# C

目标明确

方法可行

流程合理

激励到位

考核有效

C语言编程风格和标准：

<https://tonybai.com/2012/03/07/the-chinese-translation-of-recommended-c-style-and-coding-standards/>

## 编程资源

### 入门教程

<http://www.runoob.com/cprogramming/c-tutorial.html>

网络运行环境

<http://www.runoob.com/try/runcode.php?filename=helloworld&type=c>

### 官方文档

#### API文档函数库

<http://www.runoob.com/cprogramming/c-standard-library.html>

#### 论坛文章搜索

#### 编程风格

#### 文档生成

#### 日志模块

## ★★编程环境

### 程序特性

### 高级编程

### 跨平台

#### Windows下使用GCC

MinGw 是 Minimal GNU on Windows 的缩写，允许在 GNU/Linux 和 Windows 平台生成本地的 Windows 程序而不需要第三方运行时库。本文主要介绍 MinGw 的安装和使用。

**(一)安装**

* 1.下载 min-gw 安装程序，链接为：<http://sourceforge.net/projects/mingw/files/>，下载 mingw-get-setup.exe (86.5 kB)
* 2.运行 mingw-get-setup.exe (86.5 kB) ,点击“运行”，continue等，注意记住安装的目录，如 **C:\MinGw**,下面修改环境变量时还会用到。
* 3.修改环境变量: 选择计算机—属性---高级系统设置---环境变量，在系统变量中找到 Path 变量，在后面加入 min-gw的安装目录，如 **C:\MinGw\bin**
* 4.在开始菜单中，点击“运行”，输入 **cmd**,打开命令行:输入 **mingw-get.exe**,如果弹出 MinGw installation manager 窗口，说明安装正常。此时，关闭 MinGw installation manager 窗口，否则接下来的步骤会报错
* 5.在cmd中输入命令 **mingw-get install gcc**,等待一会，gcc 就安装成功了。

如果想安装 g++,gdb,只要输入命令 **mingw-get install g++** 和 **mingw-get install gdb**

**(二)使用**

在 cmd 的当前工作目录写 C 程序 test.c：

# include <stdio.h>

int main()

{

    printf("%s\n","hello world");

    return 0;

}

在 cmd 中输入命令 **gcc test.c**

在当前目录下会生成 a.exe 的可执行文件，在 cmd 中输入 a.exe 就可以执行程序了。

如果想调试程序，可以输入 gdb a.exe

进入 gdb 的功能，使用 gdb 常用的命令就可以调试程序了。

#### UNIX/Linux 上的安装

如果您使用的是 **Linux 或 UNIX**，请在命令行使用下面的命令来检查您的系统上是否安装了 GCC：

$ gcc –v

如果未安装 GCC，那么请按照 <http://gcc.gnu.org/install/> 上的详细说明安装 GCC。

#### Mac OS 上的安装

如果您使用的是 Mac OS X，最快捷的获取 GCC 的方法是从苹果的网站上下载 Xcode 开发环境，并按照安装说明进行安装。一旦安装上 Xcode，您就能使用 GNU 编译器。

#### 安卓下安装GCC

<http://www.runoob.com/cprogramming/c-environment-setup.html>

#### Cloud Studio

[Cloud Studio](https://studio.coding.net/) 是基于浏览器的集成式开发环境，支持绝大部分编程语言，包括 HTML5、PHP、Python、Java、Ruby、C/C++、.NET 等等，无需下载安装程序，一键切换开发环境。 [Cloud Studio](https://studio.coding.net/) 提供了完整的 Linux 环境，并且支持自定义域名指向，动态计算资源调整，可以完成各种应用的开发编译与部署。

## ★★程序结构和编译运行

### 基本程序结构

#### C 程序主要包括以下部分：

* 预处理器指令
* 函数
* 变量
* 语句 & 表达式
* 注释

#### C语言的程序构成方式

* １）c语言程序由函数构成，每个函数可以实现一个或多个功能。
* ２）一个正规程序可以有多个函数，但是有且只有一个主函数。
* ３）函数只有在被调用的时候才执行，主函数由系统调用执行。
* ４）函数的格式必须按照规范书写。
* ５）C 语言程序文件的后缀为 .c

#### Hello World示例程序

#include <stdio.h>

int main()

{

/\* 我的第一个 C 程序 \*/

printf("Hello, World! \n");

return 0;

}

**实例解析：**

* 所有的 C 语言程序都需要包含 **main()** 函数。 代码从 **main()** 函数开始执行。
* **/\* ... \*/** 用于注释说明。
* **printf()** 用于格式化输出到屏幕。**printf()** 函数在 **"stdio.h"** 头文件中声明。
* **stdio.h** 是一个头文件 (标准输入输出头文件) , **#include** 是一个预处理命令，用来引入头文件。 当编译器遇到 **printf()** 函数时，如果没有找到 **stdio.h** 头文件，会发生编译错误。
* **return 0;** 语句用于表示退出程序。
* 当是 **void main()** 时，**main()** 的返回值是空，所以可以不写或者是 **return;** 但这是以前的写法了，现在很少用 **void main()** 了，也不推荐大家这么用。写出来的目的就是给大家提个醒，别犯一些低级错误。

##### main参数

运行 **C** 程序，**main(int argc, char \*argv[])** 函数传参，**argc** 为参数个数，**argv** 是字符串数组， 下标从 **0** 开始，第一个存放的是可执行程序的文件名字，然后依次存放传入的参数，举个例子 **HelloWorld.c** ：

#include <stdio.h>

int main(int argc, char \*argv[])

{

/\* 我的第一个 C 程序 \*/

printf("可执行程序 %s ,参数个数为[%d], 运行输出：[%s]\n",argv[0],argc,argv[1]);

return 0;

}

编译 **gcc HelloWorld.c**，得到可执行程序 **a.out**，运行程序：

./a.out Hello,World!

可执行程序 ./a.out ,参数个数为[2], 运行输出：[Hello,World!]

**warning: implicitly declaring library function 'printf' with type'int (const char \*, ...)' [-Wimplicit-function-declaration]**

以上警告一般发生在没有包含头文件 **#include <stdio.h>** 就使用 **printf** 的情况

int main(){

printf("Hello, World!\n");

return 0;

}

这句话提示的意思是：**用类型“int (const char \*,...)”隐式声明了库函数printf。**

**该提示的背景知识：**因为 printf 这一类库函数太常用了，所以编译器在编译的时候，发现源文件中并没有声明 printf 这个函数就直接使用了它，那么编译器就会为 printf 函数隐式生成一个声明。你那个编译器提示你，它为printf生成的隐式声明是：**int printf(const char \*, ...)**

**该提示后果：**通常情况下，这个提示只是一个 warning，而不是 error，编译仍然可以通过。

**解决方案：**要消除这个提示，只要加上 printf 的声明就行了。printf 函数是一个库函数，它的声明在 stdio.h 文件里。所以，你只要在源文件中 **#include <stdio.h>** 就可以了。

### 大型程序结构

### 在线编译器

【在线编译器】 <http://www.compileonline.com/compile_c_online.php>

【在线编译器】 <http://www.runoob.com/try/runcode.php?filename=helloworld&type=c>

### 编译 & 执行 C# 程序

**gcc** 命令如果不指定目标文件名时默认生成的可执行文件名为 **a.out(linux)** 或 **a.exe(windows)**。

可用 **gcc [源文件名] -o [目标文件名]** 来指定目标文件路径及文件名。

例如，**windows** 系统上，**gcc hello.c -o target/hello** 会在 **target** 目录下生成 **hello.exe** 文件(Linux 系统生成 hello 可执行文件)，**target** 目录必须已存在，**[源文件名] 和 -o [目标文件名]** 的顺序可互换， **gcc -o target/hello hello.c** 依然有效。

因编译器的原因，生成的 **.exe** 文件打开时会一闪而过，从而观察不到其运行的结果，这是因为 **main()** 函数结束时，DOS 窗口会自动关闭。为了避免这个问题可在 **return 0;** 前加入 **system("pause");** 语句。

system("pause"); //暂停函数，请按任意键继续...

#### 简单的编译执行

1. 打开一个文本编辑器，添加上述代码。
2. 保存文件为 *hello.c*。
3. 打开命令提示符，进入到保存文件所在的目录。
4. 键入 *gcc hello.c*，输入回车，编译代码。
5. 如果代码中没有错误，命令提示符会跳到下一行，并生成 *a.out* 可执行文件。
6. 现在，键入 *a.out* 来执行程序。
7. 您可以看到屏幕上显示 *"Hello World"*。
8. $ gcc hello.c
9. $ ./a.out
10. Hello, World!

#### ****gcc 进行 c 语言编译分为四个步骤：****

1.预处理，生成预编译文件（.i 文件）：

gcc –E hello.c –o hello.i

2.编译，生成汇编代码（.s 文件）：

gcc –S hello.i –o hello.s

3.汇编，生成目标文件（.o 文件）：

**gcc –c hello.s –o hello.o**

4.链接，生成可执行文件：

**gcc hello.o –o hello**

有时候，进行调试，可能会用到某个步骤哦。

## ★★基础语法

### 基本语法

### C的令牌（Tokens）

C 程序由各种令牌组成，令牌可以是关键字、标识符、常量、字符串值，或者是一个符号。例如，下面的 C 语句包括五个令牌：

printf("Hello, World! \n");

这五个令牌分别是：

printf

(

"Hello, World! \n"

)

;

### 关键字

表列出了 C 中的保留字。这些保留字不能作为常量名、变量名或其他标识符名称。

|  |  |
| --- | --- |
| **关键字** | **说明** |
| auto | 声明自动变量 |
| break | 跳出当前循环 |
| case | 开关语句分支 |
| char | 声明字符型变量或函数返回值类型 |
| const | 声明只读变量 |
| continue | 结束当前循环，开始下一轮循环 |
| default | 开关语句中的"其它"分支 |
| do | 循环语句的循环体 |
| double | 声明双精度浮点型变量或函数返回值类型 |
| else | 条件语句否定分支（与 if 连用） |
| enum | 声明枚举类型 |
| extern | 声明变量或函数是在其它文件或本文件的其他位置定义 |
| float | 声明浮点型变量或函数返回值类型 |
| for | 一种循环语句 |
| goto | 无条件跳转语句 |
| if | 条件语句 |
| int | 声明整型变量或函数 |
| long | 声明长整型变量或函数返回值类型 |
| register | 声明寄存器变量 |
| return | 子程序返回语句（可以带参数，也可不带参数） |
| short | 声明短整型变量或函数 |
| signed | 声明有符号类型变量或函数 |
| sizeof | 计算数据类型或变量长度（即所占字节数） |
| static | 声明静态变量 |
| struct | 声明结构体类型 |
| switch | 用于开关语句 |
| typedef | 用以给数据类型取别名 |
| unsigned | 声明无符号类型变量或函数 |
| union | 声明共用体类型 |
| void | 声明函数无返回值或无参数，声明无类型指针 |
| volatile | 说明变量在程序执行中可被隐含地改变 |
| while | 循环语句的循环条件 |

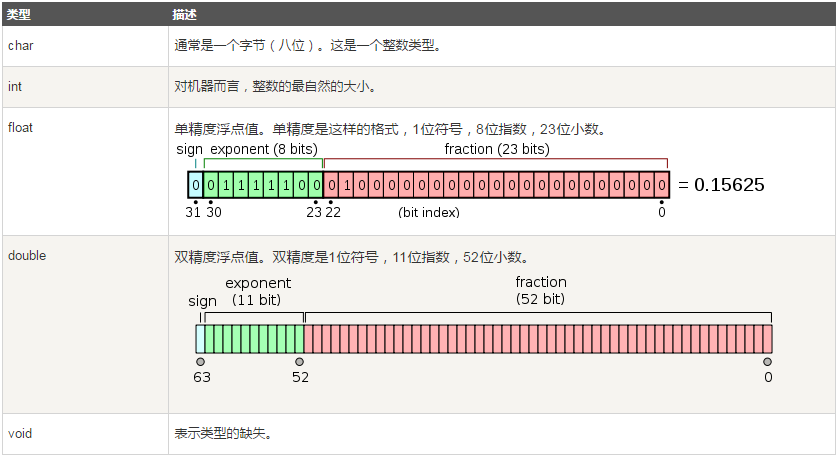
#### C99 新增关键字

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| \_Bool | \_Complex | \_Imaginary | inline | restrict |

#### C11 新增关键字

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| \_Alignas | \_Alignof | \_Atomic | \_Generic | \_Noreturn |
| \_Static\_assert | \_Thread\_local |  |  |  |

### 数据类型



标准C没有这个类型，byte表示一个字节,对应C的unsigned char，你可以自己按下面的样子定义一个即可：  
typedef unsigned char byte;

C语言种的整数：长度（所占字节数）关系为：short <= int <= long

<https://blog.csdn.net/pingdouble/article/details/50826453>

C 中的类型可分为以下几种：

|  |  |
| --- | --- |
| **序号** | **类型与描述** |
| 1 | **基本类型：** 它们是算术类型，包括两种类型：整数类型和浮点类型。 |
| 2 | **枚举类型：** 它们也是算术类型，被用来定义在程序中只能赋予其一定的离散整数值的变量。 |
| 3 | **void 类型：** 类型说明符 *void* 表明没有可用的值。 |
| 4 | **派生类型：** 它们包括：指针类型、数组类型、结构类型、共用体类型和函数类型。 |

数组类型和结构类型统称为聚合类型。函数的类型指的是函数返回值的类型。在本章节接下来的部分我们将介绍基本类型，其他几种类型会在后边几个章节中进行讲解。

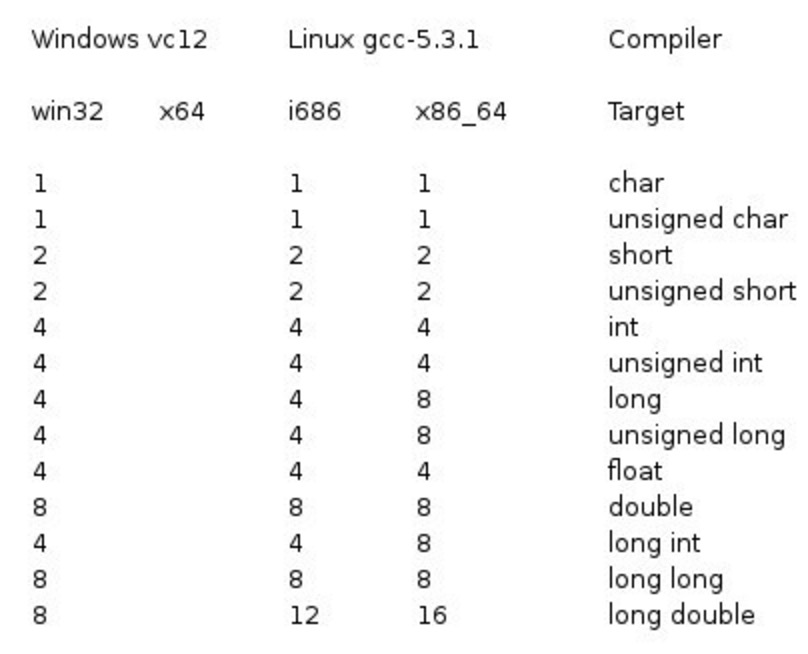
#### 整数类型

下表列出了关于标准整数类型的存储大小和值范围的细节：

|  |  |  |
| --- | --- | --- |
| **类型** | **存储大小** | **值范围** |
| char | 1 字节 | -128 到 127 或 0 到 255 |
| unsigned char | 1 字节 | 0 到 255 |
| signed char | 1 字节 | -128 到 127 |
| int | 2 或 4 字节 | -32,768 到 32,767 或 -2,147,483,648 到 2,147,483,647 |
| unsigned int | 2 或 4 字节 | 0 到 65,535 或 0 到 4,294,967,295 |
| short | 2 字节 | -32,768 到 32,767 |
| unsigned short | 2 字节 | 0 到 65,535 |
| long | 4 字节 | -2,147,483,648 到 2,147,483,647 |
| unsigned long | 4 字节 | 0 到 4,294,967,295 |

*注意，各种类型的存储大小与系统位数有关，但目前通用的以64位系统为主。*

*以下列出了32位系统与64位系统的存储大小的差别（windows 相同）：*

**

为了得到某个类型或某个变量在特定平台上的准确大小，您可以使用 **sizeof** 运算符。表达式 *sizeof(type)* 得到对象或类型的存储字节大小。下面的实例演示了获取 int 类型的大小：

#### size\_t类型

有关于 **size\_t**:

size\_t 是一种数据类型，近似于无符号整型，但容量范围一般大于 int 和 unsigned。这里使用 size\_t 是为了保证 arraysize 变量能够有足够大的容量来储存可能大的数组。

size\_t 类型在C语言标准库函数原型使用的很多，数值范围一般是要大于int和unsigned.

但凡不涉及负值范围的表示size取值的，都可以用size\_t；比如array[size\_t]。

size\_t 在stddef.h头文件中定义。

在其他常见的宏定义以及函数中常用到有：

1，sizeof运算符返回的结果是size\_t类型；

2，void \*malloc(size\_t size)...

#### 浮点类型

下表列出了关于标准浮点类型的存储大小、值范围和精度的细节：

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **存储大小** | **值范围** | **精度** |
| float | 4 字节 | 1.2E-38 到 3.4E+38 | 6 位小数 |
| double | 8 字节 | 2.3E-308 到 1.7E+308 | 15 位小数 |
| long double | 16 字节 | 3.4E-4932 到 1.1E+4932 | 19 位小数 |

头文件 float.h 定义了宏，在程序中可以使用这些值和其他有关实数二进制表示的细节。下面的实例将输出浮点类型占用的存储空间以及它的范围值：

实例

#include <stdio.h>

#include <float.h>

int main()

{

printf("float 存储最大字节数 : %lu \n", sizeof(float));

printf("float 最小值: %E\n", FLT\_MIN );

printf("float 最大值: %E\n", FLT\_MAX );

printf("精度值: %d\n", FLT\_DIG );

return 0;

}**%E** 为以指数形式输出单、双精度实数，详细说明查看 [C 库函数 - printf()](http://www.runoob.com/cprogramming/c-function-printf.html)。

当您在 Linux 上编译并执行上面的程序时，它会产生下列结果：

float 存储最大字节数 : 4

float 最小值: 1.175494E-38

float 最大值: 3.402823E+38

精度值: 6

#### void类型

void 类型指定没有可用的值。它通常用于以下三种情况下：

|  |  |
| --- | --- |
| **序号** | **类型与描述** |
| 1 | **函数返回为空** C 中有各种函数都不返回值，或者您可以说它们返回空。不返回值的函数的返回类型为空。例如 **void exit (int status);** |
| 2 | **函数参数为空** C 中有各种函数不接受任何参数。不带参数的函数可以接受一个 void。例如 **int rand(void);** |
| 3 | **指针指向 void** 类型为 void \* 的指针代表对象的地址，而不是类型。例如，内存分配函数 **void \*malloc( size\_t size );** 返回指向 void 的指针，可以转换为任何数据类型。 |

如果现在您还是无法完全理解 void 类型，不用太担心，在后续的章节中我们将会详细讲解这些概念。

#### 笔记

常用基本数据类型占用空间（64位机器为例）

 char ： 1个字节

 int ：4个字节

 float：4个字节

 double：8个字节

基本类型书写

<p**整数**

 a，默认为10进制 ，10 ，20。

 b，以0开头为8进制，045，021。

 c.，以0b开头为2进制，0b11101101。

 d，以0x开头为16进制，0x21458adf。

**小数**

单精度常量：2.3f 。

双精度常量：2.3，默认为双精度。

**字符型常量**

用英文单引号括起来，只保存一个字符'a'、'b' 、'\*' ，还有转义字符 '\n' 、'\t'。

**字符串常量**

用英文的双引号引起来 可以保存多个字符："abc"。

**</p**

#### 类型转换

1、数据类型转换：C 语言中如果一个表达式中含有不同类型的常量和变量，在计算时，会将它们自动转换为同一种类型；在 C 语言中也可以对数据类型进行强制转换；

2、自动转换规则：

 a）浮点数赋给整型，该浮点数小数被舍去；

 b）整数赋给浮点型，数值不变，但是被存储到相应的浮点型变量中；

3、强制类型转换形式: **(类型说明符)(表达式)**

实例程序：

#include<stdio.h>

int main()

{

    float f,x=3.6,y=5.2;

    int i=4,a,b;

    a=x+y;

    b=(int)(x+y);

    f=10/i;

    printf("a=%d,b=%d,f=%f,x=%f\n",a,b,f,x);

}

例中先计算 x+y 值为 8.8，然后赋值给 a，因为a为整型，所以自取整数部分8，a=8;

接下来 b 把 x+y 强制转换为整型;

最后 10/i 是两个整数相除，结果仍为整数 2，把 2 赋给浮点数 f;

x 为浮点型直接输出。

### 数据结构类型

#### 字符串

在 C 语言中，字符串实际上是使用 **null** 字符 '\0' 终止的一维字符数组。因此，一个以 null 结尾的字符串，包含了组成字符串的字符。

下面的声明和初始化创建了一个 "Hello" 字符串。由于在数组的末尾存储了空字符，所以字符数组的大小比单词 "Hello" 的字符数多一个。

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

依据数组初始化规则，您可以把上面的语句写成以下语句：

char greeting[] = "Hello";

以下是 C/C++ 中定义的字符串的内存表示：



其实，您不需要把 *null* 字符放在字符串常量的末尾。C 编译器会在初始化数组时，自动把 '\0' 放在字符串的末尾。让我们尝试输出上面的字符串：

##### 操作字符串的函数：

|  |  |
| --- | --- |
| **序号** | **函数 & 目的** |
| 1 | **strcpy(s1, s2);** 复制字符串 s2 到字符串 s1。 |
| 2 | **strcat(s1, s2);** 连接字符串 s2 到字符串 s1 的末尾。 |
| 3 | **strlen(s1);** 返回字符串 s1 的长度。 |
| 4 | **strcmp(s1, s2);** 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回小于 0；如果 s1>s2 则返回大于 0。 |
| 5 | **strchr(s1, ch);** 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。 |
| 6 | **strstr(s1, s2);** 返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。 |



本节涉及函数英文全称

strcmp: string compare

strcat: string catenate

strcpy: string copy

strlen: string length

strlwr: string lowercase

strupr: string upercase



##### strlen 与 sizeof的区别：

sizeof 是函数，sizeof 是运算操作符，二者得到的结果类型为 size\_t，即 unsigned int 类型。

sizeof 计算的是变量的大小，不受字符 **\0** 影响；

而 strlen 计算的是字符串的长度，以 **\0** 作为长度判定依据。

#include <stdio.h>

#include <string.h>

int main ()

{

char str[] = "12345";

printf("str=%s, strlen=%ld, sizeof=%ld \n", str, strlen(str), sizeof(str));

}

str=12345, strlen=5, sizeof=6

#### 数组

##### 声明数组

double balance[10];

##### 初始化数组

在 C 中，您可以逐个初始化数组，也可以使用一个初始化语句，如下所示：

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

大括号 { } 之间的值的数目不能大于我们在数组声明时在方括号 [ ] 中指定的元素数目。

如果您省略掉了数组的大小，数组的大小则为初始化时元素的个数。因此，如果：

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

您将创建一个数组，它与前一个实例中所创建的数组是完全相同的。下面是一个为数组中某个元素赋值的实例：

balance[4] = 50.0;

上述的语句把数组中第五个元素的值赋为 50.0。所有的数组都是以 0 作为它们第一个元素的索引，也被称为基索引，数组的最后一个索引是数组的总大小减去 1。

##### 多维数组

type name[size1][size2]...[sizeN];

###### 初始化二维数组

多维数组可以通过在括号内为每行指定值来进行初始化。下面是一个带有 3 行 4 列的数组。

int a[3][4] = {

{0, 1, 2, 3} , /\* 初始化索引号为 0 的行 \*/

{4, 5, 6, 7} , /\* 初始化索引号为 1 的行 \*/

{8, 9, 10, 11} /\* 初始化索引号为 2 的行 \*/

};

内部嵌套的括号是可选的，下面的初始化与上面是等同的：

int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};

##### 传递数组给函数

如果您想要在函数中传递一个一维数组作为参数，您必须以下面三种方式来声明函数形式参数，这三种声明方式的结果是一样的，因为每种方式都会告诉编译器将要接收一个整型指针。同样地，您也可以传递一个多维数组作为形式参数。

**方式 1**

形式参数是一个指针（您可以在下一章中学习到有关指针的知识）：

void myFunction(int \*param) { . . . }

**方式 2**

形式参数是一个已定义大小的数组：

void myFunction(int param[10]) { . . . }

**方式 3**

形式参数是一个未定义大小的数组：

void myFunction(int param[]) { . . . }

#include <stdio.h>

/\* 函数声明 \*/

double getAverage(int arr[], int size);

int main ()

{

/\* 带有 5 个元素的整型数组 \*/

int balance[5] = {1000, 2, 3, 17, 50};

double avg;

/\* 传递一个指向数组的指针作为参数 \*/

avg = getAverage( balance, 5 ) ;

/\* 输出返回值 \*/

printf( "平均值是： %f ", avg );

return 0;

}

double getAverage(int arr[], int size)

{

int i;

double avg;

double sum=0;

for (i = 0; i < size; ++i)

{

sum += arr[i];

}

avg = sum / size;

return avg;

}

当上面的代码被编译和执行时，它会产生下列结果：

##### 从函数返回数组

C 语言不允许返回一个完整的数组作为函数的参数。但是，您可以通过指定不带索引的数组名来返回一个指向数组的指针。

如果您想要从函数返回一个一维数组，您必须声明一个返回指针的函数，如下：

int \* myFunction() { . . . }

另外，C 不支持在函数外返回局部变量的地址，除非定义局部变量为 **static** 变量。

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

/\* 要生成和返回随机数的函数 \*/

int \* getRandom( )

{

static int r[10];

int i;

/\* 设置种子 \*/

srand( (unsigned)time( NULL ) );

for ( i = 0; i < 10; ++i)

{

r[i] = rand();

printf( "r[%d] = %d\n", i, r[i]);

}

return r;

}

/\* 要调用上面定义函数的主函数 \*/

int main ()

{

/\* 一个指向整数的指针 \*/

int \*p;

int i;

p = getRandom();

for ( i = 0; i < 10; i++ )

{

printf( "\*(p + %d) : %d\n", i, \*(p + i));

}

return 0;

}

##### 指向数组的指针

数组名是一个指向数组中第一个元素的常量指针。因此，在下面的声明中：

double balance[50];

**balance** 是一个指向 &balance[0] 的指针，即数组 balance 的第一个元素的地址。因此，下面的程序片段把 **p** 赋值为 **balance** 的第一个元素的地址：

double \*p;

double balance[10];

p = balance;

使用数组名作为常量指针是合法的，反之亦然。因此，\*(balance + 4) 是一种访问 balance[4] 数据的合法方式。

一旦您把第一个元素的地址存储在 p 中，您就可以使用 \*p、\*(p+1)、\*(p+2) 等来访问数组元素。下面的实例演示了上面讨论到的这些概念：

#include <stdio.h>

int main ()

{

/\* 带有 5 个元素的整型数组 \*/

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

double \*p;

int i;

p = balance;

/\* 输出数组中每个元素的值 \*/

printf( "使用指针的数组值\n");

for ( i = 0; i < 5; i++ )

{

printf("\*(p + %d) : %f\n", i, \*(p + i) );

}

printf( "使用 balance 作为地址的数组值\n");

for ( i = 0; i < 5; i++ )

{

printf("\*(balance + %d) : %f\n", i, \*(balance + i) );

}

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

使用指针的数组值

\*(p + 0) : 1000.000000

\*(p + 1) : 2.000000

\*(p + 2) : 3.400000

\*(p + 3) : 17.000000

\*(p + 4) : 50.000000

使用 balance 作为地址的数组值

\*(balance + 0) : 1000.000000

\*(balance + 1) : 2.000000

\*(balance + 2) : 3.400000

\*(balance + 3) : 17.000000

\*(balance + 4) : 50.000000

在上面的实例中，p 是一个指向 double 型的指针，这意味着它可以存储一个 double 类型的变量。一旦我们有了 p 中的地址，**\*p** 将给出存储在 p 中相应地址的值，正如上面实例中所演示的。

#### 枚举enum

枚举是 C 语言中的一种基本数据类型，它可以让数据更简洁，更易读。

枚举语法定义格式为：

enum　枚举名　{枚举元素1,枚举元素2,……};

接下来我们举个例子，比如：一星期有 7 天，如果不用枚举，我们需要使用 #define 来为每个整数定义一个别名：

#define MON 1 #define TUE 2 #define WED 3 #define THU 4 #define FRI 5 #define SAT 6 #define SUN 7

这个看起来代码量就比较多，接下来我们看看使用枚举的方式：

enum DAY

{

MON=1, TUE, WED, THU, FRI, SAT, SUN

};

这样看起来是不是更简洁了。

**注意：**第一个枚举成员的默认值为整型的 0，后续枚举成员的值在前一个成员上加 1。我们在这个实例中把第一个枚举成员的值定义为 1，第二个就为 2，以此类推。

*可以在定义枚举类型时改变枚举元素的值：*

enum season {spring, summer=3, autumn, winter};

*没有指定值的枚举元素，其值为前一元素加 1。也就说 spring 的值为 0，summer 的值为 3，autumn 的值为 4，winter 的值为 5*

##### 枚举变量的定义

前面我们只是声明了枚举类型，接下来我们看看如何定义枚举变量。

我们可以通过以下三种方式来定义枚举变量

**1、先定义枚举类型，再定义枚举变量**

enum DAY

{

MON=1, TUE, WED, THU, FRI, SAT, SUN

};

enum DAY day;

**2、定义枚举类型的同时定义枚举变量**

enum DAY

{

MON=1, TUE, WED, THU, FRI, SAT, SUN

} day;

**3、省略枚举名称，直接定义枚举变量**

enum

{

MON=1, TUE, WED, THU, FRI, SAT, SUN

} day;

#include<stdio.h>

enum DAY

{

MON=1, TUE, WED, THU, FRI, SAT, SUN

};

int main()

{

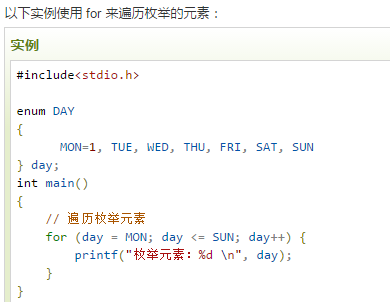
enum DAY day;

day = WED;

printf("%d",day);

return 0;

}



枚举在 switch 中的使用：

#include <stdio.h>

#include <stdlib.h>

int main()

{

enum color { red=1, green, blue };

enum color favorite\_color;

/\* ask user to choose color \*/

printf("请输入你喜欢的颜色: (1. red, 2. green, 3. blue): ");

scanf("%d", &favorite\_color);

/\* 输出结果 \*/

switch (favorite\_color)

{

case red:

printf("你喜欢的颜色是红色");

break;

case green:

printf("你喜欢的颜色是绿色");

break;

case blue:

printf("你喜欢的颜色是蓝色");

break;

default:

printf("你没有选择你喜欢的颜色");

}

return 0;

}

##### 将整数转换为枚举

以下实例将整数转换为枚举：

#include <stdio.h>

#include <stdlib.h>

int main()

{

enum day

{

saturday,

sunday,

monday,

tuesday,

wednesday,

thursday,

friday

} workday;

int a = 1;

enum day weekend;

weekend = ( enum day ) a; //类型转换

//weekend = a; //错误

printf("weekend:%d",weekend);

return 0;

}

#### 指针

通过指针，可以简化一些 C 编程任务的执行，还有一些任务，如动态内存分配，没有指针是无法执行的。所以，想要成为一名优秀的 C 程序员，学习指针是很有必要的。

正如您所知道的，每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。请看下面的实例，它将输出定义的变量地址：

#include <stdio.h>

int main ()

{

int var1;

char var2[10];

printf("var1 变量的地址： %p\n", &var1 );

printf("var2 变量的地址： %p\n", &var2 );

return 0;

}

##### 什么是指针？

**指针**是一个变量，其值为另一个变量的地址，即，内存位置的直接地址。就像其他变量或常量一样，您必须在使用指针存储其他变量地址之前，对其进行声明。指针变量声明的一般形式为：

type \*var-name;

在这里，**type** 是指针的基类型，它必须是一个有效的 C 数据类型，**var-name** 是指针变量的名称。

用来声明指针的星号 \* 与乘法中使用的星号是相同的。但是，在这个语句中，星号是用来指定一个变量是指针。以下是有效的指针声明：

int \*ip; /\* 一个整型的指针 \*/

double \*dp; /\* 一个 double 型的指针 \*/

float \*fp; /\* 一个浮点型的指针 \*/

char \*ch; /\* 一个字符型的指针 \*/

所有指针的值的实际数据类型，不管是整型、浮点型、字符型，还是其他的数据类型，都是一样的，都是一个代表内存地址的长的十六进制数。不同数据类型的指针之间唯一的不同是，指针所指向的变量或常量的数据类型不同。

##### 如何使用指针？

使用指针时会频繁进行以下几个操作：定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些是通过使用一元运算符 **\*** 来返回位于操作数所指定地址的变量的值。下面的实例涉及到了这些操作：

#include <stdio.h>

int main ()

{

int var = 20; /\* 实际变量的声明 \*/

int \*ip; /\* 指针变量的声明 \*/

ip = &var; /\* 在指针变量中存储 var 的地址 \*/

printf("Address of var variable: %p\n", &var );

/\* 在指针变量中存储的地址 \*/

printf("Address stored in ip variable: %p\n", ip );

/\* 使用指针访问值 \*/

printf("Value of \*ip variable: %d\n", \*ip );

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Address of var variable: bffd8b3c

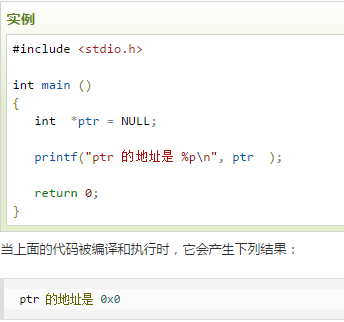
Address stored in ip variable: bffd8b3c

Value of \*ip variable: 20

##### C 中的 NULL 指针

在变量声明的时候，如果没有确切的地址可以赋值，为指针变量赋一个 NULL 值是一个良好的编程习惯。赋为 NULL 值的指针被称为**空**指针。

NULL 指针是一个定义在标准库中的值为零的常量。请看下面的程序：



在大多数的操作系统上，程序不允许访问地址为 0 的内存，因为该内存是操作系统保留的。然而，内存地址 0 有特别重要的意义，它表明该指针不指向一个可访问的内存位置。但按照惯例，如果指针包含空值（零值），则假定它不指向任何东西。

如需检查一个空指针，您可以使用 if 语句，如下所示：

if(ptr) /\* 如果 p 非空，则完成 \*/

if(!ptr) /\* 如果 p 为空，则完成 \*/

##### 指针的算术运算

C 指针是一个用数值表示的地址。因此，您可以对指针执行算术运算。可以对指针进行四种算术运算：++、--、+、-。

在执行完上述的运算之后，**ptr** 将指向位置 1004，因为 ptr 每增加一次，它都将指向下一个整数位置，即当前位置往后移 4 个字节。这个运算会在不影响内存位置中实际值的情况下，移动指针到下一个内存位置。如果 **ptr** 指向一个地址为 1000 的字符，上面的运算会导致指针指向位置 1001，因为下一个字符位置是在 1001。

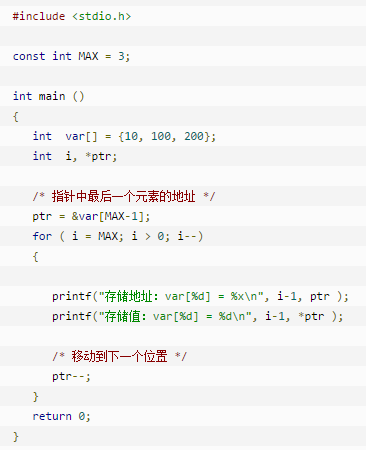
###### 递增一个指针

我们喜欢在程序中使用指针代替数组，因为变量指针可以递增，而数组不能递增，因为数组是一个常量指针。下面的程序递增变量指针，以便顺序访问数组中的每一个元素：



###### 递减一个指针

同样地，对指针进行递减运算，即把值减去其数据类型的字节数，如下所示：



###### 指针的比较

指针可以用关系运算符进行比较，如 ==、< 和 >。如果 p1 和 p2 指向两个相关的变量，比如同一个数组中的不同元素，则可对 p1 和 p2 进行大小比较。

下面的程序修改了上面的实例，只要变量指针所指向的地址小于或等于数组的最后一个元素的地址 &var[MAX - 1]，则把变量指针进行递增：

#include <stdio.h>

const int MAX = 3;

int main ()

{

int var[] = {10, 100, 200};

int i, \*ptr;

/\* 指针中第一个元素的地址 \*/

ptr = var;

i = 0;

while ( ptr <= &var[MAX - 1] )

{

printf("Address of var[%d] = %x\n", i, ptr );

printf("Value of var[%d] = %d\n", i, \*ptr );

/\* 指向上一个位置 \*/

ptr++;

i++;

}

return 0;

}

上面的代码被编译和执行时，它会产生下列结果：

Address of var[0] = bfdbcb20

Value of var[0] = 10

Address of var[1] = bfdbcb24

Value of var[1] = 100

Address of var[2] = bfdbcb28

Value of var[2] = 200

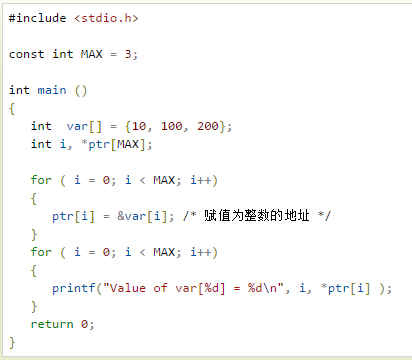
##### 指针数组

int var[] = {10, 100, 200};

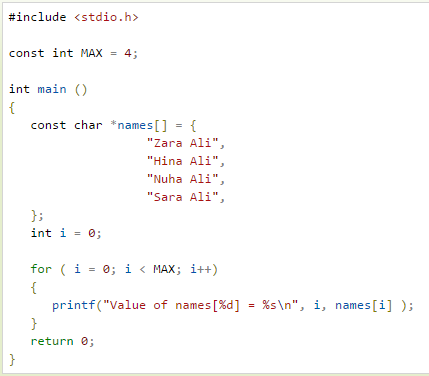
可能有一种情况，我们想要让数组存储指向 int 或 char 或其他数据类型的指针。下面是一个指向整数的指针数组的声明：

int \*ptr[MAX];

在这里，把 **ptr** 声明为一个数组，由 MAX 个整数指针组成。因此，ptr 中的每个元素，都是一个指向 int 值的指针。下面的实例用到了三个整数，它们将存储在一个指针数组中，如下所示：



您也可以用一个指向字符的指针数组来存储一个字符串列表，如下：



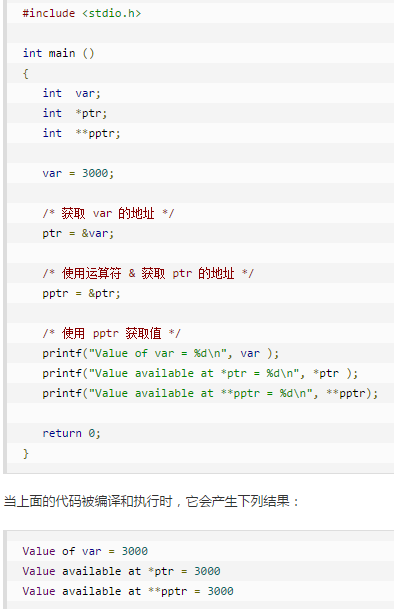
##### 指向指针的指针

指向指针的指针是一种多级间接寻址的形式，或者说是一个指针链。通常，一个指针包含一个变量的地址。当我们定义一个指向指针的指针时，第一个指针包含了第二个指针的地址，第二个指针指向包含实际值的位置。

一个指向指针的指针变量必须如下声明，即在变量名前放置两个星号。例如，下面声明了一个指向 int 类型指针的指针：

int \*\*var;

当一个目标值被一个指针间接指向到另一个指针时，访问这个值需要使用两个星号运算符，如下面实例所示：



##### 传递指针给函数

C 语言允许您传递指针给函数，只需要简单地声明函数参数为指针类型即可。

下面的实例中，我们传递一个无符号的 long 型指针给函数，并在函数内改变这个值：



**能接受指针作为参数的函数，也能接受数组作为参数**，如下所示：

#include <stdio.h>

/\* 函数声明 \*/

double getAverage(int \*arr, int size);

int main ()

{

/\* 带有 5 个元素的整型数组 \*/

int balance[5] = {1000, 2, 3, 17, 50};

double avg;

/\* 传递一个指向数组的指针作为参数 \*/

avg = getAverage( balance, 5 ) ;

/\* 输出返回值 \*/

printf("Average value is: %f\n", avg );

return 0;

}

double getAverage(int \*arr, int size)

{

int i, sum = 0;

double avg;

for (i = 0; i < size; ++i)

{

sum += arr[i];

}

avg = (double)sum / size;

return avg;

}

##### 从函数返回指针

我们已经了解了 C 语言中如何从函数返回数组，类似地，C 允许您从函数返回指针。为了做到这点，您必须声明一个返回指针的函数，如下所示：

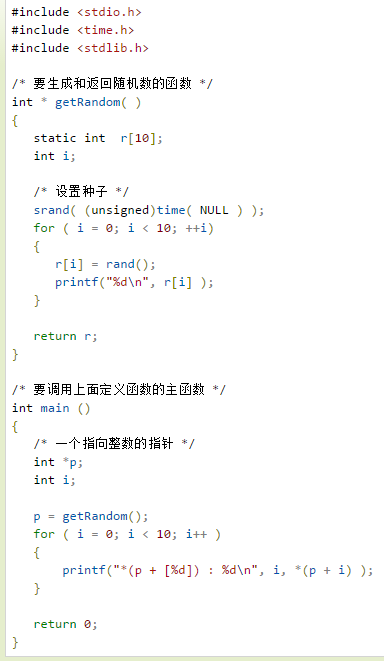
int \* myFunction()

{

}

另外，C 语言不支持在调用函数时返回局部变量的地址，除非定义局部变量为 **static** 变量。

现在，让我们来看下面的函数，它会生成 10 个随机数，并使用表示指针的数组名（即第一个数组元素的地址）来返回它们，具体如下：



#### C函数指针和回调函数

<http://www.runoob.com/cprogramming/c-fun-pointer-callback.html>

函数指针是指向函数的指针变量。

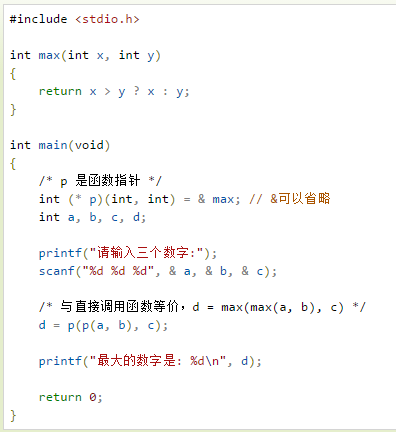
通常我们说的指针变量是指向一个整型、字符型或数组等变量，而函数指针是指向函数。

函数指针可以像一般函数一样，用于调用函数、传递参数。

函数指针变量的声明：

typedef int (\*fun\_ptr)(int,int); // 声明一个指向同样参数、返回值的函数指针类型

以下实例声明了函数指针变量 p，指向函数 max：



/\* p 是函数指针 \*/

int (\* p)(int, int) = & max; // &可以省略

int (\* p)(int, int) = max; // &可以省略

/\* 与直接调用函数等价，d = max(max(a, b), c) \*/

d = p(p(a, b), c);

##### 回调函数

###### 函数指针作为某个函数的参数

函数指针变量可以作为某个函数的参数来使用的，回调函数就是一个通过函数指针调用的函数。

简单讲：回调函数是由别人的函数执行时调用你实现的函数。

*以下是自知乎作者常溪玲的解说：*

*你到一个商店买东西，刚好你要的东西没有货，于是你在店员那里留下了你的电话，过了几天店里有货了，店员就打了你的电话，然后你接到电话后就到店里去取了货。在这个例子里，你的电话号码就叫回调函数，你把电话留给店员就叫登记回调函数，店里后来有货了叫做触发了回调关联的事件，店员给你打电话叫做调用回调函数，你到店里去取货叫做响应回调事件。*

###### 实例

实例中 populate\_array 函数定义了三个参数，其中第三个参数是函数的指针，通过该函数来设置数组的值。

实例中我们定义了回调函数 getNextRandomValue，它返回一个随机值，它作为一个函数指针传递给 populate\_array 函数。

populate\_array 将调用 10 次回调函数，并将回调函数的返回值赋值给数组。



#### 结构体

C 数组允许定义可存储相同类型数据项的变量，**结构**是 C 编程中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。

结构用于表示一条记录，假设您想要跟踪图书馆中书本的动态，您可能需要跟踪每本书的下列属性：

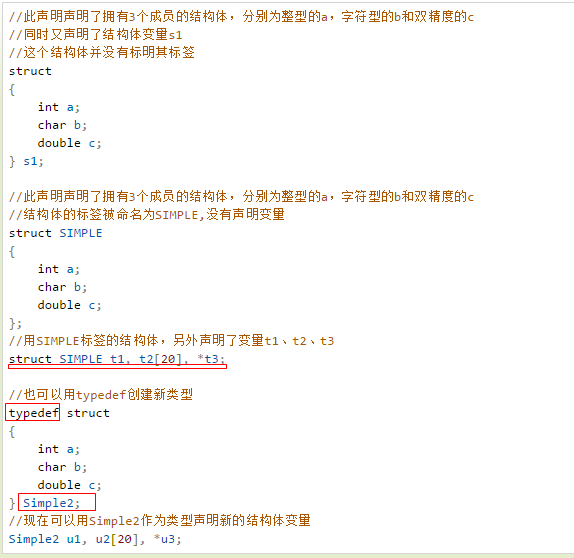
* Title
* Author
* Subject
* Book ID

##### 定义结构

为了定义结构，您必须使用 **struct** 语句。struct 语句定义了一个包含多个成员的新的数据类型，struct 语句的格式如下：

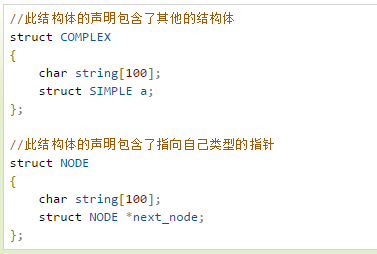


在一般情况下，**tag、member-list、variable-list** 这 3 部分至少要出现 2 个。以下为实例：



在上面的声明中，第一个和第二声明被编译器当作两个完全不同的类型，即使他们的成员列表是一样的，如果令 t3=&s1，则是非法的。

结构体的成员可以包含其他结构体，也可以包含指向自己结构体类型的指针，而通常这种指针的应用是为了实现一些更高级的数据结构如链表和树等。



如果两个结构体互相包含，则需要对其中一个结构体进行不完整声明，如下所示：

struct B; //对结构体B进行不完整声明

//结构体A中包含指向结构体B的指针

struct A {

struct B \*partner;

//other members;

};

//结构体B中包含指向结构体A的指针，在A声明完后，B也随之进行声明 struct B { struct A \*partner; //other members; };

##### 访问结构成员

为了访问结构的成员，我们使用**成员访问运算符（.）**。成员访问运算符是结构变量名称和我们要访问的结构成员之间的一个句号。您可以使用 **struct** 关键字来定义结构类型的变量。

下面的实例演示了结构的用法：



##### 结构作为函数参数

您可以把结构作为函数参数，传参方式与其他类型的变量或指针类似。您可以使用上面实例中的方式来访问结构变量：

#include <stdio.h>

#include <string.h>

struct Books

{

char title[50];

char author[50];

char subject[100];

int book\_id;

};

/\* 函数声明 \*/

void printBook( struct Books book );

int main( )

{

struct Books Book1; /\* 声明 Book1，类型为 Books \*/

struct Books Book2; /\* 声明 Book2，类型为 Books \*/

/\* Book1 详述 \*/

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

/\* Book2 详述 \*/

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Zara Ali");

strcpy( Book2.subject, "Telecom Billing Tutorial");

Book2.book\_id = 6495700;

/\* 输出 Book1 信息 \*/

printBook( Book1 );

/\* 输出 Book2 信息 \*/

printBook( Book2 );

return 0;

}

void printBook( struct Books book )

{

printf( "Book title : %s\n", book.title);

printf( "Book author : %s\n", book.author);

printf( "Book subject : %s\n", book.subject);

printf( "Book book\_id : %d\n", book.book\_id);

}

##### 指向结构的指针

您可以定义指向结构的指针，方式与定义指向其他类型变量的指针相似，如下所示：

struct Books \*struct\_pointer;

现在，您可以在上述定义的指针变量中存储结构变量的地址。为了查找结构变量的地址，请把 & 运算符放在结构名称的前面，如下所示：

struct\_pointer = &Book1;

为了使用指向该结构的指针访问结构的成员，您必须使用 -> 运算符，如下所示：

struct\_pointer->title;

让我们使用结构指针来重写上面的实例，这将有助于您理解结构指针的概念：

#include <stdio.h>

#include <string.h>

struct Books

{

char title[50];

char author[50];

char subject[100];

int book\_id;

};

/\* 函数声明 \*/

void printBook( struct Books \*book );

int main( )

{

struct Books Book1; /\* 声明 Book1，类型为 Books \*/

struct Books Book2; /\* 声明 Book2，类型为 Books \*/

/\* Book1 详述 \*/

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

/\* Book2 详述 \*/

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Zara Ali");

strcpy( Book2.subject, "Telecom Billing Tutorial");

Book2.book\_id = 6495700;

/\* 通过传 Book1 的地址来输出 Book1 信息 \*/

printBook( &Book1 );

/\* 通过传 Book2 的地址来输出 Book2 信息 \*/

printBook( &Book2 );

return 0;

}

void printBook( struct Books \*book )

{

printf( "Book title : %s\n", book->title);

printf( "Book author : %s\n", book->author);

printf( "Book subject : %s\n", book->subject);

printf( "Book book\_id : %d\n", book->book\_id);

}

##### 结构体分配空间

结构体中成员变量分配的空间是按照成员变量中占用空间最大的来作为分配单位,同样成员变量的存储空间也是不能跨分配单位的,如果当前的空间不足,则会存储到下一个分配单位中。

#include <stdio.h>

typedef struct

{

unsigned char a;

unsigned int b;

unsigned char c;

} debug\_size1\_t;

typedef struct

{

unsigned char a;

unsigned char b;

unsigned int c;

} debug\_size2\_t;

int main(void)

{

printf("debug\_size1\_t size=%lu,debug\_size2\_t size=%lu\r\n", sizeof(debug\_size1\_t), sizeof(debug\_size2\_t));

return 0;

}

编译执行输出结果：

debug\_size1\_t size=12,debug\_size2\_t size=8

结构体占用存储空间,以32位机为例

* 1.debug\_size1\_t 存储空间分布为a(1byte)+空闲(3byte)+b(4byte)+c(1byte)+空闲(3byte)=12(byte)。
* 1.debug\_size2\_t 存储空间分布为a(1byte)+b(1byte)+空闲(2byte)+c(4byte)=8(byte)。

##### 结构体数组

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAXTITL 41

#define MAXAUTL 31

#define MAVXBKS 100

char \* s\_gets(char \* st, int n);

struct book {

char title[MAXTITL];

char author[MAXAUTL];

float value;

};

int main()

{

struct book library[MAVXBKS]; //book类型的结构体数组

int i;

int index;

printf("请问你要录入多少本书的信息\n");

do

{

scanf("%d", &index);

} while (index > MAVXBKS);

getchar();

for (i = 0; i < index; i++)

{

printf("请输入第%d本书的名称:\n",i+1);

s\_gets(library[i].title, MAXTITL);

printf("输入其作者的名字:\n");

s\_gets(library[i].author, MAXAUTL);

printf("请输入书本的价格:\n");

scanf("%f", &library[i].value);

getchar();

}

for (i = 0; i < index; i++)

{

printf("%d\t%s 是 %s 写的 定价为%f元\n", i,library[i].title, library[i].author, library[i].value);

}

system("pause");

return 0;

}

char \* s\_gets(char \* st, int n)

{

char \* ret\_val;

char \* find;

ret\_val = fgets(st, n, stdin);

if (ret\_val)

{

find = strchr(st, '\n'); //查找换行符

if (find) // 查找地址不为空

\*find = '\0'; //在此处放入一个空字符

else

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

continue; //处理剩余字符

}

return ret\_val;

}

#### ★位域

有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用 1 位二进位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为"位域"或"位段"。

所谓"位域"是把一个字节中的二进位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。

典型的实例：

* 用 1 位二进位存放一个开关量时，只有 0 和 1 两种状态。
* 读取外部文件格式——可以读取非标准的文件格式。例如：9 位的整数。

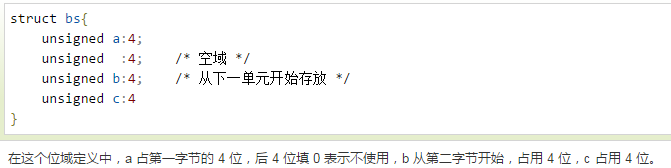
##### 位域的定义和位域变量的说明

位域定义与结构定义相仿，其形式为：**位域的长度不能大于一个字节的长度。**

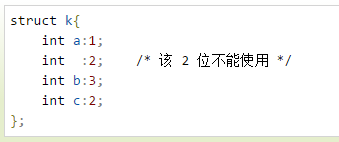


##### 对于位域的定义尚有以下几点说明：

* 一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：



* 由于位域不允许跨两个字节，因此**位域的长度不能大于一个字节的长度**，也就是说不能超过8位二进位。如果最大长度大于计算机的整数字长，一些编译器可能会允许域的内存重叠，另外一些编译器可能会把大于一个域的部分存储在下一个字中。
* 位域可以是无名位域，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：



从以上分析可以看出，位域在本质上就是一种结构类型，不过其成员是按二进位分配的。

##### 位域的使用

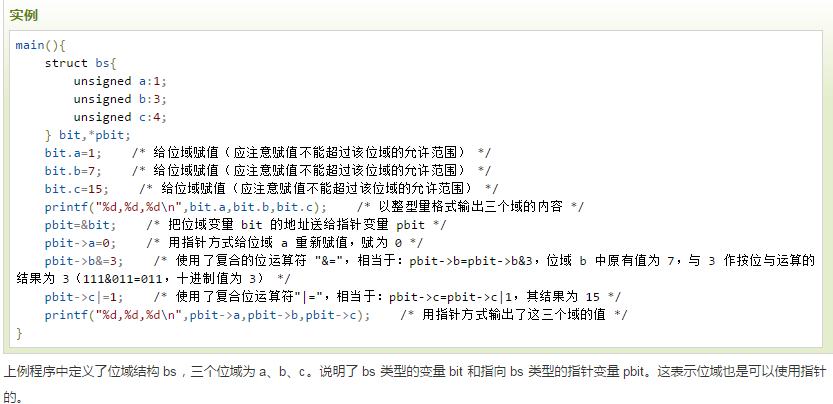
位域的使用和结构成员的使用相同，其一般形式为：

位域变量名.位域名

位域变量名->位域名

位域允许用各种格式输出。

请看下面的实例：



#### 公用体

**共用体**是一种特殊的数据类型，允许您在相同的内存位置存储不同的数据类型。您可以定义一个带有多成员的共用体，但是任何时候只能有一个成员带有值。共用体提供了一种使用相同的内存位置的有效方式。

##### 定义共用体

为了定义共用体，您必须使用 **union** 语句，方式与定义结构类似。union 语句定义了一个新的数据类型，带有多个成员。union 语句的格式如下：

union [union tag]

{

member definition;

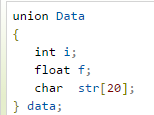
member definition;

...

member definition;

} [one or more union variables];

**union tag** 是可选的，每个 member definition 是标准的变量定义，比如 int i; 或者 float f; 或者其他有效的变量定义。在共用体定义的末尾，最后一个分号之前，您可以指定一个或多个共用体变量，这是可选的。下面定义一个名为 Data 的共用体类型，有三个成员 i、f 和 str：



现在，**Data** 类型的变量可以存储一个整数、一个浮点数，或者一个字符串。这意味着一个变量（相同的内存位置）可以存储多个多种类型的数据。您可以根据需要在一个共用体内使用任何内置的或者用户自定义的数据类型。

共用体占用的内存应足够存储共用体中最大的成员。例如，在上面的实例中，Data 将占用 20 个字节的内存空间，因为在各个成员中，字符串所占用的空间是最大的。下面的实例将显示上面的共用体占用的总内存大小：

#include <stdio.h>

#include <string.h>

union Data

{

int i;

float f;

char str[20];

};

int main( )

{

union Data data;

printf( "Memory size occupied by data : %d\n", sizeof(data));

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Memory size occupied by data : 20

##### 访问共用体成员

为了访问共用体的成员，我们使用**成员访问运算符（.）**。成员访问运算符是共用体变量名称和我们要访问的共用体成员之间的一个句号。您可以使用 **union** 关键字来定义共用体类型的变量。下面的实例演示了共用体的用法：

在这里，我们可以看到共用体的 **i** 和 **f** 成员的值有损坏，因为最后赋给变量的值占用了内存位置，这也是 **str** 成员能够完好输出的原因。现在让我们再来看一个相同的实例，这次我们在同一时间只使用一个变量，这也演示了使用共用体的主要目的：

#include <stdio.h>

#include <string.h>

union Data

{

int i;

float f;

char str[20];

};

int main( )

{

union Data data;

data.i = 10;

printf( "data.i : %d\n", data.i);

data.f = 220.5;

printf( "data.f : %f\n", data.f);

strcpy( data.str, "C Programming");

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

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

data.i : 10

data.f : 220.500000

data.str : C Programming

在这里，所有的成员都能完好输出，因为同一时间只用到一个成员。

##### 共用体作用

节省内存，有两个很长的数据结构，不会同时使用，比如一个表示老师，一个表示学生，如果要统计教师和学生的情况用结构体的话就有点浪费了！用共用体的话，只占用最长的那个数据结构所占用的空间，就足够了！

##### 共用体应用场景

通信中的数据包会用到共用体:因为不知道对方会发一个什么包过来，用共用体的话就很简单了，定义几种格式的包，收到包之后就可以直接根据包的格式取出数据。

union Data{

int i;

float f;

char str[9];

double d;

}data;

共用体所占内存**并非是**9个char，即9个字节，**而是double的两倍，即16个字节，**而且每次输出都是它前面离它最近的值。

##### 共用体占用内存分析

看了前面大家的内存占用大小计算，都没有分析到点，下面给出部分概念：

* **位：**"位(bit)"是电子计算机中最小的数据单位。每一位的状态只能是0或1。
* **字节：**8个二进制位构成1个"字节(Byte)"，它是存储空间的基本计量单位。1个字节可以储存1个英文字母或者半个汉字，换句话说，1个汉字占据2个字节的存储空间。
* **字：**"字"由若干个字节构成，字的位数叫做字长，不同档次的机器有不同的字长。例如一台8位机，它的1个字就等于1个字节，字长为8位。如果是一台16位机，那么，它的1个字就由2个字节构成，字长为16位。字是计算机进行数据处理和运算的单位。
* 一般的计算机都已经到了64位机  也就是说 一个基本单位就是64位，也就是8字节了。这样再综合上面的分析就不难看出，结构体，共用体，位域的定义中，按顺序分配内存，下一个字段所占大小如果超出了上一个字段占的内存单元剩余部分，那么它会重新申请下一个内存单元，而上一个多出部分将空着。

#### C 位域

<http://www.runoob.com/cprogramming/c-bit-fields.html>

如果程序的结构中包含多个开关量，只有 TRUE/FALSE 变量，如下：

struct

{

unsigned int widthValidated;

unsigned int heightValidated;

} status;

这种结构需要 8 字节的内存空间，但在实际上，在每个变量中，我们只存储 0 或 1。在这种情况下，C 语言提供了一种更好的利用内存空间的方式。如果您在结构内使用这样的变量，您可以定义变量的宽度来告诉编译器，您将只使用这些字节。例如，上面的结构可以重写成：

struct

{

unsigned int widthValidated : 1;

unsigned int heightValidated : 1;

} status;

现在，上面的结构中，status 变量将占用 4 个字节的内存空间，但是只有 2 位被用来存储值。如果您用了 32 个变量，每一个变量宽度为 1 位，那么 status 结构将使用 4 个字节，但只要您再多用一个变量，如果使用了 33 个变量，那么它将分配内存的下一段来存储第 33 个变量，这个时候就开始使用 8 个字节。让我们看看下面的实例来理解这个概念：

#include <stdio.h>

#include <string.h>

/\* 定义简单的结构 \*/

struct

{

unsigned int widthValidated;

unsigned int heightValidated;

} status1;

/\* 定义位域结构 \*/

struct

{

unsigned int widthValidated : 1;

unsigned int heightValidated : 1;

} status2;

int main( )

{

printf( "Memory size occupied by status1 : %d\n", sizeof(status1));

printf( "Memory size occupied by status2 : %d\n", sizeof(status2));

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Memory size occupied by status1 : 8

Memory size occupied by status2 : 4

##### 位域声明

在结构内声明位域的形式如下：

struct

{

type [member\_name] : width ;

};

下面是有关位域中变量元素的描述：

|  |  |
| --- | --- |
| **元素** | **描述** |
| type | 整数类型，决定了如何解释位域的值。类型可以是整型、有符号整型、无符号整型。 |
| member\_name | 位域的名称。 |
| width | 位域中位的数量。宽度必须小于或等于指定类型的位宽度。 |

带有预定义宽度的变量被称为**位域**。位域可以存储多于 1 位的数，例如，需要一个变量来存储从 0 到 7 的值，您可以定义一个宽度为 3 位的位域，如下：

struct

{

unsigned int age : 3;

} Age;

上面的结构定义指示 C 编译器，age 变量将只使用 3 位来存储这个值，如果您试图使用超过 3 位，则无法完成。让我们来看下面的实例：

#include <stdio.h>

#include <string.h>

struct

{

unsigned int age : 3;

} Age;

int main( )

{

Age.age = 4;

printf( "Sizeof( Age ) : %d\n", sizeof(Age) );

printf( "Age.age : %d\n", Age.age );

Age.age = 7;

printf( "Age.age : %d\n", Age.age );

Age.age = 8; // 二进制表示为 1000 有四位，超出

printf( "Age.age : %d\n", Age.age );

return 0;

}

**如果超出范围，则直接丢掉了，存不进去。**

当上面的代码被编译时，它会带有警告，当上面的代码被执行时，它会产生下列结果：

Sizeof( Age ) : 4

Age.age : 4

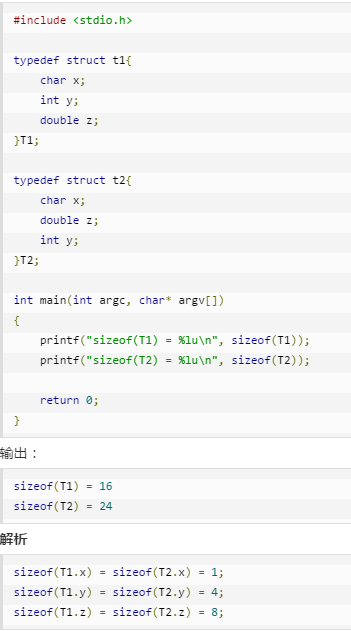
Age.age : 7

Age.age : 0

##### **结构体内存分配原则**

**原则一**：结构体中元素按照定义顺序存放到内存中，但并不是紧密排列。从结构体存储的首地址开始 ，每一个元素存入内存中时，它都会认为内存是以自己的宽度来划分空间的，因此元素存放的位置一定会在自己大小的整数倍上开始。

**原则二**： 在原则一的基础上，检查计算出的存储单元是否为所有元素中最宽的元素长度的整数倍。若是，则结束；否则，将其补齐为它的整数倍。



**T1**: 若从第 0 个字节开始分配内存，则 T1.x 存入第 0 字节，T1.y 占 4 个字节，由于第一的 4 字节已有数据，所以 T1.y 存入第 4-7 个字节，T1.z 占 8 个字节，由于第一个 8 字节已有数据，所以 T1.z 存入 8-15 个字节。共占有 16 个字节。

**T2**: 若从第 0 个字节开始分配内存，则 T1.x 存入第 0 字节，T1.z 占 8 个字节，由于第一的 8 字节已有数据，所以 T1.z 存入第 8-15 个字节，T1.y 占 4 个字节，由于前四个 4 字节已有数据，所以 T1.z 存入 16-19 个字节。共占有 20 个字节。此时所占字节不是最宽元素（double 长度为 8）的整数倍，因此将其补齐到 8 的整数倍，最终结果为 24。

#### typedef

<http://www.runoob.com/cprogramming/c-typedef.html>

C 语言提供了 **typedef** 关键字，您可以使用它来为类型取一个新的名字。下面的实例为单字节数字定义了一个术语 **BYTE**：

typedef unsigned char BYTE;

在这个类型定义之后，标识符 BYTE 可作为类型 **unsigned char** 的缩写，例如：

BYTE b1, b2;

按照惯例，定义时会大写字母，以便提醒用户类型名称是一个象征性的缩写，但您也可以使用小写字母，如下：

typedef unsigned char byte;

您也可以使用 **typedef** 来为用户自定义的数据类型取一个新的名字。例如，您可以对结构体使用 typedef 来定义一个新的数据类型名字，然后使用这个新的数据类型来直接定义结构变量，如下：

#include <stdio.h>

#include <string.h>

typedef struct Books

{

char title[50];

char author[50];

char subject[100];

int book\_id;

} Book;

int main( )

{

Book book;

strcpy( book.title, "C 教程");

strcpy( book.author, "Runoob");

strcpy( book.subject, "编程语言");

book.book\_id = 12345;

printf( "书标题 : %s\n", book.title);

printf( "书作者 : %s\n", book.author);

printf( "书类目 : %s\n", book.subject);

printf( "书 ID : %d\n", book.book\_id);

return 0;

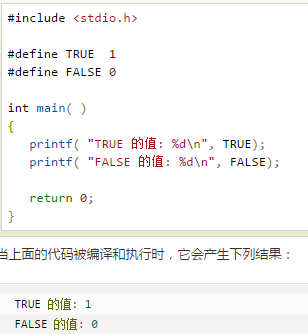
}

##### typedef vs #define

**#define** 是 C 指令，用于为各种数据类型定义别名，与 **typedef** 类似，但是它们有以下几点不同：

* **typedef** 仅限于为类型定义符号名称，**#define** 不仅可以为类型定义别名，也能为数值定义别名，比如您可以定义 1 为 ONE。
* **typedef** 是由编译器执行解释的，**#define** 语句是由预编译器进行处理的。

下面是 #define 的最简单的用法：



**typedef 与 #define 的区别**

（1）#define可以使用其他类型说明符对宏类型名进行扩展，但对 typedef 所定义的类型名却不能这样做。例如：

#define INTERGE int

unsigned INTERGE n; //没问题

typedef int INTERGE;

unsigned INTERGE n; //错误，不能在 INTERGE 前面添加 unsigned

（2） 在连续定义几个变量的时候，typedef 能够保证定义的所有变量均为同一类型，而 #define 则无法保证。例如：

#define PTR\_INT int \*

PTR\_INT p1, p2; //p1、p2 类型不相同，宏展开后变为int \*p1, p2;

typedef int \* PTR\_INT

PTR\_INT p1, p2; //p1、p2 类型相同，它们都是指向 int 类型的指针。

##### typedef定义别名

用 **typedef** 为数组去别名：

typedef int A[6];

表示用 **A** 代替 **int [6]**。

即：**A a;** 等于 **int a[6];**

**typedef 还有一个作用，就是为复杂的声明定义一个新的简单的别名。用在回调函数中特别好用：**

1. 原声明：**int \*(\*a[5])(int, char\*);**

在这里，变量名为 **a**，直接用一个新别名 **pFun** 替换 **a** 就可以了：

typedef int \*(\*pFun)(int, char\*);

于是，原声明的最简化版：

pFun a[5];

2. 原声明：**void (\*b[10]) (void (\*)());**

这里，变量名为 b，先替换右边部分括号里的，pFunParam 为别名一：

typedef void (\*pFunParam)();

再替换左边的变量 **b**，**pFunx** 为别名二：

typedef void (\*pFunx)(pFunParam);

于是，原声明的最简化版：

pFunx b[10];

其实，可以这样理解:

typedef int \*(\*pFun)(int, char\*);

由 **typedef** 定义的函数 **pFun**，为一个新的类型，所以这个新的类型可以像 **int** 一样定义变量，于是，**pFun a[5];**就定义了 **int \*(\*a[5])(int, char\*);**

所以我们可以用来定义回调函数，特别好用。

另外，也要注意，typedef 在语法上是一个存储类的关键字（如 auto、extern、mutable、static、register 等一样），虽然它并不真正影响对象的存储特性，如：

typedef static int INT2; // 不可行

编译将失败，会提示“指定了一个以上的存储类”。

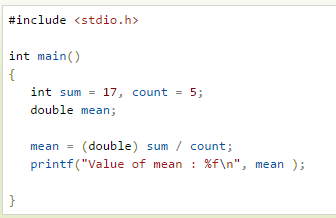
### 强制类型转换

<http://www.runoob.com/cprogramming/c-type-casting.html>

强制类型转换是把变量从一种类型转换为另一种数据类型。例如，如果您想存储一个 long 类型的值到一个简单的整型中，您需要把 long 类型强制转换为 int 类型。您可以使用**强制类型转换运算符**来把值显式地从一种类型转换为另一种类型，如下所示：

(type\_name) expression

请看下面的实例，使用强制类型转换运算符把一个整数变量除以另一个整数变量，得到一个浮点数：



当上面的代码被编译和执行时，它会产生下列结果：

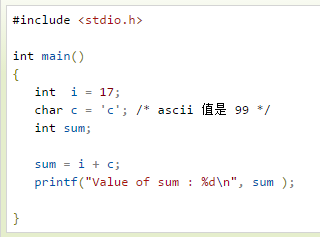
Value of mean : 3.400000

这里要注意的是强制类型转换运算符的优先级大于除法，因此 **sum** 的值首先被转换为 **double** 型，然后除以 count，得到一个类型为 double 的值。

类型转换可以是隐式的，由编译器自动执行，也可以是显式的，通过使用**强制类型转换运算符**来指定。在编程时，有需要类型转换的时候都用上强制类型转换运算符，是一种良好的编程习惯。

#### 整数提升

整数提升是指把小于 **int** 或 **unsigned int** 的整数类型转换为 **int** 或 **unsigned int** 的过程。请看下面的实例，在 int 中添加一个字符：



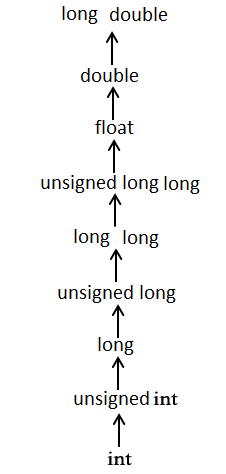
当上面的代码被编译和执行时，它会产生下列结果：

Value of sum : 116

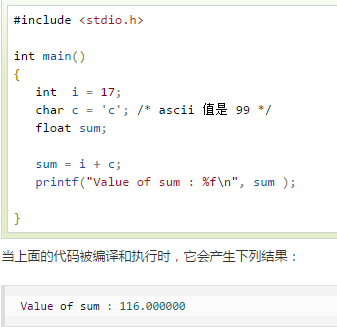
在这里，sum 的值为 116，因为编译器进行了整数提升，在执行实际加法运算时，把 'c' 的值转换为对应的 ascii 值。

#### 常用的算术转换

**常用的算术转换**是隐式地把值强制转换为相同的类型。编译器首先执行**整数提升**，如果操作数类型不同，则它们会被转换为下列层次中出现的最高层次的类型：

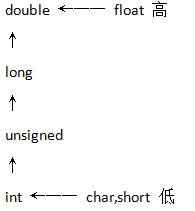


常用的算术转换不适用于赋值运算符、逻辑运算符 && 和 ||。让我们看看下面的实例来理解这个概念：



在这里，c 首先被转换为整数，但是由于最后的值是 double 型的，所以会应用常用的算术转换，编译器会把 i 和 c 转换为浮点型，并把它们相加得到一个浮点数。

如果一个运算符两边的运算数类型不同，先要将其转换为相同的类型，即较低类型转换为较高类型，然后再参加运算，转换规则如下图所示。



### 变量

变量其实只不过是程序可操作的存储区的名称。C 中每个变量都有特定的类型，类型决定了变量存储的大小和布局，该范围内的值都可以存储在内存中，运算符可应用于变量上。

#### 变量定义

变量定义就是告诉编译器在何处创建变量的存储，以及如何创建变量的存储。变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表，如下所示：

type variable\_list;

在这里，**type** 必须是一个有效的 C 数据类型，可以是 char、w\_char、int、float、double、bool 或任何用户自定义的对象，**variable\_list** 可以由一个或多个标识符名称组成，多个标识符之间用逗号分隔。下面列出几个有效的声明：

int i, j, k;

char c, ch;

float f, salary;

double d;

行 **int i, j, k;** 声明并定义了变量 i、j 和 k，这指示编译器创建类型为 int 的名为 i、j、k 的变量。

变量可以在声明的时候被初始化（指定一个初始值）。初始化器由一个等号，后跟一个常量表达式组成，如下所示：

type variable\_name = value;

下面列举几个实例：

extern int d = 3, f = 5; // d 和 f 的声明与初始化

int d = 3, f = 5; // 定义并初始化 d 和 f

byte z = 22; // 定义并初始化 z

char x = 'x'; // 变量 x 的值为 'x'

不带初始化的定义：带有静态存储持续时间的变量会被隐式初始化为 NULL（所有字节的值都是 0），其他所有变量的初始值是未定义的。

#### 变量声明

变量声明向编译器保证变量以指定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明。

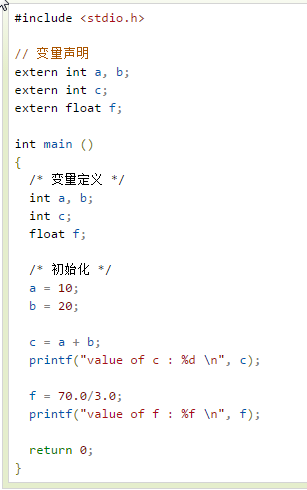
变量的声明有两种情况：

* 1、一种是需要建立存储空间的。例如：int a 在声明的时候就已经建立了存储空间。
* 2、另一种是不需要建立存储空间的，通过使用extern关键字声明变量名而不定义它。 例如：extern int a 其中变量 a 可以在别的文件中定义的。
* 除非有extern关键字，否则都是变量的定义。

extern int i; //声明，不是定义

int i; //声明，也是定义

尝试下面的实例，其中，变量在头部就已经被声明，但是定义与初始化在主函数内：



当上面的代码被编译和执行时，它会产生下列结果：

value of c : 30

value of f : 23.333334

#### extern

声明之后你不能直接使用这个变量，需要定义之后才能使用。

第四个等于第三个，都是定义一个可以被外部使用的全局变量，并给初值。

糊涂了吧，他们看上去可真像。但是定义只能出现在一处。也就是说，不管是 int a 还是 int a=0 都只能出现一次，而那个 extern int a 可以出现很多次。

当你要引用一个全局变量的时候，你就要声明 extern int a 这时候 extern 不能省略，因为省略了，就变成 int a 这是一个定义，不是声明。



#### C 中的左值（Lvalues）和右值（Rvalues）

C 中有两种类型的表达式：

**左值（lvalue）：**指向内存位置的表达式被称为左值（lvalue）表达式。左值可以出现在赋值号的左边或右边。

**右值（rvalue）：**术语右值（rvalue）指的是存储在内存中某些地址的数值。右值是不能对其进行赋值的表达式，也就是说，右值可以出现在赋值号的右边，但不能出现在赋值号的左边。

变量是左值，因此可以出现在赋值号的左边。数值型的字面值是右值，因此不能被赋值，不能出现在赋值号的左边。下面是一个有效的语句：

 1、 当需要保存数据的时候，需要lvalues。

 2、 当需要读取数据的时候，需要rvalues。

#### 变量的内存寻址(与系统有关)

(1)内存寻址由大到小，优先分配内存地址比较大的字节给变量，所以说变量越先定义，内存地址就越大。 如下面代码，先定义变量 a,再定义变量 b,打印出 a 的地址是 0x7fff5fbff828,b 的值是 0x7fff5fbff824。a 的地址比 b 的地址大 4 字节。

(2)变量地址的获取方式：& 变量名。

(3)输出地址的方式：%p。

一个变量一定要先初始化才可以使用，因为 c 语言中默认一个没有初始化的变量值是一个不可知的很大值。如下面所示，a 没有初始化，打印出 a 的值是 1606422582。

### 变量作用域

<http://www.runoob.com/cprogramming/c-scope-rules.html>

任何一种编程中，作用域是程序中定义的变量所存在的区域，超过该区域变量就不能被访问。C 语言中有三个地方可以声明变量：

1. 在函数或块内部的**局部**变量
2. 在所有函数外部的**全局**变量
3. 在**形式**参数的函数参数定义中

让我们来看看什么是**局部**变量、**全局**变量和**形式**参数。

#### 局部变量

在某个函数或块的内部声明的变量称为局部变量。它们只能被该函数或该代码块内部的语句使用。局部变量在函数外部是不可知的。下面是使用局部变量的实例。在这里，所有的变量 a、b 和 c 是 main() 函数的局部变量。

实例

#include <stdio.h> int main () { /\* 局部变量声明 \*/ int a, b; int c; /\* 实际初始化 \*/ a = 10; b = 20; c = a + b; printf ("value of a = %d, b = %d and c = %d\n", a, b, c); return 0; }

#### 全局变量

全局变量是定义在函数外部，通常是在程序的顶部。全局变量在整个程序生命周期内都是有效的，在任意的函数内部能访问全局变量。

全局变量可以被任何函数访问。也就是说，全局变量在声明后整个程序中都是可用的。下面是使用全局变量和局部变量的实例：

实例

#include <stdio.h> /\* 全局变量声明 \*/ int g; int main () { /\* 局部变量声明 \*/ int a, b; /\* 实际初始化 \*/ a = 10; b = 20; g = a + b; printf ("value of a = %d, b = %d and g = %d\n", a, b, g); return 0; }

在程序中，局部变量和全局变量的名称可以相同，但是在函数内，局部变量的值会覆盖全局变量的值。下面是一个实例：

实例

#include <stdio.h> /\* 全局变量声明 \*/ int g = 20; int main () { /\* 局部变量声明 \*/ int g = 10; printf ("value of g = %d\n", g); return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

value of g = 10

#### 形式参数

函数的参数，形式参数，被当作该函数内的局部变量，它们会优先覆盖全局变量。下面是一个实例：

实例

#include <stdio.h> /\* 全局变量声明 \*/ int a = 20; int main () { /\* 在主函数中的局部变量声明 \*/ int a = 10; int b = 20; int c = 0; int sum(int, int); printf ("value of a in main() = %d\n", a); c = sum( a, b); printf ("value of c in main() = %d\n", c); return 0; } /\* 添加两个整数的函数 \*/ int sum(int a, int b) { printf ("value of a in sum() = %d\n", a); printf ("value of b in sum() = %d\n", b); return a + b; }

当上面的代码被编译和执行时，它会产生下列结果：

value of a in main() = 10

value of a in sum() = 10

value of b in sum() = 20

value of c in main() = 30

#### 初始化局部变量和全局变量

当局部变量被定义时，系统不会对其初始化，您必须自行对其初始化。定义全局变量时，系统会自动对其初始化，如下所示：

|  |  |
| --- | --- |
| **数据类型** | **初始化默认值** |
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

正确地初始化变量是一个良好的编程习惯，否则有时候程序可能会产生意想不到的结果，因为未初始化的变量会导致一些在内存位置中已经可用的垃圾值。

### 常量

#### 整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，不带前缀则默认表示十进制。

整数常量也可以带一个后缀，后缀是 U 和 L 的组合，U 表示无符号整数（unsigned），L 表示长整数（long）。后缀可以是大写，也可以是小写，U 和 L 的顺序任意。

下面列举几个整数常量的实例：

212 /\* 合法的 \*/

215u /\* 合法的 \*/

0xFeeL /\* 合法的 \*/

078 /\* 非法的：8 不是八进制的数字 \*/

032UU /\* 非法的：不能重复后缀 \*/

以下是各种类型的整数常量的实例：

85 /\* 十进制 \*/

0213 /\* 八进制 \*/

0x4b /\* 十六进制 \*/

30 /\* 整数 \*/

30u /\* 无符号整数 \*/

30l /\* 长整数 \*/

30ul /\* 无符号长整数 \*/

#### 浮点常量

浮点常量由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

当使用小数形式表示时，必须包含整数部分、小数部分，或同时包含两者。当使用指数形式表示时， 必须包含小数点、指数，或同时包含两者。带符号的指数是用 e 或 E 引入的。

下面列举几个浮点常量的实例：



#### 字符常量

字符常量是括在单引号中，例如，'x' 可以存储在 **char** 类型的简单变量中。

字符常量可以是一个普通的字符（例如 'x'）、一个转义序列（例如 '\t'），或一个通用的字符（例如 '\u02C0'）。

在 C 中，有一些特定的字符，当它们前面有反斜杠时，它们就具有特殊的含义，被用来表示如换行符（\n）或制表符（\t）等。

下表列出了一些这样的转义序列码：



#### 字符串常量

在 C 语言中，单引号与双引号是有很大区别的。

在 C 语言中没有专门的字符串类型，因此双引号内的字符串会被存储到一个数组中，这个字符串代表指向这个数组起始字符的指针；

而单引号中的内容是一个 char 类型，是一个字符，这个字符对应的是 ASCII 表中的序列值。

字符串字面值或常量是括在双引号 "" 中的。一个字符串包含类似于字符常量的字符：普通的字符、转义序列和通用的字符。

您可以使用空格做分隔符，把一个很长的字符串常量进行分行。

下面的实例显示了一些字符串常量。下面这三种形式所显示的字符串是相同的。

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

反斜杠(**\**) 开头是叫转义序列(Escape Sequence)。

**\ooo** 是对用三位八进制数转义表示任意字符的形象化描述。

比如 **char ch = '\101';** 等价于 **char ch = 0101;** (以0开头的表示八进制）。

**\xhh** 里面是 x 是固定的，表示十六进制(hexadecimal)，h 也表示十六进制。

举例，**char ch = '\x41';** 就是用十六进制来表示，它与前面的 **\101** 是等价的。

可用如下代码证明它们等价：

#include <stdio.h>

int main(){

printf("%c,%c,%c,%c", 0101, '\101', '\x41', 'A');

return 0;

}

#### 定义常量

在 C 中，有两种简单的定义常量的方式：

使用 **#define** 预处理器。

使用 **const** 关键字。

#define LENGTH 10

#define WIDTH 5

#define NEWLINE '\n'

const int LENGTH = 10;

const int WIDTH = 5;

const char NEWLINE = '\n';

请注意，把常量定义为大写字母形式，是一个很好的编程实践。

**#define** 是宏定义，它不能定义常量，但宏定义可以实现在字面意义上和其它定义常量相同的功能，本质的区别就在于 **#define** 不为宏名分配内存，

而 **const** 也不为常量分配内存，怎么回事呢，其实 **const** 并不是去定义一个常量，而是去改变一个变量的存储类，把该变量所占的内存变为只读！

**const** 定义的是变量不是常量，只是这个变量的值不允许改变是常变量！带有类型。编译运行的时候起作用存在类型检查。

**define** 定义的是不带类型的常数，只进行简单的字符替换。在预编译的时候起作用，不存在类型检查。

1、两者的区别

**(1) 编译器处理方式不同**

#define 宏是在预处理阶段展开。

 const 常量是编译运行阶段使用。

**(2) 类型和安全检查不同**

 #define 宏没有类型，不做任何类型检查，仅仅是展开。

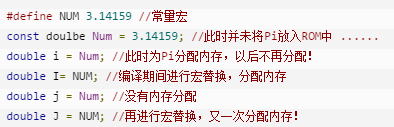
 const 常量有具体的类型，在编译阶段会执行类型检查。

**(3) 存储方式不同**

#define宏仅仅是展开，有多少地方使用，就展开多少次，不会分配内存。（宏定义不分配内存，变量定义分配内存。）

const常量会在内存中分配(可以是堆中也可以是栈中)。

**(4) const 可以节省空间，避免不必要的内存分配。 例如：**



const 定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是象 #define 一样给出的是立即数，所以，const 定义的常量在程序运行过程中只有一份拷贝（因为是全局的只读变量，存在静态区），而 #define 定义的常量在内存中有若干个拷贝。

**(5) 提高了效率。 编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。**

**(6) 宏替换只作替换，不做计算，不做表达式求解;**

宏预编译时就替换了，程序运行时，并不分配内存。

define 注意“边缘效应”，例：**#define N 2+3**, N 的值是 5。

int a = N/2

在编译时我们预想 **a=2.5**，实际打印结果是 **3.5** 原因是在预处理阶段，编译器将 **a=N/2** 处理成 **a=2+3/2**，这就是 **define** 宏的边缘效应，所以我们应该写成 **#define N (2+3)**。

#include <stdio.h>

#define LENGTH 10+10

//正确写法 #define LENGTH (10+10)

#define WIDTH 5

#define NEWLINE '\n'

int main(){

    int area;

    area = LENGTH \* WIDTH;

    printf("value of area : %d", area);

    printf("%c", NEWLINE);

    return 0;

}

以上实例输出结果为：

value of area : 60

所以如果我们需要得到正确结果应该将 **#define LENGTH 10+10** 修改为 **#define LENGTH (10+10)**。

### 存储类

<http://www.runoob.com/cprogramming/c-storage-classes.html>

存储类定义 C 程序中变量/函数的范围（可见性）和生命周期。这些说明符放置在它们所修饰的类型之前。下面列出 C 程序中可用的存储类：

* auto
* register
* static
* extern

#### auto 存储类

**auto** 存储类是所有局部变量默认的存储类。

{

int mount;

auto int month;

}

上面的实例定义了两个带有相同存储类的变量，auto 只能用在函数内，即 auto 只能修饰局部变量。

#### register 存储类

**register** 存储类用于定义存储在寄存器中而不是 RAM 中的局部变量。这意味着变量的最大尺寸等于寄存器的大小（通常是一个词），

且不能对它应用一元的 '&' 运算符（因为它没有内存位置）。

{

register int miles;

}

寄存器只用于需要快速访问的变量，比如计数器。还应注意的是，定义 'register' 并不意味着变量将被存储在寄存器中，它意味着变量可能存储在寄存器中，这取决于硬件和实现的限制。

#### static 存储类

**static** 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 static 修饰局部变量可以在函数调用之间保持局部变量的值。

static 修饰符也可以应用于全局变量。当 static 修饰全局变量时，会使变量的作用域限制在声明它的文件内。

static 是全局变量的默认存储类，以下两个变量 (count 和 road) 都有一个 static 存储类。

static int Count;

int Road;

main()

{

printf("%d\n", Count);

printf("%d\n", Road);

}

以下实例演示了 static 修饰全局变量和局部变量的应用：



实例中 count 作为全局变量可以在函数内使用，thingy 使用 static 修饰后，不会再每次调用时重置。

可能您现在还无法理解这个实例，因为我已经使用了函数和全局变量，这两个概念目前为止还没进行讲解。即使您现在不能完全理解，也没有关系，后续的章节我们会详细讲解。当上面的代码被编译和执行时，它会产生下列结果：

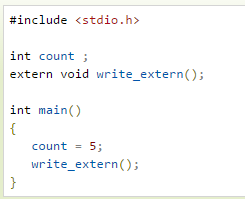
#### extern 存储类

**extern** 存储类用于提供一个全局变量的引用，全局变量对所有的程序文件都是可见的。当您使用 'extern' 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置。

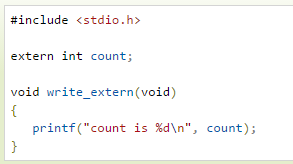
当您有多个文件且定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 *extern* 来得到已定义的变量或函数的引用。可以这么理解，*extern* 是用来在另一个文件中声明一个全局变量或函数。

extern 修饰符通常用于当有两个或多个文件共享相同的全局变量或函数的时候，如下所示：

**第一个文件：main.c**



**第二个文件：support.c**



在这里，第二个文件中的 *extern* 关键字用于声明已经在第一个文件 main.c 中定义的 *count*。现在 ，编译这两个文件，如下所示：

$ gcc main.c support.c

这会产生 **a.out** 可执行程序，当程序被执行时，它会产生下列结果：

count is 5

### 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C 语言内置了丰富的运算符，并提供了以下类型的运算符：

* 算术运算符
* 关系运算符
* 逻辑运算符
* 位运算符
* 赋值运算符
* 杂项运算符

本章将逐一介绍

#### 算术运算符

下表显示了 C 语言支持的所有算术运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| + | 把两个操作数相加 | A + B 将得到 30 |
| - | 从第一个操作数中减去第二个操作数 | A - B 将得到 -10 |
| \* | 把两个操作数相乘 | A \* B 将得到 200 |
| / | 分子除以分母 | B / A 将得到 2 |
| % | 取模运算符，整除后的余数 | B % A 将得到 0 |
| ++ | 自增运算符，整数值增加 1 | A++ 将得到 11 |
| -- | 自减运算符，整数值减少 1 | A-- 将得到 9 |

#### 关系运算符

下表显示了 C 语言支持的所有关系运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| == | 检查两个操作数的值是否相等，如果相等则条件为真。 | (A == B) 不为真。 |
| != | 检查两个操作数的值是否相等，如果不相等则条件为真。 | (A != B) 为真。 |
| > | 检查左操作数的值是否大于右操作数的值，如果是则条件为真。 | (A > B) 不为真。 |
| < | 检查左操作数的值是否小于右操作数的值，如果是则条件为真。 | (A < B) 为真。 |
| >= | 检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。 | (A >= B) 不为真。 |
| <= | 检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。 | (A <= B) 为真。 |

#### 逻辑运算符

下表显示了 C 语言支持的所有关系逻辑运算符。假设变量 **A** 的值为 1，变量 **B** 的值为 0，则：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| && | 称为逻辑与运算符。如果两个操作数都非零，则条件为真。 | (A && B) 为假。 |
| || | 称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。 | (A || B) 为真。 |
| ! | 称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。 | !(A && B) 为真。 |

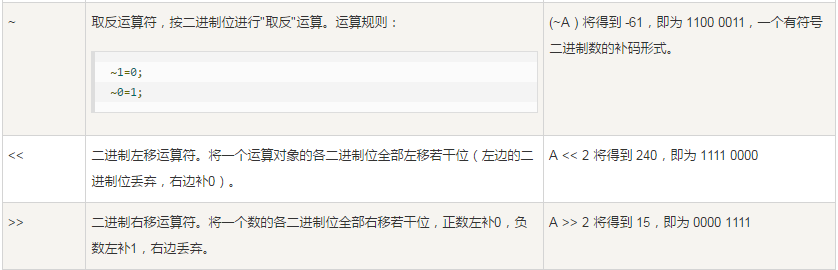
#### 位运算符

位运算符作用于位，并逐位执行操作。

A = 0011 1100

B = 0000 1101





#### 赋值运算符

下表列出了 C 语言支持的赋值运算符：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| = | 简单的赋值运算符，把右边操作数的值赋给左边操作数 | C = A + B 将把 A + B 的值赋给 C |
| += | 加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数 | C += A 相当于 C = C + A |
| -= | 减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数 | C -= A 相当于 C = C - A |
| \*= | 乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数 | C \*= A 相当于 C = C \* A |
| /= | 除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数 | C /= A 相当于 C = C / A |
| %= | 求模且赋值运算符，求两个操作数的模赋值给左边操作数 | C %= A 相当于 C = C % A |
| <<= | 左移且赋值运算符 | C <<= 2 等同于 C = C << 2 |
| >>= | 右移且赋值运算符 | C >>= 2 等同于 C = C >> 2 |
| &= | 按位与且赋值运算符 | C &= 2 等同于 C = C & 2 |
| ^= | 按位异或且赋值运算符 | C ^= 2 等同于 C = C ^ 2 |
| |= | 按位或且赋值运算符 | C |= 2 等同于 C = C | 2 |

#### 杂项运算符：sizeof() & \* 三元(?:)

下表列出了 C 语言支持的其他一些重要的运算符，包括 **sizeof** 和 **? :**。

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| sizeof() | 返回变量的大小。 | sizeof(a) 将返回 4，其中 a 是整数。 |
| & | 返回变量的地址。 | &a; 将给出变量的实际地址。 |
| \* | 指向一个变量。 | \*a; 将指向一个变量。 |
| ? : | 条件表达式 | 如果条件为真 ? 则值为 X : 否则值为 Y |

#### C 中的运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 x = 7 + 3 \* 2，在这里，x 被赋值为 13，而不是 20，因为运算符 \* 具有比 + 更高的优先级，所以首先计算乘法 3\*2，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

|  |  |  |
| --- | --- | --- |
| **类别** | **运算符** | **结合性** |
| 后缀 | () [] -> . ++ - - | 从左到右 |
| 一元 | + - ! ~ ++ - - (type)\* & sizeof | 从右到左 |
| 乘除 | \* / % | 从左到右 |
| 加减 | + - | 从左到右 |
| 移位 | << >> | 从左到右 |
| 关系 | < <= > >= | 从左到右 |
| 相等 | == != | 从左到右 |
| 位与 AND | & | 从左到右 |
| 位异或 XOR | ^ | 从左到右 |
| 位或 OR | | | 从左到右 |
| 逻辑与 AND | && | 从左到右 |
| 逻辑或 OR | || | 从左到右 |
| 条件 | ?: | 从右到左 |
| 赋值 | = += -= \*= /= %=>>= <<= &= ^= |= | 从右到左 |
| 逗号 | , | 从左到右 |

#### 笔记

##### 利用异或 ^ 来交换两个数的值，而且不引入其他变量。

unsigned int a=60; //0011 1100

unsigned int b=13; //0000 1101

a=a^b; //a=a^b=0011 0001

b=a^b; //b=a^b=0011 1100 相当于b1=(a^b)^b

a=a^b; //a=a^b=0000 1101 相当于a1=(a^b)^((a^b)^b)

##### 利用位与 & 运算，判断一个整数是否是2的整数次幂。

二进制数的位权是以2为底的幂，如果一个整数 m 是 2 的 n 次幂，那么转换为二进制之后只有最高位为 1，其余位置为 0，再观察 m-1 转换为二进制后的形式以及 m&(m-1) 的结果，例如：

可以看出所有的 1 完美的错过了，根据位与的特点可知 m&(m-1) 的结果为 0。

int func(int num)

{

return ((num > 0) && ((num & (num - 1)) == 0));//2的n次幂大于0

}

##### 不同长度的数据进行位运算

如果两个不同长度的数据进行位运算时，系统会将二者按右端对齐，然后进行位运算。

以“与”运算为例说明如下：我们知道在 C 语言中 long 型占 4 个字节，int 型占 2 个字节，如果一个 long 型数据与一个 int 型数据进行“与”运算，右端对齐后，左边不足的位依下面三种情况补足:

* （1）如果整型数据为正数，左边补 16 个 0。
* （2）如果整型数据为负数，左边补 16 个 1。
* （3）如果整形数据为无符号数，左边也补 16 个 0。

如：**long a=123;int b=1;** 计算 **a & b**。

如：**long a=123;int b=-1;** 计算 **a & b**。

如：**long a=123; unsigned int b=1;** 计算 **a & b**。

### 语言条件判断

#### 判断语句

C 语言提供了以下类型的判断语句。点击链接查看每个语句的细节。

|  |  |
| --- | --- |
| **语句** | **描述** |
| [if 语句](http://www.runoob.com/cprogramming/c-if.html) | 一个 **if 语句** 由一个布尔表达式后跟一个或多个语句组成。 |
| [if...else 语句](http://www.runoob.com/cprogramming/c-if-else.html) | 一个 **if 语句** 后可跟一个可选的 **else 语句**，else 语句在布尔表达式为假时执行。 |
| [嵌套 if 语句](http://www.runoob.com/cprogramming/c-nested-if.html) | 您可以在一个 **if** 或 **else if** 语句内使用另一个 **if** 或 **else if** 语句。 |
| [switch 语句](http://www.runoob.com/cprogramming/c-switch.html) | 一个 **switch** 语句允许测试一个变量等于多个值时的情况。 |
| [嵌套 switch 语句](http://www.runoob.com/cprogramming/c-nested-switch.html) | 您可以在一个 **switch** 语句内使用另一个 **switch**语句。 |

#### ? : 运算符(三元运算符)

我们已经在前面的章节中讲解了 **条件运算符 ? :**，可以用来替代 **if...else** 语句。它的一般形式如下：

Exp1 ? Exp2 : Exp3;

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

#### switch语句一般形式：

switch(表达式)

{

    case 常量表达式1:语句1;

    case 常量表达式2:语句2;

    ...

    default:语句n+1;

}

意思是先计算表达式的值，再逐个和 case 后的常量表达式比较，若不等则继续往下比较，若一直不等，则执行 default 后的语句；若等于某一个常量表达式，则从这个表达式后的语句开始执行，并执行后面所有 case 后的语句。

与 if 语句的不同：if 语句中若判断为真则只执行这个判断后的语句，执行完就跳出 if 语句，不会执行其他 if 语句；而 switch 语句不会在执行判断为真后的语句之后跳出循环，而是继续执行后面所有 case 语句。在每一 case 语句之后增加 break 语句，使每一次执行之后均可跳出 switch 语句，从而避免输出不应有的结果。

### 语言循环语句

#### 循环类型

C 语言提供了以下几种循环类型。点击链接查看每个类型的细节。

|  |  |
| --- | --- |
| **循环类型** | **描述** |
| [while 循环](http://www.runoob.com/cprogramming/c-while-loop.html) | 当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。 |
| [for 循环](http://www.runoob.com/cprogramming/c-for-loop.html) | 多次执行一个语句序列，简化管理循环变量的代码。 |
| [do...while 循环](http://www.runoob.com/cprogramming/c-do-while-loop.html) | 除了它是在循环主体结尾测试条件外，其他与 while 语句类似。 |
| [嵌套循环](http://www.runoob.com/cprogramming/c-nested-loops.html) | 您可以在 while、for 或 do..while 循环内使用一个或多个循环。 |

while(num2<=100){

sum2=sum2+num2;

num2=num2+2;

}

for(num2=2;num2<=100;num2=num2+2){

sum2=sum2+num2;

}

#### 循环控制语句

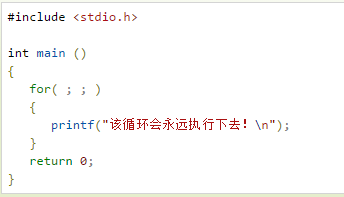
循环控制语句改变你代码的执行顺序。通过它你可以实现代码的跳转。

C 提供了下列的循环控制语句。点击链接查看每个语句的细节。

|  |  |
| --- | --- |
| **控制语句** | **描述** |
| [break 语句](http://www.runoob.com/cprogramming/c-break-statement.html) | 终止**循环**或 **switch** 语句，程序流将继续执行紧接着循环或 switch 的下一条语句。 |
| [continue 语句](http://www.runoob.com/cprogramming/c-continue-statement.html) | 告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。 |
| [goto 语句](http://www.runoob.com/cprogramming/c-goto-statement.html) | 将控制转移到被标记的语句。但是不建议在程序中使用 goto 语句。 |

#### 无限循环

如果条件永远不为假，则循环将变成无限循环。**for** 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。



当条件表达式不存在时，它被假设为真。您也可以设置一个初始值和增量表达式，但是一般情况下，C 程序员偏向于使用 for(;;) 结构来表示一个无限循环。

**注意：**您可以按 **Ctrl + C** 键终止一个无限循环。

### 函数

<http://www.runoob.com/cprogramming/c-functions.html>

函数是一组一起执行一个任务的语句。每个 C 程序都至少有一个函数，即主函数 **main()** ，所有简单的程序都可以定义其他额外的函数。

您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

函数**声明**告诉编译器函数的名称、返回类型和参数。函数**定义**提供了函数的实际主体。

C 标准库提供了大量的程序可以调用的内置函数。例如，函数 **strcat()** 用来连接两个字符串，函数 **memcpy()** 用来复制内存到另一个位置。

#### 函数定义

return\_type function\_name( parameter list )

{

body of the function

}

在 C 语言中，函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：

* **返回类型：**一个函数可以返回一个值。**return\_type** 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，return\_type 是关键字 **void**。
* **函数名称：**这是函数的实际名称。函数名和参数列表一起构成了函数签名。
* **参数：**参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。
* **函数主体：**函数主体包含一组定义函数执行任务的语句。

#### 函数声明

函数**声明**会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

函数声明包括以下几个部分：

return\_type function\_name( parameter list );

针对上面定义的函数 max()，以下是函数声明：

int max(int num1, int num2);

在函数声明中，参数的名称并不重要，只有参数的类型是必需的，因此下面也是有效的声明：

int max(int, int);

当您在一个源文件中定义函数且在另一个文件中调用函数时，函数声明是必需的。在这种情况下，您应该在调用函数的文件顶部声明函数。

函数声明和函数原型的参数名可以不一样，编译器他想知道的是函数参数的类型，与函数参数的名字没有关系，eg:

#

include<stdio.h>

int sum(int c,int d); // 函数声明

int main(int argc ,char\*argv[])

{

int a=2,b=3;

printf("输出结果为: %d \n",sum(a,b));

return 0;

}

int sum(int a,int b)

{

return a+b ;

}

甚至函数声明可以写成:

int sum(int ,int );

编译器只要检查到函数返回类型，名称和参数类型正确即可。

#### 调用函数

创建 C 函数时，会定义函数做什么，然后通过调用函数来完成已定义的任务。

当程序调用函数时，程序控制权会转移给被调用的函数。被调用的函数执行已定义的任务，当函数的返回语句被执行时，或到达函数的结束括号时，会把程序控制权交还给主程序。

调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值。例如：

#include <stdio.h>

/\* 函数声明 \*/

int max(int num1, int num2);

int main ()

{

/\* 局部变量定义 \*/

int a = 100;

int b = 200;

int ret;

/\* 调用函数来获取最大值 \*/

ret = max(a, b);

printf( "Max value is : %d\n", ret );

return 0;

}

/\* 函数返回两个数中较大的那个数 \*/

int max(int num1, int num2)

{

/\* 局部变量声明 \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

把 max() 函数和 main() 函数放一块，编译源代码。当运行最后的可执行文件时，会产生下列结果：

Max value is : 200

#### 函数参数

如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的**形式参数**。

形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁。

当调用函数时，有两种向函数传递参数的方式：

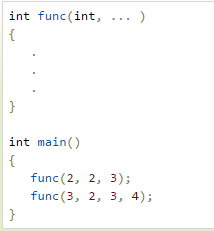
|  |  |
| --- | --- |
| **调用类型** | **描述** |
| [传值调用](http://www.runoob.com/cprogramming/c-function-call-by-value.html) | 该方法把参数的实际值复制给函数的形式参数。在这种情况下，修改函数内的形式参数不会影响实际参数。 |
| [引用调用](http://www.runoob.com/cprogramming/c-function-call-by-pointer.html) | 通过指针传递方式，形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作。 |

默认情况下，C 使用**传值调用**来传递参数。一般来说，这意味着函数内的代码不能改变用于调用函数的实际参数。

#### 可变参数

<http://www.runoob.com/cprogramming/c-variable-arguments.html>

有时，您可能会碰到这样的情况，您希望函数带有可变数量的参数，而不是预定义数量的参数。C 语言为这种情况提供了一个解决方案，它允许您定义一个函数，能根据具体的需求接受可变数量的参数。下面的实例演示了这种函数的定义。



请注意，函数 **func()** 最后一个参数写成省略号，即三个点号（**...**），省略号之前的那个参数是 **int**，代表了要传递的可变参数的总数。为了使用这个功能，您需要使用 **stdarg.h** 头文件，该文件提供了实现可变参数功能的函数和宏。具体步骤如下：

* 定义一个函数，最后一个参数为省略号，省略号前面可以设置自定义参数。
* 在函数定义中创建一个 **va\_list** 类型变量，该类型是在 stdarg.h 头文件中定义的。
* 使用 **int** 参数和 **va\_start** 宏来初始化 **va\_list** 变量为一个参数列表。宏 va\_start 是在 stdarg.h 头文件中定义的。
* 使用 **va\_arg** 宏和 **va\_list** 变量来访问参数列表中的每个项。
* 使用宏 **va\_end** 来清理赋予 **va\_list** 变量的内存。

现在让我们按照上面的步骤，来编写一个带有可变数量参数的函数，并返回它们的平均值：

#include <stdio.h>

#include <stdarg.h>

double average(int num,...)

{

va\_list valist;

double sum = 0.0;

int i;

/\* 为 num 个参数初始化 valist \*/

va\_start(valist, num);

/\* 访问所有赋给 valist 的参数 \*/

for (i = 0; i < num; i++)

{

sum += va\_arg(valist, int);

}

/\* 清理为 valist 保留的内存 \*/

va\_end(valist);

return sum/num;

}

int main()

{

printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));

printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));

}

当上面的代码被编译和执行时，它会产生下列结果。应该指出的是，函数 **average()** 被调用两次，每次第一个参数都是表示被传的可变参数的总数。省略号被用来传递可变数量的参数。

Average of 2, 3, 4, 5 = 3.500000

Average of 5, 10, 15 = 10.000000

##### C 函数要在程序中用到以下这些宏:

void va\_start( va\_list arg\_ptr, prev\_param );

type va\_arg( va\_list arg\_ptr, type );

void va\_end( va\_list arg\_ptr );

**va\_list**: 用来保存宏va\_start、va\_arg和va\_end所需信息的一种类型。为了访问变长参数列表中的参数，必须声明 va\_list 类型的一个对象，定义： **typedef char \* va\_list;**

**va\_start**: 访问变长参数列表中的参数之前使用的宏，它初始化用 va\_list 声明的对象，初始化结果供宏 va\_arg 和 va\_end 使用；

**va\_arg**: 展开成一个表达式的宏，该表达式具有变长参数列表中下一个参数的值和类型。每次调用 va\_arg 都会修改用 va\_list 声明的对象，从而使该对象指向参数列表中的下一个参数；

**va\_end**: 该宏使程序能够从变长参数列表用宏 va\_start 引用的函数中正常返回。

**va** 在这里是 variable-argument(可变参数) 的意思。

这些宏定义在 stdarg.h 中，所以用到可变参数的程序应该包含这个头文件。

下面我们写一个简单的可变参数的函数，改函数至少有一个整数参数，第二个参数也是整数，是可选的。函数只是打印这两个参数的值。

#include <stdio.h>;

#include <string.h>;

#include <stdarg.h>;

/\* ANSI标准形式的声明方式，括号内的省略号表示可选参数 \*/

int demo(char \*msg, ... )

{

va\_list argp; /\* 定义保存函数参数的结构 \*/

int argno = 0; /\* 纪录参数个数 \*/

char \*para; /\* 存放取出的字符串参数 \*/

va\_start( argp, msg ); /\* argp指向传入的第一个可选参数， msg是最后一个确定的参数 \*/

while (1)

{

para = va\_arg( argp, char \*); /\* 取出当前的参数，类型为char \*. \*/

if ( strcmp( para, "/0") == 0 )

/\* 采用空串指示参数输入结束 \*/

break;

printf("Parameter #%d is: %s/n", argno, para);

argno++;

}

va\_end( argp ); /\* 将argp置为NULL \*/

return 0;

}

void main( void )

{

demo("DEMO", "This", "is", "a", "demo!" ,"333333", "/0");

}

从这个函数的实现可以看到,我们使用可变参数应该有以下步骤:

* 1)首先在函数里定义一个 va\_list 型的变量,这里是 arg\_ptr，这个变量是指向参数的指针。
* 2)然后用 va\_start 宏初始化变量 arg\_ptr，这个宏的第二个参数是第一个可变参数的前一个参数，是一个固定的参数。
* 3)然后用 va\_arg 返回可变的参数，并赋值给整数 j。va\_arg 的第二个参数是你要返回的参数的类型，这里是int型。
* 4)最后用 va\_end 宏结束可变参数的获取。然后你就可以在函数里使用第二个参数了。如果函数有多个可变参数的，依次调用 va\_arg 获取各个参数。

#### 内部函数和外部函数

根据函数能否被其他源文件调用，将函数区分为内部函数和外部函数。

##### 内部函数static

如果一个函数只能被本文件中其他函数所调用，它称为内部函数。在定义内部函数时，在函数名和函数类型的前面加 static，即

static 类型名 函数名 （形参表）

例如，函数的首行：

static int max(int a,int b)

内部函数又称静态函数。使用内部函数，可以使函数的作用域只局限于所在文件。即使在不同的文件中有同名的内部函数，也互不干扰。提高了程序的可靠性。

##### 外部函数extern

如果在定义函数时，在函数的首部的最左端加关键字 extern，则此函数是外部函数，可供其它文件调用。

如函数首部可以为

extern int max (int a,int b)

C 语言规定，如果在定义函数时省略 extern，则默认为外部函数。

在需要调用此函数的其他文件中，需要对此函数作声明（不要忘记，即使在本文件中调用一个函数，也要用函数原型来声明）。在对此函数作声明时，要加关键字 extern，表示该函数是在其他文件中定义的外部函数。

file1.c(文件1)

#include <stdio.h>

static void delete\_string(char str[],char ch);

int main()

{

    extern void enter(char str[]); // 对函数的声明

    extern void print(char str[]); // 对函数的声明

    char c,str[100];

    enter(str);

    scanf("%c",&c);

    delete\_string(str,c);

    print(str);

    return 0;

}

#### 内联函数

内联函数是指用**inline**关键字修饰的函数。在类内定义的函数被默认成内联函数。内联函数从源代码层看，有函数的结构，而在编译后，却不具备函数的性质。

内联扩展是用来**消除函数调用时的时间开销**。它通常用于频繁执行的函数，对于小内存空间的函数非常受益。

使用内联函数的时候要注意：

* 递归函数不能定义为内联函数。
* 内联函数一般适合于不存在while和switch等复杂的结构且只有1~5条语句的小函数上，否则编译系统将该函数视为普通函数。
* 内联函数只能先定义后使用，否则编译系统也会把它认为是普通函数。
* 对内联函数不能进行异常的接口声明。

#### 递归函数

函数实现了代码的重用，大大简化缩短了程序员的工作量，更使得程序的可读性大大提高，函数的递归调用更是一种简化程序代码的方法；递归调用其实机是函数自己调用自己。

下面的实例是一个求1+2+3.....+n的递归实例：

* #include <stdio.h> //导如输入输出头文件
* int sum(int n);//声明函数
* int main(){
* //主函数
* int a=sum(4);
* printf("%d",a);
* return 0;
* }
* int sum(int n){
* //求和函数实现
* //如果n为1，无需求和，直接返回1
* if(n==1){
* return 1;
* }
* //如果n大于1，就返回n加上1+2+3.....+n-1的和
* return n+sum(n-1);
* }

可以发现，递归的代码很少，但是，递归也有缺点，递归占用的内存要比递推大，而且时间也要比递推长。

#### 关于 main 函数的参数

在有些很专业的书会看到如下代码

int main( int argc, char \*argv[] )

上面的代码中 main 函数带了参数。

但是有时又会看见main函数没有参数，如下：

int main()

**那么 main 函数到底有没有参数，有没有参数会不会有什么影响？**

main 函数其实与我们写的函数没有什么区别，它也会有自己的参数。

argc 和 argv 是 main 函数的形式参数。

这两个形式参数的类型是系统规定的。如果 main 函数要带参数，就是这两个类型的参数；否则main函数就没有参数。

变量名称argc和argv是常规的名称，当然也可以换成其他名称。在传入参数后main函数收到参数后就会做自己的事。那么，实际参数是如何传递给main函数的argc和argv的呢？我们知道，C程序在编译和链接后，都生成一个exe文件，执行该exe文件时，可以直接执行；也可以在命令行下带参数执行，命令行执行的形式为：可执行文件名称 参数1 参数2 ... ... 参数n。可执行文件名称和参数、参数之间均使用空格隔开。

如果按照这种方法执行，命令行字符串将作为实际参数传递给main函数。具体为：

 (1) 可执行文件名称和所有参数的个数之和传递给 argc；

 (2) 可执行文件名称（包括路径名称）作为一个字符串，首地址被赋给 argv[0]，参数1也作为一个字符串，首地址被赋给 argv[1]，... ...依次类推。

#### C 的形参与实参

在 C 语言中，形参与实参虽然很简单，但是，是大家比较容易混淆的一个点，这里将为大家详细的讲解。

**概念：**从字面上理解，所谓形式参数即只只是声明了一个作为参数的变量，并未直接进行赋值使用，而实际参数则相反。

如下例

#include <stdio.h>

int test(int,int); // 形参，只声明

int main()

{

int a,b;

printf("%d",test(5,3)); // 实参，已赋值

}

int test(int a,int b) // 形参

{

a=a+b;

return a;

}

像上面在 test() 函数里只声明了最为参数的变量，而 main() 函数里则对它赋了值。

#### 函数参数传递常用的三种方式

示例程序均以交换两个整数为例。

##### ****1. 值传递****

#include <stdio.h>

void swap(int x, int y);

void swap(int x, int y)

{

    int temp;

    temp = x;

    x = y;

    y = temp;

}

int main( int argc, char \*argv[] )

{

    int a = 5;

    int b = 10;

    swap(a, b); //调用交换函数

    printf("交换结果为 a = %d, b = %d\n",a,b);

    return 0;

}

由于值传递是单向传递，传递过程中只是改变了形参的数值，并未改变实参的数值，因此并不会改变a和b原有的值。

##### ****2. 指针传递****

#include <stdio.h>

void swap(int \*x, int \*y);

void swap(int \*x, int \*y)

{

    int temp;

    temp = \*x;

    \*x = \*y;

    \*y = temp;

}

int main( int argc, char \*argv[] )

{

    int a = 5;

    int b = 10;

    swap(&a, &b); //调用交换函数

    printf("交换结果为 a = %d, b = %d\n",a,b);

    return 0;

}

指针传递过程中，将a和b的地址分别传递给了x和y，在函数体内部改变了a、b所在地址的值，即交换了a、b的数值。

##### ****3. 引用传递【****C语言中是没有引用传递，这是C++中语言特性****】****

#include <stdio.h>

void swap(int &x, int &y);

void swap(int &x, int &y)

{

    int temp;

    temp = x;

    x = y;

    y = temp;

}

int main( int argc, char \*argv[] )

{

    int a = 5;

    int b = 10;

    swap(a, b); //调用交换函数

    printf("交换结果为 a = %d, b = %d\n",a,b);

    return 0;

}

引用传递中，在调用swap(a, b);时函数会用a、b分别代替x、y，即x、y分别引用了a、b变量，这样函数体中实际参与运算的其实就是实参a、b本身，因此也能达到交换数值的目的。

**注：**严格来说，C语言中是没有引用传递，这是C++中语言特性，因此在.c文件中使用引用传递会导致程序编译出错。

#### ★占位符

占位符就是先占住一个固定的位置，等着你再往里面添加内容的符号，广泛用于计算机中各类文档的编辑。

格式占位符(%)是在C/C++语言中格式输入函数，如 scanf、printf 等函数中使用。其意义就是起到格式占位的意思，表示在该位置有输入或者输出。

**%d, %i** 代表整数

**%f** 浮点

**%s** 字符串

**%c** char

**%p** 指针

**%fL** 长log

**%e** 科学计数

**%g** 小数或科学计数。

**%a,%A** 读入一个浮点值(仅C99有效)。

**%c** 读入一个字符。

**%d** 读入十进制整数。

**%i** 读入十进制，八进制，十六进制整数。

**%o** 读入八进制整数。

**%x,%X** 读入十六进制整数。

**%s** 读入一个字符串，遇空格、制表符或换行符结束。

**%f,%F,%e,%E,%g,%G** 用来输入实数，可以用小数形式或指数形式输入。

**%p** 读入一个指针。

**%u**读入一个无符号十进制整数。

**%n** 至此已读入值的等价字符数。

**%[]** 扫描字符集合。

**%%** 读 % 符号

**实例：**

scanf("%d,%d,%d",&a,&b,&c); // 从键盘输入三个整数，用逗号分隔

scanf("%c", &s); // 从键盘输入一个字符

scanf("%f", &f); // 从键盘输入一个浮点型数据

printf("%d\n",a); // 输出一个整数

printf("%f\n",b); // 输出一个浮点数

printf("%s\n",c); // 输出一个字符, 其中\n表示换行

#### 递归

<http://www.runoob.com/cprogramming/c-recursion.html>

递归指的是在函数的定义中使用函数自身的方法。

C 语言支持递归，即一个函数可以调用其自身。但在使用递归时，程序员需要注意定义一个从函数退出的条件，否则会进入死循环。

递归函数在解决许多数学问题上起了至关重要的作用，比如计算一个数的阶乘、生成斐波那契数列，等等。

递归是一个简洁的概念，同时也是一种很有用的手段。但是，使用递归是要付出代价的。与直接的语句(如while循环)相比，递归函数会耗费更多的运行时间，并且要占用大量的栈空间。递归函数每次调用自身时，都需要把它的状态存到栈中，以便在它调用完自身后，程序可以返回到它原来的状态。未经精心设计的递归函数总是会带来麻烦。

采用递归方法来解决问题，必须符合以下三个条件：

1、可以把要解决的问题转化为一个新问题，而这个新的问题的解决方法仍与原来的解决方法相同，只是所处理的对象有规律地递增或递减。

说明：解决问题的方法相同，调用函数的参数每次不同（有规律的递增或递减），如果没有规律也就不能适用递归调用。

2、可以应用这个转化过程使问题得到解决。

说明：使用其他的办法比较麻烦或很难解决，而使用递归的方法可以很好地解决问题。

3、必定要有一个明确的结束递归的条件。

说明：一定要能够在适当的地方结束递归调用。不然可能导致系统崩溃。

##### 数的阶乘

下面的实例使用递归函数计算一个给定的数的阶乘：

#include <stdio.h>

double factorial(unsigned int i)

{

if(i <= 1)

{

return 1;

}

return i \* factorial(i - 1);

}

int main()

{

int i = 15;

printf("%d 的阶乘为 %f\n", i, factorial(i));

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

15 的阶乘为 1307674368000.000000

##### 斐波那契数列

下面的实例使用递归函数生成一个给定的数的斐波那契数列：

#include <stdio.h>

int fibonaci(int i)

{

if(i == 0)

{

return 0;

}

if(i == 1)

{

return 1;

}

return fibonaci(i-1) + fibonaci(i-2);

}

int main()

{

int i;

for (i = 0; i < 10; i++)

{

printf("%d\t\n", fibonaci(i));

}

return 0;

}

### 内置函数

#### 随机数

srand((unsigned)time(NULL))是初始化随机函数种子：

* 1、是拿当前系统时间作为种子，由于时间是变化的，种子变化，可以产生不相同的随机数。计算机中的随机数实际上都不是真正的随机数，如果两次给的种子一样，是会生成同样的随机序列的。 所以，一般都会以当前的时间作为种子来生成随机数，这样更加的随机。
* 2、使用时，参数可以是unsigned型的任意数据，比如srand（10）；
* 3、如果不使用srand，用rand（）产生的随机数，在多次运行，结果是一样的。

参考如下：

void test\_rand(void)

{

    unsigned long n;

    srand((unsigned)time(NULL));

    for(int i = 0; i < 100; i++)

    {

        n = rand();

        printf("d\n", n);

    }

}

## 基本功能

### 内存管理

<http://www.runoob.com/cprogramming/c-memory-management.html>

本章将讲解 C 中的动态内存管理。C 语言为内存的分配和管理提供了几个函数。这些函数可以在 **<stdlib.h>** 头文件中找到。

|  |  |
| --- | --- |
| **序号** | **函数和描述** |
| 1 | **void \*calloc(int num, int size);** 在内存中动态地分配 num 个长度为 size 的连续空间，并将每一个字节都初始化为 0。所以它的结果是分配了 num\*size 个字节长度的内存空间，并且每个字节的值都是0。 |
| 2 | **void free(void \*address);** 该函数释放 address 所指向的内存块,释放的是动态分配的内存空间。 |
| 3 | **void \*malloc(int num);** 在堆区分配一块指定大小的内存空间，用来存放数据。这块内存空间在函数执行完成后不会被初始化，它们的值是未知的。 |
| 4 | **void \*realloc(void \*address, int newsize);** 该函数重新分配内存，把内存扩展到 **newsize**。 |

**注意：**void \* 类型表示未确定类型的指针。C、C++ 规定 void \* 类型可以通过类型转换强制转换为任何其它类型的指针。

#### 动态分配内存

编程时，如果您预先知道数组的大小，那么定义数组时就比较容易。例如，一个存储人名的数组，它最多容纳 100 个字符，所以您可以定义数组，如下所示：

char name[100];

但是，如果您预先不知道需要存储的文本长度，例如您向存储有关一个主题的详细描述。在这里，我们需要定义一个指针，该指针指向未定义所需内存大小的字符，后续再根据需求来分配内存，如下所示：

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main()

{

char name[100];

char \*description;

strcpy(name, "Zara Ali");

/\* 动态分配内存 \*/

description = malloc( 200 \* sizeof(char) );

if( description == NULL )

{

fprintf(stderr, "Error - unable to allocate required memory\n");

}

else

{

strcpy( description, "Zara ali a DPS student in class 10th");

}

printf("Name = %s\n", name );

printf("Description: %s\n", description );

}

当上面的代码被编译和执行时，它会产生下列结果：

Name = Zara Ali

Description: Zara ali a DPS student in class 10th

上面的程序也可以使用 **calloc()** 来编写，只需要把 malloc 替换为 calloc 即可，如下所示：

calloc(200, sizeof(char));

当动态分配内存时，您有完全控制权，可以传递任何大小的值。而那些预先定义了大小的数组，一旦定义则无法改变大小。

#### 重新调整内存的大小和释放内存

当程序退出时，操作系统会自动释放所有分配给程序的内存，但是，建议您在不需要内存时，都应该调用函数 **free()** 来释放内存。

或者，您可以通过调用函数 **realloc()** 来增加或减少已分配的内存块的大小。让我们使用 realloc() 和 free() 函数，再次查看上面的实例：

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main()

{

char name[100];

char \*description;

strcpy(name, "Zara Ali");

/\* 动态分配内存 \*/

description = malloc( 30 \* sizeof(char) );

if( description == NULL )

{

fprintf(stderr, "Error - unable to allocate required memory\n");

}

else

{

strcpy( description, "Zara ali a DPS student.");

}

/\* 假设您想要存储更大的描述信息 \*/

description = realloc( description, 100 \* sizeof(char) );

if( description == NULL )

{

fprintf(stderr, "Error - unable to allocate required memory\n");

}

else

{

strcat( description, "She is in class 10th");

}

printf("Name = %s\n", name );

printf("Description: %s\n", description );

/\* 使用 free() 函数释放内存 \*/

free(description);

}

当上面的代码被编译和执行时，它会产生下列结果：

Name = Zara Ali

Description: Zara ali a DPS student.She is in class 10th

您可以尝试一下不重新分配额外的内存，strcat() 函数会生成一个错误，因为存储 description 时可用的内存不足。

### 命令行参数

执行程序时，可以从命令行传值给 C 程序。这些值被称为**命令行参数**，它们对程序很重要，特别是当您想从外部控制程序，而不是在代码内对这些值进行硬编码时，就显得尤为重要了。

命令行参数是使用 main() 函数参数来处理的，其中，**argc** 是指传入参数的个数，**argv[]** 是一个指针数组，指向传递给程序的每个参数。下面是一个简单的实例，检查命令行是否有提供参数，并根据参数执行相应的动作：

#include <stdio.h>

int main( int argc, char \*argv[] )

{

if( argc == 2 )

{

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 )

{

printf("Too many arguments supplied.\n");

}

else

{

printf("One argument expected.\n");

}

}

应当指出的是，**argv[0]** 存储程序的名称，**argv[1]** 是一个指向第一个命令行参数的指针，\*argv[n] 是最后一个参数。如果没有提供任何参数，argc 将为 1，否则，如果传递了一个参数，**argc** 将被设置为 2。

多个命令行参数之间用空格分隔，但是如果参数本身带有空格，那么传递参数的时候应把参数放置在双引号 "" 或单引号 '' 内部。让我们重新编写上面的实例，有一个空间，那么你可以通过这样的观点，把它们放在双引号或单引号""""。让我们重新编写上面的实例，向程序传递一个放置在双引号内部的命令行参数：

#include <stdio.h>

int main( int argc, char \*argv[] )

{

printf("Program name %s\n", argv[0]);

if( argc == 2 )

{

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 )

{

printf("Too many arguments supplied.\n");

}

else

{

printf("One argument expected.\n");

}

}

使用一个用空格分隔的简单参数，参数括在双引号中，编译并执行上面的代码，它会产生下列结果：

$./a.out "testing1 testing2"

Progranm name ./a.out

The argument supplied is testing1 testing2

### 输入输出

<http://www.runoob.com/cprogramming/c-input-output.html>

#### 标准文件

C 语言把所有的设备都当作文件。所以设备（比如显示器）被处理的方式与文件相同。以下三个文件会在程序执行时自动打开，以便访问键盘和屏幕。



文件指针是访问文件的方式，本节将讲解如何从屏幕读取值以及如何把结果输出到屏幕上。

C 语言中的 I/O (输入/输出) 通常使用 printf() 和 scanf() 两个函数。

scanf() 函数用于从标准输入（键盘）读取并格式化， printf() 函数发送格式化输出到标准输出（屏幕）。

#### getchar() & putchar() 函数：读写单一字符

**int getchar(void)** 函数从屏幕读取下一个可用的字符，并把它返回为一个整数。这个函数在同一个时间内只会读取一个单一的字符。您可以在循环内使用这个方法，以便从屏幕上读取多个字符。

**int putchar(int c)** 函数把字符输出到屏幕上，并返回相同的字符。这个函数在同一个时间内只会输出一个单一的字符。您可以在循环内使用这个方法，以便在屏幕上输出多个字符。



#### gets() & puts() 函数【linux系统下需要这样编译：不支持 gets 与 puts, 需要用 fgets 和 fputs。】

linux系统下需要这样编译：不支持 gets 与 puts, 需要用 fgets 和 fputs。

**char \*gets(char \*s)** 函数从 **stdin** 读取一行到 **s** 所指向的缓冲区，直到一个终止符或 EOF。

**int puts(const char \*s)** 函数把字符串 s 和一个尾随的换行符写入到 **stdout**。

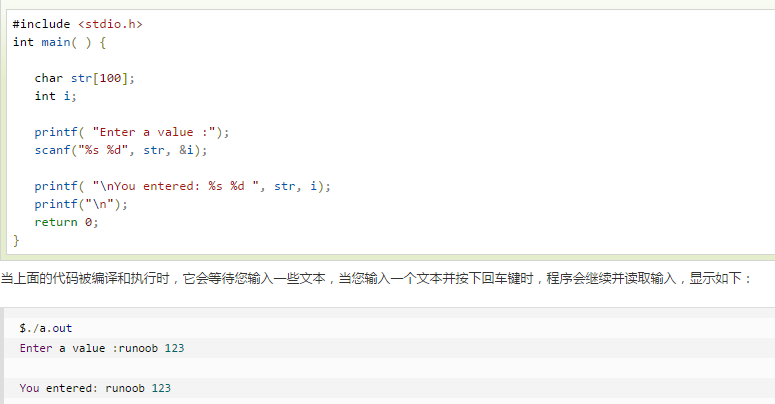


#### scanf() 和 printf() 函数

**int scanf(const char \*format, ...)** 函数从标准输入流 **stdin** 读取输入，并根据提供的 **format** 来浏览输入。

**int printf(const char \*format, ...)** 函数把输出写入到标准输出流 **stdout**，并根据提供的格式产生输出。

**format** 可以是一个简单的常量字符串，但是您可以分别指定 %s、%d、%c、%f 等来输出或读取字符串、整数、字符或浮点数。还有许多其他可用的格式选项，可以根据需要使用。如需了解完整的细节，可以查看这些函数的参考手册。现在让我们通过下面这个简单的实例来加深理解：



#### fgets 和 fputs函数【p = fgets(c, 100, stdin );】

linux系统下需要这样编译：不支持 gets 与 puts, 需要用 fgets 和 fputs。

将以下代码放到 test.c 文件：



#include <stdio.h>

#include <string.h>

int main ()

{

char \*p;

char c[100];

printf("Enter a value:");

p = fgets(c, 100, stdin );

printf("res=%s", p);

printf("\nyou entered:");

fputs(c, stdout );

return 0;

}

#### gets()与fgets()

**gets()**

gets函数原型：char\*gets(char\*buffer);//读取字符到数组：gets(str);str为数组名。

gets函数功能：从键盘上输入字符，直至接受到换行符或EOF时停止，并将读取的结果存放在buffer指针所指向的字符数组中。

读取的换行符被转换为null值，做为字符数组的最后一个字符，来结束字符串。

**注意：**gets函数由于没有指定输入字符大小，所以会无限读取，一旦输入的字符大于数组长度，就会发生内存越界，

从而造成程序崩溃或其他数据的错误。

**fgets()**

fgets函数原型：char \*fgets(char \*s, int n, FILE \*stream);//我们平时可以这么使用：fgets(str, sizeof(str), stdin);

其中str为数组首地址，sizeof(str)为数组大小，stdin表示我们从键盘输入数据。

fgets函数功能：从文件指针stream中读取字符，存到以s为起始地址的空间里，直到读完N-1个字符，或者读完一行。

**注意：**调用fgets函数时，最多只能读入n-1个字符。读入结束后，系统将自动在最后加'\0'，并以str作为函数值返回。

借用教程实例，我把char str[100] 改为 char str[5] 。





#### Windows、Unix、Mac不同操作系统的换行问题 回车符\r和换行符\n

**一、概念：**

**换行符‘\n’**和**回车符‘\r’**

（1）换行符就是另起一行  --- '**\n**' 10 换行（newline）

（2）回车符就是回到一行的开头 --- '**\r**' 13 回车（return）

所以我们平时编写文件的回车符应该确切来说叫做**回车换行符**

**CR: 回车(Carriage Return) \rLF: 换行(Line Feed) \n**

**二、应用：**

（1）在微软的MS-DOS和Windows中，使用“回车CR('\r')”和“换行LF('\n')”两个字符作为换行符;

（2）Windows系统里面，每行结尾是 回车+换行(CR+LF)，即“\r\n”；

（3）Unix系统里，每行结尾只有 换行LF，即“\n”；

（4）Mac系统里，每行结尾是 回车CR 即'\r'。

Mac OS 9 以及之前的系统的换行符是 CR，从 Mac OS X （后来改名为“OS X”）开始的换行符是 LF即‘\n'，和Unix/Linux统一了。

**三、影响：**

（1）一个直接后果是，Unix/Mac系统下的文件在Windows里打开的话，所有文字会变成一行；

（2）而Windows里的文件在Unix/Mac下打开的话，在每行的结尾可能会多出一个^M符号。

（3）Linux保存的文件在windows上用记事本看的话会出现黑点。

**四、可以相互转换**：

在linux下，命令unix2dos 是把linux文件格式转换成windows文件格式，命令dos2unix 是把windows格式转换成linux文件格式。

在不同平台间使用FTP软件传送文件时, 在ascii文本模式传输模式下, 一些FTP客户端程序会自动对换行格式进行转换. 经过这种传输的文件字节数可能会发生变化.

 如果你不想ftp修改原文件, 可以使用bin模式(二进制模式)传输文本。

一个程序在windows上运行就生成CR/LF换行格式的文本文件，而在Linux上运行就生成LF格式换行的文本文件。

学 C 语言的时候，字符输入曾经困扰过我，例如这段代码：

int i;

char c;

scanf("%d%c", &i,&c);

这时候变量 c 中存储的往往不是你想输入的字符，而是一个空格，然后我们又会这样来写：

int i;

char c;

scanf("%d", &i);

scanf("%d", &c);

这时候，我们发现，根本没有输入字符C的机会，这是为什么？因为输入流是有缓冲区的，我们输入的字符存储在那，然后再赋值给我们的变量。我们可以这样改：

int i;

char c;

scanf("%d", &i);

while((c=getchar())==' ' || c=='\n');

c = getchar();

这个办法是一直读取，读到没有空格和换行就跳出循环，但是有一个更好的解决办法；

int i;

char c;

scanf("%d%[^' '^'\n']", &i, &c);

这是用正则表达来控制输入格式为非空格非换行。

#### 在进行输出时，若要用到用来输出实数的 f 格式符（以小数形式输出），有以下几种用法：

**1、基本型，用 %f**

不指定输出类型的长度，用系统根据情况决定，一般是实数中的整数部分全部输出，小数部分输出六位。例：

#include<stdio.h>

int main()

{

    double a=1.0;

    printf("%f\n",a/3);

    return 0;

}

运行结果：**0.333333**

**2、指定数据宽度和小数位数，用 %m.nf**

例：将上个程序的双精度变量 a 输出 15 位小数，用 **%20.15f** 的格式声明，指定输出的数据占 20 列，其中包括 15 位小数。改动上面程序如下：

#include<stdio.h>

int main()

{

    double a=1.0;

    printf("%20.15f\n",a/3);

    return 0;

}

运行结果：**0.333333333333333**

注意在 0 的前面有 3 个空格，且双精度数只保证 15 位有效数字的准确性。

**3、输出的数据相左对齐，用 %-m.nf**

在 **m.n** 前加一个负号，其作用与 **%m.nf** 形式作用基本相同，但当数据长度不长过 **m** 时，数据向左靠，右端补空格。

### 文件读写

<http://www.runoob.com/cprogramming/c-file-io.html>

上一章我们讲解了 C 语言处理的标准输入和输出设备。本章我们将介绍 C 程序员如何创建、打开、关闭文本文件或二进制文件。

一个文件，无论它是文本文件还是二进制文件，都是代表了一系列的字节。C 语言不仅提供了访问顶层的函数，也提供了底层（OS）调用来处理存储设备上的文件。本章将讲解文件管理的重要调用。

#### 打开文件

您可以使用 **fopen( )** 函数来创建一个新的文件或者打开一个已有的文件，这个调用会初始化类型 **FILE** 的一个对象，类型 **FILE** 包含了所有用来控制流的必要的信息。下面是这个函数调用的原型：

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

在这里，**filename** 是字符串，用来命名文件，访问模式 **mode** 的值可以是下列值中的一个：

|  |  |
| --- | --- |
| **模式** | **描述** |
| r | 打开一个已有的文本文件，允许读取文件。 |
| w | 打开一个文本文件，允许写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会从文件的开头写入内容。如果文件存在，则该会被截断为零长度，重新写入。 |
| a | 打开一个文本文件，以追加模式写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会在已有的文件内容中追加内容。 |
| r+ | 打开一个文本文件，允许读写文件。 |
| w+ | 打开一个文本文件，允许读写文件。如果文件已存在，则文件会被截断为零长度，如果文件不存在，则会创建一个新文件。 |
| a+ | 打开一个文本文件，允许读写文件。如果文件不存在，则会创建一个新文件。读取会从文件的开头开始，写入则只能是追加模式。 |

如果处理的是二进制文件，则需使用下面的访问模式来取代上面的访问模式：

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

#### 关闭文件

为了关闭文件，请使用 fclose( ) 函数。函数的原型如下：

int fclose( FILE \*fp );

如果成功关闭文件，**fclose( )** 函数返回零，如果关闭文件时发生错误，函数返回 **EOF**。这个函数实际上，会清空缓冲区中的数据，关闭文件，并释放用于该文件的所有内存。EOF 是一个定义在头文件 **stdio.h** 中的常量。

C 标准库提供了各种函数来按字符或者以固定长度字符串的形式读写文件。

#### 写入文件【fprintf，fputs】

下面是把字符写入到流中的最简单的函数：

int fputc( int c, FILE \*fp );

函数 **fputc()** 把参数 c 的字符值写入到 fp 所指向的输出流中。如果写入成功，它会返回写入的字符，如果发生错误，则会返回 **EOF**。您可以使用下面的函数来把一个以 null 结尾的字符串写入到流中：

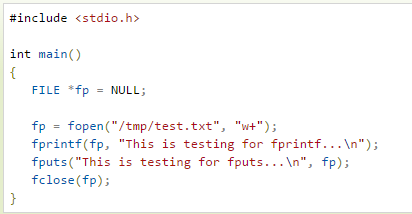
int fputs( const char \*s, FILE \*fp );

函数 **fputs()** 把字符串 **s** 写入到 fp 所指向的输出流中。如果写入成功，它会返回一个非负值，如果发生错误，则会返回 **EOF**。您也可以使用 **int fprintf(FILE \*fp,const char \*format, ...)** 函数来写把一个字符串写入到文件中。尝试下面的实例：

***注意：****请确保您有可用的****tmp****目录，如果不存在该目录，则需要在您的计算机上先创建该目录。*

***/tmp****一般是 Linux 系统上的临时目录，如果你在 Windows 系统上运行，则需要修改为本地环境中已存在的目录，例如:****C:\tmp****、****D:\tmp****等。*

实例



当上面的代码被编译和执行时，它会在 /tmp 目录中创建一个新的文件 **test.txt**，并使用两个不同的函数写入两行。接下来让我们来读取这个文件。

#### 读取文件【fscanf，fgets】

下面是从文件读取单个字符的最简单的函数：

int fgetc( FILE \* fp );

**fgetc()** 函数从 fp 所指向的输入文件中读取一个字符。返回值是读取的字符，如果发生错误则返回 **EOF**。下面的函数允许您从流中读取一个字符串：

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

函数 **fgets()** 从 fp 所指向的输入流中读取 n - 1 个字符。它会把读取的字符串复制到缓冲区 **buf**，并在最后追加一个 **null** 字符来终止字符串。

如果这个函数在读取最后一个字符之前就遇到一个换行符 '\n' 或文件的末尾 EOF，则只会返回读取到的字符，包括换行符。您也可以使用 **int fscanf(FILE \*fp, const char \*format, ...)** 函数来从文件中读取字符串，但是在遇到第一个空格字符时，它会停止读取。

#include <stdio.h>

int main()

{

FILE \*fp = NULL;

char buff[255];

fp = fopen("/tmp/test.txt", "r");

fscanf(fp, "%s", buff);

printf("1: %s\n", buff );

fgets(buff, 255, (FILE\*)fp);

printf("2: %s\n", buff );

fgets(buff, 255, (FILE\*)fp);

printf("3: %s\n", buff );

fclose(fp);

}

当上面的代码被编译和执行时，它会读取上一部分创建的文件，产生下列结果：

1: This

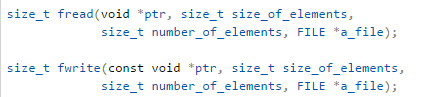
2: is testing for fprintf...

3: This is testing for fputs...

首先，**fscanf()** 方法只读取了 **This**，因为它在后边遇到了一个空格。其次，调用 **fgets()** 读取剩余的部分，直到行尾。最后，调用 **fgets()** 完整地读取第二行。

#### 二进制 I/O 函数【fread，fwrite】

下面两个函数用于二进制输入和输出：



这两个函数都是用于存储块的读写 - 通常是数组或结构体。

#### fseek 可以移动文件指针到指定位置读,或插入写:

int fseek(FILE \*stream, long offset, int whence);

fseek 设置当前读写点到 offset 处, whence 可以是 SEEK\_SET,SEEK\_CUR,SEEK\_END 这些值决定是从文件头、当前点和文件尾计算偏移量 offset。

你可以定义一个文件指针**FILE \*fp**,当你打开一个文件时，文件指针指向开头，你要指到多少个字节，只要控制偏移量就好，例如, 相对当前位置往后移动一个字节：**fseek(fp,1,SEEK\_CUR);** 中间的值就是偏移量。 如果你要往前移动一个字节，直接改为负值就可以：**fseek(fp,-1,SEEK\_CUR)**。

执行以下实例前，确保当前目录下 **test.txt** 文件已创建：

#include <stdio.h>

int main(){

FILE \*fp = NULL;

fp = fopen("test.txt", "r+"); // 确保 test.txt 文件已创建

fprintf(fp, "This is testing for fprintf...\n");

fseek(fp, 10, SEEK\_SET);

if (fputc(65,fp) == EOF) {

printf("fputc fail");

}

fclose(fp);

}

执行结束后，打开 test.txt 文件：

This is teAting for fprintf...

**注意：** 只有用 **r+** 模式打开文件才能插入内容，**w** 或 **w+** 模式都会清空掉原来文件的内容再来写，**a** 或 **a+** 模式即总会在文件最尾添加内容，哪怕用 fseek() 移动了文件指针位置。

### 预处理器

<http://www.runoob.com/cprogramming/c-preprocessors.html>

**C 预处理器**不是编译器的组成部分，但是它是编译过程中一个单独的步骤。简言之，C 预处理器只不过是一个文本替换工具而已，它们会指示编译器在实际编译之前完成所需的预处理。我们将把 C 预处理器（C Preprocessor）简写为 CPP。

所有的预处理器命令都是以井号（#）开头。它必须是第一个非空字符，为了增强可读性，预处理器指令应从第一列开始。下面列出了所有重要的预处理器指令：

|  |  |
| --- | --- |
| **指令** | **描述** |
| #define | 定义宏 |
| #include | 包含一个源代码文件 |
| #undef | 取消已定义的宏 |
| #ifdef | 如果宏已经定义，则返回真 |
| #ifndef | 如果宏没有定义，则返回真 |
| #if | 如果给定条件为真，则编译下面代码 |
| #else | #if 的替代方案 |
| #elif | 如果前面的 #if 给定条件不为真，当前条件为真，则编译下面代码 |
| #endif | 结束一个 #if……#else 条件编译块 |
| #error | 当遇到标准错误时，输出错误消息 |
| #pragma | 使用标准化方法，向编译器发布特殊的命令到编译器中 |

#### 预处理器实例

分析下面的实例来理解不同的指令。

#define MAX\_ARRAY\_LENGTH 20

这个指令告诉 CPP 把所有的 MAX\_ARRAY\_LENGTH 替换为 20。使用 *#define* 定义常量来增强可读性。

#include <stdio.h>

#include "myheader.h"

这些指令告诉 CPP 从**系统库**中获取 stdio.h，并添加文本到当前的源文件中。下一行告诉 CPP 从本地目录中获取 **myheader.h**，并添加内容到当前的源文件中。

#undef FILE\_SIZE

#define FILE\_SIZE 42

这个指令告诉 CPP 取消已定义的 FILE\_SIZE，并定义它为 42。

#ifndef MESSAGE

#define MESSAGE "You wish!"

#endif

这个指令告诉 CPP 只有当 MESSAGE 未定义时，才定义 MESSAGE。

#ifdef DEBUG

/\* Your debugging statements here \*/

#endif

这个指令告诉 CPP 如果定义了 DEBUG，则执行处理语句。在编译时，如果您向 gcc 编译器传递了 *-DDEBUG* 开关量，这个指令就非常有用。它定义了 DEBUG，您可以在编译期间随时开启或关闭调试。

#### 预定义宏

ANSI C 定义了许多宏。在编程中您可以使用这些宏，但是不能直接修改这些预定义的宏。

|  |  |
| --- | --- |
| **宏** | **描述** |
| \_\_DATE\_\_ | 当前日期，一个以 "MMM DD YYYY" 格式表示的字符常量。 |
| \_\_TIME\_\_ | 当前时间，一个以 "HH:MM:SS" 格式表示的字符常量。 |
| \_\_FILE\_\_ | 这会包含当前文件名，一个字符串常量。 |
| \_\_LINE\_\_ | 这会包含当前行号，一个十进制常量。 |
| \_\_STDC\_\_ | 当编译器以 ANSI 标准编译时，则定义为 1。 |

让我们来尝试下面的实例：



#### 预处理器运算符

C 预处理器提供了下列的运算符来帮助您创建宏：

##### 宏延续运算符（\）

一个宏通常写在一个单行上。但是如果宏太长，一个单行容纳不下，则使用宏延续运算符（\）。例如：

#define message\_for(a, b) \

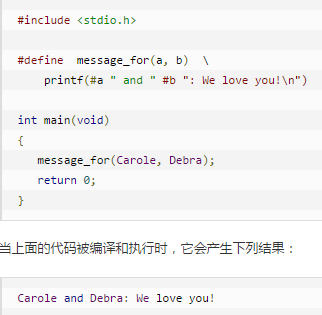
printf(#a " and " #b ": We love you!\n")

##### 字符串常量化运算符（#）

在宏定义中，当需要把一个宏的参数转换为字符串常量时，则使用字符串常量化运算符（#）。在宏中使用的该运算符有一个特定的参数或参数列表。例如：

#define message\_for(a, b) \

printf(#a " and " #b ": We love you!\n")



##### ★★标记粘贴运算符（##）

宏定义内的标记粘贴运算符（##）会合并两个参数。它允许在宏定义中两个独立的标记被合并为一个标记。例如：

#define tokenpaster(n) printf ("token" #n " = %d", token##n)



这个实例演示了 token##n 会连接到 token34 中，在这里，我们使用了**字符串常量化运算符（#）**和**标记粘贴运算符（##）**。

##### defined() 运算符

预处理器 **defined** 运算符是用在常量表达式中的，用来确定一个标识符是否已经使用 #define 定义过。如果指定的标识符已定义，则值为真（非零）。如果指定的标识符未定义，则值为假（零）。下面的实例演示了 defined() 运算符的用法：



#### 参数化的宏

CPP 一个强大的功能是可以使用参数化的宏来模拟函数。例如，下面的代码是计算一个数的平方：

int square(int x) {

return x \* x;

}

我们可以使用宏重写上面的代码，如下：

#define square(x) ((x) \* (x))

在使用带有参数的宏之前，必须使用 **#define** 指令定义。参数列表是括在圆括号内，且必须紧跟在宏名称的后边。宏名称和左圆括号之间不允许有空格。例如：

#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)

{

printf("Max between 20 and 10 is %d\n", MAX(10, 20));

return 0;

}

##### 使用#define含参时，参数括号很重要，如上例中省略括号会导致运算错误：

#include <stdio.h>

#define square(x) ((x) \* (x))

#define square\_1(x) (x \* x)

int main(void)

{

printf("square 5+4 is %d\n", square(5+4));

printf("square\_1 5+4 is %d\n", square\_1(5+4));

return 0;

}

输出结果为：

square 5+4 is 81

square\_1 5+4 is 29

原因:

square 等价于 （5+4）\*（5+4）=81

square\_1 等价于 5+4\*5+4=29

用#define宏定义将a,b交换，不使用中间变量，两种方法实现swap(x,y);

#include <stdio.h>

#define MAX(x,y) ((x>y)?(x):(y))

#define SWAP1(x,y) {x=x+y;y=x-y;x=x-y;}

#define SWAP2(x,y) {x=x^y;y=x^y;x=x^y;}

int main()

{

    int a,b;

    scanf("%d %d",&a,&b);

    printf("Max number is:%d\n",MAX(a,b));

    printf("交换前：x=%d,y=%d\n",a,b);

    SWAP1(a,b);

    printf("交换后：x=%d,y=%d\n",a,b);

    SWAP2(a,b);

    printf("再次交换后：x=%d,y=%d\n",a,b);

    return 0;

}

输出结果为：

2 4

Max number is:4

交换前：x=2,y=4

交换后：x=4,y=2

再次交换后：x=2,y=4

### 头文件

头文件是扩展名为 **.h** 的文件，包含了 C 函数声明和宏定义，被多个源文件中引用共享。

有两种类型的头文件：程序员编写的头文件和编译器自带的头文件。

在程序中要使用头文件，需要使用 C 预处理指令 **#include** 来引用它。前面我们已经看过 **stdio.h** 头文件，它是编译器自带的头文件。

引用头文件相当于复制头文件的内容，但是我们不会直接在源文件中复制头文件的内容，因为这么做很容易出错，特别在程序是由多个源文件组成的时候。

A simple practice in C 或 C++ 程序中，建议把所有的常量、宏、系统全局变量和函数原型写在头文件中，在需要的时候随时引用这些头文件。

#### 引用头文件的语法

使用预处理指令 **#include** 可以引用用户和系统头文件。它的形式有以下两种：

#include <file>

这种形式用于引用系统头文件。它在系统目录的标准列表中搜索名为 file 的文件。在编译源代码时，您可以通过 -I 选项把目录前置在该列表前。

#include "file"

这种形式用于引用用户头文件。它在包含当前文件的目录中搜索名为 file 的文件。在编译源代码时，您可以通过 -I 选项把目录前置在该列表前。

#### 引用头文件的操作

**#include** 指令会指示 C 预处理器浏览指定的文件作为输入。预处理器的输出包含了已经生成的输出，被引用文件生成的输出以及 **#include** 指令之后的文本输出。例如，如果您有一个头文件 header.h，如下：

char \*test (void);

和一个使用了头文件的主程序 *program.c*，如下：

int x;

#include "header.h"

int main (void)

{

puts (test ());

}

编译器会看到如下的代码信息：

int x;

char \*test (void);

int main (void)

{

puts (test ());

}

#### 只引用一次头文件

如果一个头文件被引用两次，编译器会处理两次头文件的内容，这将产生错误。为了防止这种情况，标准的做法是把文件的整个内容放在条件编译语句中，如下：

#ifndef HEADER\_FILE

#define HEADER\_FILE

the entire header file file

#endif

这种结构就是通常所说的包装器 **#ifndef**。当再次引用头文件时，条件为假，因为 HEADER\_FILE 已定义。此时，预处理器会跳过文件的整个内容，编译器会忽略它。

#### 有条件引用

有时需要从多个不同的头文件中选择一个引用到程序中。例如，需要指定在不同的操作系统上使用的配置参数。您可以通过一系列条件来实现这点，如下：

#if SYSTEM\_1

# include "system\_1.h"

#elif SYSTEM\_2

# include "system\_2.h"

#elif SYSTEM\_3

...

#endif

但是如果头文件比较多的时候，这么做是很不妥当的，预处理器使用宏来定义头文件的名称。这就是所谓的**有条件引用**。它不是用头文件的名称作为 **#include** 的直接参数，您只需要使用宏名称代替即可：

#define SYSTEM\_H "system\_1.h"

...

#include SYSTEM\_H

SYSTEM\_H 会扩展，预处理器会查找 system\_1.h，就像 **#include** 最初编写的那样。SYSTEM\_H 可通过 -D 选项被您的 Makefile 定义。

#### global.h 的头文件来包括所有的 **.h** 文件

在有多个 **.h** 文件和多个 **.c** 文件的时候，往往我们会用一个 **global.h** 的头文件来包括所有的 **.h** 文件，然后在除 **global.h** 文件外的头文件中 包含 **global.h** 就可以实现所有头文件的包含，同时不会乱。方便在各个文件里面调用其他文件的函数或者变量。

#ifndef \_GLOBAL\_H

#define \_GLOBAL\_H

#include <fstream>

#include <iostream>

#include <math.h>

#include <Config.h>

### 错误处理

<http://www.runoob.com/cprogramming/c-error-handling.html>

C 语言不提供对错误处理的直接支持，但是作为一种系统编程语言，它以返回值的形式允许您访问底层数据。在发生错误时，大多数的 C 或 UNIX 函数调用返回 1 或 NULL，同时会设置一个错误代码 **errno**，该错误代码是全局变量，表示在函数调用期间发生了错误。您可以在 errno.h 头文件中找到各种各样的错误代码。

所以，C 程序员可以通过检查返回值，然后根据返回值决定采取哪种适当的动作。开发人员应该在程序初始化时，把 errno 设置为 0，这是一种良好的编程习惯。0 值表示程序中没有错误。

#### errno、perror() 和 strerror()

C 语言提供了 **perror()** 和 **strerror()** 函数来显示与 **errno** 相关的文本消息。

* **perror()** 函数显示您传给它的字符串，后跟一个冒号、一个空格和当前 errno 值的文本表示形式。
* **strerror()** 函数，返回一个指针，指针指向当前 errno 值的文本表示形式。

让我们来模拟一种错误情况，尝试打开一个不存在的文件。您可以使用多种方式来输出错误消息，在这里我们使用函数来演示用法。另外有一点需要注意，您应该使用 **stderr** 文件流来输出所有的错误。

#include <stdio.h>

#include <errno.h>

#include <string.h>

extern int errno ;

int main ()

{

FILE \* pf;

int errnum;

pf = fopen ("unexist.txt", "rb");

if (pf == NULL)

{

errnum = errno;

fprintf(stderr, "错误号: %d\n", errno);

perror("通过 perror 输出错误");

fprintf(stderr, "打开文件错误: %s\n", strerror( errnum ));

}

else

{

fclose (pf);

}

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

错误号: 2

通过 perror 输出错误: No such file or directory

打开文件错误: No such file or directory

#### 被零除的错误

在进行除法运算时，如果不检查除数是否为零，则会导致一个运行时错误。

为了避免这种情况发生，下面的代码在进行除法运算前会先检查除数是否为零：

#include <stdio.h>

#include <stdlib.h>

main()

{

int dividend = 20;

int divisor = 0;

int quotient;

if( divisor == 0){

fprintf(stderr, "除数为 0 退出运行...\n");

exit(-1);

}

quotient = dividend / divisor;

fprintf(stderr, "quotient 变量的值为 : %d\n", quotient );

exit(0);

}

#### 程序退出状态

通常情况下，程序成功执行完一个操作正常退出的时候会带有值 EXIT\_SUCCESS。在这里，EXIT\_SUCCESS 是宏，它被定义为 0。

如果程序中存在一种错误情况，当您退出程序时，会带有状态值 EXIT\_FAILURE，被定义为 -1。所以，上面的程序可以写成：

#include <stdio.h>

#include <stdlib.h>

main()

{

int dividend = 20;

int divisor = 5;

int quotient;

if( divisor == 0){

fprintf(stderr, "除数为 0 退出运行...\n");

exit(EXIT\_FAILURE);

}

quotient = dividend / divisor;

fprintf(stderr, "quotient 变量的值为: %d\n", quotient );

exit(EXIT\_SUCCESS);

}

当上面的代码被编译和执行时，它会产生下列结果：

quotient 变量的值为 : 4

### 日志

## 编程实例

<http://www.runoob.com/cprogramming/c-examples.html> 【编程实例】

<http://www.runoob.com/cprogramming/c-100-examples.html> 【经典100例】

## C标准库

<http://www.runoob.com/cprogramming/c-standard-library.html>

C 语言是一种通用的、面向过程式的计算机程序设计语言。1972 年，为了移植与开发 UNIX 操作系统，丹尼斯·里奇在贝尔电话实验室设计开发了 C 语言。

C 语言是一种广泛使用的计算机语言，它与 Java 编程语言一样普及，二者在现代软件程序员之间都得到广泛使用。

C 标准库是一组 C 内置函数、常量和头文件，比如 <stdio.h>、<stdlib.h>、<math.h>，等等。这个标准库可以作为 C 程序员的参考手册。

### 编译/执行 C 程序

如果您想要在 Linux 服务器上学习 C 编程，但是又没有相关的配置环境，那么可以访问 [compileonline.com](http://www.compileonline.com/compile_c_online.php)。您只需进行简单的点击动作，即可在高端的服务器上体验真实的编程经验。这是完全免费的在线工具。

【在线编译器】 <http://www.compileonline.com/compile_c_online.php>

### 库函数



#### C 标准库 - <stdarg.h>

<http://www.runoob.com/cprogramming/c-standard-library-stdarg-h.html>

##### 简介

**stdarg.h** 头文件定义了一个变量类型 **va\_list** 和三个宏，这三个宏可用于在参数个数未知（即参数个数可变）时获取函数中的参数。

可变参数的函数通在参数列表的末尾是使用省略号(,...)定义的。

##### 库变量

下面是头文件 stdarg.h 中定义的变量类型：

|  |  |
| --- | --- |
| **序号** | **变量 & 描述** |
| 1 | **va\_list** 这是一个适用于 **va\_start()、va\_arg()** 和 **va\_end()** 这三个宏存储信息的类型。 |

##### 库宏

下面是头文件 stdarg.h 中定义的宏：

|  |  |
| --- | --- |
| **序号** | **宏 & 描述** |
| 1 | [void va\_start(va\_list ap, last\_arg)](http://www.runoob.com/cprogramming/c-macro-va_start.html) 这个宏初始化 **ap** 变量，它与 **va\_arg** 和 **va\_end** 宏是一起使用的。**last\_arg** 是最后一个传递给函数的已知的固定参数，即省略号之前的参数。 |
| 2 | [type va\_arg(va\_list ap, type)](http://www.runoob.com/cprogramming/c-macro-va_arg.html) 这个宏检索函数参数列表中类型为 **type** 的下一个参数。 |
| 3 | [void va\_end(va\_list ap)](http://www.runoob.com/cprogramming/c-macro-va_end.html) 这个宏允许使用了 **va\_start** 宏的带有可变参数的函数返回。如果在从函数返回之前没有调用 **va\_end**，则结果为未定义。 |

## 结束