



地球与空间科学系
DEPARTMENT OF EARTH AND SPACE SCIENCES

C程序设计基础

Introduction to C programming Lecture 14: Head files

张振国 zhangzg@sustech.edu.cn

南方科技大学/理学院/地球与空间科学系

Review on L13 Self-defined types

Struct, union, enumerate

Typedef , #define and #include

Structure in life



Student

- ☐ Name
- ☐ Age
- ☐ Gender
- ☐ ID
- ☐ Major
- ☐ Grade
- ☐ Birthday



Professor

- ☐ Name
- ☐ Age
- ☐ Gender
- ☐ Hat
- ☐ Paper
- ☐ Project
- ☐ Honour

Three types of structure

Struct
(结构体)



Used very often

Union
(共用体)

Useless

Enum
(枚举型)

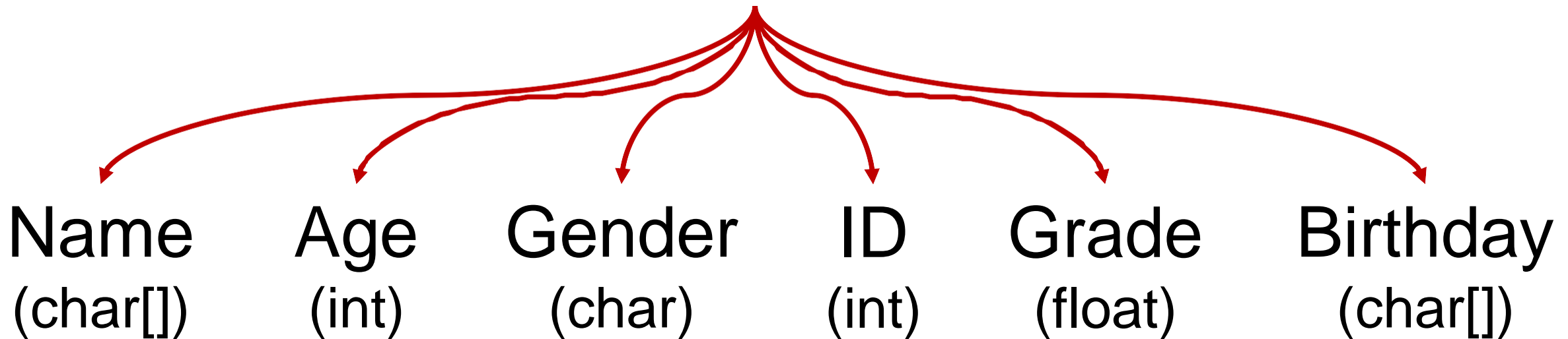


Used but not often

Struct

You cannot use **array** to group data with different types

Struct (结构体)



How to define struct?

Student

name
age
gender
ID
grade
birthday



struct **student**

{

char name[20];

int age;

char gender;

int ID;

int grade;

char birthday[50];

};

Struct name



Member list

How to define struct?


```
#include<stdio.h>
#include<string.h>
```

```
struct student
{
    char name[20];
    int age;
    char gender;
    int ID;
    int grade;
    char birthday[50];
};
```

define the struct data



```
int main(void)
{
    struct student student1;
    strcpy(student1.name, "Jack Chen");
    student1.age = 25;
    student1.gender = 'M';
    student1.ID = 123;
    student1.grade = 80;
    strcpy(student1.birthday, "2005-October-10");
}
```



(1) 先声明结构体类型再定义变量名

How to define struct?

```
#include<stdio.h>
#include<string.h>
```

```
struct student
{
    char name[20];
    int age;  char
gender;  int ID;
    int grade;
    char birthday[50];
} student1, student2, student3, student4;
```

```
int main(void)
{
    strcpy(student1.name, "Jack Chen");
    student1.age = 25;
    student1.gender = 'M';
    student1.ID = 123;
    student1.grade = 80;
    strcpy(student1.birthday, "2005-October-10");
}
```

Struct variable is defined globally!

(2) 在声明类型的同时定义变量

How to define struct?

(3) 直接定义结构体类型变量，可以不出现结构体名

```
#include<stdio.h>
#include<string.h>
struct
{
    char name[20];
    int age;
    char gender;
    int ID;
    int grade;
    char birthday[50];
} student1, student2, student3, student4;

int main(void)
{
    strcpy(student1.name, "Jack Chen");
    student1.age = 25;
    student1.gender = 'M';
    student1.ID = 123;
    student1.grade = 80;
    strcpy(student1.birthday, "2005-October-10");
}
```

注意:

- (1) 类型与变量是不同的概念，不要混同。只能对变量赋值、存取或运算，而不能对一个类型赋值、存取或运算。在编译时，对类型是不分配空间的，只对变量分配空间。
- (2) 对结构体中的成员（即“域”），可以单独使用，它的作用与地位相当于普通变量。
- (3) 成员也可以是一个结构体变量。
- (4) 成员名可以与程序中的变量名相同，二者不代表同一对象。

How to define struct?

Different structs can be used in one program



```
struct student
{
    char name[20];
    int age;
    char gender;
    int ID;
    int grade;
    char birthday[50];
};
```



```
struct teacher
{
    char name[20];
    int age;
    char gender;
    int hat;
    int paper;
    int project ;
    int honour;
};
```

```
int main(void)
{
    struct student student1 = {"Jack Chen", 25, 'M', 123, 80, "2005-October-10"};
    struct teacher teacher1 = {"Li Liang", 45, 'M', 1, 50, 5, 0};
}
```

Nested structs

```
#include<stdio.h>
```

```
struct birthday
```

```
{
```

```
    int year;
```

```
    int month;
```

```
    int day;
```

```
};
```

```
struct student
```

```
{
```

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

```
    int age;
```

```
    char gender;
```

```
    int ID;
```

```
    int grade;
```

```
    struct birthday birth;
```

```
};
```

```
int main(void)
```

```
{
```

```
    struct student student1;
```

```
    strcpy(student1.name, "Jack Chen");
```

```
    student1.age = 25;
```

```
    student1.gender = 'M';
```

```
    student1.ID = 123;
```

```
    student1.grade = 80;
```

```
    student1.birth.year = 2005;
```

```
    student1.birth.month = 10;
```

```
    student1.birth.day = 10;
```

```
    printf("student1 name = %s\n", student1.name);
```

```
    printf("student1 age = %d\n", student1.age);
```

```
    printf("student1 gender = %c\n", student1.gender);
```

```
    printf("student1 ID = %d\n", student1.ID);
```

```
    printf("student1 grade = %d\n", student1.grade);
```

```
    printf("student1 birthday = %d-%d-%d\n", student1.birth.year, student1.birth.month, student1.birth.day);
```

```
}
```

- 如果成员本身又属一个结构体类型, 则要用若干个成员运算符, 一级一级地找到最低的一级的成员。
- 只能对最低级的成员进行赋值或存取以及运算。

Array of structs

① declare struct array and initialize it separately

```
struct student
{
    char name[20];
    int ID;
    char gender;
};
```

```
int main(void)
{
    struct student stu[2];
    strcpy(stu[0].name, "Jack");
    stu[0].ID = 1;
    stu[0].gender = 'M';

    strcpy(stu[1].name, "Merry");
    stu[1].ID = 2;
    stu[1].gender = 'F';
}
```

Array of structs

② declare struct array and initialize it jointly

```
struct student
{
    char name[20];
    int ID;
    char gender;
};
```

```
int main(void)
{
    struct student stu[2] = {
        {"Jack", 1, 'M'}, {"Merry", 2, 'F'}
    };
}
```

Pointer to structs

```
#include<stdio.h>
```

```
struct student
{
    char name[4];
    int ID;
    char gender;
};
```

```
int main(void)
{
    struct student stu = {"Sam", 1, 'M'};

    printf("Address of stu: %x\n", &stu);
    printf("Address of nam: %x\n", &stu.name);
    printf("Address of ID: %x\n", &stu.ID);
    printf("Address of gender: %x\n", &stu.gender);
}
```

**You can check
the address of
struct!!!**

stu	affafb70
name	affafb70
ID	affafb74
gender	affafb78

Pointer to structs

```
#include<stdio.h>
```

```
struct student
{
    char name[4];
    int ID;
    char gender;
};
```

```
int main(void)
{
    struct student stu[2] = {{ "Sam", 1, 'M' }, { "Jen", 1, 'F' }};

    for (int i = 0; i < 2; i++) {
        printf("Address of stu: %x\n", &stu[i]);
        printf("Address of nam: %x\n", &stu[i].name);
        printf("Address of ID: %x\n", &stu[i].ID);
        printf("Address of gender: %x\n", &stu[i].gender);
    }
}
```

可以引用结构体变量成员的地址，也可以引用结构体变量的地址。

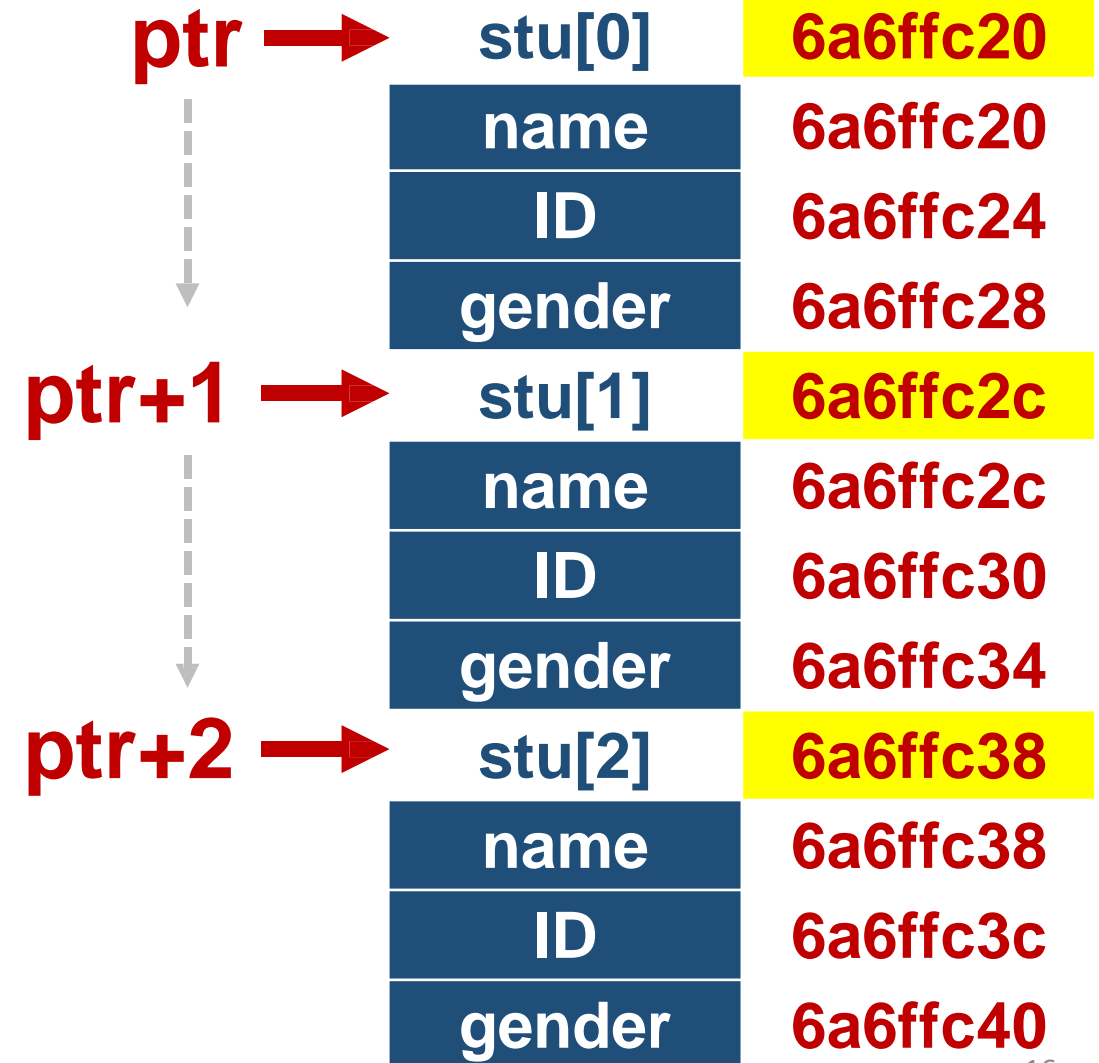
stu[0]	d3cff880
name	d3cff880
ID	d3cff884
gender	d3cff888
stu[1]	d3cff88c
name	d3cff88c
ID	d3cff890
gender	d3cff894

Pointer to structs

```
#include<stdio.h>
```

```
struct student
{
    char name[4];
    int ID;
    char gender;
};
```

```
int main(void)
{
    struct student stu[3] = {{ "Sam", 1, 'M' }, { "Jen", 2, 'F' }, { "Mik", 3, 'M' } };
    struct student* ptr = stu; //&stu[0]
    printf("address of prt = %x\n", ptr);
    printf("address of prt+1 = %x\n", ptr+1);
    printf("address of prt+2 = %x\n", ptr+2);
}
```



Pointer to structs

```
#include<stdio.h>
```

```
struct student
{
    char name[4];
    int ID;
    char gender;
};
```

```
int main(void)
{
    struct student stu[3] = {{ "Sam", 1, 'M' }, { "Jen", 2, 'F' },
                              { "Mik", 3, 'M' } };
    struct student* ptr = stu; //&stu[0];
```

```
    printf("stu 1 name = %s\n", (*ptr).name);
    printf("stu 2 name = %s\n", (*(ptr+1)).name);
    printf("stu 3 name = %s\n", (*(ptr+2)).name);
```

```
    printf("stu 1 name = %s\n", ptr -> name);
    printf("stu 2 name = %s\n", (ptr+1) -> name);
    printf("stu 3 name = %s\n", (ptr+2) -> name);
}
```

How to access members of struct using pointer?

①

(*ptr).name
(*ptr).ID
(*ptr).gender

②

ptr->name
ptr->ID
ptr->gender

Pointer to structs

以下3种形式等价：

- ① 结构体变量. 成员名
- ② (* p). 成员名
- ③ p ->成员名

其中->称为指向运算符。

Pointer to structs

分析以下几种运算：

- $p \rightarrow n$ 得到 p 指向的结构体变量中的成员 n 的值。
- $p \rightarrow n++$ 得到 p 指向的结构体变量中的成员 n 的值，用完该值后使它加 1。
- $++p \rightarrow n$ 得到 p 指向的结构体变量中的成员 n 的值加 1，然后再使用它。 (\rightarrow 优先级高于 $++$)

$(++p) \rightarrow n$
 $(p++) \rightarrow n$

?

Struct for functions

Struct as input parameters for function

Struct as output results of function

Struct as function input

```
#include<stdio.h>
struct student
{
    char name[5];
    int ID;
    char gender;
};
void input(struct student stu);
int main(void)
{
    struct student stu = {"Jack",1, 'M'};
    input(stu);
    printf("%s - %d - %c", stu.name, stu.ID, stu.gender);
}
void input(struct student stu)
{
    strcpy(stu.name, "Lily");
    stu.ID = 5;
    stu.gender = 'F';
}
```

What is value of stu?

Jack - 1 - M



or

Lily - 5 - F

- 函数间传递参数（结构struct）的值

Struct as function input

```
#include<stdio.h>
struct student
{
    char name[5];
    int ID;
    char gender;
};
void input(struct student *stu);
int main(void)
{
    struct student stu = {"Jack",1, 'M'};
    input(&stu);
    printf("%s - %d - %c", stu.name, stu.ID, stu.gender);
}
void input(struct student *stu)
{
    strcpy(stu->name, "Lily");
    stu->ID = 5;
    stu->gender = 'F';
}
```

What is value of stu?

Jack - 1 - M

or

Lily - 5 - F



➤ 函数间传递指针（结构struct）的地址

Struct as function input

```
#include<stdio.h>
#include<string.h>
struct student
{
    char name[5];
    int ID;
    char gender;
};
void input(struct student stu[]);
int main(void)
{
    struct student stu[2];
    input(stu);
    printf("%s - %d - %c\n", stu[0].name, stu[0].ID, stu[0].gender);
    printf("%s - %d - %c", stu[1].name, stu[1].ID, stu[1].gender);
}
void input(struct student stu[])
{
    strcpy(stu[0].name, "Lily"); stu[0].ID = 5; stu[0].gender= 'F';
    strcpy(stu[1].name, "Chen"); stu[1].ID = 7; stu[1].gender = 'M';
}
```

Pass the array of
structs to the function



Lily - 5 - F
Chen - 7 - M

➤ 函数间传递数组被当成指针

Struct as function input

Return a struct from
function to main

```
#include<stdio.h>
#include<string.h>
struct student
{
    char name[5];
    int ID;
    char gender;
};
```

```
struct student get();
```

```
int main(void)
{
    struct student stu = get();
    printf("%s - %d - %c", stu.name, stu.ID, stu.gender);
}
```

```
struct student get()
```

```
{
    struct student stu;
    strcpy(stu.name, "Lily"); stu.ID = 5; stu.gender = 'F';
    return stu;
}
```

➡ Lily - 5 - F

Struct as function output

Return struct array
from function to main

```
#include<stdio.h>
#include <string.h>
struct student
{
    char name[5];
    int ID;
    char gender;
};
struct student* get();
struct student stu[2];

int main(void)
{
    struct student *stu=get();
    printf("%s - %d - %c", stu[0].name, stu[0].ID,stu[0].gender);
    printf("%s - %d - %c",stu[1].name, stu[1].ID, stu[1].gender);
}
struct student *get()
{
    //struct student stu[2];
    strcpy(stu[0].name, "Lily"); stu[0].ID = 5; stu[0].gender= 'F';
    strcpy(stu[1].name, "Chen"); stu[1].ID = 7; stu[1].gender = 'M';
    return stu;
}
```



Lily - 5 - F
Chen - 1 - M

L13_struct_pointer_arr.cpp

Case study: car model

Case: if you want to buy a car after growing up?



```
#include<stdio.h>
```

```
struct Car {  
    char brand[50];  
    char model[50];  
    int price;  
};
```

```
int main(void)  
{
```

```
    struct Car car1 = {"Benz" , "MAYBACH" , 5000000};  
    struct Car car2 = {"Bentley" , "Flying Spur" , 4000000};  
    struct Car car3 = {"Maserati", "Quattroporte" , 3000000};
```

```
    printf("%s %s %d\n", car1.brand, car1.model, car1.price);  
    printf("%s %s %d\n", car2.brand, car2.model, car2.price);  
    printf("%s %s %d\n", car3.brand, car3.model, car3.price);
```

```
}
```

```
Benz MAYBACH 5000000  
Bentley Flying Spur 4000000  
Maserati Quattroporte 3000000
```

Case study: calculate GPA

```
#include<stdio.h>

struct course
{
    char ame[20];
    int score;
    float credit;
};

float cal_GPA(struct course course[], int num)
{
    float GPA = 0;
    float weight = 0;
    for (int i = 0; i < num; i++)
    {
        GPA += course[i].credit * course[i].score;
        weight += course[i].credit;
    }
    return GPA / weight;
}
```

Case: calculate your GPA!

科目	C程序基础	高等数学	线性代数
得分	98	85	90
学分	4	5	3.5

$$\text{GPA} = (98 \times 4 + 85 \times 5 + 90 \times 3.5) / (4 + 5 + 3.5) = 90.56$$

```
main()
{
    struct course course_[3]={ { "C程序设计基础",98,4
    },{ "高等数学",85,5 }, { " 线 性 代 数 ",90,3.5 }
    };
    printf("Your GPA is %f", cal_GPA(course_, 3));
}
```

Your GPA is 90.559998

Case study: transmit packet

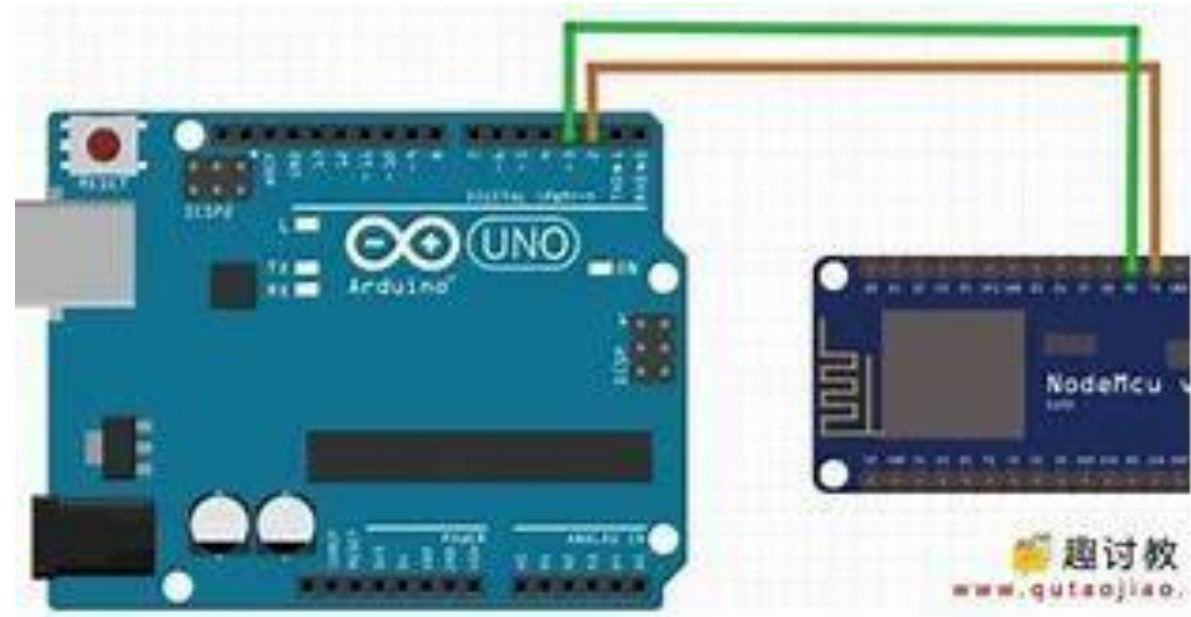
```
#include<stdio.h>

struct point
{
    float x;
    float y;
    float distance;
    float verify;
};

cal_verify(struct point *in) {
    in->verify = in->x + in->y + in->distance;
}

int main(void)
{
    struct point p = { 2.5, 2.5, 6, 0 };
    cal_verify(&p);
    printf("%f", p.verify);
}
```

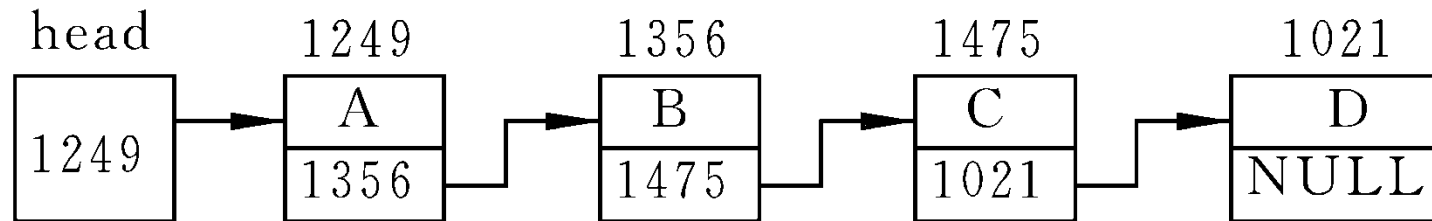
Case: verify if the transmission is correct?



Transmit 3 values (a, b, a+b), check if received values are correct

Linked List

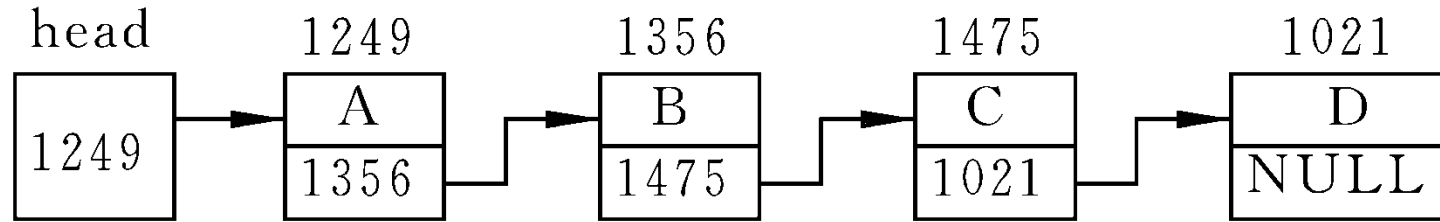
链表是一种常见的重要的数据结构, 是动态地进行存储分配的一种结构。可以根据需要开辟内存空间



头指针: 存放一个地址, 该地址指向一个元素
结点: 用户需要的实际数据和链接节点的指针

用户需要的实际数据
下一个节点的位置

Linked List

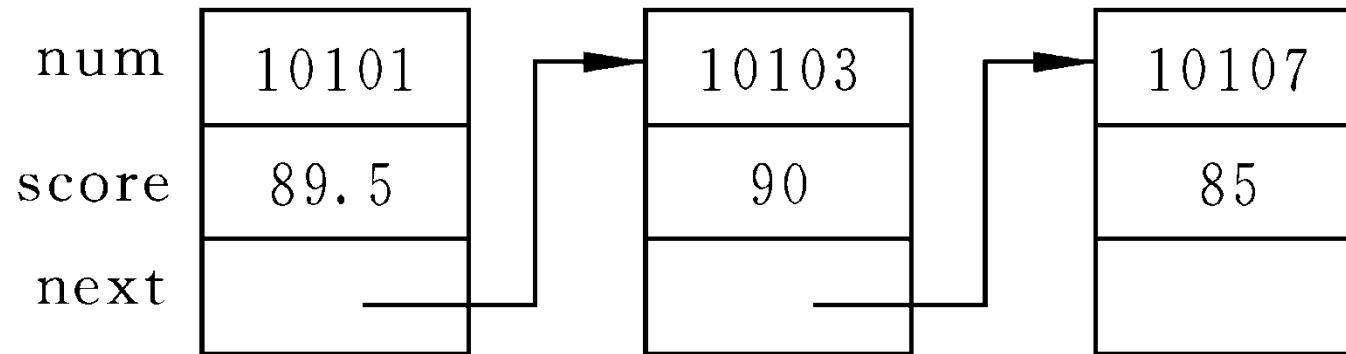


- 链表中各元素在内存中的地址可以是不连续的。要找某一元素，必须先找到上一个元素，根据它提供的下一元素地址才能找到下一个元素。如果不提供“头指针”(head)，则整个链表都无法访问。链表如同一条铁链一样，一环扣一环，中间是不能断开的。
- 链表这种数据结构，必须利用指针变量才能实现，即一个结点中应包含一个指针变量，用它存放下一结点的地址。

Linked List

一个结构体变量包含若干成员，这些成员可以是数值类型、字符类型、数组类型，也可以是指针类型。用指针类型成员来存放下一个结点的地址。例如：

```
struct student
{   int num;
    float score;
    struct student *next; };
```



Linked List

静态链表

```
#include <stdio.h>
#define NULL 0
struct student {
    long num; float score; struct student *next;
};

int main(void) {
    struct student a, b, c, *head, *p;
    a.num = 99101;      a.score = 89.5;
    b.num = 99103;      b.score = 90;
    c.num = 99107;      c.score = 85;
    head = &a;          a.next = &b;
    b.next = &c;          c.next = NULL;
    p = head;
    do {
        printf("%ld %5.1f\n", p->num, p->score);
        p = p->next;
    } while (p != NULL);
}
```

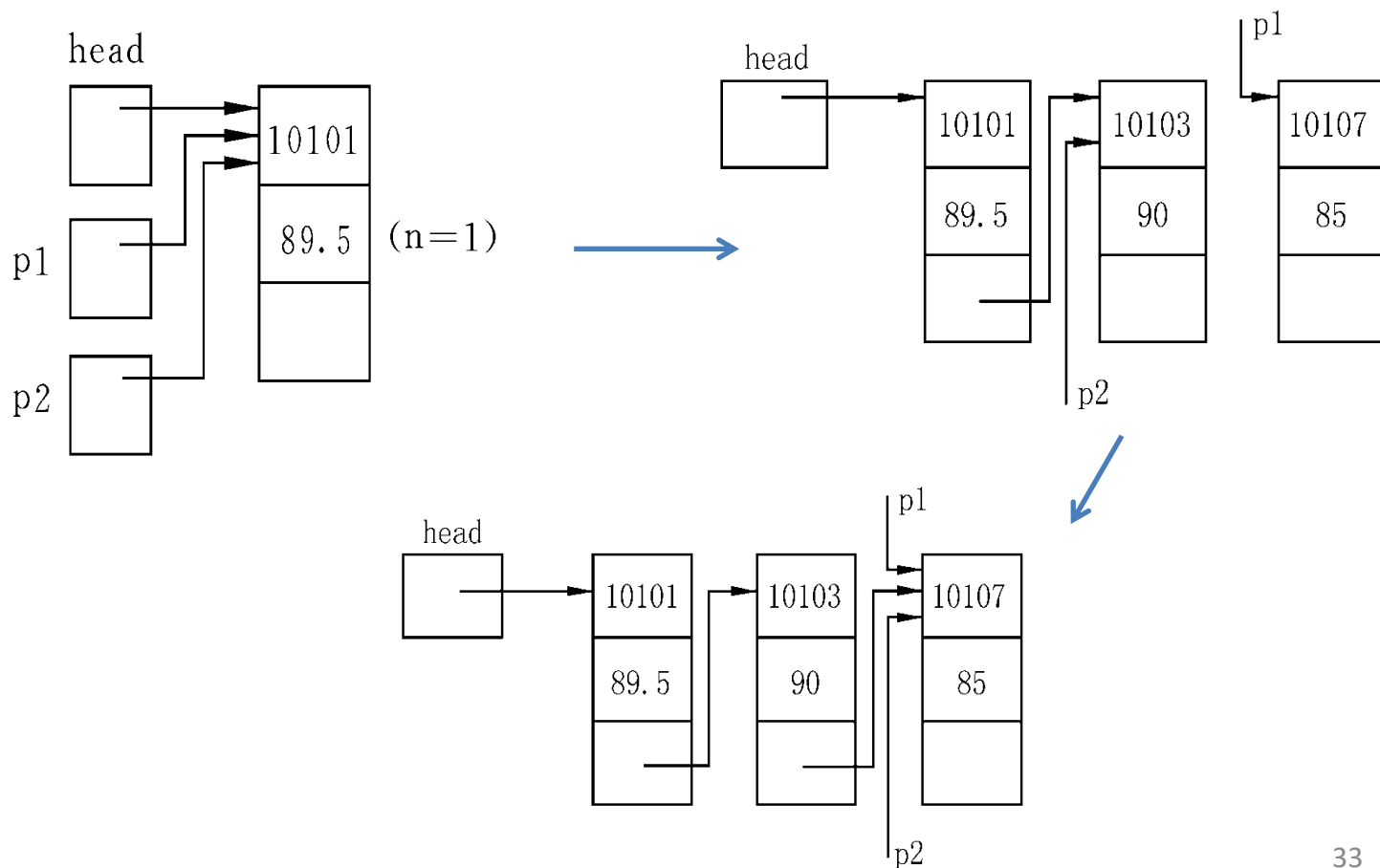
99101	89.5
99103	90.0
99107	85.0

Linked List

动态链表：所谓建立动态链表是指在程序执行过程中从无到有地建立起一个链表，即一个一个地开辟结点和输入各结点数据，并建立起前后相链的关系。

写一函数建立一个有**3**名学生数据的单向动态链表：

开辟一个新结点，并使p1、p2指向它	
读入一个学生数据给p1所指的结点	
head=NULL,n=0	
当读入的p1->num不是零	
n=n+1	
n等于1?	
真	假
head=p1 (把p1所指的结点作为第一个结点)	p2->next=p1 (把p1所指的结点连接到表尾)
p2=p1 (p2移到表尾)	
再开辟一个新结点，使p1指向它	
读入一个学生数据给p1所指结点	
表尾结点的指针变量置NULL	



Linked List

```
#include <stdio.h>
#include <malloc.h>
#define LEN sizeof(struct student)
struct student {
    long num; float score; struct student *next;
};
int n;
struct student *creat() {
    struct student *head,*p1, *p2;n = 0;
    p1 = p2 = ( struct student *) malloc(LEN);
    scanf("%ld,%f", &p1->num, &p1->score);
    head = NULL;
    while (p1->num != 0) {
        n = n + 1;
        if (n == 1)head = p1;else    p2->next = p1;
        p2 = p1;
        p1 = (struct student *)malloc(LEN);
        scanf("%ld,%f", &p1->num, &p1->score);
    }
    p2->next = NULL;return (head);}
```

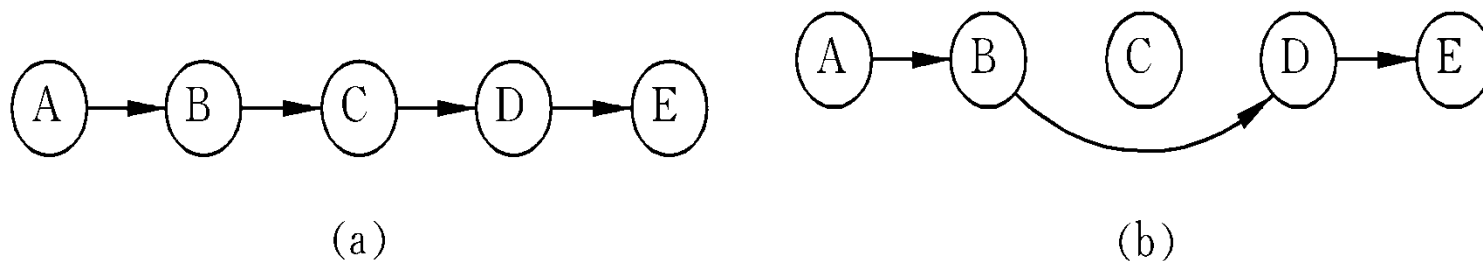
```
int main(void) {
    struct student *pt;
    pt = creat();
    do {
        printf("num:%ld
score: %5.1f\n", pt->num, pt-
>score);
        pt = pt->next;
    } while (pt != NULL);
}
```

```
1001, 67.5
1004, 87
1003, 99.5
0
num:1001    score: 67.5
num:1004    score: 87.0
num:1003    score: 99.5
```

Linked List

对链表的删除操作

从一个动态链表中删去一个结点，并不是真正从内存中把它抹掉，而是把它从链表中分离开来，只要撤销原来的链接关系即可。



Linked List

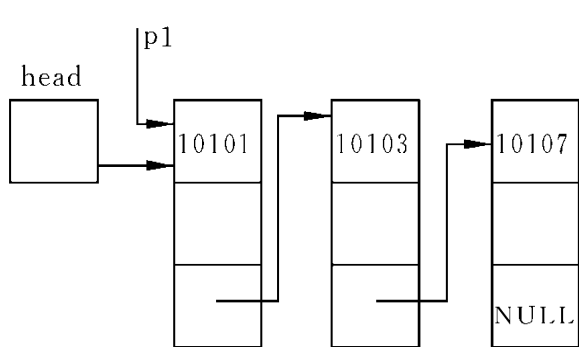
思路:

从p指向的第一个结点开始，检查该结点中的num值是否等于输入的要求删除的那个学号。如果相等就将该结点删除，如不相等，就将p后移一个结点，再如此进行下去，直到遇到表尾为止。

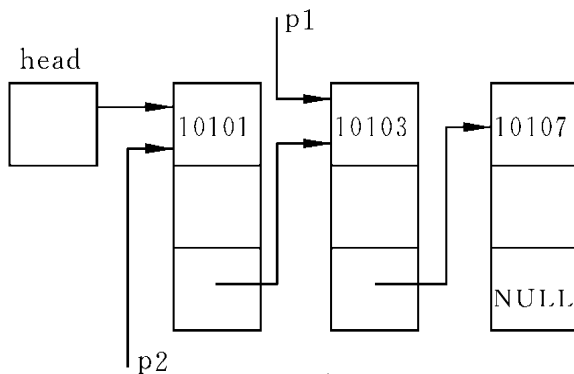
真	链表是一个空表		假
输出 “空表”	p1=head		
	当num≠p1->num以及p1所指的结点不是表尾结点		
	p2=p1 (p2后移一个位置) p1=p1->next (p1后移一个位置)		
	p1是要删除的结点		
	是	否	
	是	否	输出“找不到”的信息
head=p1->next (删除头结点)	p2->next=p1->next (删除一个结点)		

Linked List

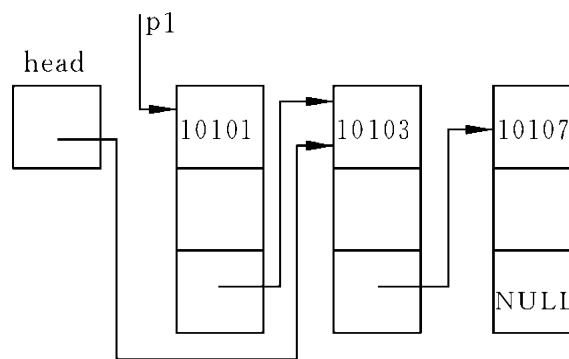
- ✓ 要删的是第一个结点（ $p1$ 的值等于 $head$ 的值, 图c），则应将 $p1 \rightarrow next$ 赋给 $head$ 。这时 $head$ 指向原来的第二个结点。第一个结点虽然仍存在，但它已与链表脱离，因为链表中没有一个结点或头指针指向它。虽然 $p1$ 还指向它，它仍指向第二个结点，但仍无济于事，现在链表的第一个结点是原来的第二个结点，原来第一个结点已“丢失”，即不再是链表的一部分了。
- ✓ 如果要删除的不是第一个结点，则将 $p1 \rightarrow next$ 赋给 $p2 \rightarrow next$ (图d)。 $p2 \rightarrow next$ 原来指向 $p1$ 指向的结点（图中第二个结点），现在 $p2 \rightarrow next$ 改为指向 $p1 \rightarrow next$ 所指向的结点（图中第三个结点）。 $p1$ 所指向的结点不再是链表的一部分。



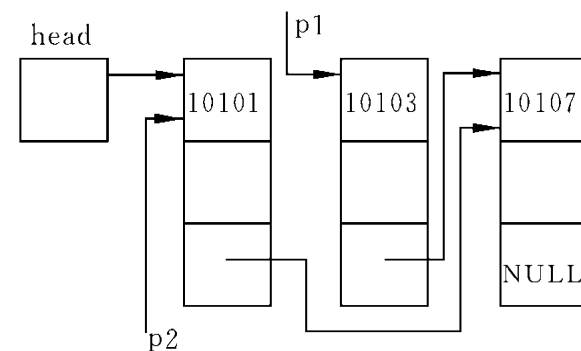
(a)



(b)



(c)



(d)

Linked List

```
struct student *del(struct student *head, int num) {
    struct student *p1, *p2;
    if (head == NULL) {
        printf("\nlist null!\n");
        return (head);
    }
    p1 = head;
    while (num != p1->num && p1->next != NULL) {
        p2 = p1;
        p1 = p1->next;
    }
    if (num == p1->num) {
        if (p1 == head)
            head = p1->next;
        else
            p2->next = p1->next;
        printf("delete:%ld\n", num); n = n - 1;
    } else
        printf("%ld not been found!\n", num);
    return (head);
}
```

input records :

10101, 89
10102, 85
10103, 98
0, 0

Now, These 3 records are:

num:10101 score: 89.0
num:10102 score: 85.0
num:10103 score: 98.0
input the deleted number:10102
delete:10102

Now, These 2 records are:

num:10101 score: 89.0
num:10103 score: 98.0

Linked List

对链表的插入操作

对链表的插入是指将一个结点插入到一个已有的链表中。

为了能做到正确插入，必须解决两个问题：

- ① 怎样找到插入的位置；
- ② 怎样实现插入。

Linked List

case: 若已建立了学生链表，结点是按照其成员项（num）的值由小到大顺序排列，今要插入一个新生的结点，要求按学号的顺序插入。

思路:

先用指针变量p0指向待插入的结点，p1指向第一个结点。

将p0->num与p1->num相比较，如果p0->num > p1-> num ，则待插入的结点不应插在p1所指的结点之前。此时将p1后移，并使p2指向刚才p1所指的结点。

p1 = head, p0 = stud		
原来的链表是空表		
是	否	
将p0所指的结点作为惟一结点	当p0->num > p1->num 以及 p1所指的不是表尾结点	
	p2指向p1位置 p1向后移一个结点	
	p0->num ≤ p1->num	
	真	假
	p1指向头结点	
是	否	p1->next = p0 p0->next = NULL (插到表尾之后)
	是	
head = p0	p2->next = p0	
p0->next = p1	p0->next = p1	
(插到表头之前)	(插到表中间)	
n = n + 1		

Linked List

```
struct student *insert(struct student *head, struct student *stud) {
    struct student *p0, *p1, *p2;
    p1 = head;      p0 = stud;
    if (head == NULL) {
        head = p0;
        p0->next = NULL;
    } else {
        while ((p0->num > p1->num) && (p1->next != NULL)) {
            p2 = p1; p1 = p1->next;
        }
        if (p0->num <= p1->num) {
            if (head == p1) head = p0;
            else p2->next = p0;
            p0->next = p1;
        } else {p1->next = p0; p0->next = NULL;} //p1->==NULL
    }
    n = n + 1;
    return (head);
}
```

input records:

10101, 80
10103, 95
10105, 92
0

Now, These 3 records are:

10101 80.0
10103 95.0
10105 92.0

input the inserted record:

10102, 90

Now, These 4 records are:

10101 80.0
10102 90.0
10103 95.0
10105 92.0

When to use struct?

When you want to group different types of data in a single unit!

Union

Union defines a new data type that allows using variables with different types at the same memory location!

```
union [union tag]
{
    type variable;
    type variable;
    ...
};
```

Union (共用体)

Union

```
struct student
{
    char name[20];
    int ID;
    int grade;
};
```

```
union student
{
    char name[20];
    int ID;
    int grade;
};
```

sizeof(student) = 20

20 bytes = max(20, 4, 4)

Store at the same memory location!!!

Union versus struct

```
#include <stdio.h>
```

```
union Data
{
    int i;
    float f;
    char str[20];
};
```

```
data.i : 1917853763
data.f : 
4122360580327794860452
759994368.000000
data.str : C
Programming
```

```
int main(void)
{
    union Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy(data.str, "C Programming");
    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);
}
```

```
#include <stdio.h>
```

```
struct Data
{
    int i;
    float f;
    char str[20];
};
```

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

```
int main(void)
{
    struct Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy(data.str, "C Programming");
    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);
}
```

Union versus struct

```
#include <stdio.h>
```

```
union Data
{
    int i;
    float f;
    char str[20];
};
```

```
main( )
{
    union Data data;
    data.i = 10;
    ➔ printf( "data.i : %d\n", data.i);
    data.f = 220.5;
    ➔ printf( "data.f : %f\n", data.f);
    strcpy(data.str, "C Programming");
    ➔ printf( "data.str : %s\n", data.str);
}
```

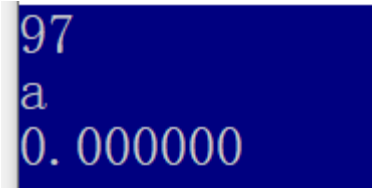
data.i: 10
data.f: 220.500000
data.str: C Programming



Union

- 同一个内存段可以用来存放几种不同类型的成员，但在每一瞬时只能存放其中一种，而不是同时存放几种。

```
union Date {  
    int i;  
    char ch;  
    float f;  
} a;  
  
void main() {  
    a.i = 97;  
    printf("%d\n", a.i);  
    printf("%c\n", a.ch);  
    printf("%f", a.f);  
}
```



```
97  
a  
0.000000
```

- 共用体变量中起作用的成员是最后一次存放的成员，在存入一个新的成员后原有的成员就失去作用。
- 共用体变量的地址和它的各成员的地址都是同一地址。
- 不能把共用体变量作为函数参数，也不能使函数带回共用体变量，但可以使用指向共用体变量的指针。C99允许用共用体变量作为函数参数

Union

- 不能对共用体变量名赋值，也不能企图引用变量名来得到一个值，又不能在定义共用体变量时对它初始化。

a = 1;

× 不能对共用体变量赋值，赋给谁？

m = a;

× 企图引用共用体变量名以得到一个值赋给整型变量m

b = a;

✓ a和b是同类型的共用体变量，合法

- 共用体类型可以出现在结构体类型定义中，也可以定义共用体数组。反之，结构体也可以出现在共用体类型定义中，数组也可以作为共用体的成员。

case

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

num	name	sex	job	class(班) position(职务)
101	Li	f	s	501
102	Wang	m	t	prof

```
struct {
    int num;
    char name[10];
    char sex;
    char job;
    union {
        int banji;
        char position[10];
    } category;
} person[2];
```

case

```
int main(void) {
    int i;
    for (i = 0; i < 2; i++) {
        scanf("%d %s %c %c", &person[i].num, &person[i].name, &person[i].sex,
&person[i].job);
        if (person[i].job == 's')    scanf("%d", &person[i].category.banji);
        else if (person[i].job == 't') scanf("%s", person[i].category.position);
        else    printf("Input error!");
    }
    printf("\n");
    printf("No. name sex job class/position\n");
    for (i = 0; i < 2; i++) {
        if (person[i].job == 's')
            printf(" % 6d % 10s % 3c % 3c % 6d\n", person[i].num,
                person[i].name, person[i].sex, person[i].job, person[i].category.banji);
        else
            printf(" % 6d % 10s % 3c % 3c % 6s\n", person[i].num, person[i].name,
                person[i].sex, person[i].job, person[i].category.position);
    }
}
```

101 Li f s 501

102 Wang m t prof

No.	name	sex	job	class/position
-----	------	-----	-----	----------------

101	Li	f	s	501
-----	----	---	---	-----

102	Wang	m	t	prof
-----	------	---	---	------

When to use union?

**Do NOT use union,
use struct as
much as you
can!!!**

Enumerate

Enum is a user defined data type in C, assign names to integer constants, for a program easy to read and maintain.

```
enum [union tag]
{
    variable;
    variable;
    ...
};
```

← **All integers by default!**

Enum (枚举型)

Enumerate

```
enum week { Mon, Tue, Wed, Thu, Fri, Sat, Sun}  
          0   1   2   3   4   5   6
```

默认从0开始

```
enum week { Mon=1, Tue, Wed, Thu, Fri, Sat, Sun}  
          1   2   3   4   5   6   7
```

```
enum week day;  
day = Mon;
```

Enumerate

Enum assigns names to integer constants

```
#include<stdio.h>

enum week {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

int main(void)
{
    enum week day;

    for (day = Mon; day <=Sun; day++)
    {
        printf("%d\n", day);
    }
}
```

c支持(.c)

c++不支持(.cpp)

```
#include<stdio.h>

enum week {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

int main(void)
{
    for (int i = Mon; i <=Sun; i++)
    {
        printf("%d\n", i);
    }
}
```

day = (enum week) (day + 1)

Enumerate

Enum assigns names to integer constants

```
#include<stdio.h>

enum Year {Jan, Feb, Mar, Apr, May, June,
July, Aug, Sept, Oct, Nov, Dec};

int main(void)
{
    enum Year year;

    for (year = Jan; year <= Dec; year++)
    {
        printf("%d\n", year);
    }
}
```

```
#include<stdio.h>

enum Year {Jan, Feb, Mar, Apr, May, June,
July, Aug, Sept, Oct, Nov, Dec};

int main(void)
{
    for (int i = Jan; i <= Dec; i++)
    {
        printf("%d\n", i);
    }
}
```

Case study: check weekday

```
#include <stdio.h>

int main(void) {
    enum week { Mon = 1, Tues, Wed, Thurs,
               Fri, Sat, Sun } day;
    scanf_s("%d", &day);
    switch (day) {
        case Mon: puts("Monday"); break;
        case Tues: puts("Tuesday"); break;
        case Wed: puts("Wednesday"); break;
        case Thurs: puts("Thursday"); break;
        case Fri: puts("Friday"); break;
        case Sat: puts("Saturday"); break;
        case Sun: puts("Sunday"); break;
        default: puts("Error!");
    }
}
```

Case: input a number, check the weekday

4
Thursday

6
Saturday

15
Error!

Case study: check season

```
#include <stdio.h>

enum Season{spring, summer, fall, winter};

int main(void)
{
    enum Season now = spring;

    if (now < summer)
    {
        printf("It's spring now");
    }
}
```

Case: is it spring now?

It's spring now

When to use enum?

When you want to assign a sequential of names with integers



Jan, Feb, Mar, Apr, May, June, July, Aug, Sept, Oct, Nov, Dec



Mon, Tue, Wed, Thu, Fri, Sat, Sun



Jack, Lily, Tom, John, Wim, Kevin, Henk

Summary of three data types

Struct
(结构体)



Used very often

Union
(共用体)

Very useless

Enum
(枚举型)



Used but not often

Content

1. Struct, union, enumerate

2. Typedef , #define and #include

Typedef

Typedef allows you to create a type with new name.

```
Typedef type name;
```

```
typedef unsigned char BYTE;
```

```
typedef struct Books  
{  
    char title[50];  
    char author[50];  
    int book_id;  
} Book;
```

Use typedef for variable

Variable

```
unsigned char var1;  
unsigned char var2;  
unsigned char var3;
```

Variable with typedef

```
typedef unsigned char BYTE;
```

```
BYTE var1;  
BYTE var2;  
BYTE var3;
```

A new type



Use typedef for struct

Struct

```
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};
```

```
struct Books book1;
struct Books book2;
struct Books book3;
```

Struct with typedef

```
typedef struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} Book;
```

```
Book book1;
Book book2;
Book book3;
```

A new type



Usage of typedef

- ① 先按定义变量的方法写出定义体（如：int i）。
- ② 将变量名换成新类型名（例如：将i换成COUNT）。
- ③ 在最前面加typedef（例如：typedef int COUNT）。
- ④ 然后可以用新类型名去定义变量。

习惯上，常把用typedef声明的类型名的第一个字母用大写表示，以便与系统提供的标准类型标识符相区别

声明NUM为整型数组类型：typedef int NUM[100]

声明STRING为字符指针类型：typedef char *STRING;

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

typedef

- 用typedef可以声明各种类型名，但不能用来定义变量。
- 用typedef只是对已经存在的类型增加一个类型名，而没有创造新的类型。
- 当不同源文件中用到同一类型数据时，常用typedef声明一些数据类型，把它们单独放在一个文件中，然后在需要用到它们的文件中用#include命令把它们包含进来。
- 使用typedef有利于程序的通用与移植。

preprocessing

- ANSI C标准规定可以在C源程序中加入一些“预处理命令”，以改进程序设计环境，提高编程效率。
- 这些预处理命令是由ANSI C统一规定的，但是它不是C语言本身的组成部分，不能直接对它们进行编译（因为编译程序不能识别它们）。必须在对程序进行通常的编译之前，先对程序中这些特殊的命令进行“预处理”。
- 经过预处理后程序可由编译程序对预处理后的源程序进行通常的编译处理，得到可供执行的目标代码。

preprocessing

- C语言与其他高级语言的一个重要区别是可以使用预处理命令和具有预处理的功能。

C 提供的预处理功能主要有以下三种：

1. 宏定义
2. 文件包含
3. 条件编译

这些功能分别用宏定义命令、文件包含命令、条件编译命令来实现。为了与一般 C 语句相区别，这些命令以符号“#”开头。例如：

`#define`

`#include`

#define

#define allows you to define **macros** (constant values) as pre-processors before compilation.

```
#define name value
```

Macro (宏) definitions must be constant,
there is no symbols like = and ;

在预编译时将宏名替换成字符串的过程称为“宏展开”。
#define是宏定义命令。

#define

#define sets macros for numbers, strings or expressions

numbers {
#define ONE 1
#define PI 3.14

strings {
#define CHAR 'a'
#define NAME "SUSTech"

Expr. {
#define MIN(a, b) (a < b ? a : b)
#define SUM(a, b, c) (a + b + c)

#define

Use const global variable

```
#include<stdio.h>
```

```
const int ONE = 1;  
const float PI = 3.14;  
const char CHAR = 'a';  
const char NAME[] = "SUSTech";
```

```
main()  
{  
    printf("%d\n", ONE);  
    printf("%f\n", PI);  
    printf("%c\n", CHAR);  
    printf("%s\n", NAME);  
}
```

Use #define macro

```
#include<stdio.h>
```

```
#define ONE 1  
#define PI 3.14  
#define CHAR 'a'  
#define NAME "SUSTech"
```

```
main()  
{  
    printf("%d\n", ONE);  
    printf("%f\n", PI);  
    printf("%c\n", CHAR);  
    printf("%s\n", NAME);  
}
```

#define

```
#include<stdio.h>
```

```
#define MIN(a, b) (a < b ? a : b)
#define SUM(a, b, c) (a + b + c)
#define POW(x) (x * x)
```

**Use #define for
macro expressions**

```
main()
{
    printf("%d\n", MIN(10, 100));
    printf("%d\n", SUM(10, 20, 30));
    printf("%f\n", SUM(2.3, 0.8, -3.5));
    printf("%d\n", POW(5));
}
```

#define

- (1) 宏名一般习惯用**大写字母**表示，以便与变量名相区别。但这并非规定，也可用小写字母。
- (2) 使用宏名代替一个字符串，可以减少程序中重复书写某些字符串的工作量。
- (3) 宏定义是用宏名代替一个字符串，只作简单置换，不作正确性检查。只有在编译已被宏展开后的源程序时才会发现语法错误并报错。
- (4) 宏定义不是 C 语句，**不必在行末加分号**。如果加了分号则会连分号一起进行置换。
- (5) **#define**命令出现在程序中函数的外面，宏名的有效范围为定义命令之后到本源文件结束。通常，**#define**命令写在文件开头，函数之前，作为文件一部分，在此文件范围内有效。

#define

(6) 可以用 `#undef` 命令终止宏定义的作用域。

例如：

```
#define G 9.8
    void main()
    {
        ...
    }
    #undef G
    f1()
    {
        ...
    }
```



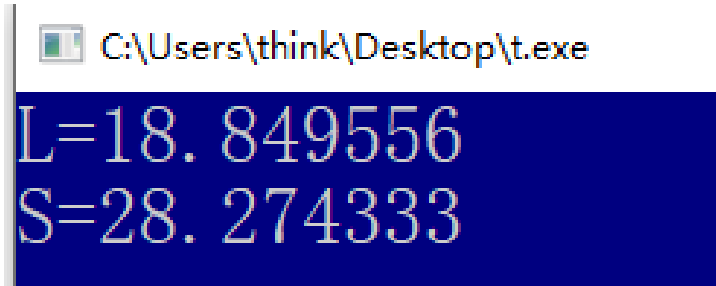
G的有效范围

在f1函数中，G不再代表9.8。这样可以灵活控制宏定义的作用范围。

#define

(7) 在进行宏定义时，可以引用已定义的宏名，可以层层置换。

```
#include <stdio.h>
#define R 3.0
#define PI 3.1415926
#define L 2*PI*R
#define S PI*R*R
void main()
{
    printf("L=%f\nS=%f\n", L, S);
}
```

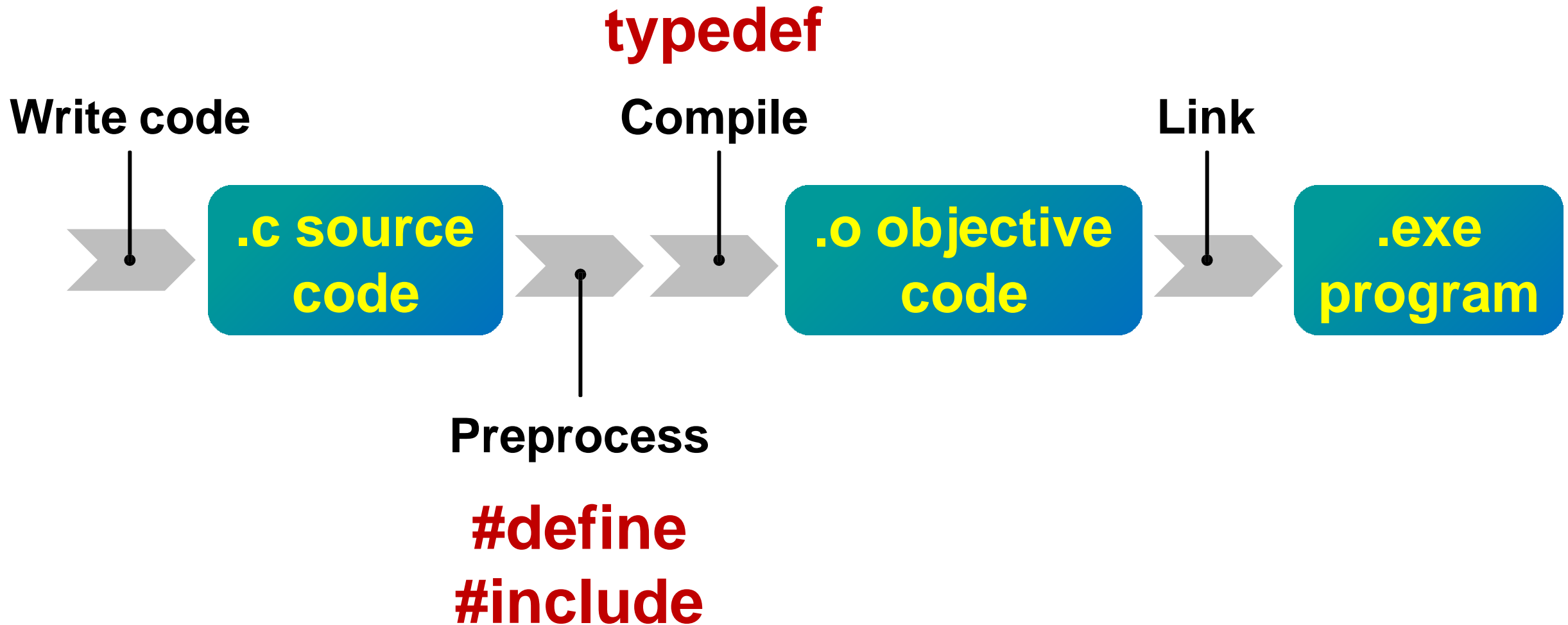


```
C:\Users\think\Desktop\t.exe
L=18.849556
S=28.274333
```

(8) 对程序中用双撇号括起来的字符串内的字符，即使与宏名相同，也不进行置换。

(9) 宏定义是专门用于预处理命令的一个专用名词，它与定义变量的含义不同，只作字符替换，不分配内存空间。

typedef versus #define



typedef versus #define

typedef

- Processed by **compiler**, actual definition of a new type
- Give symbolic names to types
- With scope rules. If defined inside function, only visible to the function

#define

- Processed by **preprocessor**, copy-paste the definition in place
- Define alias for values (#define ONE 1)
- No scope rules, it replaces all occurrences, visible everywhere

typedef versus #define

C source code

```
#include<stdio.h>
```

```
#define SQUARE(X) ((X)*(X))
```

```
#define PR(x) printf("The result is  
%d.\n",x)
```

```
int main()  
{
```

```
    int z = 5;
```

```
    PR(SQUARE(z));
```

```
    PR(SQUARE(z + 2));
```

```
    PR(100 / SQUARE(2));
```

```
    return 0;
```

```
}
```



Preprocessed code

```
int main()  
{
```

```
    int z = 5;
```

```
    printf("The result is %d.\n",z*z);
```

```
    printf("The result is %d.\n", \  
           (z +2)*(z + 2));
```

```
    printf("The result is%d.\n",100/(2*2));
```

```
    return 0;
```

```
}
```

Preprocessor

C preprocessor (#) is a text substitution tool that instructs compiler to do required pre-processing before actual compilation.

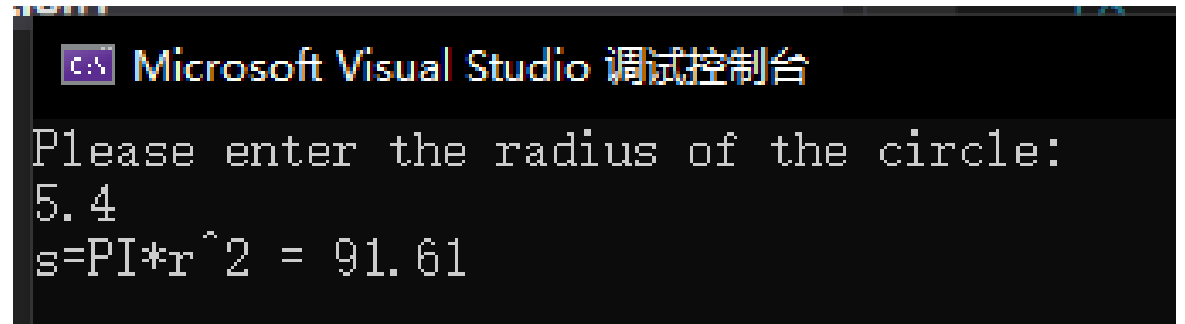
<code>#define</code>	Substitutes a preprocessor macro
<code>#include</code>	Inserts a particular header from another file
<code>#undef</code>	Undefines a preprocessor macro
<code>#ifdef</code>	Returns true if this macro is defined
<code>#ifndef</code>	Returns true if this macro is not defined
<code>#if</code>	Tests if a compile time condition is true
<code>#else</code>	The alternative for <code>#if</code>
<code>#elif</code>	<code>#else</code> an <code>#if</code> in one statement
<code>#endif</code>	Ends preprocessor conditional
<code>#error</code>	Prints error message on stderr
<code>#pragma</code>	Issues special commands to the compiler, using a standardized method

Case study: area of circle

```
# include <stdio.h>
# define PI 3.14159

int main(void)
{
    float r, s;
    printf("Please enter the radius of
the circle:\n");
    scanf_s("%f", &r);
    s = PI * r * r;
    printf("s=PI*r^2 = %.2f\n", s);
}
```

Case: calculate the area of a circle.



```
Microsoft Visual Studio 调试控制台
Please enter the radius of the circle:
5.4
s=PI*r^2 = 91.61
```

Case study: basic statistics

Case: calculate the maximum, minimum, mean of two values

```
#include<stdio.h>
#define MAX(X,Y) (X>Y ? X:Y)
#define MIN(X,Y) (X<Y ? X:Y)
#define MEAN(X,Y) ((X+Y)/2)

int main(void)
{
    float x, y;
    printf("Enter two numbers:\n");
    scanf_s("%f %f", &x, &y);
    printf("The mean is
%f.\n",MEAN(x,y));
    printf("The max number is %f.\n",MAX(x,
y));
    printf("The min number is %f.\n",
MIN(x, y));
    return 0;
}
```

```
Enter two numbers:
3 5
The mean is 4.000000.
The max number is 5.000000.
The min number is 3.000000.
```

- (1) 带参数的宏展开只是将语句中的宏名后面括号内的实参字符串代替 **#define** 命令行中的形参。
- (2) 在宏定义时，在宏名与带参数的括弧之间不应加空格，否则将空格以后的字符都作为替代字符串的一部分。

#define VS function

带参数的宏和函数的区别:

- (1) 函数调用时，先求出实参表达式的值，然后代入形参。而使用带参的宏只是进行简单的字符替换。
- (2) 函数调用是在程序运行时处理的，为形参分配临时的内存单元。而宏展开则是在编译前进行的，在展开时并不分配内存单元，不进行值的传递处理，也没有“返回值”的概念。
- (3) 对函数中的实参和形参类型要求一致。而宏名无类型，它的参数也无类型，只是一个符号代表，展开时代入指定的字符串即可。宏定义时，字符串可以是任何类型的数据。

#define VS function

带参数的宏和函数的区别:

- (4) 使用宏次数多时，宏展开后源程序长，因为每展开一次都使程序增长，而函数调用不会使源程序变长。
- (5) 宏替换不占运行时间，只占编译时间。而函数调用则占运行时间（分配单元、保留现场、值传递、返回）。
- (6) 调用函数只可得到一个返回值，而用宏可以设法得到几个结果。

```
#include <stdio.h>
#define PI 3.1415926
#define CIRCLE(R,L,S,V)
L=2*PI*R;S=PI*R*R;V=4.0/3.0*PI*R*R*R

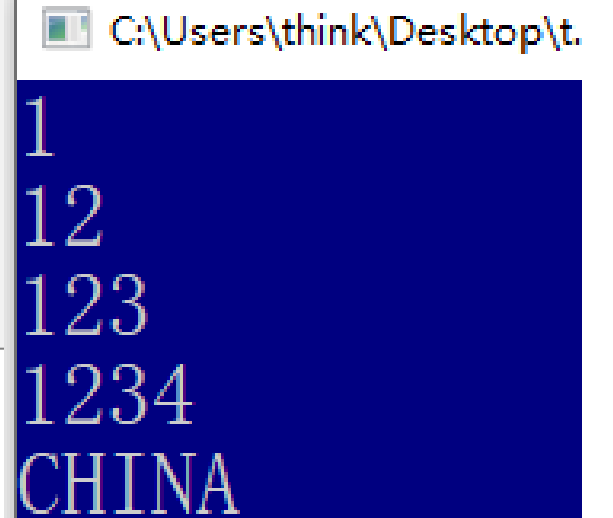
void main() {
    float r, l, s, v;
    scanf("%f", &r);
    CIRCLE(r, l, s, v);
    printf("r=%6.2f,l=%6.2f,s=%6.2f,v=
%6.2f\n", r, l, s, v);}
```

```
4
r= 4.00, l= 25.13, s= 50.27, v=268.08
```

Case study

```
#include <stdio.h>
#define PR printf
#define NL "\n"
#define D "%d"
#define D1 D NL
#define D2 D D NL
#define D3 D D D NL
#define D4 D D D D NL
#define S "%s"
```

```
void main()
{ int a, b, c, d;
  char string[]="CHINA";
  a=1;b=2;c=3;d=4;
  PR(D1, a);
  PR(D2, a, b);
  PR(D3, a, b, c);
  PR(D4, a, b, c, d);
  PR(S, string);
}
```



```
C:\Users\think\Desktop\t.
1
12
123
1234
CHINA
```

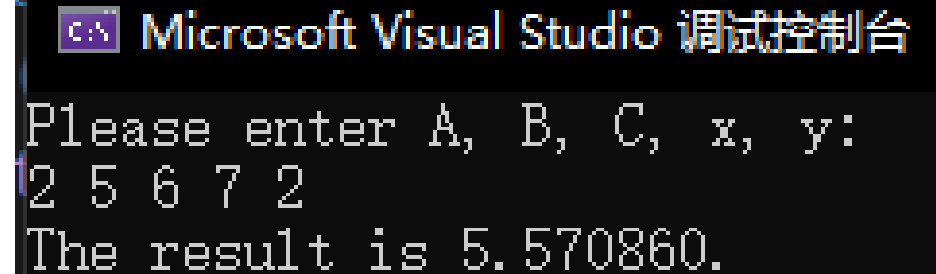
Case study: point to line distance

```
#include<stdio.h>
#include<math.h>

#define PRINT(x) printf("The result is %f.\n",x)
#define ABS(X) (X < 0 ? -X:X)

float DST(int A, int B, int C, int x, int y)
{
    float distance = (A * x + B * y + C) / sqrt(A *
A + B * B);
    return ABS(distance);
}
int main() {
    int A, B, C, x, y;
    printf("Please enter A, B, C, x, y:\n");
    scanf_s("%d %d %d %d %d", &A, &B, &C, &x, &y);
    float dis = DST(A, B, C, x, y);
    PRINT(dis);
    return 0;
}
```

Case: calculate the distance from point to line



```
Microsoft Visual Studio 调试控制台
Please enter A, B, C, x, y:
2 5 6 7 2
The result is 5.570860.
```

Summary

- **Struct** can be used to define a new data type for grouping data with different types.
- Struct is very useful and has been commonly used. It can be used with **arrays, pointers, and functions**.
- **Union** is not useful (only need know). **Enum** can be used to assign a sequence of names with integers.
- **Typedef** can define a new type of data by combining existing ones (short int, struct), **#define** can define macros for pre-processing.
- Time to define your own data types!

Content

1. Head files

2. Multi-threading (不做要求!)

What is head file?

A header file is **a file with extension .h**, which contains C function declarations and macro definitions that can be used by different source files.

This is c file



run.c

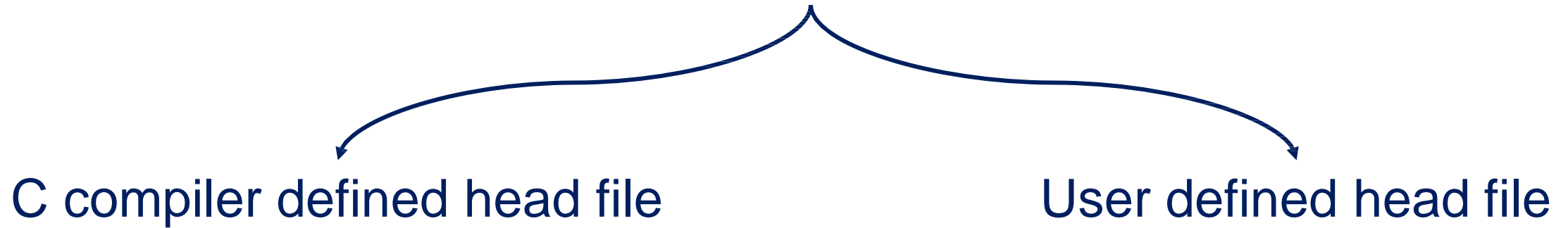
This is head file



stdio.h

What is head file?

Two types of head file



stdio.h

myFun.h

Avoid using names of C compiler
defined head files!

Why using head file?

- When your program grows large, it's impossible to keep all functions in one file.
- You can move parts of a program (functions) to separate files, and link them by head file.

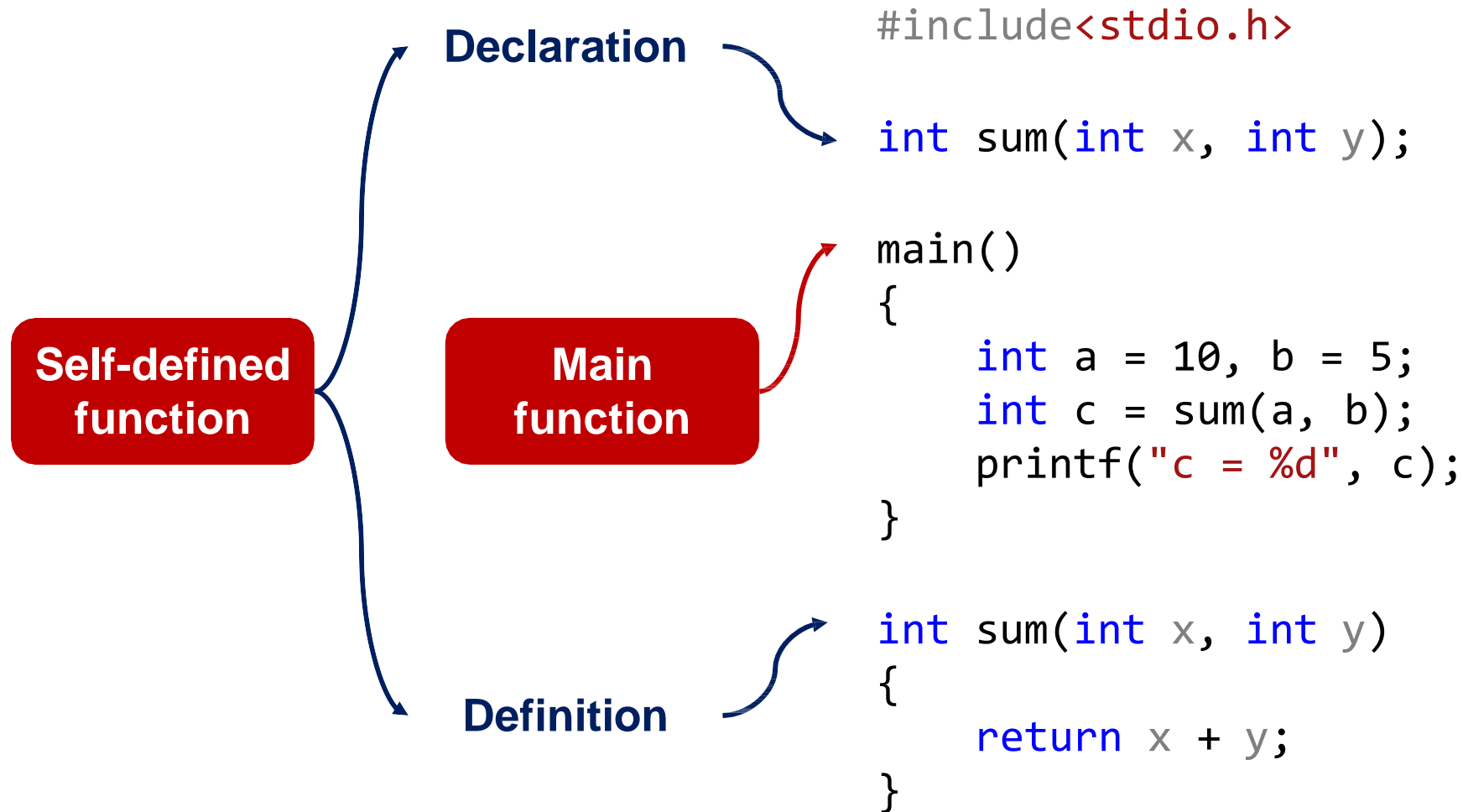
You already used C
Compile defined head file



```
#include<stdio.h>
```

```
main()  
{  
    printf("Hello World!");  
}
```

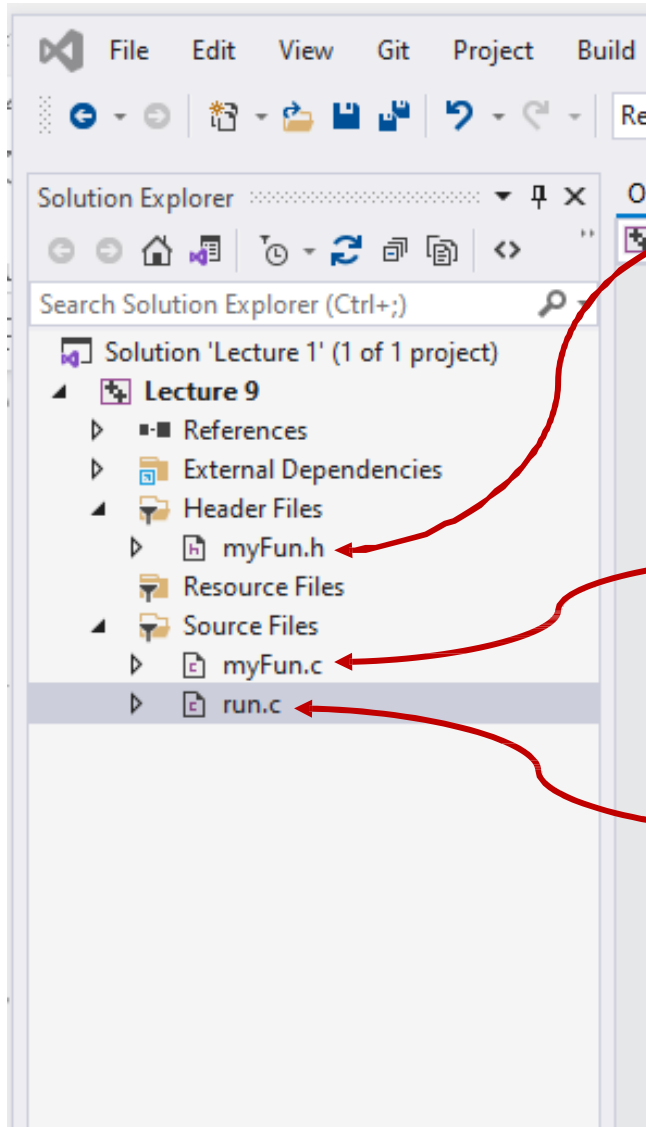
An example of function



Microsoft Visual Studio Debug Console

```
c = 15
C:\Users\wer...
e\Lecture 1\
To automatic
```

How to use head files?



1. 右键单击“Header Files”，添加一个新的头文件 myFun.h，声明方程

2. 右键单击“Source Files”，添加一个新的C文件 myFun.c，实现方程

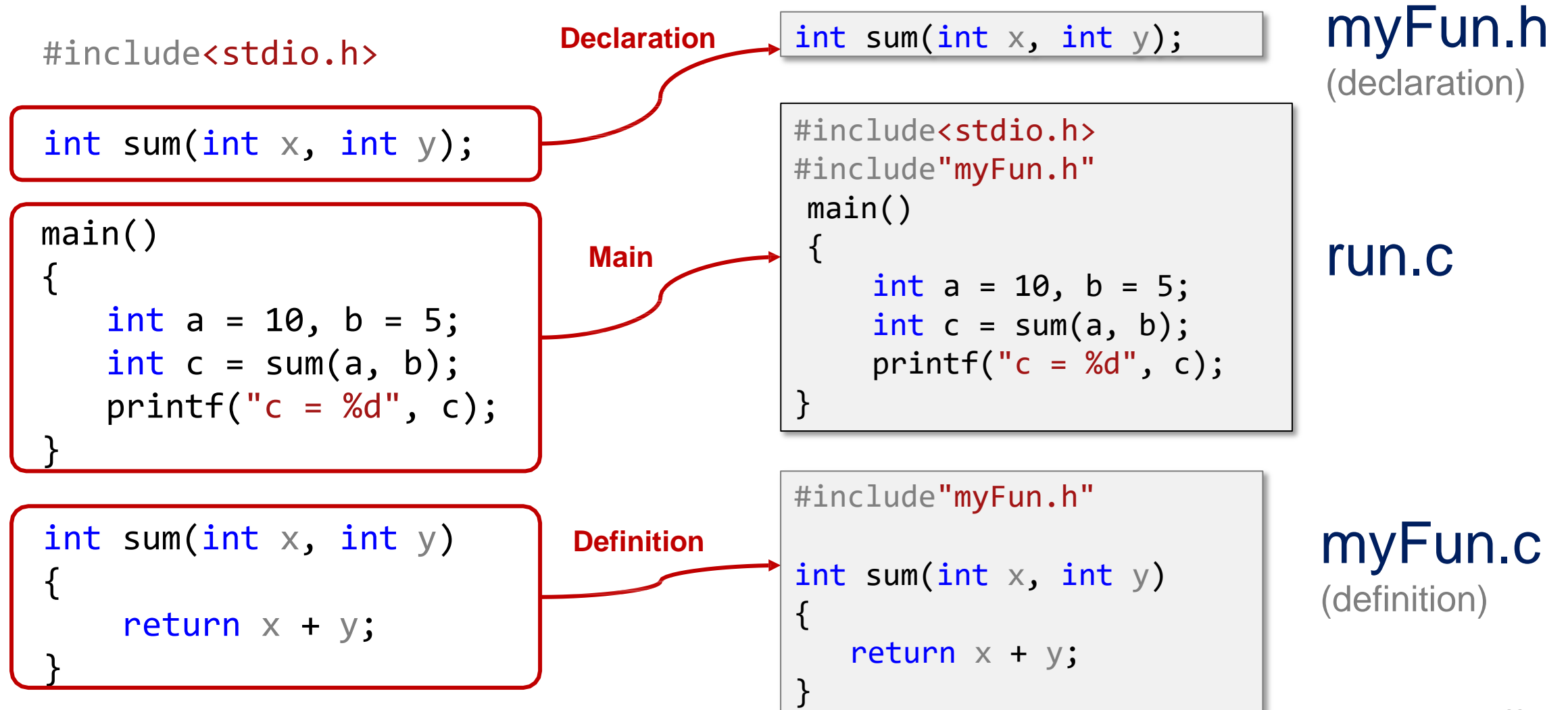
3. 在run.c里添加代码 `#include "myFun.c"`，包含方程声明

How to use head files?

方程声明放入头文件 → myFun. h

方程定义放入c文件 → myFun. c

How to use head files?



How to use head files?

Include the declaration of
sum() by including myfun.h

```
#include<stdio.h>
#include"myFun.h"

main()
{
    int a = 10, b = 5;
    int c = sum(a, b);
    printf("c = %d", c);
}
```

```
int sum(int x, int y);
```

myFun.h
(declaration)

```
#include"myFun.h"

int sum(int x, int y)
{
    return x + y;
}
```

myFun.c
(definition)

How to use head files?

Use < > to include
C compiled head file



```
#include<stdio.h>
```

Use " " to include
user defined head file

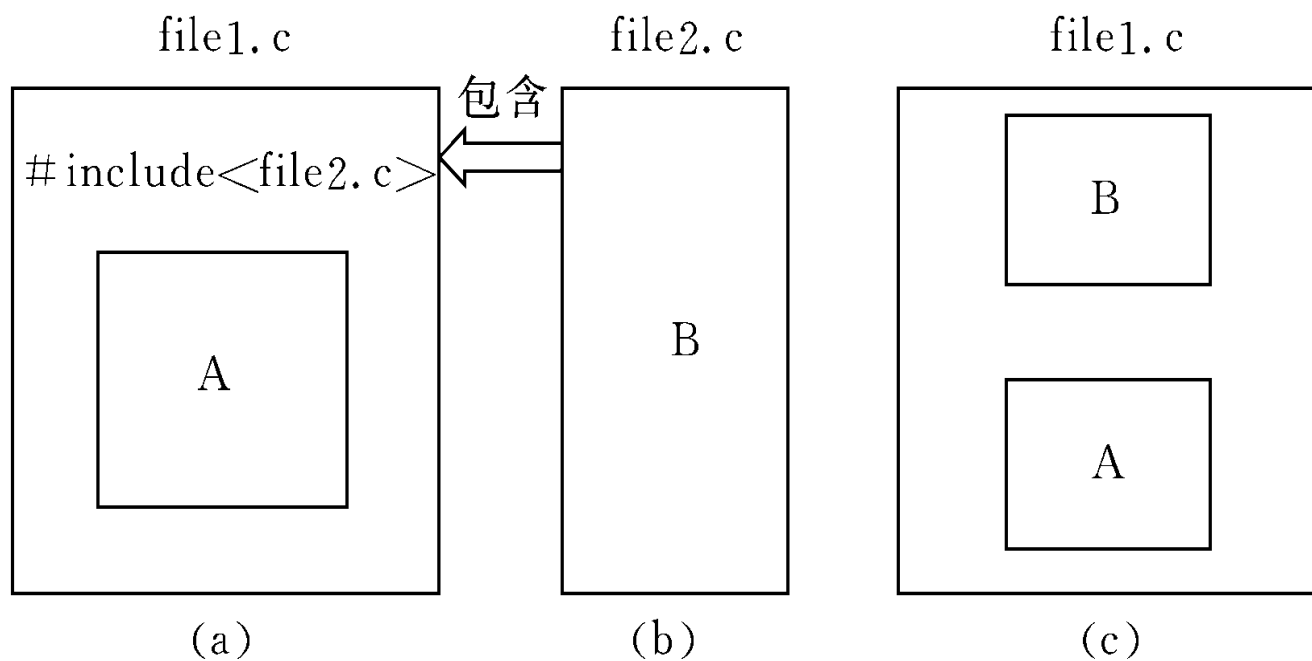


```
#include"myFun.h"
```

#include

- “文件包含”处理是指一个源文件可以将另外一个源文件的全部内容包含进来。C语言提供了#include命令用来实现“文件包含”的操作。
- 其一般形式为：

#include “文件名”
或 **#include <文件名>**



#include

注意:

在编译时并不是分别对两个文件分别进行编译，然后再将它们的目标程序连接的，而是在经过编译预处理后将头文件format.h包含到主文件中，得到一个新的源程序，然后对这个文件进行编译，得到一个目标（.obj）文件。被包含的文件成为新的源文件的一部分，而单独生成目标文件。

(1) 将格式宏做成头文件

format.h

```
#include <stdio.h>
#define PR printf
#define NL "\n"
#define D "%d"
#define D1 D NL
#define D2 D D NL
#define D3 D D D NL
#define D4 D D D D NL
#define S "%s"
```

(2) 主文件file1.c

```
#include <stdio.h>
#include "format.h"
int main(void)
{ int a, b, c, d;
  char string[]="CHINA";
  a=1;b=2;c=3;d=4;
  PR(D1, a);
  PR(D2, a, b);
  PR(D3, a, b, c);
  PR(D4, a, b, c, d);
  PR(S, string);
}
```

#include

- (1) 一个#include命令只能指定一个被包含文件，如果要包含 n 个文件，要用 n 个#include命令。
- (2) 如果文件 1 包含文件 2，而在文件 2 中要用到文件 3 的内容，则可在文件 1 中用两个include命令分别包含文件 2 和文件 3，而且文件 3 应出现在文件 2 之前，即在file1.c中定义。
- (3) 在一个被包含文件中又可以包含另一个被包含文件，即文件包含是可以嵌套的。
- (4) 在#include命令中，文件名可以用双撇号或尖括号括起来。
- (5) 被包含文件（file2.h）与其所在的文件（即用#include命令的源文件file2.c），在预编译后已成为同一个文件（而不是两个文件）。因此，如果file2.h中有全局静态变量，它也在file1.h文件中有效，不必用extern声明。

Use head file to declare more functions

```
#include<stdio.h>
```

```
int sum(int x, int y);  
int mul(int x, int y);
```

```
main()  
{  
    int a = 10, b = 5;  
    int c = sum(a, b);  
    int d = mul(a, b);  
}
```

```
int sum(int x, int y)  
{  
    return x + y;  
}  
int mul(int x, int y)  
{  
    return x * y;  
}
```

Declaration

```
int sum(int x, int y);  
int mul(int x, int y);
```

myFun.h
(declaration)

Main

```
#include<stdio.h>  
#include"myFun.h"  
  
main()  
{  
    int a = 10, b = 5;  
    int c = sum(a, b);  
    int d = mul(a, b);  
}
```

run.c

Definition

```
#include"myFun.h"  
int sum(int x, int y)  
{  
    return x + y;  
}  
int mul(int x, int y)  
{  
    return x * y;  
}
```

myFun.c
(definition)

Use head file to declare more functions

```
#include<stdio.h>
```

```
int sum(int x, int y);  
int mul(int x, int y);
```

Declaration

myFun1.h

```
int sum(int x, int y);
```

myFun2.h

```
int mul(int x, int y);
```

```
main()  
{  
    int a = 10, b = 5;  
    int c = sum(a, b);  
    int d = mul(a, b);  
}
```

Main

```
#include<stdio.h>  
#include"myFun1.h"  
#include"myFun2.h"  
  
main()  
{  
    int a = 10, b = 5;  
    int c = sum(a, b);  
    int d = mul(a, b);  
}
```

run.c

```
int sum(int x, int y)  
{  
    return x + y;  
}  
int mul(int x, int y)  
{  
    return x * y;  
}
```

Definition

myFun1.c

```
#include"myFun1.h"  
  
int sum(int x, int y)  
{  
    return x + y;  
}
```

myFun2.c

```
#include"myFun2.h"  
  
int mul(int x, int y)  
{  
    return x * y;  
}
```

Use head file to declare more functions

myFun1.h

```
int sum(int x, int y);
```

myFun1.c

```
#include "myFun1.h"

int sum(int x, int y)
{
    return x + y;
}
```

Library 1
函数库

myFun2.h

```
int mul(int x, int y);
```

myFun2.c

```
#include "myFun2.h"

int mul(int x, int y)
{
    return x * y;
}
```

Library 2
函数库

run.c

```
#include <stdio.h>
#include "myFun1.h"
#include "myFun2.h"

main()
{
    int a = 10, b = 5;
    int c = sum(a, b);
    int d = mul(a, b);
}
```

Main function

Use head file to declare more functions

```
#include<stdio.h>
```

```
int sum(int x, int y);  
int mul(int x, int y);
```

Declaration

myFun1.h

```
int sum(int x, int y);
```

myFun2.h

```
int mul(int x, int y);
```

```
main()  
{  
    int a = 10, b = 5;  
    int c = sum(a, b);  
    int d = mul(a, b);  
}
```

Main

```
#include<stdio.h>  
#include"myFun.h"  
  
main()  
{  
    int a = 10, b = 5;  
    int c = sum(a, b);  
    int d = mul(a, b);  
}
```

run.c

```
int sum(int x, int y)  
{  
    return x + y;  
}  
int mul(int x, int y)  
{  
    return x * y;  
}
```

Definition

myFun1.c

```
#include"myFun1.h"  
  
int sum(int x, int y)  
{  
    return x + y;  
}
```

myFun2.c

```
#include"myFun1.h"  
#include"myFun2.h"  
  
int mul(int x, int y)  
{  
    return sum(x + y)*y;  
}
```

Use head file to declare more functions

myFun1.h

```
int sum(int x, int y);
```

myFun1.c

```
#include "myFun1.h"

int sum(int x, int y)
{
    return x + y;
}
```

Library 1

myFun2.h

```
int mul(int x, int y);
```

myFun2.c

```
#include "myFun1.h"
#include "myFun2.h"

int mul(int x, int y)
{
    return sum(x, y) * y;
}
```

Library 2

dependency

run.c

```
#include <stdio.h>
#include "myFun1.h"
#include "myFun2.h"

main()
{
    int a = 10, b = 5;
    int c = sum(a, b);
    int d = mul(a, b);
}
```

Main function

Step-by-step to use head files

- ① write head file **xxx.h** (declare functions)

```
int sum(int x, int y);
```

myFun.h

- ② write c file **xxx.c** (include xxx.h, define functions)

```
#include "myFun.h"

int sum(int x, int y)
{
    return x + y;
}
```

myFun.c

- ③ write **run.c** (include xxx.h, call functions)

```
#include <stdio.h>
#include "myFun.h"
main()
{
    int a = 10, b = 5;
    int c = sum(a, b);
}
```

run.c

Case study of using head file

Write a function that put elements in the array in the reverse order and return the pointer of the array (e.g. 1,2,3,4,5,6 -> 6,5,4,3,2,1)

run.c

```
#include <stdio.h>
#include "test.h"

main()
{
    int array[] =
    { 0,10,20,30,40,50,60,70 };
    inv_array(array, 8);
    for (int i = 0; i < 8; i++)
        printf("%d ", array[i]);
}
```

```
#include <stdio.h>
void swap(int* a, int* b);
void inv_array(int* a, int size);
```

test.h

```
#include "test.h"
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
void inv_array(int* a, int size)
{
    for (int i = 0; i < size / 2; i++)
        swap(&a[i], &a[size - i - 1]);
}
```

test.c

Microsoft Visual Studio 调试控制台

70 60 50 40 30 20 10 0

You can use head files of C

```
#include<stdio.h>
```

```
#include<iostream>
```

```
#include<math.h>
```

```
#include<string.h>
```

```
...
```

```
#include<pthread.h> 线程
```