

《C 语言程序设计》笔记^{*}

潘建瑜

华东师范大学 • 数学科学学院

2023 年 3 月 3 日

^{*}本笔记仅供课堂教学使用

目 录

第一讲 编程基础	1
1.1 C 语言概述	1
1.1.1 C 语言源程序结构	1
1.1.2 C 语言源程序书写规范	1
1.1.3 程序编译	2
1.2 C 语言基础知识	2
1.2.1 C 语言字符集, 标识符, 关键字	2
1.2.2 C 语言数据类型	3
1.2.3 变量与常量	5
1.2.4 运算与表达式	6
1.2.5 常用数学函数	8
1.3 C 语言格式化输入输出	8
1.3.1 C 语言格式化输出	8
1.3.2 C 语言格式化输入	9
第二讲 选择与循环	10
2.1 关系运算与逻辑运算	10
2.2 选择结构	10
2.2.1 IF 语句	10
2.2.2 SWITCH 结构	11
2.3 循环结构	12
2.3.1 WHILE 循环	12
2.3.2 DO WHILE 循环	12
2.3.3 FOR 循环	13
2.3.4 循环的非正常终止	13
第三讲 数组	14
3.1 一维数组	14
3.2 二维数组	15
3.3 字符串	15
第四讲 函数	18
4.1 函数的声明、定义与调用	18
4.2 作用域、局部变量与全局变量	21
4.3 函数间的参数传递	22
4.4 函数嵌套与递归	23
4.5 内联函数	24
第五讲 指针	25
5.1 指针的定义与运算	25

5.2	指针与一维数组	27
5.3	指针与二维数组	28
5.4	指针与函数	29
5.5	持久动态内存分配	30
5.6	应用: 矩阵乘积的快速算法	31
5.6.1	矩阵乘积的普通方法	31
5.6.2	Strassen 方法	32
5.7	应用: Gauss 消去法求解线性方程组	34
5.7.1	Gauss 消去过程	35
5.7.2	选主元 Gauss 消去法	36
5.8	上机练习	36
第六讲	文件读写操作	38
6.1	文件的打开和关闭	38
6.2	文本文件的读写	38
6.2.1	二进制文件的读写	39
6.2.2	其他文件操作	40
第七讲	预编译处理与多文件结构	42
7.1	预编译处理	42
7.2	多文件结构	43
7.3	编译选项	43





第一讲 编程基础

1.1 C 语言概述

C 语言是一门广泛使用的面向过程高级程序设计语言, 其语法丰富, 且灵活高效, 是当前高性能科学计算和大型软件开发的常用编程语言.

- 1972 年, 贝尔实验室 D. Ritchie 开发;
- 1978 年, B. Kernighan 和 D. Ritchie 《C 程序设计语言》, 被称为 K&R 标准;
- 1989 年, 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 年, 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 年, 发布 C17/C18, 是当前标准, 仅进行技术更正

1.1.1 C 语言源程序结构

```
1 #include <stdio.h> // 预处理命令, 载入头文件
2 int main() // 主函数
3 {
4     printf("Hello Math!\n"); // 标准输出
5     return 0;
6 }
```

- 一个 C 语言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, 即主函数;
- 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 (以 “#” 开头), 通常应放在源文件或源程序的最前面;

1.1.2 C 语言源程序书写规范

- 每条语句以分号 “;” 结尾, 但预处理命令, 函数头和右花括号 “}” 之后不需要加分号 (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 (如运算符、括号等);
- C 语言区分大小写;
- 注释符: `/* */` (整段注释, 称为段注释符), 当前的大部分 C 语言编译器也支持行注释符 `//` (单行注释)
- 一行可以写多个语句, 一个语句也可以分几行书写;
- 常用锯齿形书写格式.
- 注意: 所有标点符号必须在英文状态下输入!



书写规范的程序建议:

- 花括号 { } 要对齐;
- 一行写一个语句, 一个语句写一行;

- 使用 TAB 缩进, 有合适的空行, 提升程序的可读性;
- 有足够的注释。

1.1.3 程序编译

- 编译器: 将“高级语言”翻译为“机器语言”的工具;
- 一个现代编译器的主要工作流程:

源代码 $\xrightarrow{\text{编译}}$ 目标代码 $\xrightarrow{\text{链接}}$ 可执行程序

- 常见的 C/C++ 语言编译器: Visual C++, GNU C++, Intel C++, Clang C/C++ 等等
- IDE (Integrated Development Environment 集成开发环境): 用于程序开发的应用软件, 一般包括代码编辑器、编译器、调试器和图形用户界面等, 常用的 IDE 有
 - Visual Studio: 由微软开发, 适用 Windows 平台, 大而全, 有社区版 (免费), 支持 clang.
 - Dev C++: 小巧, 免费, Windows 平台上的 gcc, 非常适合学习 C/C++.
 - Code::Blocks: 开放源码的全功能跨平台集成开发环境, 免费;
 - 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
 - VS Code + MinGW: 微软免费 IDE + GCC (微软有配置方法指导)

1.2 C 语言基础知识

1.2.1 C 语言字符集, 标识符, 关键字

- 合法的字符集有
 - (1) 字母: 包括大写和小写, 共 52 个;
 - (2) 数字: 0 到 9 共 10 个;
 - (3) 空白符: 空格符、制表符、换行符;
 - (4) 标点符号和特殊字符:

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

- C 语言标识符: 用来标识变量名、函数名、对象名等的字符序列。
 - (1) 由字母、数字、下划线组成, 第一个字符必须是字母或下划线;
 - (2) 区分大小写, 不能用关键字;
 - (3) C 语言不限制标识符长度, 实际长度与编译器有关, 建议不要超过 32 个字符;
 - (4) 命名原则: 见名知意, 不宜混淆。
- 分隔符: 逗号、冒号、分号、空格、()、{ }
- 注释符: 段注释符, 以 “/*” 开头并以 “*/” 结尾, 另外, 当前大部分编译器也支持行注释符 “//”
- C 语言关键字: 具有特定意义的字符串, 也称为保留字, 包括: 类型标识符、语句定义符 (控制命令)、预处理命令等;



表 1.1. C 语言关键字

auto	break	case	char	const (C90)	continue
default	do	double	else	enum (C90)	extern
float	for	goto	if	inline (C99)	int
long	register	restrict	return	short	signed (C90)
sizeof	static	struct	switch	typedef	union
unsigned	void	volatile (C90)	while		
_Alignas	_Alignof	_Bool	_Complex	_Generic	_Imaginary
_Noreturn	_Static_assert	_Thread_local			


1.2.2 C 语言数据类型

C 语言的数据类型可分为基本数据类型和派生（扩展）数据类型。

- 基本数据类型: 整型, 实型, 字符型 (`char`) 和布尔型 (`_Bool`)
 - 整型: `int`, `short`, `long`, `unsigned int`, `unsigned short`, `unsigned long`
 - 实型: `float`, `double`

表 1.2. C 语言基本数据类型

数据类型	关键字	所占字节	表示范围
整型	<code>short</code>	2	$-2^{15} \sim 2^{15} - 1$
	<code>int</code>	2 / 4	$-2^{15} \sim 2^{15} - 1 / -2^{31} \sim 2^{31} - 1$
	<code>long</code>	4 / 8	$-2^{31} \sim 2^{31} - 1 / -2^{63} \sim 2^{63} - 1$
	<code>unsigned short</code>	2	$0 \sim 2^{16} - 1$
	<code>unsigned int</code>	2 / 4	$0 \sim 2^{16} - 1 / 0 \sim 2^{32} - 1$
	<code>unsigned long</code>	4 / 8	$0 \sim 2^{32} - 1 / 0 \sim 2^{64} - 1$
实型	<code>float</code>	4 (6 位有效数字)	$10^{-38} \sim 10^{38}$
	<code>double</code>	8 (15 位有效数字)	$10^{-308} \sim 10^{308}$
布尔型	<code>_Bool</code>	1	0, 1
字符型	<code>char</code>	1	

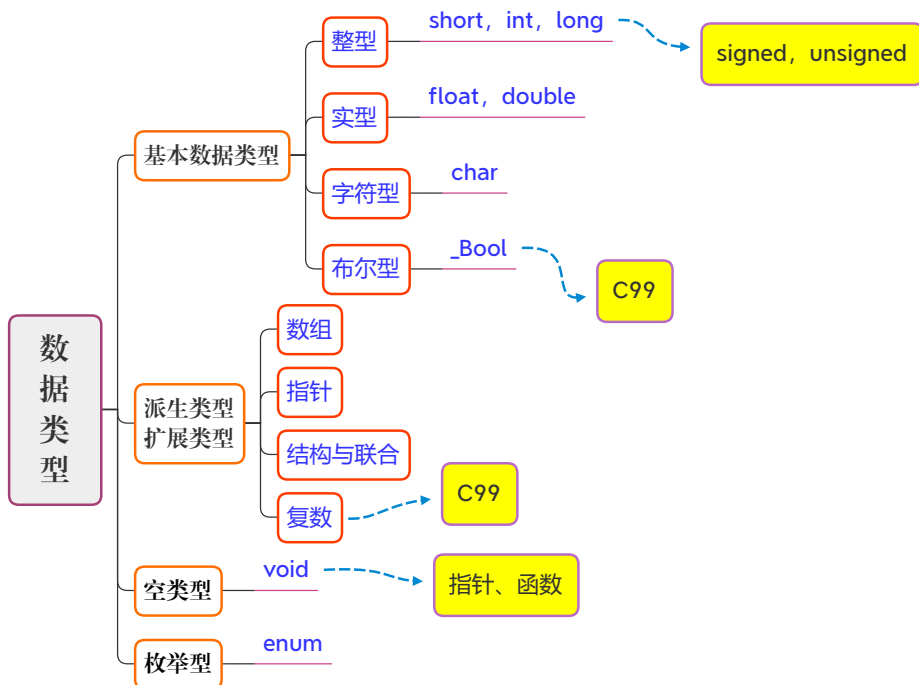
 **注记:** 事实上, C 语言标准没有规定每种数据类型的具体字节数和表示范围, 只规定大小顺序, 即长度满足下面的关系式

`char <= short <= int <= long <= long long`

具体长度由编译器决定。

- 派生（扩展、导出、自定义）数据类型: 数组, 指针, 枚举, 结构, 联合, 复数, 等等





- 为一个已有的数据类型另外命名: **typedef**

typedef 已有数据类型名 新数据类型名

例 1.1 给已有数据类型取别名.

(C_datatype_typedef.c)

```

1 #include <stdio.h>
2
3 typedef _Bool bool;
4 typedef float real;
5
6 main()
7 {
8     bool flag=1;
9     real x=3.14;
10
11     printf("flag=%d, x=%f\n", flag, x);
12
13     flag=-1;
14     x=2.78;
15     printf("flag=%d, x=%f\n", flag, x);
16
17     return 0;
18 }
  
```

- 数据类型的转换

(1) 自动转换/隐式转换:

- 相同类型的数据进行运算时, 其结果仍然是同一数据类型, 如 $3/2$ 的结果是 **1**;
- 不同类型的数据进行运算时, 需要先转换成同一数据类型, 然后再进行相应的运算;



- 转换按数据长度增加的方向进行, 以尽可能地保证精度不会降低;
- 所有的浮点运算都是以双精度进行的;
- 赋值号两边数据的类型不同时, 需先将右边表达式的值转换为左边数据的类型, 然后再赋值;
- `char` 型数据和 `short` 型数据进行运算时, 需转换成 `int` 型;
- 字符变量直接参与算术运算时, 先转化位相应的 ASCII 码, 然后进行运算.

```
char --> short --> int --> unsigned --> long
--> unsigned long --> double <-- float
```


(2) 强制转换/显式转换

(类型标识符)表达式 // 将表达式的值转化为指定的数据类型

例 1.2 (类型转换) 已知有下面的代码

```
1  int a=2, b=5;
2  double x, y, z;
3
4  x = b/a;           // x = 2.0
5  y = (double)b/a;  // y = 2.5
6  z = (double)(b/a); // z = 2.0
```

- (1) 由于 `a` 和 `b` 都是整型, 因此表达式 `b/a` 的值也是整型, 即 `b/a=2`. 将其赋值给 `double` 型变量 `x`, 因此 `x=2.0`.
- (2) 在计算 `(double)b/a` 时, 先将 `b` 的值转化为 `double` 型 (注意, 不是将 `b` 转化为 `double` 型, 变量 `b` 的类型是不会改变的!), 然后与 `a` 相除. 由于是一个 `double` 型的数据与一个 `int` 型的数据进行运算, 所以系统会自动将 `int` 型的数据转化为 `double` 型的数据, 然后再做运算. 所以最后的结果是 `2.5`.
- (3) 在计算 `(double)(b/a)` 时, 是先计算 `b/a`, 然后将结果转化为 `double` 型, 所以最后的结果是 `z=double(2)=2.0`
- (4) 需要注意的是, 变量 `a, b` 的类型在整个计算过程中是始终不变的, 即一直是 `int` 型.

 **笔记:** 类型转换是临时性的, 类型转换不会改变变量本身的数据类型!

(5) 类型转换规则:

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

1.2.3 变量与常量

- 变量: 存储数据, 值可以改变

- (1) 变量名: 命名规则与标识符相同
- (2) 变量必须先声明, 后使用; 先赋值, 后使用.
- (3) 变量声明:

类型标识符 变量名列表;



- 可以同时声明多个变量, 用逗号隔开
- 变量声明时可以初始化

```
1  int i, j, k=0; // 同时声明 3 个整型变量, 但只对变量 k 进行初始化
2  double pi=3.14159;
3  char c;
```

- 常量 (常数、字符串): 在程序运行中值不能改变的量
 - (1) 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;
 - (2) 实型常量: 缺省为双精度实数, 后面加 `f` 或 `F` 表示单精度, 加 `l` 或 `L` 表示 `long double`
 - (3) 字符型常量: 用单引号括起来的单个字符和转义字符;
 - (4) 字符串常量: 用双引号括起来的字符序列;
 - (5) 布尔常量: `true` 和 `false` (需加头文件: `#include <stdbool.h>`)
- 符号常量: 用标识符表示常量
 - (1) 声明方式:

```
const 类型标识符 变量名 = 值;
```

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

- (2) 符号常量在声明时必须初始化;
 - (3) 符号常量的值在程序中不能被修改 (即不能被重新赋值)
- 枚举: 定义新的数据类型

```
enum 枚举类型名 {变量可取值列表, 即枚举元素};
```

1.2.4 运算与表达式

- 运算符
 - (1) 算术运算符: `+`、`-`、`*`、`/`、`%`、`++` (自增)、`--` (自减)
 - (2) 赋值运算符: `=`、`+=`、`-=`、`*=`、`/=`、`%=`、`&=`、`|=`、`^=`、`>>=`、`<<=`
 - (3) 逗号运算符: `,` (把若干表达式组合成一个表达式)
 - (4) 关系运算符: 用于比较运算, `>`、`<`、`==`、`>=`、`<=`、`!=`
 - (5) 逻辑运算符: 用于逻辑运算, `&&`、`||`、`!`
 - (6) 条件运算符: `? :`, 是一个三目运算符, 用于条件求值
 - (7) 求字节数运算符: `sizeof` (变量/数据/类型标识符)
 - (8) 位操作运算符: 按二进制位进行运算, `&`、`|`、`~`、`^` (异或)、`<<` (左移)、`>>` (右移)
 - (9) 指针运算符: `*` (取内容)、`&` (取地址)
- 运算符优先级: (https://en.cppreference.com/w/c/language/operator_precedence)
- 语句
 - (1) 空语句 (只有分号)
 - (2) 声明语句;
 - (3) 表达式语句;
 - (4) 复合语句 (将多个语句用 `{ }` 括起来组成的一个语句);
 - (5) 选择语句, 循环语句, 跳转语句;
 - (6)



- 表达式: 由运算符连接常量、变量、函数等所组成的式子;
- 赋值语句

(1) 标准赋值语句

变量=表达式;

```
1 x=3;
2 x=y=3; // 等价于 y=3; x=y; 这种赋值方式不能用于初始化
```

(2) 自增自减: ++, --

- 前置: 先自增或自减, 然后参与表达式运算;
- 后置: 先参与表达式运算, 然后自增或自减;
- 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. 另外, 也不要出现 $y=x++*x$; 以及类似的语句.

```
1 x++; // 等价于 x=x+1;
2 ++x; // 等价于 x=x+1;
3 y=x++*3; // 等价于 y=x*3; x=x+1; 如果是 y=x++*x, 则结果怎样?
4 y=++x*3; // 等价于 x=x+1; y=x*3;
```

(3) 复合赋值运算符: +=、-=、*=、/=、%=


```
1 int x
2 x+=3; // 等价于 x=x+3;
3 x/=3; // 等价于 x=x/3;
```

- 逗号运算符:

表达式1, 表达式2

(1) 先计算 表达式 1, 再计算 表达式 2, 并将 表达式 2 的值作为整个表达式的结果.

```
1 int a=2, b;
2 a = 3*5, a+10; // a=12 or 15?
3 b = (3*5, a+10); // b=?
```

 **注记:** 为了避免由运算优先级所导致的低级错误, 建议多使用小括号.

- 位运算符: 按二进制位进行运算

&、|、^ (异或)、~ (取反)、<< (左移)、>> (右移)

- 求字节数运算符:

sizeof(变量) // 返回指定变量所占的字节数
 sizeof(数据类型) // 返回存储单个指定数据类型的数据所需的字节数
 sizeof(表达式) // 返回存储表达式结果所需的字节数



```

1  int a, b, c, d;
2  a = sizeof(b);    // 变量 b 所占的字节数
3  b = sizeof(int);   // 存储单个 int 型数据所需的字节数
4  c = sizeof(3 + 5); // 存储表达式结果所需的字节数
5  d = sizeof(3.0L + 5);

```

1.2.5 常用数学函数

需加入 `math.h` 头文件 (<https://zh.cppreference.com/w/c/numeric/math>)

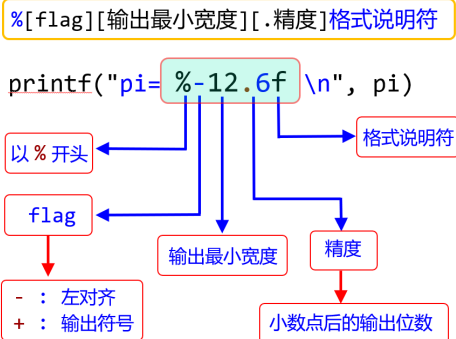
绝对值	<code>abs(x)</code>
平方根	<code>sqrt(x)</code>
指数函数	<code>exp(x)</code> , <code>exp2(x)</code>
x^y	<code>pow(x,y)</code>
对数函数	<code>log(x)</code> , <code>log10(x)</code> , <code>log2(x)</code>
取整函数	<code>ceil(x)</code> , <code>floor(x)</code> , <code>round(x)</code> , <code>trunc(x)</code>
三角函数	<code>sin</code> , <code>cos</code> , <code>tan</code> , <code>asin</code> , <code>acos</code> , <code>atan</code>
双曲三角函数	<code>sinh</code> , <code>cosh</code> , <code>tanh</code> , <code>asinh</code> , <code>acosh</code> , <code>atanh</code>

1.3 C 语言格式化输入输出

1.3.1 C 语言格式化输出

```
printf("格式控制字符串", 输出变量列表); // 建议加头文件 stdio.h
```

- (1) **格式控制字符串**: 包括“普通字符串”、“格式字符串”、“转义字符”
- (2) 普通字符串: 原样输出
- (3) 格式字符串: 以 `%` 开头, 后面跟**格式说明符**和其它选项



```

1  int k=5;
2  double a=3.14;
3  printf("k=%d, a=%f\n", k, a); // 普通字符串按原样输出; 一个格式字符串对应一个变量.

```



- 常见的格式说明符

c	字符型	g	浮点数（系统自动选择输出格式）
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

- 常见的转义字符（输出特殊符号）

\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

1.3.2 C 语言格式化输入

```
scanf("格式控制字符串", 输入变量地址列表); // 建议加头文件 stdio.h
```

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



第二讲 选择与循环

2.1 关系运算与逻辑运算

- 关系运算, 即比较大小: `>` `<` `==` `>=` `<=` `!=`
 - (1) 结论是真则返回 `1`, 否则返回 `0`
 - (2) C 语言中用 `1` 表示 `true`, `0` 表示 `false`
 - (3) `bool` 型变量的值为 `0` 时表示 `false`, 其他它值都表示 `true`
 - (4) 注意 `==` 与 `=` 的区别
 - (5) 对浮点数进行比较运算时尽量不要使用 `==`
- 逻辑运算: `&&` (逻辑与), `||` (逻辑或), `!` (逻辑非)
 - (1) `表达式 1 && 表达式 2`
 - 先计算 `表达式 1` 的值, 若是 `true`, 再计算 `表达式 2` 的值;
 - 若 `表达式 1` 的值是 `false`, 则不再计算 `表达式 2`.
 - (2) `表达式 1 || 表达式 2`
 - 先计算 `表达式 1` 的值, 若是 `false`, 再计算 `表达式 2` 的值;
 - 若 `表达式 1` 的值是 `true`, 则不再计算 `表达式 2`.

 注记: 注意 `&&` 和 `||` 的运算方式.

- (3) 优先级: `!` 优于 `&&` 优于 `||`.
- 条件运算符: `?:`

条件表达式 `?:` 表达式1 `:` 表达式2


- (1) C 语言中唯一的 **三目运算符**;
- (2) **条件表达式** 为真时返回 **表达式 1** 的值, 否则返回 **表达式 2** 的值;
- (3) **表达式 { }1** 的值和 **表达式 2** 的值的数据类型要一致.

2.2 选择结构

2.2.1 IF 语句

- (1) 单分支:

`if` (条件表达式) 语句 *// 如果“条件表达式”的值非零, 则执行后面的语句*

 注记: 这里的语句可以是复合语句 (如果是复合语句的话, 别忘了大括号!)

- (2) 双分支:

```
if (条件表达式)
    语句1
else
    语句2
```

```
// 如果“条件表达式”的值为 true, 则执行“语句1”, 否则执行“语句2”
```

(3) 多分支:

```
if (条件表达式)
    语句1
else if (条件表达式)
    语句2
else if (条件表达式)
    语句3
:
else
    语句n
```

 注记: 条件表达式两边的小括号不能省略!

(4) `if` 语句可以嵌套;

(5) 嵌套时每一层 `if` 都要和 `else` 配套, 若没有 `else`, 则需将该层 `if` 语句用 `{ }` 括起来.

2.2.2 SWITCH 结构

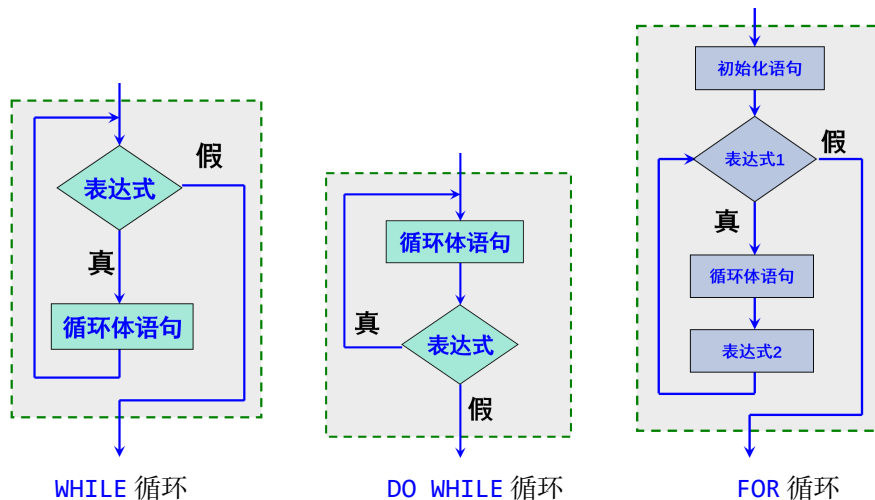
```
switch(表达式) // 这里的“表达式”的值可以是整型、字符型或枚举型
{
    case 常量表达式1:
        语句
    case 常量表达式2:
        语句
    :
    case 常量表达式n:
        语句
    default:
        语句
}
```

- (1) 先计算 `switch` 后面的“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, 则开始执行其后面的语句, 包括其后面所有 `case` 以及 `default` 的语句 (除非遇到 `break`);
- (2) 如果没有匹配的, 则执行 `default` 后面的语句;
- (3) `default` 不是必需的, 即可以没有;
- (4) 每个 `case` 分支最后一般都会加上 `break` 语句;
- (5) 每个 `case` 后面的常量表达式的值不能相同;
- (6) 每个 `case` 后面可以有多个语句 (复合语句), 但可以不用 `{ }`.



2.3 循环结构


- 重复执行: 代码不变, 但数据在变;
- 循环结构的三种实现方式: `while` 循环, `do while` 循环和 `for` 循环.
- 循环可以嵌套.



2.3.1 WHILE 循环

```
while(条件表达式)
{
    循环体语句
}
```

- 执行过程
 - (1) 计算条件表达式的值;
 - (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.

 **注记:** 如果循环体语句是复合语句, 别忘了大括号!

2.3.2 DO WHILE 循环

```
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, 如果是“真”, 则返回第 (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.



2.3.3 FOR 循环

```
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, 返回第二步.

- † 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;
- † 初始化语句 与 表达式 2 可以是逗号语句;
- † 若省略初始化语句 和 表达式 2, 只有表达式 1, 则等同于 while 循环.

- for 循环有时也可以描述为

```
for(循环变量赋初值; 循环条件; 循环变量增量)
{
    循环体语句
}
```


```
1  int i, s=0;
2  for (i=1; i<=10; i++)
3      s = s + i;
```

- 循环变量可以在初始化语句中声明, 这样, 循环变量只在该循环内有效, 循环结束后, 循环变量即被释放.

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

2.3.4 循环的非正常终止

```
break    // 跳出循环体, 但只能跳出一层循环, 一般用在循环语句和 switch 语句中.
continue // 结束本轮循环, 执行下一轮循环, 一般用在循环语句中.
goto     // 跳转语句, 不建议使用.
```

 注记: break 和 continue 通常与 if 语句配合使用.



第三讲 数组

3.1 一维数组

数组 (array) : 具有一定顺序关系的若干相同类型数据的集合, 是基本数据类型的推广.

- 一维数组的声明:

类型标识符 数组名[n] // 声明一个长度为 n 的一维数组 (向量)

- (1) 类型标识符: 数组元素的数据类型;
- (2) 数组名代表的是数组在内存中的首地址;
- (3) n 为数组的长度, 可以是一个表达式, 但值必须是一个确定的正整数.

- 一维数组的引用:

数组名[k] // 注: 下标 k 的取值为 0 到 n-1

- (1) 只能通过循环逐个引用数组元素;
- (2) 数组元素在内存中是按顺序连续存放的, 它们在内存中的地址是连续的.

 注记: 数组的下标不能越界, 否则可能会引起严重的后果!

- 一维数组的初始化: 在声明时可以同时赋初值.

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

- (1) 全部元素都初始化时可以不指定数组长度, 系统会根据所给的数据自动确定数组的长度, 如 `int x[]={0,2,4,6,8};`
- (2) 可以部分初始化, 即只给部分元素赋初值, 如 `int x[5]={0,2,4};`
- (3) 若数组声明时进行了部分初始化, 则没有初始化的元素自动赋值为 0;
- (4) 声明数组时, 若长度中含有变量, 则不能初始化!

```
1 const int n=3;
2 int x[n]; // OK
3 int y[n]={1,2,3}; // ERROR
```

- (5) 只能对数组元素赋值, 不能对数组名赋值!

```
1 const int n=5;
2 int x[n];
3
4 x={1,2,3,4,5}; // ERROR
5 for(int i=0; i<n; i++)
6     x[i]=i;
```

3.2 二维数组

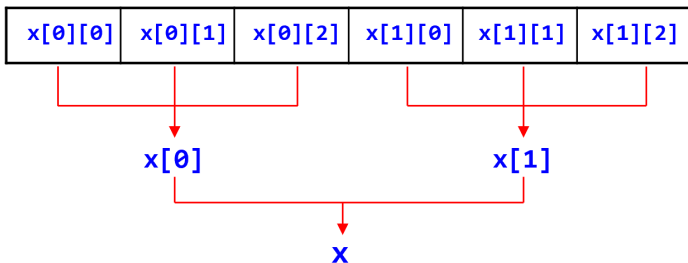
- 二维数组的声明: m 行 n 列

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: i 的取值是 0 到 $m-1$, j 的取值是 0 到 $n-1$

变量名[i][j] // 注意下标不要越界!

- 二维数组的存储: 按行存储.



注记: 在 C 语言中, 二维数组可以看作是由一维数组所组成的数组.

- 二维数组的初始化

- (1) 全部初始化, 此时可以省略第一维的大小 (但不能省略其他维数)

```
1 int x[2][3]={1,3,5,2,6,10};
2 int x[][3]={1,3,5,2,6,10}; // 省略第一维的大小
```

- (2) 可以分组初始化, 也可以部分初始化

```
1 int x[2][3]={{1,3,5}, {2,6,10}};
2 int x[2][3]={{1}, {2,6}}; // 部分初始化
```

- 多维数组: 多维数组的赋值、引用、初始化与二维数组类似

类型标识符 变量名[n1][n2][n3]...


3.3 字符串

- 字符串: 在 C 语言中, 字符串是通过字符数组来实现的.

- (1) 字符串以 “\0” 为结束标志 (称为**字符串结束标志符**);
- (2) 使用双引号表示的字符串常量进行初始化时, 会自动添加结束标志符.

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



 **注记：**字符串可以看作是由字符组成的数组, 但与普通数组 (数值型数值) 不同, 在运用上更加灵活方便.

- 字符串的输出: `printf` 和 `puts`

```
1 char str[20]="C and Matlab"; // 字符数组的长度为 20, 但仅包含 12 个字符
2 printf("str=%s\n", str); // 整体输出, 系统自动逐个输出字符, 直到遇见字符串结束标志符
```


 **注记：**输出字符串中不含 **字符串结束标志符** “\0”

- 字符串的输入: `scanf` 或 `gets`

```
1 char str[20];
2 scanf("%s", str); // 字符串数组名代表首地址, 因此无需加 &
```

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

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

 **注记：**这些函数只能作用在字符串上, 不能作用在字符上.


```
1 int x;
2 double y;
3 x=atoi("66"); // x=66, 字符串中只能包括数字
4 y=atof("14.5"); // y=14.5, 字符串中只能包括数字和小数点
```

- C 语言字符检测函数 (头文件 `cctype`)

函数	含义	示例
<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>isprint</code>	是否为可打印字符, 包含空格	<code>isprint('A')</code>
<code>isgraph</code>	是否为可打印字符, 不含空格	<code>isgraph('a')</code>



<code>ispunct</code>	除字母数字空格外的可打印字符	<code>ispunct('*')</code>
<code>iscntrl</code>	是否为控制符	<code>iscntrl('\n')</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>

 **注记：** 以上检测和转换函数只针对单个字符, 而不是字符串.

- 字符与整数的运算: 字符参加算术运算时, 自动转换为整数 (按 ASCII 码转换)

```
1 char x='8';
2 int y=x-3;
3 int z=x-'3';
4 printf("y=%d, z=%d\n", y, z); // 输出是什么?
```



第四讲 函数

4.1 函数的声明、定义与调用

C 语言程序是由一个或多个函数构成的,且必须有一个 `main` 函数,通常称为 **主函数**.

- 函数的定义: 函数由两部分组成,分别是**函数头**和**函数体**.

```
类型标识符 函数名(形式参数列表) // 函数头
{
    函数体
}
```

- (1) “**类型标识符**”指明了函数的类型,即函数返回值的类型,若没有返回值,则使用 `void`
- (2) “**形式参数列表**”:可以有多个形式参数,也可以没有,格式如下:

```
类型标识符 变量1, 类型标识符 变量2, ... ..
```

- **形式参数** (通常简称**形参**) 需要指定数据类型
(形参在函数定义时不会分配任何存储空间,也没有具体的值,因此称为形式参数)
- 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,但括号不能省.
- 形参只在函数内部有效/可见.

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

- 函数的返回值
 - 函数返回值通过 `return` 语句给出.
 - 若没有返回值,可以不写 `return`,也可以写不带任何表达式的 `return`.

```
1 int my_max(int x, int y)
2 {
3     if (x > y) return x;
4     else return y;
5 }
```

- 函数的调用

```
函数名(实际参数列表)
```

- (1) **实际参数** (通常简称**实参**) 必须是实际存在的变量或表达式,即有具体的值.
- 主调函数**与**被调函数**: 为了描述方便,如果在函数 A 中调用函数 B,我们称 A 是主调函数, B 是被调函数.
 - (1) 如果被调函数在主调函数之前已经定义,则可直接调用.
 - (2) 如果被调函数是在主调函数后面才定义,则需要在调用前事先声明.
函数声明: 函数头加分号,比如 `int my_max(int x, int y);`

- (3) 被调函数的声明: 可以在主调函数中声明, 也可以在所有函数之前声明.
- (4) 被调函数可以出现在表达式中, 此时必须要有返回值.
- (5) 被调函数执行过程中遇到 `return` 语句时, 就返回主调函数, 如果 `return` 后面带有表达式, 则将该表达式的值带回到主调函数, 如果 `return` 后面没有表达式, 则表示直接返回到主调函数.

例 4.1 随机数的生成. (需包含头文件 `stdlib.h`)

- `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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int x=rand();
7     printf("x=%d\n\n", x);
8
9     int seed=2023;
10    x=rand();
11    printf("seed=%d, x=%d\n\n", seed, x);
12
13    int a=5, b=10;
14    x=rand()%(b-a+1) + a;
15    printf("x=%d in [%d,%d]\n\n", x,a,b);
16
17    double y;
18    y=rand()/((double)RAND_MAX);
19    printf("y=%f in [0,1], RAND_MAX=%d\n\n", y, RAND_MAX);
20
21    return 0;
22 }
```

例 4.2 (显示系统当前的时间)

头文件 `time.h` 中函数 `time(0)` 或 `time(NULL)` 返回当前时间与 1970 年 1 月 1 日零时零分零秒的时间差 (格林威治时间, 以秒为单位), 北京时间: 格林威治时间 + 8 小时.

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main()
5 {
6     long Second, Minute, Hour;
7
8     Second = time(0);
9     Minute = Second / 60;
10    Hour = Minute / 60;
11    printf("当前北京时间是 %02d:%02d:%02d\n", (Hour+8)%24, Minute%60, Second%60);
12
13    return 0;
14 }
```



14 }

例 4.3 计时函数`clock` (需包含头文件 `time.h`)

- `clock()` : 返回进程启动后所使用的 cpu 总毫秒数.

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main()
5 {
6     const int N=300;
7     float A[N][N], B[N][N], C[N][N];
8     time_t t0, t1;
9     double tt, tc;
10
11     for(int i=0; i<N; i++)
12     {
13         for(int j=0; j<N; j++)
14         {
15             A[i][j] = 1.0*(i+1);
16             B[i][j] = 1.0*(j+1);
17             C[i][j] = 0.0;
18         }
19     }
20
21     t0 = time(0);
22     tc = clock();
23
24     for(int i=0; i<N; i++)
25     {
26         for(int j=0; j<N; j++)
27         {
28             for(int k=0; k<N; k++)
29                 C[i][j] += A[i][k]*B[k][j];
30         }
31     }
32
33     tc = (double)(clock()-tc)/CLOCKS_PER_SEC;
34     t1 = time(0);
35     tt = t1-t0;
36
37     printf("Elapsed time is %.6e or %.6e \n", tt, tc);
38
39     return 0;
40 }
41 // CLOCKS_PER_SEC: 每秒的滴答数, 通常为 1000.
```

☕ Tips: 如何使得代码在每次执行时产生不同的随机整数?

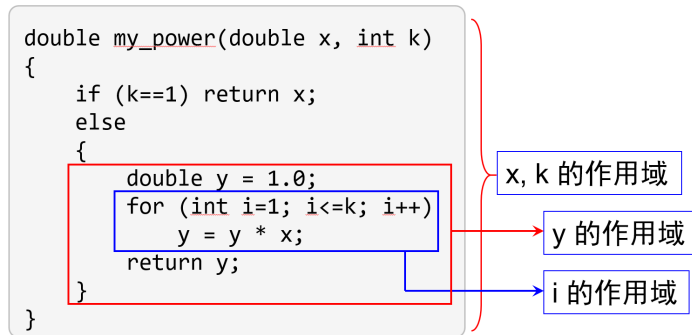
```
1 srand(time(0));
2 x = rand();
```



4.2 作用域、局部变量与全局变量

每个变量都有作用域,即在程序中哪些地方可以使用该变量.

- 数据（通常指变量）的作用域：数据在程序中的有效区域
- C 语言中常见的作用域
 - (1) 函数原型作用域：函数原型声明时形参的作用域, 仅限形参列表的左右括号之间, 因此在函数声明时, 形参的变量名可省略（但类型不能省）
 - (2) 函数作用域
 - (3) 语句块作用域（循环体，语句组 ... 等）
- **局部变量**：数据只在某个局部区域内有效（可见），如：
 - (1) 函数的形参和函数中定义的变量，只在该函数内有效
 - (2) 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
 - (3) 语句块中定义的变量只在该语句块中有效



- **全局变量**：数据在整个程序中都有效（可见）
 - (1) 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
 - (2) 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`
- 若局部变量与全局变量同名，则优先使用局部变量


```
1 #include <stdio.h>
2 int k = 2; // 全局变量
3 int main()
4 {
5     int i=5, x; // 局部变量
6     x=i+k;
7     printf("x=%d\n", x);
8
9     {
10         int k=16; // 局部变量
11         x=i+k;
12         printf("x=%d\n", x);
13     }
14
15     x=i+k;
16     printf("x=%d\n", x);
```



17 }

4.3 函数间的参数传递

- 传递方式一: **值传递**
 - (1) 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
 - (2) 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;
 - (3) 将实参的值传递给对应的形参, 即**单向传递**;
 - (4) 实参和形参在数量、类型、顺序上应严格一致;
 - (5) 形参获得实参传递过来的值后, 便与实参脱离关系, 即对形参的任何改变都不会对实参产生任何影响。
- 将数组中的某个元素传递给被调函数: 值传递
- 将整个数组传递给被调函数, 传递方式二: **地址传递**
 - (1) 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)
 - (2) 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

 **注记:** 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!

例 4.4 交换两个数组.

```

1 #include <stdio.h>
2
3 void my_swap(int a[], int b[], int n)
4 {
5     int t, i;
6
7     for (i=0; i<n; i++)
8     {
9         t=a[i]; a[i]=b[i]; b[i]=t;
10    }
11 }
12
13 int main()
14 {
15     int x[3]={1,2,3}, y[3]={2,4,6};
16     int n = sizeof(x)/sizeof(x[0]);
17
18     printf("\nBefore swapping:\n");
19     for(int i=0; i<n; i++)
20         printf("x[%d]=%d, y[%d]=%d\n", i, x[i], i, y[i]);
21
22     my_swap(x,y,n);
23
24     printf("\nAfter swapping:\n");
25     for(int i=0; i<n; i++)
26         printf("x[%d]=%d, y[%d]=%d\n", i, x[i], i, y[i]);

```



```

27
28     return 0;
29 }

```

- 几点说明:

- 若形参是数组, 为增加灵活性, 通常省略长度, 如果是二维数组, 则只能省略行数;
- 如果数组形参没有指定长度, 则需另加一个形参, 用来传递实参数组的大小, 或者通过全局变量实现;

```

1 void my_swap(int a[], int b[], int n); // 可以省略长度, 但中括号不能省
2 void sum_col(double A[][n], double s[]); // 这里的 n 不能省略, 只能省略行数

```

- 函数调用时, 只需用数组名.

例 4.5 计算矩阵各列的和 (函数形式).

```

1 #include <stdio.h>
2
3 const int m=3, n=4;
4
5 void sum_col(double A[][n], double s[])
6 {
7     int i, j;
8     for(j=0; j<n; j++) s[j]=0.0;
9     for(j=0; j<n; j++)
10         for(i=0; i<m; i++)
11             s[j] = s[j] + A[i][j];
12 }
13
14 int main()
15 {
16     double H[m][n], s[n];
17
18     for(int i=0; i<m; i++)
19         for(int j=0; j<n; j++)
20             H[i][j]=1.0/(i+j+1);
21     sum_col(H, s);
22     printf("s[0]=%f, s[%d]=%f\n", s[0], n-1, s[n-1]);
23
24     return 0;
25 }

```

4.4 函数嵌套与递归

- 函数的嵌套调用

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

例 4.6 利用下边的公式计算阶乘:


$$n! = \begin{cases} 1, & n = 0, \\ n * (n - 1)!, & n > 0. \end{cases}$$



```

1 #include <stdio.h>
2
3 int factorial_loop(int n); // 普通方式 (循环)
4 int factorial_recursion(int n); // 递归方式
5
6 int main()
7 {
8     int n, y;
9     printf("Please input n: ");
10    scanf("%d", &n);
11
12    y = factorial_loop(n);
13    printf("普通方式: %d!=%d\n", n, y);
14
15    y = factorial_recursion(n);
16    printf("递归方式: %d!=%d\n", n, y);
17
18    return 0;
19 }
20
21 int factorial_loop(int n) // 普通方式
22 {
23     int y = 1;
24     for (int i=1; i<=n; i++)
25         y = y * i;
26     return y;
27 }
28
29 int factorial_recursion(int n) // 递归方式
30 {
31     if (n==0) return 1;
32     else return n*factorial_recursion(n-1);
33 }

```

 **注记:** 对同一个函数的多次不同调用中, 编译器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。

4.5 内联函数

- 内联函数: **inline**

- (1) 使用内联函数能节省参数传递、控制转移等开销, 提高代码的执行效率;
- (2) 内联函数通常应该是功能简单、规模小、使用频繁的函数;
- (3) 内联函数体内不建议使用循环语句和 **switch** 语句;
- (4) 有些函数无法定义成内联函数, 如递归调用函数等。

```

1 inline double f(double x) // 内联函数
2 {
3     return 2*x*x - 1; // f(x) = 2x^2 - 1
4 }

```



第五讲 指针

指针变量, 简称**指针**, 用来存放其它变量的**内存地址**. 通过指针, 可以直接访问系统内存, 从而提高程序执行效率.

 **笔记:** 程序中的变量、函数等, 在内存中都有相应的地址.

5.1 指针的定义与运算

- 指针的定义

类型标识符 * 指针变量名


- (1) **类型标识符**: 表示该指针可指向的对象的数据类型, 即该指针能存放哪类数据的地址.
- (2) 星号和指针变量名之间可以没有空格

- 指针的两个基本运算

- (1) 提取变量的内存地址: **&变量名**
- (2) 提取指针所指向的变量 (即目标对象) 的值: ***指针**


```
1 int x;  
2 int * px; // 声明指针 px  
3 px = &x; // 将 x 的地址赋给指针 px  
4 *px = 3; // 等价于 x = 3, 注意星号与 px 之间不能有空格!
```

 **笔记:** 此时, 我们通常称 **px** 是指向 **x** 的指针, 即目标对象.

 **Tips:** 内存空间的访问方式: 1) 变量名; 2) 内存地址. 指针提供了一种访问变量的高效方法.


- 指针的初始化: 声明指针变量时, 可以赋初值. 注意指针的类型必须与其指向的对象的类型一致.

```
1 int x;  
2 int * px = &x; // 初始化
```

 **笔记:** 在使用指针时, 我们通常关心的是指针的目标对象!

- 指针赋值: 给指针赋值时, 只能使用以下的值

- 空指针: **0**, **NULL** 或值为 **0/NULL** 的常量;
- 类型匹配的目标对象的地址;
- 同类型的另一个有效指针;
- 类型匹配的对象的前后地址 (相对位置).

 **笔记:** 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后果.

- void** 类型的指针

void * 指针名

- (1) **void** 类型的指针可以存储任何类型的对象的地址;
- (2) 不允许直接使用 **void** 指针访问其目标对象;
- (3) 必须通过**显式类型转换**, 才可以访问 **void** 类型指针的目标对象.

```

1  int x;
2  int * px;
3  void * pv;
4  pv = &x;      // OK, void 型指针指向整型变量
5  px = (int *)pv; // OK, 使用 void 型指针时需要强制类型转换

```

• 指向常量的指针

const 类型标识符 * 指针名

- (1) 指向常量的指针必须用 **const** 声明;
- (2) 这里的 **const** 限定了指针所指对象的属性, 即目标对象是常量, 而不是指针是常量;
- (3) 允许把非 **const** 对象的地址赋给指向常量的指针, 即指向常量的指针也可以指向普通变量;
- (4) 不允许使用指向常量的指针来修改其目标对象的值, 即使它所指向的对象不是常量.

```

1  const int a = 3;
2  int b = 5;
3  const int * cpa = &a; // OK
4  *cpa = 5; // ERROR
5  cpa = &b; // OK
6  *cpa = 9; // ERROR
7  b = 9; // OK

```

• 常量指针, 简称常指针: 指针本身的值不能修改.

类型标识符 * const 指针名

```

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

```

• 指向常量的常指针

const 类型标识符 * const 指针名

- (1) 指针本身的值不能修改, 也不能通过该指针修改其目标对象的值.
- 指针算术运算: 指针可以和整数或整型变量进行加减运算, 运算规则与指针的类型有关.
 - (1) 指针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
 - (2) 指针的算术运算通常是与数组配合使用, 非常方便
 - (3) 指针加上或减去一个整型数值 k , 实际移动距离 (字节数) 与其数据类型有关: 假定 px 是 **int** 型 (占 4 个字节), 则 $px+3$ 与 px 相差 12 个字节.



```

1 int x[5] = {0,1,2,3,4};
2 int * px = &x[0]; // px 指向 x[0]
3 cout << *px << endl; // 输出 x[0] 的值
4 cout << *(px+1) << endl; // 输出的 x[1] 值
5 px = px + 2; // px 改为指向 x[2]

```

- 指针数组: 由指针变量组成的数组

(1) 指针数组的声明:

类型标识符 * 指针数组名[n]

5.2 指针与一维数组

在 C 语言中, 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.

- 引用数组元素的四种方式
 - (1) 数组名 + 下标;
 - (2) 数组名 + 指针运算;
 - (3) 指针 + 指针运算;
 - (4) 指针 + 下标 (数组运算) .

例 5.1 指针与一维数组: 引用数组元素的四种方式.


```

1 #include <stdio.h>
2
3 int main()
4 {
5     int a[5]={1,2,3,4,5};
6
7     printf("数组名+下标:\n");
8     for(int i=0; i<5; i++)
9         printf("%2d", a[i]);
10    printf("\n\n");
11
12    printf("数组名+指针运算:\n");
13    for(int i=0; i<5; i++)
14        printf("%2d", *(a+i));
15    printf("\n\n");
16
17    printf("指针+指针运算:\n");
18    for(int *pa=a; pa<a+5; pa++)
19        printf("%2d", *pa);
20    printf("\n\n");
21
22    printf("指针+下标:\n");
23    int *pa=a;
24    for(int i=0; i<5; i++)
25        printf("%2d", pa[i]);
26    printf("\n\n");
27
28    return 0;

```



29 }

 **注记：** 数组名代表数组的首地址，当数组名出现在表达式中时，等效于一个常指针。

```
1 int a[]={0,2,4,8};
2 int * pa = a; // OK, 数组名代表数组的首地址, 即 pa=&a[0]
3 *pa = 3; // OK, 等价于 a[0]=3
4 *(pa+2) = 5; // OK, 等价于 a[2]=5
5 *(a+2) = 5; // OK, 等价于 a[2]=5
6 *(pa++) = 3; // OK, 等价于 a[0]=3; pa = pa+1;
7 *(a++) = 3; // ERROR! a代表数组首地址, 等效于常指针!
```

- 小结：一维数组 $a[n]$ 与指针 $pa=a$
 - (1) 一维数组名 a 是地址常量，数组名 a 与 $\&a[0]$ 等价；
 - (2) $a+i$ 是 $a[i]$ 的地址， $a[i]$ 与 $*(a+i)$ 等价；
 - (3) 若指针 pa 存储的是数组 a 的首地址，则 $*(pa+i)$ 与 $pa[i]$ 等价；
 - (4) 数组元素的下标访问方式也是按地址进行的；
 - (5) 指针的值可以随时改变，即可以指向不同的元素，通过指针访问数组的元素更加灵活；
 - (6) 数组名等效于常量指针，值不能改变；
 - (7) $pa++$ 或 $++pa$ 合法，但 $a++$ 不合法；

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

5.3 指针与二维数组

- 在 C 语言中，二维数组是按行存储的，可以理解为由一维数组组成的数组，其元素也是连续存放的。

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

可以理解为： $A \begin{cases} A[0] \text{ — } A_{00} \ A_{01} \ A_{02} \\ A[1] \text{ — } A_{10} \ A_{11} \ A_{12} \end{cases}$

$A[0][0]$	$A[0][1]$	$A[0][2]$	$A[1][0]$	$A[1][1]$	$A[1][2]$
-----------	-----------	-----------	-----------	-----------	-----------

$A[0]$
(第一行的首地址)

$A[1]$
(第二行的首地址)

$A[0], A[1]$ 称为行数组

- 访问二维数组的元素可以通过数组名和指针运算进行；
- 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：
 - (1) $A[0]$ 是一维数组名，它指向的是行数组 $A[0]$ 的首元素，即 $*A[0]$ 与 $A[0][0]$ 等价；
 - (2) A 是二维数组名，它指向的是它的首元素，而它的元素都是一维数组（即行数组），因此 $*A$ 与 $A[0]$ 等价。另外，它的指针移动单位是“行”，所以 $A+i$ 对应的是第 i 个行数组，即 $A[i]$ 。

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

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




```

1  int A[2][3]={1,2,3},{7,8,9}};
2  int * p = A[0]; // OK, p 指向 A[0][0]
3  int * p = A; // ERROR! p 只能指向一个具体的整型变量, 不能是向量

```

- 指针与二维数组: 设指针 `pa=&A[0][0]`, 则

`A[i][j] <=> *(pa+n*i+j)` // 这里 `n` 是 `A` 的列数

例 5.2 指针与二维数组举例.


```

1  #include <stdio.h>
2
3  const int N = 5;
4
5  int main()
6  {
7      int A[N][N];
8
9      for(int i=0; i<N; i++)
10         for(int j=0; j<N; j++)
11             A[i][j] = i+j;
12
13     printf("使用数组名: \n");
14     for(int i=0; i<N; i++)
15     {
16         for(int j=0; j<N; j++)
17             printf("%3d", A[i][j]);
18         printf("\n");
19     }
20
21     printf("使用指针: \n");
22     int *pa = &A[0][0];
23     for(int i=0; i<N; i++)
24     {
25         for(int j=0; j<N; j++)
26             printf("%3d", *(pa+N*i+j));
27         printf("\n");
28     }
29
30     return 0;
31 }


```

5.4 指针与函数

- 指针作为函数参数:
 - (1) 指针作为函数参数时, 以地址方式传递数据;
 - (2) 形参是指针时, 实参可以是同类型指针或同类型数据的地址;

 **注记:** 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!



 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.

- 指针型函数: 函数的返回值是地址或指针, 一般形式如下:


```
数据类型 * 函数名 (形参列表)
{
    函数体
}
```

- 指向函数的指针, 即**函数指针**

- (1) 在程序运行过程中, 不仅数据要占用内存空间, 函数也要在内存中占用一定的空间;
- (2) 函数名就代表函数在内存空间中的首地址;
- (3) 用来存放这个地址的指针就是指向该函数的指针;
- (4) 函数指针的定义:

```
数据类型 (* 函数指针名)(形参列表)
```


- (5) 这里的 数据类型 和 形参列表 应与其指向的函数相同;
- (6) 可以象使用函数名一样使用函数指针.

 **注记:** 函数名除了表示函数的首地址外, 还带有函数的形参, 返回值类型等信息.

5.5 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. 此时可以使用 C 语言提供的持久动态内存分配方法.

每个程序在执行时都会占用一块可用的内存空间, 用于存放动态分配的对象, 此内存空间称为自由存储区 (free store) 或堆 (heap).

 **注记:** 由于实现机制和管理方式的不同, 与栈相比, 堆通常比较大, 因此在处理大数组时建议使用这种方法.

- 申请持久动态存储单元: `malloc`

```
px = (数据类型*)malloc(size);
```

- (1) 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, 并赋给指针 `px`;
- (2) 若申请不成功, 则返回 `0` 或 `NULL`;
- (3) 如果是在初始化语句中使用, 可以省略 (数据类型 *)

```
1 int *px;
2 px = (int*)malloc(sizeof(int)*8); // 申请长度为 32 字节的内存空间
3
4 double *py = malloc(sizeof(double)*8); // 申请长度为 64 字节的内存空间
```



- 释放由 `malloc` 申请的存储单元: `free`

```
free px;
```

- (1) `px` 是 `malloc` 操作的返回值.
- (2) 通过 `malloc` 申请的存储空间一定要通过 `free` 手工释放, 否则容易造成内存泄露.

例 5.3 二维动态数组, 通过普通指针实现.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     const int N=500;
7     float *A = malloc(sizeof(float)*N*N);
8     float *B = malloc(sizeof(float)*N*N);
9     float *C = malloc(sizeof(float)*N*N);
10    int i, j, k;
11
12    for(i=0; i<N; i++)
13    {
14        for(j=0; j<N; j++)
15        {
16            *(A+N*i+j) = 1.0*(i+1);
17            *(B+N*i+j) = 1.0*(j+1);
18            *(C+N*i+j) = 0.0;
19        }
20    }
21
22    for(i=0; i<N; i++)
23    {
24        for(j=0; j<N; j++)
25        {
26            for(k=0; k<N; k++)
27                *(C+N*i+j) += *(A+N*i+k) * (*(B+N*k+j));
28        }
29    }
30
31    free(A); free(B); free(C); // 释放动态数组
32
33    return 0;
34 }
```

5.6 应用: 矩阵乘积的快速算法

5.6.1 矩阵乘积的普通方法

设 $A = [a_{ij}], B = [b_{ij}] \in \mathbb{R}^{n \times n}$, 则 $C = [c_{ij}] = AB$, 其中

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}.$$



```

1  for (i=0; i<n; i++)
2      for (j=0; j<n; j++)
3          for (k=0; k<n; k++)
4              C[i,j] = C[i,j] + A[i][k]*B[k][j];

```

易知, 总运算量为: n^3 (乘法) + n^3 (加法) = $2n^3$.

5.6.2 Strassen 方法

德国数学家 Strassen 在 1969 年提出了计算矩阵乘积的快速算法, 将运算量降为约 $O(n^{2.81})$. 下面以二阶矩阵为例, 描述 Strassen 方法的实现过程. 设

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

则 $C = AB$ 的每个分量为

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21}, & c_{12} &= a_{11}b_{12} + a_{12}b_{22}, \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21}, & c_{22} &= a_{21}b_{12} + a_{22}b_{22}. \end{aligned}$$

在 Strassen 算法中, 我们并不直接通过上面的公式来计算 C , 而是先计算下面 7 个量:

$$\begin{aligned} x_1 &= (a_{11} + a_{22})(b_{11} + b_{22}), \\ x_2 &= (a_{21} + a_{22})b_{11}, \\ x_3 &= a_{11}(b_{12} - b_{22}), \\ x_4 &= a_{22}(b_{21} - b_{11}), \\ x_5 &= (a_{11} + a_{12})b_{22}, \\ x_6 &= (a_{21} - a_{11})(b_{11} + b_{12}), \\ x_7 &= (a_{12} - a_{22})(b_{21} + b_{22}). \end{aligned}$$

于是, C 的各元素可以表示为:

$$\begin{aligned} c_{11} &= x_1 + x_4 - x_5 + x_7, \\ c_{12} &= x_3 + x_5, \\ c_{21} &= x_2 + x_4, \\ c_{22} &= x_1 + x_3 - x_2 + x_6. \end{aligned}$$

易知, 总共需要做 7 次乘法和 18 次加法.

下面考虑一般情形. 我们采用分而治之的思想, 先将矩阵 A, B 进行 2×2 分块, 即

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

则 $C = AB$ 也可以写成 2×2 分块形式, 即

$$C_{11} = X_1 + X_4 - X_5 + X_7,$$



$$\begin{aligned}C_{12} &= X_3 + X_5, \\C_{21} &= X_2 + X_4, \\C_{22} &= X_1 + X_3 - X_2 + X_6,\end{aligned}$$

其中

$$\begin{aligned}X_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\X_2 &= (A_{21} + A_{22})B_{11}, \\X_3 &= A_{11}(B_{12} - B_{22}), \\X_4 &= A_{22}(B_{21} - B_{11}), \\X_5 &= (A_{11} + A_{12})B_{22}, \\X_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\X_7 &= (A_{12} - A_{22})(B_{21} + B_{22}).\end{aligned}$$

需要 7 次子矩阵的乘积和 18 次子矩阵加法. 假定采用普通方法计算子矩阵的乘积, 即需要 $(n/2)^3$ 乘法和 $(n/2)^3$ 次加法, 则采用 Strassen 方法计算 A 和 B 乘积的运算量为

$$7 \times ((n/2)^3 + (n/2)^3) + 18 \times (n/2)^2 = \frac{7}{4}n^3 + \frac{9}{2}n^2.$$

大约是普通矩阵乘积运算量的 $\frac{7}{8}$. 在计算子矩阵的乘积时, 我们仍然可以采用 Strassen 算法. 依此类推, 于是, 由递归思想可知, 则总运算量大约为 (只考虑最高次项, 并假定 n 可以不断对分下去)

$$7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807...}$$

记两个 n 阶矩阵乘积的运算量为 $O(n^\alpha)$. 在 Strassen 算法出现之前, 大家一直认为 $\alpha = 3$. 但随着 Strassen 算法的出现, 大家意识到这个量级是可以小于 3 的. 但 α 能不能进一步减少? 如果能的话, 它的极限又是多少? 这些问题引起了众多学者的极大兴趣. 1978 年, Pan 证明了 $\alpha < 2.796$. 一年后, Bini 等又将这个上界改进到 2.78. 在 Pan 和 Bini 等的工作的基础上, Schönhage 于 1981 年证明了 $\alpha < 2.522$. 而上界首次突破 2.5 是由 Coppersmith 和 Winograd 提出来的, 他们证明了 $\alpha < 2.496$. 1990 年, 他们又在 Strassen 算法的基础上提出了一种新的矩阵乘积计算方法, 将运算量级降至著名的 2.376. 这个记录一直持续了近二十年, 直到 2010 年前后, Stothers, Vassilevska-Williams 和 Le Gall 分别将这个上界降到 2.37293, 2.3728642 和 2.3728639. 虽然有许多学者相信, α 的极限应该是 2, 但至今无法证实.

相关参考文献:

- (1) V. Strassen, Gaussian elimination is not optimal, *Numer. Math.*, 13 (1969), 354–356.
- (2) V. Y. Pan, Strassen's algorithm is not optimal, In *Proc. FOCS*, 19 (1978), 166–176.
- (3) D. Bini, M. Capovani, F. Romani and G. Lotti, $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication, *Inf. Process. Lett.*, 8 (1979), 234–235.
- (4) A. Schönhage, Partial and total matrix multiplication, *SIAM J. Comput.*, 10 (1981), 434–455.
- (5) D. Coppersmith and S. Winograd, On the asymptotic complexity of matrix multiplication, In *Proc. SFCS*, 82–90, 1981.
- (6) D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, *J. Symbolic Computation*, 9 (1990), 251–280.



5.7.1 Gauss 消去过程

本节给出 Gauss 消去过程的详细执行过程, 写出相应算法, 并编程实现.

记 $A^{(1)} = [a_{ij}^{(1)}]_{n \times n} = A$, $b^{(1)} = [b_1^{(1)}, b_2^{(1)}, \dots, b_n^{(1)}]^\top = b$, 即

$$a_{ij}^{(1)} = a_{ij}, \quad b_i^{(1)} = b_i, \quad i, j = 1, 2, \dots, n.$$

第 1 步: 消第 1 列.

设 $a_{11}^{(1)} \neq 0$, 计算 $m_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}, i = 2, 3, \dots, n$. 对增广矩阵进行 $n - 1$ 次初等变换, 即依次将增广矩阵的第 i 行减去第 1 行的 m_{i1} 倍, 将新得到的矩阵记为 $A^{(2)}$, 即

$$A^{(2)} = \left[\begin{array}{cccc|c} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ & \vdots & \ddots & \vdots & \vdots \\ & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} & b_n^{(2)} \end{array} \right],$$

其中

$$a_{ij}^{(2)} = a_{ij}^{(1)} - m_{i1}a_{1j}^{(1)}, \quad b_i^{(2)} = b_i^{(1)} - m_{i1}b_1^{(1)}, \quad i, j = 2, 3, \dots, n.$$

第 2 步: 消第 2 列.

设 $a_{22}^{(2)} \neq 0$, 计算 $m_{i2} = \frac{a_{i2}^{(2)}}{a_{22}^{(2)}}, i = 3, 4, \dots, n$. 依次将 $A^{(2)}$ 的第 i 行减去第 2 行的 m_{i2} 倍, 将新得到的矩阵记为 $A^{(3)}$, 即

$$A^{(3)} = \left[\begin{array}{ccccc|c} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ & & a_{33}^{(2)} & \cdots & a_{3n}^{(2)} & b_3^{(3)} \\ & & \vdots & \ddots & \vdots & \vdots \\ & & a_{n3}^{(3)} & \cdots & a_{nn}^{(3)} & b_n^{(3)} \end{array} \right],$$

其中

$$a_{ij}^{(3)} = a_{ij}^{(2)} - m_{i2}a_{2j}^{(2)}, \quad b_i^{(3)} = b_i^{(2)} - m_{i2}b_2^{(2)}, \quad i, j = 3, 4, \dots, n.$$

依此类推, 经过 $k - 1$ 步后, 可得新矩阵 $A^{(k)}$:

$$A^{(k)} = \left[\begin{array}{ccccc|c} a_{11}^{(1)} & \cdots & a_{1k}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & \ddots & \vdots & & \vdots & \vdots \\ & & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} & b_k^{(k)} \\ & & \vdots & \ddots & \vdots & \vdots \\ & & a_{nk}^{(k)} & \cdots & a_{nn}^{(3k)} & b_n^{(k)} \end{array} \right],$$

第 k 步: 消第 k 列.

设 $a_{kk}^{(k)} \neq 0$, 计算 $m_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, i = k + 1, k + 2, \dots, n$. 依次将 $A^{(k)}$ 的第 i 行减去第 k 行的 m_{ik} 倍, 将新得到的矩阵记为 $A^{(k+1)}$, 矩阵元素的更新公式为

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - m_{ik}a_{kj}^{(k)}, \quad b_i^{(k+1)} = b_i^{(k)} - m_{ik}b_k^{(k)}, \quad i, j = k + 1, k + 2, \dots, n. \quad (5.1)$$



这样, 经过 $n-1$ 步后, 即可得到一个上三角矩阵 $A^{(n)}$:

$$A^{(n)} = \left[\begin{array}{cccc|c} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ & & \ddots & \vdots & \vdots \\ & & & a_{nn}^{(n)} & b_n^{(n)} \end{array} \right].$$

最后, 回代求解

$$x_n = \frac{b_n^{(n)}}{a_{nn}^{(n)}}, \quad x_i = \frac{1}{a_{ii}^{(i)}} \left(b_i^{(i)} - \sum_{j=i+1}^n a_{ij}^{(i)} x_j \right), \quad i = n-1, n-2, \dots, 1.$$

Gauss 消去法的运算量

整个 Gauss 消去法的乘除运算为

$$\frac{n^3}{3} + n^2 - \frac{n}{3}.$$

加减运算次数为

$$\frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6}.$$

5.7.2 选主元 Gauss 消去法

我们知道, 只要系数矩阵 A 非奇异, 则线性方程组就存在唯一解, 但 Gauss 消去法却不一定有效.

例 5.5 求解线性方程组 $Ax = b$, 其中 $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, $b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

由于主元 $a_{11} = 0$, 因此 Gauss 消去法无法顺利进行下去.

在实际计算中, 即使主元都不为零, 但如果主元的值很小, 由于舍入误差的原因, 也可能会给计算结果带来很大的误差. 解决这个问题一个有效方法就是选主元, 具体做法就是: 在执行 Gauss 消去过程的第 k 步之前, 插入下面的选主元过程.

① 选取 **列主元**: $|a_{ik,k}^{(k)}| = \max_{k \leq i \leq n} \{|a_{i,k}^{(k)}|\}$

② 交换: 如果 $i_k \neq k$, 则交换第 k 行与第 i_k 行, 相应地, 也要交换 b 的第 k 个与第 i_k 个元素.

上面选出的 $a_{i_k,k}^{(k)}$ 就称为 **列主元**. 加入这个选主元过程后, 就不会出现主元为零的情况 (除非 A 是奇异的). 由此, Gauss 消去法就不会失效. 这种带选主元的 Gauss 消去法就称为 **列主元 Gauss 消去法** 或 **部分选主元 Gauss 消去法** (Gaussian Elimination with Partial Pivoting, GEPP). 这是当前求解中小规模线性方程组的首选方法.

5.8 上机练习

练习 5.1 编写函数, 实现矩阵乘积的 Strassen 算法, 在主函数中生成两个 $n \times n$ 的随机整数矩阵进行测试, 比较与普通矩阵乘积的计算效率 (消耗的时间), 取 2^{10} . (程序取名 `hw06_01.cpp`)




```
void matrix_prod_strassen(double * px, double * py, double * pz, int n);
```

练习 5.2 编写函数, 实现求解线性方程组的 Gauss 消去法, 并在主函数中进行测试: 求解线性方程组 $Hx = b$, 其中

$$H = [h_{ij}] \in \mathbb{R}^{8 \times 8}, \quad h_{ij} = \frac{1}{i+j-1}, i, j = 1, 2, \dots, 8, \quad b = [1, 1, \dots, 1]^T.$$

(程序取名 hw06_2.cpp)

```
void GE(double A[n][n], double b[n], double x[n], int n);
```

练习 5.3 编写函数, 实现求解线性方程组的 **列主元 Gauss 消去法**, 并在主函数中进行测试: 求解线性方程组 $Ax = b$, 其中

$$A = \begin{bmatrix} 0 & 1 & 1 & & & & & \\ 1 & 0 & 1 & 1 & & & & \\ 1 & 1 & 0 & 1 & 1 & & & \\ & 1 & 1 & 0 & 1 & 1 & & \\ & & 1 & 1 & 0 & 1 & 1 & \\ & & & 1 & 1 & 0 & 1 & 1 \\ & & & & 1 & 1 & 0 & 1 \\ & & & & & 1 & 1 & 0 \end{bmatrix}_{8 \times 8}, \quad b = [1, 1, \dots, 1]^T \in \mathbb{R}^8.$$

(程序取名 hw06_3.cpp)

```
void GEPP(double * a, double * b, double * x, int n);
```



第六讲 文件读写操作

6.1 文件的打开和关闭

- 文件分类: 按存储方式 (即数据的组织形式), 文件可以分为文本文件和二进制文件.
- 文件读写分三个步骤: (1) 打开文件, (2) 进行相应的读取或写入操作, (3) 关闭文件.
- 在 C 语言中, 用 `fopen` 打开文件 (首先声明一个文件指针)

```
FILE * pf; // 声明一个文件指针
pf=fopen(文件名, 打开方式);
```

- (1) `fopen` 按指定的方式打开文件, 返回一个文件指针, 之后就可以通过该文件指针进行读写操作.
- (2) 文件指针由关键字 `FILE` 声明, 是库文件 `stdio.h` 中定义的一种特殊数据类型.
- (3) 文件名: 普通字符串, 可包含路径, `fopen` 将文件名对应的文件与文件指针 `pf` 绑定在一起.
- (4) 打开方式: 指定读写方式和文件类型, 取值有

```
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
```

- (5) 若文件打开成功, 返回指向文件的指针; 若打开失败, 则返回一个空指针 (`NULL`)
- 文件关闭:

```
fclose(pf);
```

- (1) 正常关闭则返回值为 `0`; 出错时, 返回值为非 `0`

6.2 文本文件的读写

- 写文本文件: `fprintf`

```
fprintf(pf, "格式控制字符串", 输出变量列表); // fprintf 的用法与 printf 类似
```

- 读文本文件: `fscanf`

```
fscanf(pf, "格式控制字符串", 地址列表); // 注意最后一个参数是地址
```

- 从文件中读取数据 (可以是数值型或字符型).
- 如果是读取数值型数据, 则建议文件中不要含有字符型数据, 并且每个数据单独占一行.
- 如果是读取浮点数, 则格式控制字符串要使用 `%lf`, 不是 `%f`.
- 如果是读取字符串, 则缺省以空格为结束符.

例 6.1 C 语言文件读写: 文本文件.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
```

```

6  FILE * pf;
7  char str[20];
8
9  pf = fopen("out.txt","wt");
10 fprintf(pf,"MATH ECNU"); // 写文件
11 fclose(pf);
12
13 pf = fopen("out.txt","rt");
14 fscanf(pf,"%s", str);    // 读文件
15 fclose(pf);
16 printf("str=%s\n",str);
17
18 return 0;
19 }

```

- 其他文本文件读写函数: `fputc`, `fputs`, `fgetc`, `fgets`

6.2.1 二进制文件的读写

- 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)

例 6.2 C 语言文件读写: 二进制文件.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int A[3][3]={11,12,13},{21,22,23},{31,32,33}};
7      int i, j;
8      FILE * pf;
9
10     pf = fopen("data1.dat","wb");
11     fwrite(A,sizeof(int),9, pf); // fwrite(A,sizeof(A),1, pf);
12     fclose(pf); // 一定要先关闭, 后面读的时候再打开
13
14     pf = fopen("data1.dat","rb");
15     int B[3][3];
16     fread(B,sizeof(int),9, pf);
17     fclose(pf);
18
19     printf("A=\n");
20     for (i=0; i<3; i++)
21     {

```



```

22     for (j=0; j<3; j++)
23         printf("%4d", A[i][j]);
24     printf("\n");
25 }
26
27 printf("\nB=\n");
28 for (i=0; i<3; i++)
29 {
30     for (j=0; j<3; j++)
31         printf("%4d", B[i][j]);
32     printf("\n");
33 }
34
35 return 0;
36 }

```

6.2.2 其他文件操作

文件打开后, 默认是从最前面开始读或者写. 有时可能需要从中间某个地方读取数据, 此时需要进行一些位移操作.

- **fseek** 可以像对待数组那样对待文件

```
fseek(pf, offset, whence);
```

- (1) **pf** 是文件指针, **offset** 是偏移量 (相对于 **whence**, 以字节为单位, 长整型 **long**, 可正可负)
- (2) **whence** 是位置, 取值有
SEEK_SET: 文件开头
SEEK_CUR: 当前位置
SEEK_END: 文件末尾

```

1  fseek(pf, 0L, SEEK_SET); // 定位到文件开头
2  fseek(pf, 10L, SEEK_SET); // 定位到文件第 10 个字节数
3  fseek(pf, -2L, SEEK_CUR); // 从当前位置后移 2 个字节
4  fseek(pf, -10L, SEEK_END); // 定位到文件末尾到第 10 个字节处

```

- **ftell** 返回当前位置, 即距离文件开头的字节数, 长整型.

```
long int ftell(FILE * pf);
```

- **feof** 判断是否到达文件末尾.

```
int feof(FILE * pf);
```

例 6.3 C 语言文件读写: feof

```

1 #include <iostream>
2 #include <cstdio>
3
4 using namespace std;
5
6 int main()

```



```
7 {  
8     FILE * pf;  
9     char str[20];  
10  
11     pf = fopen("namelist.txt", "rt"); // namelist.txt 存放姓名, 每人占一行  
12     while(!feof(pf))  
13     {  
14         fscanf(pf, "%s", str);  
15         cout << str << endl;  
16     }  
17     fclose(pf);  
18  
19     return 0;  
20 }
```



第七讲 预编译处理与多文件结构

7.1 预编译处理

- 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" // 先在当前目录中寻找，然后再按标准方式搜索
```

表 7.1. 常用头文件

stdio.h	输入输出 printf, scanf, fopen, fclose, fprintf, fscanf, sprintf
stdlib.h	rand, RAND_MAX, srand, malloc, calloc, realloc, free
stdbool.h	bool, true, false
stdint.h	int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, ...
math.h	数学函数
time.h	计时函数, clock, time, clock_t, time_t, CLOCKS_PER_SEC
string.h	字符串
ctype.h	字符操作

- 定义宏

```
1 #define PI 3.14159 // 定义宏
2 #undef PI // 删除由 #define 定义的宏
3 #define S PI*r*r // 宏也可以是表达式
4 #define MAX(a,b) (a>b) ? a:b // 宏也可以带参数
```

- 条件编译

```
#if 常量表达式
    程序正文 // 当“常量表达式”非零时编译
#endif
```

```
#if 常量表达式
    程序正文 // 当“常量表达式”非零时编译
#else
    程序正文 // 否则编译这段程序
#endif
```

```
#if 常量表达式1
    程序正文 // 当“常量表达式1”非零时编译
```

```
#elif 常量表达式2
    程序正文 // 否则, 当“常量表达式2”非零时编译
#elif 常量表达式3
    程序正文 // 否则, 当“常量表达式3”非零时编译
... ..
#else
    程序正文 // 否则编译这段程序
#endif
```

```
#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
```

7.2 多文件结构

一个程序可以由多个文件组成, 编译时可以使用工程/项目来组合. 若使用命令行编译, 则需要同时编译.

- 外部变量: 如果需要用到其它文件中定义的变量, 则需要用 `extern` 声明其为外部变量.


```
extern 类型名 变量名;
```

- 外部函数: 如果需要用到其它文件中定义的函数, 则需要用 `extern` 声明其为外部函数.

```
extern 函数声明/函数原型;
```

- 系统函数

- (1) 标准 C++ 函数 (库函数): 参见 <http://www.cppreference.com>
- (2) C 语言的系统库中提供了几百个函数可供程序员直接使用, 使用库函数时要包含相应的头文件;
- (3) 非标准 C 语言函数: 操作系统或编译环境提高的系统函数

 **注记:** 充分使用库函数不仅可以大大减少编程工作量, 还可以提高代码可靠性和执行效率.

7.3 编译选项

- 优化选项和版本选项: `-O2 -std=c11`



```
1 gcc -O2 -std=c11 -o hello hello.c
```

- Dev-C++ 设置方法见下图: 工具 -> 编译选项

