

# C

## introduction

### for beginners

*first edition*

cyclone\_dll  
Evie

## 声明

本资料由作者整理、编写，仅供技术交流使用。

若用于教学或商业目的，需联系作者取得授权，否则视为侵权。

于 2024 年 6 月 23 日星期日 完成于昆明

# 目 录

1 基础介绍 .....	1
1.1 你的第一个 C 程序 .....	1
1.2 注释 .....	3
1.3 变量、常量和字面量 .....	7
1.4 C 的数据类型 .....	11
1.5 输入输出 (I/O) .....	16
1.6 运算符 .....	24
2 流程控制 .....	33
2.1 If..else 语句 .....	33
2.2 for 循环 .....	40
2.3 while and do..while 循环 .....	44
2.4 break 和 continue 语句 .....	49
2.5 switch 语句 .....	52
2.6 goto 语句 .....	55
3 函数 .....	58
3.1 函数 .....	58
3.2 用户自定义函数 .....	61
3.3 用户自定义函数的类型 .....	65
3.4 递归 .....	70
3.5 变量的存储方式 .....	73
4 数组 .....	77
4.1 数组 .....	77
4.2 多维数组 .....	83
4.3 在 C 中将数组传递给函数 .....	89
5 指针 .....	93
5.1 指针 .....	93
5.2 数组和指针之间的关系 .....	100
5.3 传递地址和指针 .....	104
5.4 动态内存分配 .....	106
5.5 数组和指针示例 .....	111
6 字符串 .....	130
6.1 字符串 .....	130
6.2 使用库函数进行字符串操作 .....	136
6.3 字符串示例 .....	138
7 结构体和联合 .....	148
7.1 结构体 .....	148
7.2 结构和指针 .....	155
7.3 结构与函数 .....	158
7.4 联合 .....	162
8 操作文件 .....	166
8.1 文件处理 .....	166

8.2 文件操作示例 .....	176
9 额外主题 .....	179
9.1 关键字和标识符 .....	179
9.2 运算符的优先级和关联性 .....	182
9.3 按位运算符 .....	186
9.4 预处理器和宏 .....	193
9.5 标准库函数 .....	199
9.6 枚举 .....	202
10 测试题 .....	206
10.1 判断奇偶 .....	206
10.2 求圆的面积 .....	208
10.3 利用循环画星号 .....	209
10.4 输入多个数值，求最大值 .....	210
10.5 反转字符串 .....	212
10.6 计算两个数字的和 .....	215
10.7 最大公约数 .....	216
10.8 判断素数 .....	217
10.9 生成斐波那契数列 .....	218
10.10 计算一个数的阶乘 .....	219
10.11 创建简单的计算器 .....	220

# 1 基础介绍

## 1.1 你的第一个 C 程序

现在，让我们学习如何编写一个简单的 C 程序。

我们将编写一个显示在屏幕上的简单程序 `Hello, World!` 。

```
#include <stdio.h>
int main() {

    printf("Hello, World!");

    return 0;
}
```

将会输出：

```
Hello World!
```

**注：** Hello World 程序用于介绍一种新的编程语言。这是每个初学者编写的第一个程序。

如果你不了解这个程序是如何工作的，那也没关系。我们将在即将到来的教程中了解所有关于它们的内容。现在，只需编写准确的程序并运行即可。

## C 程序的工作

祝贺你编写了第一个 C 程序。现在，让我们看看这个程序是如何工作的。

```
#include <stdio.h>
int main() {

    printf("Hello, World!");

    return 0;
}
```

请注意以下代码行：

```
printf("Hello, World!");
```



在这里，`printf` 语句打印文本 `Hello, World!` 到屏幕。  
请记住以下关于 `printf` 的重要事项：

- 要打印的所有内容都应放在括号 `()` 内。
- 要打印的文本用双引号 `"` 括起来。
- 每个 `printf` 语句都以分号 `;` 结尾。

不遵守上述规则将导致错误，并且代码将无法成功运行。

---

## C 程序的基本结构

正如我们从上一个示例中看到的，即使对于一个简单的程序，C 语言编程也需要很多行。

现在，请记住，我们将编写的每个 C 语言编程都将遵循以下结构：

```
#include <stdio.h>

int main() {

    // 你的代码

    return 0;
}
```

而且，我们通常在大括号 `{}` 内写出代码。

## 1.2 注释

在上一教程中，你学习了编写第一个 C 程序。现在，让我们了解一下 C 语言的注释。

**提示：**我们将在本教程系列的早期引入注释，因为从现在开始，我们将使用它们来解释我们的代码。

注释是我们添加到代码中的提示，使其更易于理解。

C 编译器完全忽略注释中的内容。

例如：

```
#include <stdio.h>
int main()
{
    // 将 Hello World 打印到屏幕上
    printf("Hello World");
    return 0;
}
```

输出：

```
Hello World
```

这里，`// 将 Hello World 打印到屏幕上`是 C 中的注释。编译器会忽略符号后面的所有内容。

**注：**你可以忽略编程概念，而只关注注释。我们将在后面的教程中重新讨论这些概念。

---

### C 语言中的单行注释

在 C 中，单行注释以符号 `//` 开头。它在同一行开始和结束。例如：

```
#include <stdio.h>
int main()
{
    // 创建整数型变量
    int age = 25;
}
```



```
// 打印 age 变量
printf("Age: %d", age);

return 0;
}
```

输出

```
Age: 25
```

在上面的示例中，我们使用了两个单行注释：

- `// 创建整数型变量`
- `// 打印 age 变量`

我们还可以在代码中使用单行注释。

```
int age = 25; // 创建整数型变量
```

在这里，编译器将执行 `//` 左边的代码，而 `//` 右边的代码将被忽略。

## C 语言中的多行注释

在 C 编程中，还有另一种类型的注释，它允许我们一次对多行进行注释，它们是多行注释。

要编写多行注释，我们使用 `/*....*/` 符号。例如：

```
/* 此程序接受用户的年龄输入
它将其存储在年龄变量中并且
，使用 printf（）打印值 */
#include <stdio.h>
int main()
{

    // 创建整数型变量
    int age = 25;

    // 打印 age 变量
    printf("Age: %d", age);

    return 0;
}
```

输出

```
Age: 25
```

在这种类型的注释中，C 编译器会忽略从 `/*` 到 `*/` 的所有内容。

注：记住使用注释的键盘快捷键：

- 单行注释：ctrl + / (windows) 和 cmd + / (mac)
- 多行注释：ctrl + shift + / (windows) 和 cmd + shift + / (mac)

## 使用注释阻止代码执行

在调试时，可能会出现我们不需要代码的某些部分的情况。例如：

在下面的程序中，假设我们不需要与身高相关的数据。因此，我们可以简单地将它们转换为注释，而不是删除与高度相关的代码。

```
#include <stdio.h>
int main()
{
    int number1 = 10;
    int number2 = 15;
    int sum = number1 + number2;

    printf("The sum is: %d\n", sum);
    printf("The product is: %d\n", product);
    return 0;
}
```

在这里，代码抛出一个错误，因为我们没有定义一个 `product` 变量。

我们可以注释掉导致错误的代码。

例如：

```
#include <stdio.h>
int main()
{
    int number1 = 10;
    int number2 = 15;
    int sum = number1 + number2;

    printf("The sum is: %d\n", sum);
    // printf("The product is: %d\n", product);
}
```

```
}  
    return 0;  
}
```

现在，代码运行没有任何错误。

在这里，我们通过注释掉与 `product` 相关的代码来解决错误。

如果我们需要在不久的将来计算乘积，我们可以取消注释。

---

## 为什么要使用注释？

出于以下原因，我们应该使用注释：

- 注释使我们的代码可读，以供将来参考。
- 注释被用于调试目的。
- 我们可以使用注释进行代码协作，因为它可以帮助同行开发人员理解我们的代码。

**注：**注释不是也不应该用作解释写得不好的代码的替代品。始终尝试编写干净、易于理解的代码，然后使用注释作为补充。

在大多数情况下，总是使用注释来解释“为什么”而不是“如何”，这样更好。

## 1.3 变量、常量和字面量

### 变量

在编程中，变量是保存数据的容器（存储区域）。

为了指示存储区域，应为每个变量指定一个唯一的名称（标识符）。变量名称只是内存位置的符号表示形式。例如：

```
int age = 25;
```

这里，`age` 是一个 `int` 类型的变量，我们给它分配了一个整数值。

变量的值可以更改，因此称为变量。

```
char ch = 'a';  
// 一些代码  
ch = '1';
```

如需要深入了解，可以阅读 1.4 C 的数据类型。

#### 命名变量的规则是什么？

1. 变量名称只能包含字母（大写和小写字母）、数字和下划线。
2. 变量的第一个字母应该是字母或下划线。
3. 没有关于变量名称（标识符）可以有多长的规定。但是，如果变量名称超过 31 个字符，则在某些编译器中可能会遇到问题。

**注：**你应该始终尝试为变量提供有意义的名称。例如：`firstName` 是比 `fn` 更好的变量名称。

#### 我们可以更改变量的数据类型吗？

C 是一种强类型语言。这意味着变量类型一旦声明就无法更改。例如：

```
int number = 5;      // 整数型变量  
number = 5.5;        // 错误  
double number;       // 错误
```

这里，数字变量的类型是 `int`。你不能将浮点（十进制）值 5.5 分配给这个变量。此外，不能将变量的数据类型重新定义为双精度。顺便说一句，要将十进制值存储在 C 中，你需要将其类型声明为 `double` 或 `float`。

## 常量

如果要定义其值无法更改的变量，可以使用 `const` 关键字。这将创建一个常量。例如：

```
const double PI = 3.14;
```

请注意，我们添加了 `const` 关键字。

这里，PI 是一个符号常数，其值无法更改。

```
const double PI = 3.14;  
PI = 2.9; //错误
```

你也可以使用 `#define` 预处理器指令定义一个常量。我们将在 [C 语言的宏教程](#) 中了解它。

## 字面量

字面量是用于表示固定值的数据。它们可以直接在代码中使用。例如：`1`、`2.5`、`'c'` 等。

这里，`1`、`2.5` 和 `'c'` 是字面量。为什么？你不能为这些特殊的项分配不同的值。

### 1. 整数

整数是没有任何小数或指数部分的数字文字（与数字相关联）。C 语言编程中有三种类型的整数文字：

- 十进制（以 10 为基数）
- 八进制（以 8 为基数）
- 十六进制（以 16 为基数）

例如：

```
Decimal: 0, -9, 22 etc  
Octal: 021, 077, 033 etc
```

Hexadecimal: 0x7f, 0x2a, 0x521 等等……

在 C 语言编程中，八进制以 `0` 开头，十六进制以一个 `0x` 开头。

## 2. 浮点数字面量

浮点字面量是具有小数形式或指数形式的数字文字。例如：

```
-2.0
0.0000234
-0.22E-5
```

注：E-5 =  $10^{-5}$

## 3. 字符

一个字符字面量是通过将单个字符括在单引号内来创建的。例如：'`a`' '`m`' '`F`' '`2`' '`}`' 等。

## 4. 字符串字面量

字符串文本是用双引号括起来的字符序列。例如：

```
"good"           //字符串字面量
""               //空字符串字面量
"      "         //有六个白空格的字符串字面量
"x"              //只有一个字符的字符串字面量
"Earth is round\n" //字符串后续跟着换行符
```

## 5. 转义序列

有时，有必要使用无法键入或在 C 中具有特殊含义的字符。例如：换行符（Enter）、制表符、问号等。

为了使用这些字符，使用转义序列。

转义序列	对应字符
<code>\b</code>	退格符
<code>\f</code>	表单填充、走纸符、换页
<code>\n</code>	换行符
<code>\r</code>	回车、返回
<code>\t</code>	水平制表符
<code>\v</code>	垂直制表符
<code>\\</code>	反斜杠
<code>\'</code>	单引号
<code>\"</code>	双引号
<code>\?</code>	问号
<code>\0</code>	Null（空）字符

如：`\n` 用于创建新行，这里的反斜杠 `\` 会使得编译器不以对待一般字符的方式对待它。



## 1.4 C 的数据类型

在 C 中，数据类型是变量的声明。这决定了与变量关联的数据的类型和大小。例如

```
int myVar;
```

这里，`myVar` 是 `int` (`integer`) 类型的变量。`int` 的大小为 4 个字节。

### 基本类型

下面是一个表格，其中包含 C 中常用的类型，以便快速访问。

类型	大小（字节）	格式说明符
<code>int</code>	至少 2 个，通常为 4 个	<code>%d,%i</code>
<code>char</code>	1	<code>%c</code>
<code>float</code>	4	<code>%f</code>
<code>double</code>	8	<code>%lf</code>
<code>short int</code>	通常是 2 个	<code>%hd</code>
<code>unsigned int</code>	至少 2 个，通常为 4 个	<code>%u</code>
<code>long int</code>	至少 4 个，通常为 8 个	<code>%ld,%li</code>
<code>long long int</code>	至少 8 个	<code>%lld,%lli</code>
<code>unsigned long int</code>	至少 4 个	<code>%lu</code>
<code>unsigned long long int</code>	至少 8 个	<code>%llu</code>
<code>signed char</code>	1	<code>%c</code>

类型	大小（字节）	格式说明符
<code>unsigned char</code>	1	<code>%c</code>
<code>long double</code>	至少 10 个，通常为 12 或 16 个	<code>%Lf</code>

## int 整数型

整数是可以同时具有零、正值和负值但没有十进制值的整数。例如 `0`，`-5`，`10`。

我们可以使用 `int` 来声明一个整数变量。

```
int id;
```

这里，`id` 是一个整数类型的变量。

你可以在 C 语言编程中一次声明多个变量。例如

```
int id, age;
```

`int` 的大小通常为 4 个字节（32 位）。从 -2147483648 到 2147483647，它可以有  $2^{32}$  个不同的状态。

## float 和 double

`float` 和 `double` 用于保存实数。

```
float salary;
double price;
```

在 C 语言中，浮点数也可以用指数表示。例如

```
float normalizationFactor = 22.442e2;
```

`float` 和 `double` 有什么不同？

`float`（单精度浮点数据类型）的大小为 4 字节。`double`（双精度浮点数据类型）的大小为 8 字节。

---

## char 关键字

关键字 `char` 用于声明字符类型变量。例如

```
char test = 'h';
```

字符变量的大小为 1 字节。

---

## void

`void` 是一个不完整的类型。它的意思是“没有”或“没有类型”。你可以把 `void` 看作缺失。

例如，如果一个函数没有返回任何内容，那么它的返回类型应该是 `void`。

请注意，不能创建 `void` 类型的变量。

---

## short 和 long

如果需要使用较大的数字，可以使用类型 `long` 说明符。方法如下：

```
long a;  
long long b;  
long double c;
```

在这里，变量 `a` 和 `b` 可以存储整数值。而且，`c` 可以存储浮点数。

如果你确信只会使用一个小整数（[-32767, +32767]范围），则可以使用 `short`。

```
short d;
```

你始终可以使用运算符 `sizeof()` 检查变量的大小。

```
#include <stdio.h>
int main()
{
    short a;
    long b;
    long long c;
    long double d;

    printf("size of short = %d bytes\n", sizeof(a));
    printf("size of long = %d bytes\n", sizeof(b));
    printf("size of long long = %d bytes\n", sizeof(c));
    printf("size of long double= %d bytes\n", sizeof(d));
    return 0;
}
```

## 有符号和无符号

在 C 语言中，`signed` 和 `unsigned` 是类型修饰符。你可以使用它们来修改一个数据类型的数据存储方式：

- `signed`（有符号）- 允许存储正数和负数
- `unsigned`（无符号）- 只允许存储正数

例如：

```
// 有效代码
unsigned int x = 35;
int y = -35; // 有符号整数
int z = 36; // 有符号整数

// 无效代码
unsigned int num = -35;
```

这里，变量 `x` 和 `num` 只能包含零值和正值，因为我们使用了 `unsigned` 修饰符。

考虑到 `int` 的大小是 4 个字节，变量 `y` 可以保存  $-2^{31}-1$  到  $2^{31}-1$  的值，而变量 `x` 可以保存 0 到  $2^{32}-1$  的值。

## 派生数据类型

派生自基本数据类型的数据类型是派生类型。例如：数组、指针、函数类型、结构等。

我们将在后面的教程中了解这些派生数据类型。

- 布尔型
- 枚举类型
- 复杂类型

## 1.5 输入输出 (I/O)

### 输出

在 C 中，`printf()` 是主要的输出函数之一。该函数将格式化的输出发送到屏幕。例如：

#### 示例 1：输出

```
#include <stdio.h>
int main()
{
    // 显示双引号内的字符串
    printf("C Programming");
    return 0;
}
```

输出

```
C Programming
```

这个程序是如何运作的？

所有有效的 C 程序都必须包含 `main()` 函数。代码执行从 `main()` 函数开始。

`printf()` 是一个库函数，用于将格式化的输出发送到屏幕。函数打印引号中的字符串。

要在程序中使用 `printf()`，我们需要使用包含 `stdio.h` 的头文件来使用 `#include <stdio.h>` 语句。

返回 `0`；表示 `main()` 函数中的语句是程序的“退出状态”。这是可选的。

#### 示例 2：整数输出

```
#include <stdio.h>
int main()
```

```
{  
    int testInteger = 5;  
    printf("Number = %d", testInteger);  
    return 0;  
}
```

输出

```
Number = 5
```

我们使用 `%d` 格式说明符来打印 `int` 类型。这里，引号中的 `%d` 将被 `testInteger` 的值所取代。

---

### 示例 3：浮点数和双精度输出

```
#include <stdio.h>  
int main()  
{  
    float number1 = 13.5;  
    double number2 = 12.4;  
  
    printf("number1 = %f\n", number1);  
    printf("number2 = %lf", number2);  
    return 0;  
}
```

输出

```
number1 = 13.500000  
number2 = 12.400000
```

要打印浮点值，我们使用 `%f` 格式说明符。同样，我们使用 `%lf` 来打印双值。

---

### 示例 4：打印字符



```
#include <stdio.h>
int main()
{
    char chr = 'a';
    printf("character = %c", chr);
    return 0;
}
```

输出

```
character = a
```

要打印字符，我们使用%c 格式说明符。

---

## 输入

在 C 中，scanf () 是从用户那里获取输入的常用函数之一。scanf () 函数从标准输入（如键盘）读取格式化的输入。

---

### 示例 5：整数输入/输出

```
#include <stdio.h>
int main()
{
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d", &testInteger);
    printf("Number = %d", testInteger);
    return 0;
}
```

输出

```
Enter an integer: 4
Number = 4
```

这里，我们在 `scanf()` 函数中使用了 `%d` 格式说明符来接受用户的 `int` 输入。当用户输入一个整数时，它会存储在 `testInteger` 变量中。

注：请注意，我们在 `scanf()` 中使用了 `&testInteger`。这是因为 `&testInteger` 获取 `testInteger` 的地址，用户输入的值存储在该地址中。

## 示例 6：单精度浮点数（float）和双精度浮点数（double）的输入/输出

```
#include <stdio.h>
int main()
{
    float num1;
    double num2;

    printf("Enter a number: ");
    scanf("%f", &num1);
    printf("Enter another number: ");
    scanf("%lf", &num2);

    printf("num1 = %f\n", num1);
    printf("num2 = %lf", num2);

    return 0;
}
```

输出

```
Enter a number: 12.523
Enter another number: 10.2
num1 = 12.523000
num2 = 10.200000
```

我们分别使用 `%f` 和 `%lf` 针对 `float` 和 `double` 作为格式化说明符。

## 示例 7：字符输入/输出

```
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c",&chr);
    printf("You entered %c.", chr);
    return 0;
}
```

输出

```
Enter a character: g
You entered g
```

当用户在上述程序中输入字符时，不会存储字符本身。相反，将存储整数值（ASCII 值）。

当我们使用文本格式 `%c` 显示该值时，将显示输入的字符。如果我们使用 `%d` 显示字符，则打印它的 ASCII 值。

## 示例 8：ASCII 值

```
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c", &chr);

    // 当 %c 被使用，将显示一个字符
    printf("You entered %c.\n",chr);

    // 当 %d 被使用，ASCII 值将被显示出来
}
```

```
printf("ASCII value is %d.", chr);  
return 0;  
}
```

输出

```
Enter a character: g  
You entered g.  
ASCII value is 103.
```

---

## 输入/输出 (I/O) 多个值

下面介绍了如何从用户那里获取多个输入并显示它们。

```
#include <stdio.h>  
int main()  
{  
    int a;  
    float b;  
  
    printf("Enter integer and then a float: ");  
  
    // Taking multiple inputs  
    scanf("%d%f", &a, &b);  
  
    printf("You entered %d and %f", a, b);  
    return 0;  
}
```

输出

```
Enter integer and then a float: -3  
3.4  
You entered -3 and 3.400000
```

## 输入/输出（I/O）的格式说明符

从上面的例子中可以看出，我们针对不同的类型，使用不同的格式说明符，如：

- `%d` 针对 `int`
- `%f` 针对 `float`
- `%lf` 针对 `double`
- `%c` 针对 `char`

下面是常用的 C 的数据类型及其格式说明符的列表。

数据类型	格式说明符
<code>int</code>	<code>%d</code>
<code>char</code>	<code>%c</code>
<code>float</code>	<code>%f</code>
<code>double</code>	<code>%lf</code>
<code>short int</code>	<code>%hd</code>
<code>unsigned int</code>	<code>%u</code>
<code>long int</code>	<code>%li</code>
<code>long long int</code>	<code>%lli</code>
<code>unsigned long int</code>	<code>%lu</code>
<code>unsigned long long int</code>	<code>%llu</code>
<code>signed char</code>	<code>%c</code>
<code>unsigned char</code>	<code>%c</code>

数据类型	格式说明符
long double	%Lf

## 1.6 运算符

运算符是对值或变量进行操作的符号。例如：`+`是执行加法的运算符。

C 具有丰富的运算符来执行各种操作。

### 算术运算符

算术运算符对数值（常数和变量）执行数学运算，例如加法、减法、乘法、除法等。

运算符	运算符的含义
<code>+</code>	求和或一元加号
<code>-</code>	求差或一元减号
<code>*</code>	求积
<code>/</code>	求商
<code>%</code>	求余（求模），除后的余数

#### 示例 1：算术运算符

```
// 算术运算符的使用
#include <stdio.h>
int main()
{
    int a = 9, b = 4, c;

    c = a+b;
    printf("a+b = %d \n", c);
    c = a-b;
    printf("a-b = %d \n", c);
    c = a*b;
    printf("a*b = %d \n", c);
    c = a/b;
    printf("a/b = %d \n", c);
    c = a%b;
    printf("Remainder when a divided by b = %d \n", c);

    return 0;
}
```



}

输出

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

运算符`+`、`-`和`*`分别计算加法、减法和乘法。

在常规计算中， $9/4=2.25$ 。但是，程序中的输出为 2。

这是因为变量 `a` 和 `b` 都是整数。因此，输出也是一个整数。编译器忽略小数点后的项，并显示答案 2 而不是 2.25。

模运算符`%`计算余数。当 `a=9` 除以 `b=4` 时，余数为 1。`%`运算符只能与整数一起使用。

假设 `a=5.0`，`b=2.0`，`c=5`，`d=2`。然后在 C 编程中，

```
// 任意一个操作数都是浮点数
a/b = 2.5
a/d = 2.5
c/b = 2.5

// 两个操作数都是整数
c/d = 2
```

## 递增和递减运算符

C 编程有两个运算符，递增`++`和递减`--`将操作数（常量或变量）的值更改 1。

递增`++`使值增加 1，而递减`--`使值减少 1。这两个运算符是一元运算符，这意味着它们只对单个操作数进行运算。

### 示例 2：递增和递减运算符

```
// 递增和递减运算符
```

```
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;

    printf("++a = %d \n", ++a);
    printf("--b = %d \n", --b);
    printf("++c = %f \n", ++c);
    printf("--d = %f \n", --d);

    return 0;
}
```

输出

```
++a = 11
--b = 99
++c = 11.500000
--d = 99.500000
```

这里，运算符++和--用作前缀。这两个运算符也可以用作后缀，如一个a++和一个a--。

## 赋值运算符

赋值运算符用于为变量赋值。最常见的赋值运算符是=。

运算符	示例	等效于
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b

运算符	示例	等效于
<code>%=</code>	<code>a %= b</code>	<code>a = a%b</code>

### 示例 3：赋值运算符

```
// 赋值运算符
#include <stdio.h>
int main()
{
    int a = 5, c;

    c = a;      // c is 5
    printf("c = %d\n", c);
    c += a;     // c 为 10
    printf("c = %d\n", c);
    c -= a;     // c 为 5
    printf("c = %d\n", c);
    c *= a;     // c 为 25
    printf("c = %d\n", c);
    c /= a;     // c 为 5
    printf("c = %d\n", c);
    c %= a;     // c = 0
    printf("c = %d\n", c);

    return 0;
}
```

输出

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

### 关系运算符

关系运算符检查两个操作数之间的关系。如果关系为 `true`，则返回 1；如果关系为 `false`，则返回值 0。

关系运算符用于决策和循环。

运算符	运算符的含义	例
==	等于	5 == 3 计算结果为 0
>	大于	5 > 3 计算结果为 1
<	小于	5 < 3 计算结果为 0
!=	不等于	5 != 3 计算结果为 1
>=	大于或等于	5 >= 3 计算结果为 1
<=	小于或等于	5 <= 3 计算结果为 0

#### 示例 4：关系运算符

```
// 关系运算符的工作原理
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;

    printf("%d == %d is %d \n", a, b, a == b);
    printf("%d == %d is %d \n", a, c, a == c);
    printf("%d > %d is %d \n", a, b, a > b);
    printf("%d > %d is %d \n", a, c, a > c);
    printf("%d < %d is %d \n", a, b, a < b);
    printf("%d < %d is %d \n", a, c, a < c);
    printf("%d != %d is %d \n", a, b, a != b);
    printf("%d != %d is %d \n", a, c, a != c);
    printf("%d >= %d is %d \n", a, b, a >= b);
    printf("%d >= %d is %d \n", a, c, a >= c);
    printf("%d <= %d is %d \n", a, b, a <= b);
    printf("%d <= %d is %d \n", a, c, a <= c);

    return 0;
}
```

输出

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
```

```

5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1

```

## 逻辑运算符

包含逻辑运算符的表达式返回 0 或 1，具体取决于表达式结果是 true 还是 false。逻辑运算符通常用于 C 编程中的决策。

运算符	意义	例
&&	逻辑 与 仅当所有操作数均为 true 时才为 True	如果 c = 5 且 d = 2，则表达式等于 0。 <code>((c==5) &amp;&amp; (d&gt;5))</code>
	逻辑 或 仅当任一操作数为 true 时才为 True	如果 c = 5 且 d = 2，则表达式等于 1。 <code>((c==5)    (d&gt;5))</code>
!	逻辑 非 仅当操作数为 0 时为 True	如果 c = 5，则表达式等于 0。 <code>!(c==5)</code>

### 示例 5：逻辑运算符

```

// 逻辑运算符示例

#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;

    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) is %d \n", result);

    result = (a == b) && (c < b);
    printf("(a == b) && (c < b) is %d \n", result);

    result = (a == b) || (c < b);
    printf("(a == b) || (c < b) is %d \n", result);

    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) is %d \n", result);

    result = !(a != b);
    printf("(a != b) is %d \n", result);

    result = !(a == b);

```

```
printf("!(a == b) is %d \n", result);

return 0;
}
```

输出

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

## 逻辑运算符程序说明

- `(a==b)&&(c>5)` 计算结果为 1，因为两个操作数 `(a==b)` 和 `(c>5)` 都是 1（true）。
- `(a==b)&&(c<b)` 计算结果为 0，因为操作数 `(c<b)` 为 0（false）。
- `(a==b)|| (c<b)` 计算结果为 1，因为 `(a==b)` 为 1（true）。
- `(a!=b)|| (c<b)` 计算结果为 0，因为操作数 `(a!=b)` 和 `(c<b)` 均为 0（false）。
- `!(a!=b)` 计算结果为 1，因为操作数 `(a!=b)` 为 0（false）。因此 `!(a!=b)` 为 1（true）。
- `!(a==b)` 计算结果为 0，因为 `(a==b)` 为 1（true）。因此，`!(a==b)` 为 0（false）。

## 位运算符

在计算过程中，加法、减法、乘法、除法等数学运算被转换为位级，从而加快处理速度并节省功耗。按位运算符在 C 编程中用于执行位级别的运算。

### 运算符

### 运算符的含义

&

按位 与

|

按位 或

^

按位 异或

运算符	运算符的含义
~	按位补码
<<	左移
>>	右移

## 其他运算符

---

### 逗号运算符

逗号运算符用于将相关表达式链接在一起。例如：

```
int a, c = 5, d;
```

### sizeof 运算符

这 `sizeof` 是一个一元运算符，返回数据的大小（常量、变量、数组、结构等）。

#### 示例 6: sizeof 运算符

```
#include <stdio.h>
int main()
{
    int a;
    float b;
    double c;
    char d;
    printf("Size of int=%lu bytes\n",sizeof(a));
    printf("Size of float=%lu bytes\n",sizeof(b));
    printf("Size of double=%lu bytes\n",sizeof(c));
    printf("Size of char=%lu byte\n",sizeof(d));

    return 0;
}
```



## 输出

```
Size of int = 4 bytes  
Size of float = 4 bytes  
Size of double = 8 bytes  
Size of char = 1 byte
```

---

其他运算符，如三元运算符 `?:`、引用运算符 `&`、取消引用运算符 `*` 和成员选择运算符 `->` 将在后面的教程中讨论。

## 2 流程控制

### 2.1 If...else 语句

#### if 语句

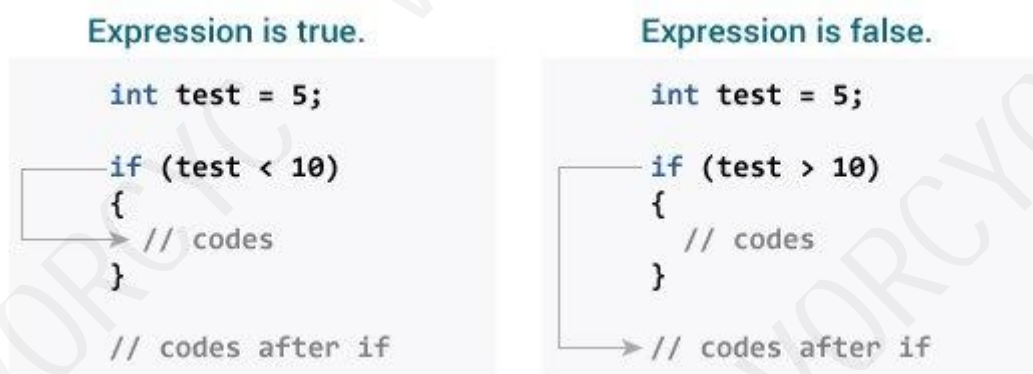
C 中 `if` 语句的语法为：

```
if (测试表达式)
{
    // 代码
}
```

#### if 语句如何工作？

该 `if` 语句计算圆括号（）内的测试表达式。

- 如果测试表达式的求值结果为 `true`，则执行 `if` 主体内的语句。
- 如果测试表达式的计算结果为 `false`，则不执行 `if` 正文中的语句。



if 语句的工作原理

若要详细了解何时将测试表达式计算为 `true`（非零值）和 `false`（0），可以了解逻辑运算符和关系运算符。

## 示例 1: if 语句

```
// 显示一个数字是否为负数
#include <stdio.h>
int main()
{
    int number;

    printf("Enter an integer: ");
    scanf("%d", &number);

    // 若数值小于 0 则为 true
    if (number < 0)
    {
        printf("You entered %d.\n", number);
    }
    printf("The if statement is easy.");

    return 0;
}
```

### 输出 1

```
Enter an integer: -2
You entered -2.
The if statement is easy.
```

当用户输入-2时，测试表达式 `number<0` 的计算结果为 `true`。因此，屏幕上将显示你输入的-2。

### 输出 2

```
Enter an integer: 5
The if statement is easy.
```

当用户输入 5 时，测试表达式 `number<0` 将被评估为 `false`，并且不执行 if 正文中的语句。

## if...else 语句

if 语句可能有一个可选的 else 块。if...else 语句的语法为：

```
if (测试表达式)
{
    // 如果测试表达式为 true 则执行这里的代码
}
else
{
    // 如果测试表达式为 false 则执行这里的代码
}
```

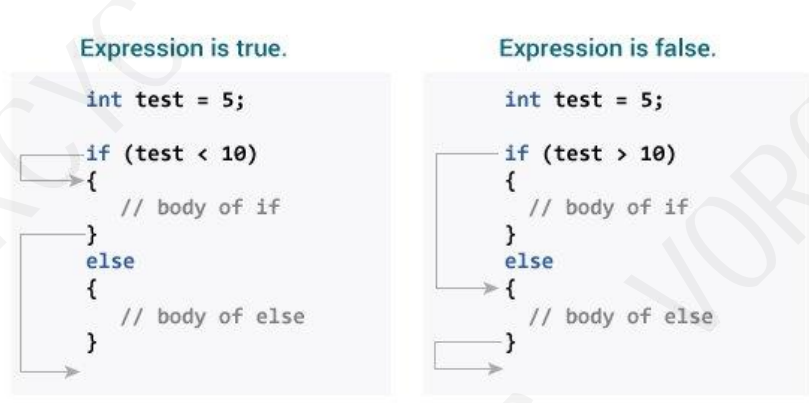
### if...else 如何工作？

如果测试表达式被评估为 true，

- 执行 if 主体内部的语句。
- else 主体内的语句将从执行中跳过。

如果测试表达式的计算结果为 false，

- 执行 else 主体内的语句。
- 从执行中跳过 if 主体内的语句。



If...else 语句

### 示例：if...else 语句

```
//检查一个整数是奇数还是偶数
#include <stdio.h>

int main()
{
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);
    // 若余数为0 则为 true
    if (number%2 == 0)
    {
        printf("%d is an even integer.",number);
    }
    else
    {
        printf("%d is an odd integer.",number);
    }
    return 0;
}
```

输出

```
Enter an integer: 7
7 is an odd integer.
```

当用户输入 7 时，测试表达式 `number%2==0` 的计算结果为 `false`。因此，`else` 主体内部的语句被执行。

## if...else 分支

`if...else` 语句执行两个不同的代码，具体取决于测试表达式是 `true` 还是 `false`。有时，必须从两种以上的可能性中做出选择。

`if...else` 分支允许你在多个测试表达式之间进行检查并执行不同的语句。

## if...else 分支的语法

```
if (测试表达式 1)
{
    // 代码语句
}
else if(测试表达式 2)
{
```

```

    // 代码语句
}
else if (测试表达式 3)
{
    // 代码语句
}
.
.
else
{
    // 代码语句
}

```

### 示例 3: if...else 分支

```

// 使用 == , > 或 < 对比两个整数
#include <stdio.h>

int main()
{
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    //检查两个整数是否相等
    if(number1 == number2)
    {
        printf("Result: %d = %d", number1, number2);
    }

    //检查是否 number1 大于 number2.
    else if (number1 > number2)
    {
        printf("Result: %d > %d", number1, number2);
    }
    //检查是否两个表达式都是 false
    else
    {
        printf("Result: %d < %d", number1, number2);
    }
    return 0;
}

```

输出

```

Enter two integers: 12
23
Result: 12 < 23

```

## 嵌套 if...else

有可能在另一个 `if...else` 语句的正文中包含一个 `if...else` 语句。

### 示例 4：嵌套 if...else

下面给出的这个程序使用 `<`、`>` 和 `=` 将两个整数关联起来，类似于 `if...else` 分支的例子。但是，我们将使用嵌套的 `if...else` 语句来解决此问题。

```
#include <stdio.h>

int main()
{
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    if (number1 >= number2)
    {
        if (number1 == number2)
        {
            printf("Result: %d = %d", number1, number2);
        }
        else
        {
            printf("Result: %d > %d", number1, number2);
        }
    }
    else
    {
        printf("Result: %d < %d", number1, number2);
    }

    return 0;
}
```

注：如果 `If...else` 语句的主体只有一个语句，则不需要使用方括号 `{}`。

例如，以下代码

```
if (a > b)
{
```

```
    printf("Hello");  
}  
printf("Hi");
```

相当于

```
if (a > b)  
    printf("Hello");  
printf("Hi");
```



## 2.2 for 循环

在编程中，循环用于重复代码块，直到满足指定的条件。

C 语言中有三种类型的循环：

1. `for` 循环
2. `while` 循环
3. `do...while` 循环

我们将在本教程中学习 `for` 循环。在下一个教程中，我们将学习 `while` 和 `do...while` 循环。

---

### for 循环

`for` 循环的语法为：

```
for (初始化语句 ; 测试表达式 ; 更新状态)
{
    // 循环体中的语句
}
```

---

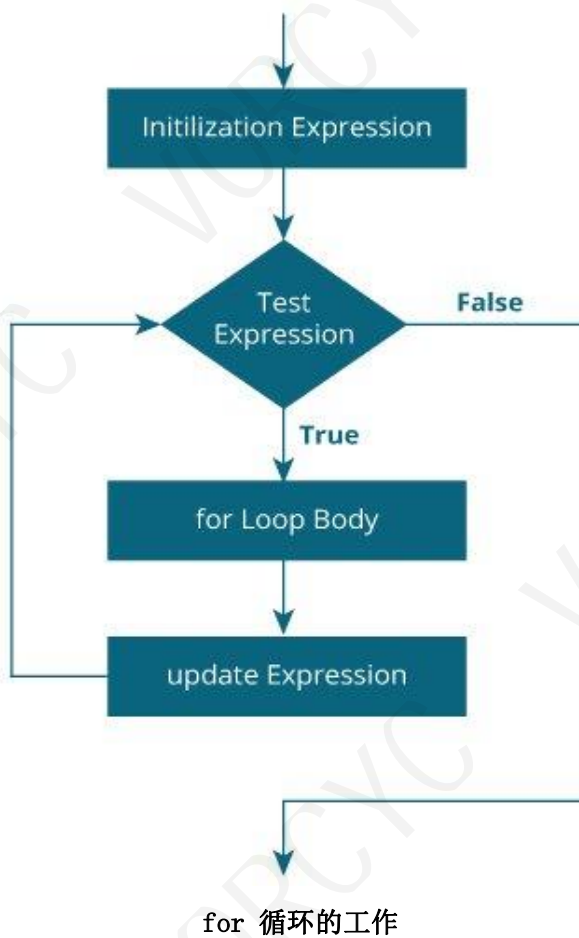
### for 循环是如何工作的？

- 初始化语句仅执行一次。
- 然后，计算测试表达式。如果测试表达式的计算结果为 `false`，则 `for` 循环将终止。
- 但是，如果测试表达式的求值结果为 `true`，则执行 `for` 循环主体内的语句，并更新更新表达式。
- 再次计算测试表达式。

此过程一直持续到测试表达式为 `false`。当测试表达式为 `false` 时，循环终止。

若要了解有关测试表达式的详细信息（当测试表达式计算为 `true` 和 `false` 时），请查看关系运算符和逻辑运算符。

## for 循环流程图



## 示例 1: for 循环

```
// 打印从 1 到 10 的数字
#include <stdio.h>

int main()
{
    int i;

    for (i = 1; i < 11; ++i)
    {
        printf("%d ", i);
    }
    return 0;
}
```

输出

```
1 2 3 4 5 6 7 8 9 10
```

1. `i` 被初始化为 1。
2. 评估测试表达式 `i<11`。由于小于 11 的 1 为真，因此执行 `for` 循环的主体。这将在屏幕上打印 1 (`i` 的值)。
3. 执行更新语句 `++i`。现在，`i` 的值将是 2。再次，测试表达式被求值为 `true`，并执行 `for` 循环的主体。这将在屏幕上打印 2 (`i` 的值)。
4. 再次，执行更新语句 `++i`，并评估测试表达式 `i<11`。这个过程一直持续到 `i` 成为 11。
5. 当 `i` 变为 11 时，`i<11` 将为 `false`，`for` 循环终止。

## 示例 2: for 循环

```
// 计算前 n 个自然数之和的程序
// 正整数 1,2,3...n 称为自然数
#include <stdio.h>

int main()
{
    int num, count, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    // 当 num 小于 count 时，for 循环终止
    for(count = 1; count <= num; ++count)
    {
        sum += count;
    }

    printf("Sum = %d", sum);

    return 0;
}
```

### 输出

```
Enter a positive integer: 10
Sum = 55
```

用户输入的值存储在变量中。假设用户输入了 10。

`count` 被初始化为 1，并计算测试表达式。由于测试表达式 `count<=num` (1 小于或等于 10) 为 `true`，因此执行 `for` 循环的主体，`sum` 的值将等于 1。

然后，执行更新 `++count` 语句，`count` 将等于 2。再次评估测试表达式。由于 2 也小于 10，因此测试表达式被评估为 `true`，并执行 `for` 循环的主体。现在，和等于 3。

这个过程继续进行，并计算总和，直到计数 `count` 达到 11。

当计数 `count` 为 11 时，测试表达式被求值为 0 (`false`)，循环终止。

然后，`sum` 的值被打印在屏幕上。

---

我们将在下一个教程中学习 `while` 循环和 `do...while` 循环。

## 2.3 while and do...while 循环

在编程中，循环用于重复代码块，直到满足指定的条件。

C 语言有三种类型的循环：

4. `for` 循环
5. `while` 循环
6. `do...while` 循环

在上一个教程中，我们了解了 `for` 循环。在本教程中，我们将学习 `while` 和 `do...while` 循环。

---

### while 循环

`while` 循环的语法是：

```
while (测试表达式)
{
    // 循环体
}
```

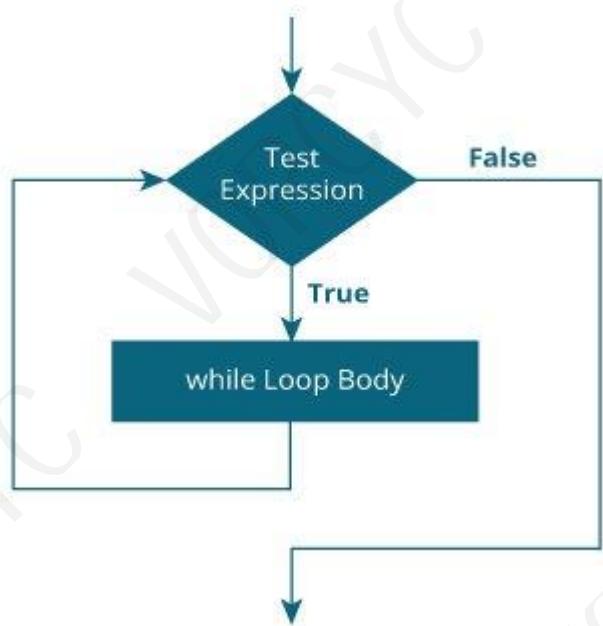
---

### while 循环如何工作？

- `while` 循环计算圆括号 ( ) 内的测试表达式。
- 如果测试表达式为 true，则执行 `while` 循环正文中的语句。然后，再次评估测试表达式。
- 该过程一直持续到测试表达式的计算结果为 false。
- 如果测试表达式为 false，则循环终止（结束）。

要了解有关测试表达式的更多信息（当测试表达式的计算结果为 true 和 false 时），请查看关系运算符和逻辑运算符。

### while 循环流程图



while 循环的工作流程

## 示例 1: while 循环

```
// 打印从 1 到 5 的数字
#include <stdio.h>
int main()
{
    int i = 1;

    while (i <= 5)
    {
        printf("%d\n", i);
        ++i;
    }

    return 0;
}
```

### 输出

```
1
2
3
4
5
```

在这里，我们已初始化 `i` 为 1。

1. 当 `i=1` 时，测试表达式 `i<=5` 为真。因此，`while` 循环的主体被执行。这将在屏幕上打印 1，`i` 的值将增加到 2。
2. 现在，`i=2`，测试表达式 `i<=5` 再次为真。`while` 循环的主体将再次执行。这将在屏幕上打印 2，`i` 的值将增加到 3。
3. 这个过程一直持续到当 `i` 变成 6。然后，测试表达式 `i<=5` 将为 `false`，循环终止。

---

## do...while 循环

`do...while` 循环与 `while` 循环相似，但有一个重要区别。`do...while` 循环的主体至少执行一次。只有这样，才会对测试表达式求值。

`do...while` 循环的语法为：

```
do
{
    // 循环体
}while (测试表达式);
```

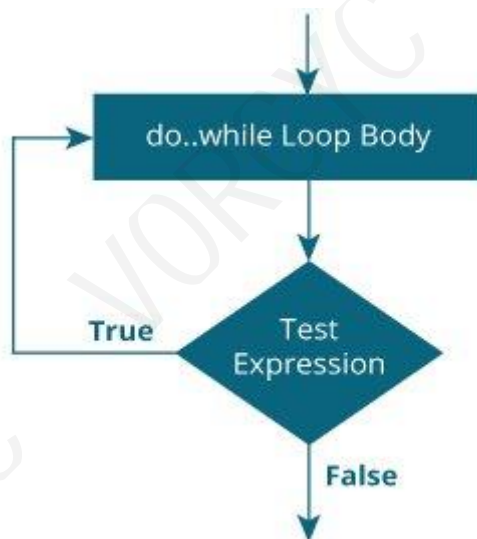
---

## do...while 循环有效？

- `do...while` 循环的主体执行一次。只有到那时，才会计算测试表达式。
- 如果测试表达式为 `true`，则会再次执行循环的主体，并再次计算测试表达式。
- 这个过程一直持续到测试表达式变为 `false`
- 如果测试表达式为 `false`，则循环结束。

---

## do while 循环执行流程图



do...while 循环的工作流程

## 示例 2：执行...while 循环

```
//程序添加数字直到用户输入零
#include <stdio.h>int main() {
    double number, sum = 0;

    // 循环的主体至少执行一次
    do
    {
        printf("Enter a number: ");
        scanf("%lf", &number);
        sum += number;
    }
    while(number != 0.0);

    printf("Sum = %.2lf",sum);

    return 0;
}
```

### 输出

```
Enter a number: 1.5
Enter a number: 2.4
Enter a number: -3.4
Enter a number: 4.2
Enter a number: 0
Sum = 4.70
```



在这里，我们使用了 `do...while` 循环来提示用户输入一个数字。只要输入的数字不是 0，循环就会工作。

`do...while` 循环至少执行一次，即第一次迭代在不检查条件的情况下运行。只有在执行了第一次迭代之后，才会检查条件。

```
do {  
    printf("Enter a number: ");  
    scanf("%lf", &number);  
    sum += number;  
}while(number != 0.0);
```

因此，如果第一个输入是一个非零值，那么这个数字将被添加到 `sum` 变量中，循环将继续到下一次迭代。重复此过程，直到用户输入 0。

但是，如果第一个输入为 0，则不会有循环的第二次迭代，并且 `sum` 变为 0.0。

在循环外，我们打印 `sum` 的值。

## 2.4 break 和 continue 语句

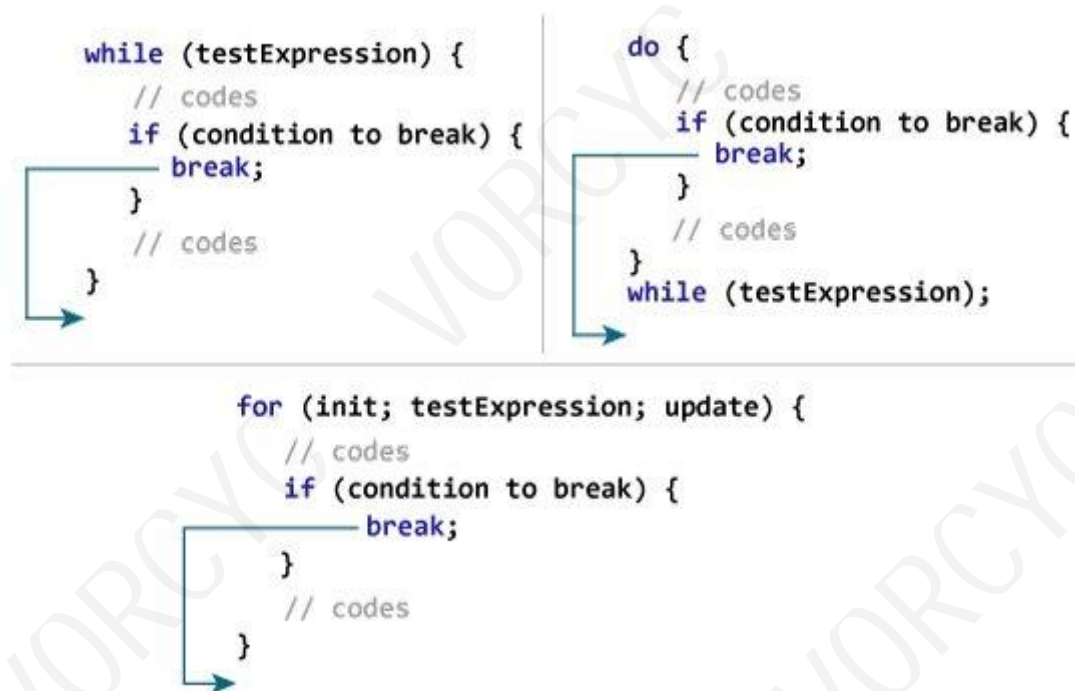
### break 语句

break 语句在遇到循环时立即结束循环。它的语法是：

```
break;
```

break 语句在循环中几乎总是与 if...else 语句一起使用。

break 语句的工作原理是什么？



break 的工作原理

### 示例 1: break 语句

```
//计算数字总和的程序（最多 10 个数字）
//如果用户输入负数，循环终止
```

```
#include <stdio.h>
int main() {
    int i;
    double number, sum = 0.0;

    for (i = 1; i <= 10; ++i) {
        printf("Enter n%d: ", i);
        scanf("%lf", &number);

        // 如果用户输入负数，则中断循环
        if (number < 0.0) {
            break;
        }

        sum += number; // sum = sum + number;
    }

    printf("Sum = %.2lf", sum);

    return 0;
}
```

输出

```
Enter n1: 2.4
Enter n2: 4.5
Enter n3: 3.4
Enter n4: -3
Sum = 10.30
```

该程序最多计算 10 个数字的总和。为什么最多有 10 个数字？这是因为如果用户输入负数，则执行 `break` 语句。这将结束 `for` 循环，并显示 `sum`。

在 C 中，`break` 也与 `switch` 语句一起使用。这将在下一个教程中讨论。

## 示例 2: continue 语句

```
// 计算数字总和的程序（最多 10 个数字）
// 如果用户输入负数，则不会将其添加到结果中
#include <stdio.h>

int main()
{
    int i;
    double number, sum = 0.0;

    for (i = 1; i <= 10; ++i) {
        printf("Enter a n%d: ", i);
```

```
scanf("%lf", &number);

if (number < 0.0) {
    continue;
}

sum += number; // sum = sum + number;
}

printf("Sum = %.2lf", sum);

return 0;
}
```

## 输出

```
Enter n1: 1.1
Enter n2: 2.2
Enter n3: 5.5
Enter n4: 4.4
Enter n5: -3.4
Enter n6: -45.5
Enter n7: 34.5
Enter n8: -4.2
Enter n9: -1000
Enter n10: 12
Sum = 59.70
```

在这个程序中，当用户输入一个正数时，使用 `sum += number;` 语句计算总和。

当用户输入负数时，将执行 `continue` 该语句，并从计算中跳过负数。

## 2.5 switch 语句

switch 语句允许我们在许多备选方案中执行一个代码块。

你也可以用 `if...else...if` 分支完成同样的事情。但是，`switch` 语句的语法读写起来要容易得多。

### switch...case 的语法

```
switch (表达式)
{
    case 常量 1:
        // 代码语句
        break;

    case 常量 2:
        // 代码语句
        break;
    .
    .
    .
    default:
        // 默认语句
}
```

### switch 语句如何工作？

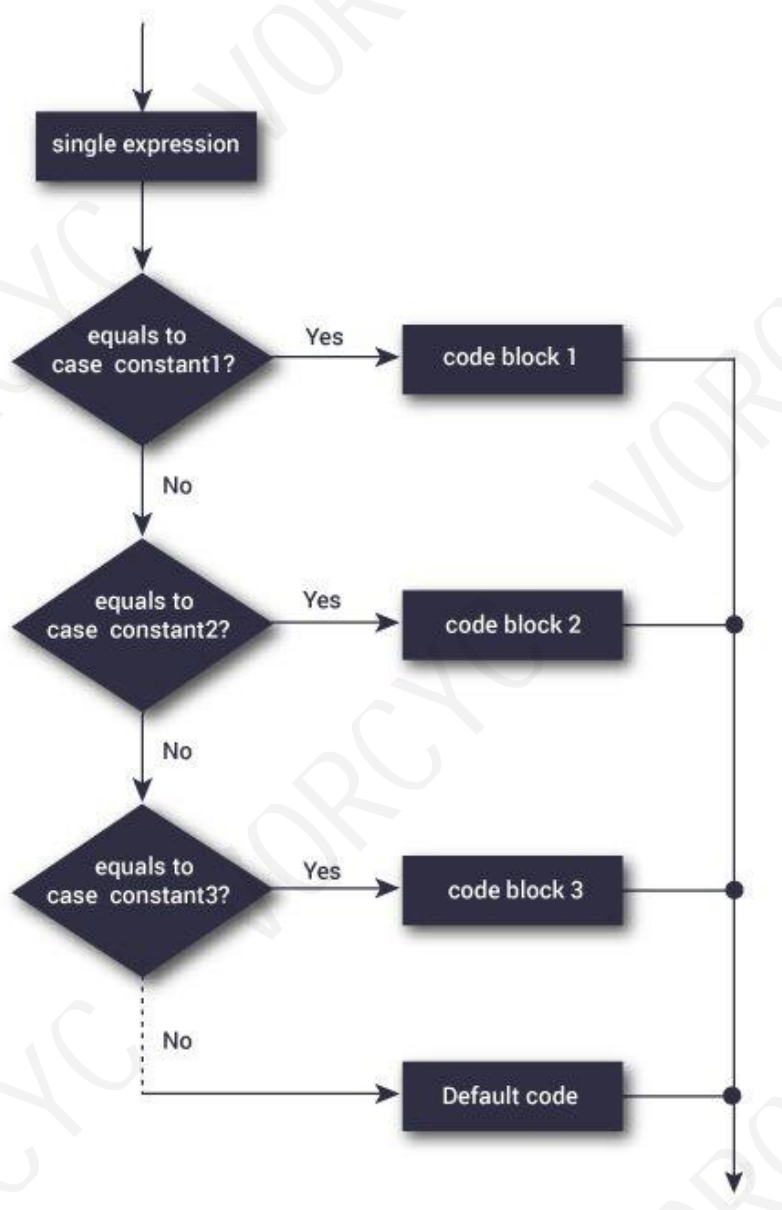
评估一次表达式，并与每个 `case` 标签的值进行比较。

- 如果存在匹配项，则执行匹配标签后的相应语句。例如，如果表达式的值等于 `常量 1`，则执行 `case 常量 2:` 后的语句，直到遇到 `break`。
- 如果没有匹配项，则执行 `default` 后面的语句。

**注意：**

- 如果我们不使用 `break` 语句，则匹配标签后的所有语句都会被执行。
- `switch` 语句中的 `default` 子句是可选的。

## switch 语句流程图



switch 语句流程图

## 示例：简单计算器

```
// 创建简单计算器的程序
#include <stdio.h>

int main()
{
    char operation;
```

```

double n1, n2;

printf("Enter an operator (+, -, *, /): ");
scanf("%c", &operation);
printf("Enter two operands: ");
scanf("%lf %lf",&n1, &n2);

switch(operation)
{
    case '+':
        printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
        break;
    case '-':
        printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
        break;
    case '*':
        printf("%.1lf * %.1lf = %.1lf",n1, n2, n1*n2);
        break;
    case '/':
        printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
        break;
    // 运算符不匹配任何大小写常量+, -, */
    default:
        printf("Error! operator is not correct");
}

return 0;
}

```

输出

```

Enter an operator (+, -, *, /): -
Enter two operands: 32.5
12.4
32.5 - 12.4 = 20.1

```

用户输入的-运算符存储在运算变量中。并且，两个操作数 32.5 和 12.4 分别存储在变量 n1 和 n2 中。

由于 operation 是-，因此程序的控制跳转到

```
printf("%.1lf - %.1lf = %.1lf", n1, n2, n1-n2);
```

最后，break 语句终止 switch 语句。

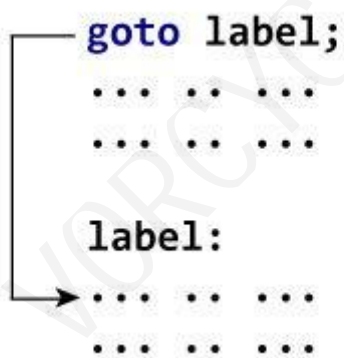
## 2.6 goto 语句

`goto` 语句允许我们将程序的控制权转移到指定的 `label`。

### goto 语句的语法

```
goto label;  
... ..  
... ..  
label:  
statement;
```

`label` 是一个标识符。当遇到 `goto` 语句时，程序的控制将跳转到 `label:` 并开始执行代码。



goto 语句的工作

### 示例：goto 语句

```
// 计算正数总和和平均值的程序  
//如果用户输入负数，则会显示总和和平均数。  
  
#include <stdio.h>  
  
int main()  
{  
  
    const int maxInput = 100;  
    int i;  
    double number, average, sum = 0.0;
```



```

for (i = 1; i <= maxInput; ++i) {
    printf("%d. Enter a number: ", i);
    scanf("%lf", &number);

    // 如果用户输入负数，转到跳转
    if (number < 0.0)
    {
        goto jump;
    }
    sum += number;
}

jump:
average = sum / (i - 1);
printf("Sum = %.2f\n", sum);
printf("Average = %.2f", average);

return 0;
}

```

输出

```

1. Enter a number: 3
2. Enter a number: 4.3
3. Enter a number: 9.3
4. Enter a number: -2.9
Sum = 16.60
Average = 5.53

```

## 避免使用 goto 的原因

使用 `goto` 语句可能会导致代码有错误且难以理解。例如

```

one:
for (i = 0; i < number; ++i)
{
    test += i;
    goto two;
}
two:
if (test > 5)
{
    goto three;
}
... ..

```

此外，`goto` 语句还允许你做坏事，例如跳出范围。

话虽如此，`goto` 有时可能很有用。例如：从嵌套循环中断。

---

## 你应该使用 `goto` 语句吗？

如果你认为 `goto` 语句的使用简化了你的程序，你可以使用它。也就是说，`goto` 很少有用，你可以在不完全使用 `goto` 的情况下创建任何 C 程序。

这里引用 C++ 的创建者 Bjarne Stroustrup 的一句话，“‘`goto`’可以做任何事情，这正是我们不使用它的原因。”

## 3 函数

### 3.1 函数

函数是执行特定任务的代码块。

假设，你需要创建一个程序来创建一个圆圈并为其着色。你可以创建两个函数来解决此问题：

- 创建 `circle` 函数
- 创建 `color` 函数

将一个复杂的问题分成更小的块，使我们的程序易于理解和重用。

---

#### 功能类型

C 编程中有两种类型的函数：

- 标准库函数
  - 用户自定义函数
- 

#### 标准库函数

标准库函数是 C 编程中的内置函数。

这些函数在头文件中定义。例如：

- `printf()` 是一个标准库函数，用于将格式化的输出发送到屏幕(在屏幕上显示输出)。这个函数在 `stdio.h` 头文件中定义。因此，要使用 `printf()` 函数，我们需要使用 `#include <stdio.h>` 包含 `stdio.h` 头文件。
  - `sqrt()` 函数计算数字的平方根。该函数在 `math.h` 头文件中定义。
- 

#### 用户自定义函数

你还可以根据需要创建函数。用户创建的此类函数称为用户定义函数。

## 用户自定义函数如何工作？

```
#include <stdio.h>
void functionName()
{
    ... ..
    ... ..
}

int main()
{
    ... ..
    ... ..

    functionName();

    ... ..
    ... ..
}
```

C 程序的执行从函数 `main()` 开始。

当编译器遇到 `functionName();` 时，程序的控制将跳转到

```
void functionName()
```

然后，编译器开始执行 `functionName()` 中的代码。

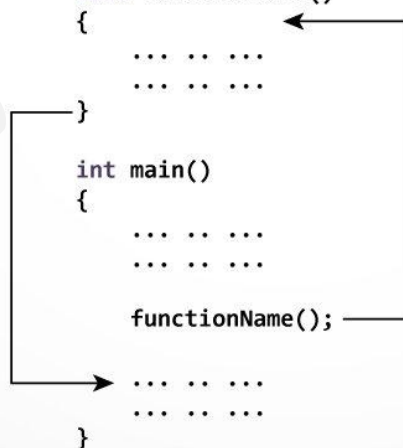
一旦函数定义内的代码被执行，程序的控制权就会跳转回 `main()` 函数。

### How function works in C programming?

```
#include <stdio.h>

void functionName()
{
    ... ..
}

int main()
{
    ... ..
    functionName();
    ... ..
}
```



The diagram illustrates the execution flow in a C program. It shows a function definition for `void functionName()` and a `main()` function. An arrow originates from the `functionName();` call inside `main()`, points to the opening curly brace of `functionName()`, and then returns to the line following the function call in `main()`. This visualizes how the program jumps to the function's code and then resumes execution after the function completes.

C 函数的工作原理

请注意，函数名称是标识符，应该是唯一的。

### 用户自定义函数的优点

1. 该程序将更易于理解、维护和调试。
2. 可在其他程序中使用的可重用代码
3. 一个大程序可以分为更小的模块。因此，一个大项目可以分配给许多程序员。

## 3.2 用户自定义函数

函数是执行特定任务的代码块。

C 允许你根据需要定义函数。这些函数称为用户定义函数。例如：

假设你需要创建一个圆圈并根据半径和颜色为其着色。你可以创建两个函数来解决此问题：

- `createCircle()` 函数
- `color()` 函数

### 示例：用户定义函数

下面是一个添加两个整数的示例。为了执行此任务，我们创建了一个用户自定义的 `addNumbers()`。

```
#include <stdio.h>
int addNumbers(int a, int b);           // 函数原型
int main()
{
    int n1,n2,sum;

    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);           // 函数调用
    printf("sum = %d",sum);

    return 0;
}

int addNumbers(int a, int b)           // 函数定义
{
    int result;
    result = a+b;
    return result;                       // 返回语句
}
```

---

### 函数原型

函数原型只是指定函数名称、参数和返回类型的函数的声明。它不包含函数体。

函数原型向编译器提供该函数以后可能在程序中使用的信息。

## 函数原型的语法

```
返回类型 函数名(类型 1 形参 1, 类型 2 形参 2, ...);
```

在上面的示例中，`int addNumbers(int a, int b)` 是向编译器提供以下信息的函数原型：

- 1.函数的名称是 `addNumbers()`
- 2.函数的返回类型为 `int`
- 3.将两个 `int` 类型的参数传递给函数

如果用户定义的函数在 `main()` 函数之前定义，则不需要函数原型。

## 调用函数

程序的控制权通过调用它转移到用户定义的函数。

## 函数调用的语法

```
函数名(形参 1, 形参 2, ...);
```

在上面的例子中，函数调用使用 `addNumbers(n1, n2);main()` 函数内的语句。

## 函数定义

函数定义包含用于执行特定任务的代码块。在我们的示例中，将两个数字相加并返回它。

## 函数定义的语法

```
返回类型 函数名(类型 1 形参 1, 类型 2 形参 2, ...)
{
    //函数体
}
```

调用函数时，程序的控制权将转移到函数定义中。并且，编译器开始执行函数主体内的代码。

## 将参数传递给函数

在编程中，参数是指传递给函数的变量。在上面的示例中，在函数调用期间传递两个变量 `n1` 和 `n2`。形参 `a` 和 `b` 接受函数定义中传递的实参。这些参数称为函数的形式参数。

### How to pass arguments to a function?

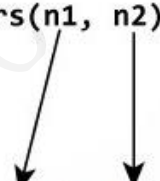
```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..

    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    ... ..
}
```



#### 将参数传递给函数

传递给函数的参数类型和形式参数必须匹配，否则编译器将引发错误。

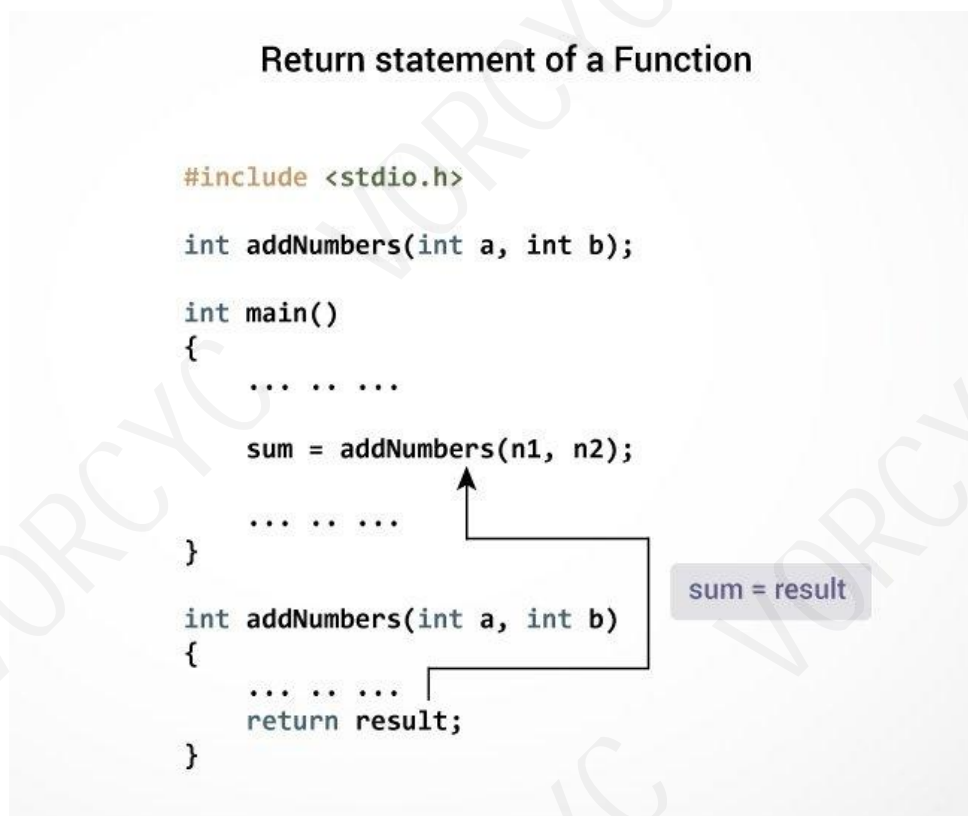
如果 `n1` 是 `char` 类型，那么 `a` 也应该是 `char` 类型。如果 `n2` 是浮点型，变量 `b` 也应该是浮点型。也可以在不传递参数的情况下调用函数。

## 返回语句

`return` 语句终止函数的执行，并向调用函数返回一个值。程序控制在返回语句之后被转移到调用函数。



在上面的例子中，`result` 变量的值被返回给 `main` 函数。`main()` 函数中的 `sum` 变量被赋予这个值。



返回函数语句

## return 语句的语法

```
return (表达式);
```

例如

```
return a;
return (a+b);
```

从函数返回的值类型与函数原型和函数定义中指定的返回类型必须匹配。

### 3.3 用户自定义函数的类型

下面的这 4 个程序检查用户输入的整数是否为质数。

下面所有这些程序的输出都是相同的，我们在每个示例中都创建了一个用户定义的函数。但是，我们在每个示例中采用的方法都不同。

#### 示例 1：未传递任何参数且未返回值

```
#include <stdio.h>

void checkPrimeNumber();

int main()
{
    checkPrimeNumber();    // 未传递实参
    return 0;
}

// 无返回类型代表不返回任何值
void checkPrimeNumber()
{
    int n, i, flag = 0;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // 0 和 1 不是质数
    if (n == 0 || n == 1)
        flag = 1;

    for(i = 2; i <= n/2; ++i)
    {
        if(n%i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);
}
```

`checkPrimenNumber()` 函数接受用户的输入，检查它是否是素数，并将其显示在屏幕上。

`checkPrimenNumber();` 中的空括号 `main()` 函数内部表明没有向函数传递参数。

函数的返回类型为 `void`。因此，该函数不返回任何值。

## 示例 2：未传递任何参数，但返回一个值

```
#include <stdio.h>
int getInteger();

int main()
{
    int n, i, flag = 0;

    // 无参数传递
    n = getInteger();

    // 0 和 1 不是质数
    if (n == 0 || n == 1)
        flag = 1;

    for(i = 2; i <= n/2; ++i)
    {
        if(n%i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);

    return 0;
}

// 返回用户输入的整数
int getInteger()
{
    int n;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    return n;
}
```

`n=getInteger();` 语句表示没有向函数传递参数。函数返回的值被赋给 `n`。

这里，`getInteger()` 函数从用户获取输入并返回。检查一个数字是否是素数的代码在 `main()` 函数中。

### 示例 3：参数已传递，但没有返回值

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);

int main()
{
    int n;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    //n 被传递给函数
    checkPrimeAndDisplay(n);

    return 0;
}

// 无返回类型代表不返回任何值
void checkPrimeAndDisplay(int n)
{
    int i, flag = 0;

    // 0 和 1 不是质数
    if (n == 0 || n == 1)
        flag = 1;

    for(i = 2; i <= n/2; ++i)
    {
        if(n%i == 0)
        {
            flag = 1;
            break;
        }
    }

    if(flag == 1)
        printf("%d is not a prime number.",n);
    else
        printf("%d is a prime number.", n);
}
```

用户输入的整数值被传递给 `checkPrimeAndDisplay()` 函数。

这里，`checkPrimeAndDisplay()` 函数检查传递的参数是否是质数，并显示相应的消息。

## 示例 4：传递参数并返回值

```
#include <stdio.h>
int checkPrimeNumber(int n);

int main()
{
    int n, flag;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n 被传递给 checkPrimeNumber() 函数
    // 并将函数的返回值被分配给 flag 变量
    flag = checkPrimeNumber(n);

    if(flag == 1)
        printf("%d is not a prime number",n);
    else
        printf("%d is a prime number",n);

    return 0;
}

// int 从函数中返回
int checkPrimeNumber(int n)
{
    // 0 和 1 不是质数
    if (n == 0 || n == 1)
        return 1;

    int i;

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
            return 1;
    }

    return 0;
}
```

来自用户的输入被传递给 `checkPrimeNumber()` 函数。

`checkPrimeNumber()` 函数检查传递的参数是否是素数。

如果传递的参数是质数，则函数返回 0。如果传递的参数是非素数，则该函数返回 1。返回值被赋给标志变量。

根据 `flag` 是 0 还是 1，将从 `main()` 函数打印相应的消息。

---

## 哪种方法更好？

这取决于你要解决的问题。在这种情况下，传递一个参数并从函数返回一个值(示例 4)会更好。

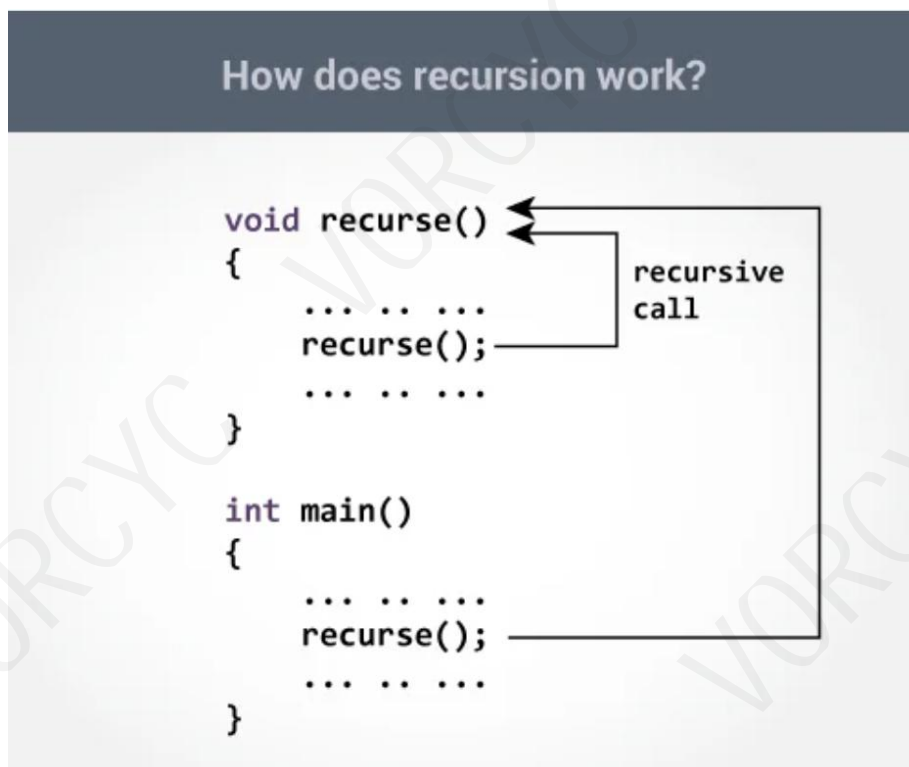
函数应该执行特定的任务。`checkPrimeNumber()`函数不接受用户的输入，也不显示适当的消息。它只检查一个数是否是素数。

## 3.4 递归

调用自身的函数称为递归函数。而且，这种技术被称为递归。

### 递归如何工作？

```
void recurse()  
{  
    ... ..  
    recurse();  
    ... ..  
}  
  
int main()  
{  
    ... ..  
    recurse();  
    ... ..  
}
```



递归工作

递归会继续，直到满足某些条件来阻止它。

为了防止无限递归，`if...else` 语句（或类似的方法）可以在一个分支进行递归调用而另一个分支不进行递归调用的情况下使用。

## 示例：使用递归的自然数之和

```
#include <stdio.h>
int sum(int n);

int main()
{
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);

    printf("sum = %d", result);
    return 0;
}

int sum(int n)
{
    if (n != 0)
        // sum() 调用自身
        return n + sum(n-1);
    else
        return n;
}
```

### 输出

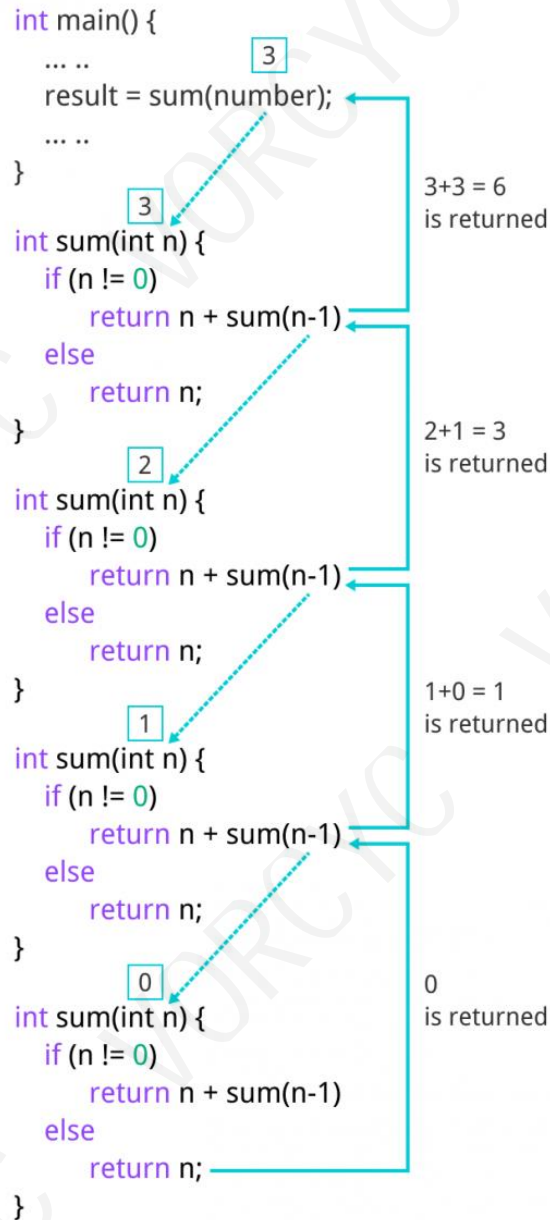
```
Enter a positive integer:3
sum = 6
```

最初，`sum()`从`main()`函数调用，`number`作为参数传递。

假设`sum()`中`n`的值最初为3。在下次函数调用期间，2被传递给`sum()`函数。这个过程一直持续到`n=0`。

当`n`等于0时，`if`条件失败，执行`else`部分，最终将整数和返回给`main()`函数。





自然数之和

## 递归的优缺点

递归使程序变得优雅。但是，如果性能至关重要，请改用循环，因为递归通常要慢得多。

话虽如此，递归是一个重要的概念。它经常用于数据结构和算法。例如，在树遍历等问题中使用递归是很常见的。

## 3.5 变量的存储方式

C 编程中的每个变量都有两个属性：类型和存储方式。

类型是指变量的数据类型。而且，存储方式决定了变量的作用范围、可见性和生存期。

有 4 种类型的存储方式：

1. 自动变量
2. 外部变量
3. 静态变量
4. 寄存器变量

---

### 局部变量

在块内声明的变量是自动变量或局部变量。局部变量仅存在于声明它的块中。

让我们举个例子。

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 5; ++i)
    {
        printf("C programming");
    }

    // 错误 : i 未申明
    printf("%d", i);
    return 0;
}
```

当你运行上面的程序时，你会得到一个错误的未声明标识符 `undeclared identifier i`。这是因为 `i` 是在 `for` 循环块中声明的。在块之外，它是未声明的。

让我们再举一个例子。

```
int main()
{
    int n1; // n1 对于 main() 函数是一个本地变量
```

```

}

void func()
{
    int n2; // n2 对于 func() 函数是一个本地变量
}

```

在上面的例子中，`n1` 对与 `main()` 而言是局部的，`n2` 对于 `func()` 而言是局部的。这意味着你不能访问 `func()` 中的 `n1` 变量，因为它只存在于 `main()` 中。同样，你不能访问 `main()` 中的 `n2` 变量，因为它只存在于 `func()` 中。

## 全局变量

在所有函数之外声明的变量称为外部变量或全局变量。它们可以从程序内的任何函数访问。

### 示例 1：全局变量

```

#include <stdio.h>
void display();

int n = 5; // 全局变量

int main()
{
    ++n;
    display();
    return 0;
}

void display()
{
    ++n;
    printf("n = %d", n);
}

```

输

```
n = 7
```

假设在 `file1` 中声明了一个全局变量。如果你试图在另一个文件 `file2` 中使用该变量，编译器将会报错。为了解决这个问题，在 `file2` 中使用关键字 `external` 来指示在另一个文件中声明外部变量。

---

## 寄存器变量

`register` 关键字用于声明寄存器变量。寄存器变量被认为比局部变量更快。

然而，现代编译器非常擅长代码优化，使用寄存器变量使程序更快的可能性很小。

除非你在嵌入式系统上工作，并且知道如何为给定的应用程序优化代码，否则不需要使用寄存器变量。

---

## 静态变量

静态变量是使用关键字 `static` 声明的。例如；

```
static int i;
```

静态变量的值将持续到程序结束。

---

### 示例 2：静态变量

```
#include <stdio.h>
void display();

int main()
{
    display();
    display();
}

void display()
{
    static int c = 1;
    c += 5;
    printf("%d ", c);
}
```

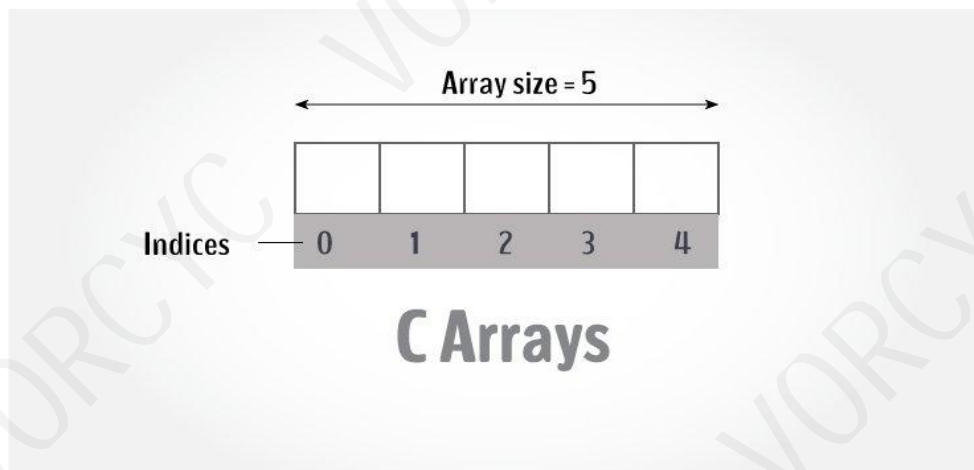
输出

```
6 11
```

在第一次函数调用期间，`c` 的值被初始化为 1。它的值增加 5。现在，`c` 的值是 6，它被打印在屏幕上。  
在第二次函数调用期间，`c` 不再初始化为 1。因为 `c` 是静态变量。值 `c` 增加 5。现在，它的值将是 11，它被打印在屏幕上。

## 4 数组

### 4.1 数组



### C 中的数组

数组是可以存储多个值的变量。例如，如果要存储 100 个整数，则可以为其创建一个数组。

```
int data[100];
```

### 如何声明数组？

```
数据类型 数组名[数组尺寸];
```

例如

```
float mark[5];
```

这里，我们声明了一个浮点类型的数组 `mark`。它的大小是 5。也就是说，它可以保存 5 个浮点值。重要的是要注意，一旦声明数组，就不能更改数组的大小和类型。

## 访问数组元素

你可以通过索引访问数组的元素。

假设如上所述声明了一个 `mark` 数组。第一个元素是 `mark[0]`，第二个元素是 `mark[1]`，以此类推。

<code>mark[0]</code>	<code>mark[1]</code>	<code>mark[2]</code>	<code>mark[3]</code>	<code>mark[4]</code>

声明数组

重点：

- 数组的第一个索引是 0，而不是 1。在这个例子中，`mark[0]` 是第一个元素。
- 如果数组的大小是 `n`，要访问最后一个元素，则使用 `n-1` 索引。在这个例子中，最后一个元素是 `mark[4]`。
- 假设 `mark[0]` 的起始地址为 2120d。那么，`mark[1]` 的地址为 2124d。同样，`mark[2]` 的地址为 2128d，以此类推。
- 这是因为 `float` 的大小是 4 字节。

## 如何初始化数组？

可以在声明期间初始化数组。例如

```
int mark[5] = {19, 10, 8, 17, 9};
```

你也可以像这样初始化数组。

```
int mark[] = {19, 10, 8, 17, 9};
```

在这里，我们没有指定大小。但是，编译器知道它的大小是 5，因为我们用 5 个元素初始化它。

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
19	10	8	17	9

初始化数组

## 更改数组元素的值

```
int mark[5] = {19, 10, 8, 17, 9}

// 让第三个元素的值为 -1
mark[2] = -1;

// 设置第五个元素的值为 0
mark[4] = 0;
```

## 输入和输出数组元素

下面介绍如何从用户那里获取输入并将其存储在数组元素中。

```
// 接受用户输入，并将其存储在第三个元素中
scanf("%d", &mark[2]);

// 接受用户输入，并将其存储在第 i 个元素中
scanf("%d", &mark[i-1]);
```

以下是打印数组的单个元素的方法。

```
// 打印数组的第一个元素
printf("%d", mark[0]);

// 打印数组的第三个元素
printf("%d", mark[2]);

// 打印数组的第 i 个元素
printf("%d", mark[i-1]);
```



## 示例 1：数组的输入/输出

// 本程序接受 5 个用户输入的值，然后将他们存储到数组中  
// 然后逐个打印数组中的元素

```
#include <stdio.h>

int main()
{
    int values[5];

    printf("Enter 5 integers: ");

    // 接受 5 次输入，并将它们存储到数组中
    for(int i = 0; i < 5; ++i)
    {
        scanf("%d", &values[i]);
    }

    printf("Displaying integers: ");

    // 逐个打印数组元素
    for(int i = 0; i < 5; ++i)
    {
        printf("%d\n", values[i]);
    }
    return 0;
}
```

输出

```
Enter 5 integers: 1
-3
34
0
3
Displaying integers: 1
-3
34
0
3
```

在这里，我们使用一个 **for** 循环从用户那里获取 5 个输入并将它们存储在一个数组中。然后，使用另一个 **for** 循环，这些元素显示在屏幕上。

## 示例 2：计算平均值

```
// 本程序使用数组来查找 N 个数的平均值
#include <stdio.h>

int main()
{
    int marks[10], i, n, sum = 0;
    double average;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    for(i=0; i < n; ++i)
    {
        printf("Enter number%d: ", i+1);
        scanf("%d", &marks[i]);

        // 把用户输入的整数累加到 sum 变量
        sum += marks[i];
    }

    // 显示地把变量 sum 转成 double 类型
    // 然后计算平均值
    average = (double) sum / n;

    printf("Average = %.21f", average);

    return 0;
}
```

### 输出

```
Enter number of elements: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39.60
```

在这里，我们计算了用户输入的数字 `n` 的平均值。

访问超出其界限的元素！

假设你声明了一个包含 10 个元素的数组。比方说，

```
int testArray[10];
```

你可以访问从 `testArray[0]` 到 `testArray[9]` 的数组元素。

现在假设你尝试访问 `testArray[12]`。该元素不可用。这可能会导致意外输出(未定义行为)。有时你可能会得到一个错误，而其他时候你的程序可能会正常运行。

因此，永远不要访问超出其边界的数组元素。

## 多维数组

在本教程中，你了解了数组。这些数组称为一维数组。

在下一个教程中，将了解多维数组（数组的数组）。

## 4.2 多维数组

在 C 语言中，你可以创建一个数组的数组。而这些数组则被称为多维数组。例如：

```
float x[3][4];
```

此时，`x` 是一个二维（2d）数组。该数组可以容纳 12 个元素。你可以将该数组视为具有 3 行，每行有 4 列的表格。

	Column 1	Column 2	Column 3	Column 4
Row 1	<code>x[0][0]</code>	<code>x[0][1]</code>	<code>x[0][2]</code>	<code>x[0][3]</code>
Row 2	<code>x[1][0]</code>	<code>x[1][1]</code>	<code>x[1][2]</code>	<code>x[1][3]</code>
Row 3	<code>x[2][0]</code>	<code>x[2][1]</code>	<code>x[2][2]</code>	<code>x[2][3]</code>

二维数组

同样，你可以声明一个三维（3d）数组。例如

```
float y[2][4][3];
```

在这里，数组 `y` 可以容纳 24 个元素。

---

### 初始化多维数组

以下是初始化二维和三维数组的方法：

## 二维数组的初始化

```
// 不同方式初始化二维数组

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};
```

## 三维数组的初始化

你可以采用与二维数组类似的方式初始化三维数组。下面是一个示例，

```
int test[2][3][4] = {
    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

### 示例 1：用于存储和打印值的二维数组

```
// 存储一周内两个城市的每天的气温，并显示出来
#include <stdio.h>

const int CITY = 2;
const int WEEK = 7;

int main()
{
    int temperature[CITY][WEEK];

    // 使用嵌套循环以将值存储到二维数组
    for (int i = 0; i < CITY; ++i)
    {
        for (int j = 0; j < WEEK; ++j)
        {
            printf("City %d, Day %d: ", i + 1, j + 1);
            scanf("%d", &temperature[i][j]);
        }
    }
    printf("\nDisplaying values: \n\n");
```

```
// 使用嵌套数组以显示二维数组的值
for (int i = 0; i < CITY; ++i)
{
    for (int j = 0; j < WEEK; ++j)
    {
        printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
    }
}
return 0;
}
```

## 输出

```
City 1, Day 1: 33
City 1, Day 2: 34
City 1, Day 3: 35
City 1, Day 4: 33
City 1, Day 5: 32
City 1, Day 6: 31
City 1, Day 7: 30
City 2, Day 1: 23
City 2, Day 2: 22
City 2, Day 3: 21
City 2, Day 4: 24
City 2, Day 5: 22
City 2, Day 6: 25
City 2, Day 7: 26
```

Displaying values:

```
City 1, Day 1 = 33
City 1, Day 2 = 34
City 1, Day 3 = 35
City 1, Day 4 = 33
City 1, Day 5 = 32
City 1, Day 6 = 31
City 1, Day 7 = 30
City 2, Day 1 = 23
City 2, Day 2 = 22
City 2, Day 3 = 21
City 2, Day 4 = 24
City 2, Day 5 = 22
City 2, Day 6 = 25
City 2, Day 7 = 26
```

## 示例 2：两个矩阵的总和

```
// 计算两个 2*2 矩阵的和

#include <stdio.h>
int main()
{
    float a[2][2], b[2][2], result[2][2];

    // 使用嵌套的 for 循环接受用户输入
    printf("Enter elements of 1st matrix\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
        {
            printf("Enter a%d%d: ", i + 1, j + 1);
            scanf("%f", &a[i][j]);
        }

    // 使用嵌套的 for 循环接受用户输入
    printf("Enter elements of 2nd matrix\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
        {
            printf("Enter b%d%d: ", i + 1, j + 1);
            scanf("%f", &b[i][j]);
        }

    // 逐个对两个数组的元素求和
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
        {
            result[i][j] = a[i][j] + b[i][j];
        }

    // 显示求和后的结果
    printf("\nSum Of Matrix:");

    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
        {
            printf("%.1f\t", result[i][j]);

            if (j == 1)
                printf("\n");
        }
    return 0;
}
```

输出

```
Enter elements of 1st matrix
Enter a11: 2;
```

```

Enter a12: 0.5;
Enter a21: -1.1;
Enter a22: 2;
Enter elements of 2nd matrix
Enter b11: 0.2;
Enter b12: 0;
Enter b21: 0.23;
Enter b22: 23;

```

```

Sum Of Matrix:
2.2      0.5
-0.9     25.0

```

### 示例 3：三维数组

```

// 存储和打印用户输入的 12 个值

#include <stdio.h>
int main()
{
    int test[2][3][2];

    printf("Enter 12 values: \n");

    for (int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for (int k = 0; k < 2; ++k)
            {
                scanf("%d", &test[i][j][k]);
            }
        }
    }

    // 用适当的索引打印值

    printf("\nDisplaying values:\n");
    for (int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for (int k = 0; k < 2; ++k)
            {
                printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
            }
        }
    }
}

```



```
}  
  
return 0;  
}
```

## 输出

```
Enter 12 values:  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
  
Displaying Values:  
test[0][0][0] = 1  
test[0][0][1] = 2  
test[0][1][0] = 3  
test[0][1][1] = 4  
test[0][2][0] = 5  
test[0][2][1] = 6  
test[1][0][0] = 7  
test[1][0][1] = 8  
test[1][1][0] = 9  
test[1][1][1] = 10  
test[1][2][0] = 11  
test[1][2][1] = 12
```

## 4.3 在 C 中将数组传递给函数

在 C 中，你可以将整个数组传递给函数。在我们了解这一点之前，让我们看看如何将数组的各个元素传递给函数。

### 传递单个数组元素

将数组元素传递给函数类似于将变量传递给函数。

#### 示例 1：传递单个数组元素

```
#include <stdio.h>

void display(int age1, int age2)
{
    printf("%d\n", age1);
    printf("%d\n", age2);
}

int main()
{
    int ageArray[] = {2, 8, 4, 12};

    // 传递数组的第二个和第三个元素到 display() 函数
    display(ageArray[1], ageArray[2]);
    return 0;
}
```

输出

```
8
4
```

这里，我们将数组参数传递给 `display()` 函数，与将变量传递给函数的方式相同。

```
// 传递数组的第二个和第三个元素到 display() 函数
display(ageArray[1], ageArray[2]);
```

我们可以在函数定义中看到这一点，其中函数参数是单个变量：

```
void display(int age1, int age2)
{
    // 代码
}
```

## 示例 2：将数组传递给函数

// 通过将一个数组传递给函数，利用函数求数组所有元素的和

```
#include <stdio.h>
float calculateSum(float num[]);

int main()
{
    float result, num[] = {23.4, 55, 22.6, 3, 40.5, 18};

    // num 数组被传递到 calculateSum() 函数
    result = calculateSum(num);
    printf("Result = %.2f", result);
    return 0;
}

float calculateSum(float num[])
{
    float sum = 0.0;

    for (int i = 0; i < 6; ++i)
    {
        sum += num[i];
    }

    return sum;
}
```

输出

```
Result = 162.50
```

若要将整个数组传递给函数，则仅将数组的名称作为参数传递。

```
result = calculateSum(num);
```

但是，请注意在函数定义中使用了[]。

```
float calculateSum(float num[]) {  
    ...  
}
```

这会通知编译器你正在将一维数组传递给函数。

---

## 将多维数组传递给函数

若要将多维数组传递给函数，则仅将数组的名称传递给函数（类似于一维数组）。

### 示例 3：传递二维数组

```
#include <stdio.h>  
void displayNumbers(int num[2][2]);  
  
int main()  
{  
    int num[2][2];  
    printf("Enter 4 numbers:\n");  
    for (int i = 0; i < 2; ++i)  
    {  
        for (int j = 0; j < 2; ++j)  
        {  
            scanf("%d", &num[i][j]);  
        }  
    }  
  
    // 传递一个多维数组到函数  
    displayNumbers(num);  
    return 0;  
}  
  
void displayNumbers(int num[2][2])  
{  
    printf("Displaying:\n");  
    for (int i = 0; i < 2; ++i)  
    {  
        for (int j = 0; j < 2; ++j)  
        {
```

```

        printf("%d\n", num[i][j]);
    }
}
}

```

输出

```

Enter 4 numbers:
2
3
4
5
Displaying:
2
3
4
5

```

注意函数原型和函数定义中的参数 `int num[2][2]`

```

// 函数原型
void displayNumbers(int num[2][2]);

```

这表示该函数将二维数组作为参数。我们还可以将二维以上的数组作为函数参数传递。  
传递二维数组时，不强制指定数组中的行数。但是，应始终指定列数。

例如：

```

void displayNumbers(int num[][2]) {
    // 代码
}

```

## 5 指针

### 5.1 指针

指针是 C 和 C++ 编程的强大功能。在学习指针之前，让我们先了解一下 C 中的地址。

#### C 语言中的地址

如果你的程序中有一个变量 `var`，`&var` 会给你它在内存中的地址。

在使用 `scanf()` 函数时，我们已经多次使用了地址。

```
scanf("%d", &var);
```

在这里，用户输入的值存储在 `var` 变量的地址中。让我们举一个实际的例子。

```
#include <stdio.h>
int main()
{
    int var = 5;
    printf("var: %d\n", var);

    // 请注意 var 之前使用了&
    printf("address of var: %p", &var);
    return 0;
}
```

输出

```
var: 5
address of var: 2686778
```

注：运行上述代码时，可能会得到不同的地址。

## 指针

指针（指针变量）是用于存储地址而不是值的特殊变量。

### 指针语法

以下是我们如何声明指针。

```
int* p;
```

这里，我们声明了一个 `int` 类型的指针 `p`。

你也可以用这些方式声明指针：

```
int *p1;  
int * p2;
```

让我们再举一个声明指针的例子：

```
int* p1, p2;
```

在这里，我们声明了一个指针 `p1` 和一个普通变量 `p2`。

---

## 将地址分配给指针

让我们举个例子。

```
int* pc, c;  
c = 5;  
pc = &c;
```

这里，5 被赋值给 `c` 变量。并且，`c` 的地址被分配给 `pc` 指针。

## 获取指针指向的事物的值

要获取指针所指对象的值，使用 `*` 操作符。例如：

```
int* pc, c;  
c = 5;  
pc = &c;  
printf("%d", *pc);    // 输出: 5
```

这里，`c` 的地址被分配给 `pc` 指针。为了获取存储在该地址中的值，我们使用了 `*pc`。

注：在上面的例子中，`pc` 是一个指针，而不是 `*pc`。你不能也不应该做类似 `*pc = &c;`；顺便说一下，`*` 被称为解引用操作符（当使用指针时）。它对指针进行操作，并给出存储在该指针中的值。

## 更改指针指向的值

让我们举个例子。

```
int* pc, c;  
c = 5;  
pc = &c;  
c = 1;  
printf("%d", c);      // 输出: 1  
printf("%d", *pc);    // 输出: 1
```

我们已经把 `c` 的地址赋给了 `pc` 指针。

然后，我们把 `c` 的值改为 1。因为 `pc` 和 `c` 的地址是一样的，所以 `*pc` 的结果是 1。

让我们再举一个例子。

```
int* pc, c;  
c = 5;  
pc = &c;  
*pc = 1;
```



```
printf("%d", *pc); // 输出 : 1
printf("%d", c);   // 输出 : 1
```

我们已经把 `c` 的地址赋给了 `pc` 指针。

然后，我们使用 `*pc = 1`; 将 `*pc` 改为 1。因为 `pc` 和 `c` 的地址是一样的，所以 `c` 等于 1。

让我们再举一个例子：

```
int* pc, c, d;
c = 5;
d = -15;

pc = &c; printf("%d", *pc); // 输出: 5
pc = &d; printf("%d", *pc); // 输出: -15
```

最初，使用 `pc = &c`; 将 `c` 的地址分配给 `pc` 指针。因为 `c` 等于 5，所以 `*pc` 等于 5。

然后，使用 `pc = &d`; 将 `d` 的地址分配给 `pc` 指针。因为 `d = -15`，所以 `*pc = -15`。

## 示例：指针的原理

让我们举一个例子：

```
#include <stdio.h>

int main()
{
    int* pc, c;

    c = 22;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c); // 22

    pc = &c;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc); // 22

    c = 11;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc); // 11

    *pc = 2;
```

```
printf("Address of c: %p\n", &c);
printf("Value of c: %d\n\n", c); // 2
return 0;
}
```

输出

```
Address of c: 2686784
Value of c: 22

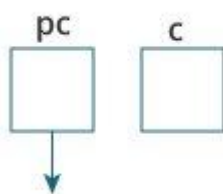
Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

## 程序说明

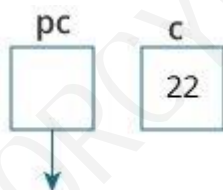
1. `int* pc, c;`



在这里，创建了一个指针 `pc` 和一个普通变量 `c`，它们都是 `int` 类型。

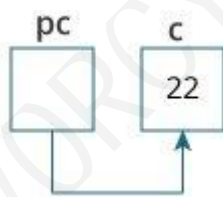
由于 `pc` 和 `c` 在初始化时没有初始化，因此指针 `pc` 要么指向没有地址，要么指向随机地址。变量 `c` 有一个地址，但是包含随机的垃圾值。

2. `c = 22;`



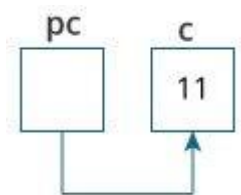
这将 22 赋值给变量 `c`，也就是说，22 存储在变量 `c` 的内存位置。

3. `pc = &c;`



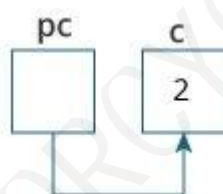
这将变量 `c` 的地址赋给指针 `pc`。

4. `c = 11;`



这个将 `11` 赋值给变量 `c`。

5. `*pc = 2;`



这将改变指针 `pc` 所指向的内存位置的值为 `2`。

## 使用指针时的常见错误

假设你想让指针 `pc` 指向 `c` 的地址，那么

```
int c, *pc;

// pc 是地址，但是 c 不是
pc = c; // 错误

// &c 是地址，但是 *pc 不是
*pc = &c; // 错误

// &c 和 pc 都是地址
pc = &c; // 正确

// c 和 *pc 是值
```

```
*pc = c; // 正确
```

下面是一个初学者经常感到困惑的指针语法示例。

```
#include <stdio.h>
int main()
{
    int c = 5;
    int *p = &c;

    printf("%d", *p); // 5
    return 0;
}
```

为什么我们在使用 `int*p = &c;` 时没有收到错误?

这是因为

```
int *p = &c;
```

相当于

```
int *p;
p = &c;
```

在这两种情况下，我们都创建了一个指针 `p` (不是 `*p`) 并将 `&c` 赋值给它。

为了避免这种混淆，我们可以使用这样的语句：

```
int* p = &c;
```

## 5.2 数组和指针之间的关系

在了解数组和指针之间的关系之前，请务必了解以下两部分知识：

- 数组
- 指针

---

### 数组和指针之间的关系

数组是一个连续的数据块。让我们编写一个程序来打印数组元素的地址。

```
#include <stdio.h>

int main()
{
    int x[4];
    int i;

    for(i = 0; i < 4; ++i)
    {
        printf("&x[%d] = %p\n", i, &x[i]);
    }

    printf("Address of array x: %p", x);

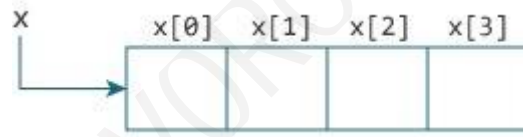
    return 0;
}
```

输出

```
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448
```

数组 `x` 的两个连续元素之间有 4 字节的差异，这是因为 `int` 的大小是 4 字节(对于当前我们的编译器而言)。

注意，`&x[0]`和`x`的地址是相同的。这是因为变量名`x`指向数组的第一个元素。



数组和指针之间的关系

从上面的例子中，很明显，`&x[0]`等于`x`，而`x[0]`等于`*x`。

同样的，

`&x[1]`等于`x+1`，`x[1]`等于`*(x+1)`。

`&x[2]`等于`x+2`，`x[2]`等于`*(x+2)`。

.....

基本上，`&x[i]`等于`x+i`，`x[i]`等于`*(x+i)`。

## 示例 1：指针和数组

```
#include <stdio.h>
int main()
{
    int i, x[6], sum = 0;

    printf("Enter 6 numbers: ");

    for(i = 0; i < 6; ++i)
    {
        // 等效于 scanf("%d", &x[i]);
        scanf("%d", x+i);

        // 等效于 sum += x[i]
        sum += *(x+i);
    }

    printf("Sum = %d", sum);

    return 0;
}
```

```
}
```

运行程序时，输出将为：

```
Enter 6 numbers: 2
3
4
4
12
4
Sum = 29
```

这里，我们声明了一个包含 6 个元素的数组 `x`。为了访问数组的元素，我们使用了指针。

在大多数情况下，数组名称会降格为指针。简单来说，数组名称被转换为指针。这就是为什么你可以使用指针来访问数组的元素的原因。但是，应该记住，**指针和数组是不一样的**。

## 示例 2：数组和指针

```
#include <stdio.h>
int main() {

    int x[5] = {1, 2, 3, 4, 5};
    int* ptr;

    // ptr 被分配为第三个元素的地址
    ptr = &x[2];

    printf("*ptr = %d \n", *ptr);    // 3
    printf("*(ptr+1) = %d \n", *(ptr+1)); // 4
    printf("*(ptr-1) = %d", *(ptr-1)); // 2

    return 0;
}
```

运行程序时，输出将为：

```
*ptr = 3  
*(ptr+1) = 4  
*(ptr-1) = 2
```

在本例中，第三个元素的地址`&x[2]`被赋值给 `ptr` 指针。因此，当我们输出 `*ptr` 时显示的是 `3`。输出 `*(ptr+1)` 得到第四个元素。类似地，输出 `*(ptr-1)` 给出第二个元素。



## 5.3 传递地址和指针

在 C 编程中，也可以将地址作为参数传递给函数。

为了在函数定义中接受这些地址，我们可以使用指针，这是因为指针用于存储地址。让我们举个例子：

### 示例 1：将地址传递给函数

```
#include <stdio.h>
void swap(int *n1, int *n2);

int main()
{
    int num1 = 5, num2 = 10;

    // num1 和 num2 的地址被传递
    swap( &num1, &num2);

    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
    return 0;
}

void swap(int* n1, int* n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

运行程序时，输出将为：

```
num1 = 10
num2 = 5
```

使用 `swap(&num1, &num2);` 将 `num1` 和 `num2` 的地址传递给 `swap()` 函数。

指针 `n1` 和 `n2` 在函数定义中接受这些参数。

```
void swap(int* n1, int* n2) {
    ... ..
}
```

```
}
```

当 `swap()` 函数中的 `*n1` 和 `*n2` 被改变时，`main()` 函数中的 `num1` 和 `num2` 也会被改变。

在 `swap()` 函数内部，`*n1` 和 `*n2` 交换。因此，也交换了 `num1` 和 `num2`。

注意，`swap()` 没有返回任何东西；，它的返回类型是 `void`。

## 示例 2：传递指向函数的指针

```
#include <stdio.h>

void addOne(int* ptr)
{
    (*ptr)++; // *ptr 加 1
}

int main()
{
    int* p, i = 10;
    p = &i;
    addOne(p);

    printf("%d", *p); // 11
    return 0;
}
```

这里，存储在 `p` 处的值 `*p` 最初是 10。

然后将指针 `p` 传递给 `addOne()` 函数。`ptr` 指针在 `addOne()` 函数中获取这个地址。

在函数内部，使用 `(*ptr)++`；将存储在 `ptr` 中的值增加 1。由于 `ptr` 和 `p` 指针都有相同的地址，所以 `main()` 中的 `*p` 也是 11。

## 5.4 动态内存分配

数组是固定数量值的集合。一旦声明了数组的大小，就无法更改它。

有时，声明的数组的大小可能不够。要解决这个问题，可以在运行时手动分配内存。这在 C 编程中称为动态内存分配。

要动态分配内存，可以使用库函数 `malloc()`、`calloc()`、`realloc()` 和 `free()`。这些函数在 `<stdlib.h>` 头文件中定义。

---

### `malloc()` 函数

名称 “malloc” 代表内存分配，是 Memory-Allocate 两个单词的合并。

`malloc()` 函数保留指定字节数的内存块。并且，它返回一个 `void` 指针，该指针可以被强制转换为任何形式的指针。

### `malloc()` 函数的语法

```
ptr = (castType*)malloc(size);
```

例：

```
ptr = (float*)malloc(100 * sizeof(float));
```

上面的语句分配 400 字节的内存。这是因为 `float` 的大小是 4 字节。并且，指针 `ptr` 保存分配的内存中第一个字节的地址。

如果不能分配内存，表达式将导致 NULL 指针。

---

## calloc() 函数

“calloc”这个名字代表连续分配，是 Continue-Allocate 两个单词的合并。

`malloc()` 函数分配内存并保留未初始化的内存，而 `calloc()` 函数分配内存并将所有位初始化为零。

## calloc() 函数的语法

```
ptr = (castType*)calloc(n, size);
```

例如：

```
ptr = (float*)calloc(25, sizeof(float));
```

上面的语句在内存中为 25 个 `float` 类型的元素分配连续空间。

## free() 函数

使用 `calloc()` 或 `malloc()` 创建的动态分配内存不会自行释放。必须显式地使用 `free()` 来释放空间。

```
free(ptr);
```

该语句释放在 `ptr` 所指向的内存中分配的空间。

## 示例 1: malloc() 函数 和 free() 函数

```
// 计算用户输入的 n 个数字之和的程序
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n, i, *ptr, sum = 0;
```

```

printf("Enter number of elements: ");
scanf("%d", &n);

ptr = (int*) malloc(n * sizeof(int));

// 如果无法分配内存
if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
}

printf("Enter elements: ");
for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
}

printf("Sum = %d", sum);

// 取消分配内存
free(ptr);

return 0;
}

```

输出

```

Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156

```

在这里，我们为 `n` 个 `int` 类型动态分配了内存。

## 示例 2: `calloc()` 函数和 `free()` 函数

```

// 计算用户输入的 n 个数字之和的程序

#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, i, *ptr, sum = 0;

```

```

printf("Enter number of elements: ");
scanf("%d", &n);

ptr = (int*) calloc(n, sizeof(int));
if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
}

printf("Enter elements: ");
for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
}

printf("Sum = %d", sum);
free(ptr);
return 0;
}

```

输出

```

Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156

```

## realloc() 函数

如果动态分配的内存不足或超过所需，你可以使用 `realloc()` 函数更改先前分配的内存大小。

### realloc() 函数的语法

```
ptr = realloc(ptr, x);
```

这里，`ptr` 被重新分配为新的大小 `x`。

### 示例 3: realloc() 函数

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

```
#include <stdlib.h>

int main()
{
    int *ptr, i, n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Addresses of previously allocated memory:\n");
    for(i = 0; i < n1; ++i)
        printf("%pc\n", ptr + i);

    printf("\nEnter the new size: ");
    scanf("%d", &n2);

    // 重新分配内存
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Addresses of newly allocated memory:\n");
    for(i = 0; i < n2; ++i)
        printf("%pc\n", ptr + i);

    free(ptr);

    return 0;
}
```

## 输出

```
Enter size: 2
Addresses of previously allocated memory:
26855472
26855476

Enter the new size: 4
Addresses of newly allocated memory:
26855472
26855476
26855480
26855484
```

## 5.5 数组和指针示例

若要了解本文中的所有程序，应有下面的知识储备：

1. 数组
  2. 多维数组
  3. 指针
  4. 数组和指针的关系
  5. 按引用调用
  6. 动态内存分配
- 

### 数组和指针示例

#### 1. 计算数组元素的平均值

若要理解此示例，应了解以下 C 的知识：

- while 和 do...while 循环
- for 循环
- 数组

#### 示例：存储数字并使用数组计算平均值

```
#include <stdio.h>
int main()
{
    int n;
    double arr[100];
    printf("Enter the number of elements (1 to 100): ");
    scanf("%d", &n);

    for (int i = 0; i < n; ++i) {
        printf("Enter number%d: ", i + 1);
        scanf("%lf", &arr[i]);
    }

    // 将最大数字存储到 arr[0]
    for (int i = 1; i < n; ++i) {
        if (arr[0] < arr[i]) {
            arr[0] = arr[i];
        }
    }
}
```



```

}

printf("Largest element = %.2lf", arr[0]);

return 0;
}

```

输出

```

Enter the numbers of elements: 6
1. Enter number: 45.3
2. Enter number: 67.5
3. Enter number: -45.6
4. Enter number: 20.34
5. Enter number: 33
6. Enter number: 45.6
Average = 27.69

```

在这里，首先要求用户输入元素的数量。这个数被赋给 `n`。

如果用户输入的整数大于 1 或大于 100，则要求用户再次输入该数字。这是使用 `while` 循环完成的。然后，从 `i = 0` 迭代一个 `for` 循环到 `i`。在循环的每次迭代中，用户被要求输入数字来计算平均值。这些数字存储在 `num[]` 数组中。

```
scanf("%f", &num[i]);
```

并且，计算每个输入元素的总和。

```
sum += num[i];
```

一旦 `for` 循环完成，计算平均值并打印在屏幕上。

## 2. 求数组的最大元素

若要理解此示例，应了解以下 C 语言编程知识：

- `for` 循环
- 数组

### 示例：数组中最大的元素

```
#include <stdio.h>

int main()
{
    int n;
    double arr[100];
    printf("Enter the number of elements (1 to 100): ");
    scanf("%d", &n);

    for (int i = 0; i < n; ++i) {
        printf("Enter number%d: ", i + 1);
        scanf("%lf", &arr[i]);
    }

    // 将最大数字存储到 arr[0]
    for (int i = 1; i < n; ++i) {
        if (arr[0] < arr[i]) {
            arr[0] = arr[i];
        }
    }

    printf("Largest element = %.2lf", arr[0]);

    return 0;
}
```

输出

```
Enter the number of elements (1 to 100): 5
Enter number1: 34.5
Enter number2: 2.4
Enter number3: -35.5
Enter number4: 38.7
Enter number5: 24.5
Largest element = 38.70
```

该程序从用户处获取 `n` 个元素，并将其存储在 `arr` 数组中。

为了找到最大的元素，

- 检查数组的前两个元素，并将这两个元素中最大的元素放入 `arr[0]` 中。
- 检查第一个和第三个元素，这两个元素中最大的一个放在 `arr[0]` 中。
- 这个过程一直持续到第一个和最后一个元素被检查
- 最大的数字将存储在 `arr[0]` 位置

```
// 存储 arr[0] 处的最大数字
for (int i = 1; i < n; ++i) {
    if (arr[0] < arr[i]) {
        arr[0] = arr[i];
    }
}
```

```
}
```

### 3. 计算标准差

若要理解此示例，应了解以下 C 语言的知识：

- 数组
- 在 C 中将数组传递给函数

该程序使用数组计算单个序列的标准差。

为了计算标准差，我们创建了一个名为 `calculateSD()` 的函数。

#### 示例：总体标准差

```
// 一个群体的标准差
#include <math.h>
#include <stdio.h>

float calculateSD(float data[]);

int main()
{
    int i;
    float data[10];
    printf("Enter 10 elements: ");
    for (i = 0; i < 10; ++i)
        scanf("%f", &data[i]);
    printf("\nStandard Deviation = %.6f", calculateSD(data));
    return 0;
}

float calculateSD(float data[])
{
    float sum = 0.0, mean, SD = 0.0;
    int i;
    for (i = 0; i < 10; ++i) {
        sum += data[i];
    }
    mean = sum / 10;
    for (i = 0; i < 10; ++i)
    {
        SD += pow(data[i] - mean, 2);
    }
    return sqrt(SD / 10);
}
```

输出

```
Enter 10 elements: 1
2
3
4
5
6
7
8
9
10

Standard Deviation = 2.872281
```

这里，包含 10 个元素的数组被传递给 `calculateSD()` 函数。该函数首先计算了表示平均值的变量 `mean`，然后再利用 `mean` 计算标准差并返回最终结果。

注：该程序计算总体的标准差。如果需要查找样本的标准差，则公式略有不同。

## 4. 使用多维数组计算两个矩阵之和

若要理解此示例，应了解以下 C 语言的编程知识：

- 数组
- 多维数组

### 示例：添加两个矩阵的程序

```
#include <stdio.h>
int main() {
    int r, c, a[100][100], b[100][100], sum[100][100], i, j;
    printf("Enter the number of rows (between 1 and 100): ");
    scanf("%d", &r);
    printf("Enter the number of columns (between 1 and 100): ");
    scanf("%d", &c);

    printf("\nEnter elements of 1st matrix:\n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("Enter element a%d%d: ", i + 1, j + 1);
            scanf("%d", &a[i][j]);
        }
}
```

```

printf("Enter elements of 2nd matrix:\n");
for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
        printf("Enter element b%d%d: ", i + 1, j + 1);
        scanf("%d", &b[i][j]);
    }

// 将两个矩阵相加
for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
        sum[i][j] = a[i][j] + b[i][j];
    }

// 打印结果
printf("\nSum of two matrices: \n");
for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
        printf("%d ", sum[i][j]);
        if (j == c - 1) {
            printf("\n\n");
        }
    }

return 0;
}

```

输出

```

Enter the number of rows (between 1 and 100): 2
Enter the number of columns (between 1 and 100): 3

Enter elements of 1st matrix:
Enter element a11: 2
Enter element a12: 3
Enter element a13: 4
Enter element a21: 5
Enter element a22: 2
Enter element a23: 3
Enter elements of 2nd matrix:
Enter element b11: -4
Enter element b12: 5
Enter element b13: 3
Enter element b21: 5
Enter element b22: 6
Enter element b23: 3

Sum of two matrices:
-2  8  7

10  8  6

```

在这个程序中，用户被要求输入行数 **r** 和列数 **c**，然后，用户还需要输入两个矩阵(顺序为 **rx****c**)的元素。

然后将两个矩阵的对应元素相加，保存到另一个矩阵(二维数组)中。最后，结果被打印在屏幕上。

## 5. 将两个矩阵相乘

若要理解此示例，应了解以下 C 语言编程知识：

- 数组
- 多维数组

该程序要求用户输入两个矩阵的大小（行和列）。

要将两个矩阵相乘，第一个矩阵的列数应等于第二个矩阵的行数。

下面的程序要求提供两个矩阵的行数和列数，直到满足上述条件。

然后，执行两个矩阵的乘法，结果显示在屏幕上。

为此，我们创建了三个函数：

- `getMatrixElements()`– 从用户那里获取矩阵元素输入。
- `multiplyMatrices()`– 将两个矩阵相乘。
- `display()`– 显示乘法后的结果矩阵。

### 示例：通过将矩阵传递给函数来乘以矩阵

```
#include <stdio.h>

// 函数来获取用户输入的矩阵元素
void getMatrixElements(int matrix[][10], int row, int column) {

    printf("\nEnter elements: \n");

    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < column; ++j) {
            printf("Enter a%d%d: ", i + 1, j + 1);
            scanf("%d", &matrix[i][j]);
        }
    }
}
```

```

// 两个矩阵相乘的函数
void multiplyMatrices(int first[][10],
                     int second[][10],
                     int result[][10],
                     int r1, int c1, int r2, int c2) {

    // 正在将矩阵 mult 的元素初始化为 0。
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            result[i][j] = 0;
        }
    }

    // 将第一和第二矩阵相乘并存储在结果中
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            for (int k = 0; k < c1; ++k) {
                result[i][j] += first[i][k] * second[k][j];
            }
        }
    }
}

// 显示矩阵的函数
void display(int result[][10], int row, int column) {

    printf("\nOutput Matrix:\n");
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < column; ++j) {
            printf("%d ", result[i][j]);
            if (j == column - 1)
                printf("\n");
        }
    }
}

int main() {
    int first[10][10], second[10][10], result[10][10], r1, c1, r2, c2;
    printf("Enter rows and column for the first matrix: ");
    scanf("%d %d", &r1, &c1);
    printf("Enter rows and column for the second matrix: ");
    scanf("%d %d", &r2, &c2);

    // 输入直到
    // 第一个矩阵列不等于第二个矩阵行
    while (c1 != r2) {
        printf("Error! Enter rows and columns again.\n");
        printf("Enter rows and columns for the first matrix: ");
        scanf("%d %d", &r1, &c1);
        printf("Enter rows and columns for the second matrix: ");
        scanf("%d %d", &r2, &c2);
    }

    // 获取第一个矩阵的元素
    getMatrixElements(first, r1, c1);

    // 获取第二个矩阵的元素

```

```
getMatrixElements(second, r2, c2);

// 将两个矩阵相乘。
multiplyMatrices(first, second, result, r1, c1, r2, c2);

// 显示结果
display(result, r1, c2);

return 0;
}
```

输出

```
Enter rows and column for the first matrix: 2
3
Enter rows and column for the second matrix: 3
2

Enter elements:
Enter a11: 2
Enter a12: -3
Enter a13: 4
Enter a21: 53
Enter a22: 3
Enter a23: 5

Enter elements:
Enter a11: 3
Enter a12: 3
Enter a21: 5
Enter a22: 0
Enter a31: -3
Enter a32: 4

Output Matrix:
-21  22
159 179
```

## 6. 计算矩阵的转置

若要理解此示例，应了解以下 C 语言编程知识：

- 数组
- 多维数组



矩阵的转置是通过交换行和列得到的新矩阵。

在这个程序中，要求用户输入行数 **r** 和列数 **c**。在这个程序中，它们的值应该小于 10。

然后，要求用户输入矩阵的元素(阶为 **r\*c**)。

然后，下面的程序计算矩阵的转置并将其打印在屏幕上。

## 示例：查找矩阵转置的程序

```
#include <stdio.h>
int main()
{
    int a[10][10], transpose[10][10], r, c;
    printf("Enter rows and columns: ");
    scanf("%d %d", &r, &c);

    // 为矩阵分配元素
    printf("\nEnter matrix elements:\n");
    for (int i = 0; i < r; ++i)
        for (int j = 0; j < c; ++j)
        {
            printf("Enter element a%d%d: ", i + 1, j + 1);
            scanf("%d", &a[i][j]);
        }

    // 打印矩阵 a[][]
    printf("\nEnter matrix: \n");
    for (int i = 0; i < r; ++i)
        for (int j = 0; j < c; ++j)
        {
            printf("%d ", a[i][j]);
            if (j == c - 1)
                printf("\n");
        }

    // 计算转置
    for (int i = 0; i < r; ++i)
        for (int j = 0; j < c; ++j) {
            transpose[j][i] = a[i][j];
        }

    // 打印转置
    printf("\nTranspose of the matrix:\n");
    for (int i = 0; i < c; ++i)
        for (int j = 0; j < r; ++j) {
            printf("%d ", transpose[i][j]);
            if (j == r - 1)
                printf("\n");
        }
    return 0;
}
```

```
}
```

输出

```
Enter rows and columns: 2
3
```

```
Enter matrix elements:
```

```
Enter element a11: 1
```

```
Enter element a12: 4
```

```
Enter element a13: 0
```

```
Enter element a21: -5
```

```
Enter element a22: 2
```

```
Enter element a23: 7
```

```
Entered matrix:
```

```
1 4 0
```

```
-5 2
```

```
Transpose of the matrix:
```

```
1 -5
```

```
4 2
```

```
0 7
```

## 7. 将两个矩阵相乘

若要理解此示例，应了解以下 C 语言编程知识：

- 数组
- 多维数组
- 数组传递给函数

该程序要求用户输入矩阵的大小（行和列）。

然后，它要求用户输入这些矩阵的元素，最后添加并显示结果。

为了执行此任务，需要执行三个功能：

1. 从用户那里获取矩阵元素 `enterData()`
2. 将两个矩阵相乘 `multiplyMatrices()`
3. 显示乘法后的结果矩阵 `display()`

**示例：通过将矩阵传递给函数来乘以矩阵**

```

#include <stdio.h>

void enterData(int firstMatrix[][10], int secondMatrix[][10], int rowFirst, int columnFirst,
int rowSecond, int columnSecond);

void multiplyMatrices(int firstMatrix[][10], int secondMatrix[][10], int multResult[][10], int
rowFirst, int columnFirst, int rowSecond, int columnSecond);

void display(int mult[][10], int rowFirst, int columnSecond);

int main()
{
    int firstMatrix[10][10], secondMatrix[10][10], mult[10][10], rowFirst, columnFirst,
rowSecond, columnSecond, i, j, k;

    printf("Enter rows and column for first matrix: ");
    scanf("%d %d", &rowFirst, &columnFirst);

    printf("Enter rows and column for second matrix: ");
    scanf("%d %d", &rowSecond, &columnSecond);

    // 如果第一个矩阵的列不等于第二个矩阵的行, 请用户重新输入矩阵的大小。
    while (columnFirst != rowSecond)
    {
        printf("Error! column of first matrix not equal to row of second.\n");
        printf("Enter rows and column for first matrix: ");
        scanf("%d%d", &rowFirst, &columnFirst);
        printf("Enter rows and column for second matrix: ");
        scanf("%d%d", &rowSecond, &columnSecond);
    }

    // 获取矩阵数据的函数
    enterData(firstMatrix, secondMatrix, rowFirst, columnFirst, rowSecond, columnSecond);

    // 两个矩阵相乘的函数
    multiplyMatrices(firstMatrix, secondMatrix, mult, rowFirst, columnFirst, rowSecond,
columnSecond);

    // 函数显示相乘后的结果矩阵
    display(mult, rowFirst, columnSecond);

    return 0;
}

void enterData(int firstMatrix[][10], int secondMatrix[][10], int rowFirst, int columnFirst,
int rowSecond, int columnSecond)
{
    int i, j;
    printf("\nEnter elements of matrix 1:\n");
    for(i = 0; i < rowFirst; ++i)
    {
        for(j = 0; j < columnFirst; ++j)
        {
            printf("Enter elements a%d%d: ", i + 1, j + 1);
            scanf("%d", &firstMatrix[i][j]);
        }
    }
}

```

```

printf("\nEnter elements of matrix 2:\n");
for(i = 0; i < rowSecond; ++i)
{
    for(j = 0; j < columnSecond; ++j)
    {
        printf("Enter elements b%d%d: ", i + 1, j + 1);
        scanf("%d", &secondMatrix[i][j]);
    }
}

void multiplyMatrices(int firstMatrix[][10], int secondMatrix[][10], int mult[][10], int
rowFirst, int columnFirst, int rowSecond, int columnSecond)
{
    int i, j, k;

    // 正在将矩阵 mult 的元素初始化为 0
    for(i = 0; i < rowFirst; ++i)
    {
        for(j = 0; j < columnSecond; ++j)
        {
            mult[i][j] = 0;
        }
    }

    // 将矩阵 firstMatrix 和 secondMatrix 相乘并存储在多个数组中
    for(i = 0; i < rowFirst; ++i)
    {
        for(j = 0; j < columnSecond; ++j)
        {
            for(k=0; k<columnFirst; ++k)
            {
                mult[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
            }
        }
    }
}

void display(int mult[][10], int rowFirst, int columnSecond)
{
    int i, j;
    printf("\nOutput Matrix:\n");
    for(i = 0; i < rowFirst; ++i)
    {
        for(j = 0; j < columnSecond; ++j)
        {
            printf("%d ", mult[i][j]);
            if(j == columnSecond - 1)
                printf("\n\n");
        }
    }
}

```

输出

```
Enter rows and column for first matrix: 3
2
Enter rows and column for second matrix: 3
2
Error! column of first matrix not equal to row of second.

Enter rows and column for first matrix: 2
3
Enter rows and column for second matrix: 3
2

Enter elements of matrix 1:
Enter elements a11: 3
Enter elements a12: -2
Enter elements a13: 5
Enter elements a21: 3
Enter elements a22: 0
Enter elements a23: 4

Enter elements of matrix 2:
Enter elements b11: 2
Enter elements b12: 3
Enter elements b21: -9
Enter elements b22: 0
Enter elements b31: 0
Enter elements b32: 4

Output Matrix:
24 29

6 25
```

---

## 8. 使用指针访问数组的元素

若要理解此示例，应了解以下 C 语言编程知识：

- for 循环
- 数组
- 指针
- 数组和指针之间的关系

### 示例：使用指针访问数组元素

```
#include <stdio.h>

int main()
{
```

```

int data[5];

printf("Enter elements: ");
for (int i = 0; i < 5; ++i)
    scanf("%d", data + i);

printf("You entered: \n");
for (int i = 0; i < 5; ++i)
    printf("%d\n", *(data + i));
return 0;
}

```

输出

```

Enter elements: 1
2
3
5
4
You entered:
1
2
3
5
4

```

在这个程序中，元素存储在整数数组 `data[]` 中。

然后，使用指针表示法访问数组的元素。顺便说一下，

- `Data[0]` 相当于 `* Data`，`&data[0]` 相当于 `Data`
- `Data[1]` 相当于 `*(Data + 1)`，`&data[1]` 相当于 `Data + 1`
- `Data[2]` 相当于 `*(Data + 2)`，`&data[2]` 相当于 `Data + 2`
- . . .
- `Data [i]` 相当于 `*(Data + i)`，`&data[i]` 相当于 `Data + i`

## 9. 使用按引用调用并按循环顺序交换数字

若要理解此示例，应了解以下 C 语言编程知识：

- 指针
- 传递地址和指针

## 示例：使用按引用调用交换元素的程序

```
#include <stdio.h>
void cyclicSwap(int *a, int *b, int *c);
int main() {
    int a, b, c;

    printf("Enter a, b and c respectively: ");
    scanf("%d %d %d", &a, &b, &c);

    printf("Value before swapping:\n");
    printf("a = %d \nb = %d \nc = %d\n", a, b, c);

    cyclicSwap(&a, &b, &c);

    printf("Value after swapping:\n");
    printf("a = %d \nb = %d \nc = %d", a, b, c);

    return 0;
}

void cyclicSwap(int *n1, int *n2, int *n3) {
    int temp;
    // 按循环顺序交换
    temp = *n2;
    *n2 = *n1;
    *n1 = *n3;
    *n3 = temp;
}
```

### 输出

```
Enter a, b and c respectively: 1
2
3
Value before swapping:
a = 1
b = 2
c = 3
Value after swapping:
a = 3
b = 1
c = 2
```

在这里，用户输入的三个数字分别存储在变量 **a**、**b** 和 **c** 中。这些数字的地址被传递给 `cyclicSwap()` 函数。

```
cyclicSwap(&a, &b, &c);
```

在 `cyclicSwap()` 的函数定义中，我们将这些地址赋给了指针。

```
cyclicSwap(int *n1, int *n2, int *n3) {
...
}
```

当 `cyclicSwap()` 中的 `n1`、`n2` 和 `n3` 被改变时，`main()` 中的 `a`、`b` 和 `c` 的值也会被改变。

注意：`cyclicSwap()` 函数不返回任何东西。

## 10. 查找最大数字（使用动态内存分配）

若要理解此示例，应了解以下 C 的知识：

- 指针
- 动态内存分配
- for 循环

### 示例：使用动态内存分配查找最大数字

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

int main() {

    int n;
    double *data;
    printf("Enter the total number of elements: ");
    scanf("%d", &n);

    // 分配 n 个元素的内存
    data = (double *)calloc(n, sizeof(double));
    if (data == NULL) {
        printf("Error!!! memory not allocated.");
        exit(0);
    }

    // 存储用户输入的数字
    for (int i = 0; i < n; ++i) {
```



```

    printf("Enter number%d: ", i + 1);
    scanf("%lf", data + i);
}

// 找到最大数
for (int i = 1; i < n; ++i) {
    if (*data < *(data + i)) {
        *data = *(data + i);
    }
}
printf("Largest number = %.2lf", *data);

free(data);

return 0;
}

```

输出

```

Enter the total number of elements: 5
Enter number1: 3.4
Enter number2: 2.4
Enter number3: -5
Enter number4: 24.2
Enter number5: 6.7
Largest number = 24.20

```

## 解释

在程序中，我们要求用户输入存储在变量 `n` 中的元素总数。然后，我们为 `n` 个 `double` 值分配了内存。

```

// 为 n 个 double 值分配内存
data = (double *)calloc(n, sizeof(double));

```

然后，我们使用 `for` 循环从用户那里获取 `n` 个数据。

```

// 存储元素
for (int i = 0; i < n; ++i) {
    printf("Enter Number%d: ", i + 1);
    scanf("%lf", data + i);
}

```

最后，我们使用另一个 `for` 循环来计算最大的数。

```
// 计算最大数
for (int i = 1; i < n; ++i) {
    if (*data < *(data + i))
        *data = *(data + i);
}
```

**注:**除了 `calloc()` 之外，也可以使用 `malloc()` 函数来解决此问题。

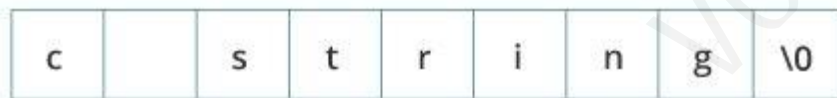
## 6 字符串

### 6.1 字符串

在 C 中，字符串是以空字符 `\0` 结尾的字符序列。例如：

```
char c[] = "c string";
```

当编译器遇到包含在双引号中的字符序列时，默认情况下，它会在末尾追加一个空字符 `\0`。

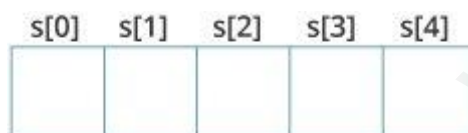


内存图

### 如何声明字符串？

以下是声明字符串的方法：

```
char s[5];
```



C 中的字符串声明

在这里，我们声明了一个包含 5 个字符的字符串。

## 如何初始化字符串？

你可以通过多种方式初始化字符串。

```
char c[] = "abcd";  
char c[50] = "abcd";  
char c[] = {'a', 'b', 'c', 'd', '\\0'};  
char c[5] = {'a', 'b', 'c', 'd', '\\0'};
```

c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\\0

C 中的字符串初始化

让我们再举一个例子：

```
char c[5] = "abcde";
```

在这里，我们试图将 6 个字符（最后一个字符是 ‘\\0’）分配给具有 5 个字符的 `char` 数组。这不正确！你不应该这样做。

## 为字符串赋值

数组和字符串在 C 语言中是二等公民；一旦声明赋值运算符，它们就不再支持分配。例如

```
char c[100];  
c = "C programming"; // 错误！数组类型不可分配。
```

注：请改用 `strcpy()` 函数复制字符串。

## 从用户那里读取字符串

可以使用 `scanf()` 函数读取字符串。

`scanf()` 函数读取字符序列，直到遇到空格（空格、换行符、制表符等）。

### 示例 1: `scanf()` 读取字符串

```
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

输出

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

即使在上面的程序中输入了 `Dennis Ritchie`，名称字符串中只存储了“`Dennis`”。因为丹尼斯后面有个空格。

还请注意，我们在 `scanf()` 中使用了 `name` 而不是 `&name`。

```
scanf("%s", name);
```

这是因为 `name` 是一个 `char` 数组，而我们知道在 C 语言中数组名会衰变成指针。

因此，`scanf()` 中的名称已经指向字符串中第一个元素的地址，这就是我们不需要使用 `&` 的原因。

## 如何读取一行文本？

你可以使用 `fgets()` 函数读取一行字符串。并且，你可以使用 `puts()` 来显示字符串。

### 示例 2: `fgets()` 和 `puts()`

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin); // 读字符串
    printf("Name: ");
    puts(name); // 显示字符串
    return 0;
}
```

输出

```
Enter name: Tom Hanks
Name: Tom Hanks
```

在这里，我们使用 `fgets()` 函数从用户读取字符串。

`fgets (name, sizeof(name), stdin);` // 读取字符串

`sizeof(name)` 的结果为 30。因此，我们最多可以接受 30 个字符作为输入，这是 `name` 字符串的大小。为了打印字符串，我们使用了 `puts(name);`

注: `gets()` 函数也可以从用户获取输入。但是，它从 C 标准中删除了。  
这是因为 `gets()` 允许你输入任意长度的字符。因此，可能会出现缓冲区溢出。

---

## 将字符串传递给函数

字符串可以以与数组类似的方式传递给函数。

---

### 示例 3：将字符串传递给函数

```
#include <stdio.h>
void displayString(char str[]);

int main()
{
    char str[50];
    printf("Enter string: ");
    fgets(str, sizeof(str), stdin);
    displayString(str);    // 将字符串传递给函数
    return 0;
}

void displayString(char str[])
{
    printf("String Output: ");
    puts(str);
}
```

---

## 字符串和指针

与数组类似，字符串名称对指针是“降格”的。因此，你可以使用指针来操作字符串的元素。

---

## 示例 4：字符串和指针

```
#include <stdio.h>

int main(void) {
    char name[] = "Harry Potter";

    printf("%c", *name);    // 输出: H
    printf("%c", *(name+1)); // 输出: a
    printf("%c", *(name+7)); // 输出: o

    char *namePtr;

    namePtr = name;
    printf("%c", *namePtr);  // 输出: H
    printf("%c", *(namePtr+1)); // 输出: a
    printf("%c", *(namePtr+7)); // 输出: o
}
```

## 常用的字符串函数

- `Strle()` - 计算字符串的长度
- `strcpy()` - 将一个字符串复制到另一个字符串
- `strcmp()` - 比较两个字符串
- `strcat()` - 连接两个字符串



## 6.2 使用库函数进行字符串操作

你需要经常根据问题的需要操作字符串。大多数时候字符串操作都可以手动完成，但是，这使得编程变得复杂而庞大。

为了解决这个问题，C 语言在标准库“`string.h`”中支持大量字符串处理函数。

下面讨论几个常用的字符串处理函数：

函数	功能
<code>strlen()</code>	计算字符串的长度
<code>strcpy()</code>	将一个字符串复制到另一个字符串
<code>strcat()</code>	连接（连接）两个字符串
<code>strcmp()</code>	比较两个字符串
<code>strlwr()</code>	将字符串转换为小写
<code>strupr()</code>	将字符串转换为大写

字符串处理函数在“`string.h`”头文件下定义。

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

**注意：**必须包含以上代码才能运行字符串处理函数。

---

### `gets()` 和 `puts()`

函数 `gets()` 和 `puts()` 是两个字符串函数，用于从用户那里获取字符串输入并分别显示它，如上一章所述。

```
#include<stdio.h>

int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name);    //Function to read string from user.
    printf("Name: ");
    puts(name);    //Function to display string.
    return 0;
}
```

注意：虽然 `gets()` 和 `puts()` 函数处理字符串，但这两个函数都在“`stdio.h`”头文件中定义。

## 6.3 字符串示例

字符串是以空字符`\0`结尾的字符数组。

页面中提到的所有示例都与 C 中的字符串有关。要了解本页上的所有示例，你应该了解：

- C 中的字符串
  - 如何将字符串传递给函数
  - 用于处理字符串的常用库函数
- 

### 字符串示例

#### 1. 查找字符串中字符的频率

若要理解此示例，应了解以下 C 的编程知识：

- 数组
- 字符串

#### 示例：查找字符的频率

```
#include <stdio.h>
int main()
{
    char str[1000], ch;
    int count = 0;

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    printf("Enter a character to find its frequency: ");
    scanf("%c", &ch);

    for (int i = 0; str[i] != '\0'; ++i) {
        if (ch == str[i])
            ++count;
    }

    printf("Frequency of %c = %d", ch, count);
    return 0;
}
```

输出

```
Enter a string: This website is awesome.
Enter a character to find its frequency: e
```

Frequency of e = 4

在这个程序中，用户输入的字符串存储在 `str` 中。

然后，要求用户输入要查找的频率的字符。它存储在变量 `ch` 中。

然后，使用 `for` 循环遍历字符串中的字符。在每次迭代中，如果字符串中的字符等于 `ch`，则计数增加 1。

最后，打印存储在 `count` 变量中的频率。

注：该程序区分大小写，即它将同一字母表的大写和小写版本视为不同的字符。

## 2. 查找元音、辅音、数字和空格的数量

若要理解此示例，应了解以下 C 的知识：

- 数组
- 字符串

### 示例：计算元音、辅音等的程序

```
#include <stdio.h>

int main() {

    char line[150];
    int vowels, consonant, digit, space;

    // 将所有变量初始化为 0
    vowels = consonant = digit = space = 0;

    // 获取整行字符串输入
    printf("Enter a line of string: ");
    fgets(line, sizeof(line), stdin);
```

```

// 循环遍历字符串的每个字符
for (int i = 0; line[i] != '\0'; ++i) {

    // 将字符转换为小写
    line[i] = tolower(line[i]);

    // 检查字符是否为元音
    if (line[i] == 'a' || line[i] == 'e' || line[i] == 'i' ||
        line[i] == 'o' || line[i] == 'u') {

        // 元音值增加 1
        ++vowels;
    }

    // 如果它不是元音，如果它是字母表，它就是辅音
    else if ((line[i] >= 'a' && line[i] <= 'z')) {
        ++consonant;
    }

    // 检查字符是否为数字
    else if (line[i] >= '0' && line[i] <= '9') {
        ++digit;
    }

    // 检查字符是否为空白
    else if (line[i] == ' ') {
        ++space;
    }
}

printf("Vowels: %d", vowels);
printf("\nConsonants: %d", consonant);
printf("\nDigits: %d", digit);
printf("\nWhite spaces: %d", space);

return 0;
}

```

## 输出

```

Enter a line of string: C++ 20 is the latest version of C++ yet.
Vowels: 9
Consonants: 16
Digits: 2
White spaces: 8

```

在这里，用户输入的字符串存储在 `line` 变量中。

最初，`vowel`, `consonant`, `digit` 和 `space` 初始化为 0。

然后，使用 `for` 循环遍历字符串中的字符。在每次迭代中，我们：

- 使用 `tolower()` 函数将字符转换为小写。

- 检查字符是否为元音、辅音、数字或空格。假设这个字符是辅音。然后，`consonant` 变量增加 1。

当循环结束时，元音、辅音、数字和空格的数量分别存储在 `vowel`, `consonant`, `digit` 和 `space` 中。

注：我们使用了 `tolower()` 函数来简化我们的程序。要使用这个函数，我们需要导入 `ctype.h` 头文件。

### 3. 使用递归反转字符串

若要理解此示例，应了解以下 C 的编程知识：

- 函数
- 用户自定义函数
- 递归

#### 示例：使用递归反转句子

```
#include <stdio.h>
void reverseSentence();
int main() {
    printf("Enter a sentence: ");
    reverseSentence();
    return 0;
}

void reverseSentence() {
    char c;
    scanf("%c", &c);
    if (c != '\n') {
        reverseSentence();
        printf("%c", c);
    }
}
```

输出

```
Enter a sentence: margorp emosewa
awesome program
```

这个程序首先打印。然后，调用函数。输入一个句子:`reverseSentence()`

这个函数存储用户输入的第一个字母。如果变量是(`newline`)以外的任何字符，则再次调用。

这个过程一直持续到用户点击回车。

当用户按 `enter` 键时，该函数开始从 `last.reverseSentence()` 中打印字符。

## 4. 求字符串的长度

若要理解此示例，应了解以下 C 的编程知识：

- 字符串
- 使用库函数的 C 编程中的字符串操作
- `for` 循环

如你所知，查找字符串长度的最佳方法是使用函数 `strlen()`。然而，在本例中，我们将手动查找字符串的长度。

### 示例：在不使用 `strlen()` 函数的情况下计算字符串的长度

```
#include <stdio.h>
int main() {
    char s[] = "Programming is fun";
    int i;

    for (i = 0; s[i] != '\0'; ++i);

    printf("Length of the string: %d", i);
    return 0;
}
```

输出

```
Length of the string: 18
```

在这里，使用 `for` 循环，我们对字符串的字符进行迭代，从 `i=0` 开始，直到遇到 “`\0`”（`null` 字符）。在每次迭代中，`i` 的值增加 1。

注：这里，数组 `s[]` 有 19 个元素。最后一个元素 `s[18]` 是空字符 '`\0`'。但是我们的循环不会计算这个字符，因为它会在遇到它时终止。

## 5. 连接两个字符串

若要理解此示例，应了解以下 C 的编程知识：

- 数组
- 编程字符串
- 循环

如你所知，在 C 编程中连接两个字符串的最佳方法是使用 `strcat()` 函数。但是，在此示例中，我们将手动连接两个字符串。

示例：在不使用 `strcat()` 的情况下连接两个字符串

```
#include <stdio.h>
int main() {
    char s1[100] = "programming ", s2[] = "is awesome";
    int length, j;

    // 在长度变量中存储 s1 的长度
    length = 0;
    while (s1[length] != '\0') {
        ++length;
    }

    // 将 s2 连接到 s1
    for (j = 0; s2[j] != '\0'; ++j, ++length) {
        s1[length] = s2[j];
    }

    // 终止 s1 字符串
    s1[length] = '\0';

    printf("After concatenation: ");
    puts(s1);

    return 0;
}
```



输出

```
After concatenation: programming is awesome
```

这里，两个字符串 `s1` 和 `s2` 连接起来，结果存储在 `s1` 中。

重要的是要注意，`s1` 的长度应该足以容纳连接后的字符串。否则，你可能会得到意想不到的输出。

## 6. 用于复制字符串的 C 程序

若要理解此示例，应了解以下 C 的编程知识：

- 数组
- 字符串
- for 循环

正如你了解的一样，复制字符串的最佳方法是使用 `strcpy()`。但是，在此示例中，我们将复制一个字符串，而不使用 `strcpy()`。

### 示例：在不使用 `strcpy()` 的情况下复制字符串

```
#include <stdio.h>
int main()
{
    char s1[100], s2[100], i;
    printf("Enter string s1: ");
    fgets(s1, sizeof(s1), stdin);

    for (i = 0; s1[i] != '\0'; ++i) {
        s2[i] = s1[i];
    }

    s2[i] = '\0';
    printf("String s2: %s", s2);
    return 0;
}
```

输出

```
Enter string s1: Hey fellow programmer.
String s2: Hey fellow programmer.
```

上面的程序手动将字符串 `s1` 的内容复制到字符串 `s2`。

## 7. 删除字符串中除字母表以外的所有字符

若要理解此示例，应了解以下 C 的编程知识：

- 数组
- 字符串
- for 循环
- while 和 do...while 循环

### 示例：删除字符串中除字母表之外的字符

```
#include <stdio.h>
int main() {
    char line[150];

    printf("Enter a string: ");
    fgets(line, sizeof(line), stdin); // take input

    for (int i = 0, j; line[i] != '\0'; ++i) {
        // 如果字符不是字母表，则进入循环
        // 而不是空字符
        while (!(line[i] >= 'a' && line[i] <= 'z') && !(line[i] >= 'A' && line[i] <= 'Z') && !(line[i]
== '\0')) {
            for (j = i; line[j] != '\0'; ++j) {

                // 如果行的第 j 个元素不是字母表，
                // 将第 (j+1) 个元素的值分配给第 j 个元素
                line[j] = line[j + 1];
            }
            line[j] = '\0';
        }
    }
    printf("Output String: ");
    puts(line);
    return 0;
}
```

```
}
```

输出

```
Enter a string: p2'r-o@gram84iz./
Output String: programiz
```

这个程序接受用户的字符串输入，并将其存储在 `line` 变量中。然后，使用 `for` 循环遍历字符串中的字符。

如果字符串中的字符不是字母，则从字符串中删除该字符，并将其余字符的位置向左移动 1 个位置。

## 8. 按字典顺序对元素进行排序

若要理解此示例，应了解以下 C 的编程知识：

- 多维数组
- 字符串
- 使用库函数的 C 编程中的字符串操作

### 示例：按字典顺序对字符串进行排序

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

int main() {
    char str[5][50], temp[50];
    printf("Enter 5 words: ");

    // 获取字符串输入
    for (int i = 0; i < 5; ++i) {
        fgets(str[i], sizeof(str[i]), stdin);
    }

    // 按字典顺序存储字符串
    for (int i = 0; i < 5; ++i) {
        for (int j = i + 1; j < 5; ++j) {

            // 如果字符串不在字典顺序中，则交换它们
            if (strcmp(str[i], str[j]) > 0) {
                strcpy(temp, str[i]);
```

```

        strcpy(str[i], str[j]);
        strcpy(str[j], temp);
    }
}

printf("\nIn the lexicographical order: \n");
for (int i = 0; i < 5; ++i) {
    fputs(str[i], stdout);
}
return 0;
}

```

输出

```

Enter 5 words: R programming
JavaScript
Java
C programming
C++ programming

In the lexicographical order:
C programming
C++ programming
Java
JavaScript
R programming

```

为了解决这个程序，创建了一个名为 `str` 的二维字符串。字符串最多可容纳 5 个字符串，每个字符串最多可包含 50 个字符(包括 null 字符)。

在程序中，我们使用了两个库函数：

- `strcmp()`-比较字符串
- `strcpy()`-复制字符串

这些函数用于比较字符串并按正确的顺序对它们进行排序。

## 7 结构体和联合

### 7.1 结构体

在 C 编程中，结构体 `struct` 或 `structure` 是单个名称下的变量（可以是不同类型的）的集合。

---

#### 定义结构

在创建结构变量之前，需要定义它的数据类型。要定义结构，需要使用 `struct` 关键字。

#### `struct` 的语法

```
struct 结构体名称{  
    数据类型 成员 1;  
    数据类型 成员 2;  
    ...  
};
```

例如

```
struct Person {  
    char name[50];  
    int citNo;  
    float salary;  
};
```

这里定义了一个派生类型 `struct Person`。现在，你可以创建这种类型的变量。

---

#### 创建结构变量

声明 `struct` 类型时，不会分配存储空间或内存。为了分配给定结构类型的内存并使用它，我们需要创建变量。

以下是我们创建结构变量的方法：

```
struct Person {  
    // 代码  
};  
  
int main() {  
    struct Person person1, person2, p[20];  
    return 0;  
}
```

另一种创建 `struct` 变量的方法是：

```
struct Person {  
    // 代码  
} person1, person2, p[20];
```

在这两种情况下，

- `person1` 和 `person2` 是 `struct Person` 变量
- `p[]` 是一个 `struct Person` 数组，大小为 20。

---

## 访问结构的成员

有两种类型的运算符用于访问结构的成员。

1. `.` 成员操作符
2. `->` 结构指针运算符（将在下一个教程中讨论）

假设你想访问 `person2` 的 `salary`。你可以这样做：

```
person2.salary
```

## 示例 1: C 的结构

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

// 使用 person1 变量创建结构
struct Person
{
    char name[50];
    int citNo;
    float salary;
} person1;

int main() {

    // 为 person1 的 name 赋值
    strcpy(person1.name, "George Orwell");

    // 为其他 person1 变量赋值
    person1.citNo = 1984;
    person1.salary = 2500;

    // 打印结构体的变量
    printf("Name: %s\n", person1.name);
    printf("Citizenship No.: %d\n", person1.citNo);
    printf("Salary: %.2f", person1.salary);

    return 0;
}
```

输出

```
Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00
```

在这个程序中，我们创建了一个名为 `Person` 的结构体。我们还创建了一个名为 `person1` 的 `Person` 变量。

在 `main()` 中，我们为 `person1` 对象在 `Person` 中定义的变量赋值

```
strcpy(person1.name, "George Orwell");
person1.citNo = 1984;
person1.salary = 2500;
```

注意，我们使用了 `strcpy()` 函数将值赋给 `person1.name`。

这是因为 `name` 是一个 `char` 数组 (C-string)，在声明了该字符串之后，不能对其使用赋值操作符 `=`。

最后，我们打印了 `person1` 的数据。

---

## 关键字 typedef

我们使用 `typedef` 关键字为数据类型创建别名。它通常与结构一起使用，以简化声明变量的语法。

例如，让我们看一下以下代码：

```
struct Distance{
    int feet;
    float inch;
};

int main() {
    struct Distance d1, d2;
}
```

我们可以使用 `typedef` 以简化的语法编写等价的代码：

```
typedef struct Distance {
    int feet;
    float inch;
} distances;

int main() {
    distances d1, d2;
}
```

---

## 示例 2: typedef

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

// struct with typedef person
typedef struct Person {
    char name[50];
    int citNo;
```



```

float salary;
} person;

int main() {

    // 创建 Person 变量
    person p1;

    // 为 p1 的 name 分配值
    strcpy(p1.name, "George Orwell");

    // 为 p1 的其他变量分配值
    p1.citNo = 1984;
    p1.salary = 2500;

    // 打印结构体的变量
    printf("Name: %s\n", p1.name);
    printf("Citizenship No.: %d\n", p1.citNo);
    printf("Salary: %.2f", p1.salary);

    return 0;
}

```

## 输出

```

Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00

```

在这里，我们使用 `typedef` 创建了结构体 `Person` 并创建其别名 `person`。

```

//具有 typedef person 的结构
typedef struct Person {
    // code
} person;

```

现在，我们可以简单地使用 `Person` 别名声明一个 `Person` 变量：

```

//相当于结构体 Person p1
person p1;

```

## 嵌套结构

你可以在 C 中的结构中创建结构。例如

```
struct complex {
    int imag;
    float real;
};

struct number {
    struct complex comp;
    int integers;
} num1, num2;
```

假设你想将 `imag_num2` 变量设置为 11。你可以这样做:

```
num2.comp.imag = 11;
```

---

### 示例 3：嵌套结构

```
#include <stdio.h>

struct complex {
    int imag;
    float real;
};

struct number {
    struct complex comp;
    int integer;
} num1;

int main() {

    // 初始化 complex 变量
    num1.comp.imag = 11;
    num1.comp.real = 5.25;

    // 初始化 number 变量
    num1.integer = 6;

    // 打印结构体变量
    printf("Imaginary Part: %d\n", num1.comp.imag);
    printf("Real Part: %.2f\n", num1.comp.real);
    printf("Integer: %d", num1.integer);
}
```

```
return 0;  
}
```

输出

```
Imaginary Part: 11  
Real Part: 5.25  
Integer: 6
```

---

## 为什么需要结构体？

假设你想存储一个人的信息:他/她的名字、公民号码和工资。你可以创建不同的变量 `name`, `citNo` 和 `salary` 来存储这些信息。

如果需要存储多个人的信息怎么办?现在, 你需要为每个人的每个信息创建不同的变量:`name1`, `citNo1`, `salary1`, `name2`, `citNo2`, `salary2`, 等等。

更好的方法是将所有相关信息集合在一个人名 `Person` 结构下, 并对每个人使用它。

## 7.2 结构和指针

在了解指针如何与结构一起使用之前，请务必查看以下教程：

- 指针
- 结构体

---

### 指向结构的指针

下面介绍如何创建指向结构的指针。

```
struct name {  
    member1;  
    member2;  
    .  
    .  
};  
  
int main()  
{  
    struct name *ptr, Harry;  
}
```

这里，`ptr` 是指向 `struct` 的指针。

---

### 示例：使用指针访问成员

要使用指针访问结构体的成员，可使用 `->` 操作符。

```
#include <stdio.h>  
struct person  
{  
    int age;  
    float weight;  
};  
  
int main()  
{  
    struct person *personPtr, person1;  
    personPtr = &person1;  
  
    printf("Enter age: ");  
    scanf("%d", &personPtr->age);
```

```

printf("Enter weight: ");
scanf("%f", &personPtr->weight);

printf("Displaying:\n");
printf("Age: %d\n", personPtr->age);
printf("weight: %f", personPtr->weight);

return 0;
}

```

在这个例子中，使用 `personPtr = &person1;` 将 `person1` 的地址存储在指针 `personPtr` 中。

现在，可以使用 `personPtr` 指针访问 `person1` 的成员。

顺便一提，

- `personPtr->age` 相当于 `(*personPtr).age`
- `personPtr->weight` 相当于 `(*personPtr).weight`

## 结构体的动态内存分配

在继续本节之前，建议你先学习动态内存分配。

有时，你声明的结构变量的数量可能不足。你可能需要在运行时分配内存。以下是如何在 C 中实现这一点。

### 示例：结构的动态内存分配

```

#include <stdio.h>
#include <stdlib.h>

struct person {
    int age;
    float weight;
    char name[30];
};

int main()
{
    struct person *ptr;
    int i, n;

    printf("Enter the number of persons: ");
    scanf("%d", &n);

```

```
// 为 n 个结构 Person 分配内存
ptr = (struct person*) malloc(n * sizeof(struct person));

for(i = 0; i < n; ++i)
{
    printf("Enter first name and age respectively: ");

    // 为了访问 person 结构体的第一个成员,
    // 使用 ptr->name 和 ptr->age

    // 为了访问 person 结构体的第二个成员,
    // 使用 (ptr+1)->name 和 (ptr+1)->age
    scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);
}

printf("Displaying Information:\n");
for(i = 0; i < n; ++i)
    printf("Name: %s\tAge: %d\n", (ptr+i)->name, (ptr+i)->age);

return 0;
}
```

运行程序时，输出将为：

```
Enter the number of persons: 2
Enter first name and age respectively: Harry 24
Enter first name and age respectively: Gary 32
Displaying Information:
Name: Harry      Age: 24
Name: Gary       Age: 32
```

在上面的示例中，创建了 `n` 个结构变量，其中 `n` 由用户输入。

为了给 `n` 个 `struct person` 分配内存，我们使用：

```
ptr = (struct person*) malloc(n * sizeof(struct person));
```

然后，使用 `ptr` 指针访问 `person` 的元素。

## 7.3 结构与函数

与内置类型的变量类似，也可以将结构变量传递给函数。

### 将结构传递给函数

我们建议你在学习如何将结构传递给函数之前先学习这些教程：

- 结构
  - 函数
  - 用户自定义函数
- 

下面介绍如何将结构传递给函数：

```
#include <stdio.h>
struct student {
    char name[50];
    int age;
};

// 函数原型
void display(struct student s);

int main() {
    struct student s1;

    printf("Enter name: ");

    // 读取用户输入的字符串，直到\n输入为止
    // \n 被丢弃
    scanf("%[^\n]%*c", s1.name);

    printf("Enter age: ");
    scanf("%d", &s1.age);

    display(s1); // 将结构作为参数传递

    return 0;
}

void display(struct student s) {
    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nAge: %d", s.age);
}
```

输出

```
Enter name: Bond
Enter age: 13

Displaying information
Name: Bond
Age: 13
```

这里创建了一个 `struct` 变量 `s1`，类型为 `struct student`。使用 `display(s1)` 将该变量传递给 `display()` 函数。

---

## 从函数返回结构

下面介绍如何从函数返回结构：

```
#include <stdio.h>
struct student
{
    char name[50];
    int age;
};

// 功能原型
struct student getInformation();

int main()
{
    struct student s;

    s = getInformation();

    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nRoll: %d", s.age);

    return 0;
}

struct student getInformation()
{
    struct student s1;

    printf("Enter name: ");
    scanf ("%[^\\n]%*c", s1.name);

    printf("Enter age: ");
    scanf ("%d", &s1.age);
```



```
return s1;
}
```

这里，使用 `s=getInformation()` 调用 `getInformation()` 函数。该函数返回一个 `struct student` 类型的结构体。返回的结构体通过 `main()` 函数显示。

注意，`getInformation()` 的返回类型也是 `struct student`。

## 通过引用传递结构

你还可以通过引用传递结构（以类似于通过引用传递内置类型的变量的方式）。

在引用传递期间，`struct` 变量的内存地址将传递给函数。

```
#include <stdio.h>
typedef struct Complex
{
    float real;
    float imag;
} complex;

void addNumbers(complex c1, complex c2, complex *result);

int main()
{
    complex c1, c2, result;

    printf("For first number,\n");
    printf("Enter real part: ");
    scanf("%f", &c1.real);
    printf("Enter imaginary part: ");
    scanf("%f", &c1.imag);

    printf("For second number, \n");
    printf("Enter real part: ");
    scanf("%f", &c2.real);
    printf("Enter imaginary part: ");
    scanf("%f", &c2.imag);

    addNumbers(c1, c2, &result);
    printf("\nresult.real = %.1f\n", result.real);
    printf("result.imag = %.1f", result.imag);

    return 0;
}

void addNumbers(complex c1, complex c2, complex *result)
{
    result->real = c1.real + c2.real;
    result->imag = c1.imag + c2.imag;
}
```

输出

```
For first number,  
Enter real part: 1.1  
Enter imaginary part: -2.4  
For second number,  
Enter real part: 3.4  
Enter imaginary part: -3.2  
  
result.real = 4.5  
result.imag = -5.6
```

在上面的程序中,三个结构变量 `c1`、`c2` 和 `result` 的地址被传递给 `addNumbers()` 函数。这里, `result` 是通过引用传递的。

当 `addNumbers()` 中的 `result` 变量改变时, `main()` 函数中的 `result` 变量也会相应改变。

## 7.4 联合

联合是一种用户定义的类型，类似于 C 中的结构，但有一个关键区别。

结构分配足够的空间来存储其所有成员，而联合一次只能保存一个成员值。

---

### 如何定义联合？

我们使用 `union` 关键字来定义联合。下面是一个例子：

```
union car
{
    char name[50];
    int price;
};
```

上面的代码定义了一个派生类型 `union car`。

---

### 创建联合变量

定义联合时，它会创建用户定义的类型。但是，不会分配内存。要为给定的联合类型分配内存并使用它，我们需要创建变量。

以下是我们创建联合变量的方法。

```
union car
{
    char name[50];
    int price;
};

int main()
{
    union car car1, car2, *car3;
    return 0;
}
```

```
}
```

创建联合变量的另一种方法是：

```
union car
{
    char name[50];
    int price;
} car1, car2, *car3;
```

在这两种情况下，都会创建联合变量 `car1`、`car2` 和 `union car` 类型的联合指针 `car3`。

## 访问联合成员

我们用 `.` 访问联合成员的操作符。为了访问指针变量，我们使用 `->` 操作符。

在上面的例子中，

- 要访问 `car1` 的 `price`，使用 `car1.price`。
- 要访问 `car3` 的 `price`，要么使用 `(*car3).price` 或 `car3->price`。

## 联合和结构之间的区别

让我们举个例子来演示联合和结构之间的区别：

```
#include <stdio.h>
union unionJob
{
    // 定义一个联合
    char name[32];
    float salary;
    int workerNo;
} uJob;

struct structJob
{
    char name[32];
    float salary;
```

```
int workerNo;
} sJob;

int main()
{
    printf("size of union = %d bytes", sizeof(uJob));
    printf("\nsize of structure = %d bytes", sizeof(sJob));
    return 0;
}
```

输出

```
size of union = 32
size of structure = 40
```

为什么联合变量和结构变量的大小存在这种差异？

这里，`sJob` 的大小是 40 字节，因为

- `name[32]` 的大小为 32 字节
- `salary` 的大小是 4 字节
- `workerNo` 的大小是 4 字节

然而，`uJob` 的大小是 32 字节。这是因为 `union` 变量的大小始终是其最大元素的大小。在上面的例子中，其最大元素 (`name[32]`) 的大小是 32 字节。

通过联合，所有成员共享相同的内存。

---

示例：访问联合成员

```
#include <stdio.h>
union Job {
    float salary;
    int workerNo;
} j;

int main() {
    j.salary = 12.3;
}
```

```
// 当给 j.workerNo 分配一个值时，  
// j.salary 将不再为 12.3  
j.workerNo = 100;  
  
printf("Salary = %.1f\n", j.salary);  
printf("Number of workers = %d", j.workerNo);  
return 0;  
}
```

输出

```
Salary = 0.0  
Number of workers = 100
```

## 8 操作文件

### 8.1 文件处理

文件是计算机存储设备中用于存储数据的容器。

---

#### 为什么需要文件？

- 当程序终止时，整个数据将丢失。即使程序终止，存储在文件中也会保留你的数据。
- 如果必须输入大量数据，则需要花费大量时间才能全部输入。
- 但是，如果你有一个包含所有数据的文件，则可以使用 C 中的几个命令轻松访问文件的内容。
- 你可以轻松地将数据从一台计算机移动到另一台计算机，而无需进行任何更改。

#### 文件类型

处理文件时，你应该了解两种类型的文件：

1. 文本文件
2. 二进制文件

##### 1. 文本文件

文本文件是普通的 **.txt** 文件。你可以使用任何简单的文本编辑器（如记事本）轻松创建文本文件。当你打开这些文件时，你将看到文件中的所有内容都是纯文本。你可以轻松编辑或删除内容。它们需要最少的维护工作量，易于阅读，并提供最低的安全性并占用更大的存储空间。

##### 2. 二进制文件

二进制文件大多是计算机中的 **.bin** 文件。他们不是以纯文本形式存储数据，而是以二进制形式（0 和 1）存储数据。它们可以容纳更多的数据，不容易读取，并且提供比文本文件更好的安全性。

## 文件操作

在 C 语言中，可以对文件（文本或二进制文件）执行四个主要操作：

1. 创建新文件
  2. 打开现有文件
  3. 关闭文件
  4. 读取文件和写入文件的信息
- 

## 使用文件

使用文件时，需要声明 `file` 类型的指针。文件和程序之间的通信需要此声明。

```
FILE *fptr;
```

---

## 打开文件-用于创建和编辑

打开文件是使用定义在 `stdio.h` 头文件中的 `fopen()` 函数执行的。

在标准 [I/O](#) 中打开文件的语法为：

```
ptr = fopen("filename", "mode");
```

例如

```
fopen("E:\\cprogram\\newprogram.txt", "w");  
fopen("E:\\cprogram\\oldprogram.bin", "rb");
```



- 假设文件 `newprogram.txt` 在位置 `E:\cprogram` 中不存在。第一个函数创建了一个名为 `newprogram.txt` 的新文件，并以 ‘w’ 模式打开它进行写入。
  - 写入模式允许你创建和编辑(覆盖)文件的内容。
  - 现在假设第二个二进制文件 `oldprogram.bin` 存在于位置 `E:\cprogram`。第二个函数打开现有文件，以二进制模式 ‘rb’ 读取。
- 读取模式只允许读取文件，不能写入文件。

标准 I/O 中的打开模式		
模式	模式的含义	在文件不存在期间
r	打开以供读取	如果文件不存在， <code>fopen()</code> 返回 NULL。
rb	以二进制模式打开读取	如果文件不存在， <code>fopen()</code> 返回 NULL。
w	打开以供写入	如果文件存在，则其内容将被覆盖。 如果文件不存在，则将创建该文件。
wb	以二进制模式打开写入	如果文件存在，则其内容将被覆盖。 如果文件不存在，则将创建该文件。
a	打开以供追加 数据将添加到文件末尾	如果文件不存在，则将创建该文件。
ab	在二进制模式下打开追加 数据将添加到文件末尾	如果文件不存在，则将创建该文件。
r+	打开以供读/写	如果文件不存在， <code>fopen()</code> 返回 NULL。
rb+	以二进制模式打开以供读/写	如果文件不存在， <code>fopen()</code> 返回 NULL。
w+	打开以供读/写	如果文件存在，则其内容将被覆盖。 如果文件不存在，则将创建该文件。
wb+	以二进制模式打开以供读/写	如果文件存在，则其内容将被覆盖。 如果文件不存在，则将创建该文件。
a+	打开以供读取和追加	如果文件不存在，则将创建该文件。
ab+	以二进制模式打开以供读取和追加	如果文件不存在，则将创建该文件。

## 关闭文件

文件（文本和二进制文件）应在读取/写入后关闭。

使用 `fclose()` 函数关闭文件。

```
fclose(fp_ptr);
```

这里，`fp_ptr` 是一个文件指针，关联到要关闭的文件。

---

## 读取和写入文本文件

要读写文本文件，我们使用函数 `fprintf()` 和 `fscanf()`。

它们只是 `printf()` 和 `scanf()` 的文件版本。唯一的区别是 `fprintf()` 和 `fscanf()` 需要一个指向 `FILE` 结构的指针。

---

### 示例 1：写入文本文件

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

int main()
{
    int num;
    FILE *fp_ptr;

    // 如果你使用的是 MacOS 或 Linux，请使用适当的位置
    fp_ptr = fopen("C:\\program.txt", "w");

    if(fp_ptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    printf("Enter num: ");
    scanf("%d", &num);
```

```

fprintf(fptr,"%d",num);
fclose(fptr);

return 0;
}

```

这个程序从用户那里获取一个数字，并将其存储在文件 `program.txt` 中。

编译并运行这个程序后，你可以看到在计算机的 **C** 驱动器中创建了一个文本文件 `program.txt`。当你打开这个文件时，可以看到你输入的整数。

## 示例 2：从文本文件中读取

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.txt","r")) == NULL)
    {
        printf("Error! opening file");

        // 如果文件指针返回 NULL，程序将退出。
        exit(1);
    }

    fscanf(fptr,"%d", &num);

    printf("Value of n=%d", num);
    fclose(fptr);

    return 0;
}

```

这个程序读取 `program.txt` 文件中的整数，并将其打印到屏幕上。

如果你成功地根据示例 1 创建了文件，运行这个程序将得到你输入的整数。

其他函数如 `fgetchar()`，`fputc()` 等可以以类似的方式使用。

## 读取和写入二进制文件

函数 `fread()` 和 `fwrite()` 分别用于从磁盘上读取和写入二进制文件。

## 写入二进制文件

要将数据写入二进制文件，需要使用 `fwrite()` 函数。这些函数接受以下 4 个参数。

1. 要写入磁盘的数据的地址
2. 要写入磁盘的数据的大小
3. 此类数据的数量
4. 指向要写入的文件的指针。

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

### 示例 3：使用 `fwrite()` 写入二进制文件

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

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin", "wb")) == NULL)
    {
```

```

    printf("Error! opening file");

    // 如果文件指针返回 NULL，程序将退出。
    exit(1);
}

for(n = 1; n < 5; ++n)
{
    num.n1 = n;
    num.n2 = 5*n;
    num.n3 = 5*n + 1;
    fwrite(&num, sizeof(struct threeNum), 1, fptr);
}
fclose(fptr);

return 0;
}

```

在这个程序中，我们在 C 驱动器中创建了一个新文件 `program.bin`。

我们声明了一个结构 `threeNum`，它包含 3 个数字——`n1`、`n2` 和 `n3`，并在 `main` 函数中定义它为 `num`。

现在，在 `for` 循环中，我们使用 `fwrite()` 将值存储到文件中。

第一个参数是 `num` 的地址，第二个参数是结构体 `threeNum` 的大小。

因为我们只插入一个 `num` 实例，所以第三个参数是 1。最后一个参数 `*fptr` 指向存储数据的文件。

最后，关闭文件。

## 从二进制文件读取

函数 `fread()` 也接受 4 个参数，与上面的 `fwrite()` 函数类似。

```
要写入磁盘的数据的地址(addressData, sizeData, numbersData, pointerToFile);
```

### 示例 4：使用 `fread()` 从二进制文件中读取

```

#include <stdio.h>
#include <stdlib.h>

```

```

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin", "rb")) == NULL)
    {
        printf("Error! opening file");

        // 如果文件指针返回 NULL，程序将退出。
        exit(1);
    }

    for(n = 1; n < 5; ++n)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
    }
    fclose(fptr);

    return 0;
}

```

在这个程序中，读取同一个文件 `program.bin`，并逐个遍历记录。

简单来说，从 `*fptr` 指向的文件中读取一条 `threeNum` 大小的 `threeNum` 记录到结构 `num` 中。

你将获得与示例 3 中插入的相同记录。

---

## 使用 `fseek()` 获取数据

如果文件中有许多记录，并且需要访问特定位置的记录，则需要遍历之前的所有记录以获取记录。

这将浪费大量内存和操作时间。使用 `fseek()` 可以更容易地获取所需的数据。

顾名思义，`fseek()` 将游标查找到文件中的给定记录。

---

## fseek() 的语法

```
fseek(FILE * stream, long int offset, int whence);
```

第一个参数流是指向文件的指针。第二个参数是要查找的记录的位置，第三个参数指定偏移量开始的位置。

### fseek() 中的不同之处

何处	意义
SEEK_SET	从文件开头开始偏移。
SEEK_END	从文件末尾开始偏移。
SEEK_CUR	从文件中光标的当前位置开始偏移。

### 示例 5: fseek()

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

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
        printf("Error! opening file");

        // 如果文件指针返回 NULL，程序将退出。
        exit(1);
    }
}
```

```
// 将光标移动到文件的末尾
fseek(fptr, -sizeof(struct threeNum), SEEK_END);

for(n = 1; n < 5; ++n)
{
    fread(&num, sizeof(struct threeNum), 1, fptr);
    printf("n1: %d\\tn2: %d\\tn3: %d\\n", num.n1, num.n2, num.n3);
    fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
}
fclose(fptr);

return 0;
}
```

这个程序将从 `program.bin` 文件中以相反的顺序(从后到前)读取记录并打印出来。



## 8.2 文件操作示例

要学习本节知识，你应该了解以下主题。

- 数组
- 指针
- 数组和指针关系
- 文件 I/O

### 文件操作示例

1. 读取 n 个学生的姓名和分数，并将其存储在文件中。

```
#include <stdio.h>

int main()
{
    char name[50];
    int marks, i, num;

    printf("Enter number of students: ");
    scanf("%d", &num);

    FILE *fptr;
    fptr = (fopen("C:\\student.txt", "w"));
    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    for(i = 0; i < num; ++i)
    {
        printf("For student%d\nEnter name: ", i+1);
        scanf("%s", name);

        printf("Enter marks: ");
        scanf("%d", &marks);

        fprintf(fptr, "\nName: %s \nMarks=%d \n", name, marks);
    }

    fclose(fptr);
    return 0;
}
```

2. 读取 n 个学生的姓名和标记，并将其存储在文件中。如果文件已经存在，再将信息追加到文件中。

```
#include <stdio.h>

int main()
{
    char name[50];
    int marks, i, num;

    printf("Enter number of students: ");
    scanf("%d", &num);

    FILE *fptr;
    fptr = (fopen("C:\\student.txt", "a"));
    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    for(i = 0; i < num; ++i)
    {
        printf("For student%d\nEnter name: ", i+1);
        scanf("%s", name);

        printf("Enter marks: ");
        scanf("%d", &marks);

        fprintf(fptr, "\nName: %s \nMarks=%d \n", name, marks);
    }

    fclose(fptr);
    return 0;
}
```

3. 使用 fwrite() 将一个结构的数组的所有成员写到文件，然后再从文件读取数组并将每个元素显式到屏幕上。

```
#include <stdio.h>

struct student
{
    char name[50];
    int height;
};

int main()
{
    struct student stud1[5], stud2[5];
```

```
FILE *fptr;
int i;

fptr = fopen("file.txt", "wb");
for(i = 0; i < 5; ++i)
{
    fflush(stdin);
    printf("Enter name: ");
    gets(stud1[i].name);

    printf("Enter height: ");
    scanf("%d", &stud1[i].height);
}

fwrite(stud1, sizeof(stud1), 1, fptr);
fclose(fptr);

fptr = fopen("file.txt", "rb");
fread(stud2, sizeof(stud2), 1, fptr);
for(i = 0; i < 5; ++i)
{
    printf("Name: %s\nHeight: %d", stud2[i].name, stud2[i].height);
}
fclose(fptr);
}
```

## 9 额外主题

### 9.1 关键字和标识符

#### 字符集

字符集是一组在 C 语言中有效的字母、字母和一些特殊字符。

#### 字母表

```
大写: A B C ..... X Y Z
小写: a b c ..... x y z
```

C 接受小写和大写字母作为变量和函数。

#### 数字

```
0 1 2 3 4 5 6 7 8 9
```

#### 特殊字符

C 语言编程中的特殊字符

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

- \ ~ +

## 空白字符

空格、换行符、水平制表符、回车符和表单填充符。

## 语言关键字

关键字是编程中使用的预定义保留字，对编译器具有特殊含义。关键字是语法的一部分，不能用作标识符。例如：

```
int money;
```

这里的 `int` 是关键字，表示 `money` 是 `int` (整数) 类型的变量。

由于 C 是一种区分大小写的语言，因此所有关键字都必须以小写形式书写。以下是 ANSI C 中允许的所有关键字的列表。

关键字			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
continue	for	signed	void
do	if	static	while
default	goto	sizeof	volatile
const	float	short	unsigned

所有这些关键字、它们的语法和应用都将在各自的主题中讨论。

---

## C 语言的标识符

标识符是指赋予实体的名称，例如变量、函数、结构等。

标识符必须是唯一的。创建它们是为了给实体一个唯一的名称，以便在程序执行期间标识它。例如：

```
int money;  
double accountBalance;
```

在这里，`money` 和 `accountBalance` 是标识符。

还要记住，标识符的名称必须不同于关键字。`int` 是关键字，不能用作标识符。

---

## 命名标识符的规则

- 有效标识符可以包含字母（大写和小写字母）、数字和下划线。
- 标识符的第一个字母应为字母或下划线。
- 不能使用像 `int`, `while` 等关键字作为标识符。
- 没有关于标识符可以有多长的规定。但是，如果标识符的长度超过 31 个字符，则在某些编译器中可能会遇到问题。

如果你遵循上述规则，你可以选择任何名称作为标识符，但是，请为有意义的标识符提供有意义的名称。

## 9.2 运算符的优先级和关联性

### 运算符的优先级

如果表达式中有多个运算符，运算符的优先级决定了首先执行哪个运算符。

让我们考虑一个例子：

```
int x = 5 - 17 * 6;
```

在 C 语言中，`*` 的优先级高于 `-` 和 `=`。因此，首先计算 `17 * 6`。那么包含 `-` 的表达式的计算结果是 `-` 的优先级高于 `=`。

下面是操作符优先级从高到低的表。稍后将讨论关联性的性质。

### 运算符优先级和关联性表

运算符	运算符的含义	关联性
<code>()</code> <code>[]</code> <code>-&gt;</code> <code>.</code>	函数调用 数组元素引用 间接成员选择 直接成员选择	从左到右
<code>!</code> <code>~</code> <code>+</code> <code>-</code> <code>++</code> <code>--</code> <code>&amp;</code> <code>*</code>	逻辑否定 按位 (1's) 补码 一元加 一元减 递增 递减 取消引用 (地址) 指针引用	从右到左

<b>sizeof</b> (type)	返回对象的大小 类型转换	
* / %	乘以 除以 除以的余数	从左到右
+ -	二元加（加法） 二元减（减法）	从左到右
<< >>	左移 右移	从左到右
< <= > >=	小于 小于或等于 大于 大于或等于	从左到右
== !=	等于 不等于	从左到右
&	按位与	从左到右
^	按位异或	从左到右
	按位或	从左到右
&&	逻辑与	从左到右
	逻辑或	从左到右
?:	条件运算符	从右到左



=	简单分配	从右到左
*=	乘以后分配	
/=	除以后分配	
%=	求余后分配	
+=	求和后分配	
-=	求差后分配	
&=	按位与运算后分配	
^=	按位异或后分配	
=	按位或后分配	
<<=	左移后分配	
>>=	右移后分配	
,	表达式的分隔符	从左到右

## 运算符的关联性

运算符的关联性决定了计算表达式的方向。例如

```
b = a;
```

在这里，**a** 的值被赋给 **b**，而不是反过来。这是因为**=**运算符的结合性是从右向左的。

此外，如果存在两个具有相同优先级（优先级）的运算符，则关联性将确定它们的执行方向。

让我们考虑一个例子：

```
1 == 2 != 3
```

在这里，操作符**==**和**!=**具有相同的优先级。它们的结合律是从左到右。因此，**1 == 2** 会首先被执行。

上面的表达式等价于：

```
(1 == 2) != 3
```

**注：**如果语句有多个运算符，则可以使用括号（）使代码更具可读性。

## 9.3 按位运算符

在算术逻辑单元（位于 CPU 内）中，加法、减法、乘法和除法等数学运算都是在位级完成的。为了在 C 编程中执行位级运算，使用了按位运算符。

运算符	运算符的含义
&	按位 与
	按位 或
^	按位 异或
~	按位 补码
<<	左移
>>	右移

### 按位 与(AND) 运算符 &

如果两个操作数的对应位为 1，则按位与的输出为 1。如果操作数的任一为 0，则相应位的结果计算为 0。

在 C 中，按位与运算符用 `&` 表示。

让我们假设两个整数 12 和 25 的按位与运算。

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bit Operation of 12 and 25
  00001100
& 00011001
  -----
  00001000 = 8 (In decimal)
```

## 示例 1：按位与(AND)

```
#include <stdio.h>

int main() {

    int a = 12, b = 25;
    printf("Output = %d", a & b);

    return 0;
}
```

输出

```
Output = 8
```

## 按位 或(OR) 运算符 |

如果两个操作数的至少一个对应位为 1，则按位 OR 的输出为 1。在 C 编程中，按位或运算符用 | 表示。

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
  00001100
| 00011001
-----
  00011101 = 29 (In decimal)
```

## 示例 2：按位或

```
#include <stdio.h>

int main() {

    int a = 12, b = 25;
    printf("Output = %d", a | b);
}
```

```
    return 0;
}
```

输出

```
Output = 29
```

## 按位 异或 运算符 ^

如果两个操作数的相应位相反，则按位 异或 运算符的结果为 1。它用 ^ 表示。

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25
  00001100
^ 00011001
  -----
  00010101 = 21 (In decimal)
```

### 示例 3：按位异或

```
#include <stdio.h>

int main() {

    int a = 12, b = 25;
    printf("Output = %d", a ^ b);

    return 0;
}
```

输出

```
Output = 21
```

## 按位补码运算符 ~

按位补码运算符是一元运算符（仅处理一个操作数）。它将 1 更改为 0，将 0 更改为 1。它用~表示。

```
35 = 00100011 (In Binary)

Bitwise complement Operation of 35
~ 00100011

11011100 = 220 (In decimal)
```

## C 中按位补码运算符中的诡异现象

35(~35)的补码是-36 而不是 220，这是为什么呢？

对于任何整数  $n$ ,  $n$  的位补数将是  $-(n+1)$ 。要理解这一点，你应该了解 2 的补码。

## 2 的补码

二的补码是对二进制数的运算。一个数字的 2 的补码等于该数字的补码加上 1。例如：

Decimal	Binary	2's complement
0	00000000	$-(11111111+1) = -00000000 = -0(\text{decimal})$
1	00000001	$-(11111110+1) = -11111111 = -256(\text{decimal})$
12	00001100	$-(11110011+1) = -11110100 = -244(\text{decimal})$
220	11011100	$-(00100011+1) = -00100100 = -36(\text{decimal})$

Note: Overflow is ignored while computing 2's complement.

35 的位补数是 220(十进制)。220 的 2 的补数是-36。因此，输出是-36，而不是 220。

任意数字  $n$  的按位补码为  $-(N+1)$ 。方法如下：

```
bitwise complement of N = ~N (represented in 2's complement form)
2's complement of ~N = -(~N+1) = -(N+1)
```

#### 示例 4：按位补码

```
#include <stdio.h>

int main() {

    printf("Output = %d\n", ~35);
    printf("Output = %d\n", ~-12);

    return 0;
}
```

输出

```
Output = -36
Output = 11
```

## C 编程中的移位运算符

C 中有两个移位运算符：

- 右移运算符
- 左移运算符

### 右移运算符

右移运算符将所有位向右移动一定数量的指定位。它用 `>>` 表示。

```
212 = 11010100 (In binary)
212 >> 2 = 00110101 (In binary) [Right shift by two bits]
212 >> 7 = 00000001 (In binary)
212 >> 8 = 00000000
```

```
212 >> 0 = 11010100 (No Shift)
```

## 左移运算符

左移运算符将所有位向左移动一定数量的指定位。左移运算符腾出的位位置用 0 填充。左移运算符的符号是<<。

```
212 = 11010100 (In binary)
212<<1 = 110101000 (In binary) [Left shift by one bit]
212<<0 = 11010100 (Shift by 0)
212<<4 = 110101000000 (In binary) =3392(In decimal)
```

## 示例 5：移位运算符

```
#include <stdio.h>

int main() {

    int num=212, i;

    for (i = 0; i <= 2; ++i) {
        printf("Right shift by %d: %d\n", i, num >> i);
    }
    printf("\n");

    for (i = 0; i <= 2; ++i) {
        printf("Left shift by %d: %d\n", i, num << i);
    }

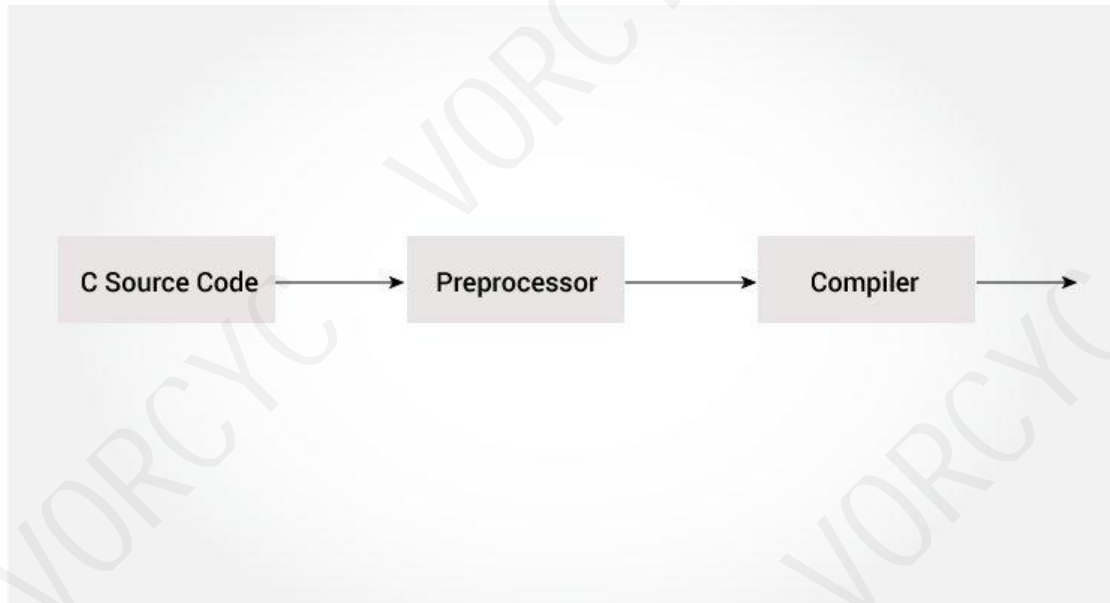
    return 0;
}
```



```
Right Shift by 0: 212  
Right Shift by 1: 106  
Right Shift by 2: 53
```

```
Left Shift by 0: 212  
Left Shift by 1: 424  
Left Shift by 2: 848
```

## 9.4 预处理器和宏



C 中预处理器的的工作原理

C 预处理器是一个宏预处理器（允许你定义宏），可在编译程序之前对其进行转换。这些转换可以是包含头文件、宏扩展等。

所有预处理指令都以符号#开头。例如

```
#define PI 3.14
```

C 预处理器的一些常见用途是：

### 包含头文件：#include

**#include** 预处理器用于将头文件包含到 C 程序中。例如

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

这里，**stdio.h** 是一个头文件。**#include** 预处理器指令将上面的行替换为 **stdio.h** 头文件的内容。

这就是为什么在使用 `scanf()` 和 `printf()` 等函数之前需要使用 `#include <stdio.h>` 的原因。你还可以创建自己的包含函数声明的头文件，并使用此预处理器指令将其包含在程序中。

```
#include "my_header.h"
```

## 使用 `#define` 的宏

宏是一个有名称的代码片段。你可以在 C 语言中使用 `#define` 预处理器指令定义宏。下面是一个示例。

```
#define c 299792458 // 光速
```

在这里，当我们在程序中使用 `c` 时，它被替换为 `299792458`。

---

### 示例 1: `#define` 预处理器

```
#include <stdio.h>
#define PI 3.1415

int main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%f", &radius);

    // 注意，这里使用了 PI
    area = PI*radius*radius;

    printf("Area=%.2f",area);
    return 0;
}
```

## 像函数一样的宏

你还可以定义以与函数调用类似的方式工作的宏。这称为类似函数的宏。例如

```
#define circleArea(r) (3.1415*(r)*(r))
```

每当程序遇到 `circleArea(argument)` 时，它都会被替换为 `(3.1415*(argument)*(argument))`。

假设，我们传递 5 作为参数，然后它展开如下：

```
circleArea(5) 展开为 (3.1415*5*5)
```

---

### 示例 2：使用 #define 预处理器

```
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)

int main() {
    float radius, area;

    printf("Enter the radius: ");
    scanf("%f", &radius);
    area = circleArea(radius);
    printf("Area = %.2f", area);

    return 0;
}
```

---

## 条件编译

在 C 中，你可以指示预处理器是否包含代码块。为此，可以使用条件指令。

它类似于 `if` 语句，但有一个主要区别。

`if` 语句在执行期间测试代码块是否应该执行，而条件语句用于在程序执行前包含(或跳过)代码块。

## 条件编译的用途

- 根据机器、操作系统使用不同的代码
- 在两个不同的程序中编译相同的源文件
- 从程序中排除某些代码，但将其保留为将来的参考

## 如何使用条件编译？

要使用条件编译指令，可以使用 `#ifdef`、`#if`、`#defined`、`#else` 和 `#elif` 指令。

### `#ifdef` 指令

```
#ifdef 宏
// 条件化的代码
#endif
```

在这里，只有定义了 **MACRO**（宏），条件代码才会包含在程序中。

### `#if`、`#elif` 和 `#else` 指令

```
#if 表达式
// 条件化的代码
#endif
```

这里，**表达式**是一个整数类型的表达式(可以是整数、字符、算术表达式、宏等)。

只有当表达式求值为非零值时，条件代码才会包含在程序中。

可选的 `#else` 指令可以和 `#if` 指令一起使用。

```
#if 表达式
    如果表达式为非 0 的条件化代码
#else
    如果表达式为 0 的条件化代码
#endif
```

你也可以在`#if`中添加嵌套条件语句`#if...#else`完成其他操作。

```
#if 表达式 1
    // 如果 if 表达式为 1 非 0 的条件化代码
#elif 表达式 2
    // 如果 elif 表达式 2 为非 0 的条件化代码
#elif 表达式 3
    // 如果 elif 表达式 3 为非 0 的条件化代码
#else
    // 所有的表达式都为 0 的条件化代码
#endif
```

---

## #defined

特殊运算符`#defined`用于测试某个宏是否定义。它经常与`#if`指令一起使用。

```
#if defined BUFFER_SIZE && BUFFER_SIZE >= 2048
    // 代码
```

## 预定义宏

以下是 C 中的一些预定义宏。

宏	值
<code>__DATE__</code>	包含当前日期的字符串。
<code>__FILE__</code>	包含文件名的字符串。
<code>__LINE__</code>	表示当前行号的整数。
<code>__STDC__</code>	如果遵循 ANSI 标准 C，则该值为非零整数。
<code>__TIME__</code>	包含当前时间的字符串。

### 示例 3：使用 `__TIME__` 获取当前时间

下面的程序使用 `__TIME__` 宏输出当前时间。

```
#include <stdio.h>
int main()
{
    printf("Current time: %s",__TIME__);
}
```

输出

```
Current time: 19:54:39
```

## 9.5 标准库函数

C 标准库函数或简称 C 库函数是 C 中的内置函数。

这些函数的原型和数据定义存在于它们各自的头文件中。要使用这些函数，我们需要在程序中包含头文件。例如：

如果你想使用 `printf()` 函数，头文件 `<stdio.h>` 应该包含在内。

```
#include <stdio.h>
int main()
{
    printf("Catch me if you can.");
}
```

如果你试图使用 `printf()` 而不包含 `stdio.h` 头文件，就会得到一个错误。

---

### 使用 C 库函数的优点

#### 1. 即得即用

你应该使用库函数的最重要原因之一仅仅是因为它们可以工作。这些功能经过多次严格测试，易于使用。

#### 2. 功能针对性能进行了优化

由于这些函数是“标准库”函数，因此专门的开发人员团队不断改进它们。在此过程中，他们能够创建最有效的代码，并针对最佳性能进行优化。

#### 3. 节省了大量的开发时间

由于打印到屏幕、计算平方根等通用功能已经写入。你不必担心再次创建它们。

#### 4. 功能便携

随着现实世界需求的不断变化，你的应用程序有望随时随地工作。而且，这些库函数可以帮助你在每台计算机上执行相同的操作。

---



## 示例：使用 `sqrt()` 函数的平方根

假设，你想找到一个数字的平方根。

要计算一个数的平方根，可以使用 `sqrt()` 库函数。该函数定义在 `math.h` 头文件中。

```
#include <stdio.h>
#include <math.h>
int main()
{
    float num, root;
    printf("Enter a number: ");
    scanf("%f", &num);

    // Computes the square root of num and stores in root.
    root = sqrt(num);

    printf("Square root of %.2f = %.2f", num, root);
    return 0;
}
```

运行程序时，输出将为：

```
Enter a number: 12
Square root of 12.00 = 3.46
```

## 不同头文件中的库函数

C 的头文件	描述
<code>&lt;assert.h&gt;</code>	程序断言函数
<code>&lt;ctype.h&gt;</code>	字符类型函数
<code>&lt;locale.h&gt;</code>	本地化函数
<code>&lt;math.h&gt;</code>	数学函数
<code>&lt;setjmp.h&gt;</code>	跳转函数

C 的头文件	描述
<signal.h>	信号处理功能
<stdarg.h>	变量参数处理函数
<stdio.h>	标准输入/输出功能
<stdlib.h>	标准实用程序函数
<string.h>	字符串处理函数
<time.h>	日期时间函数

## 9.6 枚举

在 C 编程中，枚举类型 (也叫 enum) 是一种由整型常量组成的数据类型。要定义枚举，可以使用 `enum` 关键字。

```
enum 标识 {常量 1, 常量 2, ..., 常量 N};
```

默认情况下，常量 1 为 0, 常量 2 为 1, 以此类推。你可以在声明时更改枚举元素的默认值 (如有必要)。

```
// 修改枚举常量的默认值
enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3,
};
```

### 声明枚举类型

定义枚举类型时，将创建变量的蓝图。下面介绍如何创建枚举类型的变量。

```
enum boolean {false, true};
enum boolean check; // 声明一个枚举变量
```

这里创建了类型为 `enum boolean` 的变量 `check`。

你也可以像这样声明枚举变量。

```
enum boolean {false, true} check;
```

这里，`false` 的值等于 0, `true` 的值等于 1。

## 示例：枚举类型

```
#include <stdio.h>

enum week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

int main()
{
    // 创建枚举 week 类型的 today 变量
    enum week today;
    today = Wednesday;
    printf("Day %d",today+1);
    return 0;
}
```

输出

```
Day 4
```

## 为什么要使用枚举？

枚举变量只能采用一个值。下面是一个示例来演示它：

```
#include <stdio.h>

enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3
} card;

int main()
{
    card = club;
    printf("Size of enum variable = %d bytes", sizeof(card));
    return 0;
}
```

输出

```
Size of enum variable = 4 bytes
```

在这里，我们得到 4，因为 `int` 的大小是 4 字节。  
这使得枚举成为处理标志的不错选择。

## 如何使用枚举作为标志？

让我们举个例子：

```
enum designFlags {  
    ITALICS = 1,  
    BOLD = 2,  
    UNDERLINE = 4  
} button;
```

假设你正在为 Windows 应用程序设计一个按钮。你可以将文本的标志设置为斜体、粗体和下划线。  
在上面的伪代码中，所有整数常数都是 2 的幂是有原因的。

```
// 二进制形式  
  
ITALICS = 00000001  
BOLD = 00000010  
UNDERLINE = 00000100
```

由于整数常数是 2 的幂，因此可以使用按位 或 `|` 运算符同时组合两个或多个标志而不会重叠。这允许你一次选择两个或多个标志。例如

```
#include <stdio.h>  
  
enum designFlags {
```

```

    BOLD = 1,
    ITALICS = 2,
    UNDERLINE = 4
};

int main() {
    int myDesign = BOLD | UNDERLINE;

    //    00000001
    //    | 00000100
    //    -----
    //    00000101

    printf("%d", myDesign);

    return 0;
}

```

输出

5

当输出为 5 时，你始终知道使用了粗体和下划线。

此外，你可以根据需要添加标志。

```

if (myDesign & ITALICS) {
    // 处理斜体的代码
}

```

这里，我们在设计中添加了斜体。注意，只有斜体的代码写在 `if` 语句中。

在 C 编程中，你几乎可以完成任何操作，而无需使用枚举。但是，在某些情况下，它们可能非常方便。

## 10 测试题

### 10.1 判断奇偶

```
#include <stdio.h>

int main()
{
    int number;
    printf("请输入一个整数: ");
    scanf("%d", &number);

    // 使用模运算符 (%) 来判断奇偶性
    if (number % 2 == 0)
    {
        printf("%d 是偶数.\n", number);
    }
    else
    {
        printf("%d 是奇数.\n", number);
    }

    return 0;
}
```

这段代码是一个简单的 C 语言程序，用于判断输入的整数是奇数还是偶数。让我来解释一下它的工作原理：

1. 用户输入一个整数。
2. 程序使用模运算符 (%) 来判断这个整数是否能被 2 整除。
3. 如果余数为 0，那么这个整数是偶数；否则，它是奇数。
4. 根据判断结果，程序会输出相应的信息。

这段代码的输出结果将根据用户输入的整数而变化。例如，如果用户输入的是偶数，输出将类似于：

请输入一个整数：10  
10 是偶数。

如果用户输入的是奇数，输出将类似于：

请输入一个整数：7

7 是奇数。

这个程序很简单，但它演示了 C 语言中的基本控制流和条件判断。



## 10.2 求圆的面积

下面是一个使用 C 语言编写的程序，它可以计算圆的面积。这个程序会要求用户输入圆的半径，然后使用公式

$$A = \pi r^2$$

来计算面积，其中（ A ）是面积，（ r ）是半径，（ \pi ）是圆周率。

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

int main() {
    float radius, area;

    // 提示用户输入圆的半径
    printf("请输入圆的半径: ");
    scanf("%f", &radius);

    // 计算圆的面积
    area = PI * radius * radius;

    // 输出圆的面积
    printf("圆的面积为: %.2f\n", area);

    return 0;
}
```

在这个程序中，我们定义了圆周率（ \pi ）的近似值为 3.14159。用户输入半径后，程序会计算出面积并打印结果。如果你想要更精确的结果，可以使用 `math.h` 头文件中定义的 `M_PI` 常量来代替 3.14159。

### 函数化

```
double GetArea(double r)
{
    const PI = 3.14;
    return PI * r * r;
}
```

## 10.3 利用循环画星号

```
#include <stdio.h>

int main() {
    int i, j, rows;

    // 提示用户输入行数
    printf("请输入行数: ");
    scanf("%d", &rows);

    // 外层循环控制行数
    for (i = 1; i <= rows; i++) {
        // 内层循环控制每行打印的星号数量
        for (j = 1; j <= i; j++) {
            printf("*");
        }
        // 每打印完一行星号后换行
        printf("\n");
    }

    return 0;
}
```

这是一个C语言程序，它会逐行打印星号（\*），每行的星号数量与行数相对应。例如，第1行有1个星号，第2行有2个星号，以此类推。

当你运行这个程序时，它会首先要求你输入一个整数，表示你想打印的行数。然后，程序会根据你输入的行数打印相应数量的星号。

## 10.4 输入多个数值，求最大值

下面是一个 C 语言程序，它可以接受任意数量的输入并找出最大值：

```
#include <stdio.h>

int main() {
    int n, number;
    int max = -2147483648; // 初始化为最小的整数

    printf("请输入数字的个数: ");
    scanf("%d", &n);

    printf("请输入 %d 个数字: \n", n);
    for(int i = 0; i < n; i++) {
        scanf("%d", &number);
        if(number > max) {
            max = number;
        }
    }

    printf("最大值是: %d\n", max);
    return 0;
}
```

这个程序首先询问用户要输入多少个数字，然后接受用户输入的每个数字，并通过比较找出最大的一个。max 变量被初始化为整数可能的最小值，以确保任何输入的数字都会被考虑进去。每次输入一个新数字时，程序都会检查它是否比当前的 max 更大，如果是，就更新 max 的值。最后，程序输出最大的数字。

采用动态分配数组：

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

// 函数声明
double findMax(double* array, int size);

int main() {
    int size, i;
    double *array, max;

    printf("请输入要比较的数值数量: ");
```

```

scanf("%d", &size);

if(size > 0) {
    // 动态分配内存
    array = (double*) malloc(size * sizeof(double));
    if(array == NULL) {
        printf("内存分配失败。\\n");
        return -1; // 如果内存分配失败，则退出程序
    }

    for(i = 0; i < size; i++) {
        printf("请输入数值 #%d: ", i + 1);
        scanf("%lf", &array[i]);
    }

    // 调用函数找到最大值
    max = findMax(array, size);
    printf("最大的数值是: %.2lf\\n", max);

    // 释放内存
    free(array);
} else {
    printf("错误！数值数量必须大于零。\\n");
}

return 0;
}

// 函数定义
double findMax(double* array, int size) {
    double max = array[0];
    for(int i = 1; i < size; i++) {
        if(array[i] > max) {
            max = array[i];
        }
    }
    return max;
}

```

## 10.5 反转字符串

在 C 语言中，反转字符串是一个基本操作，可以通过多种方法实现。以下是一个使用循环来反转字符串的简单 C 程序示例：

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

int main() {
    char str[100] = "hello";
    printf("Original String: %s\n", str);

    int len = strlen(str);
    for (int i = 0, j = len - 1; i < j; i++, j--) {
        char temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }

    printf("Reversed String: %s\n", str);
    return 0;
}
```

这个程序首先定义了一个字符串 `str`，然后使用 `strlen` 函数计算字符串的长度。接着，使用一个 `for` 循环，通过交换字符串两端的字符来实现反转。循环继续进行，直到中间的字符，最终输出反转后的字符串。

除了使用循环之外，还可以使用递归、指针或库函数来实现。以下是一些不同的方法：

1. **使用递归：** 递归方法通过函数自己调用自己来实现字符串的反转。这是一个递归反转字符串的示例代码：

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

void reverse(char* str, int len, int index) {
    if (index < len / 2) {
        char temp = str[index];
        str[index] = str[len - index - 1];
        str[len - index - 1] = temp;
        reverse(str, len, index + 1);
    }
}
```

```

    }
}

int main() {
    char str[100] = "example";
    printf("Original String: %s\n", str);
    reverse(str, strlen(str), 0);
    printf("Reversed String: %s\n", str);
    return 0;
}

```

2. **使用指针：** 指针方法通过交换字符串首尾指针所指向的字符来实现反转。这是一个使用指针反转字符串的示例代码：

```

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

void stringReverse(char* str) {
    char* start = str;
    char* end = str + strlen(str) - 1;
    while (start < end) {
        char temp = *start;
        *start++ = *end;
        *end-- = temp;
    }
}

int main() {
    char str[] = "example";
    printf("Original String: %s\n", str);
    stringReverse(str);
    printf("Reversed String: %s\n", str);
    return 0;
}

```

3. 使用库函数 `strrev`: `strrev` 是一个非标准的库函数，它可以直接反转字符串。但请注意，`strrev` 函数可能不在所有编译器中都可用。

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

int main() {
    char str[100] = "example";
    printf("Original String: %s\n", str);
    printf("Reversed String: %s\n", strrev(str));
    return 0;
}
```

## 10.6 计算两个数字的和

```
#include <stdio.h>

int main() {
    int num1, num2, sum;
    printf("Enter two integers: ");
    scanf("%d %d", &num1, &num2);
    sum = num1 + num2;
    printf("Sum of %d and %d is %d\n", num1, num2, sum);
    return 0;
}
```

这段代码是一个简单的 C 语言程序，它的目的是计算两个整数的和。

- `#include <stdio.h>`: 这是一个预处理指令，它告诉编译器在编译过程中将 `stdio.h` 文件包含到程序中，这个文件包含了用于输入输出的函数。
- `int main() { ... }`: 这是程序的主函数，所有执行的代码都会在这里执行。
- `int num1, num2, sum;`: 这里声明了三个整数变量，用于存储用户输入的两个整数和它们的和。
- `printf("Enter two integers: ");`: 使用 `printf` 函数打印出提示用户输入两个整数。
- `scanf("%d %d", &num1, &num2);`: 使用 `scanf` 函数读取用户输入的两个整数，并将它们存储到变量 `num1` 和 `num2` 中。
- `sum = num1 + num2;`: 计算两个整数的和，并将结果存储到变量 `sum` 中。
- `printf("Sum of %d and %d is %d\n", num1, num2, sum);`: 使用 `printf` 函数打印出两个整数及其和。
- `return 0;`: 返回 0 表示程序正常结束。



## 10.7 最大公约数

```
#include <stdio.h>

int gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}

int main() {
    int num1, num2;
    printf("Enter two positive integers: ");
    scanf("%d %d", &num1, &num2);
    printf("GCD of %d and %d is %d\n", num1, num2, gcd(num1, num2));
    return 0;
}
```

用来计算两个正整数的最大公约数（GCD）。程序使用了递归函数 gcd 来实现这一功能。这是一个经典的算法，称为欧几里得算法，它通过连续的取余操作来找到两个数的 GCD。

程序的逻辑是正确的，当 b 等于 0 时，函数返回 a，这是因为任何数和 0 的 GCD 都是其本身。如果 b 不为 0，函数递归调用自身，参数是 b 和 a % b。

如果你想要测试这个程序，可以编译并运行它，然后根据提示输入两个正整数。程序将输出它们的最大公约数。

## 10.8 判断素数

```
#include <stdio.h>

int isPrime(int n) {
    if (n <= 1) return 0;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

int main() {
    int num;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    if (isPrime(num))
        printf("%d is a prime number.\n", num);
    else
        printf("%d is not a prime number.\n", num);
    return 0;
}
```

这段代码是一个简单的 C 语言程序，它检查用户输入的整数是否是质数。

首先，它包含了一个名为 `isPrime` 的函数，这个函数接受一个整数 `n` 作为参数，并返回一个布尔值（0 表示假，1 表示真）。这个函数通过检查 `n` 是否能被任何小于或等于平方根的整数整除来确定 `n` 是否是质数。如果找到一个能够整除 `n` 的整数，则返回 0（不是质数），否则返回 1（是质数）。

在主函数 `main` 中，程序首先声明了一个整数变量 `num`。然后，它使用 `printf` 函数提示用户输入一个正整数，并使用 `scanf` 函数读取用户输入并存储在 `num` 变量中。

接下来，程序调用了 `isPrime` 函数，传递了用户输入的整数。如果返回值为 1，则表示输入的整数是质数，程序将输出相应的信息。如果返回值为 0，则表示输入的整数不是质数，程序将输出相应的信息。

最后，程序返回 0，表示执行结束。

这段代码是一个基本的示例，展示了如何在 C 语言中定义和使用函数以及如何进行输入/输出操作。

## 10.9 生成斐波那契数列

```
#include <stdio.h>

void printFibonacci(int n) {
    int t1 = 0, t2 = 1, nextTerm;
    for (int i = 1; i <= n; ++i) {
        printf("%d, ", t1);
        nextTerm = t1 + t2;
        t1 = t2;
        t2 = nextTerm;
    }
}

int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    printf("Fibonacci Series: ");
    printFibonacci(n);
    return 0;
}
```

这段代码是一个 C 语言程序，它生成并打印出前  $n$  项的斐波那契数列。以下是代码的详细解释：

首先，它包含了一个名为 `printFibonacci` 的函数，这个函数接受一个整数  $n$  作为参数，并通过打印来输出前  $n$  项的斐波那契数列。函数中定义了三个整数变量：`t1`、`t2` 和 `nextTerm`。变量 `t1` 和 `t2` 分别表示斐波那契数列中的第一个和第二个项，而变量 `nextTerm` 用于存储下一个项的值。

在函数中，使用了一个循环来迭代生成和打印出斐波那契数列。循环从第一个项开始（即 0），然后在每次迭代中计算下一个项的值（即  $t1 + t2$ ），并将其打印出来。然后，更新变量 `s1` 和 `t2` 的值，使它们指向下一个项和之前的两个项。

在主函数 `main` 中，程序首先声明了一个整数变量  $n$ 。然后，它使用 `printf` 函数提示用户输入斐波那契数列中要生成的项数，并使用 `scanf` 函数读取用户输入并存储在  $n$  变量中。

接下来，程序调用了 `printFibonacci` 函数，传递了用户输入的项数。函数会根据这个参数生成并打印出相应数量的斐波那契数列项。

最后，程序返回 0，表示执行结束。

这段代码展示了如何在 C 语言中定义和使用函数以及如何进行输入/输出操作以及生成数学序列。

## 10.10 计算一个数的阶乘

```
#include <stdio.h>

long long factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    printf("Factorial of %d = %lld\n", num, factorial(num));
    return 0;
}
```

这段代码是一个 C 语言程序，它计算并打印出用户输入的整数的阶乘。以下是代码的详细解释：

首先，它包含了一个名为 `factorial` 的函数，这个函数接受一个整数 `n` 作为参数，并返回其阶乘。函数使用了递归来计算阶乘：如果输入的整数是 0，则返回 1（因为 0 的阶乘定义为 1）。否则，函数会调用自身，传递 `n - 1` 作为参数，并将结果乘以 `n`。

在主函数 `main` 中，程序首先声明了一个整数变量 `num`。然后，它使用 `printf` 函数提示用户输入一个整数，并使用 `scanf` 函数读取用户输入并存储在 `num` 变量中。

接下来，程序调用了 `factorial` 函数，传递了用户输入的整数。函数会计算出该整数的阶乘，并返回结果。

最后，程序使用 `printf` 函数打印出用户输入的整数和其阶乘。

这段代码展示了如何在 C 语言中定义和使用递归函数以及如何进行输入/输出操作以及计算数学表达式。

## 10.11 创建简单的计算器

```
#include <stdio.h>

int main() {
    char operator;
    double first, second;
    printf("请输入运算符 (+, -, *, /): ");
    scanf("%c", &operator);
    printf("请输入两个操作数: ");
    scanf("%lf %lf", &first, &second);

    switch (operator) {
        case '+':
            printf("%.11f + %.11f = %.11f\n", first, second, first + second);
            break;
        case '-':
            printf("%.11f - %.11f = %.11f\n", first, second, first - second);
            break;
        case '*':
            printf("%.11f * %.11f = %.11f\n", first, second, first * second);
            break;
        case '/':
            if(second != 0) {
                printf("%.11f / %.11f = %.11f\n", first, second, first / second);
            } else {
                printf("错误! 除数不能为零。\\n");
            }
            break;
        default:
            printf("错误! 运算符不正确。\\n");
    }

    return 0;
}
```

段代码是一个 C 语言程序，它执行基本的算术运算：加法、减法、乘法和除法。以下是代码的详细解释：

首先，程序声明了一个字符变量 `operator` 和两个浮点数变量 `first` 和 `second`。

然后，程序使用 `printf` 函数提示用户输入一个运算符（+、-、\*、/）并使用 `scanf` 函数读取用户输入并存储在 `operator` 变量中。

接下来，程序再次使用 `printf` 函数提示用户输入两个操作数，并使用 `scanf` 函数读取用户输入并存储在 `first` 和 `second` 变量中。

程序使用了一个 `switch` 语句来根据用户输入的运算符执行相应的算术运算。如果用户输入的运算符是加法符号（+），则计算和打印出两个操作数的和。如果是减法符号（-），则计算和打印出两个操作数的差。如果是乘法符号（\*），则计算和打印出两个操作数的乘积。如果是除法符号（/），则首先检查除数是否为零，如果不为零，则计算和打印出两个操作数的商；如果为零，则打印出错误信息。

最后，程序返回 0，表示执行结束。

这段代码展示了如何在 C 语言中定义和使用变量以及如何进行输入/输出操作以及执行基本的算术运算。



