



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 15: Review

张振国 zhangzg@sustech.edu.cn

南方科技大学/理学院/地球与空间科学系

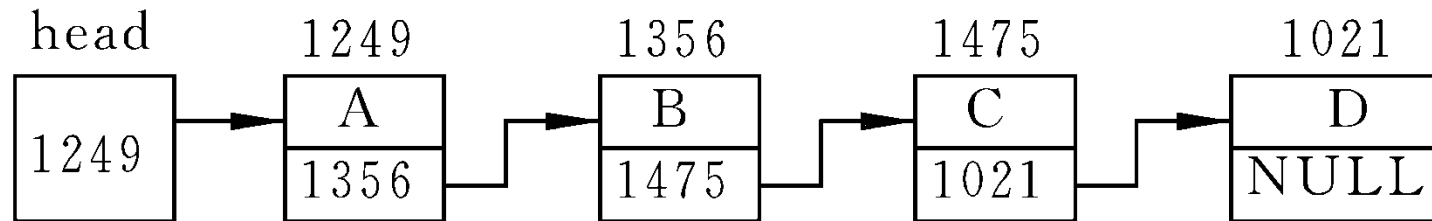
Review on L14 Head files

Struct, union, enumerate

Typedef , #define and #include

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



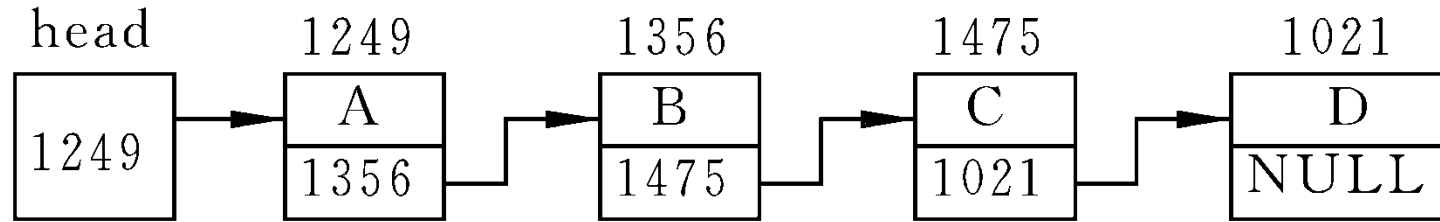
头指针: 存放一个地址, 该地址指向一个元素

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



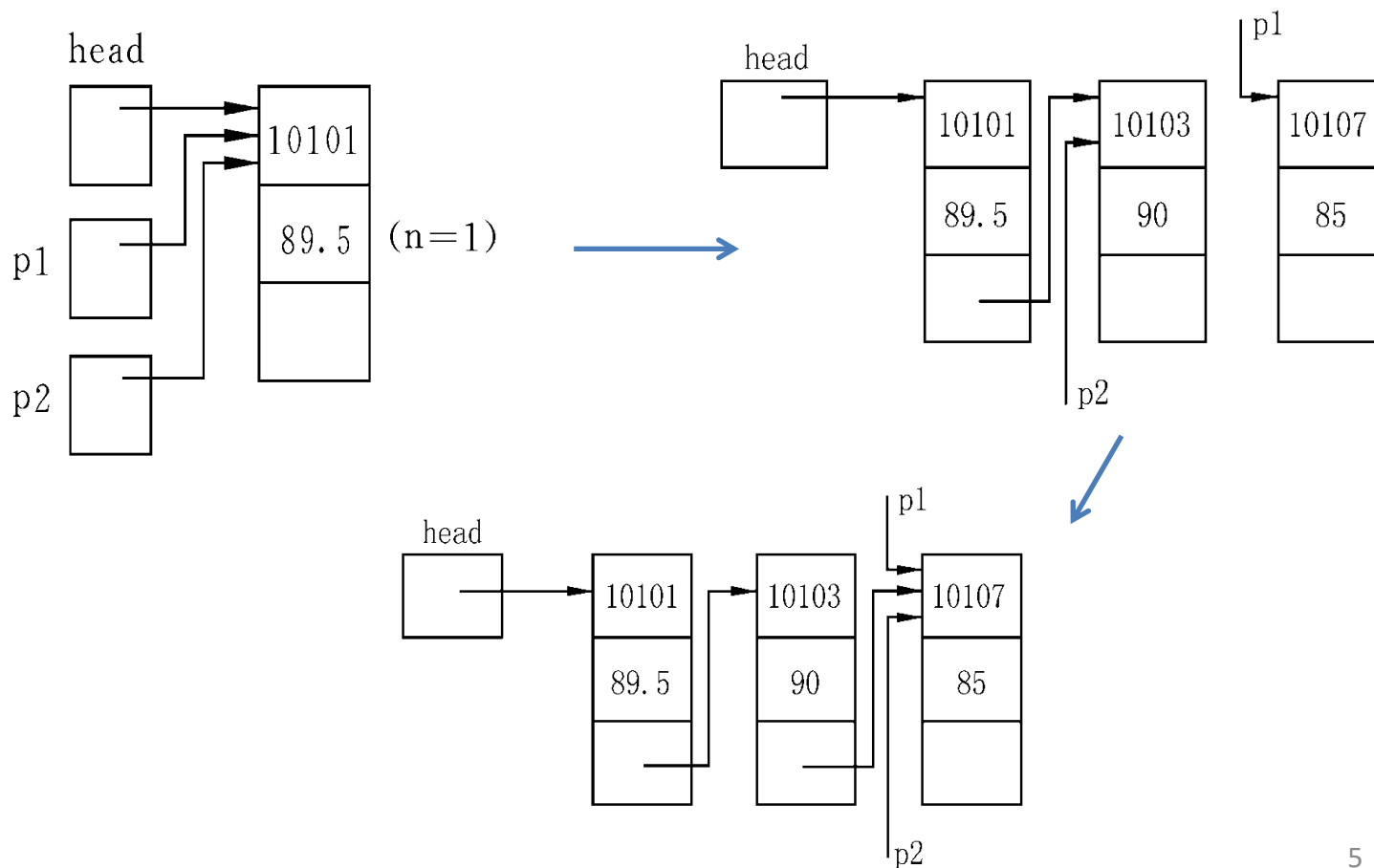
- 链表中各元素在内存中的地址可以是不连续的。要找某一元素，必须先找到上一个元素，根据它提供的下一元素地址才能找到下一个元素。如果不提供“头指针”(head)，则整个链表都无法访问。链表如同一条铁链一样，一环扣一环，中间是不能断开的。
- 链表这种数据结构，必须利用指针变量才能实现，即一个结点中应包含一个指针变量，用它存放下一结点的地址。

Linked List

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：

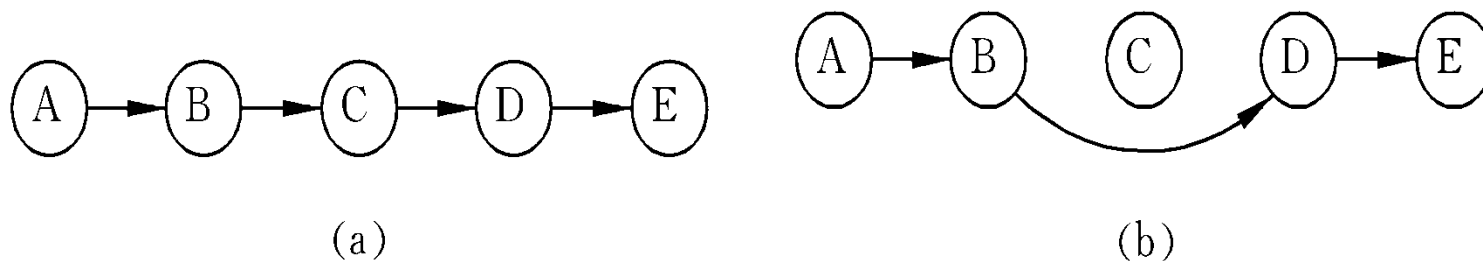
开辟一个新结点，并使p1、p2指向它	
读入一个学生数据给p1所指的结点	
head=NULL,n=0	
当读入的p1->num不是零	
n=n+1	
n等于1?	
真	假
head=p1 (把p1所指的结点作为第一个结点)	p2->next=p1 (把p1所指的结点连接到表尾)
p2=p1 (p2移到表尾)	
再开辟一个新结点，使p1指向它	
读入一个学生数据给p1所指结点	
表尾结点的指针变量置NULL	



Linked List

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

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			
是	原来的链表是空表		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		p1->next=p0 p0->next=NULL (插到表尾之后)
	是	否	
head=p0 p0->next =p1 (插到表 头之前)		p2->next =p0 p0->next =p1 (插到表 中间)	
n=n+1			

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

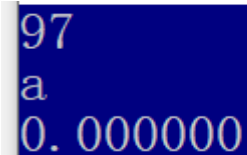
20 bytes = max(20, 4, 4)

Store at the same memory location!!!

Union

- 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

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)

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



#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

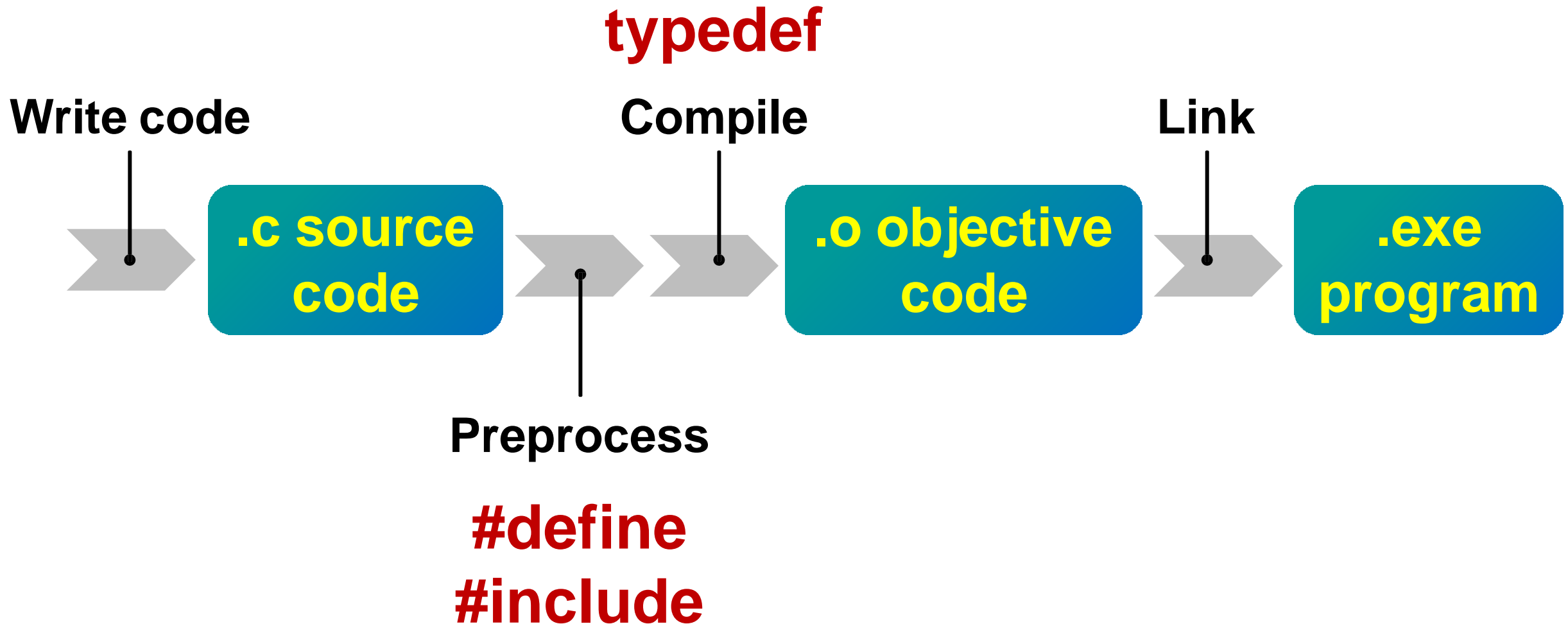
```
#include<stdio.h>
```

```
#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));
}
```

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(void)  
{
```

```
    int z = 5;
```

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

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

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

```
    return 0;
```

```
}
```



Preprocessed code

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

```
    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;
```

```
}
```

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

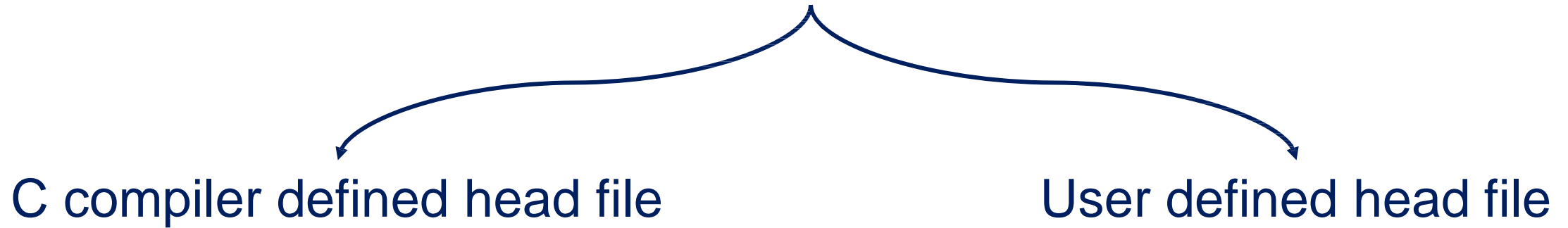
This is head file



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.

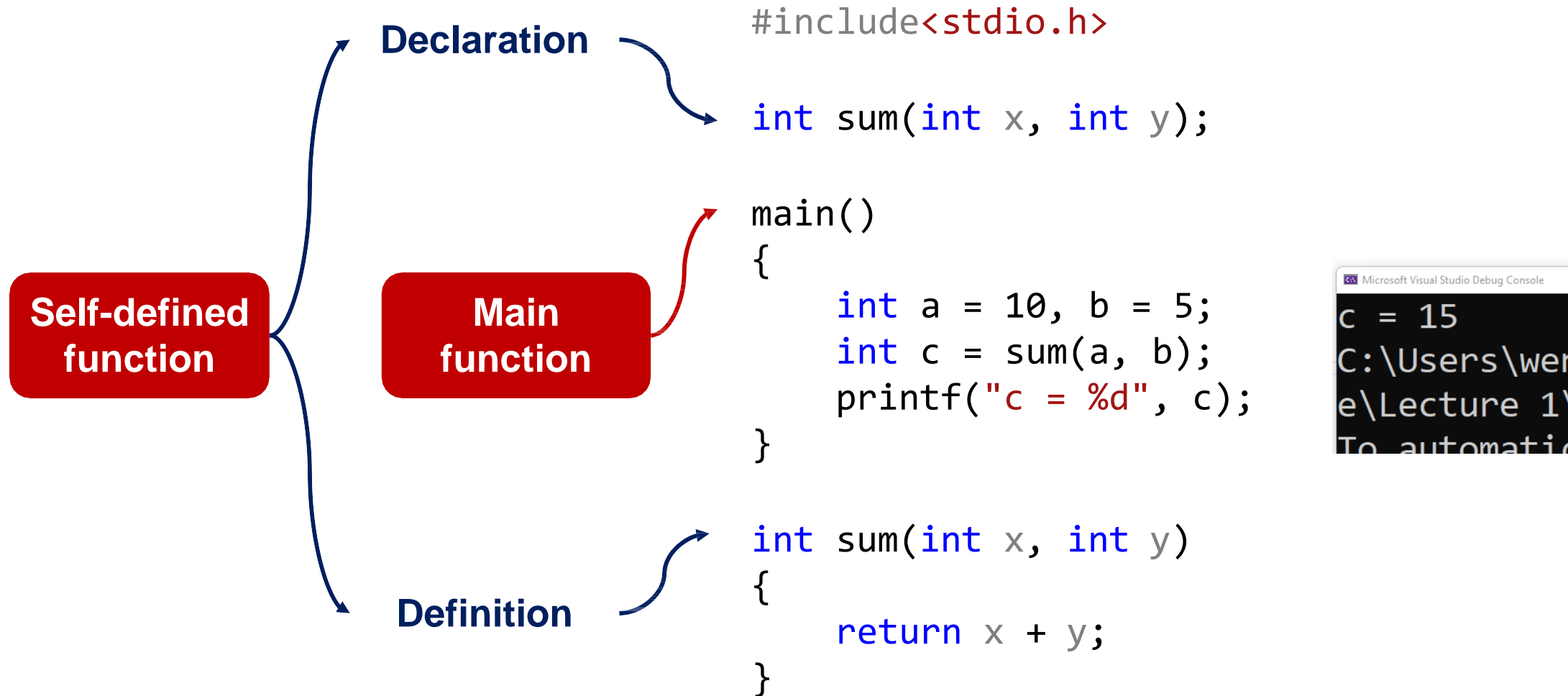
You already used C
Compile defined head file



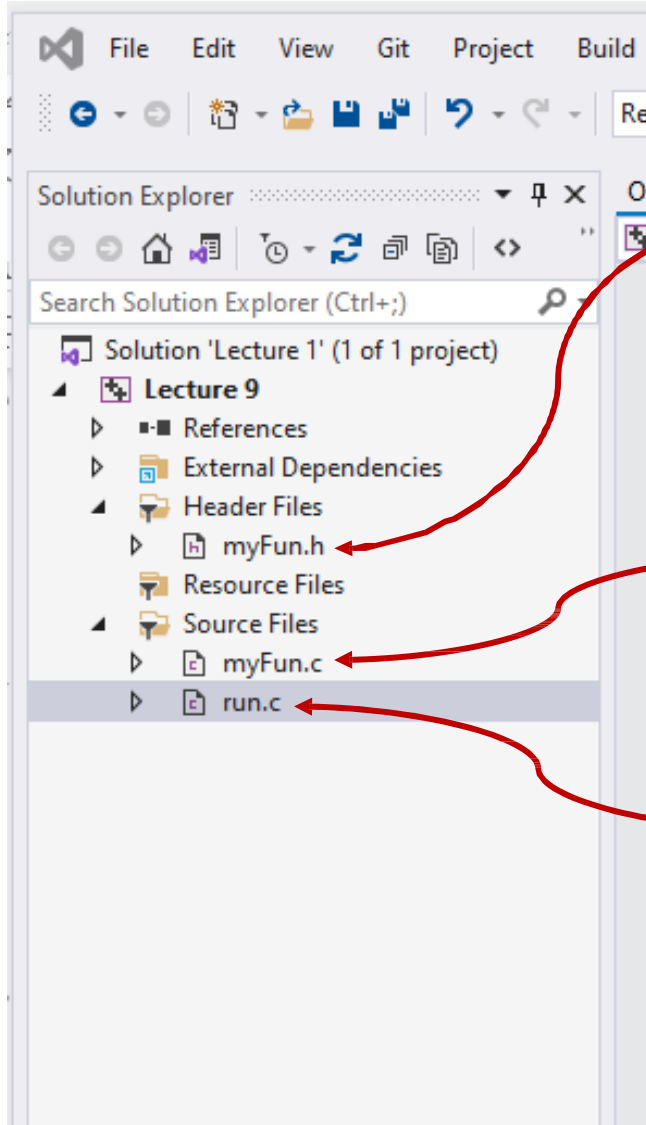
```
#include<stdio.h>
```

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


An example of function



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

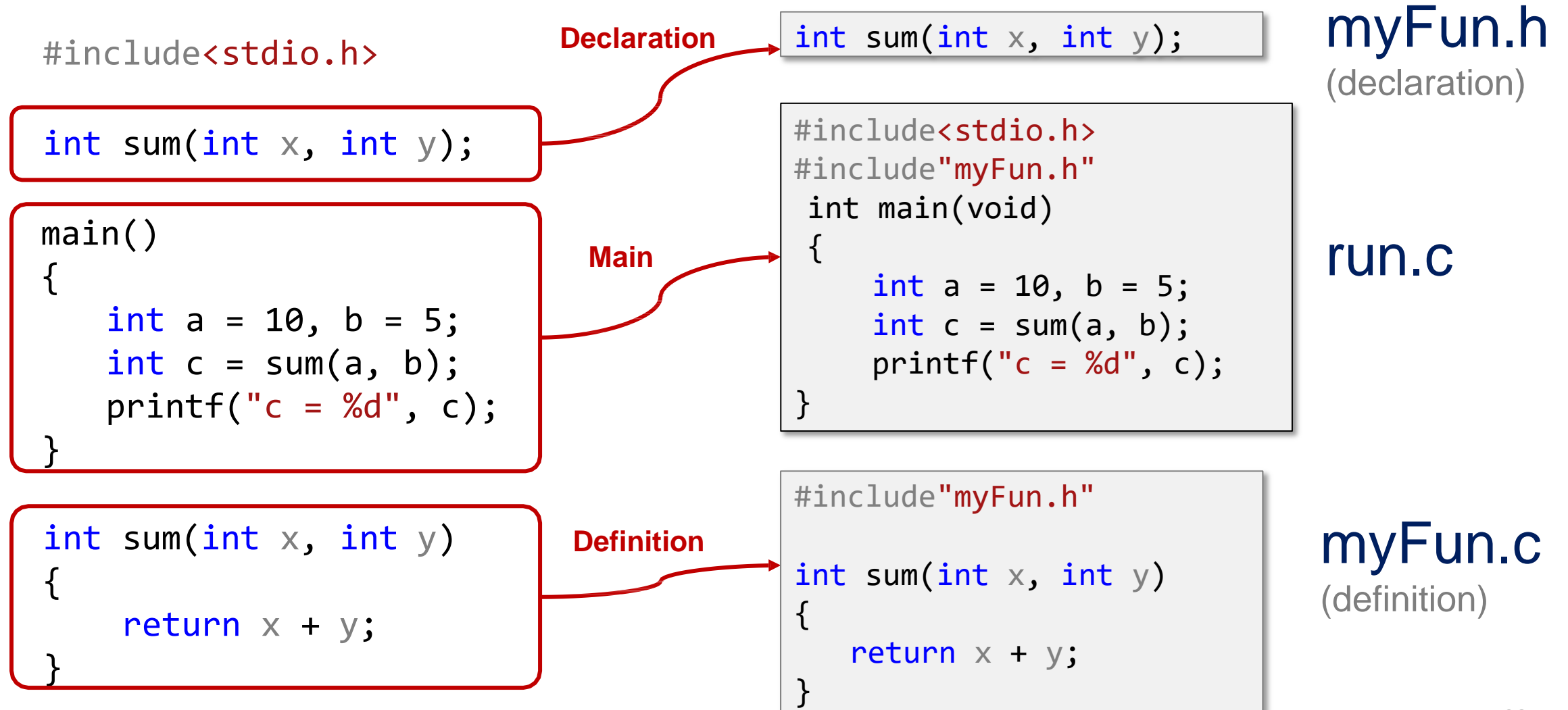
3. 在run.c里添加代码 `#include "myFun.c"`，包含方程声明

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"

int main(void)
{
    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

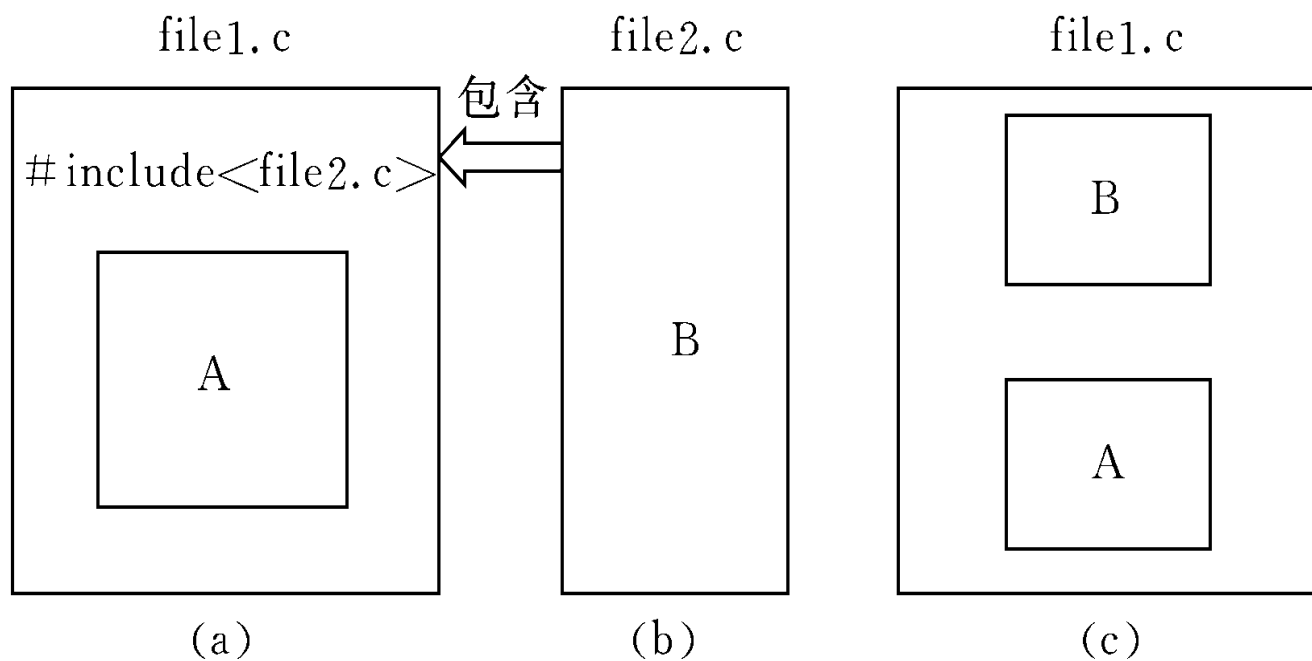


```
#include"myFun.h"
```

#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>**



You can use head files of C

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

```
#include<iostream>
```

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

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

```
...
```

```
#include<pthread.h> 线程
```

L15 review

Content

- **Bit and byte**
- **Data types and variables**
- **Operations**
- **I/O**

Basic

on Linux:

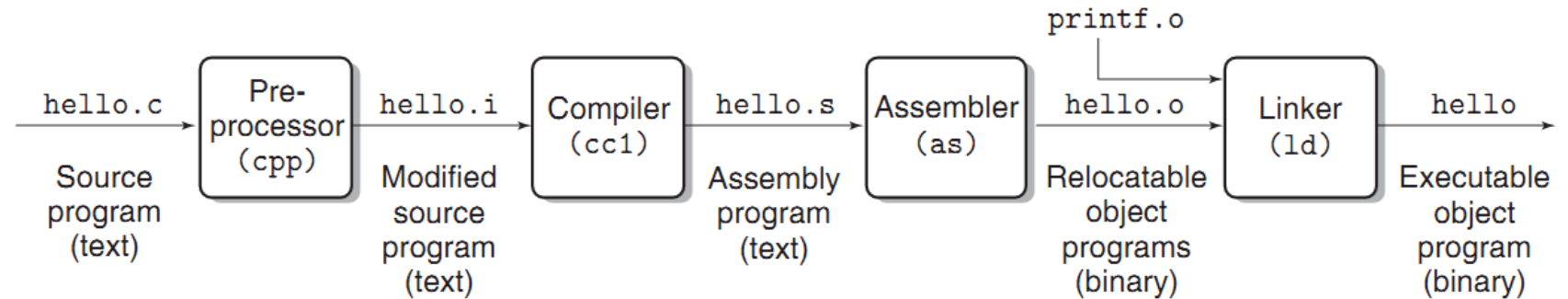
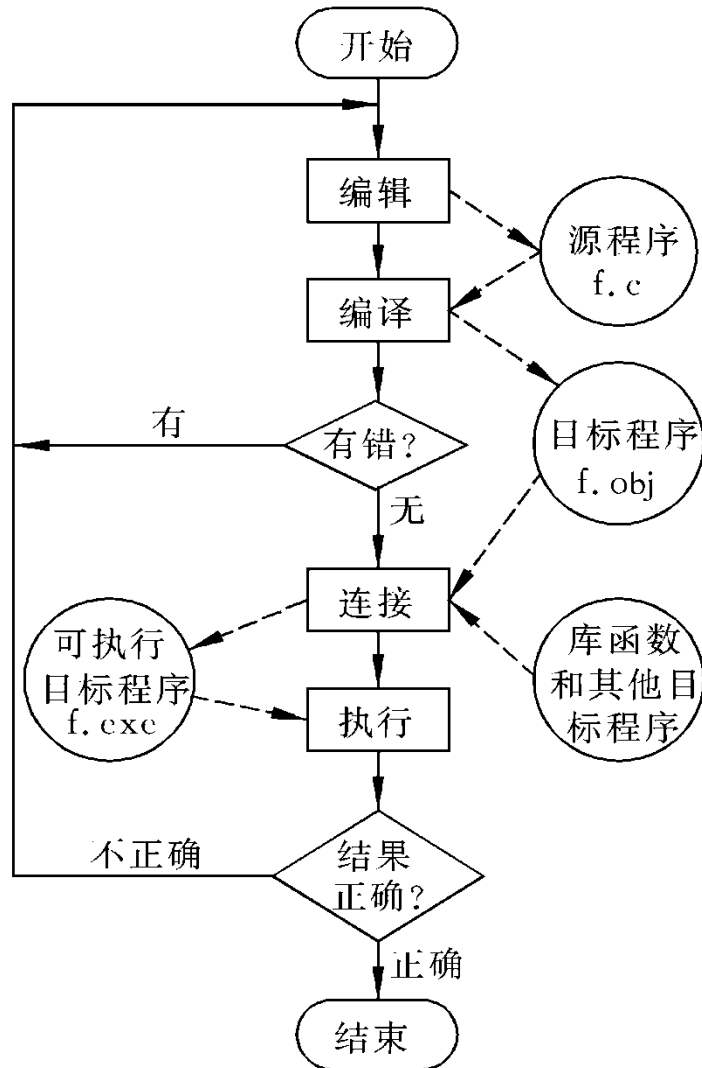


Figure 1.3 The compilation system.



- ❑ 预处理Preprocessing phase
- ❑ 编译Compilation phase
- ❑ 汇编Assembly phase
- ❑ 链接Linking phase

Basic

程序基本结构:

```
#include<stdio.h> //以#开头的语句称为预处理器指令
//以.h结尾的文件称为头文件,可以是C程序中现成的标准库也可以是自定义的库文件
//stdio.h文件包含了有关输入输出语句的函数
int main() {
    //main()函数是C程序处理的起点,程序的入口
    //大括号表示函数的主体用来包裹语句块
    printf("Hello World\n");
    //printf在屏幕上输出并换行
    return 0;
}
```

//单行注释:只能注释掉双斜杠后的当前行,被注释的内容不会被编译执行
/*多行注释*/:只要写在/*开头*/结尾的中间部分都可以被注释掉

Data

Bit (位)

The smallest unit for storage (atomic). It can hold one of two values: **0 or 1**.

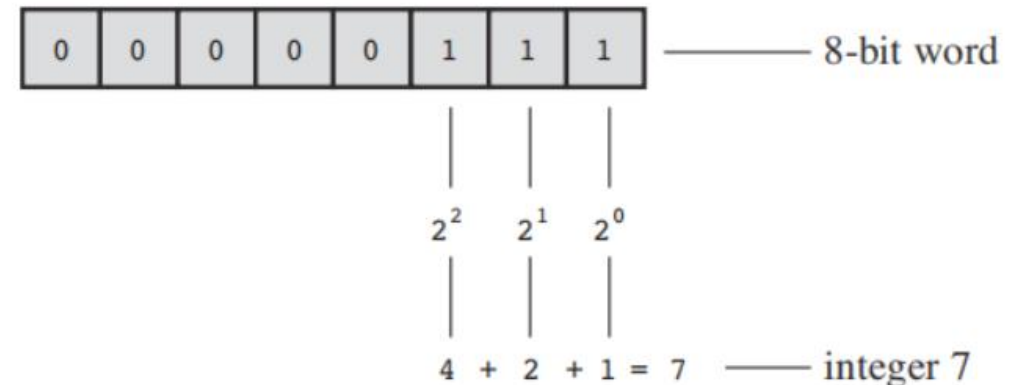
Byte (字节)

The smallest unit for information storage, **1 byte = 8 bits**

More diodes = More bits

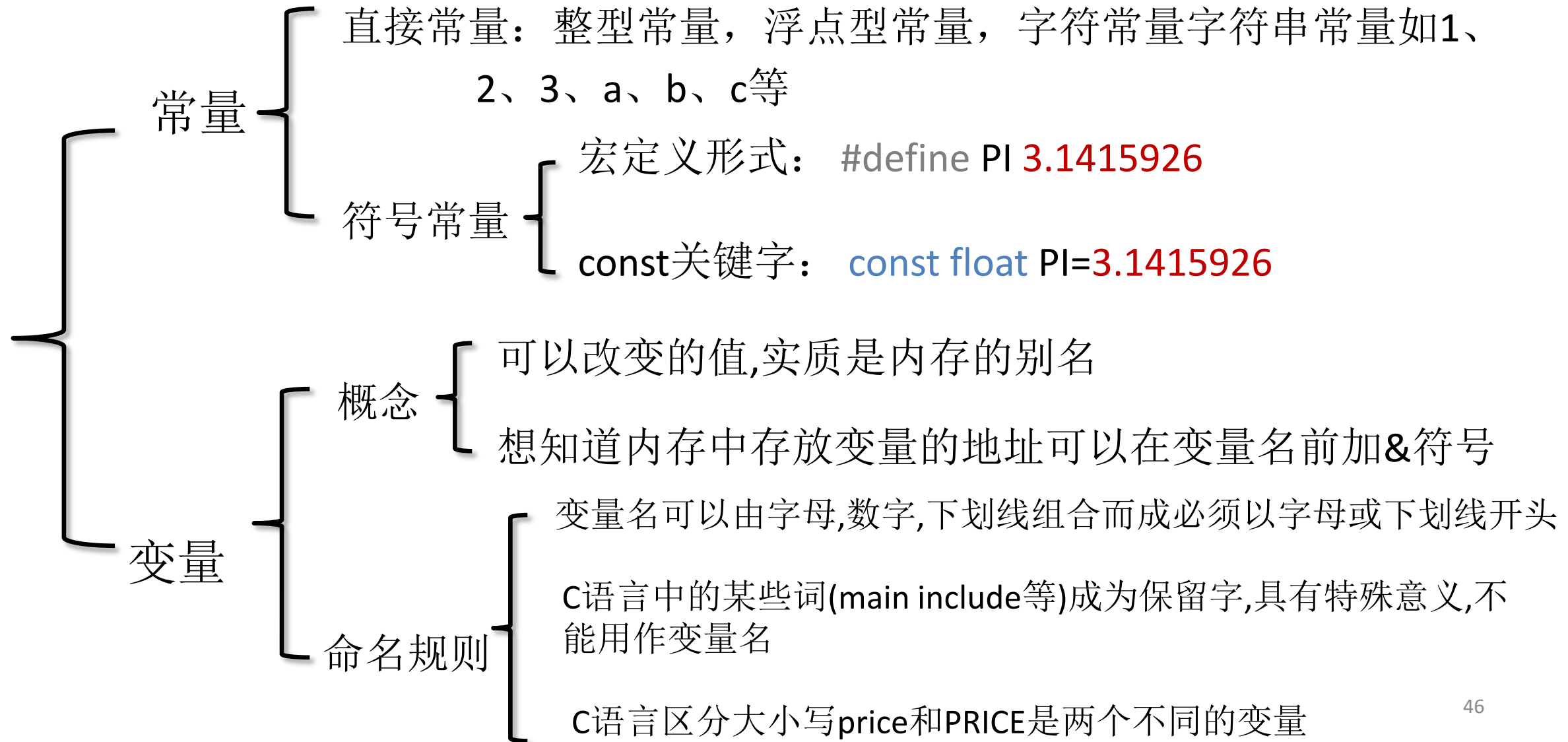


More complex information

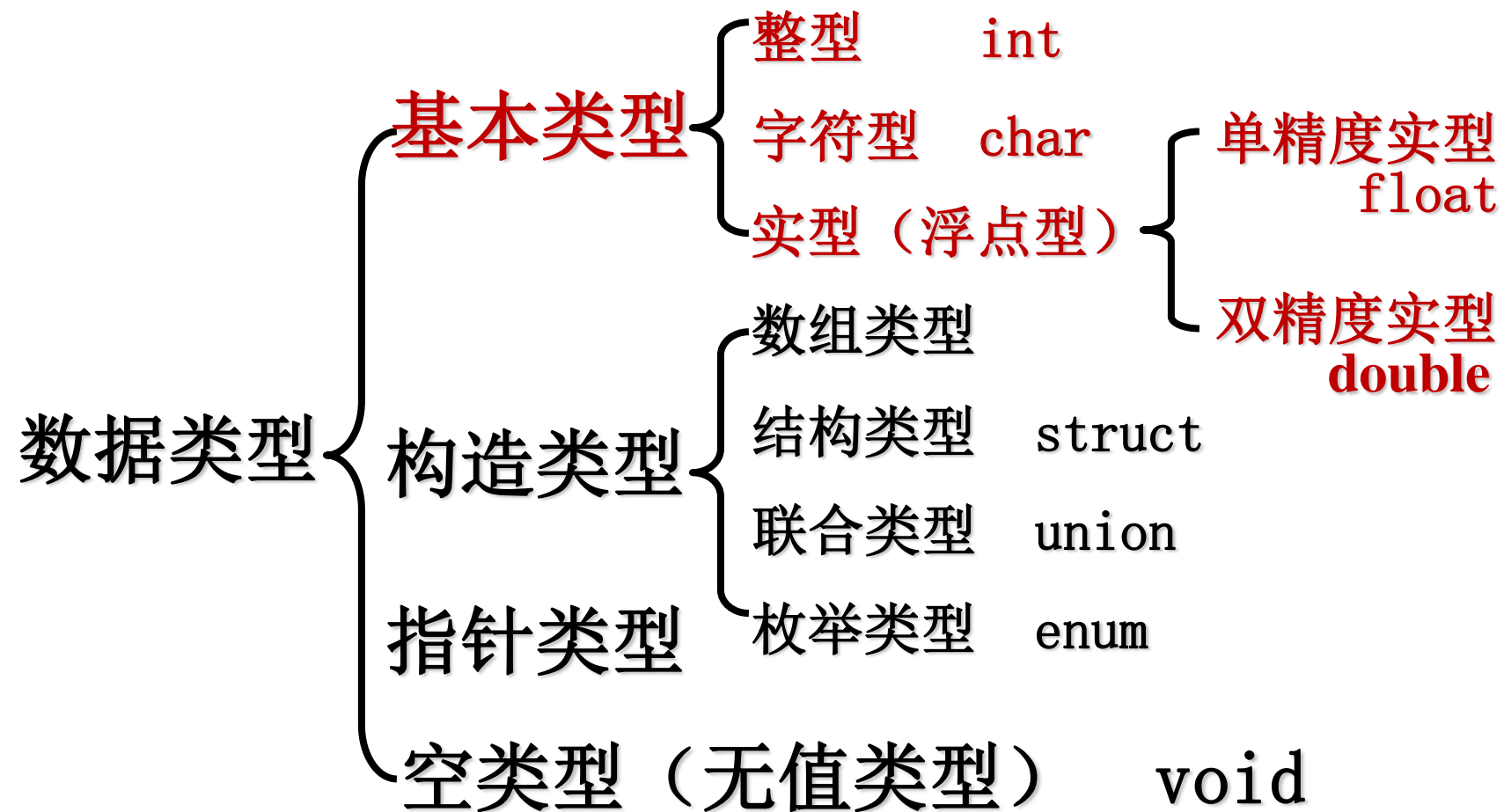


Storing the integer 7 using a binary code.

Data variables and constants



Data



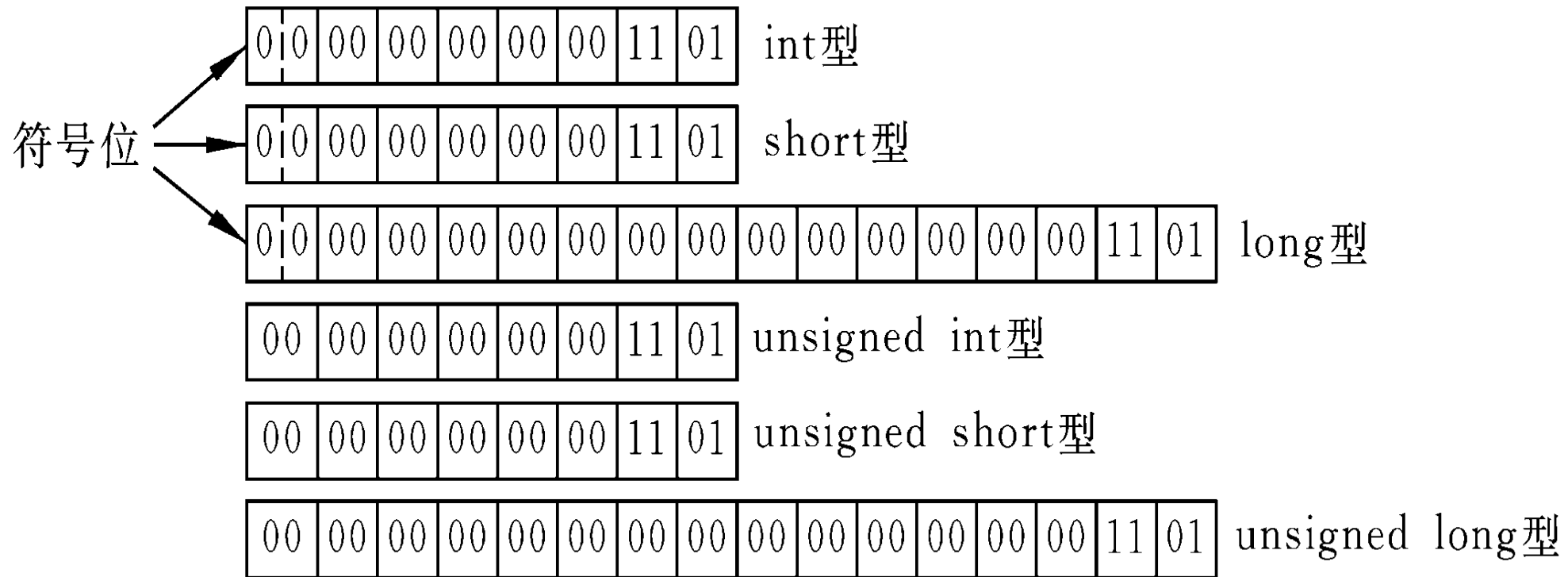
Data

最初 K&R 给出的关键字	C90 标准添加的关键字	C99 标准添加的关键字
int	signed	_Bool
long	void	_Complex
short		_Imaginary
unsigned		
char		
float		
double		

Integer

类型	类型说明符	长度	数的范围
基本型	int	4字节（大部分）	$-2^{31} \sim 2^{31}-1$
短整型	short	2字节	$-2^{15} \sim 2^{15}-1$
长整型	long	4字节	$-2^{31} \sim 2^{31}-1$
双长型	long long	8字节	$-2^{63} \sim 2^{63}-1$
无符号整型	unsigned	2字节	$0 \sim 65535$
无符号短整型	unsigned short	2字节	$0 \sim 65535$
无符号长整型	unsigned long	4字节	$0 \sim (2^{32}-1)$

Signed & unsigned



Sign bit:

0 for positive

1 for negative

Floating-Point Types

浮点型变量分为单精度（float型）、双精度（double型）和长双精度型（long double）三类形式。

类型	位数	数的范围
float	32	$10^{-38} \sim 10^{38}$
double	64	$10^{-308} \sim 10^{308}$
long double	128	$10^{-4931} \sim 10^{4932}$

两种表示形式 { 小数 0.123
指数 3e-3

注意：字母e(或E)之前必须有数字，且e后面的指数必须为整数

Using Characters: Type char

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LIVE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 2]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

```
char grade = 'A';
```

一个字符常量存放到一个字符变量中，实际上并不是把该字符的字型放到内存中去，而是将该字符的相应的ASCII代码放到存储单元中。这样使字符型数据和整型数据之间可以通用。

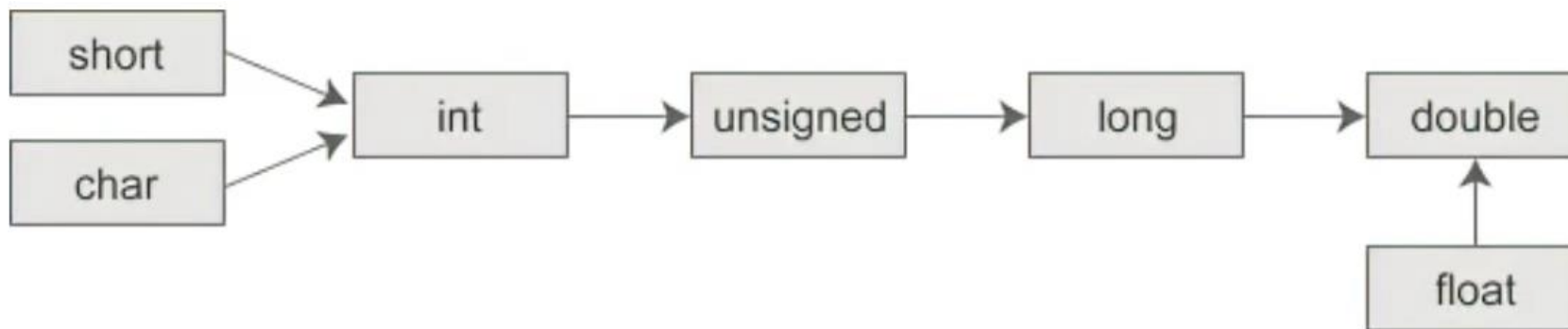
用**单引号**引起的字符实际上代表一个整数值，对于采用ASCII字符集的编译器而言，‘a’的含义与0141（八进制）或者97（十进制）严格一致

用**双引号**引起的字符串，代表的是指向无名数组起始字符的指针，该数组被双引号之间的字符以及一个额外的二进制值为零的字符‘\0’初始化。

Data

自动类型转换:

编译器默默地地进行的数据类型转换, 这种转换不需要程序员干预, 会自动发生。



若反向, 可能会导致数据失真, 或者精度降低

强制类型转换:

程序员在代码中明确地提出要进行类型转换

(type_name) expression

Char

转义序列	含义
\a	警报 (ANSI C)
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符
\\	反斜杠 (\)
\'	单引号
\"	双引号
\?	问号
\ooo	八进制值 (oo 必须是有效的八进制数, 即每个 o 可表示 0~7 中的一个数)
\xhh	十六进制值 (hh 必须是有效的十六进制数, 即每个 h 可表示 0~f 中的一个数)

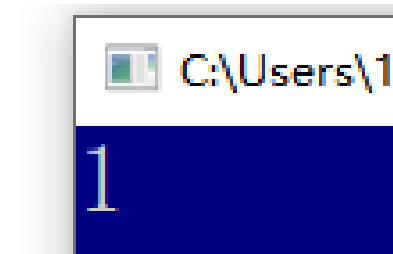
Others

- `_Bool`类型

C99标准添加了`_Bool`类型，用于表示布尔值，即逻辑值`true`和`false`。因为C语言用值1表示`true`，值0表示`false`。

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

int main() {
    bool a = true;
    bool b = false;
    if (a)
        printf("%d",a);
    if (b)
        printf("%d",b);
    return 0;
}
```



Data

占位符:

short int long	->%d
float	->%f
double	->%lf
char	->%c
十六进制	->%x
八进制	->%o
字符串	->%s

注意: %m.nf, m代表输出字段宽度, n代表小数点后的字段宽度, 即保留n位小数。如果小数点后的数大于n, 例如12.4567按照%5.2f输出得12.46 (四舍五入), 如果总位数大于m (小数只到两位), 按照实际位数输出, 例如111.4567按%5.2f输出, 111.46 (总字符宽度为6)。小数点也算一位数字, 占据一个位置。

如果一个整型常量的第一个字符是0, 那么该常量将被视为八进制数。

```
int a = 012;  
printf("a= %d\n", a);
```

a= 10

I/O

`printf([formatted text], [arguments]);`

- -:结果左对齐,右边填充格

```
float a = 5.3, b = -9.6;  
printf("a=%-8.2f\n", a);  
printf("b=%8.2f ", b);
```

```
a=5.30  
b=   -9.60
```

- +:输出符号(正负符号)

```
printf("a=%+f b=%+f\n", a, b);
```

```
a=+5.300000 b=-9.600000
```

- 空格:输出值为正(包括0)时为空格,为负时为负号
- 设置宽度及精度

`scanf([formatted text], [arguments]);`

- 易错点:
- &不要忘记,若给出变量名会出错
 - 输入多个数值时,两个输入数据之间的非格式字符需要与格式化字符串一致
 - 读取字符串遇到空格时会结束当前数据输入

I/O

字符输入\输出

- `getchar()`
`char c = getchar();`
受单个字符，输入数字也按字符处理
- `putchar()`
`putchar(c)`
作用等同于`printf("%c",ch);`

字符串输入\输出

- `gets()`
`gets(char *s);`
输入字符串，只能输入一个字符串
- `puts()`
`puts(char *s);`
输出字符串，只能输出一个字符串

Operators

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	
	()	圆括号	(表达式)/函数名 (形参表)		
	.	成员选择(对象)	对象.成员名		
	->	成员选择(指针)	对象指针->成员名		
2	-	负号运算符	-表达式	右到左	单目运算符
	(类型)	强制类型转换	(数据类型)表达式		
	++	自增运算符	++变量名/变量名++		单目运算符
	--	自减运算符	--变量名/变量名--		单目运算符
	*	取值运算符	*指针变量		单目运算符
	&	取地址运算符	&变量名		单目运算符
	!	逻辑非运算符	!表达式		单目运算符
	~	按位取反运算符	~表达式		单目运算符
	sizeof	长度运算符	sizeof(表达式)		
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		双目运算符
	%	余数(取模)	整型表达式/整型表达式		双目运算符
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		双目运算符
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		双目运算符
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		双目运算符
	<	小于	表达式<表达式		双目运算符
	<=	小于等于	表达式<=表达式		双目运算符
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!=表达式		双目运算符
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10		按位或	表达式 表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12		逻辑或	表达式 表达式	左到右	双目运算符
13	?:	条件运算符	表达式1?表达式2: 表达式3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	
	/=	除后赋值	变量/=表达式		
	=	乘后赋值	变量=表达式		
	%=	取模后赋值	变量%=表达式		
	+=	加后赋值	变量+=表达式		
	-=	减后赋值	变量-=表达式		
	<<=	左移后赋值	变量<<=表达式		
	>>=	右移后赋值	变量>>=表达式		
	&=	按位与后赋值	变量&=表达式		
	^=	按位异或后赋值	变量^=表达式		
	=	按位或后赋值	变量 =表达式		
15	,	逗号运算符	表达式, 表达式, ...	左到右	从左向右顺序运算

Operators

- 优先级最高者其实并不是真正意义上的运算符，包括**数组下标[]**、**函数调用操作符**、**各结构成员选择操作符**。
- **单目运算符**优先级仅次于前述运算符。
- 优先级比单目运算符低的，接下来是**双目运算符**。在双目运算符中，算数运算符的优先级最高，移位运算符次之，关系运算符再次之，接着是逻辑运算符、赋值运算符，最后是条件运算符。

- 任何一个逻辑运算符的优先级低于任何一个关系运算符；
- 移位运算符的优先级比算术运算符要低，但是比关系运算符要高

Operator

词法分析中的“贪心法”：对于将连续的两个符号分别对待还是合起来作为一个对待，C语言对这个问题的解决方案可以归纳为一个简单的规则：**每一个符号应该包含尽可能多的字符**，也就是说，从左到右一个一个的读入，如果该字符能组成一个符号，那么再读入下一个字符，判断已经读入的两个字符组成的字符串是否可以是一个符号的组成部分；如果可能继续读入下一个字符，重复上述过程，直到读入的字符组成的字符串已不再可能组成一个有意义的符号。

判断下列是否正确以及其表达式含义：

- `c = a --- b;`
- `y = x / *p;`
- `float *g();`
- `float (*h)();`
- `*p++;`

Operator

易错点:

- = 不同于 ==

y不等于0时,
恒为真

if(x=y){
 break;
}

if(x==y){
 break;
}

- &和| 不同于 && 和||

&和|是按位运算符, &&和||为逻辑运算符

Condition

if(condition)
{option A}

else

{option B}

没有分
号!!!

condition

Ture/Yes/1
(2<5, 3==3, 'a'!='b')

False/No/0
(1>=5, 3<1, 'a'=='b')

若不加{}, 默认读取一句!

```
if (expression)
    statement
```

```
if (expression)
    statement1
else
    statement2
```

```
if (expression1)
    statement1
else if (expression2)
    statement2
else
    statement3
```

Condition

? statement

expression1 ? expression2 : expression3

- 先求解表达式 1，若为非 0（真）则求解表达式 2，此时表达式 2 的值就作为整个条件表达式的值。若表达式 1 的值为 0（假），则求解表达式 3，表达式 3 的值就是整个条件表达式的值。
- 表达式 2”和“表达式 3”不仅可以是数值表达式，还可以是赋值表达式或函数表达式。
- 若表达式 1 为真，则表达式 3 不进行运算

Condition

```
int a = 3;
switch(a)
{
    case 1:
        //...
        break;
    case 2:
        //...
        break;
    default:
        //...
}
```

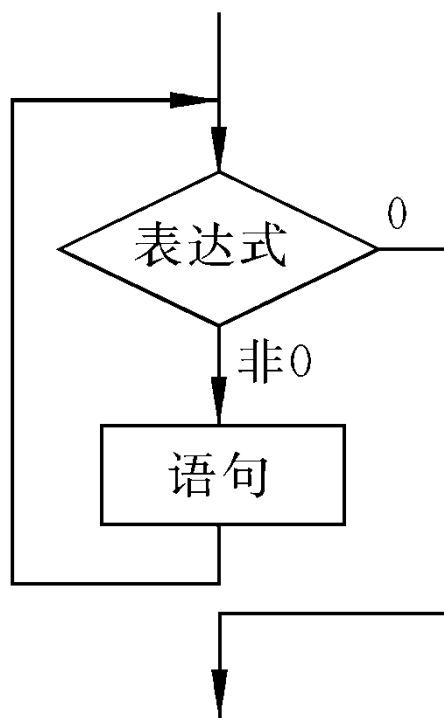
- 当表达式的值与某一个case后面的常量表达式的值相等时，就执行此case后面的语句，若所有的case中的常量表达式的值都没有与表达式的值匹配的，就执行default后面的语句。
- 每一个case的常量表达式的值必须互不相同，否则就会出现互相矛盾的现象（对表达式的同一个值，有两种或多种执行方案）。
- 各个case和default的出现次序不影响执行结果。
- 应该在执行一个case分支后,用一个break语句来终止switch语句的执行，最后一个case分支不用。
- 可以多个case 共用一组执行语句

```
switch(color)
{
    case 1:printf("red");
    case 2:printf("yellow");
    case 3:printf("blue");
}
```

如果color=2，会输出什么结果？

Loop

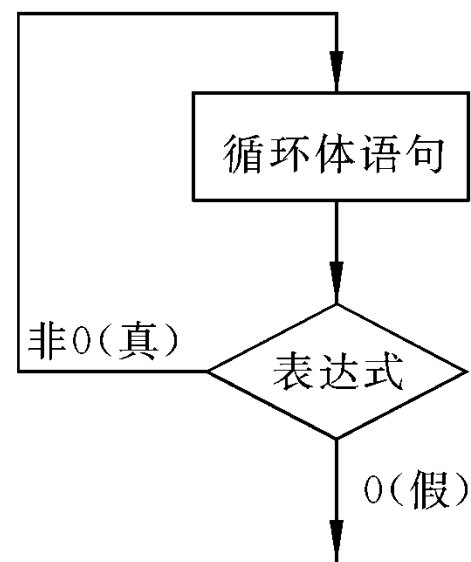
```
while(condition)
{
    statement;
}
```



有可能一次循环都不执行

```
do
{
    statement;
}while( condition );
```

不能丢!!!



至少执行一次循环

(a)

Loop

```
int a = 0;
```

```
while (a < 10)
```

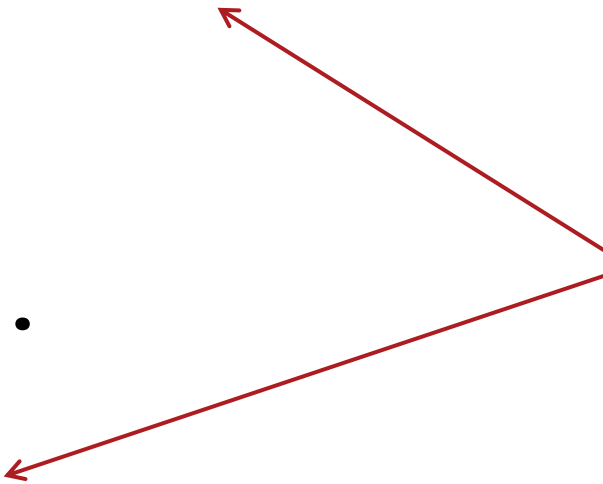
```
{
```

```
    // ...
```

```
    a++;
```

```
}
```

大小关系、增
减要匹配



- 在循环体中应有使循环趋向于结束的语句
- 循环体如果包含一个以上的语句，应该用花括弧括起来，以复合语句形式出现。

Loop

表达式1 表达式2 表达式3

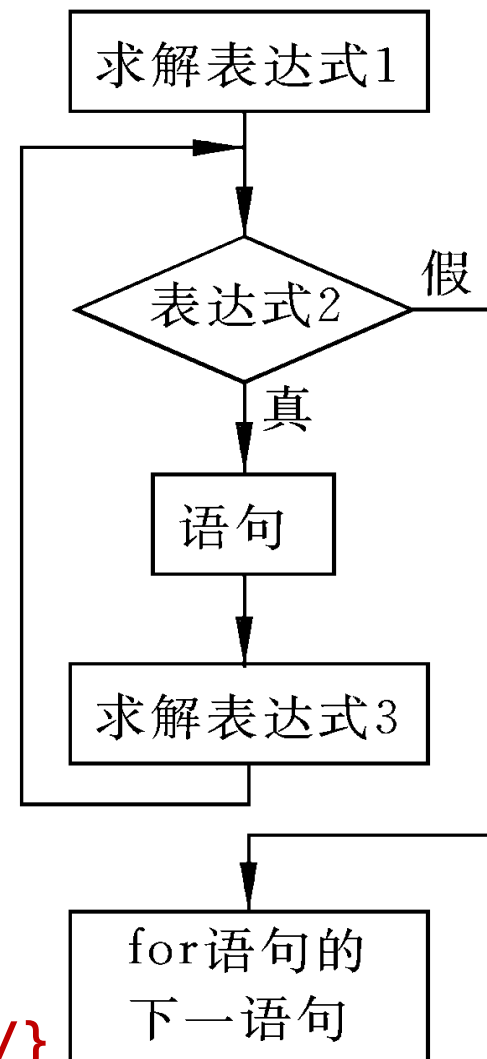
```
for ( init; condition; increment )  
{  
    statement;  
}
```

- 表达式2一般是关系表达式(如*i*≤100)或逻辑表达式(如*a*<*b* && *x*<*y*), 但也可以是数值表达式或字符表达式, 只要其值为非零, 就执行循环体。

for(*i*=0; (*c*=getchar())!=' \n'; *i*+=*c*);

- 在C99中, for 语句中的第一个表达式可以被声明替换, 但是在循环体外不可见该变量

```
for (int i = 0; i < n; i++)  
    {printf("%d", i); /*legal ; i is visible inside loop*/}  
    printf("%d", i); /******WRONG*****/
```



Break & continue

break:

- break语句不能用于循环语句和switch语句之外的任何其他语句中
- break 只能跳出一层循环。当有多层循环嵌套的时候，break只能跳出“包裹”它的最里面的那一层循环，无法一次跳出所有循环。
- 在多层 switch 嵌套的程序中，break 也只能跳出其所在的距离它最近的 switch。

continue:

- continue仅限于循环中
- continue 作用为结束本次循环，即跳过循环体中下面尚未执行的语句，然后进行下一次是否执行循环的判定。

Array

一维数组

- 概念：连续、等大的存储空间，类型相同的数据集合
- 定义：type name[size]

```
int arr[3]; // 只定义不赋值
int arr1[3]={1,2,3}; // 完全列举法
int arr2[3]={1,2}; // 部分列举法
int arr3[]={1,2,3,4}; // 省略大小列举法
```

- 数组访问：通过数组下标开始，数组下标从0开始

Array

二维数组

- 定义: `type name[size][size];`

`int a[2][3]={{1, 2, 3}, {4, 5, 6}};` //分行赋值

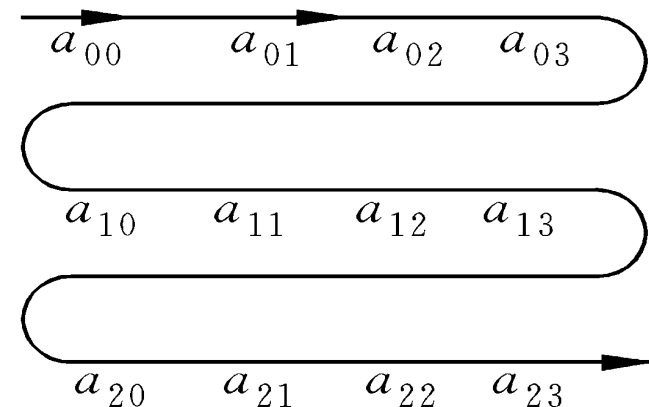
`int a[2][3]={1, 2, 3, 4, 5, 6};` //将所有数据写在一个花括号内, 按顺序对各元素赋初值

`int a[3][4]={{1}, {5}, {9}};` //可以对部分元素赋初值

`int a[][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};`

//如果对全部元素都赋初值, 则定义数组时对第一维的长度可以不指定, 但第二维的长度不能省。

- 存储方式: 按行存放, 即先顺序存放第一行的元素, 再存放第二行的元素



String

- 字符串是常量是用**双引号**括起来的任意字符序列
- C语言中没有专门的字符串变量，通常用一个字符数组来存放一个字符串
- 字符数组和字符串的区别是字符串的末尾有一个空字符**\0**
- `printf`输出字符串，如果一个字符数组中包含一个以上'`\0`'，则遇第一个'`\0`'时输出就结束。
- 利用一个`scanf`函数输入的字符串中若输入空格，其后都为'`\0`'

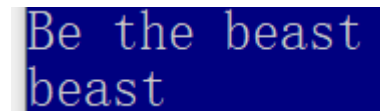
H	o	w	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
---	---	---	----	----	----	----	----	----	----	----	----	----

String

strcpy(str1, str2);

- 字符串的拷贝需要用到strcpy()。
- “str1”必须写成数组名形式(如str1), “str2”可以是字符数组名, 也可以是一个字符串常量。如strcpy(str1, "China");
- 第一个参数不必指向数组开始。

```
char orig[40] = "beast";  
char dest[100] = "Be the best that you can be";  
char *ps;  
ps = strcpy(dest + 7, orig);  
printf("%s\n", dest);  
printf("%s", ps);
```



```
Be the best  
beast
```

- 利用strncpy函数将str2中前面若干个字符复制到str1中去。
例如:strncpy(str1, str2, 2);

String

`strcat(str1,str2);`

- 用于拼接字符串，把第2个字符串的备份附加在第1个字符串末尾，并把拼接后形成的新字符串作为第1个字符串，并返回第一个字符串。

`strlen(str1);`

- 给出字符串中的实际字符长度（不包括'\0'在内）

```
char ch1[12] = "Hello";  
printf("%d", strlen(ch1));
```

5

`sizeof(str1);`

- 以字节为单位给出指定类型的大小

```
char ch1[12] = "Hello";  
printf("%d", sizeof(ch1));
```

12

String

`strcmp(str1,str2);`

- 将两个字符串从第一个字符开始比较，若出现不同字符，以第一对不相同的字符的比较结果为准。

`strlwr(str1)`

- 所有字符变成小写

`strupr(str1)`

- 所有字符变成大写

Function

- 一个源程序文件由一个或多个函数以及其他有关内容（如命令行、数据定义等）组成。一个源程序文件是一个编译单位，在程序编译时是以源程序文件为单位进行编译的，而不是以函数为单位进行编译的。
- C 程序的执行是从main函数开始的，如果在main函数中调用其他函数，在调用后流程返回到main函数，在main函数中结束整个程序的运行。
- 函数类型包含库函数和自定义函数。库函数由C语言系统提供，用户无须定义，也不必在程序中作类型说明，只需在程序前**包含有该函数定义的头文件**即可使用。自定义函数则是用户在程序中根据需要而编写

Function

声明:

```
int sum(int a, int b);
```

Return type
(float,double,char...)

Input parameters (形式参数)

定义:

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

- 函数声明是可以省略形式参数名称，可以于定义时不同。
- 在定义时必须指定形参类型！！

Function

C有2种参数传递方式:

- ✓ 值传递
- ✓ 地址传递

Value

System
creates a new
memory unit

Arguments

(实参)

实参与形参的类型应
相同或赋值兼容。

Parameters

(形参)

Address

System uses
the existing
memory unit

Function

- 按函数在程序中出现的位置来分，可以有以下函数语句（这时不要求函数带返回值，只要求函数完成一定的操作。）、函数表达式（要求函数带回一个确定的值以参加表达式的运算）、函数参数（递归）
- 在函数调用时，即使函数不带参数，也应该包括参数列表，比如：f()
- 如果实参表列包括多个实参，对实参求值的顺序并不是确定的，有的系统按自左至右顺序求实参的值，有的系统则按自右至左顺序。

```
int i=2,p;
```

```
p=f(i,++i);
```

如果按自左至右顺序求实参的值，
则函数调用相当于f(2,3)

如果按自右至左顺序求实参的值，
则函数调用相当于f(3,3)

Variable scope

局部变量:

- 在一个函数内部定义的变量称局部变量, 它只在本函数范围内有效。
- 不同函数中可以使用相同名字的变量, 它们代表不同的对象, 互不干扰。
- 形式参数也是局部变量。
- 在一个函数内部, 可以在复合语句中定义变量, 这些变量只在本复合语句中有效。例如 `for (int i=0; i<n; i++)`

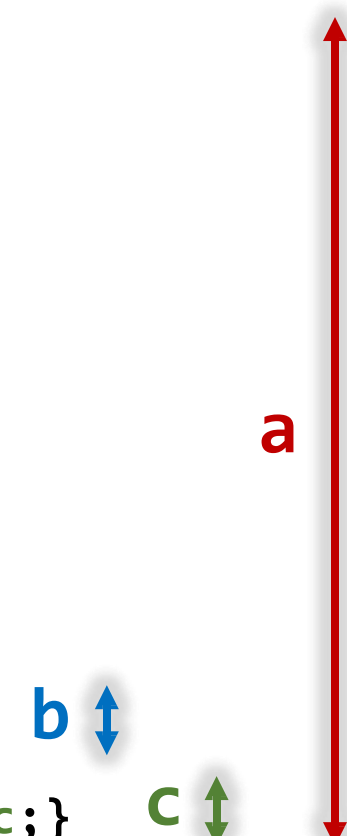
全局变量:

- 函数之外定义的变量称为外部变量。外部变量可以为本文件中其他函数所共用。它的有效范围为从定义变量的位置开始到本源文件结束。所以也称全程变量。

Variable scope

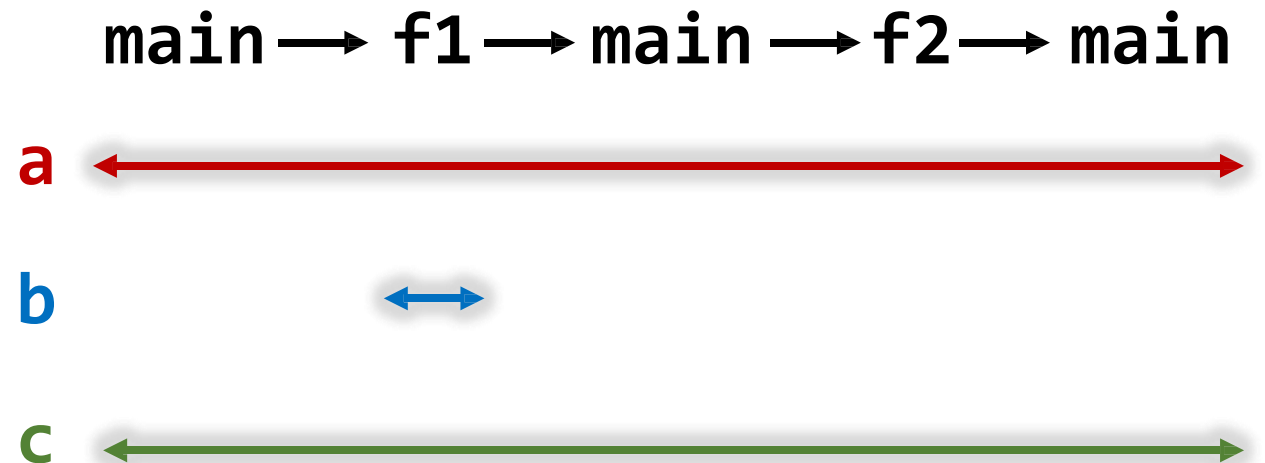
Scope in space

```
int a;  
f1();  
f2();  
main()  
{  
    f1();  
    f2();  
}  
f1(){int b;}  
f2(){static int c;}
```



The diagram illustrates the spatial scope of variables. A long red vertical double-headed arrow labeled 'a' spans the entire height of the code block, indicating that variable 'a' is in global scope. A blue vertical double-headed arrow labeled 'b' is positioned next to the 'f1()' function definition, indicating its local scope. A green vertical double-headed arrow labeled 'c' is positioned next to the 'f2()' function definition, indicating its local scope.

Scope in time



Variable scope

自动变量auto:

- 不专门声明为static存储类别的局部变量都是动态分配存储空间，在调用该函数时系统会给它们分配存储空间，在函数调用结束时就自动释放这些存储空间。因此这类局部变量称为自动变量。
- 函数中的形参和在函数中定义的变量(包括在复合语句中定义的变量)，都属此类

静态局部变量static:

- 当函数中的局部变量的值在函数调用结束后不消失而保留原值时，在程序整个运行期间都不释放，该变量称为静态局部变量。
- 对静态局部变量是在编译时赋初值的，即只赋初值一次，在程序运行时它已有初值。以后每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值。
- 虽然静态局部变量在函数调用结束后仍然存在，但其他函数不能引用它。

Variable scope

外部变量extern:

```
extern int a;
```

语句说明a是一个外部整型变量，extern关键字说明a的存储空间是在程序的其他地方分配的。从链接器角度看，上述声明是**对外部变量a的引用**，而不是定义。

- 外部变量是在函数的外部定义的全局变量，它的作用域是从变量的定义处开始，到**本程序文件的末尾**。在此作用域内，全局变量可以为程序中各个函数所引用。
- 可以不仅限于本文件使用
- 若想使得某些外部变量只限于被本文件引用，而不能被其他文件引用，可以在定义外部变量时加一个**static**声明。

Pointer

- 指针：专门存放地址的变量

type *var;

type *var2 = &var1;

- 上述例子指针变量名为var2，不是*var2
- 声明指针变量时必须指定指针所指向变量的类型
- 指针变量中只能存放地址（指针），不要将一个整数（或任何其他非地址类型的数据）赋给一个指针变量。

Pointer

对“&”和“*”运算符说明：

pointer_1=&a;

(1) **&* pointer_1** 的含义是什么？

“&”和“*”两个运算符的优先级别相同，但按自右而左方向结合。因此，&* pointer_1 与 &a 相同，即变量a的地址。

(2) ***&a** 的含义是什么？

先进行&a运算，得a的地址，再进行*运算。*&a 和*pointer_1 的作用是一样的，它们都等价于变量a。即*&a 与 a 等价。

Pointer

```
#include <stdio.h>
void swap_pointer(int *p1, int *p2);
void swap_value(int, int);

int main(void) {
    int a = 5;
    int b = 9;
    int *pointer_1, *pointer_2;
    printf("原始值:   a=%d ,b=%d\n", a, b);
    swap_value(a, b);
    printf("传递值:   a=%d ,b=%d\n", a, b);
    pointer_1 = &a;
    pointer_2 = &b;
    swap_pointer(pointer_1, pointer_2);
    printf("传递指针: a=%d ,b=%d", a, b);
}
```

```
void swap_value(int p1, int p2) {
    int temp;
    temp = p1;
    p1 = p2;
    p2 = temp;
}

void swap_pointer(int *p1, int *p2) {
    int temp;
    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

原始值:	a=5 , b=9
传递值:	a=5 , b=9
传递指针:	a=9 , b=5

Pointer & array

- 定义一个指向数组元素的指针变量的方法，与以前介绍的指向变量的指针变量相同。如果数组为int型，则指针变量的基类型亦应为int型。
- 引用一个数组元素，可以用：
 - ①下标法，如 a [i]形式;
 - ② 指针法，如*(a+i)。
- 数组作为函数参数：
 - ①void inv(int x[], int n); /*形参x是数组名*/
 - ②void inv(int *x, int n); /*形参x为指针变量*/

```
int *p;  
int a[3];  
p=a;  
p=&a
```

√
✗

&a代表指向数组的指针，而p是指向整型变量的指针

Pointer & array

指针和多维数组的关系：

```
int  
zippo[4][2]
```

数组名zippo是该数组首元素的地址，所以zippo的值和&zippo[0]的值相同。而zippo[0]本身是一个内含两个整数的数组，所以zippo[0]的值和它首元素（一个整数）的地址（即&zippo[0][0]的值）相同。简而言之，zippo[0]是一个占用一个int大小对象的地址，**而zippo是一个占用两个int大小对象的地址**。由于这个整数和内含两个整数的数组都开始于同一个地址，所以**zippo和zippo[0]的值相同**。

zippo[0]是该数组首元素（zippo[0][0]）的地址，所以*(zippo[0])表示储存在zippo[0][0]上的值（即一个int类型的值）。与此类似，*zippo代表该数组首元素（zippo[0]）的值，但是zippo[0]本身是一个int类型值的地址。该值的地址是&zippo[0][0]，所以*zippo就是&zippo[0][0]。对两个表达式应用解引用运算符表明，****zippo与*&zippo[0][0]等价，这相当于zippo[0][0]，即一个int类型的值**。简而言之，zippo是地址的地址，必须解引用两次才能获得原始值。

Pointer & array

假设有如下声明：

```
int a;  
int * pt;  
int (*pa)[3];  
int ar1[2][3];  
int ar2[3][2];  
int **p2; // 一个指向指针的指针
```

判断一下几种情况是否正确与所代表的含义：

```
pt = &ar1[0][0]; // 都是指向int的指针  
pt = ar1[0];     // 都是指向int的指针  
pt = ar1;        // 无效  
pa = ar1;        // 都是指向内含3个int类型元素数组的指针  
pa = ar2;        // 无效  
p2 = &pt;        // 都是指向int *的指针  
*p2 = ar2[0];    // 都是指向int的指针  
p2 = ar2;        // 无效  
a = *(ar[1]+2);
```

Pointer & array

- 二维数组作为函数参数:

方式一：数组形式

```
int TwoDimArraySum1(int twoDimAr[][COL], int row, int col);
```

方式二：指针形式，prArray是一个指向包含COL个int的数组的指针

```
int TwoDimArraySum2(int (*prArray)[COL], int row, int col);
```

方式三：指针形式，pr是一个指向int的指针

```
int TwoDimArraySum3(int *pr, int row, int col);
```

方式四：变长数组（C99开始支持）

```
int TwoDimArraySum4(int row, int col, int twoDimAr[row][col]);
```

Pointer

字符指针变量 VS 字符数组

1. **字符数组**由若干个元素组成，每个元素中放一个字符，而**字符指针**变量存放的是地址（字符串第1个字符的地址），决不是将字符串放到字符指针变量中。

2. 赋值方式。可以对字符指针变量赋值，但不能对数组名赋值。

```
char *a;  
a = "I love China!"; //合法赋给a的不是字符串而是第一个元素的地址
```

```
char str[14];  
str = "I love China"; //非法！数组名是地址不是常量，不能被赋值
```

3. 初始化含义。

```
char *a;  
a = "I love China!"; //把字符串第一个元素的地址赋给a
```

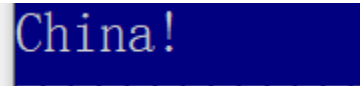
```
char str[14] = "I love China!"; //定义字符数组str，并把字符串赋值给数组中各个元素
```

Pointer

4. 存储单元的内容。编译时为字符数组分配若干存储单元，以存放各元素的值，而对字符指针变量，只分配了一个存储单元。

5. 指针变量的值是可以改变的，而字符数组名代表一个固定的值（数组首元素的地址），不能改变。

```
char *a = "I love China!";  
a = a + 7;  
printf("%s", a);
```



China!

6. 字符数组中各元素的值是可以改变的（可以对它们再赋值），但字符指针变量指向的字符串常量中的内容是不可以被取代的（不能对它们再赋值）。

```
char *a = "House";  
a[2] = 'r'; //非法，字符串常量不能改变
```

```
char *p,*q;  
p = "xyz";  
q = p;  
q[1]="Y";  
p = ?
```

Memory management

`#include <stdlib.h>`

function	Description
<code>calloc(int num, int size)</code>	Allocate an array of num elements each with size (in byte)
<code>malloc(int num)</code>	Allocate an array of num bytes and leave them initialized
<code>realloc(void *addr, int newsize)</code>	Re-allocate memory at address with newsize
<code>free(void *addr)</code>	Release a block of memory at address

Memory management

calloc()

```
int *name;
```

```
name = (int*)calloc(100, sizeof(int));
```

空间初始化为0

malloc()

```
float *name;
```

```
name = (float*)malloc(30*sizeof(float));
```

空间随机初始化

realloc()

```
char *name;
```

```
name = (char*)malloc(200*sizeof(char));
```

```
name = (char*)realloc(name, 100*sizeof(char));
```

```
free() free(name);
```

Struct

结构:是一种构造类型,由若干成员组成,每一个成员可以是一个基本数据类型或构造类型

定义:

```
struct name
{
type variable;
type variable;
type variable;
};
```

先声明结构体类型再定义变量名

```
struct name
{
type variable;
type variable;
type variable;
}variable name;
```

在声明类型的同时定义变量

```
struct
{
type variable;
type variable;
type variable;
}variable name;
```

直接定义结构体类型变量,
可以不出现结构体名

Struct

- 引用结构体变量中成员的方式为：
结构体变量名. 成员名
- 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。
- 成员也可以是一个结构体变量。
- 成员名可以与程序中的变量名相同，二者不代表同一对象。
- 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。

Structs

- 结构体数组

定义结构体数组和定义结构体变量的方法相仿，只需说明其为数组即可。
其初始化也与其他类型数组一样

- 指向结构体类型的指针

`struct name *variable name`

✓ 与和数组不同的是，结构名并不是结构的地址，因此获得其地址要在结构名前面加上&运算符。

✓ 利用指针访问成员：

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

`(++p)->n`
`(p++)->n`
`++p ->n`

Structs

- 指向结构体类型的指针
 - 将一个结构体变量的值传递给另一个函数，有3个方法：
 - ✓ 用结构体变量的成员作参数。
 - ✓ 用结构体变量作实参。
 - ✓ 用指向结构体变量（或数组）的指针作实参，将结构体变量（或数组）的地址传给形参。

Union

union（共用体、联合体）是一种数据类型，它能在**同一个内存空间**中储存不同的数据类型（不是同时储存）。

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

- 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

Enumerate

Enum (枚举型)是C中用户定义的数据类型，为整数常量分配名称，便于易于读取和维护的程序

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

- (1) 在C编译中，对枚举元素按常量处理，故称枚举常量。它们不是变量，不能对它们赋值。
- (2) 枚举元素作为常量，它们是有值的，C语言编译按定义时的顺序使它们的值为0, 1, 2 ...
- (3) 枚举值可以用来作判断比较。
- (4) 一个整数不能直接赋给一个枚举变量。

File I/O

ASCII file

.txt .csv

Plain text
(data in characters)



data.txt

Comma Separated Values
(data structured by “,”)

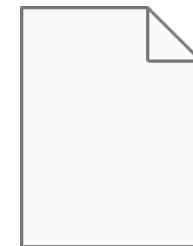


data.csv

binary file

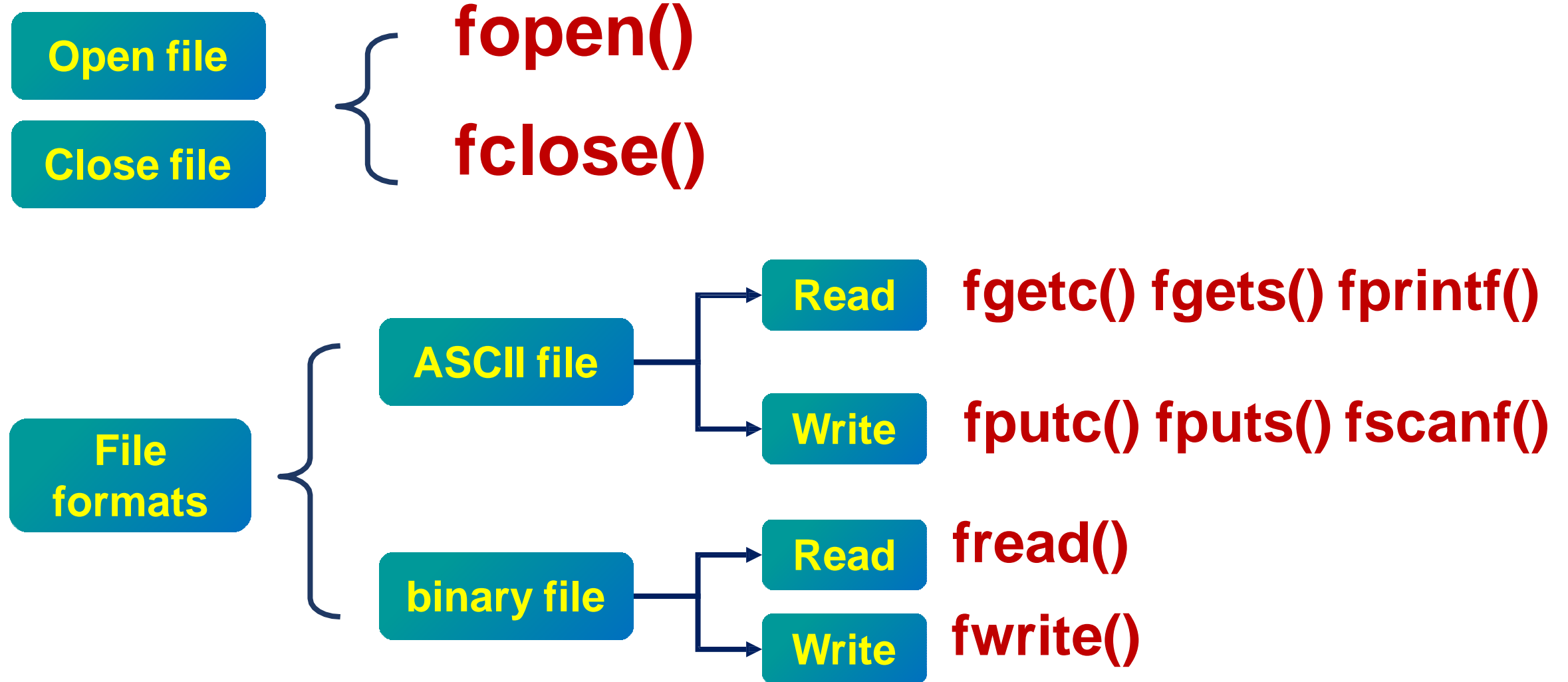
.bin

Binary values
(data in 0 and 1)



data.bin

File I/O



File I/O

Open file

```
FILE *fopen(const char *filename, const char  
*mode);
```



当前.c文件的路径下（相对路径）/系统的绝对路径

To open ASCII files (txt, csv):

"r", "w", "a", "r+", "w+", "a+"

To open binary files:

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

r = read

w = write

a = append

b = binary

如果使用任何一种"**w**"模式（不带x字母）打开一个现有文件，该文件的内容会被删除，以便程序在一个空白文件中开始操作。

File I/O

修正Lecture12 IO P45

csv file

```
#include <stdio.h>

main()
{
    FILE* fp;
    fp = fopen("test.csv", "rb");
    fp = fopen("test.csv", "wb");
    fp = fopen("test.csv", "ab");

    fp = fopen("test.csv", "r+b");
    fp = fopen("test.csv", "w+b");
    fp = fopen("test.csv", "a+b");
}
```

82



csv file

```
#include <stdio.h>

main()
{
    FILE* fp;
    fp = fopen("test.csv", "r");
    fp = fopen("test.csv", "w");
    fp = fopen("test.csv", "a");

    fp = fopen("test.csv", "r+");
    fp = fopen("test.csv", "w+");
    fp = fopen("test.csv", "a+");
}
```


File I/O

Close file

```
int fclose(FILE *stream)
```

Read and write

putchar

getchar

puts

gets

printf

scanf

只能通过标准输入、标准输出、错误输出进行输入/输出

fputc(int c, FILE *fp)

fgetc(FILE *fp)

fputs(const char *s, FILE *fp)

fgets(char *buf, int n, FILE *fp)

fprintf(FILE *fp, const char *format, ...)

fscanf(FILE *fp, const char *format, ...)

通过指定设备进行输入/输出
(stdin, stdout, stderr)

常见错误分析

1. 输入输出的数据的类型与所用格式说明符不一致。
2. 未注意int型数据的数值范围。
3. 在输入语句scanf中忘记使用变量的地址符。
4. 输入数据的形式与要求不符。
5. 误把“=”作为“等于”运算符。
6. 语句后面漏分号或者在不该加分号的地方加了分号。
7. 对应该有花括号的复合语句，忘记加花括号。
8. 引用数组元素时误用了圆括号。
9. 在定义数组时，将定义的“元素个数”误认为是“可使用的最大下标值”。
10. 引用数组时超出数组范围

常见错误分析

12. 对二维或多维数组的定义和引用的方法不对。
13. 误以为数组名代表数组中全部元素。
14. 混淆字符数组与字符指针的区别。
15. switch语句的各分支中漏写break语句。
16. 混淆字符和字符串的表示形式。
17. 使用自加（++）和自减（--）运算符时出的错误。
18. 所调用的函数在调用语句之后才定义，而又在调用前未声明。
19. 对函数声明与函数定义不匹配。
20. 在需要加头文件时没有用#include命令去包含头文件。
21. 误认为形参值的改变会影响实参的值。
22. 函数的实参和形参类型不一致。
23. 不同类型的指针混用。
24. 没有注意函数参数的求值顺序。

常见错误分析

- 25. 混淆数组名与指针变量的区别。
- 26. 混淆结构体类型与结构体变量的区别，对一个结构体类型赋值。
- 27. 使用文件时忘记打开，或打开方式与使用情况不匹配。

