.6.1. 指向对象的指针

形式:

类名 *指针变量名

Time *t;

指针的赋值:

Time t1, *t;

Time t1, *t=&t1;

t = &t1;

定义后赋值语句赋值 定义时赋初值

Time t2[10], *t;

Time t2[10], *t=t2;

t = t2; t指向t2[0], t++则指向t2[1]

t++ ⇔ t+sizeof(time)

class Time换成 int

class Time换成 struct student, 方法相同



§ 7. 结构体、类和对象

7.6. 对象指针

7.6.1. 指向对象的指针

使用:

`class Time`换成 `struct student`, 方法相同,
但要注意成员访问限定, 外部仅限`public`

```
class Time {                                Time类定义
private:
    int minute;
    int sec;
public:
    int hour; //公有
    Time(int h=0, int m=0, int s=0);
    ~Time();
    void display();
};
```

`Time t1, *t=&t1;` 指向简单变量的指针

```
t : t1对象的地址
*t : t1对象
(*t).hour    ⇔ t->hour    ⇔ t1.hour;
(*t).display() ⇔ t->display() ⇔ t1.display()
```

```
Time t2[10], *t=t2;
```

指向数组变量的指针

```
t : t2数组的第[0]个对象的地址
*t : t2数组的第[0]个对象
```

```
(*t).hour    ⇔ t->hour    ⇔ t2[0].hour
(*t).display() ⇔ t->display() ⇔ t2[0].display()
```

```
t+3 : t2数组的第[3]个对象的地址
*(t+3) : t2数组的第[3]个对象
```

```
*(t+3).hour    ⇔ t[3].hour    ⇔ (t+3)->hour ⇔ t2[3].hour
*(t+3).display() ⇔ t[3].display() ⇔ (t+3)->display() ⇔ t2[3].display()
```



§ 7. 结构体、类和对象

7. 6. 对象指针

7. 6. 1. 指向对象的指针

7. 6. 2. 指向对象成员的指针

7. 6. 2. 1. 指向对象的数据成员的指针

定义：数据成员的基类型 *指针变量名

赋值：指针变量名 = 数据成员的地址

```
Time t1;  
int *p;  
p=&t1.hour;
```

class Time换成 struct student, 方法相同,
但要注意成员访问限定, 外部仅限public

使用：

```
*p ⇔ t1.hour;
```

★ 对象的数据成员必须是public

7. 6. 2. 2. 指向对象的成员函数的指针(后续课程内容, 略)



§ 7. 结构体、类和对象

7.6. 对象指针

7.6.1. 指向对象的指针

7.6.2. 指向对象成员的指针 (其中: 指向成员函数的指针 略)

7.6.3. this指针

含义: 指向当前被访问的成员函数所对应的对象的指针, 名称固定为this, 基类型为类名

```
void Time::display()
{
    cout << hour    << endl;
    cout << minute << endl;
    cout << sec     << endl;
}
```

相当于

```
void Time::display(const Time *this)
{
    cout << this->hour    << endl;
    cout << this->minute << endl;
    cout << this->sec     << endl;
} //编译会错, 只是含义上相当于!!!
```

```
Time t1, t2;
t1.display() 时, this指向t1
               ⇔ t1.display(&t1);
t2.display() 时, this指向t2
               ⇔ t2.display(&t2);
```

```
void Time::set(int h, int m, int s)
{
    hour   = h;
    minute = m;
    sec    = s;
}
```

相当于

```
void Time::set(const Time *this, int h, int m, int s)
{
    this->hour   = h;
    this->minute = m;
    this->sec    = s;
} //编译会错, 只是含义上相当于!!!
```

```
Time t1, t2;
t1.set(14,15,23) 时, this指向t1
                   ⇔ t1.set(&t1, 14, 15, 23);
t2.set(16,30,0)  时, this指向t2
                   ⇔ t2.set(&t2, 16, 30, 0);
```



§ 7. 结构体、类和对象

7.6. 对象指针

7.6.3. this指针

含义：指向当前被访问的成员函数所对应的对象的指针，名称固定为this，基类型为类名
使用：

★ 隐式使用，相当于通过对象调用成员函数时传入该对象的自身的地址

★ 也可以显式使用(但不能显式定义)

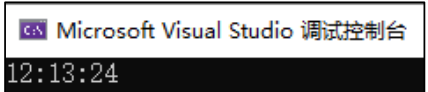
```
#include <iostream>
using namespace std;

class Time {
private:
    int hour, minute, sec;
public:
    Time(int h, int m, int s)
    {
        hour = h;
        minute = m;
        sec = s;
    }
    void display()
    {
        cout << this->hour << ':' << this->minute
              << ':' << this->sec << endl;
    }
};

int main()
{
    Time t1(12, 13, 24);
    t1.display();
}
```

隐式使用

显式使用



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

class Time {
private:
    int hour, minute, sec;
public:
    Time(int h, int m, int s)
    {
        hour = h;
        minute = m;
        sec = s;
    }
    void display(const Time *this)
    {
        cout << this->hour << ':' << this->minute
              << ':' << this->sec << endl;
    }
};

int main()
{
    Time t1(12, 13, 24);
    t1.display(&t1);
}
```

error C2143: 语法错误: 缺少“)” (在“this”的前面)
error C2143: 语法错误: 缺少“;” (在“this”的前面)
error C2059: 语法错误: “this”
error C2059: 语法错误: “)”
error C2334: “{”的前面有意外标记; 跳过明显的函数体

不能显式定义

//特殊约定, this不能显式, 其它名字可以
void display(const Time *that)
{
 cout << that->hour << ':' << this->minute
 << ':' << this->sec << endl;
}



§ 7. 结构体、类和对象

7.7. 对象的赋值与复制

7.7.1. 对象的赋值

含义：将一个对象的所有数据成员的值对应赋值给另一个**已存在**对象的数据成员

形式：类名 对象名1, 对象名2;

...

对象名1=对象名2; //执行语句的方式

```
Time t1(14, 15, 23), t2;
```

```
t2=t1;
```

★ 两个对象属于同一个类，且**不能**在定义时赋值

★ 系统**默认的赋值操作**是将右对象的全部数据成员的值对应赋给左对象的全部数据成员(**理解为整体内存拷贝，但不包括成员函数**)，在对象的数据成员**无动态内存申请时可直接使用**

★ 若对象数据成员是指针并涉及动态内存申请，则需要自行实现(**通过=运算符的重载实现，后续课程内容**)



§ 7. 结构体、类和对象

7.7. 对象的赋值与复制

7.7.2. 对象的复制

含义：建立一个**新**对象，其值与某个已有对象完全相同

使用：

类 对象名(已有对象名)

类 对象名=已有对象名

两种形式
本质一样

```
Time t1(14, 15, 23), t2(t1), t3=t1;
```

★ 与对象赋值的区别：定义语句/执行语句中

```
Time t1(14, 15, 23), t2, t3=t1; //复制，t3为新对象
```

```
t2 = t1; //赋值，t2为已有对象
```

★ 系统**默认的复制操作**是将已有对象的全部数据成员的值对应赋给新对象的全部数据成员 (**理解为整体内存拷贝，但不包括成员函数**)，在对象的数据成员**无动态内存申请时可直接使用**

★ 若对象数据成员是指针并涉及动态内存申请，则需要自行实现 (**通过重定义复制/拷贝构造函数来实现，后续课程内容**)



§ 7. 结构体、类和对象

7.8. 友元

7.8.1. 引入

当在外部访问对象时，private全部禁止，public全部允许，为使应用更灵活，引入友元(friend)的概念，允许友元访问private部分

★ 友元不是面向对象的概念，它破坏了数据的封装性，但方便使用，提高了运行效率

问题：在全局函数display(外部)中如何访问私有成员？

```
class Time {  
    private:  
        int hour;  
        int minute;  
        int sec;  
    public:  
        ...  
};  
  
void display(Time t)  
{  
    想访问 t.hour;  
}
```

方法1：通过公有函数间接访问

```
class Time {  
    private:  
        int hour;  
        int minute;  
        int sec;  
    public:  
        int get_hour() { return hour; }  
        void set_hour(int h) { hour = h; }  
        ...  
};  
  
void display(Time t)  
{  
    通过 t.get_hour() 读  
    通过 t.set_hour(12) 赋值  
}
```

缺点：当频繁调用时，效率较低

方法2：成员直接公有

```
class Time {  
    public:  
        int hour;  
        int minute;  
        int sec;  
    public:  
        ...  
};  
  
void display(Time t)  
{  
    直接 t.hour;  
}
```

缺点：所有外部函数都能访问
不仅局限于一个display()
失去了类的封装和隐蔽性



§ 7. 结构体、类和对象

7.8. 友元

7.8.1. 引入

可以成为类的友元的成分：

- ★ 全局函数
- ★ 其它类的成员函数
- ★ 其它类

友元的声明方式：

在类的声明中，相应要成为友元的函数/类前加friend关键字即可



§ 7. 结构体、类和对象

7.8. 友元

7.8.2. 声明全局函数为友元函数

```
class Time {  
    private:  
        int hour;  
        int minute;  
        int sec;  
        friend void display(Time &t);  
    public:  
        ...  
};
```

全局函数

```
void display(Time &t)  
{  
    cout << t.hour ; ✓  
}
```

```
void fun(Time &t)  
{  
    cout << t.hour ; ✗  
}
```

```
void main()  
{  
    Time t1;  
    display(t1);  
}
```

- ★ 不能直接写成员名，要通过对象来调用
因为不是成员函数，没有this指针
- ★ 声明友元的位置不限private/public



§ 7. 结构体、类和对象

7.8. 友元

7.8.3. 声明其它类的成员函数为友元函数

<pre>class Time; ← 在test中引用Time时， Time尚未定义， 因此要提前声明</pre>	<pre>class Time; ← 如无提前声明，test中 每个Time前加class也 可以（数量多时不方便）</pre>
<pre>class test { public: void display(Time &t); };</pre>	<pre>class test { public: void display(class Time &t); };</pre>
<pre>class Time { private: int hour; ... friend void test::display(Time &t); };</pre>	<pre>class Time { private: int hour; ... friend void test::display(Time &t); };</pre>
<pre>void test::display(Time &t) { cout << t.hour << ... << endl; }</pre>	<pre>void test::display(Time &t) { cout << t.hour << ... << endl; }</pre>



§ 7. 结构体、类和对象

7.8. 友元

7.8.3. 声明其它类的成员函数为友元函数

```
class Time;    在test中引用Time时，Time尚未定义因此要提前声明
```

```
class test {  
    public:  
    void display(Time &t);  
};
```

```
class Time {  
    private:    声明友元不限定private/public  
                但友元函数所在类要符合限定规则  
    int hour;  
    ...  
    friend void test::display(Time &t);  
};
```

```
void test::display(Time &t)  
{  
    cout << t.hour << ... << endl;  
}
```

//成员. 对象方式访问



§ 7. 结构体、类和对象

7.8. 友元

7.8.4. 友元类

★ 提前声明遵循刚才的原则

```
class test;//有提前声明
class Time {
    private:
        ...
        friend test;
};
class test {
    ...
};
```

test的所有成员函数
都可以访问Time的
私有成员

```
class test;//无提前声明
class Time {
    private:
        ...
        friend class test;
};
class test {
    ...
};
```

★ 友元是单向而不是双向的

本例中：Time中不能访问test的私有

★ 友元不可传递

```
class A {
    friend class B;
};
class B {
    friend class C;
};
class C {
    ...
    C不能访问A的私有成员
};
```

★ C++规定同类的不同对象互为友元

```
class Student {
    private:
        int num;
    public:
        void display();
};
void Student::display()
{ Student s;
  ...
  if (this->num > s.num) {...}
}
```

互为友元