



C 语言编程介绍

潘建瑜

MATH@ECNU 2023

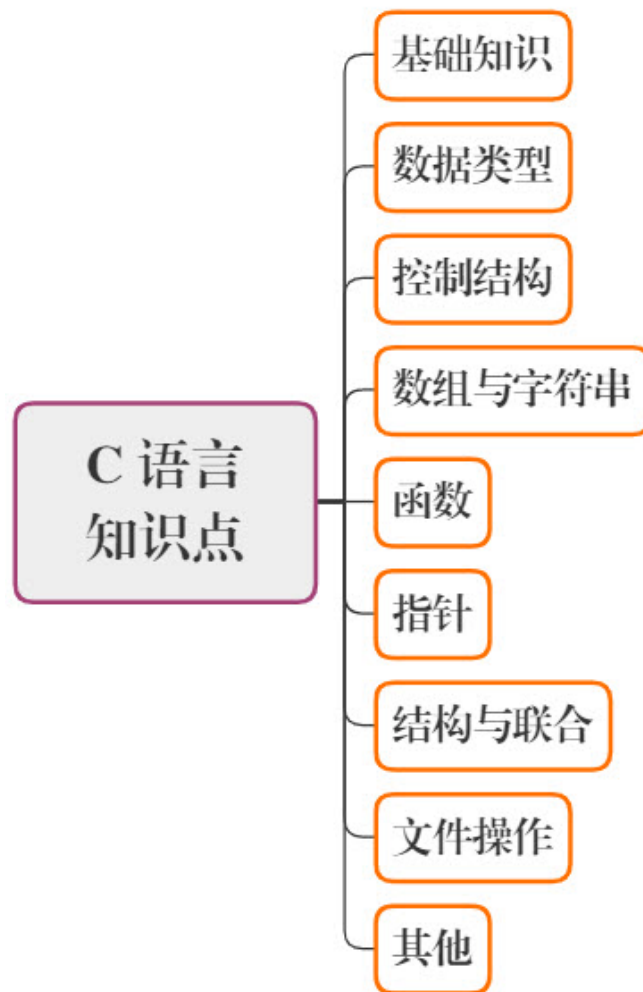


华东师范大学 | 数学科学学院
School of Mathematical Sciences, East China Normal University

目录页

Contents

- 1 C 语言基础
- 2 控制结构
- 3 数组与字符串
- 4 函数
- 5 指针
- 6 文件操作



- ❑ 零基础学 C 语言 (第 4 版), 康莉等, 2019
- ❑ C Primer Plus, 6th, S. Prata, 2013 (中文版, 2016)
- ❑ C语言教程 <https://www.runoob.com/cprogramming/c-tutorial.html>
- ❑ C 语言参考手册 <https://zh.cppreference.com/w/c>



目录
Contents

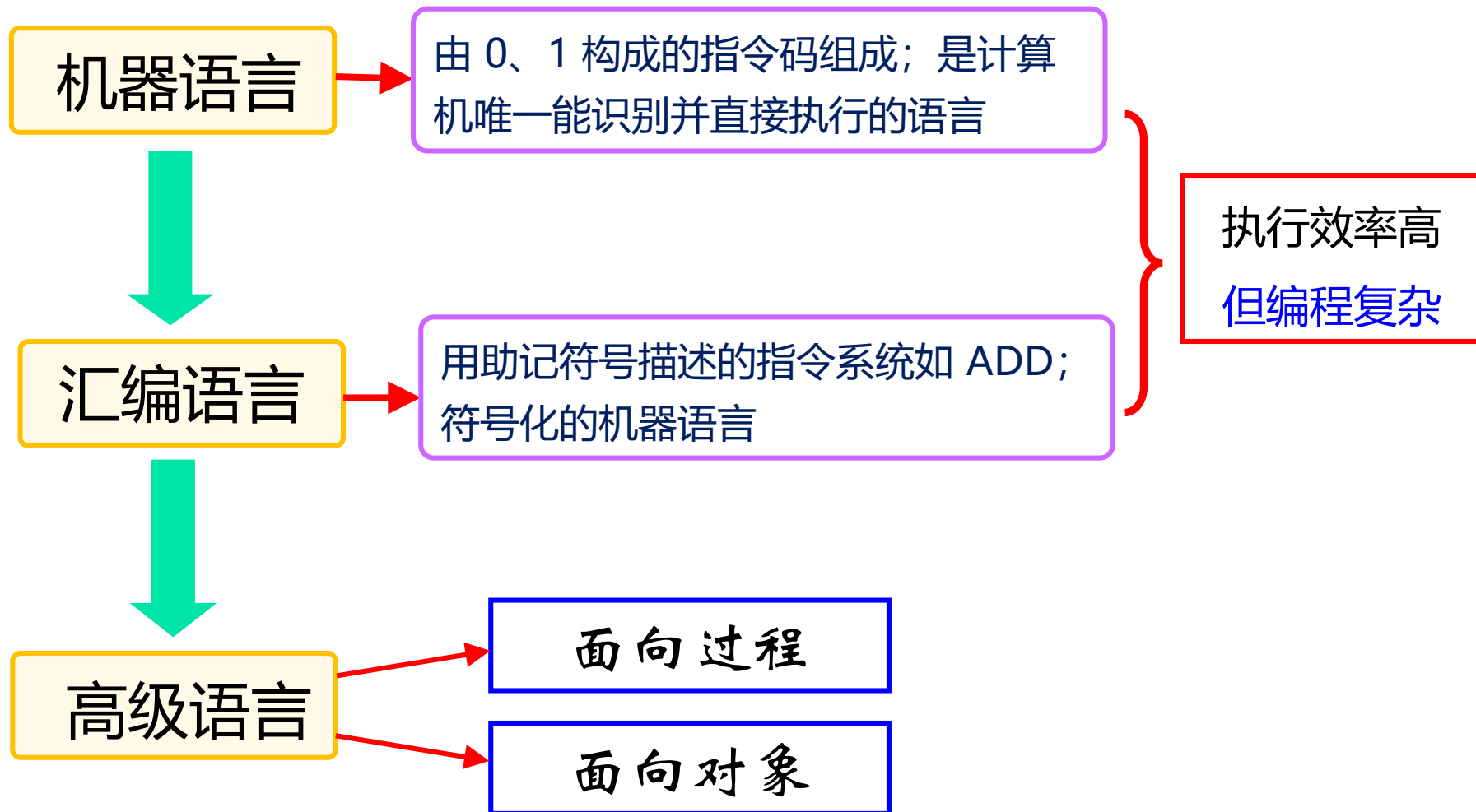
- 1 C 语言基础
- 2 控制结构
- 3 数组与字符串
- 4 函数
- 5 指针
- 6 文件操作

1

C 语言基础

- C 语言介绍
- 数据类型
- 变量、常量与基本运算
- 基本输出输入

程序设计语言的发展



高级语言典型代表

- **FORTRAN**: **F**ormula **T**ranslation

1956年，由 IBM 的 J.W. Backus（哥伦比亚大学数学学士、硕士，图灵奖获得者）带领开发，高级语言诞生的标志，科学计算主流语言。

- **C**

1972年，由贝尔实验室的 D.M. Ritchie（哈佛大学数学博士，图灵奖获得者，UNIX 之父）开发，是一种通用的、过程式的编程语言，高效、灵活、功能丰富，主流的软件开发和科学计算语言。

- **C++**

1983年，由贝尔实验室的 B. Stroustrup 在 C 语言的基础上开发，引入并扩充了面向对象的概念功能。

数学，特别是数学思维是计算机科学的一个支柱。—— B. Stroustrup

C 语言发展

C 语言的历史与发展



- 1972, 贝尔实验室 D. Ritchie 开发
- 1978, B. Kernighan 和 D. Ritchie 《C 程序设计语言》→ K&R 标准
- 1989, ANSI C 标准形成 C89, 1990 年 ISO 发布 C90
- 1999, ISO 正式发布新的标准 C99, 引入一些新特性, 如内联函数
- 2011, C11 标准发布, 添加许多新功能, 同时修改 C99 库的某些部分为可选, 提高与 C++ 的兼容性
- 2017, 发布 C17/C18, 是当前标准, 仅进行技术更正

ANSI - American National Standards Institute / 美国国家标准协会

ISO - International Organization for Standardization / 国际标准化组织

一个简单的 C 程序

C_sum.c

```
/* Example: calculate the sum of a and b */
```

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

预处理：载入头文件

注解语句

```
main()
```

主函数

```
{
```

```
    int a, b, sum;
```

```
    a=10;
```

```
    b=24;
```

```
    sum = a+b;
```

```
    printf("sum=%d\n", sum);
```

打印语句

```
    return 0;
```

```
}
```

C 源程序结构：

- 一个 C 源程序由一个或多个源文件组成
- 每个源文件可由一个或多个函数组成
- 一个源程序有且只能有一个 **main** 函数
- 程序执行从 **main** 开始，在 **main** 中结束
- 源程序中可以有预处理命令，通常应放在源文件或源程序的最前面

C 程序书写规范

书写规范

- 每个说明和语句都必须以分号 “;” 结尾
但预处理命令，函数头和花括号 “}” 之后不能加分号（结构除外）
- 标识符、关键字之间要有间隔，可以是空格或间隔符
- 区分大小写
- 注释： `/*` `*/` 为注释符，不能嵌套
- 常用锯齿形书写格式
- 一行可以写多个语句，一个语句可以分几行

书写规范的程序：

- ▶ `{`、`}` 要对齐
- ▶ 一行写一个语句
- ▶ 一个语句写一行
- ▶ 使用 `TAB` 缩进
- ▶ 有适当的空行
- ▶ 有足够的注释

† 注：所有标点符号必须在英文状态下输入（中文字符串除外）

C 语言编译器

什么是编译器

- 编译器就是将“高级语言”翻译为“机器语言”的程序
- 一个现代编译器的主要工作流程：



常见的 C/C++ 编译器

- **Visual C/C++** 微软，Windows 平台，集成在 Visual Studio 中
- **GNU C/C++** 开源免费，Linux/Unix 平台首选，非常优秀
- **Intel C/C++** Intel 编译器，对自家硬件支持很好
- **Clang C/C++** LLVM 项目中的优秀编译器

IDE (Integrated Development Environment)

IDE (集成开发环境)

- 用于程序开发的应用程序/软件，一般包括代码 **编辑器**、**编译器**、**调试器**和**图形用户界面**等
- 常见的 C/C++ 集成开发环境
 - **Visual Studio** : 微软，大而全，有社区版（免费），支持 clang
 - **Qt Creator** : 跨平台开发环境，为应用程序开发提供一站式解决方案
 - **Code::Blocks** : 开源，全功能跨平台集成开发环境，免费
 - **Dev C++** : 小巧免费，功能简单，适合初学者，
已停止开发（gcc-4.9.2, C11, OpenMP 4.0），Embarcadero Dev-C++
 - **VS Code + MinGW**: 微软免费 IDE + GCC（微软有配置方法指导）

字符集

C 语言字符集

- 字母（大写和小写，共 52 个）
- 数字（0 到 9 共 10 个）
- 空白符（空格符、制表符、换行符）
- 标点和特殊字符

!	#	%	^	&	*	()	[]
{ }	_	+	=	-	~	<	>
/	\	'	"	;	.	,	

† 注：这里的标点符号都是指在英文状态下的标点。

几个术语

- **标识符**：用来标识变量名、函数名、对象名等的字符序列
 - 由字母、数字、下划线组成，第一个字符必须是字母或下划线
 - 区分大小写，不能用关键字
 - 不限制标识符长度，实际长度与编译器有关
 - 命名原则：见名知意、不宜混淆
- **关键字**：具有特定意义的字符串，通常也称为保留字
 - 类型说明符、语句定义符（控制命令）、预处理命令等
- **运算符**（详见后面介绍）
- **分隔符**：逗号、冒号、分号、空格、`()`、`{ }`
- **注释符**：以“`/*`”开头并以“`*/`”结尾，或者 `//`（行注释符）
- **文字/词汇**：直接用字符表示的数据，即常量，如数字、字符串等

代码编写与运行

- 1) 编写源程序，以 `.c` 为扩展名（可使用任何文本编辑器）
- 2) 编译并连接源文件（可以一步完成），生成可执行文件

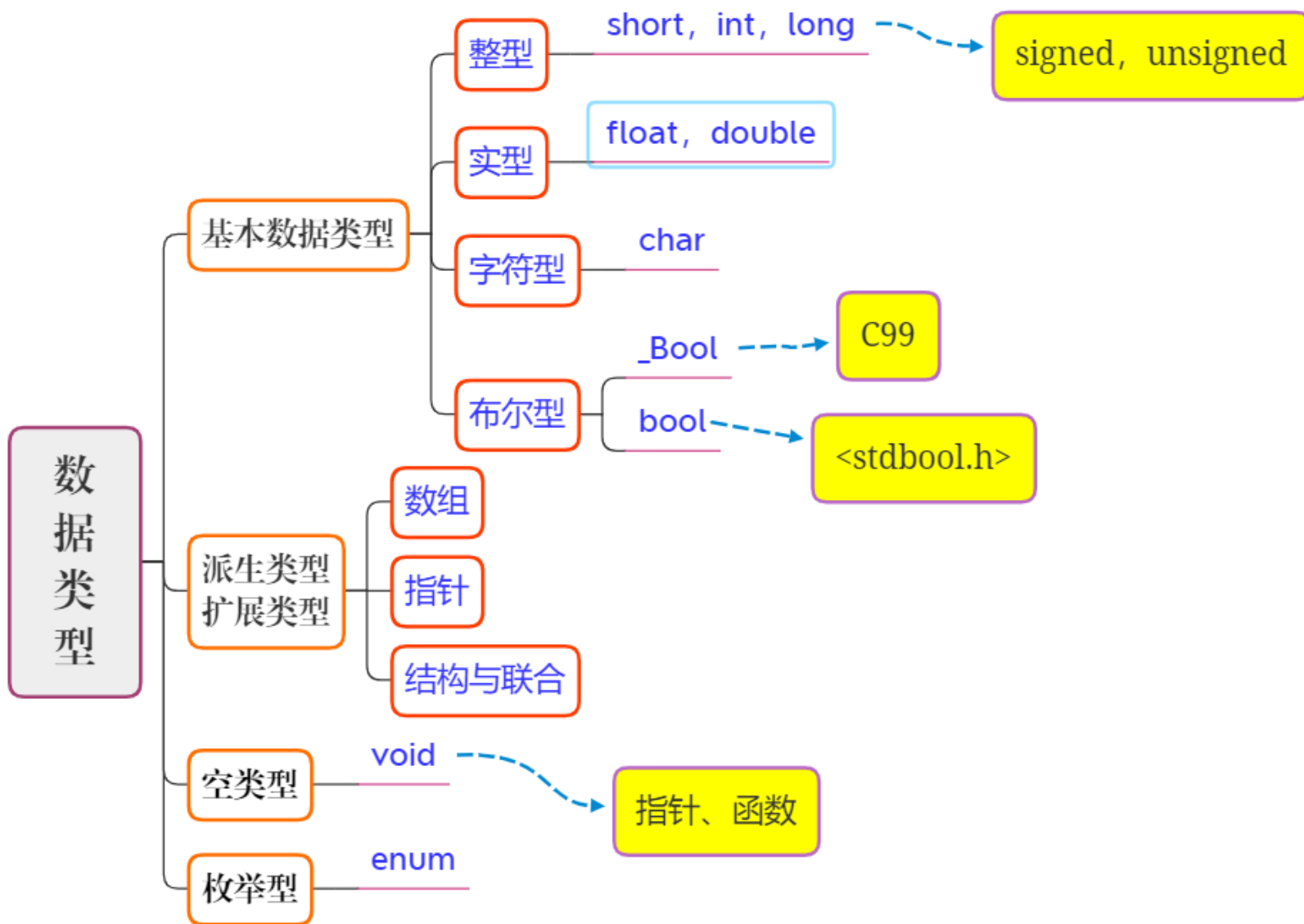
gcc [选项] 源代码文件 // 以 Linux 下 gcc 为例

gcc 常用选项

- `-o` : 指定输出文件名，缺省为 `a.out`
- `-c` : 只编译不链接，即只生产目标文件（`.o` 文件）
- `-Ipath` : 指定或增加包含文件（如 `*.h`）的搜索路径
- `-Lpath` : 指定（增加）库文件的搜索路径
- `-lname`: 与库文件 `libname.a` 链接
- `-O1, -O2, -O3` : 优化开关
- `-g` : 在目标码中加入更多信息，用于程序调试

- 3) 运行生成的可执行文件

数据类型



基本数据类型

C_datatype.c

类型	关键字	所占字节数	表示范围
整型	short	2	$-2^{15} \sim 2^{15} - 1$
	int	2 / 4	$-2^{15} \sim 2^{15} - 1$ / $-2^{31} \sim 2^{31} - 1$
	long	4 / 8	$-2^{31} \sim 2^{31} - 1$ / $-2^{63} \sim 2^{63} - 1$
	unsigned short	2	$0 \sim 2^{16} - 1$
	unsigned int	2/4	$0 \sim 2^{16} - 1$ / $0 \sim 2^{32} - 1$
	unsigned long	4/8	$0 \sim 2^{32} - 1$ / $0 \sim 2^{64} - 1$
实型	float	4 (6-7)	$10^{-38} \sim 10^{38}$
	double	8 (15-16)	$10^{-308} \sim 10^{308}$
	long double	16 (18-19)	$10^{-4932} \sim 10^{4932}$
布尔型	_Bool	1	true, false
字符型	char	1	

- ▶ C 没有规定数据类型字节长度，只规定大小顺序，具体由处理器和编译器决定
- ▶ 更长的整型：long long / unsigned long long

typedef

为一个已有的数据类型另外命名（取别名）

`typedef` 已有数据类型名 新数据类型名

```
typedef _Bool bool;  
bool flag=1;
```

```
typedef float real;  
real x=3.14;
```

`C_datatype_typedef.c`

数据类型转换

自动转换/隐式转换

- 不同类型的数据进行运算，需先转换成同一类型
- 转换按数据长度增加的方向进行，以保证精度不降低
- 所有的浮点运算都是以双精度进行的
- char 型和 short 型参与运算时，先转换成 int 型
- 赋值号两边的数据类型不同时，右边的类型将转换为左边的

char→short→int→long→unsigned long→double←float

```
printf("1/2=%f\n", 1/2);  
printf("1.0/2=%f\n", 1.0/2);
```

C_datatype.c

数据类型转换

强制转换/显式转换

(类型说明符)表达式 // 将表达式的**值**转换成指定的类型

C_datatype.c

```
printf("1/2=%f\n", 1/2);  
printf("1.0/2=%f\n", 1.0/2);  
printf("(double)1/2=%f\n", (double)(1)/2);  
printf("(double)(1/2)=%f\n", (double)(1/2));
```

转换规则

- 浮点型转整型：直接丢掉小数部分
- 字符型转整型：为字符的 ASCII 码
- 整型转字符型：ASCII 码对应的字符

变量

变量 用于存储数据，值可以改变

- ▶ **变量名**：要求与标识符相同
- ▶ **变量类型**：整型、实型、字符型、布尔型
- ▶ 变量必须先声明，后使用

变量的声明

数据类型 变量名列表；

- **变量初始化**：声明变量时直接赋值

```
int i=3, j=10;  
double pi=3.14159;  
char c='0';
```

常量（常数、字符串）

常量 在程序运行中值不能改变的量

- ▶ 整型常量：整数，后面加 *l* 或 **L** 表示长整型，后面加 *u* 或 **U** 表示无符号整型
- ▶ 实型常量：缺省为双精度，后面加 *f* 或 **F** 表示单精度，加 *l* 或 **L** 表示 long double
- ▶ 字符型常量：用单引号括起来的单个字符和转义字符
- ▶ 字符串常量：用双引号括起来的字符序列
- ▶ 布尔常量：true 和 false 

需加头文件：#include <stdbool.h>

```
123, -456, 123456789L, 123456789U;  
1.2, 1.2F, 1.2L, 1.2e8, 1.2e8F, 1.2e-8L  
'M', 'A', 'T', 'H', '?', '$'  
"MATH@ECNU"
```

常量（符号常量）

符号常量 用标识符表示常量，即值不能改变的变量

符号常量的声明

const 数据类型 标识符=常量值;

```
const float PI=3.1415926;
```

- ✦ 符号常量在声明时必须初始化
- ✦ 符号常量的值在程序中不能被修改（不能重新赋值）

运算符

- 算术运算符：+、-、*、/、%、++ (自增)、-- (自减)
- 赋值运算符：
=、+=、-=、*=、/=、%=、&=、|=、^=、>>=、<<=
- 逗号运算符：, (把若干表达式组合成一个表达式)
- 求字节数运算符：sizeof (计算数据类型所占的字节数)

- 关系运算符：用于比较运算，>、<、==、>=、<=、!=
- 逻辑运算符：用于逻辑运算，&&、||、!
- 条件运算符：是一个三目运算符，用于条件求值 (? :)
- 指针运算符：* (取内容)、& (取地址)
- 位运算符：按二进制位进行运算
&、|、^ (异或)、~ (取反)、<< (左移)、>> (右移)

赋值运算

标准赋值运算

变量 = 表达式

x=3;

x=y=3;

y=3;

x=y;

✦ 这种方式不能用于变量初始化!

复合赋值运算

+=、-=、*=、/=、%=

x+=3;

x=x+3;

x/=3;

x=x/3;

```
int a, b, c, d, e;  
a = 5;  
b = a + 3;  
a = a + (c=6);  
d = e = f = a;  
e *= d;  
f /= c - 2;
```

自增自减运算

自增自减（后置和前置）

变量++、变量--
++变量、--变量

- ▶ 前置：先自增自减，然后使用
- ▶ 后置：先使用，然后自增自减

x++;



x=x+1;

++x;



x=x+1;

y=x++*x;



y=x*x; x=x+1;

y=++x*x;



x=x+1; y=x*x;

```
int i=10, j, k;  
j=i++; // j=?  
k=++i; // k=?
```

† 不要在同一语句中包含一个变量的多个 ++ 或 --，因为它们的解释在 C/C++ 标准中没有规定，完全取决于编译器的个人行为。

逗号运算

逗号运算

表达式 1, 表达式 2

- ▶ 先计算 **表达式 1** 的值，再计算 **表达式 2** 的值
- ▶ 将 **表达式 2** 的值作为整个表达式的结果

```
int a=2, b1, b2;  
b1 = 3*a, a+10;    // b1=?  
b2 = (3*a, a+10);  // b2=?
```

† 注意运算的优先级！

运算优先级

高



低

() ++ (后置) -- (后置) 强制类型转换
! ++ (前置) -- (前置) + (正号) - (负号)
* / %
+ -
< <= > >=
== !=
&&
||
赋值 (= += -= *= /= 等)
逗号运算 (,)

[更多详见课程主页](#)

sizeof

sizeof(数据类型)

返回指定 数据类型 的 长度

sizeof(变量名)

返回指定 变量 所占的 字节数

sizeof(表达式)

返回存储 表达式结果 所需的 字节数

```
int a, b, c, d;  
a = sizeof(int);  
b = sizeof(a);  
c = sizeof(3 + 5);  
d = sizeof(3.0L + 5);
```

常用数学函数

需加头文件: `#include <math.h>`

`abs(x)` $\longrightarrow$ $|x|$

`exp(x)` $\longrightarrow$ e^x

`log(x)` $\longrightarrow$ $\ln x$

`log10(x)` $\longrightarrow$ $\lg x$

`sqrt(x)` $\longrightarrow$ $\sqrt{x}$

`pow(x,y)` $\longrightarrow$ x^y

`min(x,y)` $\longrightarrow$ 最小值

`max(x,y)` $\longrightarrow$ 最大值

取整函数: `ceil`, `floor`, `round`, `trunc`

三角函数: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`

双曲三角函数: `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`

语句

- 程序由语句构成，C 语言的语句包括：
 - ▶ 空语句（只有分号）
 - ▶ 声明语句
 - ▶ 表达式语句（赋值表达式）
 - ▶ **复合语句**（将多个语句用 { } 括起来组成的一个语句）
 - ▶ 选择语句
 - ▶ 循环语句
 - ▶ 跳转语句
 - ▶

表达式：运算符连接常量、变量、函数所组成的式子。

格式化输出

加头文件: `#include <stdio.h>`

```
printf("格式控制字符串", 输出变量列表);
```

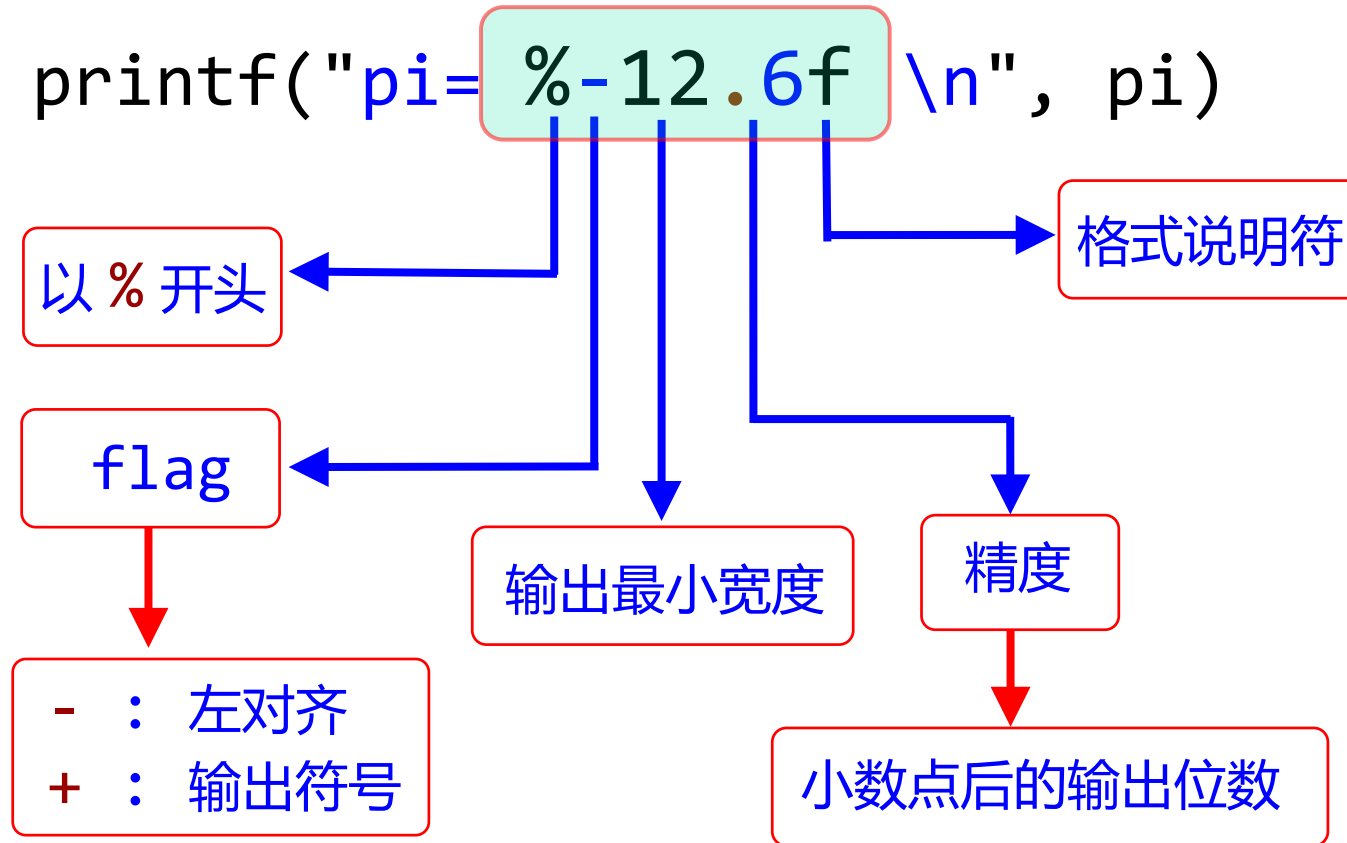
- ▶ 格式控制字符串: 普通字符串、格式字符串、转义字符
- ▶ 普通字符串: 原样输出
- ▶ 格式字符串: 以 `%` 开头, 后面跟各种 格式说明符
- ▶ 转义字符: 实现特殊功能

```
int k=5;  
double a=3.14;  
printf("k=%d, a=%f\n", k, a);    // 一个格式字符串对应一个输出变量
```

格式字符串

`%[flag][输出最小宽度][.精度]格式说明符`

```
printf("pi= %-12.6f \n", pi)
```



格式说明符与转义字符

▶ 常见的格式说明符

c	字符型	g	浮点数（自动）
d	十进制整数	o	八进制
e	浮点数（科学计数法）	s	字符串
f	浮点数（小数形式）	x/X	十六进制

▶ 常见转义字符（输出特殊字符）

\b	退后一格	\t	水平制表符
\f	换页	\\	反斜杠
\n	换行	\"	双引号
\r	回车	%%	百分号

格式化输入

加头文件: `#include <stdio.h>`

`scanf("格式控制字符串", 地址列表);`

`C_printf_scanf.c`

```
int a, b;  
printf("input a and b: ");  
scanf("%d%d", &a, &b); // 一个格式字符串对应一个输入变量地址
```



2

控制结构

- 1 C 语言基础
- 2 控制结构
- 3 数组与字符串
- 4 函数
- 5 指针
- 6 文件操作

- 关系运算与逻辑运算
- 选择结构：IF、SWITCH
- 循环结构：WHILE、FOR
- 循环的终止

关系运算： 比较大小

加头文件: `#include <stdbool.h>`

<code>></code>	大于	<code>>=</code>	大于等于	<code>==</code>	等于
<code><</code>	小于	<code><=</code>	小于等于	<code>!=</code>	不等于

- ▶ 比较大小，结论是 真 则返回 1，否则返回 0
- ▶ C 语言中用 1 表示 true，0 表示 false
- ▶ bool 型变量，赋值 0 时表示 false，其他它值都表示 true

```
bool x1=1.5;    // x1=true
bool x2=0;      // x2=false
bool x3=-11;    // x3=true
```

- † 注意 `==` 与 `=` 的区别
- † 对浮点数进行比较运算时尽量不要使用 `==`

```
pow(sqrt(2.0),2)==2; // true or false?
```

逻辑运算

&&	逻辑与		逻辑或	!	逻辑非
----	-----	--	-----	---	-----

运算法则

运算对象		与	或	非
A	B	$A \& B$	$A B$	$!A$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

逻辑运算： 两点注意

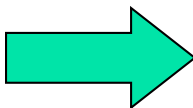
(表达式1) && (表达式2)

- 先计算 表达式1 的值，若是 **true**，则计算 表达式2 的值；
若 表达式1 的值是 **false**，则**不再**计算 表达式2 的值

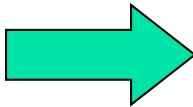
(表达式1) || (表达式2)

- 先计算 表达式1 的值，若是 **false**，则计算 表达式2 的值；
若 表达式1 的值是 **true**，则**不再**计算 表达式2 的值

```
i=3; (i>3) && (i++);
```

 i=3

```
i=3; (i>3) || (i++);
```

 i=4

条件运算符

条件表达式 **?** 表达式1 **:** 表达式2

先计算 条件表达式 的值,

- ▶ 若是 **true**, 则用 表达式 1 作为整个表达式的值,
- ▶ 否则就用 表达式 2 的值作为整个表达式的值

```
y = x > 0 ? 1 : -1;
```

```
z = x > y ? x : y;
```

选择结构：IF

if (表达式) 语句

```
if (表达式)
    语句
else
    语句
```

```
if (表达式) 语句
else if (表达式) 语句
... ..
else if (表达式) 语句
else 语句 // 可以省略
```

几点说明

- ▶ 表达式可以是任意表达式（关系、逻辑、算术等）
- ▶ 语句可以是复合语句（用大括号括起来）
- ▶ 条件判断表达式两边的小括号不能省略
- ▶ else if 中间必须留空
- ▶ if 语句可以嵌套，嵌套时每一层的 if 要与 else 配套，否则要加大括号

选择结构：SWITCH

可以是整型、字符型、枚举型

```
switch (表达式)
{
    case 常量表达式1:
        语句
    case 常量表达式2:
        语句
    ... ..
    case 常量表达式n:
        语句
    default:
        语句
}
```

- ▶ 先计算“表达式”的值，然后依次与每个 **case** 后面的“常量表达式”进行匹配，一旦匹配成功，则开始执行后面的语句，包括后面所有 **case** 以及 **default** 的语句
- ▶ 如果没有匹配的，则执行 **default** 后面的语句
- ▶ 每个 **case** 后面的常量表达式的值不能相同
- ▶ 每个 **case** 分支最后一般需加 **break**
- ▶ 每个 **case** 后面可以有多个语句（复合语句），但可以不用 **{ }**
- ▶ **default** 不是必需的

循环结构：WHILE

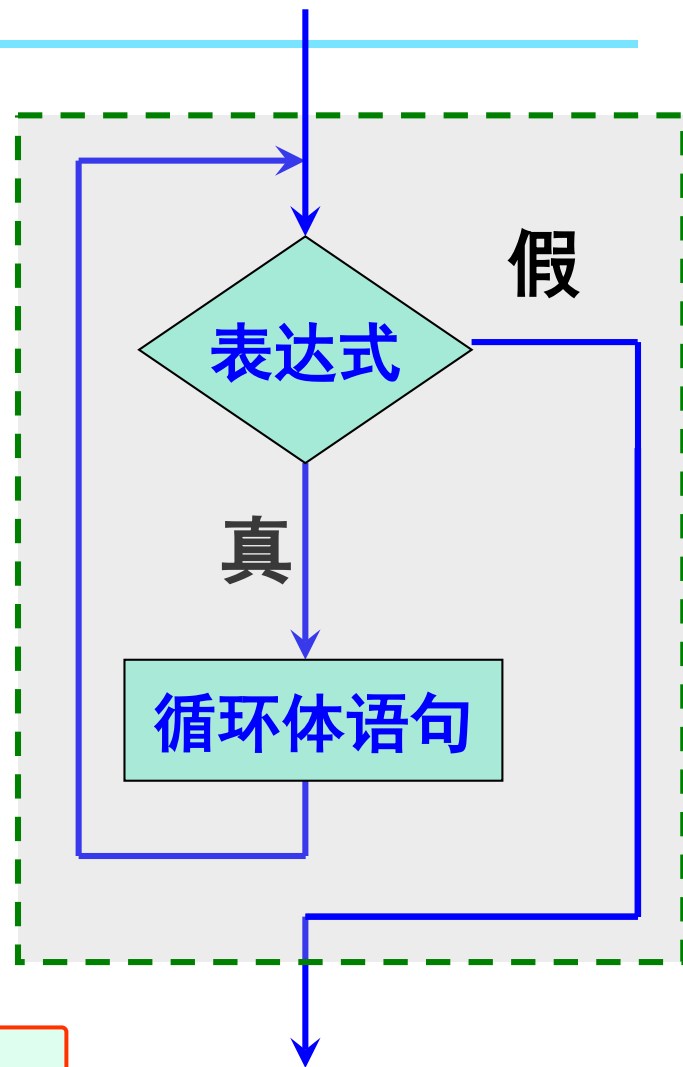
while (条件表达式)

循环体语句

执行过程

- (1) 判断条件表达式的值
- (2) 如果是“真”，则执行循环体语句；
否则退出循环。
- (3) 返回第 (1) 步

✦ 如果循环体语句是复合语句，别忘了大括号！



循环结构：DO WHILE

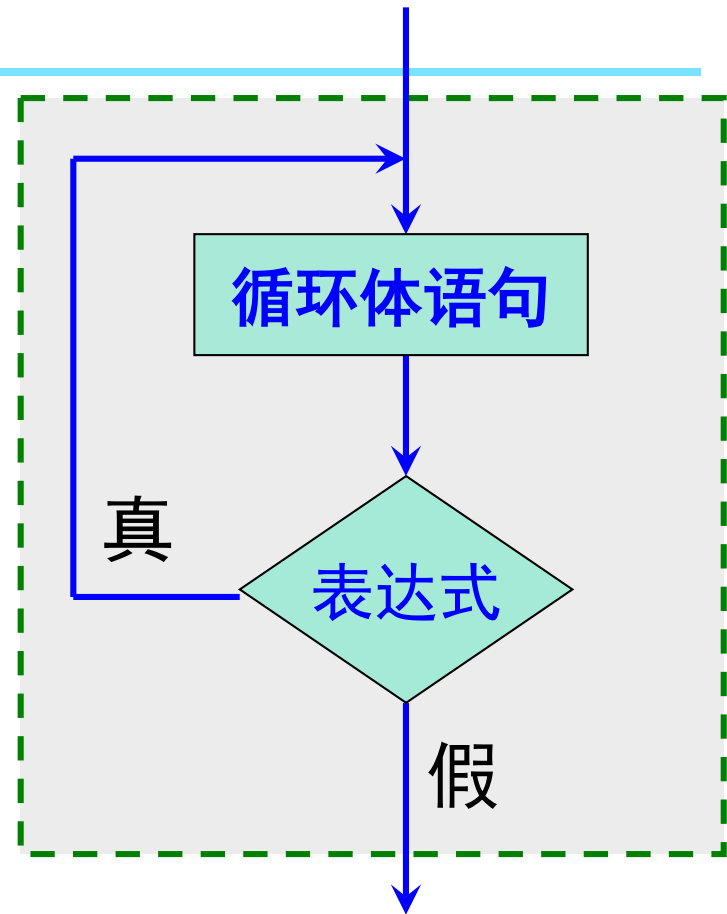
do

循环体语句

while (条件表达式);

执行过程

- (1) 执行循环体语句
- (2) 判断条件表达式的值
- (3) 如果是“真”，返回第(1)步；
否则退出循环。



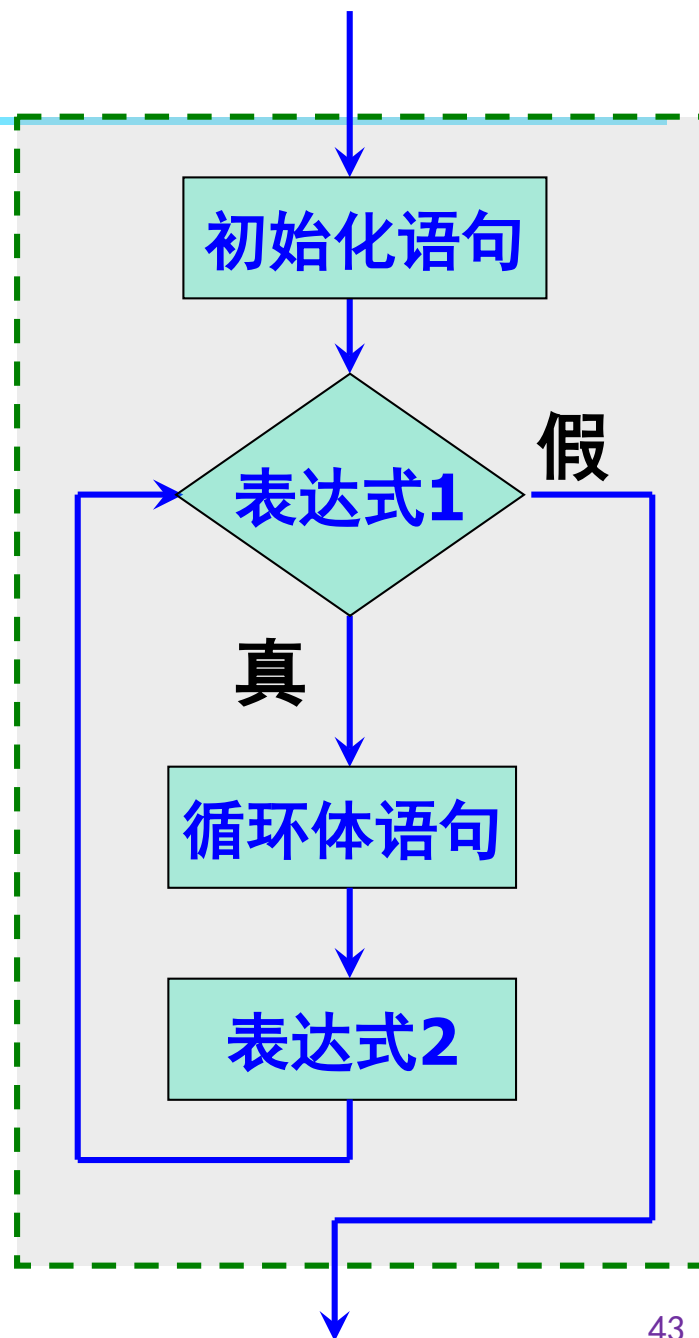
✦ 与 while 循环的区别：无论条件是否成立，循环体语句至少执行一次！

循环结构：FOR

```
for(初始化语句; 表达式1; 表达式2)  
    循环体语句
```

执行过程

- (1) 执行初始化语句
- (2) 计算表达式1 的值，
如果是“真”，则执行循环体语句，
否则退出循环
- (3) 执行表达式2，然后返回第 (2) 步



循环结构：FOR

几点说明

- † 初始化语句，表达式1，表达式2 均可省略，但分号不能省
- † 表达式1 是循环控制语句，如果省略的话就构成死循环
- † 循环体可以是单个语句，也可以是复合语句（大括号！）
- † 初始化语句 与 表达式2 可以是逗号语句

```
s=0;  
for (i=1; i<=10; i++)  
    s=s+i;
```

- 上面程序中的变量 *i* 有时也称为循环变量

for (循环变量赋初值; 循环条件; 循环变量增量)
 循环体语句

循环的终止

● break 语句

break;

- ▶ 跳出循环体，只用在循环语句和 switch 语句中
- ▶ break 只能跳出一层循环

● continue 语句

continue;

- ▶ 结束本轮循环，执行下一轮循环，一般只用在循环语句中

● goto 语句

goto 语句标号;

- ▶ 跳转到由语句标号所指定的语句
- ▶ 语句标号为 标记符后面跟冒号，即 标记符：语句；
- ▶ goto 语句破坏程序的结构性，尽量少用

† break, continue 和 goto 通常是与 if 语句配合使用。



3

数组与字符串

- 1 C 语言基础
- 2 控制结构
- 3 数组与字符串
- 4 函数
- 5 指针
- 6 文件操作

- 一维数组
- 二维数组
- 字符串

数组

数组：具有一定顺序关系的若干相同类型变量的集合体

□ 一维数组的声明

类型标识符 变量名[n]

- 声明一个长度为 n 的数组（向量）
- 类型标识符：数组元素的数据类型；
- n ：数组的长度，即元素的个数；

```
int x[5] // 声明一个长度为 5 的一维数组
```

□ 一维数组的引用

变量名[k] // 注：下标 k 的取值为 0 到 $n-1$ ，下标不能越界

一维数组

类型标识符 变量名[n]

几点注意

- n 可以常数, 如 `int x[5]`, 或者有确定值的正整数 (可变长度数组)
- n 可以用表达式代替, 但表达式的值必须是正整数

example

```
int n=5;  
int x[n]; // 可变长度数组
```

example

```
int m=2, n=3;  
int x[m*n+2];
```

- 只能**逐个**引用数组元素, 而不能一次引用整个数组
- 数组元素在内存中顺序存放, 它们的地址是连续的
- **数组名代表数组存放在内存中的首地址**

例: `x[10]` 在内存中的存放顺序是

<code>x[0]</code>	<code>x[1]</code>	<code>x[2]</code>	<code>x[3]</code>	<code>x[4]</code>	<code>x[5]</code>	<code>x[6]</code>	<code>x[7]</code>	<code>x[8]</code>	<code>x[9]</code>
-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------

一维数组

- 一维数组的初始化：在声明时可以同时赋初值

```
int x[5]={0,2,4,6,8};
```

example

- 也可以只给部分元素赋初值

```
int x[5]={0,2,4}; // 只能依次赋初值
```

example

- 全部元素赋初值时可以不指定数组长度

```
int x[]={0,2,4,6,8};
```

example

† 注意：可变长度数组不能初始化。

C_array_initializing.c

```
int x[5];  
x={1,2,3,4,5}; // ERROR! 只能对数组元素赋值，不能对数组名赋值！
```

二维数组

- 二维数组的声明：

类型标识符 变量名[m][n]

- 声明一个 $m \times n$ 的二维数组（矩阵）

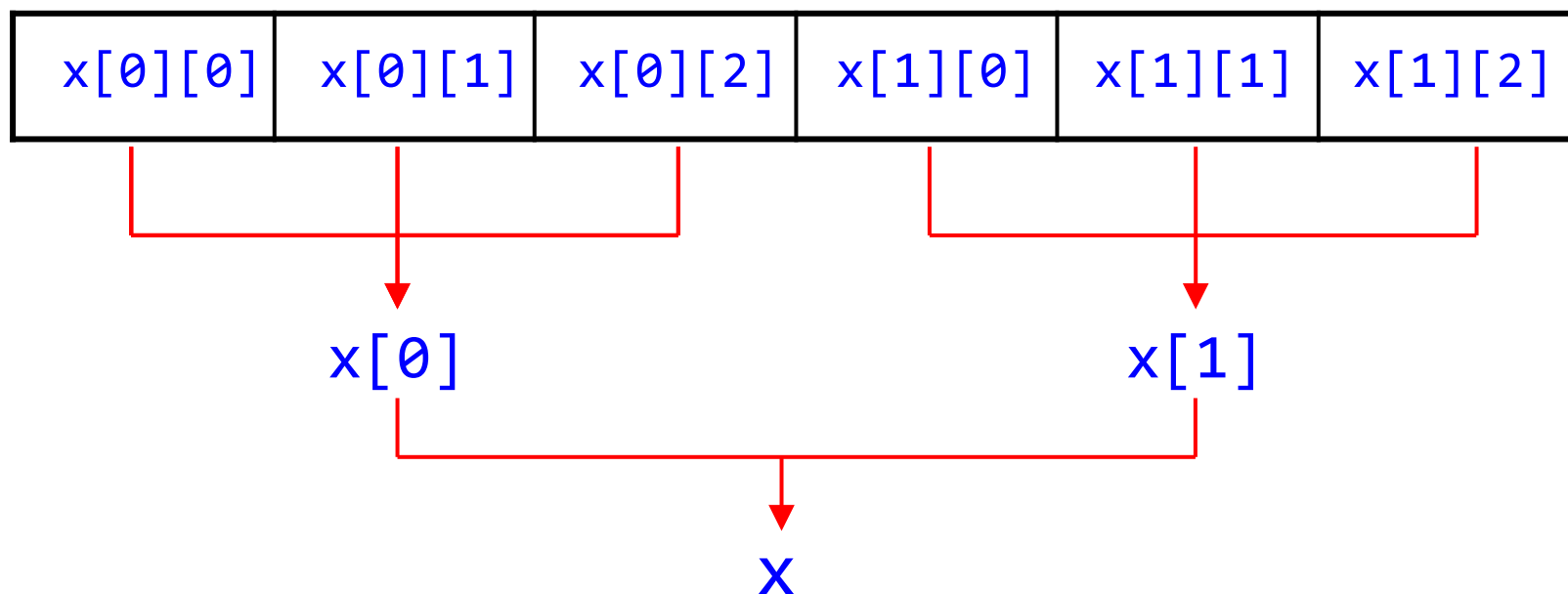
- 二维数组的引用：

变量名[i][j] // 注意下标的取值范围，不要越界！

二维数组

- 二维数组的存储：按行存储

例：x[2][3] 在内存中的存放顺序是



† 在 C 语言中，二维数组被看成是一维数组的数组。

二维数组

● 二维数组的初始化

● 全部初始化

```
int x[2][3]={1,3,5,2,6,10};
```

example

```
int x[][3]={1,3,5,2,6,10}; // 注：只能省第一维的长度！
```

● 分组初始化

example

```
int x[2][3]={{1,3,5}, {2,6,10}};
```

● 部分初始化

example

```
int x[2][3]={{1}, {2,6}};
```

多维数组

- 多维数组的声明：

类型标识符 变量名[n1][n2][n3]...

多维数组的赋值、引用、初始化与二维数组类似。

字符串

► **字符串**：由字符组成的一维数组

□ 与普通数组的区别：

字符串最后面需添加 “\0”，做为 **结束标志**

□ 数组的长度 $\geq$ 字符串长度(所含字符个数)+1

字符串

● 字符串的初始化

```
char str[5]={'m','a','t','h','\0'}; // OK, 只能用于初始化  
char str[5]="math"; // OK, 只能用于初始化  
char str[]="math"; // OK, 只能用于初始化
```

- 使用双引号时，会自动在最后添加结束标志

str →

m	a	t	h	\0
---	---	---	---	----

● 字符串赋值：逐个赋值，循环实现

```
char str[6];  
str = "Math"; // ERROR: 一维数组，不能直接赋值！
```

字符串输入输出

- 字符串的输入

```
scanf("%s", str)  
gets(str)
```

- 字符串的输出

```
printf("%s", str)  
puts(str)
```

字符串操作

● 字符串相关函数（头文件 `string.h` 和 `stdlib.h`）

C_string.c

函数	描述	用法
<code>strlen</code>	求字符串长度	<code>strlen(str)</code>
<code>strcat</code>	字符串连接	<code>strcat(dest,src)</code>
<code>strcpy</code>	字符串复制	<code>strcpy(dest,src)</code>
<code>strcmp</code>	字符串比较	<code>strcmp(str1,str2)</code>
<code>atoi</code>	将字符串转换为整数	<code>atoi(str)</code>
<code>atol</code>	将字符串转换为long	<code>atol(str)</code>
<code>atof</code>	将字符串转换为double	<code>atof(str)</code>

（更多函数可参见 <https://zh.cppreference.com/w/c>）

字符检测

头文件 `ctype.h`

函数	描述	用法
<code>isdigit</code>	是否为数字	<code>isdigit('3')</code>
<code>isalpha</code>	是否为字母	<code>isalpha('a')</code>
<code>isalnum</code>	是否为字母或数字	<code>isalnum('c')</code>
<code>islower</code>	是否为小写	<code>islower('b')</code>
<code>isupper</code>	是否为大写	<code>isupper('B')</code>
<code>isspace</code>	是否为空格	<code>isspace(' ')</code>

<code>tolower</code>	将大写转换为小写	<code>tolower('A')</code>
<code>toupper</code>	将小写转换为大写	<code>toupper('a')</code>

- 更多字符检测函数参见相关资料

† 注：以上检测和转换函数只针对单个字符，而不是字符串！



目录
Contents

- 1 C 语言基础
- 2 控制结构
- 3 数组与字符串
- 4 函数
- 5 指针
- 6 文件操作

4

函数

- 函数的定义与调用
- 作用域、
局部变量与全局变量
- 参数传递
- 嵌套与递归

函数的定义

- 函数是对功能的抽象，是 C 语言的基本模块
- C 语言程序是由函数构成的（一个或多个函数）
- C 语言程序必须有且只能有一个 **main** 函数

● 函数的定义

类型标识符 函数名(形式参数表)

{

语句组

}

函数头

函数体

† 类型标识符指明函数返回值的类型，若没有返回值，可用 **void**

形式参数表

● 形式参数列表

类型标识符 变量, 类型标识符 变量,

- 形式参数（简称 形参）需要指定数据类型
- 有多个形参时，用逗号隔开，每个形参需单独指定数据类型
- 如果函数不带参数，则形参可以省略，但小括号不能省
- 形参只在函数内部有效（局部变量）

```
int my_max(int x, int y) // OK
int my_max(int x, y)    // ERROR
```

● 函数返回值

- 通过 `return` 语句给出，如： `return x`
- 若没有返回值，可以不写，也可以写不带表达式的 `return`

函数的调用

- 函数调用前须先声明

类型标识符 函数名(形式参数表);

- 可以在主调函数中声明，也可以在所有函数之外声明
- 有时也称为函数原型

- 函数的调用方式

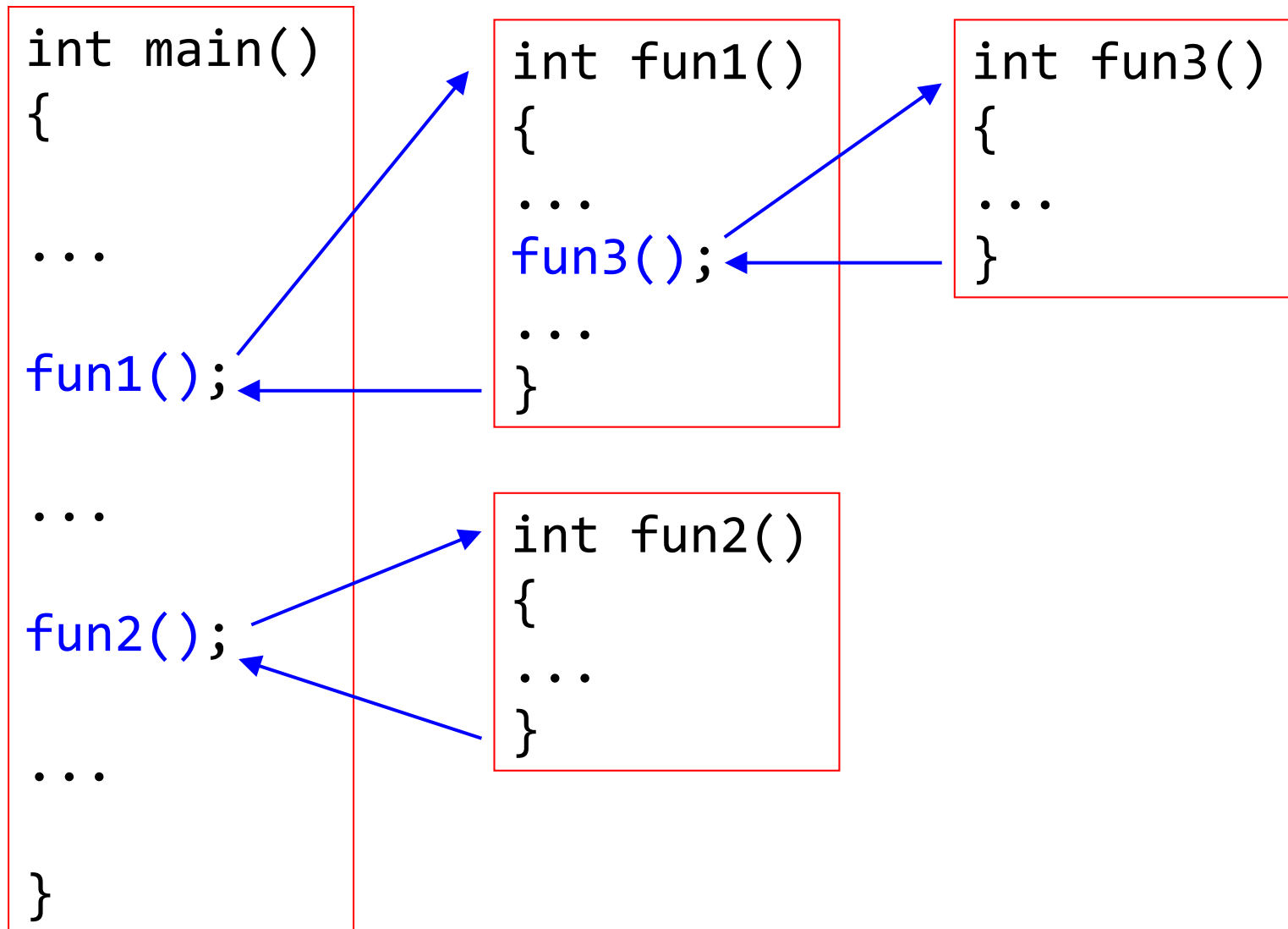
函数名(实参数列表);

- 被调函数可以出现在表达式中，此时必须要有返回值

- 主调函数与被调函数

- 被调函数在主调函数后定义，须在调用前声明
- 被调函数在主调函数前定义，则主调函数中可以直接调用

函数的调用过程



举例：随机数

例：随机数的生成

头文件 `stdlib.h`

```
seed=11;  
srand(seed);    % 设置种子  
x=rand();        % 返回一个随机整数
```

- `rand()`：返回一个 $0 \sim \text{RAND_MAX}$ 之间的伪随机整数
- `srand(seed)`：设置种子。如不设定，默认种子为 1
- 相同的种子对应相同的伪随机整数
- 每次执行 `rand()` 后，种子会自动改变, 但变化规律是固定的

思考

- † 如何生成 $[a, b]$ 之间的随机整数？
- † 如何生成 $[0, 1]$ 之间的随机小数？

`example_rand_01.c`

`example_rand_02.c`

举例：计时函数

例：计时函数：`clock`

头文件 `time.h`

```
#include <time.h>

... ..

clock_t t0, t1;
double totaltime;

... ..

t0 = clock();

... ..

t1 = clock();
totaltime=(double)(t1 - t0) / CLOCKS_PER_SEC;
```

† `clock()`：返回进程启动后所使用的 **cpu** 总滴答数（毫秒）

举例：计时函数

例：计时函数：time

头文件 `time.h`

```
#include <time.h>

... ..

time_t    t0, t1;
t0 = time(NULL); // t0 = time(0)

... ..

t1 = time(NULL);
t = t1 - t0;
```

`example_time_01.c`

`example_time_02.c`

† `time(NULL)`: 返回从 1970 年 1 月 1 日 0 时 0 分 0 秒至今的总秒数

† `time` 以秒为单位

作用域、局部变量与全局变量

- ▶ 数据（通常指变量）的作用域：数据在程序中的有效区域
- ▶ 局部变量：数据只在某个区域有效（可见）
- ▶ 全局变量：数据在整个程序中都有效（可见）

作用域

□ C 语言中常见的作用域

- ▶ 函数原型作用域
- ▶ 函数作用域
- ▶ 语句块作用域（循环体，语句组{ ... } 等）

```
bool Isprime(int m);  
  
int main()  
{  
    int n=111;  
    if (Isprime(n)) printf("%d is a prime\n", n);  
}  
  
bool Isprime(int m)  
{  
    if (m<2) return false;  
    for (int k=2; k<m; k++)  
        if (m%k==0) return false;  
    return true;  
}
```

局部变量与全局变量

局部变量

- 只在某个局部作用域内有效的变量，如：
 - ▶ 函数的形参和函数中定义的变量，只在该函数内有效
 - ▶ `for` 循环初始语句中定义的变量和循环体内定义的变量，只在循环内有效
 - ▶ 语句块中定义的变量只在该语句块中有效

全局变量

- 在整个程序执行期间都有效的变量

全局变量需在所有函数外定义，在它后面定义的函数中均可以使用；
若要在它前面定义的函数中使用该全局变量，则需声明其为外部变量

```
extern 数据类型名 变量名
```

作用域 / 可见域

```
double my_power(double x, int k)
{
    if (k==1) return x;
    else
    {
        double y = 1.0;
        for (int i=1; i<=k; i++)
            y = y * x;
        return y;
    }
}
```

x, k 的作用域

y 的作用域

i 的作用域

同名问题

- ❑ 若局部变量与全局变量同名，则优先使用局部变量！

```
#include <stdio.h>
int k = 2;  // 全局变量
int main()
{
    int i=5, x;  // 局部变量
    x = i+k;
    printf("x=%d\n", x);

    {
        int k=16; // 局部变量
        x = i+k;
        printf("x=%d\n", x);
    }

    x = i+k;
    printf("x=%d\n", x);
}
```

参数传递

- ▶ 主调函数、被调函数
- ▶ 传递方法一：值传递
- ▶ 传递方法二：地址传递

参数传递

传递方式一： 值传递

- ▶ **形参**是局部变量，在函数被调用时才分配存储单元，调用结束即被释放
- ▶ **实参**可以是常量、变量、表达式、函数(名)等，但它们必须要有确定的值，以便**把具体的值传送给形参**
- ▶ 实参和形参在**数量、类型、顺序**上应严格一致
- ▶ 传递时是将实参的值传递给对应的形参，即**单向传递**
- ▶ 形参获得实参传递过来的值后，便与实参脱离关系，即此后**形参的值的改变不会影响实参的值**

举例

```
int main()
{
    ... ..
    x = 3.0; n = 2;
    y = my_power(x, n);
    ... ..
}
```

```
double my_power(double x, int k)
{
    ... ..
}
```



数组与函数

- 将数组中的某个元素传递给被调函数：值传递

example

```
void my_swap(int a, int b)
{
    int t;
    t = a; a = b; b = t;
}

int main()
{
    int x[2]={1,3};
    my_swap(x[0], x[1]);
}
```

数组名作为函数的参数

□ 将整个数组传递给被调函数

传递方式二：地址传递

- ▶ 基本想法：为了节省资源和开销，不再另外分配存储空间，而是直接**将实参数组所在的内存地址告诉被调函数**，让被调函数直接作用在实参数组上（即传递的是数组的**首地址**）
- ▶ 实现方法：将数组名作为参数，形参和实参都是数组名，类型一样

† 由于被调函数是直接作用到实参数组上的，即实参与形参代表的是同一个数组，因此在函数中**对形参数组的任何修改都会影响到实参数组！**

地址传递

C_array_swap.c

```
void my_swap(int a[], int b[], int n)
{
    int t, i;
    for (i=0; i<n; i++)
    { t=a[i]; a[i]=b[i]; b[i]=t; }
}

int main()
{
    int x[3]={1,2,3}, y[3]={2,4,6};
    my_swap(x,y,n);
}
```

用数组作形参，一定要加中括号！
但可以指定第一维的大小

✦ 数组作为形参时一般不指定大小（长度），此时通常需要加一个参数，用来传递实参数组的大小。

二维数组传递

例：计算矩阵各列的和

C_array_sum.c

```
#include <stdio.h>
const int m=3, n=4; // 常量，矩阵维数
void sum_col(double A[][n], double s[])
{
    int i, j;
    for(j=0; j<n; j++) s[j]=0.0; // 赋初值
    for(j=0; j<n; j++)
        for(i=0; i<m; i++) s[j] = s[j] + A[i][j]; // 求和
}

int main()
{
    double H[m][n], s[n];
    for(int i=0; i<m; i++)
        for(int j=0; j<n; j++) H[i][j]=1.0/(i+j+1);
    sum_col(H, s);
    printf("s[0]=%f, s[n-1]=%f\n", s[0], s[n-1]);
    return 0;
}
```

只能省略第一维的大小

数组名作为参数总结

- ❑ 在定义函数时，须指定数组（形参）的大小（常量表达式）

```
void sum_col(double A[10][10], double s[10])
```

可以省略第 1 维的大小

```
void sum_col(double A[][10], double s[])
```

若数组的大小中含有变量，则必须是常量（全局）

```
const int n=10;  
void sum_col(double A[][n], double s[])
```

- ❑ 函数调用时，只需输入数组名（实参）即可

```
sum_col(A, s)
```

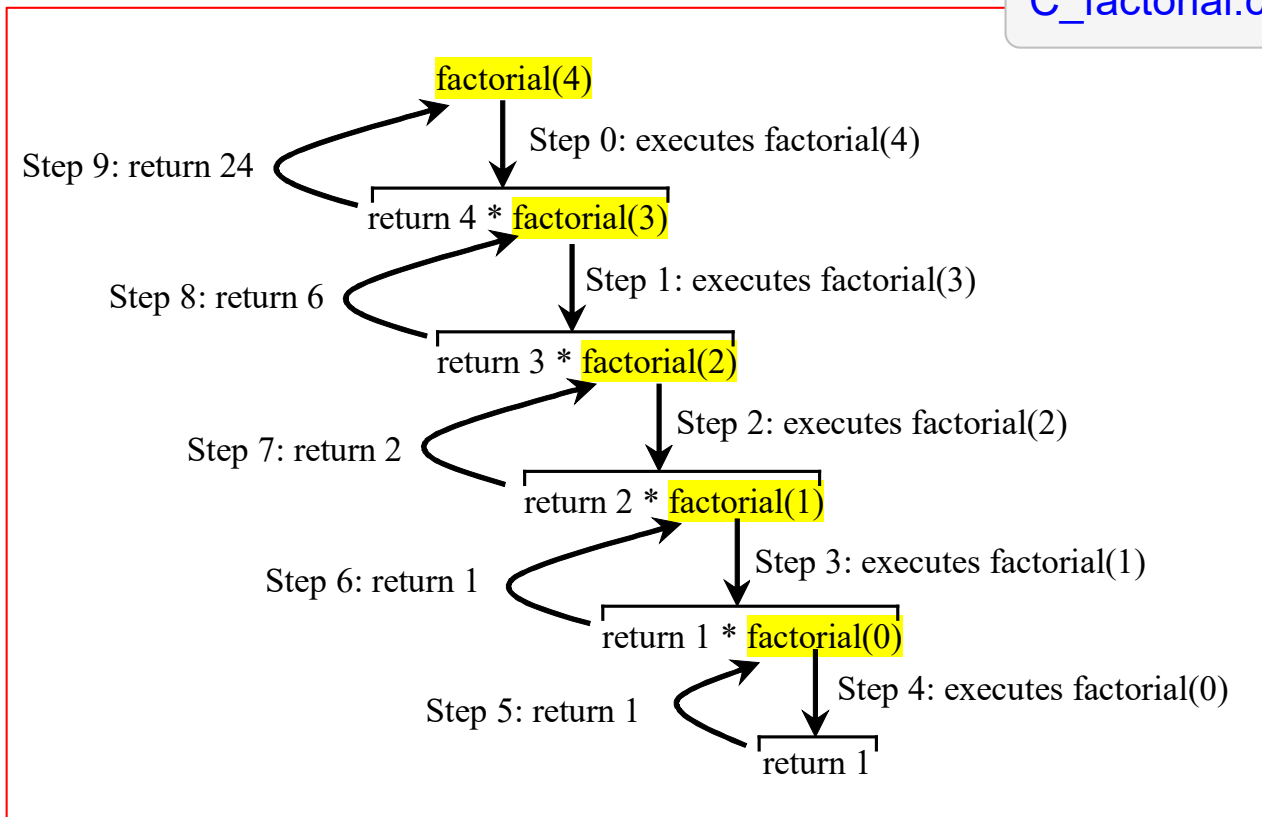
函数嵌套与递归

- ▶ 函数可以嵌套调用，但不能嵌套定义
- ▶ 函数也可以递归调用（函数可以直接或间接调用自己）

函数的嵌套

例：计算阶乘
$$n! = \begin{cases} 1 & (n = 0) \\ n(n-1)! & (n > 0) \end{cases}$$

C_factorial.c



† 对同一个函数的多次不同调用中，系统会给函数的形参和局部变量分配不同的存储空间，它们互不影响。

内联函数

□ 内联函数的声明与使用

- ▶ 定义方法：与普通函数一样，只需加关键字 `inline`
- ▶ 与普通函数的区别：编译时在调用处用函数体直接进行替换
- ▶ 优点：能节省参数传递、控制转移等开销，从而提高代码执行效率

`C_inline.c` // 编译时加 `-O2`

```
inline double f(double x) // 内联函数
{
    return 2*x*x - 1; // f(x) = 2x^2 - 1
}
```

- ✦ 内联函数通过应该功能简单、规模小、使用频繁
- ✦ 内联函数体内不建议使用循环语句和 `switch` 语句
- ✦ 有些函数无法定义成内联函数，如递归调用函数等



5 指针

- 1 C 语言基础
- 2 控制结构
- 3 数组与字符串
- 4 函数
- 5 指针
- 6 文件操作

- 指针基本用法
- 指针与常量
- 指针与数组
- 持久动态内存分配

指针

- ▶ 指针能使 C 语言象汇编语言一样处理内存地址，提高效率
- ▶ 运用指针可以很方便地使用数组和字符串
- ▶ 运用指针编程是 C 语言最主要的风格之一，也是困难的一部分

指针定义

什么是指针

- 指针变量，简称指针，用来存放其它变量的内存地址

● 指针的定义

类型标识符 * 指针变量名

- 声明一个指针类型的变量
- 类型标识符表示该指针可指向的对象的数据类型，即该指针所代表的内存单元所存放的数据的类型

Tips: 变量为什么要声明？

1) 分配内存空间; 2) 限定变量能参与的运算及运算规则。

Tips: 内存空间的访问方式: 1) 变量名; 2) 内存地址，即指针。

指针运算

指针的两个基本运算

- 提取变量的内存地址：&
- 引用指针所指向的变量：*

● 地址运算符：&

&变量名 // 提取变量在内存中的存放地址

example

```
int x=3;  
int * px; // 定义指针 px  
px=&x;    // 将 x 的地址赋给指针 px
```

- 此时，我们通常称 `px` 是指向 `x` 的指针
- 注意：指针的类型必须与其指向的对象的类型一致

指针运算

● 指针运算符：*

*指针变量 // 引用指针变量所指向的对象

```
int x;  
int * px; // 声明指针变量，星号后面可以有空格！  
px=&x;  
*px = 3;  // 等价于 x = 3，星号后面不能有空格！
```

† 在使用指针时，我们通常关心的是指针指向的元素！

```
int x = 3;  
int * px = &x; // 初始化，指针的值只能是某个变量的地址
```

Example: 指针的初始化

指针赋值

● 指针可能的取值

有效指针的三种取值

- (1) 一个对象的地址；
- (2) 指向某个对象后面的对象；
- (3) 值为 0 或 NULL（空指针）。

- ✦ 没有初始化或赋值的指针是无效的指针；
- ✦ 引用无效指针会带来难以预料的问题！

指针赋值：只能用以下四种类型的值

- (1) 0 或者值为 0 的常量，表示空指针；
- (2) 类型匹配的对象地址；
- (3) 同类型的另一有效指针；
- (4) 类型匹配的对象的下个地址（相对位置）。

```
int x=3;  
int * px=&x;  
int * py=&x+1;
```

Example

```
int * pi;  
pi=0; // OK  
pi=NULL; // OK
```

指针与常量

● 指向常量的指针

`const` 类型标识符 * 指针名

```
const int a = 3;  
int * pa = &a; // ERROR  
const int * cpa = &a; // OK
```

- 指向 `const` 对象（常量）的指针必须用 `const` 声明！
- 这里的 `const` 限定了指针所指对象的属性，不是指针本身的属性！

Example

```
const int a = 3;  
int b = 5;  
const int * cpa = &a; // OK  
*cpa = 5; // ERROR  
cpa = &b; // OK  
*cpa = 9; // ERROR  
b = 9; // OK
```

指向常量的指针所指对象的值并不一定不能修改！

- 允许把非 `const` 对象的地址赋给指向 `const` 的指针；
- 但不允许使用指向 `const` 的指针来修改它所指向的对象的值！

指针与常量

- 常量指针，简称常指针

类型标识符 * const 指针名

- 常量指针：指针本身的值不能修改

Example

```
int a = 3, b = 5;  
int * const pa = &a; // OK  
pa = &b; // ERROR
```

- 指向 const 对象的 const 指针

const 类型标识符 * const 指针名

- 指针本身的值不能修改，也不能通过它来修改其指向的对象。

指针算术运算

指针可以和整数或整型变量进行加减运算，且运算规则与指针的类型有关！

Example

```
int * pa; int k;  
pa + k    // pa 所指的当前位置之后第 k 个元素的地址  
pa - k    // pa 所指的当前位置之前第 k 个元素的地址
```

- 在指针上加上或减去一个整型数值 k ，等效于获得一个新指针，该指针指向原来的元素之后或之前的第 k 个元素
- 指针的算术运算通常是与数组的使用相联系的
- 一个指针可以加上或减去 0，其值不变

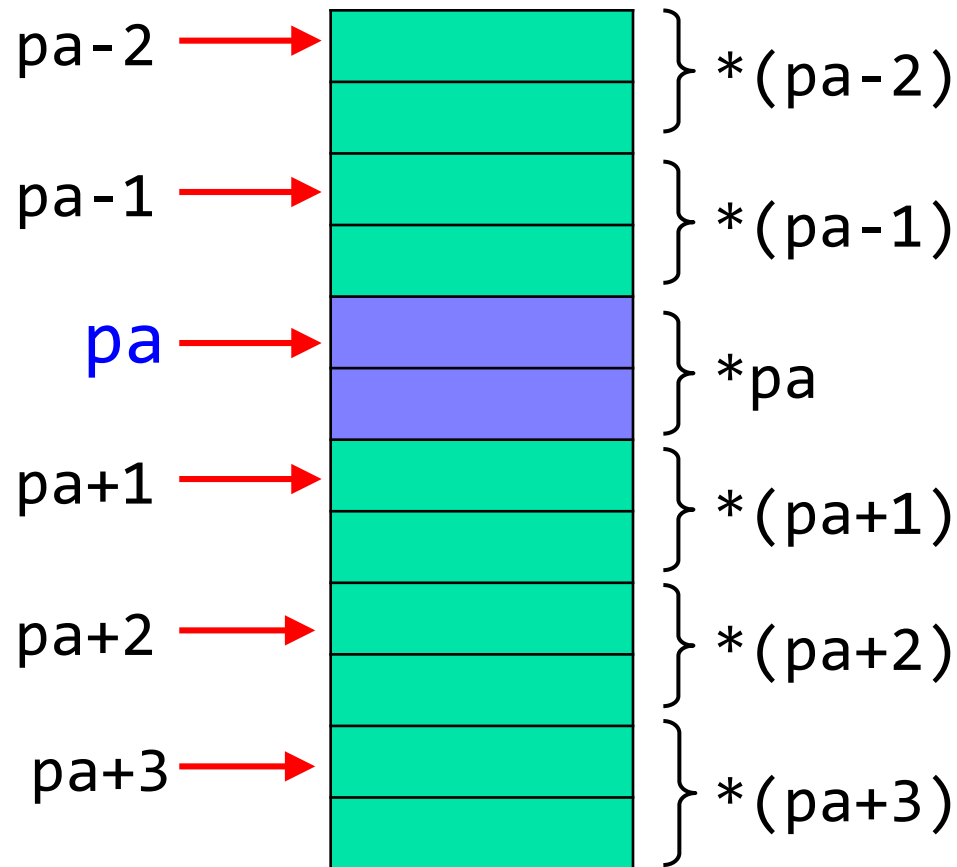
Example

```
int * pa; int k;  
pa++     // pa 所指的当前位置之后的元素的地址  
pa--     // pa 所指的当前位置之前的元素的地址
```

指针算术运算

```
short * pa
```

```
// 每个元素占两个字节
```



指针与数组

C 语言中，指针与数组密切相关：由于数组元素在内存中是连续存放的，因此使用指针可以非常方便地处理数组元素！

Example

```
int a[]={0,2,4,8};  
int * pa;  
pa = a; // OK  
pa = &a[0]; // OK, 与上式等价  
*pa = 3; // OK, 等价于 a[0]=3  
*(pa+2) = 5; // OK, 等价于 a[2]=5  
*(a+2) = 5; // OK, 等价于 a[2]=5
```

- 在 C 语言中，数组名就是数组的首地址！
- 当数组名出现在表达式中时，会自动转化成指向第一个数组元素的指针！

一维数组与指针

- 在 C 语言中，引用一维数组元素有以下三种方式：

- (1) 数组名与下标，如：a[0]
- (2) 数组名与指针运算，如：*(a+1)
- (3) 指针，如：int * pa=a; *pa

example

```
int a[]={0,2,4,8};  
int * pa = a;  
*pa = 1;    // 等价于 a[0]=1  
*(pa+2) = 5;    // 等价于 a[2]=5  
*(a+2) = 5;    // OK, 等价于 a[2]=5  
*(pa++) = 3;    // OK, 等价于 a[0]=3; pa = pa+1;  
*(a++) = 3;    // ERROR! a代表数组首地址，是常量指针！  
*(pa+1) = 10;   // 思考：修改了哪个元素的值？
```

- 指针的值可以随时改变，即可以指向不同的元素；
- 数组名等价于常量指针，值不能改变。

举例

例：使用三种方法输出一个数组的所有元素

```
// 第一种方式：数组名与下标
```

```
for (int i=0; i<n; i++)  
    printf("%d,", a[i]);
```

```
// 第二种方式：数组名与指针运算
```

```
for (int i=0; i<n; i++)  
    printf("%d,", *(a+i));
```

```
// 第三种方式：指针
```

```
for (int * pa=a; pa<a+n; pa++)  
    printf("%d,", *pa);
```

```
// 第三种方式：指针
```

```
for (int * pa=a, i=0; i<n; i++)  
    printf("%d,", pa[i]); ;
```

若pa是指针，k是整型数值，则

$*(pa+k)$ 可以写成 $pa[k]$

$*(pa-k)$ 可以写成 $pa[-k]$

C_pointer_array.c

一维数组与指针

一维数组 $a[n]$ 与指针 $pa=a$

- 数组名 a 是地址常量，数组名 a 与 $\&a[0]$ 等价；
- $a+i$ 是 $a[i]$ 的地址， $a[i]$ 与 $*(a+i)$ 等价；
- 数组元素的下标访问方式也是按地址进行的；
- 可以通过指针 pa 访问数组的任何元素，且更加灵活；
- $pa++$ 或 $++pa$ 合法，但 $a++$ 不合法；
- $*(pa+i)$ 与 $pa[i]$ 等价，表示第 $i+1$ 的元素；

$a[i] \quad \Leftrightarrow \quad pa[i] \quad \Leftrightarrow \quad *(pa+i) \quad \Leftrightarrow \quad *(a+i)$

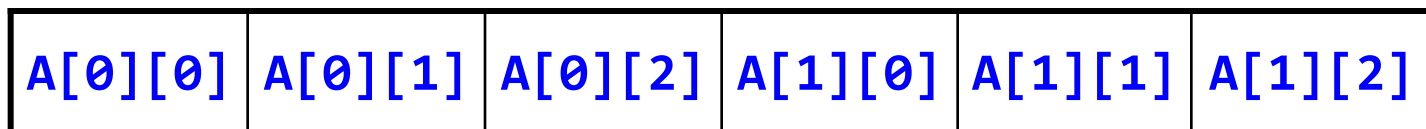
二维数组

C语言中，二维数组是按行顺序存放在内存中的，可以理解为一维数组组成的一维数组。

例： `int A[2][3]={ {1,2,3}, {7,8,9} };`

可以理解为：

A	[	A[0]	—	A ₀₀	A ₀₁	A ₀₂
		A[1]	—	A ₁₀	A ₁₁	A ₁₂



↑
A[0]

(第一行的首地址)

↑
A[1]

(第二行的首地址)

† A[0], A[1] 有时也称为行数组。

二维数组与指针

对于二维数组 A ，虽然 A 、 $A[0]$ 都是数组首地址，但二者指向的对象不同：

$A[0]$ 是一维数组的名字，它指向的是行数组 $A[0]$ 的首元素，即 $A[0][0]$

$*A[0] \longleftrightarrow A[0][0]$

$*(A[0]+1) \longleftrightarrow A[0][1]$

而 A 是一个二维数组的名字，它指向的是它的首元素，即 $A[0]$

$*A \longleftrightarrow A[0]$

$*(A+1) \longleftrightarrow A[1]$

二维数组与指针

Example

```
int A[2][3]={{1,2,3},{7,8,9}};  
int * p = A; // ERROR!  
int * p = A[0]; // OK
```

- 当 `int * p` 声明指针时，`p` 指向的是一个 `int` 型数据，而不是一个地址，因此，用 `A[0]` 对 `p` 赋值是正确的，而用 `A` 对 `p` 赋值是错误的！
- 如何用普通指针引用二维数组的元素？

设指针 `p=&A[0][0]`，则 $A[i][j] \iff *(p+n*i+j)$

C_pointer_array2D.c

指针作为函数参数

指针作为函数参数

- 以**地址**方式传递数据。
- 形参是指针时，实参可以是指针或地址。

```
void split(double x, int * n, double * f)
```

example

```
double x, x2;  
int x1;  
split(x, &x1, &x2)
```

✦ 当函数间需要传递大量数据时，开销会很大。此时，如果数据是**连续存放**的，则可以只传递数据的**首地址**，这样就可以减小开销，提高执行效率！

指针作为函数参数

指针作为函数参数的作用

- 使形参和实参指向共同的内存地址；
- 减小函数间数据传递的开销；
- 可以传递函数代码的首地址（后面介绍）。

Tips:

如果在被调函数中不需要改变指针所指向的对象的值，则可以将形参中的指针声明为**指向常量的指针**，以保护原始数据。

指针型函数

当函数的返回值是地址时，该函数就是指针型函数。

- 指针型函数的定义

```
数据类型 * 函数名(形参列表)
{
    函数体
}
```

持久动态内存分配

若在程序运行之前，不能够确切知道数组中元素的个数，如果声明为很大的数组，则可能造成浪费，如果声明为小数组，则可能不够用。此时需要动态分配空间，做到按需分配。

- 持久动态内存申请：**malloc、calloc、realloc**
- 持久动态内存释放：**free**

申请内存空间

```
px=(数据类型 *)malloc(size);
```

```
px=(数据类型 *)calloc(size);
```

- ▶ 在 **堆** 上申请大小为 size (字节数) 的内存空间, 若申请成功, 则返回该内存空间的首地址, 并赋值给**指针 px**;
- ▶ **calloc** 自动用零初始化, **calloc** 不初始化
- ▶ 若申请不成功, 则返回 0 或 NULL
- ▶ 如果是在初始化语句中使用, 可以省略 (数据类型 *)

```
int *px=malloc(100*sizeof(int));
```

```
free(px); // 释放由 malloc 申请的内存空间
```

✦ 持久动态内存必须手工释放, 否则可能会造成内存泄漏!

持久动态数组举例：矩阵运算

C_malloc.c

```
int main()
{
    const int N=500;
    float *A = malloc(sizeof(float)*N*N);
    float *B = malloc(sizeof(float)*N*N);
    float *C = calloc(sizeof(float)*N*N);

    ... ..
}
```

`px=(数据类型 *)realloc(px,newsized); // 调整大小`



目录
Contents

- 1 C 语言基础
- 2 控制结构
- 3 数组与字符串
- 4 函数
- 5 指针
- 6 文件操作

6

文件操作

- 文件打开与关闭
- 文本文件的读写
- 二进制文件的读写
- 编译预处理与条件编译

打开文件

① 声明文件指针

```
FILE *pf;
```

② 打开文件

```
pf=fopen(文件名, 打开方式);
```

▶ 文件名：普通字符串

▶ 打开方式：读 or 写，文本文件 or 二进制文件

```
rt、wt、at、rb、wb、ab、  rt+、wt+、at+、rb+、wb+、ab+
```

(r 为读, w 为写, + 为读写, t 为文本, b 为二进制)

✦ 若打开成功, 则返回指向文件的指针, 否则返回一个空指针 (NULL)

③ 读、写文件

④ 关闭文件

```
fclose(pf); // status=fclose(pf);
```

读写文本文件

● 写文本文件

```
fprintf(pf, "格式控制字符串", 输出变量列表);
```

用法与 printf 类似

```
FILE *pf;  
pf=fopen("out1.txt","wt");  
double pi=3.1415926;  
fprintf(pf,"pi=%-12.6f\n", pi);  
fclose(pf);
```

C_fprintf.c

● 读文本文件

```
fscanf(pf, "格式控制字符串", 地址列表);
```

```
fscanf(fp,"%d,%f", &i, &t);
```

读写二进制文件

● 写二进制文件

```
fwrite(buffer, size, count, pf);
```

将 `count` 个长度为 `size` 的连续数据写入到 `pf` 指向的文件中，`buffer` 是这些数据的首地址（可以是指针或数组名）

● 读二进制文件

```
fread(buffer, size, count, pf);
```

从 `pf` 指向的文件中读取 `count` 个长度为 `size` 的连续数据，`buffer` 是存放这些数据的首地址（可以是指针或数组名）

二进制文件举例

C_fwrite_fread.c

```
int A[3][3]={11,12,13},{21,22,23},{31,32,33}};
```

```
FILE *pf;
```

```
pf=fopen("data1.dat","wb");
```

```
fwrite(A,sizeof(int),9,pf);
```

```
fclose(pf);
```

```
int B[3][3];
```

```
pf=fopen("data1.dat","rb");
```

```
fread(B,sizeof(int),9,pf);
```

```
fclose(pf);
```



```
fwrite(A,sizeof(A),1,pf);
```

编译预处理

- 加入头文件
- 定义宏
- 条件编译

编译预处理

● 加入头文件

```
#include <文件名> // 按标准方式导入头文件,  
                  // 即在系统目录中寻找指定的文件  
#include “文件名” // 先搜索当前目录, 然后再搜索系统目录
```

● 宏定义

```
#define PI 3.14159 // 定义符号常量  
#undef PI // 删除由 #define 定义的符号常量  
#define S 3.1415926*r*r // 可以是表达式  
#define MAX(a,b) (a>b)?a:b // 带参数的宏  
#define false 0  
#define true 1
```

条件编译

`#ifdef` 标识符

程序正文 // 当 “标识符” 已由 `#define` 定义时编译

`#else`

程序正文 // 否则编译这段程序

`#endif`

`#ifndef` 标识符

程序正文 // 当 “标识符” 没有定义时编译

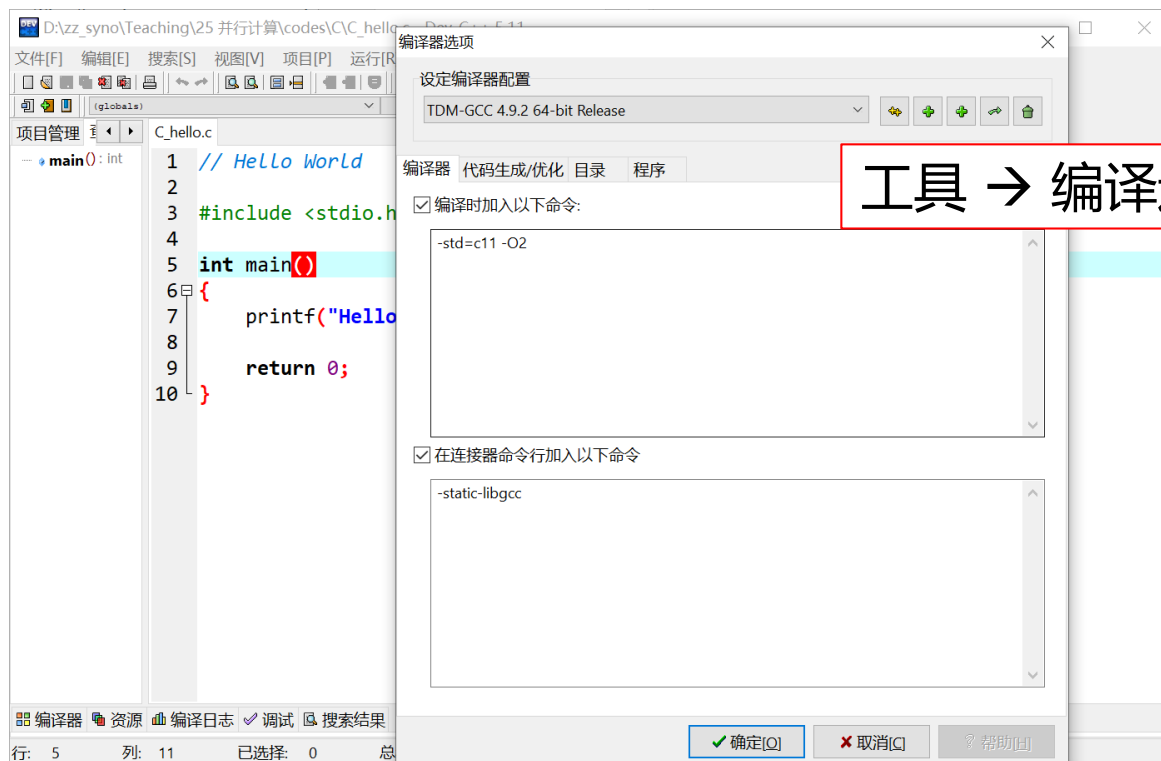
`#else`

程序正文 // 否则编译这段程序

`#endif`

编译建议

- ▶ 加选项 `-std=c11 -O2`
- ▶ Windows (Dev-C++) 缺省生成的可执行文件与源代码（主文件）同名（后缀名为 `.exe`）；
Linux 下则为 `a.out`，建议加 `-o` 选项





谢谢

Programming