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

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。

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。 |

**C 语言的基本数据类型有：**

char(1) short(2) int (4) long(4) float(4) double(8)

对比JAVA Boolean(1) byte(1) short (2) char(2) int(4) float(4) double(8) long(8)

**格式化输出**

int %d

short %d

long %ld

float %f

double %lf

char %c

%x 十六进制

%o 八进制

%s 字符串

%#x 地址

# C 存储类

# 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

# C 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
* 运算符优先级

|  |  |  |
| --- | --- | --- |
| **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 |

# C 判断

C 语言把任何**非零**和**非空**的值假定为 **true**，把**零**或 **null** 假定为 **false**。

## 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](https://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](https://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](https://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](https://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 |

C 语言规定，如果在定义函数时省略 extern，则默认为外部函数。

# C 作用域规则

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

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：

|  |  |
| --- | --- |
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

# C 数组

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

指针与数组的区别

char \*str="Hello";

char arrgs[]="Hello";

字符数组:

char \*p=(char\*)&arrgs == char \*p=arrgs ;

char \*p=(char\*)(&(arrgs[0])) == &arrgs == arrgs;

字符串指针:

char \*p=(char\*)&str != str; str==str[0] =='H'

得出结论字符数组的 首地址可以用 arrgs ，&arrgs,来表示并且它们相等，

但是指针&str和str是不同的,当遇到字符串指针时候要注意处理方式;

对于数组的初始化需要注意以下几点：

1) 可以只给部分元素赋值，当 **{ }** 中值的个数少于元素个数时，只给前面部分元素赋值。例如：

int a[10]={12, 19, 22 , 993, 344};

表示只给 **a[0]~a[4] 5** 个元素赋值，而后面 **5** 个元素自动初始化为 **0**。

当赋值的元素少于数组总体元素的时候，不同类型剩余的元素自动初始化值说明如下：

* 对于 short、int、long，就是整数 **0**；
* 对于 char，就是字符 **'\0'**；
* 对于 float、double，就是小数 **0.0**。

我们可以通过下面的形式将数组的所有元素初始化为 0：

int nums[10] = {0};

char str[10] = {0};

float scores[10] = {0.0};

由于剩余的元素会自动初始化为 0，所以只需要给第 0 个元素赋值为 0 即可。

2) 只能给元素逐个赋值，不能给数组整体赋值。例如给 10 个元素全部赋值为 1，只能写作：

int a[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

而不能写作：

int a[10] = 1;

# C 指针

学习 C 语言的指针既简单又有趣。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 **&** 运算符访问的地址，它表示了在内存中的一个地址。

|  |  |
| --- | --- |
| [指针的算术运算](https://www.runoob.com/cprogramming/c-pointer-arithmetic.html) | 可以对指针进行四种算术运算：++、--、+、- |
| [指针数组](https://www.runoob.com/cprogramming/c-array-of-pointers.html) | 可以定义用来存储指针的数组。 |
| [指向指针的指针](https://www.runoob.com/cprogramming/c-pointer-to-pointer.html) | C 允许指向指针的指针。 |
| [传递指针给函数](https://www.runoob.com/cprogramming/c-passing-pointers-to-functions.html) | 通过引用或地址传递参数，使传递的参数在调用函数中被改变。 |
| [从函数返回指针](https://www.runoob.com/cprogramming/c-return-pointer-from-functions.html) | C 允许函数返回指针到局部变量、静态变量和动态内存分配。 |

* **指针的一些复杂说明：**
* **int p;** -- 这是一个普通的整型变量
* **int \*p;** -- 首先从 p 处开始,先与\*结合,所以说明 p 是一个指针, 然后再与 int 结合, 说明指针所指向的内容的类型为 int 型。所以 p 是一个返回整型数据的指针。
* **int p[3]** -- 首先从 p 处开始,先与[] 结合,说明 p 是一个数组, 然后与 int 结合, 说明数组里的元素是整型的, 所以 p 是一个由整型数据组成的数组。
* **int \*p[3];** -- 首先从 p 处开始, 先与 [] 结合, 因为其优先级比 \* 高,所以 p 是一个数组, 然后再与 \* 结合, 说明数组里的元素是指针类型, 然后再与 int 结合, 说明指针所指向的内容的类型是整型的, 所以 p 是一个由返回整型数据的指针所组成的数组。
* **int (\*p)[3];** -- 首先从 p 处开始, 先与 \* 结合,说明 p 是一个指针然后再与 [] 结合(与"()"这步可以忽略,只是为了改变优先级), 说明指针所指向的内容是一个数组, 然后再与int 结合, 说明数组里的元素是整型的。所以 p 是一个指向由整型数据组成的数组的指针。
* **int \*\*p;** -- 首先从 p 开始, 先与 \* 结合, 说是 p 是一个指针, 然后再与 \* 结合, 说明指针所指向的元素是指针, 然后再与 int 结合, 说明该指针所指向的元素是整型数据。由于二级指针以及更高级的指针极少用在复杂的类型中, 所以后面更复杂的类型我们就不考虑多级指针了, 最多只考虑一级指针。
* **int p(int);** -- 从 p 处起,先与 () 结合, 说明 p 是一个函数, 然后进入 () 里分析, 说明该函数有一个整型变量的参数, 然后再与外面的 int 结合, 说明函数的返回值是一个整型数据。
* **int (\*p)(int);** -- 从 p 处开始, 先与指针结合, 说明 p 是一个指针, 然后与()结合, 说明指针指向的是一个函数, 然后再与()里的 int 结合, 说明函数有一个int 型的参数, 再与最外层的 int 结合, 说明函数的返回类型是整型, 所以 p 是一个指向有一个整型参数且返回类型为整型的函数的指针。
* **int \*(\*p(int))[3];** -- 可以先跳过, 不看这个类型, 过于复杂从 p 开始,先与 () 结合, 说明 p 是一个函数, 然后进入 () 里面, 与 int 结合, 说明函数有一个整型变量参数, 然后再与外面的 \* 结合, 说明函数返回的是一个指针, 然后到最外面一层, 先与[]结合, 说明返回的指针指向的是一个数组, 然后再与 \* 结合, 说明数组里的元素是指针, 然后再与 int 结合, 说明指针指向的内容是整型数据。所以 p 是一个参数为一个整数据且返回一个指向由整型指针变量组成的数组的指针变量的函数。

一个指针 ptrold 加(减)一个整数 n 后，结果是一个新的指针 ptrnew，ptrnew 的类型和 ptrold 的类型相同，ptrnew 所指向的类型和 ptrold 所指向的类型也相同。ptrnew 的值将比 ptrold 的值增加(减少)了 n 乘 sizeof(ptrold 所指向的类型)个字节。就是说，ptrnew 所指向的内存区将比ptrold 所指向的内存区向高(低)地址方向移动了 n 乘 sizeof(ptrold 所指向的类型)个字节。指针和指针进行加减：两个指针不能进行加法运算，这是非法操作，因为进行加法后，得到的结果指向一个不知所向的地方，而且毫无意义。两个指针可以进行减法操作，但必须类型相同，一般用在数组方面，不多说了。

# C 字符串

首先，**strlen** 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。大部分编译程序在编译的时候就把 **sizeof** 计算过了，而 **strlen** 的结果要在运行的时候才能计算出来。

对于以下语句：

char \*str1 = "asdfgh";

char str2[] = "asdfgh";

char str3[8] = {'a', 's', 'd'};

char str4[] = "as\0df";

执行结果是：

sizeof(str1) = 4; strlen(str1) = 6;

sizeof(str2) = 7; strlen(str2) = 6;

sizeof(str3) = 8; strlen(str3) = 3;

sizeof(str4) = 6; strlen(str4) = 2;

str1是字符指针变量，sizeof 获得的是该指针所占的地址空间，32 位操作系统对应 4 字节，所以结果是 4；strlen 返回的是该字符串的长度，遇到 **\0** 结束， **\0** 本身不计算在内，故结果是 6。

str2 是字符数组，大小由字符串常量 "asdfgh" 确定，sizeof 获得该数组所占内存空间大小，包括字符串结尾的 **\0** ，所以结果为 7；strlen 同理返回 6。

str3 也是字符数组，但大小确定为 8，故 sizeof 得到的结果是 8；strlen 统计 **\0** 之前所有字符的个数，即为 3；

str4 是常量字符数组，sizeof 得到字符总数即 6；strlen 计算至 **\0** 结束，因此返回 2；

总结一句就是 sizeof 计算的是变量的大小，而 strlen 计算的是字符串的长度，前者不受字符 **\0** 影响，后者以 **\0** 作为长度判定依据

# C 结构体

1. 结构体变量的首地址能够被其最宽基本类型成员的大小所整除。
2. 结构体每个成员相对于结构体首地址的偏移量(offset)都是成员大小的整数倍，如有需要编译器会在成员之间加上填充字节(internal adding)。即结构体成员的末地址减去结构体首地址(第一个结构体成员的首地址)得到的偏移量都要是对应成员大小的整数倍。
3. 结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要编译器会在成员末尾加上填充字节。

# C typedef

**#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。

# C 输入 & 输出

# C 文件读写

|  |  |
| --- | --- |
| **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"

**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。

# C 预处理器

**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

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。

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;

}

square 等价于 （5+4）\*（5+4）=81

square\_1 等价于 5+4\*5+4=29

# C 头文件

# C 强制类型转换

# C 可变参数

# C 内存管理

对于 void 指针，GNU 认为 **void \*** 和 **char \*** 一样，所以以下写法是正确的:

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

但按照 ANSI(American National Standards Institute) 标准，需要对 void 指针进行强制转换，如下:

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

同时，按照 ANSI(American National Standards Institute) 标准，不能对 void 指针进行算法操作:

void \* pvoid;

pvoid++; //ANSI：错误

pvoid += 1; //ANSI：错误

// ANSI标准之所以这样认定，是因为它坚持：进行算法操作的指针必须是确定知道其指向数据类型大小的。

int \*pint;

pint++; //ANSI：正确

对于我们手动分配的内存，在 C 语言中是不用强制转换类型的。

description = malloc( 200 \* sizeof(char) ); // C 语言正确。

description = malloc( 200 \* sizeof(char) ); // C++ 错误

但是 C++ 是强制要求的，不然会报错。

# C 命令行参数

main 的两个参数的参数名如下:

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

并不一定这样写，只是约定俗成罢了。但是亦可以写成下面这样:

int main( int test\_argc, char \*test\_argv[] )

任意你喜欢的名字。

但是大部分人还是写成开头那样的，如下：

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