# 电子科技大学信息与软件工程学院

# 实 验 指 导 书

（实验）课程名称 C 语言程序设计

电子科技大学教务处制表

# 目　　录

实验一 C 语言的格式化输入/输出 ..... 3

实验二 C 语言的程序控制语句 ..... 13

实验三 C 语言的字符和数组处理技术 ..... 19

实验四 C 语言的程序结构和指针的用法 ..... 43

实验五 超市商品信息管理系统 ..... 67

# 实验一 C 语言的格式化输入/输出

## 一、实验目的

1. 掌握 C 语言的集成开发环境用法。
2. 掌握 C 语言的基本程序框架。
3. 掌握 C 语言的格式化输入/输出方法。
4. 掌握 C 语言表达式的用法

## 二、实验原理

### 2.1 C 程序的开发与运行环境

1、**运行C程序的步骤**（如图 1 所示 ）

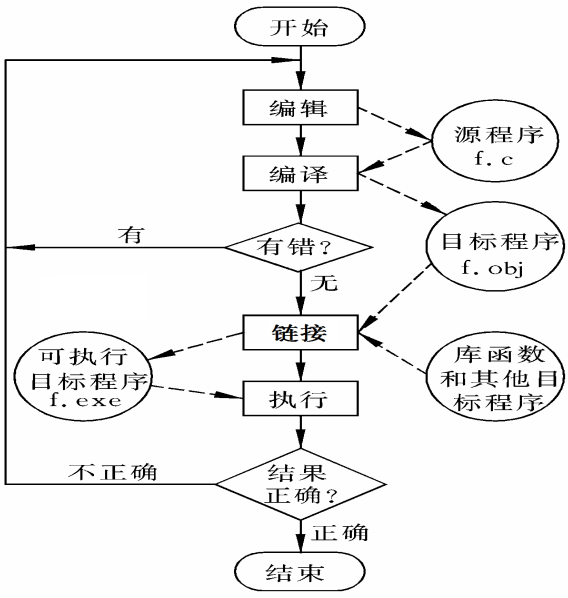



图 1 C 程序的运行步骤

具体步骤为：

①上机输入与编辑源程序

②预处理器 Preprocessor：执行以#开头的指令，类似于编辑器，可以添加和修改源程序。

③对源程序进行编译

④与库函数链接

⑤运行目标程序

## 2、常用的开发工具

①集成开发环境(IDE)对 C 程序进行编辑、编译、链接、执行甚至调试。

②Visual Studio 2010 的集成开发环境。

③轻量级的 IED：C-Free（http://www.programarts.com/cfree_ch/index.htm）

④GCC（GNU Compiler Collection）是最流行的编译器。

⑤GCC 是 Linux 环境下的编译器。

⑥Cygwin 是一个在 windows 平台上运行的 Linux 模拟环境。

⑦MinGW 是指只用自由软件来生成纯粹的 Win32 可执行文件的编译环境，它是 Minimalist GNU on Windows 的略称。

⑧Eclipse IDE for C/C++ Developers 。

## 3、Visual Studio 2010 开发环境搭建示例

①文件-新建-项目。

选择“Win32 控制台应用程序”，输入“名称”，点“确定”，如图 2 所示：

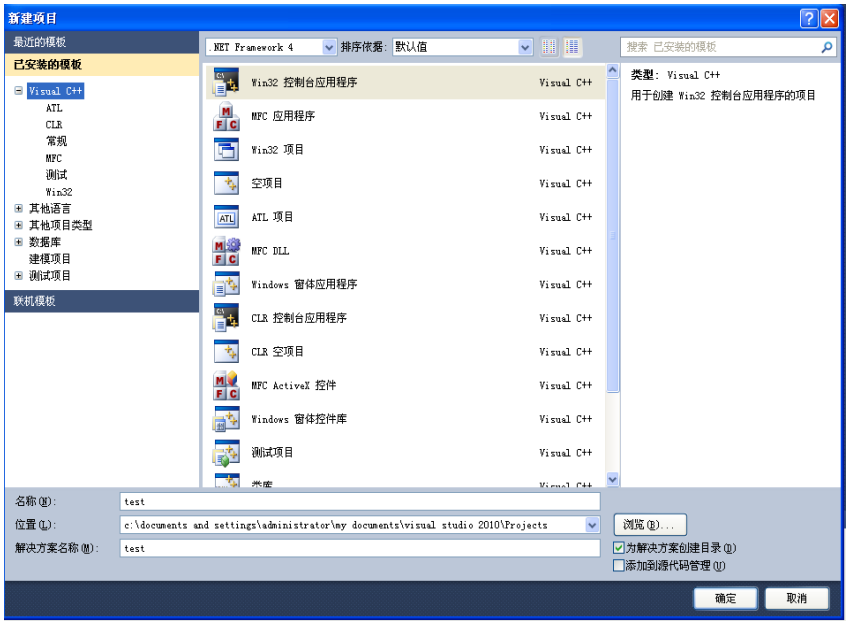



图 2 搭建步骤 1

②点“下一步”，在下面窗口选择“空项目”，点“完成”，如图 3 所示：

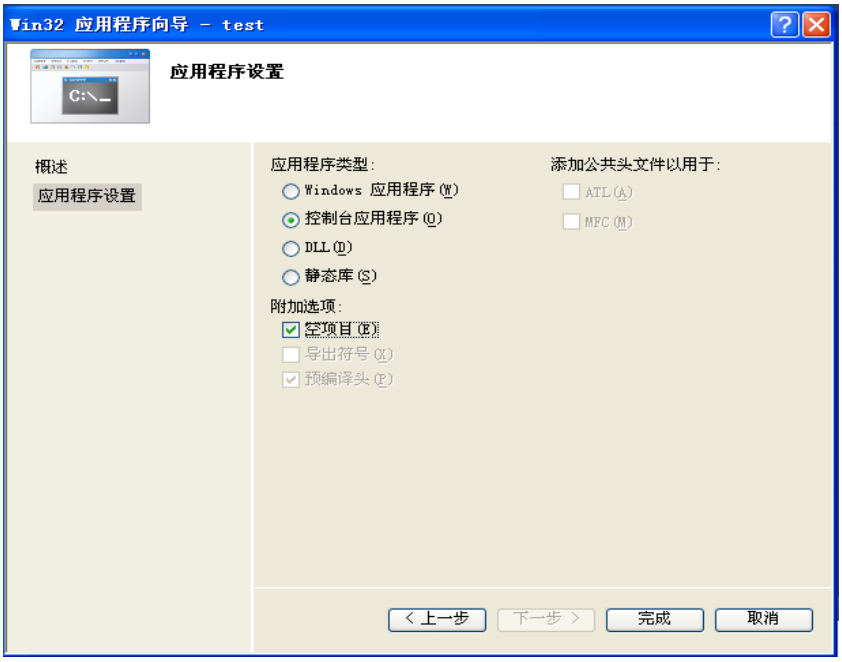



图 3 搭建步骤 2

③点鼠标右键 ，添加-新建项，如图 4 所示：

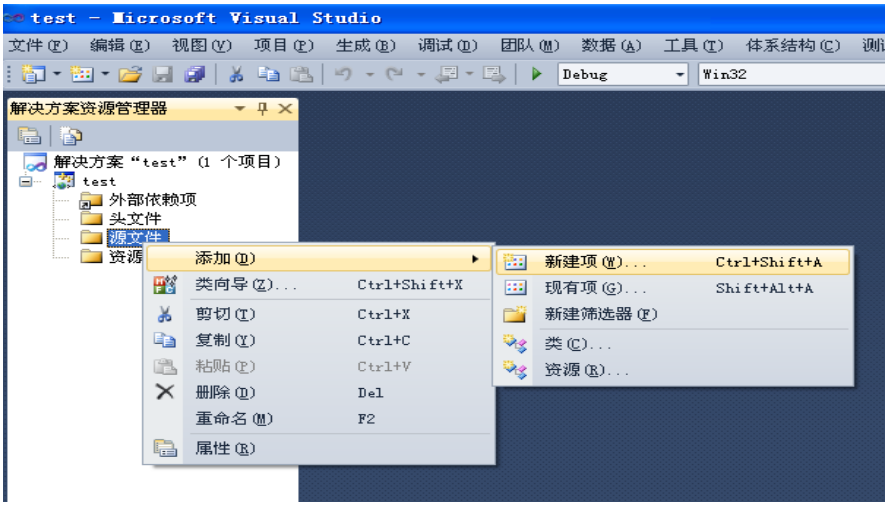



图 4 搭建步骤 3

④在下面窗口选择“C++文件（.cpp）”，名称输入“hello.c”，点“添加”。如图 5 所示：

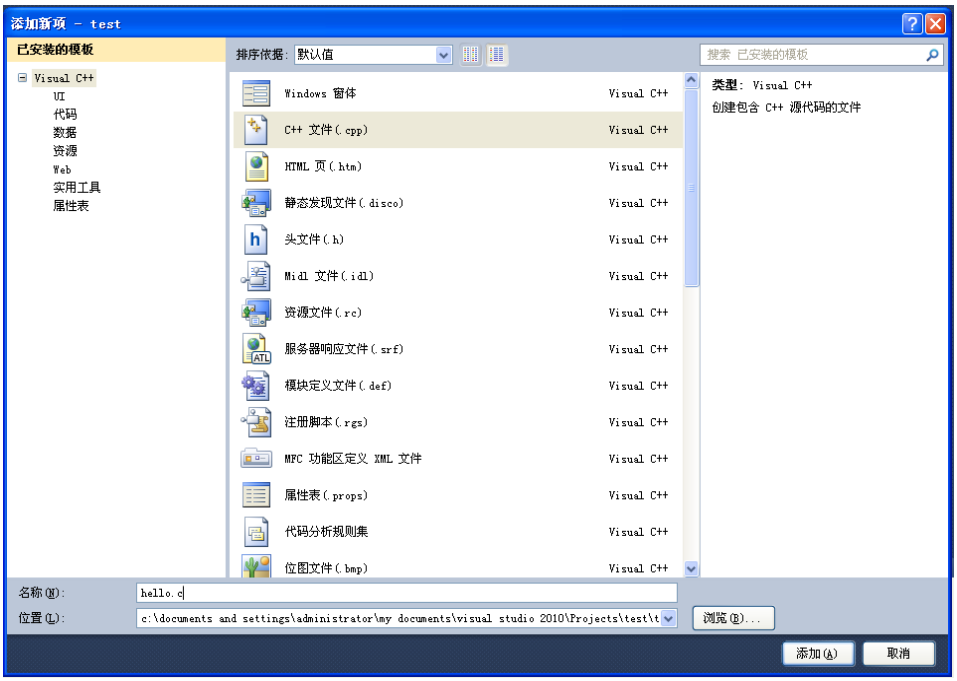



图 5 搭建步骤 4

⑤在 hello.c 中输入下列源程序，点“保存”，

```
#include <stdio.h>

void    main( )
{
    printf("Hello, This is a C program.\n");
    system("pause");              //增加该语句使字符界面可以暂时停留
}
```

⑥点菜单：调试-启动调试（或 F5），运行程序。

**4、标准格式输入函数 printf（）**

①功能介绍

基本功能：产生格式化输出的函数。

标准格式：int printf(格式化字符串,表达式 1,表达式 2,…);

一般格式：printf（“格式控制”，输出列表）;

※“格式控制字符串”中包含两种类型的字符

►格式字符 ——由%和格式字符组成。如%d、%c、%s、%f 等。将输出列表中的数据转换为指定的格式输出。

►普通字符 ——即需要原样输出的，除格式字符外的其它字符。按原样输出，在显示中起提示作用。

※ 输出表列：需要输出的一些数据，可以是表达式

如：printf(“%c,%s,%d\n”, c, “hello” ,a+b);

② 转换说明

| % | 标志 | m（最小字段宽度） | .n（精度） | 长度修饰符 l | 转换说明符 |
|---|---|---|---|---|---|

► 标志：

-：有-表示左对齐输出，如省略表示右对齐输出。

0：有 0 表示指定空位填 0，如省略指定空位不填。

► m：指域宽，即对应的输出项在输出设备上所占的最小字符数。

► .n：指精度，用于说明输出的实数的小数位数，缺省情况下 n=6。

► f 格式：用来输出实数（包括单、双精度），以小数形式输出。

%f：不指定宽度，整数部分全部输出并输出 6 位小数。

%-m.nf：输出共占 m 列，其中有 n 位小数，如数值宽度小于 m 右端补空格。

%m.nf：输出共占 m 列，其中有 n 位小数，如数值宽度小于 m 左端补空格。

► e 格式：以指数形式输出实数。可用以下形式：

%e：数字部分（又称尾数）输出 6 位小数。

%m.ne 和%-m.ne：m、n 和”-”字符含义与前相同。此处 n 指数据的数字部分的小数位数，m 表示整个输出数据所占的宽度。

► g 格式：自动选 f 格式或 e 格式中较短的一种输出，且不输出无意义的零。

► o 格式：把无符号数转换为八进制数（o）。

► x 格式：把无符号数转换为十六进制数（x）。

示例 1：

```
//tprintf.c
#include <stdio.h>
int main()
{
   int i;
   float x;
   i = 43;
   x = 839.21f;
   printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
   printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);
   printf("|%o|%x|%-5o|\n", i, i, i);
   return 0;
}
```

输出结果如图 6 所示：

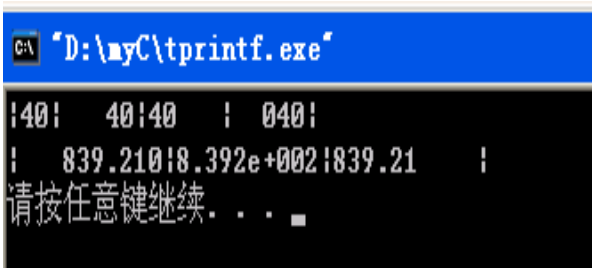



图 6　示例 1 输出结果

③转义序列

格式串中常用的“\n”，被称为转义序列（Escape Sequence）,意思是将（\）后面的字符转换成另外的意义。

常用的转义字符有：

警报（响铃）符：\a

回退符：\b

换行符：\n

水平制表符：\t

**5、标准格式输入函数 scanf()**

①功能介绍

基本功能：按指定的格式从标准输入设备 stdin（键盘）读取输入的数据。

一般格式：scanf(“格式控制串”,地址列表) ；

（注意：输入数据时不能规定精度,如: scanf(“%7.2f”,&a);是不合法的。）

② 工作方法

scanf 函数是由格式串控制的，调用时，scanf 函数从左边开始处理字符串中的信息。 对于格式串中的每一个转换说明，scanf 函数从输入的数据中定位适当类型的项，并在必要时跳过空格。

scanf 函数读入数据项，并在遇到不可能属于此项的字符时停止。 如果读入数据项成功，那么 scanf 函数会继续处理格式串中剩余部分；如果某一项不能成功读入，scanf 函数将不再查看格式串的剩余部分（或者余下的输入数据）而立即返回。

示例 2

在寻找数的起始位置时，scanf 函数会忽略空白字符（包括空格符、水平和垂直制表符、换页符和换行符。）

```
#include <stdio.h>
void main()
{
   int num1,num2;
   scanf("%d%d",&num1,&num2);
   printf("%d\t%d\n",num1,num2);
}
```


图 7　示例 2 输出结果

③ scanf 函数遵循什么规则识别整数或浮点数

在要求读入整数时， scanf 函数首先寻找正号或负号，然后读取数字直到读到一个非数字时才停止。

当要求读入浮点数时，scanf 函数会寻找一个正号或负号（可选），随后是一串数字（可能含有小数点），再后是一个指数（可选）。指数字母 e（E），可选的符号和一个或多个数字构成。

当 scanf 函数遇到一个不可能属于当前项的字符时，会把此字符“放回原处”，以便在扫描下一个输入项，或下一次调用 scanf 函数时再次读入。

示例 3 输入 1-20.3-4.0e3■

scanf (“%d%d%f%f”, &i, &j, &x, &y);

转换说明%d：第一个非空的输入字符是 1，因为整数可以以 1 开始，所以 scanf 函数接着读取下一个字符-，scanf 函数识别出字符-不能出现在整数内，所以把 1 存入变量 i 中，而把字符-，放回原处。

转换说明创%d：随后 scanf 函数读取字符-、2 、0 和.。因为整数不能包含小数点，所以 scanf 函数把-20 存入变量 j 中，而把字符．放回原处。

转换说明% f：接着 scanf 函数该取字符． 、3 和-。因为浮点数不能在数字后边有负号，所以 scanf 函数把 0.3 存入变量 x 中，而将字符-放回原处。

转换说明% f：最后 scanf 函数读取字符-、4 、.、0 、e、3 和■（换行符），浮点数不能包含换行符， 所以 scanf 函数把- 4.0×103 存入变量 y 中，而把换行符放回原处。

示例 4

```
#include <stdio.h>
void main()
{
    int num1,num2;
    scanf("%d,%d",&num1,&num2);
  printf("%d\t%d\n",num1,num2);
}
```

程序输出结果如图 8 所示:

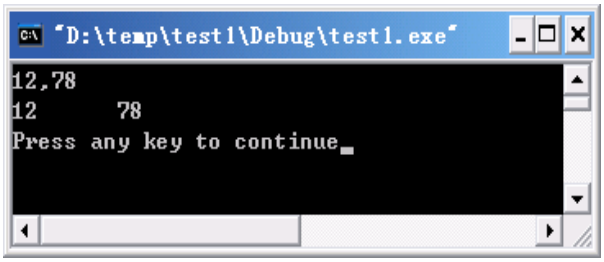



图 8　示例 4 输出结果

注意输入时避免如图 9 的错误:

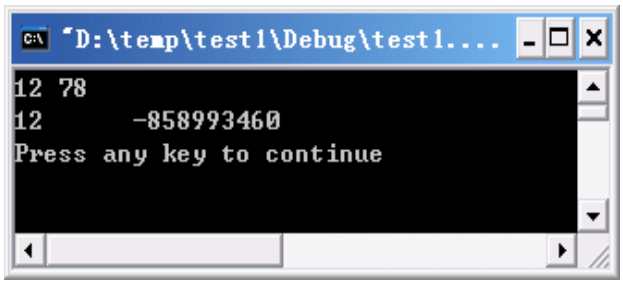



图 9　示例 4 错误输出结果

## 三、实验内容

项目 1：第 2 章编程题 4

项目 2：第 2 章编程题 7

项目 3：第 2 章编程题 8

项目 4：第 3 章编程题 1

项目 5：第 3 章编程题 6

## 四、实验步骤

1、明确项目需求

2、编写代码

3、编译代码

4、测试程序

5、根据测试结果对程序进行调试改进

# 实验二 C 语言的程序控制语句

## 一、实验目的

1. 掌握 C 语言的控制语句的用法；
2. 掌握 if、switch 语句的用法；
3. 掌握 while、do、for 语句的用法；
4. 掌握 break 语句的用法。

## 二、实验原理

1、逻辑表达式

包括 If 语句在内的一些 C 语句必须测试表达式的值为“真(true)”或“假(false)”。例如 if 语句要检测表达式 i<j，真值表明 i 小于 j。在许多编程语言里面类似 i<j 的表达式具有“布尔”类型或“逻辑”类型，值为真或假。而 C 语言的运算产生整数：0（假）或 1（真）。

2、if 语句

if 语句最简单的格式：

if (表达式) 语句

计算表达式的值，如果为真执行语句，为假执行后面的语句。

为了让 if 语句处理多条语句，就需要引入复合语句(compound statement)：

{多条语句}

例子：

```
if (line_num == MAX_LINES) {
    line_num = 0;
    page_num++; }
```

复合语句也出现在循环和其他需要多条语句，但 C 语言语法却要求是一条语句的地方。

带 else 子句的 if 语句格式：

if (表达式) 语句 1

else 语句 2

3、switch 语句

switch 语句比级联式 if 语句更容易阅读，执行速度也快。其格式如下：

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

控制表达式： switch 后边的表达式要求是整型（C 语言字符也是当成整数处理），不能用浮点数和字符串。

分支标号： case 常量表达式，常量表达式不能包含变量和函数调用。

语句：每个分支标号后可以跟任意数量的语句，不需要花括号，语句最后通常是 break 语句

多个分支可以共用一组语句，如下例所示：

```
switch (grade) {
    case 4:
    case 3:
    case 2:
    case 1:   printf("Passing");
              break;
    case 0:   printf("Failing");
              break;
    default: printf("Illegal grade");
              break;
}
```

4、while 语句

用于判定控制表达式在循环体执行之前的循环。使用 while 语句是最简单、最基本的设置循环方法。

While 语句格式如下：

while ( 表达式) 语句

表达式即为控制表达式；语句则是循环体。

while 语句示例：

```
while (i < n)    /* 控制表达式 */
  i = i * 2;     /* 循环体 */
```

执行 while 语句时，首先计算控制表达式的值。

如果值不为零（真），那么执行循环体，接着再次判定表达式。

这个过程持续进行直到最终控制表达式的值变为零。

while 语句示例：计算大于或等于数 n 的最小的 2 次幂：

```
i = 1;
while (i < n)
  i = i * 2;
```

5、do 语句

do 语句的一般格式如下：

```
do 语句 while ( 表达式 ) ;
```

执行 do 语句时，先执行循环体，再计算控制表达式的值。

如果表达式的值非零，那么再次执行循环体，然后再次计算表达式的值。

使用 do 语句重写前面的“倒数计数”程序：

```
i = 10;
do {
  printf("T minus %d and counting\n", i);
  --i;
} while (i > 0);
```

6、for 语句

for 语句适合应用在使用“计数”变量的循环中，然而它也灵活用于许多其他

类型的循环中。

for 语句的一般格式如下：

for ( 表达式 1; 表达式 2; 表达式 3 ) 语句

表达式 1、表达式 2 和表达式 3 全都是表达式。

示例：

```
for (i = 10; i > 0; i--)
  printf("T minus %d and counting\n", i);
```

7、退出循环

通常循环的退出点是在 while 或 for 循环体之前，或 do 循环体之后。

使用 break 语句可以在循环体中间设置退出点，甚至设置多个退出点。

## 三、实验内容

1. 第五章编程题 2：编写一个程序，要求用户输入 24 小时制的时间，然后显示 12 小时制的格式：

   Enter a 24-hour time: 21:11

   Equivalent 12-hour time: 9:11 PM

2. 第五章编程题 4：下面是用于测量风力的蒲福风力等级的简化版。

| 速率（海里/小时） | 描　　述 |
|---|---|
| 小于 1 | Calm（无风） |
| 1~3 | Light air（轻风） |
| 4~27 | Breeze（微风） |
| 28~47 | Gale（大风） |
| 48~63 | Storm（暴风） |
| 大于 63 | Hurricane（飓风） |

   编写一个程序，要求用户输入风速（海里/小时），然后显示相应的描述。

3. 第五章编程题 6：修改教材 4.1 节的 upc.c 程序，使其可以检测 UPC 的有

效性。在用户输入 UPC 后，程序将显示 VALID 或 NOT VALID。

4. 第五章编程题 10：利用 switch 语句编写一个程序，把用数字表示的成绩转化为字母表示的等级。

   Enter numerical grade：84
   Letter grade：B

   使用下面的等级评定规则：A 为 90~100，B 为 80~89，C 为 70~79，D 为 60~69，F 为 0~59。如果成绩高于 100 或低于 0 显示出错信息。提示：把成绩拆分成 2 个数字，然后使用 switch 语句判定十位上的数字。

5. 第六章编程题 2：编写程序，要求用户输入两个整数，然后计算这两个整数的最大公约数（GCD）：

   Enter two integers：12 28

   Greatest common divisor：4

   提示：求最大公约数的经典算法是 Euclid 算法，方法如下：分别让变量 m 和 n 存储两个数的值。如果 n 为 0，那么停止操作，m 中的值是 GCD；否则计算 m 除以 n 的余数，把 n 保存到 m 中，并把余数保存到 n 中。然后重复上述步骤，每次都先判断 n 是否为 0。

6. 第六章编程题 4：在 5.2 节的 broker.c 程序中添加循环，以便用户可以输入多笔交易并且程序可以计算每次的佣金。程序在用户输入的交易额为 0 是终止。

   Enter value of trade：30000

   Commission：$166.00

   Enter value of trade：20000

   Commission：$144.00

   Enter value of trade：0

7. 第六章编程题 6：编写程序，提示用户输入一个数 n，然后显示出 1~n 的所有偶数平方值。例如，如果用户输入 100，那么程序应该显示出下

列内容：

4

16

36

64

100

8. 第六章编程题 8：编写程序显示单月的日历。用户指定这个月的天数和该月起始日是星期几：

   Enter number of days in month：31

   Enter starting day of the week（1=Sum，7=Sat）：3

```
          1   2   3   4   5
 6   7   8   9  10  11  12
13  14  15  16  17  18  19
20  21  22  23  24  25  26
27  28  29  30  31
```

   提示：此程序不像看上去那么难。最重要的部分是一个使用变量 i 从 1 计数到 n 的 for 语句（这里 n 是此月的天数），for 语句中需要显示 i 的每个值。在循环中，用 if 语句判定 i 是否是一个星期的最后一天，如果是，就显示一个换行符。

## 四、实验步骤

1. 明确项目需求
2. 编写代码
3. 编译代码
4. 测试程序
5. 根据测试结果对程序进行调试改进。

# 实验三 C 语言的字符和数组处理技术

## 一、实验目的

1. 掌握 C 语言的字符处理技术。
2. 掌握 C 语言的数组使用方法和技术。
3. 掌握 C 语言函数的用法。

## 二、实验原理

### 2.1 字符处理

char 类型的值可以根据计算机的不同而不同，因为不同的机器可能会有不同的字符集。

当今最常用的字符集是 ASCII (美国信息交换标准码), 它用 7 位代码表示 128 个字符。ASCII 常被扩展为 8 位代码用于表示 256 个字符代码，被称为 Latin-1, 提供一些西欧和许多非洲语言所需的字符。

char 类型的变量能被赋值为任何单个字符：

```
char ch;

ch = 'a';     /* lower-case a */
ch = 'A';     /* upper-case A */
ch = '0';     /* zero         */
ch = ' ';     /* space        */
```

注意，字符常量需要用单引号括起来，而不是双引号。

字符操作:

在 C 语言中字符的操作非常简单，因为存在这样一个事实：C 语言会按小整数的方式处理字符。

在 ASCII 码中，字符的取值范围是 0000000 ~ 1111111,这个范围可以看成是 0~127 的整数。

字符 ‘a’ 的值为 97, ‘A‘的值为 65, ’0’ 的值为 48, and ‘ ’的值为 32。

字符常量实际上是 int 类型，而不是 char 类型

当计算中出现字符时，C 语言只是使用它对应的整数值。

考虑下面的例子，假设采用 ASCII 码字符集:

```
char ch;
int i;

i = 'a';            /* i is now 97     */
ch = 65;            /* ch is now 'A' */
ch = ch + 1;     /* ch is now 'B' */
ch++;               /* ch is now 'C' */
```

字符可以像数那样进行比较。

下面的 if 语句测试 ch 是否含有小写字母；如果有，那么它会把 ch 转化为相应的大写字母。

```
if ('a' <= ch && ch <= 'z')
   ch = ch - 'a' + 'A';
```

诸如‘a’ <= ch 这样的比较使用的是字符所对应的整数值

这些数值依据使用的字符集有所不同，所以程序使用<, <=, >, 和 >= 来进行字符比较可能不易移植。

字符拥有和数相同的属性，这一事实会带来一些好处。

如，for 语句中的控制变量可以简单采用大写字母：

```
for (ch = 'A'; ch <= 'Z'; ch++) …
```

以数的方式处理字符的缺点：

可能会导致编译器无法检查出来的错误。

导致编写出诸如‘a’ * ‘b’ / ‘c’的无意义的表达式。

可能会妨碍程序的可移植性，因为程序可能会基于一些对字符集的假设。

有符号和无符号字符

char 类型类似整型，存在有符号和无符号两种。

有符号字符通常的取值范围是–128 ~ 127，无符号型字符的取值范围则是

0~255。

一些编译器按有符号型处理字符，而另外一些编译器则将它们处理成无符号型数据。大多数时候，没有太大关系。

C 语言允许使用单词 signed 和 unsigned 来修饰 char 类型：

```
signed char sch;
unsigned char uch;
```

字符常量通常是用单引号括起来的字符。然而，一些特殊符号是无法采用上述这种书写方式的，比如换行符，因为它们是不可见的（无法打印的），或者无法从键盘输入的。转义序列提供了一种呈现这类特殊符号的方法。

转义序列共有两种：字符转义序列和数字转义序列。

转义序列

字符转义序列：

| 名称 | 转义序列 |
| --- | --- |
| Alert (bell) | \a |
| Backspace | \b |
| Form feed | \f |
| New line | \n |
| Carriage return | \r |
| Horizontal tab | \t |
| Vertical tab | \v |
| Backslash | \\ |
| Question mark | \? |
| Single quote | \' |
| Double quote | \" |

字符转义序列使用起来很方便，但是没有包含所有无法打印的 ASCII 字符。也无法用于表示基本的 128 个 ASCII 码字符以外的字符。

数字转义序列可以表示任何字符，所以它可以解决上述问题。

对特殊字符，数字转义序列使用这些字符的八进制或十六进制值。

例如，ASCII 码转义字符(十进制值为 27) 对应的八进制值为 33，对应的十六进制值为 1B

八进制转义序列由字符\和跟随其后的一个最多含有三位数字的八进制数组成，如 \33 或 \033.

十六进制转义序列由\x 和跟随其后的一个十六进制数组成，如\x1b 或\x1B.

其中 x 必须小写，不过十六进制的数字不限大小写。

作为字符常量使用时，转义序列必须用一对单引号括起来。

例如，一个表示成转义字符的常量可以写成'\33' (或 '\x1b').

转义序列可能有点隐晦，所以采用#define 的方式给它们命名通常会是不错的主意：

```
#define ESC '\33'
```

转义序列也可以潜入在字符串中使用。

字符处理函数

调用 C 语言的 toupper 库函数是更快捷、更易于移植的把小写字母转换成大写字母的方法：

```
ch = toupper(ch);
```

toupper 函数返回参数的大写形式.

程序调用 toupper 函数需要在顶部放置下面这条#include 指令：

```
#include <ctype.h>
```

C 函数库提供了其他有用的字符处理函数。

读/写字符

转换说明符%c 允许 scanf 和 printf 函数对单独一个字符进行读/写操作：

```
char ch;
scanf("%c", &ch);    /* reads one character */
printf("%c", ch);    /* writes one character */
```

scanf 函数不会跳过空白字符。

为了强制 scanf 函数在读入字符前跳过空白字符，需要在格式串转换说明%c

前面加上一个空格：

```
scanf(" %c", &ch);
```

读/写字符

因为通常情况下 scanf 函数不会跳过空白，所以它很容易检查到输入行的结尾：检查刚读入的字符是否为换行符。

下面的循环将读入并且忽略掉所有当前输入行中其余的字符：

```
do {
    scanf("%c", &ch);
} while (ch != '\n');
```

当下次调用 scanf 函数时，将读入下一输入行中的第一个字符。

读/写字符

对单个字符的输入和输出，可以使用 gechar 函数和 putchar 函数来代替调用 scanf 函数和 printf 函数。

putchar 函数写单独一个字符：

```
putchar(ch);
```

每次调用 gechar 函数将读并返回一个字符：

```
ch = getchar();
```

getchar 函数返回一个整数值而不是字符值。

和 scanf 函数一样，gechar 函数也不会再读取时跳过空白字符。

使用 gechar 和 puchar 函数代替 scanf 和 printf 函数可以节约执行时间。

这两个函数比 scanf 和 printf 函数简单，因为 scanf 和 printf 函数是设计来读写多种不同格式的类型数据的。

为了额外的速度提升，通常 getchar 函数和 putchar 函数是作为宏来实现的。

getchar 函数还有另一个优点，因为返回的是读入的字符，所以 getchar 函数可以应用在多种不同的 C 语言惯用法中。

思考下面这个 scanf 函数循环，它用来跳过输入行的剩余部分：

```
do {
   scanf("%c", &ch);
} while (ch != '\n');
```

用 getchar 函数重写上述循环：

```
do {
    ch = getchar();
} while (ch != '\n');
```

为了精简循环，允许把 gechar 函数的调用放入到循环的控制表达式中：

```
while ((ch = getchar()) != '\n')
    ;
```

甚至变量 ch 都可以不需要，直接把 getchar 函数的返回值与换行符进行比较：

```
while (getchar() != '\n')
    ;
```

为了精简循环，允许把 gechar 函数的调用放入到循环的控制表达式中：

```
while ((ch = getchar()) != '\n')
    ;
```

甚至变量 ch 都可以不需要，直接把 getchar 函数的返回值与换行符进行比较：

```
while (getchar() != '\n')
    ;
```

getchar 函数在用于循环中搜寻字符时和跳过字符一样有效。

利用 getchar 函数跳过无线数量的空格字符：

```
while ((ch = getchar()) == ' ')
    ;
```

当循环终止时，变量 ch 将包含 getchar 函数遇到的第一个非空字符。

当混用 getchar 函数和 scanf 函数时要小心。

scanf 函数有一种留下后边字符的趋势，即对于输入后面的字符（包括换行符）只是看一下，并没有读入。

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

在读入 i 的同时，scanf 函数调用将会留下后面没有消耗掉的任意字符，包

括换行符（但不仅限于换行符）。

getchar 函数随后将取回第一个剩余字符。

程序：确定消息的长度

程序 length.c 显示用户输入消息的长度：

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

消息的长度包括空格和标点符号，但是不包含消息结尾处的换行符。

我们即可以采用 scanf 函数也可以采用 getchar 函数读取字符，但大多数 C 程序员愿意采用 getchar 函数。

## 2.2 数组

数组(array)是含有多个数据值的数据结构，并且每个数据值具有相同的数据类型。

这些数据值被称为元素(element)，可以根据元素所处的位置对其进行单独选择。

最简单的数组类型就是一维数组。

一维数组中的元素一个接一个地编排在单独一行(或者一列)内。

为了声明一个数组，需要说明数组元素的类型和数量：

```
int a[10];
```

数组的元素可以是任何类型；数组的长度可以用任何(整数)常量表达式说明。

较好的方法是用宏来定义数组的长度：

```
#define N 10
…
int a[N];
```

为了存取特定的数组元素，可以在写数组名的同时在后边加上一个用方括号围绕的整数值。这被称为对数组进行下标(subscripting)或索引(indexing)。

长度为 n 的数组，其元素的索引是从 0 到 n-1。

如果 a 是一个长为 10 的数组，则其元素可标记为 a[0], a[1], …, a[9]:

a[i]的表达式格式是左值，所以数组元素可以和普通变量一样使用。

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

一般来说，如果一个数组所包含元素的类型为 T，则数组的每个元素都可以被当做一个类型为 T 的变量来对待。

许多程序所包含的 for 循环都是为了对数组中的每个元素执行一些操作。

下面是关于长度为 N 的数组的一些典型操作示例:

```
for (i = 0; i < N; i++)
  a[i] = 0;                    /* clears a */

for (i = 0; i < N; i++)
  scanf("%d", &a[i]);     /* reads data into a */

for (i = 0; i < N; i++)
  sum += a[i];                 /* sums the elements of a */
```

C 语言不要求检查下标的范围;当下标超出范围时，程序可能执行不可预知的行为。

一个典型的错误：忘记了对 n 个元素数组的索引是从 0 到 n-1，而不是从 1 到 n。

```
int a[10], i;

for (i = 1; i <= 10; i++)
  a[i] = 0;
```

对于某些编译器来说，这个表面上正确的 for 语句却产生了一个无限循环。

数组下标可以是任何整数表达式：

```
a[i+j*10] = 0;
```

表达式甚至可能会有副作用：

```
i = 0;
while (i < N)
  a[i++] = 0;
```

当数组下标有副作用时一定要注意：

```
i = 0;
while (i < N)
  a[i] = b[i++];
```

表达式 a[i] = b[i++] 访问了 i 的值并且修改了 i，结果导致不可预知的行为。

通过从下标中移走自增操作的方法可以很容易避免此类问题的发生：

```
for (i = 0; i < N; i++)
  a[i] = b[i];
```

程序：数列反向

reverse.c 程序要求用户录入一串数，然后按反向顺序输出这些数：

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 102 49 82 34
```

程序在读入数时将其存储在一个数组中，然后通过数组反向开始一个接一个地显示出数组元素。reverse.c

```
/* Reverses a series of numbers */
 #include <stdio.h>
 #define N 10

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = N - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");
```

```
    return 0;
}
```

像其他变量一样，数组也可以在声明时获得一个初始值。

数组初始化式(array initializer)最通用的格式是一个常量表达式列表，列表用大括号括起来，并且内部数值用逗号进行分隔：

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

如果初始化式比数组短，那么数组中剩余的元素赋值为 0：

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

利用这一特性，可以很容易地给全部数组元素初始化为零：

```
int a[10] = {0};
/* initial value of a is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

初始化式完全为空是非法的，所以要在大括号内放上一个单独的 0。

初始化式长过要初始化的数组也是非法的。

如果显示一个初始化式，那么可以忽略掉数组的长度：

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

编译器利用初始化式的长度来确定数组的大小。

程序：检查数中重复出现的数字

程序 repdigit.c 用来检查数中是否有出现多于一次的数字。

用户输入数后，程序显示信息 Repeated digit 或者 No repeated digit：

```
Enter a number: 28212
Repeated digit
```

数 28212 有一个重复的数字(2)；而数 9357 则没有。

程序：检查数中重复出现的数字

程序采用布尔型值的数组跟踪数中出现的数字。

最初的时候，digit_seen 中每个元素的值都为假。

当给出数 n 时，程序一次一个地检查 n 的数字，并且把每次的数字存储在变量 digit 中。

如果 digit_seen[digit]为真，那么表示 digit 至少在 n 中出现了两次。

如果 digit_seen[digit]为假，那么表示 digit 那么表示，因此程序会把 digit_seen[digit]设置为真并且继续执行。repdigit.c

```
/* Checks numbers for repeated digits */

#include <stdbool.h>     /* C99 only */
#include <stdio.h>

int main(void)
{
  bool digit_seen[10] = {false};
  int digit;
  long n;

  printf("Enter a number: ");
  scanf("%ld", &n);
  while (n > 0) {
    digit = n % 10;
    if (digit_seen[digit])
      break;
    digit_seen[digit] = true;
    n /= 10;
  }

  if (n > 0)
    printf("Repeated digit\n");
  else
    printf("No repeated digit\n");
```

```
    return 0;
}
```

对数组使用 sizeof 运算符

运算符 sizeof 可以确定数组的大小(字节数)。

如果数组 a 有 10 个整数，那么 sizeof(a)可以代表 40(假设每个整数用 4 字节存储)。

还可以用 sizeof 来计算数组元素的大小，比如 a[0]。

用数组的大小除以数组元素的大小可以得到数组的长度：

```
sizeof(a) / sizeof(a[0])
```

对数组使用 sizeof 运算符

当需要数组长度时，一些程序员采用上述这个表达式。

数组 a 的清零操作可以写成：

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
    a[i] = 0;
```

利用这种技术，即使数组长度在日后需要改变，也不需要改变循环。

对数组使用 sizeof 运算符

有些编译器会对 i < sizeof(a) / sizeof(a[0])表达式给出一条警告信息。

变量 i 的类型可能是 int (有符号类型)，而 sizeof 返回值的类型是 size_t (一种无符号类型)。

将有符号整数和无符号整数比较是很危险的，尽管在本例中这样做没问题。

为避免这一警告，可以把 sizeof(a) / sizeof(a[0]) 强制转化成有符号整数：

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
    a[i] = 0;
```

定义一个宏来表示上述表达式常常是很有帮助的：

```
#define SIZE ((int) (sizeof(a) / sizeof(a[0])))

for (i = 0; i < SIZE; i++)
    a[i] = 0;
```

程序：计算利息

程序 interest.c 打印出一个表格，这个表格显示了在几年时间内 100 美金投资在不同利率上的价值。

用户将输入利率和要投资的年数。

假设整合利息一年一次，表格将显示出一年间在此输入利率下和后边 4 个更高利率下投资的价值。

程序：计算利息

下面是程序运行时的情况：

```
Enter interest rate: 6
Enter number of years: 5

Years        6%        7%        8%        9%       10%
  1       106.00 107.00 108.00 109.00 110.00
  2       112.36 114.49 116.64 118.81 121.00
  3       119.10 122.50 125.97 129.50 133.10
  4       126.25 131.08 136.05 141.16 146.41
  5       133.82 140.26 146.93 153.86 161.05
```

第二行的数值要依赖于第一行的数，解决方案是把第一行的数存储在数组中。

数组中的这些值将用于计算第二行的内容。

从第三行到最后一行可以重复这个过程。

程序使用了嵌套的 for 语句。

外层循环将从 1 计数到用户要求的年数。

内层循环将从利率的最低值自增到最高值。interest.c

```
/* Prints a table of compound interest */

#include <stdio.h>

#define NUM_RATES ((int) (sizeof(value) / sizeof(value[0])))
#define INITIAL_BALANCE 100.00
```

```
int main(void)
{
    int i, low_rate, num_years, year;
    double value[5];

    printf("Enter interest rate: ");
    scanf("%d", &low_rate);
    printf("Enter number of years: ");
    scanf("%d", &num_years);

    printf("\nYears");
    for (i = 0; i < NUM_RATES; i++) {
        printf("%6d%%", low_rate + i);
        value[i] = INITIAL_BALANCE;
    }
    printf("\n");

    for (year = 1; year <= num_years; year++) {
        printf("%3d        ", year);
        for (i = 0; i < NUM_RATES; i++) {
            value[i] += (low_rate + i) / 100.0 * value[i];
            printf("%7.2f", value[i]);
        }
        printf("\n");
    }

    return 0;
}
```

## 2.3 函数

函数简单来说就是一连串组合在一起并且命名的语句。每个函数本质上是一个自带声明和语句的小程序。

函数的优点：

可以利用函数把程序划分成小块，这样便于人们理解和修改程序。

可以避免重复编写可多次使用的代码。

一个函数最初可能是某个程序的一部分，但可以将其用于其他程序中。

函数定义：

一些程序员习惯把返回类型放在函数名的上边：

```
double
average(double a, double b)
{
   return (a + b) / 2;
}
```

如果返回类型很冗长，比如 unsigned long int 类型，那么把返回类型单独放在一行是非常有用的。

函数名后边有一串形式参数列表。

每个形式参数需要说明其类型；形式参数间用逗号进行分隔。

如果函数没有形式参数，那么在圆括号内应该出现 void。

函数体可以包含声明和语句。

average 函数的变体：

```
double average(double a, double b)
{
   double sum;            /* declaration */

   sum = a + b;           /* statement */
   return sum / 2;      /* statement */
}
```

函数体内声明的变量专属于此函数，其他函数不能对这些变量进行检查或修改。

在 C89 中，变量声明必须出现在语句之前。

在 C99 中，变量声明和语句可以混在一起，只要变量在第一次使用前进行声明即可。

返回类型为 void 的函数（“void 函数”）的函数体可以为空：

```
void print_pun(void)
{
}
```

在程序开发过程中，作为一种临时措施，留下空函数体是有意义的。

函数调用由函数名和跟随其后的实际参数列表组成，其中实际参数列表用圆括号括起来：

```
average(x, y)
print_count(i)
print_pun()
```

如果丢失圆括号，那么将无法进行函数调用：

```
print_pun;     /*** WRONG ***/
```

这条语句是合法的，但是不起作用。

void 型的函数调用是语句，所以调用后边始终跟着分号：

```
print_count(i);
print_pun();
```

非 void 型的函数调用产生的值可存储在变量中，还可以进行测试、显示或者其他用途：

```
avg = average(x, y);
if (average(x, y) > 0)
   printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

如果不需要，非 void 型函数的返回值总是可以丢弃：

```
average(x, y);   /* discards return value */
```

这个调用是一个表达式语句的示例：一个计算出了表达式值但是将其丢弃的语句。

如果不需要，非 void 型函数的返回值总是可以丢弃：

```
average(x, y);  /* discards return value */
```

这个调用是一个表达式语句的示例：一个计算出了表达式值但是将其丢弃的语句。

丢弃 average 的返回值是一件很奇怪的事，但在某些情况下是有意义的：

例如：printf 返回显示的字符的个数。

在下面的调用后，num_chars 的值为 9：

```
num_chars = printf("Hi, Mom!\n");
```

通常会丢掉 printf’s 的返回值：

```
printf("Hi, Mom!\n");
  /* discards return value */
```

为了清楚地表示故意丢掉函数返回值，C 语言允许在函数调用前加上(void)：

```
(void) printf("Hi, Mom!\n");
```

使用(void)可以使别人清楚编写者是故意扔掉返回值的，而不是忘记了。

程序：判定素数

程序 prime.c 检查一个数是否是素数：

```
Enter a number: 34
Not prime
```

程序使用名为 is_prime 的函数来进行检查。该函数返回值为 true 就表示它的形式参数是素数，返回 false 就表示它的形式参数不是素数。

给定数 n 后，is_prime 函数把 n 除以从 2 到 n 的平方根之间的每一个数：如果余数永远为 0，就知道 n 不是素数。prime.c

```
/* Tests whether a number is prime */
#include <stdbool.h>      /* C99 only */
#include <stdio.h>

bool is_prime(int n)
```

```
{
    int divisor;

    if (n <= 1)
        return false;
    for (divisor = 2; divisor * divisor <= n; divisor++)
        if (n % divisor == 0)
            return false;
    return true;
}

int main(void)
{
    int n;

    printf("Enter a number: ");
    scanf("%d", &n);
    if (is_prime(n))
        printf("Prime\n");
    else
        printf("Not prime\n");
    return 0;
}
```

实际参数：

在 C 语言中，实际参数是通过值传递的：调用函数时，计算出每个实际参数的值并且把它赋值给相应的形式参数。

在函数执行过程中，对形式参数的改变不会影响实际参数的值，这是因为形式参数中包含的是实际参数值的副本。

实际参数按值传递既有利也有弊。

既然形式参数的修改不会影响到相应的实际参数，那么可以把形式参数作为函数内的变量来使用，因此可以减少真正需要的变量的数量。

思考下面这个函数，此函数用来计算数 x 的 n 次幂：

```
int power(int x, int n)
{
  int i, result = 1;

  for (i = 1; i <= n; i++)
    result = result * x;

  return result;
}
```

因为 n 只是原始指数的副本，所以可以在函数体内修改它，因此就不需要使用变量 i 了：

```
int power(int x, int n)
{
  int result = 1;

  while (n-- > 0)
    result = result * x;

  return result;
}
```

C 语言关于实际参数按值传递的要求使它很难编写某些类型的函数。

假设我们需要一个函数，它将把 double 型的值分解成整数部分和小数部分。

因为函数无法返回两个数，所以可以尝试把两个变量传递给函数并且修改它们：

```
void decompose(double x, long int_part, double frac_part)
{
  int_part = (long) x;
```

```
    frac_part = x - int_part;
}
```

假设采用下面的方法调用这个函数：　　decompose(3.14159, i, d);

可惜的是，变量 i 和 d 不会因为赋值给 int_part 和 frac_part 而受到影响。

实际参数的转换：

C 语言允许在实际参数的类型与形式参数的类型不匹配的情况下进行函数调用。

管理如何转换实际参数的规则与编译器是否在调用前遇到函数(或者函数的完整定义)的原型有关：

编译器在调用前遇到原型。

就像使用赋值一样，每个实际参数的值被隐式地转换成相应形式参数的类型。

例如：如果把 int 类型的实际参数传递给期望得到 double 型数据的函数，那么会自动把实际参数转换成 double 类型。

编译器在调用前没有遇到原型。

编译器执行默认的实际参数提升：

把 float 型的实际参数转换成 double 类型。

执行整数的提升，即把 char 型和 short 型的实际参数转换成 int 型(在 C99 中实现了整数提升)。

依赖默认的实际参数提升是危险的。

例如：

```
#include <stdio.h>

int main(void)
{
    double x = 3.0;
    printf("Square: %d\n", square(x));

    return 0;
}
```

```
int square(int n)
{
  return n * n;
}
```

在调用 square 函数时，编译器没有遇到原型，所以不知道该函数期望有 int 类型的实际参数。

取而代之的，编译器在变量 x 上执行了没有效果的默认的实际参数提升。

因为 square 函数期望有 int 类型的实际参数，但是却获得了 double 类型值，所以 square 函数将产生无效的结果。

把 square's 的实际参数强制转换为正确的类型可解决这个问题：

```
printf("Square: %d\n", square((int) x));
```

当然更好的解决方案是在调用 square 函数前提供其原型。

在 C99 中，调用 square 之前不提供声明或者定义是错误的。

当形式参数是一维数组时，可以不说明数组的长度：

```
int f(int a[])  /* no length specified */
{
  …
}
```

C 语言没有为函数提供任何简便的方法来确定传递给它的数组的长度。

但是，如果函数需要，必须把长度作为额外的实际参数提供出来。

例子：

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];

  return sum;
}
```

因为 sum_array 需要知道 a 的长度，所以我们需要将其作为第二个参数提供出来。

sum_array 函数的原型形式如下：

```
int sum_array(int a[], int n);
```

通常情况下，如果愿意可以忽略形式参数的名字：

```
int sum_array(int [], int);
```

在调用 sum_array 函数时，第一个参数是数组的名字，而第二个参数是这个数组的长度。

```
#define LEN 100

int main(void)
{
  int b[LEN], total;
  …
  total = sum_array(b, LEN);
  …
}
```

注意，在把数组名传递给函数时，不要在数组名的后边放置方括号：

```
total = sum_array(b[], LEN);     /*** WRONG ***/
```

函数无法检测通过传递是否获得了正确的数组长度。

人们可以利用这个事实，方法是告诉函数数组比实际小得多。

假设虽然数组 b 可以拥有 100 个元素，但是实际仅存储了 50 个元素。

通过书写下列语句可以对数组的前 50 个元素进行求和：

```
total = sum_array(b, 50);
```

注意不要告诉函数，数组型实际参数要比实际的大：

```
total = sum_array(b, 150);       /*** WRONG ***/
```

sum_array 函数将超出数组的末尾，导致不可知的行为。

函数可以改变数组型形式参数的元素，且改变会在相应的实际参数中体现出来。

下面的函数通过在每个数组元素中存储 0 来修改数组：

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

数组型实际参数：

调用 store_zeros:

```
store_zeros(b, 100);
```

数组型实际参数的元素可以修改似乎与 C 语言中实际参数的值传递相矛盾。

如果形式参数是多维数组，则只有第一维的长度可以省略。

如果我们修改 sum_array 函数，使得 a 是个二维数组，则必须指定 a 中列的数量：

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```

## 三、实验内容

项目 1：第 7 章编程题 4

项目 2：第 7 章编程题 12

项目 3：第 8 章编程题 2

项目 4：第 8 章编程题 9

项目 5：第 8 章编程题 15

项目 6：第 9 章编程题 3

## 四、实验步骤

1、明确项目需求

2、编写代码

3、编译代码

4、测试程序

5、根据测试结果对程序进行调试改进

# 实验四 C 语言的程序结构和指针的用法

## 一、实验目的

1. 掌握 C 语言的程序结构。
2. 掌握 C 语言指针的用法。
3. 掌握 C 语言指针和数组的关系。
4. 掌握 C 语言字符串的用法。

## 二、实验原理

### 4.1 程序结构

**1.局部变量**

（1）在函数体内声明的变量称为该函数的局部变量:

```
int sum_digits(int n)
{
int sum = 0;      /* local variable */

while (n > 0) {
      sum += n % 10;
      n /= 10;
}

return sum;
}
```

（2）局部变量的缺省性质:

自动存储期限。局部变量的存储单位是在包含该变量的函数被调用时“自动”分配的，函数返回时收回分配。

块作用域。局部变量的作用域是从变量声明的点开始一直到所在函数体的末尾。

由于 C99 并不要求在函数一开始就进行变量声明，所以局部变量的作用域可能非常小:

（3）静态局部变量

在局部变量声明中放置单词 static 可以使变量具有静态存储期限。

因为具有静态存储期限的变量拥有永久的存储单位，所以在整个程序执行期间都会保留变量的值。

Example:

```
void f(void)
{
static int i;      /* static local variable */
…
}
```

静态局部变量仍然具有块作用域特性，所以它对于其它函数是不可见的。

（4）形式参数

形式参数拥有和局部变量一样的性质：自动存储期限和块作用域。在每次函数调用时，对形式参数自动进行初始化（通过赋值获得相应实际参数的值）。

**2. 外部变量**

“参数传递”是向函数传递信息的一种方法。函数还可以通过外部变量进行通信——外部变量是声明在任何函数体外的。外部变量有时也被称为全局变量。

（1）外部变量的性质：

- 静态存储期限
- 文件作用域

变量拥有文件作用域是指：变量的可见范围从变量被声明的点开始一直到所在文件的末尾。

（2）外部变量的利与弊

在多个函数必须共享一个变量时，或少数几个函数需要共享大量变量时，采用外部变量十分方便。然而在多数情况下，在函数间通过形式参数进行通信比通过共享变量的方法更好：

- 在程序维护期间，如果改变外部变量（比方说改变它的类型），那么将需要检查同一文件中的每个函数，以确认该变化如何对函数产生影响；
- 如果外部变量被赋了错误的值，可能很难确定出错的函数；
- 很难在其它程序中复用依赖于外部变量的函数。

另外，不要在不同的函数中为不同的目的使用同一个外部变量。此外，把应该是局部变量的变量声明为外部变量可能导致一些令人烦恼的错误。

**3. 程序块**

在前述实验中我们见到过如下形式的复合语句：

{ *statements* }

C语言还允许包含声明的复合语句：

{ *declarations statements* }

我们称这种类型的复合语句为：程序块

Example of a block:

```
if (i > j) {
/* swap values of i and j */
int temp = i;
i = j;
j = temp;
}
```

默认情况下，声明在程序块中的变量的存储期限是自动的：进入程序块时为变量分配存储单元，退出程序块时收回分配的空间。

变量具有块作用域，意味着变量不能在程序块外被引用。

程序块中的变量可以被声明为static ，使之具有静态存储期限。

函数体是一个程序块。在函数体内部，当我们需要临时使用一些变量时，程序块的概念也是非常有用的。

在程序块中声明临时变量的好处是：避免函数体起始位置的声明与只是临时使用的变量相混淆，减少名字冲突。

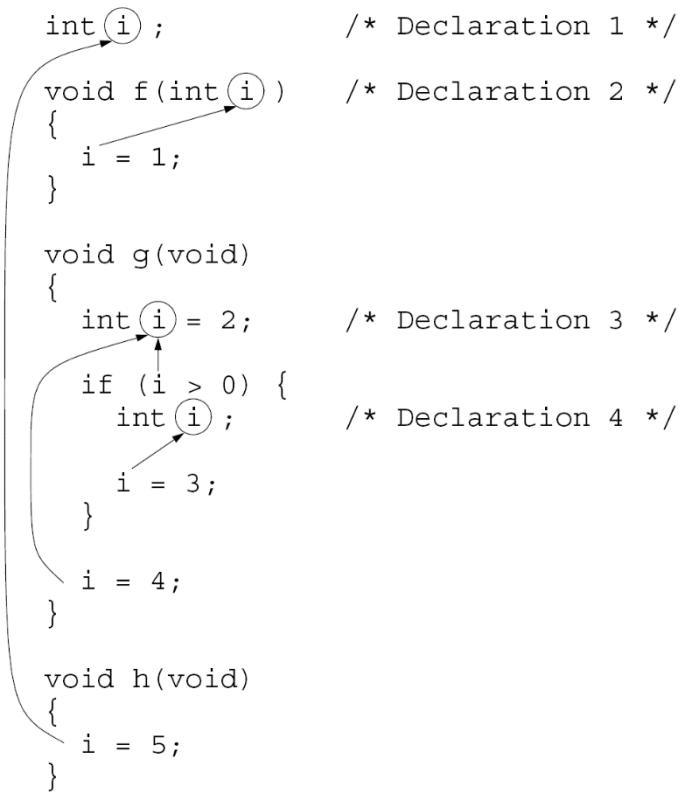
**4. 作用域**

在C程序中，相同的标识符可以有不同的含义。

C语言的作用域规则使得程序员（和编译器）能够确定与程序中给定点相关的是哪种含义。

最重要的作用域规则：当程序块内的声明命名一个标识符时，如果此标识符已经是可见的，新的声明会临时“覆盖”旧的声明，从而使该标识符获得新的含义。在程序块结束的位置，该标识符重新恢复到以前的含义。

在下面示例中，标识符 i有如下四种不同的含义：




在声明 1中：i 是具有静态存储期限和文件作用域的变量；

在声明2中：i 是具有块作用域的形式参数；

在声明3中：i 是具有块作用域的自动变量；

在声明4中：i 是具有块作用域的自动变量。

C语言的作用域规则使我们能够确定每种情况下标识符i 的具体含义（如箭头所示）。

**5. 构建 C 程序**

C程序的构成要素有：

- 预处理指令，如： #include 或 #define
- 类型定义
- 外部变量声明
- 函数原型
- 函数定义

C 语言对于上述要素的出现顺序没有过多限制。预处理指令仅在程序中出现的位置起作用，类型名只有在定义后才允许使用，变量只有在声明后才可以使用。

在调用函数之前对其进行声明或定义是一种很好的习惯。

遵循上述规则，有几种构建程序的方法。其中一种常用的顺序是：

- #include 指令
- #define 指令
- 定义类型
- 声明外部变量
- 声明除 main之外的函数原型
- 定义 main函数
- 定义其它函数

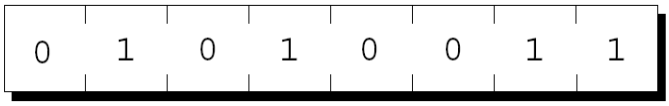
在函数定义之前增加一个注释框是一种很好的编程习惯。在注释中应当包含如下的信息：

- 函数名
- 定义该函数的目的
- 每个参数的含义
- 如果有返回值，应当给出描述信息
- 列出函数的副作用（如修改了外部变量的值）

## 4.2 指针

### 1. 指针变量

多数现代计算机将内存分割为字节（***bytes***），每个字节可以存储8位（bits）的信息：




每个字节都有唯一的地址（***address*** ）。程序中的每个变量占有一个或多个字节内存。将第一个字节的地址称为变量的地址。如果内存中有 $n$ 个字节，我们可以将内存地址的取值范围想像成0到$n-1$之间的整数。

可以用一种特殊的指针变量来存储内存地址。当采用指针变量p来存储变量i的地址时，我们通常说“p指向i”。

### 2. 声明指针变量

当声明一个指针变量时，变量名前必须加星号：

```
int *p;
```

声明的指针变量p可以指向一个int型**对象**

注意：我们采用术语“对象”而不是“变量”，是因为p 可以指向不属于变量的

内存区域。

指针变量可以和其它变量一起出现在声明中：

```
int i, j, a[10], b[20], *p, *q;
```

C语言要求每个指针变量只能指向一种特定类型（即：**引用类型**）的对象：

```
int *p;         /* points only to integers     */
double *q;    /* points only to doubles       */
char *r;        /* points only to characters */
```

至于具体是何种引用类型则没有限制。

**3. 取地址运算符和间接寻址运算符**

C语言为指针变量提供了一对特殊的运算符

（1）取地址运算符&

声明指针变量只是为指针留出空间，但并未将其指向任何对象。在使用p之前对其进行初始化是至关重要的。一种初始化指针变量的方法是采用某个变量的地址对其进行赋值：

```
int i, *p;
p = &i;
```

将变量i的地址赋给变量p，结果是令p指向i。

在声明指针变量的同时也可以对其初始化：

```
int i;
int *p = &i;
```

甚至可以进一步将整型变量 i 的声明和指针变量 p的声明合并到一条语句中：

```
int i, *p = &i;
```

（2）间接寻址运算符*

一旦指针变量指向了某个对象，就可以使用*（间接寻址运算符）访问存储在对象中的内容。

如果p指向i，可以采用如下方式输出i的值：

```
printf("%d\n", *p);
```

对变量使用&运算符产生指向变量的指针，而对指针使用 *运算符则可以返回到原始变量：

```
j = *&i;     /* same as j = i; */
```

只要p指向i，*p就是i的别名（***alias***）。*p 拥有和i相同的值，改变 *p的值，同时也会改变i的值。

将间接寻址运算符应用于未初始化的指针变量，会导致未定义的行为：

```
int *p;
printf("%d", *p);     /*** WRONG ***/
```

对*p进行赋值则更加危险：

```
int *p;
*p = 1;     /*** WRONG ***/
```

**4. 指针赋值**

C语言允许使用赋值运算符对指向同种类型变量的指针进行复制。

假设声明如下变量：

```
int i, j, *p, *q;
```

对指针进行赋值的示例如下：

```
p = &i;
```

另一个指针赋值的例子为：

```
q = p;
```

赋值的结果是：指针q与p指向相同的位置：

如果 p 和 q 都指向变量 i，可以使用对*p或*q赋值的方法来改变i的值。

任意数量的指针变量都可以指向同一个对象。

**5. 指针作为参数**

为修改传入函数的实际参数，可通过向函数传递指针（而不是变量值）来实现对实际参数的修改。

（1）例如，将一个实数的整数和小数部分分开的Decompose函数的定义：

```
void decompose(double x, long *int_part,
double *frac_part)
{
*int_part = (long) x;
*frac_part = x - *int_part;
}
```

Decompose函数可能的原型：

```
void decompose(double x, long *int_part,
double *frac_part);
void decompose(double, long *, double *);
```

调用decompose函数的方法：

decompose(3.14159, &i, &d);

函数调用的结果是：int_part 指向 i ，而 frac_part 指向 d：

Decompose函数体内的第一条赋值语句将x 的值转化为长整型（long ），并将其存储在指针变量int_part指向的对象中：

第二条赋值语句将(x - *int_part)存储到 指针变量frac_part 所指向的对象：

（2）Scanf函数接受的实参必须是指针：

int i;

…

scanf("%d", &i);

注意：如果去掉 &运算符，scanf函数实际接收到的参数是变量i的值。

尽管scanf的参数必须是指针，但在使用中却不一定非要使用取地址运算符&:

int i, *p;

…

p = &i;

scanf("%d", p);

对于以指针为参数的函数，如果参数传递时不小心传递了错误的值，可能导致灾难性的后果。

例如在调用 decompose 时，省略变量前的&运算符：

decompose(3.14159, i, d);

当decompose向*int_part 和 *frac_part所表示的位置存入数值时，将会试图修改未知的内存位置（的内容），而不是如预期地修改i和d的值。

如果在此之前声明了 decompose的函数原型，编译器就能够检测出上述错误。

然而在 scanf的例子中，编译器通常无法检测出这种指针传递上的错误。

**6. 用 const 保护参数**

当传递指向变量x的指针给一个函数f时，通常意味着x的值将被修改：

f(&x);

然而有的时候，f可能仅仅需要对x的值进行检查，而不需要对其进行修改。

采用指针参数更高效的原因：如果变量占用大量内存空间，那么传递变量的值会造成时间和空间上的浪费。

可以使用const关键字确保函数不会修改指针参数所指向的对象。

Const应放置在形式参数的声明中，位于参数的类型说明之前：

void f(const int *p)

```
{
*p = 0;     /*** WRONG ***/
}
```

如果函数中的语句试图修改 *p 的值，编译器能够捕获此类错误。

**7．指针作为返回值**

C语言允许函数返回指针。

```
int *max(int *a, int *b)
{
if (*a > *b)
return a;
else
return b;
}
```

max函数的调用方式：

```
int *p, i, j;
…
p = max(&i, &j);
```

函数调用结束后，p 指向 i 或 j其中之一。

尽管上例中 max函数返回的指针是作为实际参数传入的两个指针中的一个，但这不是唯一的选择。

函数也可以返回指向外部变量或指向声明为static的局部变量的指针。

注意：不要返回指向自动局部变量的指针。

```
int *f(void)
{
int i;
…
return &i;
}
```

一旦函数f返回，变量 i 将不复存在。

指针可以指向数组元素，设a为数组，则&a[i]为指向数组a中元素i的指针。

**4.3 指针和数组**

**1. 指针的算术运算**

C语言允许对指向数组元素的指针进行算术运算：加法和减法。

这一特性使我们能够用指针代替数组下标对数组进行处理。C语言中指针和数组的关系非常紧密。

（1）例：

int a[10], *p;

p = &a[0];

现在我们可以通过p访问a[0]，例如，可以采用如下的方式将数值5存入a[0] 中：

*p =5;

如果指针p指向数组a的元素，则可以通过对指针p进行指针算术运算（或地址算术运算）访问数组a中其他元素。

（2）C语言支持三种类型的指针算术运算（且仅有这三种类型）：

指针加上整数

指针减去整数

两个指针相减

指针加上整数

指针p加上整数j产生一个新的指针，指向p 当前所指的元素位置之后j个元素的位置。

更准确地说：如果 p指向数组元素 a[i]，则 p + j 指向数组元素 a[i+j]。

假设有如下声明：

int a[10], *p, *q, i;

- 指针加上整数

指针加法运算示例：

p = &a[2];

q = p + 3;

p += 6;

- 指针减去整数

如果p 指向 a[i]，则p - j 指向 a[i-j]。

示例：

p = &a[8];

q = p - 3;

p -= 6;

- 两个指针相减

当两个指针相减时，结果为指针之间的距离（以数组元素的个数作为度量）。

如果p指向a[i] ，q指向a[j]，则 p – q等于i – j。

示例：

```
p = &a[5];
q = &a[1];
i = p - q;     /* i is 4 */
i = q - p;     /* i is -4 */
```

**2. 指针比较**

指针可以采用关系运算符 (<, <=, >, >=) 和判等运算符 (== and !=)进行比较：

只有在两个指针指向同一数组（中的元素）时，用关系运算符进行指针比较才有意义。

指针比较的结果依赖于指针指向的数组元素在数组中的相对位置。

例如：通过如下的赋值操作

```
p = &a[5];
q = &a[1];
```

有 p <= q 的值为 0，而 p >= q 的值为 1。

**3. 指针用于数组处理**

指针的算术运算使我们能够通过对指针变量进行重复自增来逐一访问数组大元素。

下例采用循环对数组a中的元素进行求和：

```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
sum += *p;
```

for 语句中的循环条件 p < &a[N] 值得引起特别的注意。

尽管元素a[N]并不存在，但对其进行取地址运算是合法的。

采用指针算术运算通常可以节省执行时间。

然而，对某些编译器而言，采用数组下标来遍历数组的效率可能更高。

**4. * 和 ++ 运算符的组合**

（1）C程序员常在处理数组元素的语句中组合使用 *（间接访问）和 ++ 操作符。

下述语句修改当前位置（下标i）的数组元素值，并修改下标前进到下一个元素的位置：

a[i++] = j;

相应地，可以采用指针实现同样功能：

*p++ = j;

由于后缀的 ++ 运算符的优先级高于*，编译器将上述语句解释为：

*(p++) = j;

（2）可能的 * 和 ++运算符的组合：

*p++ or *(p++)  自增前表达式的值为 *p，然后自增p;

(*p)++  自增前表达式的值为 *p，然后自增*p;

*++p or *(++p)  先自增 p，自增后表达式的值为 *p;

++*p or ++(*p)  先自增 *p，自增后表达式的值为 *p;

最常见的组合是 *p++，在循环中十分方便。

例如：下述代码对数组a的元素进行求和

```
for (p = &a[0]; p < &a[N]; p++)
sum += *p;
```

可以改写为：

```
p = &a[0];
while (p < &a[N])
sum += *p++;
```

（3）* 和 -- 运算符的组合类似于* 和 ++的组合。

**5. 用数组名作为指针**

指针的算术运算体现了数组与指针间相互关联的关系。

另一种重要的关系是：

可以用数组名作为指向数组第一个元素的指针。

这种关系简化了指针的算术运算，并且使得数组和指针更加通用。

假设用如下方式声明数组a ：

int a[10];

使用数组名 a 作为指针的示例如下：

```
*a = 7;     /* stores 7 in a[0] */
*(a+1) = 12;     /* stores 12 in a[1] */
```

通常情况下，a + i 和 &a[i]是等同的,均表示指向数组 a 中元素 i 的指针。

同样地， *(a+i) 与 a[i]也是等同的，均表示数组a中的元素i自身

数组名可以用作指针这一事实，使得编写遍历数组的循环更加容易。

例如对于如下的循环：

```
for (p = &a[0]; p < &a[N]; p++)
sum += *p;
```

可以简化为如下形式：

```
for (p = a; p < a + N; p++)
sum += *p;
```

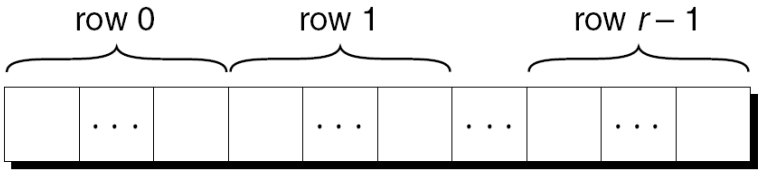
**6. 指针和多维数组**

正如指针可以指向一维数组的元素一样，指针也可以指向多维数组的元素。

（1）处理多维数组的元素

C语言按行主序存储二维数组。

一个 *r* 行的数组可以表示如下：




如果指针 p 指向二维数组中的第一个元素（0行0列），则可以通过反复自增 p 的方式访问数组中的每一个元素。

示例：将如下二维数组中的所有元素初始化为0：

```
int a[NUM_ROWS][NUM_COLS];
```

显然我们可以采用嵌套的 for 循环：

```
int row, col;
…
for (row = 0; row < NUM_ROWS; row++)
for (col = 0; col < NUM_COLS; col++)
a[row][col] = 0;
```

但是，如果将 a 视为一维的整型数组，一重循环就够了：

```
int *p;
…
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
*p = 0;
```

尽管将二维数组作为一维数组处理有点“取巧”的嫌疑，但绝大多数编译器都

接受这样的做法。然而这种做法显然破坏了程序的可读性，不过对于某些老旧的编译器而言，这种做法能够在程序执行效率方面得到一些补偿。

（2）处理多维数组的行

指针变量 p 也可以用于处理二维数组中的某一行元素。

为了访问二维数组a中的第i行元素，可以采用如下方式初始化指针p ，使之指向数组a中第i行的第0个元素：

```
p = &a[i][0];
```

或者，可以简单地写为：

```
p = a[i];
```

对于任意二维数组a，表达式a[i]的结果是一个指针，指向数组a中第i行的首元素。

为理解该表达式，让我们回忆一下a[i]和 *(a + i)的等同关系

由此可知，&a[i][0] 等同于 &(*(a[i] + 0))，因此也等同于 &*a[i]。

显然，&*a[i]就是 a[i]，因为&和 *运算符的作用相互抵消。

下述循环语句将数组a中第i行元素清零：

```
int a[NUM_ROWS][NUM_COLS], *p, i;
…
for (p = a[i]; p < a[i] + NUM_COLS; p++)
*p = 0;
```

由于a[i] 是指向数组a中第i行元素的指针，因此可以将其作为参数传递给一个接受一维数组作为形参的函数。

换言之，以一维数组为形参的函数，同样可以接受二维数组中的一行作为参数传入。

## 4.4 字符串（Strings）

### 1. 字符串字面量（String Literals）

（1）字符串字面量(string literal)是用一对双引号括起来的字符序列

```
"When you come to a fork in the road, take it."
```

字符串字面量可以像字符常量一样包含转义序列。转义字符常出现在 printf函数和 scanf函数的格式串中。

（2）延续字符串字面量

‘\’字符可以用于延续一个字符串从一行到下一行，如：

```
printf("When you come to a fork in the road, take it.  \
```

--Yogi Berra");

总的来说，可用'\'字符连接两行或者多行成为一行。但C语言提供了处理长字符串字面量更好的方法。

当两个或则多个字符串字面量相邻时，编译器会将它们连接成一个字符串。这一规则允许我们把一个字符串字面量拆分到多行，如:

printf("When you come to a fork in the road, take it.  "

"--Yogi Berra");

（3）字符串字面量的存储

当编译器遇到一个长度为n的字符串字面量时，给该字符串分配n+1个字节的内存空间。

该内存空间将存放字符串中的字符，外加一个额外的空字符，用于标志字符串的结束。

空字符是一个所有比特全为0的字节，用转义序列 \0表示。

字符串字面量 “abc” 是以四个字符的数组来存放的，字符串“”则以单个空字符的数组来存储的。

由于字符串字面量是以数组的方式存储的，编译器把把字符串字面量以char*来处理。

printf和scanf 接收 char *的值作为它们的第一个参数。下面的调用传递“abc“的地址给printf函数

printf("abc");

（3）字符串字面量的操作

可以在任何 C语言允许使用 char*指针的地方使用字符串字面量。

char *p;

p = "abc";

这个赋值操作不是复制“abc”中的字符，而仅仅是使 p指向字符串的第一个字符。C语言允许对指针添加下标，因此可以给字符串字面量添加下标：

char ch;

ch = "abc"[1];

ch的新值则为字符b。

对字符串字面量的操作会导致未定义的行为:

char *p = "abc";

*p = 'd';  /*** WRONG ***/

试图修改字符串字面量的程序可能会导致程序崩溃或者不确定的行为。

（4）字符串字面量 vs 字符常量

包含单个字符的字符串字面量与一个字符常量是不一样的：

“a” 是以指针表示的；

‘a’ 是以整数表示的。

**2. 字符串变量**

（1）任何一维字符数组均可以用于存储字符串，字符串必须以空字符结尾。

如果字符串变量需要存放80个字符，对应的字符数组必须声明为80+1个:

```
#define STR_LEN 80
…
char str[STR_LEN+1];
```

额外增加的1用于给字符串结束符留出空间

定义一个宏来表示80，然后采用+1的方式来定义数组是一个常见的实践

当声明字符串变量的时候，确保给空字符留出空间，否则程序运行时可能造成不可预料的结果。字符串的实际长度取决于字符串结束符的位置。

长度为 STR_LEN + 1 的字符数组可以存放长度为0到STR_LEN的字符串

（2）初始化字符串变量

声明一个字符串的同时可以初始化该字符串:

```
char date1[8] = "June 14";
```

编译器会自动增加一个空字符，这样即可以把字符数组 date1 作为字符串：

在初始化这种情况下，“June 14” 不是一个字符串字面量

C把这种形式作为数组初始化器的缩写。

如果初始化器太短而不能填满整个字符串变量，编译器会在后面增加额外的空字符。

字符串变量的初始化器不能超过该字符串的长度，但是可以一样长:

```
char date3[7] = "June 14";
```

由于没有存放空字符的空间，编译器就不再存放一个空字符。

声明字符串变量时也可以忽略其长度，这种情况下，由编译器来计算字符数组的长度:

```
char date4[] = "June 14";
```

编译器为date4留出8个字符的空间，用于存放“June 14” 和额外的一个空字符.

对应初始化器比较长的情况，忽略字符串变量的长度是比较有用的，避免了手工计算容易造成的错误.

**3. 字符数组 vs 字符指针**

声明：

char date[] = "June 14";

声明 date为一个数组,

而声明：

char *date = "June 14";

声明 date为一个指针 .

由于数组和指针之间的紧密关系，上述两个版本均可作为一个字符串.

然而，上述两个版本的date存在重要的区别。

a. 数组版，存储与date的字符是可以修改的；指针版，date所指向的字符串字面量是不可以修改的.

b. 数组版，date是一个数组名；指针版，date是一个指向其它字符串的变量。

声明：

char *p;

没有为字符串分配空间.

在把p作为字符串之前，p必须指向一个字符数组.

做法1，使p指向字符串变量:

char str[STR_LEN+1], *p;

p = str;

做法2，使 p 指向动态分配的字符串.

使用未初始化的指针变量作为字符串是一个严重的错误。

**4. 读写字符串**

写字符串可以采用 printf 或者 puts.

读字符串稍微困难，因为输入字符串的长度可能大于存放字符串变量的长度

可以用scanf或者gets一步读入单个字符串.

另一种方法，一次读入一个字符.

（1）用printf和puts写字符串

Printf函数可以用 %s 转换说明符来写一个字符串:

char str[] = "Are we having fun yet?";

printf("%s\n", str);

输出为：

Are we having fun yet?

printf 逐字符写字符串，直至遇到空字符.

使用转换说明符%.*p*s 来输出字符串的一部分.

*p* 表示要显示的字符的个数.

语句：

printf("%.6s\n", str);

将显示

Are we

%*m*s 转换说明符将显示字符串在m个字符宽度的域.

如果字符串少于m个字符，字符串将在域内右对齐.

在m的前面放置一个-号，可以强制字符串左对齐.

*m* 和 *p* 可以组合使用.

转换说明符%*m*.*p*s 在宽度为m的域中显示字符串的前p个字符.

printf 不是唯一可用于写字符串的函数。C库也提供 puts函数用于写字符串:

puts(str);

写完字符串后，puts函数会写一个额外的新行符.

（2）用scanf 和 gets读字符串

Scanf函数用%s 转换说明符读字符到一个字符数组:

scanf("%s", str);

str 在这里是一个指针，因此不必在str前面放置 & 运算符.

当调用 scanf时，该函数跳过空白，然后读入字符并存入str指向的空间，直至遇到一个空白字符.

scanf 函数会存放一个空字符在字符串的后面。scanf并不总能读一整行输入，新行符、空白和tab符，均导致 scanf 停止读取。

要读取整行输入，可以使用 gets函数。

gets函数的特点:

读取输入不会跳过开始的空白.

直到找到新行符才停止读入.

不存储新行符，而用空字符代替.

考虑下述程序片段:

char sentence[SENT_LEN+1];

printf("Enter a sentence:\n");

scanf("%s", sentence);

假设在提示后

Enter a sentence:

输入一行

To C, or not to C: that is the question.

scanf 将存储“To”到 sentence所指的空间.

如果用gets代替scanf：:

gets(sentence);

当用户输入相同的字符串时, gets将存放字符串

"  To C, or not to C: that is the question."

到 sentence所指向的空间.

**5. 访问字符串中的字符**

访问字符串中字符，采用数组操作还是指针操作好？采用任何一种都是合适的。传统上，C程序员喜欢使用指针操作。

**6. 使用 C 字符串库**

C语言中，字符串是作为数组处理的，因此受到数组操作本身的限制。

特别地，数组不能用运算符进行拷贝和比较。但C函数库提供了丰富的函数用于完成数组的操作。

需要进行字符串操作的程序应包含下面的一行:

#include <string.h>

在后面的例子中，假定 str1和str2是用作字符串的字符数组。

（1）strcpy （string copy）函数

strcpy 函数原型:

char *strcpy(char *s1, const char *s2);

strcpy 拷贝字符串 s2 到字符串s1.

准确地说，我们应该说“strcpy拷贝s2指向的字符串到s1指向的字符数组”。strcpy 返回s1 (指向目的字符串的指针)。

调用 strcpy将字符串 “abcd” 存储到 str2指向的字符数组:

strcpy(str2, "abcd");

/* str2 now contains "abcd" */

拷贝 str2的内容到 str1:

strcpy(str1, str2);

/* str1 now contains "abcd" */

在调用 strcpy(str1, str2)时, strcpy不检查str2字符串是否能够容纳str1指向的数组。

如果没法装入，则发生未定义的行为。

（2）调用strncpy 则是一个较慢但是更安全数组拷贝方式.

strncpy 需要第三个参数来限制拷贝的字符的个数.

调用 strncpy 来拷贝 str2 到str1:

strncpy(str1, str2, sizeof(str1));

如果str2的长度大于或者等于str1数组的长度，strncpy将保持拷贝的结果而不能给str1增加一个字符串结束符。

使用strncpy更安全的方式:

strncpy(str1, str2, sizeof(str1) - 1);

str1[sizeof(str1)-1] = '\0';

第二条预计保证了str1总是以空字符结尾的。

（3）strlen (String Length)函数

Strlen函数原型:

size_t strlen(const char *s);

size_t 是一个 typedef 名，表示 unsigned integer

strlen 返回字符串s的长度，不包括空字符.

例:

int len;

len = strlen("abc");  /* len is now 3 */

len = strlen("");  /* len is now 0 */

strcpy(str1, "abc");

len = strlen(str1);  /* len is now 3 */

（4）strcat (String Concatenation) 函数

strcat 函数原型:

char *strcat(char *s1, const char *s2);

strcat 追加字符串s2的内容到字符串s1的末尾.

返回s1 (指向结果字符串的指针).

strcat 例:

strcpy(str1, "abc");

strcat(str1, "def");

/* str1 now contains "abcdef" */

strcpy(str1, "abc");

strcpy(str2, "def");

strcat(str1, str2);

/* str1 now contains "abcdef" */

strcpy类似, strcat的返回值通常是舍弃了.

下例示范怎样使用strcat的返回值:

strcpy(str1, "abc");

strcpy(str2, "def");

strcat(str1, strcat(str2, "ghi"));

/* str1 now contains "abcdefghi";

str2 contains "defghi" */

如果str1数组没有足够的空间容纳str2的内容, 则strcat(str1, str2)会导致未定义的行为.

例如:

char str1[6] = "abc";

strcat(str1, "def");  /*** WRONG ***/

str1 限制为6个字符的数组, 导致strcat的写操作越过了数组的末尾.

（5）Strncat是strcat的安全但是较慢的版本.

与strncpy类似, 需要第三个参数来限制要拷贝的字符数.

调用strncat:

strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);

strncat 会以一个空字符结束str1 .

（6）strcmp (String Comparison) 函数

strcmp 函数原型:

int strcmp(const char *s1, const char *s2);

strcmp 比较字符串 s1 和 s2, 根据s1是小于、等于或者大于s2来返回一个小于、等于或者大于0的值.

测试str1是否小于 str2:

if (strcmp(str1, str2) < 0)  /* is str1 < str2? */

测试 str1 是否小于或者等于 str2:

if (strcmp(str1, str2) <= 0) /* is str1 <= str2? */

通过选择正确的运算符 (<, <=, >, >=, ==, !=), 我们能够测试s1和s2之间的任何可能关系.

如果下述任一条件满足，则strcmp 认为s1小于s2:

s1 和s2的前i个字符匹配，但是s1的第 (*i*+1)个字符小于s2的 第(*i*+1)个字符.

s1的所有字符都匹配 s2, 但是s1 比s2短.

就比较两个字符串而言，strcmp 查看的是字符串中字符的数值编码。

**7. 字符串数组**

存储字符串数组有多种方法.

一种方法是采用二维字符数组，每行一个字符串:

```
char planets[][8] = {"Mercury", "Venus", "Earth",
"Mars", "Jupiter", "Saturn",
"Uranus", "Neptune", "Pluto"};
```

可以忽略数组的行数，但是必须指明数组的列数.

然而，planets数组包含了一定数量的未用空白（额外的空字符):

大多数字符串集合都会有一些长的和短的字符串。

我们需要的是一种参差不齐的数组(ragged array)，可以有不同长度的行，以便节省空间.

在C中，我们可以采用指针数组的方式来满足这种需求:

```
char *planets[] = {"Mercury", "Venus", "Earth",
"Mars", "Jupiter", "Saturn",
"Uranus", "Neptune", "Pluto"};
```

这种小的改动对planets的存储具有很大的影响:

要访问一个行星的名字，我们所需要的仅是下标planets数组.

访问行星名中的一个字符与访问二维数组中的元素一致.

搜索planets数组中以字母M开头的字符串的循环为:

```
for (i = 0; i < 9; i++)
if (planets[i][0] == 'M')
printf("%s begins with M\n", planets[i]);
```

**8. 命令行参数**

当我们运行一个程序，我们常常需要给该程序提供一些信息，包括一个文件名或者一个开关用于修改程序的行为.

UNIX中 ls 命令的使用举例:

```
ls
ls –l
```

ls -l remind.c

命令行信息并不是仅限于操作系统命令,对所有程序都是可用。

要访问命令行参数， main函数必须有两个参数:

```
int main(int argc, char *argv[])
{
…
}
```

在C标准中，命令行参数称为程序参数。

argc (“argument count”) 是命令行参数的数量.

argv (“argument vector”) 是指向命令行参数的指针数组，命令行参数以字符串方式存储.

argv[0] 指向程序名，而argv[1] 至 argv[argc-1] 指向余下的命令行参数.

argv[argc] 总是空指针，不指向任何东西.

宏NULL 表示空指针.

如果用户输入的命令行为：

ls -l remind.c

则argc 为3, argv 为如下表示：

由于 argv 是一个指针数组，访问命令行参数是很容易的.

典型地，要访问命令行参数的程序会采用一个循环来顺序检查每个命令行参数.

一种方法是用一个整型变量来作为argv数组的索引:

```
int i;
for (i = 1; i < argc; i++)
printf("%s\n", argv[i]);
```

另一个方法是用一个指针指向argv[1], 然后重复增量该指针:

```
char **p;
for (p = &argv[1]; *p != NULL; p++)
printf("%s\n", *p);
```

## 三、实验内容

项目 1：第 10 章编程题 1

项目 2：第 11 章编程题 4

项目 3：第 12 章编程题 1a

项目 4：第 12 章编程题 1b

项目 5：第 12 章编程题 3

项目 6：第 13 章编程题 1

## 四、实验步骤

1、明确项目需求

2、编写代码

3、编译代码

4、测试程序

5、根据测试结果对程序进行调试改进

# 实验五 超市商品信息管理系统

## 一、实验目的

1) 掌握结构体的定义和使用方法
2) 掌握指针与指针数组的使用方法
3) 掌握字符串的定义和使用方法

## 二、实验原理

1) 结构体：结构是可能具有不同类型的值（成员）的集合，结构的元素（在C语言中的说法是结构的成员）可能具有不同的类型。而且每个结构成员都有名字，所以为了选择特定的结构成员需要指明结构成员的名字而不是它的位置。
2) 结构变量的声明：当需要存储相关数据项的集合时，结构是一种呵护逻辑的选择。例如假设需要记录存储在仓库中的零件。用才存储每种零件的信息可能包括零件的编号（证书）、零件的名称（字符串）以及现在有零件的数量。为了产生一个可以存储全部三种数据项的变量，可以使用类似下面这样的声明：

```
struct{
        int number;
        char name[NAME_LEN+1];
        int on_hand;
}part1,part2;
```

每个结构变量都有三个成员：number（零件的编号）、name（零件的名称）和 on_hand（现有数量）。注意这里的生命格式和 C 语言中其他变量的声明格式一样。struct{…}指明了类型。而part1和part2则是具有这种类型的变量。结构的成员在内存中是按照声明的顺序存储的。

3) 每个结构代表一种新的作用域。任何声明中刺作用域内的名字都不会和程序中的其他名字冲突。（用 C 语言的术语可以表述为：每个结构都为它的成员

设置了独立的名字空间（name space）。）例如，下来声明可以出现在同一程序中：

```
struct{
        int number;
        char name[NAME_LEN+1];
        int on_hand;
}part1,part2;
struct{
        int number;
        char name[NAME_LEN+1];
        char sex;
}employee1, employee2;
```

结构 part1 和 part2 中的成员 number 和成员 name 不会与结构 employee1 和 employee2 中的成员 number 和成员 name 冲突。

4) 结构变量的初始化：和数组一样，结构变量也可以在声明的通知进行初始化。为了对结构进行初始化，要把待存储到结构中的值的列表准备好并用花括号把它括起来：

```
struct{
        int number;
        char name[NAME_LEN+1];
        int on_hand;
}part1 = {528,”Disk drive”,10},
 part2 = {914,”Printer cable”,5};
```

初始化式中的值必须按照结构成员的顺序进行显示。结构初始化式遵循的原则类似于数组初始化式的原则。用于结构初始化式的表达式必须是常量。例如，不能用变量来初始结构 part1 的成员 on_hand。

5) 对结构的操作：既然最常见的数组操作是取下标，那么结构最常用的操作是选择成员也就无需惊讶了。但是结构成员是通过名字而不是通过位置访问的。为了访问结构内的成员，首先写出结构的名字，然后写一个据点，再写出成

员的名字。例如，下列语句显示结构 part1 的成员的值：

printf(“Part number:%d\n”, part1.number);

printf(“Part name:%s\n”,part1.name);

printf(“Quantity on hand:%d\n”,part1.on_hand);

结构的成员是左值，所以它们可以出现在复制运算的左侧，也可以作为自增或自减表达式的操作数：

Part1.number = 258;

Part1.on_hand++;

用于访问结构成员的句点实际上就是一个 C 语言的运算符，句点运算符的优先级几乎高于所有其他运算符。

结构的另一种主要操作是赋值运算：

part2 = part1；

这一句的效果是把 part1.number 复制到 part2.number,把 part1.name 复制到 part2.name,依次类推。

6) 结构标记：结构标记是用于标识某种特定结构的名字。下面的例子声明了名为 part 的结构标记：

```
struct part{
        int number;
        char name[NAME_LEN+1];
        int on_hand;
};
```

一旦创建了标记 part，就可以用它来声明变量了：

struct part part1，part2；

但是，不能通过漏掉单词 struct 来缩写这个声明：

part part1，part2；

part 不是类型名。如果没有单词 struct 的话，它没有任何意义。

7) 结构类型的定义：除了声明结构标记，还可以用 typedef 来定义真是的类型名。例如，可以按照如下方式定义名为 Part 的类型：

```
typedef struct {
        int number;
```

```
        char name[NAME_LEN+1];
        int on_hand;
}Part;
```

注意,类型 Part 的名字必须出现在定义的末尾,而不是在单词 struct 的后边。可以像内置类型那样使用 Part。例如，可以用它声明变量：

```
Part part1，part2;
```

8) 可以通过一个 while 条件为 1 的循环来完成程序功能的循环调用，直至输入退出系统的命令。例如：

```
while(1)
{
        switch(choice){
            case 1:Fucntion1;
                    break;
                …
    }
}
```

9) 结构指针数组：通过一个数组来存储所有的商品信息，当新插入一个商品信息的时候先分配内存，然后把指向该商品信息结构的指针赋值到结构指针数组中：

```
GoodInfo* pGoodInfo = (GoodInfo*)malloc(sizeof(GoodInfo));
strcpy(pGoodInfo->good_id,temp.good_id);
strcpy(pGoodInfo->good_name,temp.good_name);
strcpy(pGoodInfo->good_price, temp.good_price);
strcpy(pGoodInfo->good_discount, temp.good_discount);
pGoodInfo->good_amount = temp.good_amount;
pGoodInfo->good_remain = temp.good_remain;
Goods[i] = pGoodInfo;
```

## 三、实验内容

用C语言实现一个小型的超市商品管理系统，该系统需要具备商品信息录入、商品信息修改、商品信息删除、商品信息查找、商品信息的插入这几个功能。具体实现步骤如下：

1) 软件界面控制：实现一个数字选项式的启动界面，其中包含、商品信息修改、商品信息删除、商品信息查找、退出系统并保存5个选项。并且这些功能可以循环调用。
2) 商品信息的初始化：定义一个商品信息的结构体，并且声明一个指针数组，数组中的内容为指向结构体的指针。实现一个函数，从已有的商品信息文件中读入商品信息，并且分配内存保存至指针数组中。
3) 商品信息的修改：实现一个函数完成商品信息的修改功能，实现可以根据商品的名称修改商品信息。其中用字符串比较的方式来查找待修改商品。
4) 商品信息的删除：实现一个函数，实现根据商品的名称来删除对应的商品信息的功能，商品查找通过字符串比较的方式，查找到后释放对应指针指向的内存区域，完成删除。
5) 商品信息的查找：实现一个函数，函数的功能是根据输入的商品名称来查找对应的商品信息，商品名称的判断用字符串比较的方式来实现，然后调用格式化输出子函数显示查找到的商品信息。
6) 退出系统，并保存：实现一个文件写入函数将所有信息的改动写入到商品信息文件，然后清理系统运行过程中已分配的内存。

## 四、实验步骤

1．新建一个.c程序来编写实验内容中的第一小题

2．编译、调试程序直至达到第一小题的实验要求

   1) 定义用于表示某种商品的所有信息的结构体，并且定义结构体指针数组用来组织所有的商品信息库中的商品信息
   2) 定义并实现一个函数：void info_init()，完成从一个txt文件读入商品库初始化信息来初始化商品库，在读入的时候每读到一条商品信息就实时

的动态分配内存来把信息放到分配得到的结构体指针指向的内存单元中，然后把指针加入到上一步定义的结构体指针数组中

3) 定义并实现函数：void info_flush()，完成将系统运行期间改动过的商品信息库写回到存放商品信息的 txt 文件中
4) 定义并实现函数：void OutputAll()，完成将通讯录中所有同学的每项信息打印到标准输出（即屏幕上）
5) 定义并实现函数：void info_output(int i)，格式化输出某一项商品信息，其中 i 是在结构体指针数组中的对应下标
6) 定义并实现函数：void info_change()，完成商品信息的修改功能，其中要求用户输入需要修改的某项商品的名称，然后对名称进行查找，找到则继续输入该商品的各项信息，并提示修改成功；如果没有找到则提示对应商品未找到在两种选择后都返回到初始的选择菜单
7) 定义并实现函数：void info_dele()，完成删除某条商品库中信息的功能，通过输入的某项商品的名称删除对应的信息，如果在商品库中找到对应的商品便删除该商品信息（即释放指针所指向的内存，并把该指针赋值为 NULL），并提示删除成功；如果没有找到该商品要提示没有找到该商品信息，在两种选择下都回到上一步选择界面
8) 定义并实现函数：void info_search()，完成商品信息的查找功能，然后通过输入某种商品信息的名称来检索商品信息库，查找到则显示该商品的详细信息，没有查找到则提示没有该商品
9) 定义并实现函数：void info_insert()，完成商品信息的插入，在插入之前动态的分配内存用来存储插入的商品信息，然后把指向该内存的指针加入到第一步定义的结构体指针数组中的第一个空元素中；在插入之前必须考虑整个信息库的限定容量，如果超过上限要给用户以提示
10) 实现程序的入口函数即 main 函数，然后通过一个条件为 1 的 while 循环完成以上功能的循环调用，直至选择正常退出

3．编译、调试程序直至达到实验要求