12. 语句和语句分类

C语言的代码是由一条一条的**语句**构成的，C语言中的语句可为以下五类：

- 空语句
- 表达式语句
- 函数调用语句
- 复合语句
- 控制语句

空语句是最简单的，一个分号就是一条语句，是空语句。

```c
#include <stdio.h>
int main()
{
    ;//空语句
    return 0;
}
```

空语句，一般出现的地方是：这里需要一条语句，但是这个语句不需要做任何事，就可以写一个空语句。

12.2 表达式语句

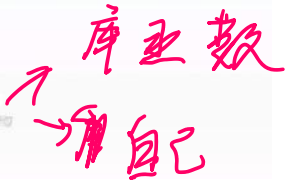
表达式语句就是在表达式的后边加上分号。如下所示：

```c
#include <stdio.h>

int main()
{
    int a = 20;
    int b = 0;
    b = a + 5; //表达式语句
    return 0;
}
```

12.3 函数调用语句

函数调用的时候，也会加上分号，就是函数调用语句。

```c
#include <stdio.h>

int Add(int x, int y)
{
    return x+y;
}

int main()
{
    printf("hehe\n");//函数调用语句
    int ret = Add(2, 3);//函数调用语句
    return 0;
}
```



12.4 复合语句

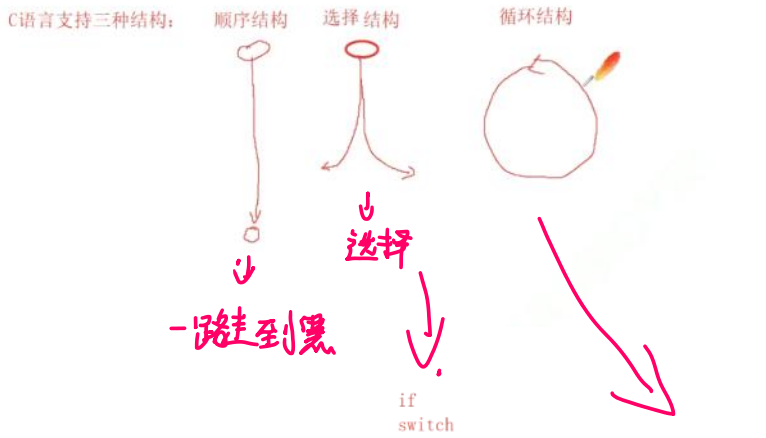
复合语句其实就是前面讲过的**代码块**，成对括号中的代码就构成一个代码块，也被称为复合语句。

```c
#include <stdio.h>

void print(int arr[], int sz) //函数的大括号中的代码也构成复合语句
{
    int i = 0;
```

```c
#include <stdio.h>

void print(int arr[], int sz) //函数的大括号中的代码也构成复合语句
{
    int i = 0;
    for(i=0; i<sz; i++)
    {
        printf("%d ", arr[i]);
    }
}

int main()
{
    int i = 0;
    int arr[10] = {0};
    for(i=0; i<10; i++) //for循环的循环体的大括号中的就是复合语句
    {
        arr[i] = 10-i;
        printf("%d\n", arr[i]);
    }
    return 0;
}
```

复合语句

C语言是一种结构化的程序设计语言



1111

12.5 控制语句

控制语句用于控制程序的执行流程，以实现程序的各种结构方式（C语言支持三种结构：顺序结构、选择结构、循环结构），它们由特定的语句定义符组成，C语言有**九种控制语句**。

可分成以下三类：

1. **条件判断语句也叫分支语句**：if语句、switch语句；
2. **循环执行语句**：do while语句、while语句、for语句；
3. **转向语句**：break语句、goto语句、continue语句、return语句。

后期会给大家——介绍控制语句。

后期会给大家——介绍控制语句。

编译时替换为空格

13. 注释是什么？为什么写注释？

注释是对代码的说明，编译器会忽略注释，也就是说，注释对实际代码没有影响。

注释是给程序员自己，或者其他程序员看的。

好的注释可以帮我们更好的理解代码，但是也不要过度注释，不要写没必要的注释。

当然不写注释可能会让后期阅读代码的人抓狂。

写注释一定程度上反应了程序作者的素质，建议大家写必要的注释，在未来找工作的时候，写代码时留下必要的注释也会给面试官留下更好的印象。

13.1 注释的2种形式

C 语言的注释有两种表示方法。

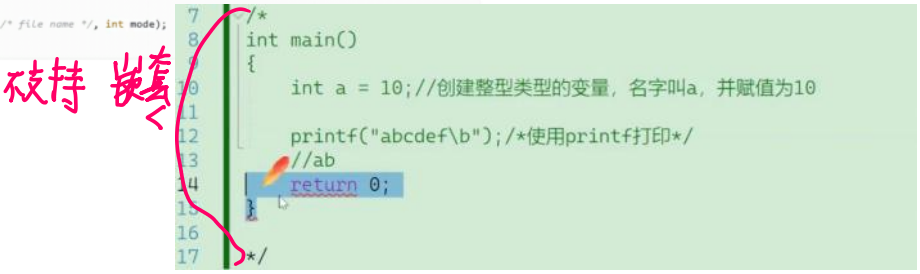
13.1.1 /**/ 的形式

第一种方法是将注释放在 /*...*/ 之间，内部可以分行。

```
/* 注释 */

/*
  这是一行注释
*/
```

这种注释可以插在行内。

```
int fopen(char* s /* file name */, int mode);
```



13.1.2 // 的形式

第二种写法是将注释放在双斜杠 // 后面，从双斜杠到行尾都属于注释。这种注释只能是单行，可以放在行首，也可以放在一行语句的结尾。这是 C99 标准新增的语法。

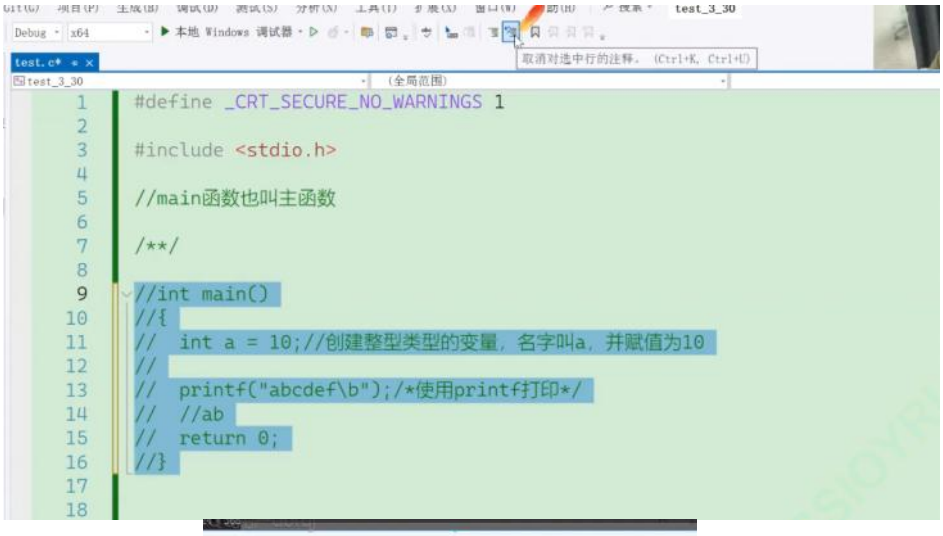
```
// 这是一行注释

int x = 1; // 这也是注释
```

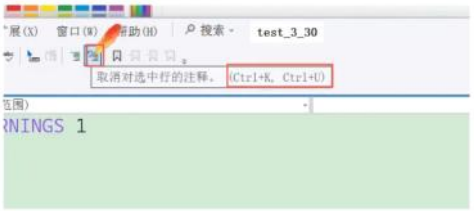
不管是哪一种注释，都不能放在双引号里面。

双引号里面的注释符号，会成为字符串的一部分，解释为普通符号，失去注释作用。

```
printf("// hello /* world */ ");
```

上面示例中，双引号里面的注释符号，都会被视为普通字符，没有注释作用。




展(X)
窗口(W)
帮助(H)
搜索
test_3_30
取消对选中行的注释。 (Ctrl+K, Ctrl+U)
范围)
RNINGS 1

字符类型 `char` 也可以设置 `signed` 和 `unsigned`。

```
signed char c; // 范围为 -128 到 127
unsigned char c; // 范围为 0 到 255
```

注意，C 语言规定 `char` 类型默认是否带有正负号，由当前系统决定。

这就是说，`char` 不等同于 `signed char`，它有可能是 `signed char`，也有可能是 `unsigned char`。

这一点与 `int` 不同，`int` 就是等同于 `signed int`。

3. 数据类型的取值范围

上述的数据类型很多，尤其数整型类型就有short、int、long、long long 四种，为什么呢?

其实每一种数据类型有自己的取值范围，也就是存储的数值的最大值和最小值的区间，有了丰富的类型，我们就可以在适当的场景下去选择适合的类型。如果要查看当前系统上不同数据类型的极限值：

`limits.h` 文件中说明了整型类型的取值范围。

`float.h` 这个头文件中说明浮点型类型的取值范围。

为了代码的可移植性，需要知道某种整数类型的极限值时，应该尽量使用这些常量。

- `SCHAR_MIN`，`SCHAR_MAX`：signed char 的最小值和最大值。
- `SHRT_MIN`，`SHRT_MAX`：short 的最小值和最大值。
- `INT_MIN`，`INT_MAX`：int 的最小值和最大值。
- `LONG_MIN`，`LONG_MAX`：long 的最小值和最大值。
- `LLONG_MIN`，`LLONG_MAX`：long long 的最小值和最大值。
- `UCHAR_MAX`：unsigned char 的最大值。
- `USHRT_MAX`：unsigned short 的最大值。
- `UINT_MAX`：unsigned int 的最大值。
- `ULONG_MAX`：unsigned long 的最大值。
- `ULLONG_MAX`：unsigned long long 的最大值。

4. 变量

4.1 变量的创建

了解清楚了类型，我们使用类型做什么呢？类型是用来**创建变量**的。

什么是变量呢？C语言中把经常变化的值称为**变量**，不变的值称为**常量**。

变量创建的语法形式是这样的：

```
data_type name;
    |       |
    |       |
数据类型   变量名
```

```c
int age; //整型变量
char ch; //字符变量
double weight; //浮点型变量
```

变量在创建的时候就给一个初始值，就叫初始化。

```c
int age = 18;
char ch = 'w';
double weight = 48.0;
unsigned int height = 100;
```

4.2 变量的分类

- **全局变量：**在大括号外部定义的变量就是全局变量

全局变量的使用范围更广，整个工程中想使用，都是有办法使用的。

- **局部变量：**在大括号内部定义的变量就是局部变量

局部变量的使用范围是比较局限，只能在自己所在的局部范围内使用的。

```c
#include <stdio.h>

int global = 2023;//全局变量

int main()
{
    int local = 2018;//局部变量
    printf("%d\n", local);
    printf("%d\n", global);
```

```
    return 0;
}
```


如果局部和全局变量，名字相同呢？

```c
#include <stdio.h>

int n = 1000;
int main()
{
    int n = 10;
    printf("%d\n" n);//打印的结果是多少呢?
    return 0;
}
```

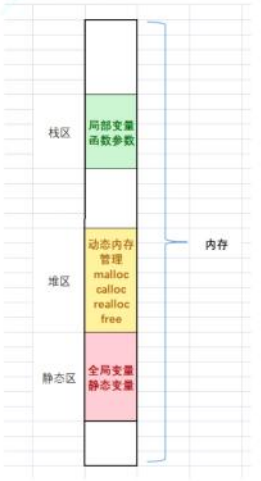
其实当局部变量和全局变量同名的时候，局部变量优先使用。

全局变量和局部变量在内存中存储在哪里呢?

一般我们在学习C/C++语言的时候，我们会关注内存中的三个区域：**栈区、堆区、静态区**。

1. 局部变量是放在内存的**栈区**
2. 全局变量是放在内存的**静态区**
3. 堆区是用来动态内存管理的（后期会介绍）

其实内存区域的划分会更加细致，以后在操作系统的相关知识的时候会介绍。



5. 算术操作符：+、-、*、/、%

在写代码时候，一定会涉及到计算。

C语言中为了方便运算，提供了一系列操作符，其中有一组操作符叫：**算术操作符**。分别是：`+ - * / %`，这些操作符都是**双目操作符**。

注：操作符也被叫做：**运算符**，是不同的翻译，意思是一样的。

5.1 `+` 和 `-`

`+` 和 `-` 用来完成加法和减法。

`+` 和 `-` 都是有2个操作数的，位于操作符两端的就是它们的操作数，这种操作符也叫**双目操作符**。

```c
#include <stdio.h>
int main()
{
    int x = 4 + 22;
    int y = 61 - 23;
    printf("%d\n", x);
    printf("%d\n", y);
    return 0;
}
```

5.2 `*`

运算符 `*` 用来完成乘法。

```c
#include <stdio.h>
int main()
{
    int num = 5;
    printf("%d\n", num * num); // 输出 25
    return 0;
}
```

5.3 `/`

运算符 `/` 用来完成除法。

除号的两端如果是整数，执行的是整数除法，得到的结果也是整数。

```c
#include <stdio.h>
int main()
{
    float x = 6 / 4;
    int y = 6 / 4;
    printf("%f\n", x); // 输出 1.000000
    printf("%d\n", y); // 输出 1
    return 0;
}
```

上面示例中，尽管变量 `x` 的类型是 `float` （浮点数），但是 `6 / 4` 得到的结果是 `1.0` ，而不是 `1.5` 。原因就在于 C 语言里面的整数除法是整除，只会返回整数部分，丢弃小数部分。

如果希望得到浮点数的结果，**两个运算数必须至少有一个浮点数**，这时 C 语言就会进行浮点数除法。

```c
#include <stdio.h>
int main()
{
    float x = 6.0 / 4; // 或者写成 6 / 4.0
    printf("%f\n", x); // 输出 1.500000
    return 0;
}
```

上面示例中， `6.0 / 4` 表示进行浮点数除法，得到的结果就是 `1.5` 。

再看一个例子：

```c
#include <stdio.h>
int main()
{
    int score = 5;
    score = (score / 20) * 100;
    return 0;
}
```

上面的代码，你可能觉得经过运算， `score` 会等于 `25` ，但是实际上 `score` 等于 `0` 。这是因为 `score / 20` 是整除，会得到一个整数值 `0` ，所以乘以 `100` 后得到的也是 `0` 。

为了得到预想的结果，可以将除数 `20` 改成 `20.0` ，让整除变成浮点数除法。

```c
#include <stdio.h>
int main()
{
    int score = 5;
    score = (score / 20.0) * 100;
    return 0;
}
```


5.4 %

运算符 % 表示求模(余)运算，即返回两个整数相除的余值。这个运算符只能用于整数，不能用于浮点数。

```c
#include <stdio.h>
int main()
{
    int x = 6 % 4; // 2
    return 0;
}
```

负数求模的规则是，结果的正负号由第一个运算数的正负号决定。

```c
#include <stdio.h>
int main()
{
    printf("%d\n", 11 % -5); // 1
    printf("%d\n",-11 % -5); // -1
    printf("%d\n",-11 % 5); // -1
    return 0;
}
```

上面示例中，第一个运算数的正负号（ 11 或 -11 ）决定了结果的正负号。

6. 赋值操作符：=和复合赋值

在变量创建的时候给一个初始值叫初始化，在变量创建好后，再给一个值，这叫赋值。

```c
int a = 100;//初始化
```

```
2 a = 200;//赋值，这里使用的就是赋值操作符
```


赋值操作符 `=` 是一个随时可以给变量赋值的操作符。

6.1 连续赋值

赋值操作符也可以连续赋值，如：

```
1 int a = 3;
2 int b = 5;
3 int c = 0;
4 c = b = a+3;//连续赋值，从右向左依次赋值的。
```

C语言虽然支持这种连续赋值，但是写出的代码不容易理解，建议还是拆开来写，这样方便观察代码的执行细节。

```
1 int a = 3;
2 int b = 5;
3 int c = 0;
4 b = a+3;
5 c = b;
```

这样写，在调试的是，每一次赋值的细节都是可以很方便的观察的。

6.2 复合赋值符

在写代码时，我们经常可能对一个数进行自增、自减的操作，如下代码：

```
1 int a = 10;
2 a = a+3;
3 a = a-2;
```

这样代码C语言给提供了更加方便的写法：

```
1 int a = 10;
2 a += 3;
3 a -= 2;
```

C语言中提供了复合赋值符，方便我们编写代码，这些赋值符有：

```
+=      -=
*=      /=      %=
//下面的操作符后期讲解
>>=     <<=
&=      |=      ^=
```

7. 单目操作符：++、--、+、-

前面介绍的操作符都是双目操作符，有2个操作数的。C语言中还有一些操作符只有一个操作数，被称为单目操作符。 `++`、`--`、`+`(正)、`-`(负) 就是单目操作符的。

7.1 ++和--

++是一种自增的操作符，又分为前置++和后置++，--是一种自减的操作符，也分为前置--和后置--.

7.1.1 前置++

```
int a = 10;
int b = ++a;//++的操作数是a，是放在a的前面的，就是前置++
printf("a=%d b=%d\n",a , b);
```

计算口诀：先+1，后使用；

a原来是10，先+1，后a变成了11，再使用就是赋值给b，b得到的也是11，所以计算技术后，a和b都是11，相当于这样的代码：

```
int a = 10;
a = a+1;
b = a;
printf("a=%d b=%d\n",a , b);
```

7.1.2 后置++

```
int a = 10;
int b = a++;//++的操作数是a，是放在a的后面的，就是后置++
printf("a=%d b=%d\n",a , b);
```

计算口诀：先使用，后+1

比特就业课主页：https://m.cctalk.com/inst/s9yewhfr

a原来是10，先使用，就是先赋值给b，b得到了10，然后再+1，然后a变成了11，所以直接结束后a是11，b是10，相当于这样的代码：

```
int a = 10;
int b = a;
a = a+1;
printf("a=%d b=%d\n",a , b);
```

7.1.3 前置--

如果你听懂了前置++，那前置--是同理的，只是把加1，换成了减1；

计算口诀：先-1，后使用

```
int a = 10;
int b = --a;//--的操作数是a，是放在a的前面的，就是前置--
printf("a=%d b=%d\n",a , b);//输出的结果是：9 9
```

7.1.4 后置--

同理后置--类似于后置++，只是把加一换成了减一

计算口诀：先使用，后-1

```
int a = 10;
int b = a--;//--的操作数是a，是放在a的后面的，就是后置--
printf("a=%d b=%d\n",a , b);//输出的结果是：9 10
```

7.2 + 和 -

这里的+是正号，-是负号，都是**单目操作符**。

运算符 `+` 对正负值没有影响，是一个完全可以省略的运算符，但是写了也不会报错。

```
int a = +10;  等价于  int a = 10;
```

运算符 `-` 用来改变一个值的正负号，负数的前面加上 `-` 就会得到正数，正数的前面加上 `-` 会得到负数。

```
int a = 10;
```

```
int b = -a;
int c = -10;
printf("b=%d c=%d\n", b, c);//这里的b和c都是-10

int a = -10;
int b = -a;
printf("b=%d\n", b);        //这里的b是10
```


8. 强制类型转换

在操作符中还有一种特殊的操作符是强制类型转换，语法形式很简单，形式如下：

```
(类型)
```

请看代码：

```
int a = 3.14;
//a的是int类型，3.14是double类型，两边的类型不一致，编译器会报警告
```

为了消除这个警告，我们可以使用强制类型转换：

```
int a = (int)3.14;//意思是将3.14强制类型转换为int类型，这种强制类型转换只取整数部分
```

俗话说，强扭的瓜不甜，我们使用强制类型转换都是万不得已的时候使用，如果不需要强制类型转化就能实现代码，这样自然更好的。

9. scanf 和 printf 介绍

9.1 printf

9.1.1 基本用法

`printf()` 的作用是将参数文本输出到屏幕。它名字里面的 `f` 代表 `format`（格式化），表示可以定制输出文本的格式。

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

```
{
    printf("Hello World");
    return 0;
}
```

上面命令会在屏幕上输出一行文字“Hello World”。

`printf()` 不会在行尾自动添加换行符，运行结束后，光标就停留在输出结束的地方，不会自动换行。

为了让光标移到下一行的开头，可以在输出文本的结尾，添加一个换行符 `\n`。

```
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

如果文本内部有换行，也是通过插入换行符来实现，如下方代码：

```
#include <stdio.h>
int main(void)
{
    printf("Hello\nWorld\n");

    printf("Hello\n");
    printf("World\n");
    return 0;
}
```

`printf()` 是在标准库的头文件 `stdio.h` 定义的。使用这个函数之前，必须在源码文件头部引入这个头文件。

9.1.2 占位符

`printf()` 可以在输出文本中指定占位符。

所谓 **“占位符”**，就是这个位置可以用其他值代入。

```
// 输出 There are 3 apples
```

```c
#include <stdio.h>
int main()
{
    printf("There are %d apples\n", 3);
    return 0;
}
```

上面示例中，`There are %d apples\n` 是输出文本，里面的 `%d` 就是占位符，表示这个位置要用其他值来替换。**占位符的第一个字符一律为百分号 `%`**，第二个字符表示占位符的类型，**`%d` 表示这里代入的值必须是一个整数**。

`printf()` 的第二个参数就是替换占位符的值，上面的例子是整数 `3` 替换 `%d`。执行后的输出结果就是 `There are 3 apples`。

常用的占位符除了 `%d`，还有 `%s` 表示代入的是字符串。

```c
#include <stdio.h>
int main()
{
    printf("%s will come tonight\n", "zhangsan");
    return 0;
}
```

上面示例中，**`%s` 表示代入的是一个字符串**，所以 `printf()` 的第二个参数就必须是字符串，这个例子是 `zhangsan`。执行后的输出就是 `zhangsan will come tonight`。

输出文本里面可以使用**多个占位符**。

```c
#include <stdio.h>
int main()
{
    printf("%s says it is %d o'clock\n", "lisi", 21);
    return 0;
}
```

上面示例中，输出文本 `%s says it is %d o'clock` 有两个占位符，第一个是字符串占位符 `%s`，第二个是整数占位符 `%d`，分别对应 `printf()` 的第二个参数（`lisi`）和第三个参数（`21`）。执行后的输出就是 `lisi says it is 21 o'clock`。

比特就业课主页：https://m.cctalk.com/inst/s9yewhfr

`printf()` 参数与占位符是一一对应关系，如果有 `n` 个占位符，`printf()` 的参数就应该有 `n + 1` 个。如果参数个数少于对应的占位符，`printf()` 可能会输出内存中的任意值。

9.1.3 占位符列举

`printf()` 的占位符有许多种类，与 C 语言的数据类型相对应。下面按照字母顺序，列出常用的占位符，方便查找，具体含义在后面章节介绍。

- `%a`：十六进制浮点数，字母输出为小写。
- `%A`：十六进制浮点数，字母输出为大写。
- **`%c`：字符。**
- **`%d`：十进制整数。// int**
- `%e`：使用科学计数法的浮点数，指数部分的 `e` 为小写。
- `%E`：使用科学计数法的浮点数，指数部分的 `E` 为大写。
- `%i`：整数，基本等同于 `%d`。
- **`%f`：小数（包含 `float` 类型和 `double` 类型）。//float %f double - %lf**
- `%g`：6个有效数字的浮点数。整数部分一旦超过6位，就会自动转为科学计数法，指数部分的 `e` 为小写。
- `%G`：等同于 `%g`，唯一的区别是指数部分的 `E` 为大写。
- **`%hd`：十进制 short int 类型。**
- `%ho`：八进制 short int 类型。
- `%hx`：十六进制 short int 类型。
- **`%hu`：unsigned short int 类型。**
- **`%ld`：十进制 long int 类型。**
- `%lo`：八进制 long int 类型。
- `%lx`：十六进制 long int 类型。
- **`%lu`：unsigned long int 类型。**
- `%lld`：十进制 long long int 类型。
- `%llo`：八进制 long long int 类型。
- `%llx`：十六进制 long long int 类型。
- `%llu`：unsigned long long int 类型。
- `%Le`：科学计数法表示的 long double 类型浮点数。
- **`%Lf`：long double 类型浮点数。**

- `%n`：已输出的字符串数量。该占位符本身不输出，只将值存储在指定变量之中。
- `%o`：八进制整数。
- **`%p`：指针（用来打印地址）。**
- **`%s`：字符串。**
- **`%u`：无符号整数（unsigned int）。**
- **`%x`：十六进制整数。**
- **`%zd`：`size_t`类型。**
- `%%`：输出一个百分号。

9.1.4 输出格式

`printf()`可以定制占位符的输出格式。

9.1.4.1 限定宽度

`printf()`允许限定占位符的**最小宽度**。

```
#include <stdio.h>
int main()
{
    printf("%5d\n", 123); // 输出为 "  123"
    return 0;
}
```

上面示例中，`%5d`表示这个占位符的宽度至少为5位。如果不满5位，对应的值的前面会添加空格。

输出的值**默认是右对齐**，即输出内容前面会有空格；如果希望改成左对齐，在输出内容后面添加空格，可以在占位符的`%`的后面插入一个`-`号。

```
#include <stdio.h>
int main()
{
    printf("%-5d\n", 123); // 输出为 "123  "
    return 0;
}
```

上面示例中，输出内容`123`的后面添加了空格。

对于小数，这个限定符会限制所有数字的最小显示宽度。

```c
// 输出 "  123.450000"
#include <stdio.h>
int main()
{
    printf("%12f\n", 123.45);
    return 0;
}
```

上面示例中，`%12f` 表示输出的浮点数最少要占据12位。由于小数的默认显示精度是小数点后6位，所以 `123.45` 输出结果的头部会添加2个空格。

9.1.4.2 总是显示正负号

默认情况下，`printf()` 不对正数显示 `+` 号，只对负数显示 `-` 号。如果想让正数也输出 `+` 号，可以在占位符的 `%` 后面加一个 `+` 。

```c
#include <stdio.h>
int main()
{
    printf("%+d\n", 12); // 输出 +12
    printf("%+d\n", -12); // 输出 -12
    return 0;
}
```

上面示例中，`%+d` 可以确保输出的数值，总是带有正负号。

9.1.4.3 限定小数位数

输出小数时，有时希望限定小数的位数。举例来说，希望小数点后面只保留两位，占位符可以写成 `%.2f` 。

```c
// 输出 Number is 0.50
#include <stdio.h>
int main()
{
    printf("Number is %.2f\n", 0.5);
    return 0;
}
```

上面示例中，如果希望小数点后面输出3位（`0.500`），占位符就要写成 `%.3f` 。

这种写法可以与限定宽度占位符，结合使用。

```
// 输出为 "  0.50"
#include <stdio.h>
int main()
{
    printf("%6.2f\n", 0.5);
    return 0;
}
```

上面示例中，`%6.2f` 表示输出字符串最小宽度为6，小数位数为2。所以，输出字符串的头部有两个空格。

最小宽度和小数位数这两个限定值，都可以用 `*` 代替，通过 `printf()` 的参数传入。

```
#include <stdio.h>
int main()
{
    printf("%*.*f\n", 6, 2, 0.5);
    return 0;
}
// 等同于printf("%6.2f\n", 0.5);
```

上面示例中，`%*.*f` 的两个星号通过 `printf()` 的两个参数 `6` 和 `2` 传入。

9.1.4.4 输出部分字符串

`%s` 占位符用来输出字符串，默认是全部输出。如果只想输出开头的部分，可以用 `%.[m]s` 指定输出的长度，其中 `[m]` 代表一个数字，表示所要输出的长度。

```
// 输出 hello
#include <stdio.h>
int main()
{
    printf("%.5s\n", "hello world");
    return 0;
}
```

比特就业课主页：https://m.cctalk.com/inst/s9yewhfr

上面示例中，占位符 `%.5s` 表示只输出字符串 “hello world” 的前5个字符，即 “hello” 。

9.2 scanf

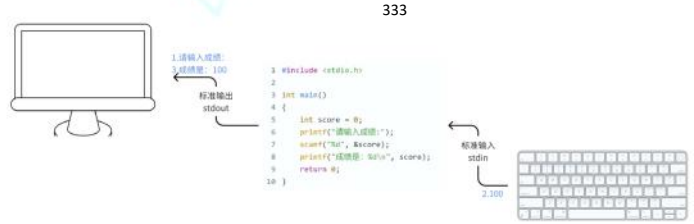
当我们有了变量，我们需要给变量输入值就可以使用 `scanf` 函数，如果需要将变量的值输出在屏幕上的时候可以使用 `prinf` 函数，下面看一个例子：

```c
#include <stdio.h>

int main()
{
    int score = 0;
    printf("请输入成绩:");
    scanf("%d", &score);
    printf("成绩是: %d\n", score);
    return 0;
}
```

运行截图：

```
Microsoft Visual Studio 调试控制台
请输入成绩:100
成绩是：100
```

画图演示：



> 注：标准输入一般指的就是键盘，标准输出一般指的就是屏幕

wwww

那接下来我们介绍一下 `scanf` 函数。

9.2.1 基本用法

`scanf()` 函数用于读取用户的键盘输入。

程序运行到这个语句时，会停下来，等待用户从键盘输入。

用户输入数据、按下回车键后， `scanf()` 就会处理用户的输入，将其存入变量。

它的原型定义在头文件 `stdio.h` 。

`scanf()` 的语法跟 `printf()` 类似。

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

它的第一个参数是一个格式字符串，里面会放置占位符（与 `printf()` 的占位符基本一致），告诉编译器如何解读用户的输入，需要提取的数据是什么类型。

这是因为 C 语言的数据都是有类型的， `scanf()` 必须提前知道用户输入的数据类型，才能处理数据。

它的其余参数就是存放用户输入的变量，格式字符串里面有多少个占位符，就有多少个变量。

上面示例中， `scanf()` 的第一个参数 `%d` ，表示用户输入的应该是一个整数。 `%d` 就是一个占位符， `%` 是占位符的标志， `d` 表示整数。第二个参数 `&i` 表示，将用户从键盘输入的整数存入变量 `i` 。

> 注意：变量前面必须加上 `&` 运算符（指针变量除外），因为 `scanf()` 传递的不是值，而是地址，即将变量 `i` 的地址指向用户输入的值。
>
> 如果这里的变量是指针变量（比如字符串变量），那就不用加 `&` 运算符。

> 注意：变量前面必须加上 `&` 运算符（指针变量除外），因为 `scanf()` 传递的不是值，而是地址，即将变量 `i` 的地址指向用户输入的值。
>
> 如果这里的变量是指针变量（比如字符串变量），那就不用加 `&` 运算符。

下面是一次将键盘输入读入多个变量的例子。

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

上面示例中，格式字符串 `%d%d%f%f` ，表示用户输入的前两个是整数，后两个是浮点数，比如 `1 -20 3.4 -4.0e3` 。这四个值依次放入 `i` 、 `j` 、 `x` 、 `y` 四个变量。

`scanf()` 处理数值占位符时，会**自动过滤空白字符，包括空格、制表符、换行符**等。

所以，用户输入的数据之间，有一个或多个空格不影响 `scanf()` 解读数据。另外，用户使用回车键，将输入分成几行，也不影响解读。


```
1
-20
3.4
-4.0e3
```

上面示例中，用户分成四行输入，得到的结果与一行输入是完全一样的。每次按下回车键以后，`scanf()` 就会开始解读，如果第一行匹配第一个占位符，那么下次按下回车键时，就会从第二个占位符开始解读。

`scanf()` 处理用户输入的原理是，用户的输入先放入缓存，等到按下回车键后，按照占位符对缓存进行解读。

解读用户输入时，会从上一次解读遗留的第一个字符开始，直到读完缓存，或者遇到第一个不符合条件的字符为止。

```
#include <stdio.h>

int main()
{
    int x;
    float y;

    // 用户输入 "    -13.45e12# 0"
    scanf("%d", &x);
    printf("%d\n", x);
    scanf("%f", &y);
    printf("%f\n", y);
    return 0;
}
```

上面示例中， `scanf()` 读取用户输入时， `%d` 占位符会忽略起首的空格，从 `-` 处开始获取数据，读取到 `-13` 停下来，因为后面的 `.` 不属于整数的有效字符。这就是说，占位符 `%d` 会读到 `-13` 。

第二次调用 `scanf()` 时，就会从上一次停止解读的地方，继续往下读取。这一次读取的首字符是 `.` ，由于对应的占位符是 `%f` ，会读取到 `.45e12` ，这是采用科学计数法的浮点数格式。后面的 `#` 不属于浮点数的有效字符，所以会停在这里。

由于 `scanf()` 可以连续处理多个占位符，所以上面的例子也可以写成下面这样。

```
#include <stdio.h>

int main()
```

```c
{
    int x;
    float y;

    // 用户输入 "    -13.45e12# 0"
    scanf("%d%f", &x, &y);
    return 0;
}
```

9.2.2 scanf的返回值

`scanf()` 的返回值是一个整数，表示成功读取的变量个数。

如果没有读取任何项，或者匹配失败，则返回 `0` 。

如果在成功读取任何数据之前，发生了读取错误或者遇到读取到文件结尾，则返回常量 EOF (-1)。

EOF - end of file 文件结束标志

```c
#include <stdio.h>
int main()
{
    int a = 0;
    int b = 0;
    float f = 0.0f;
    int r = scanf("%d %d %f", &a, &b, &f);
    printf("a=%d b=%d f=%f\n", a, b, f);
    printf("r = %d\n", r);
    return 0;
}
```

输入输出测试：

```
Microsoft Visual Studio 调试控制台
1 2 3.14
a=1 b=2 f=3.140000
r = 3
```

如果输入2个数后，按 `ctrl+z` ，提前结束输入：

```
Microsoft Visual Studio 调试控制台
1 2
^Z
^Z
^Z
a=1 b=2 f=0.000000
r = 2
```

在VS环境中按3次 `ctrl+z` ，才结束了输入，我们可以看到r是2，表示正确读取了2个数值。

如果一个数字都不输入，直接按3次 `ctrl+z` ,输出的r是-1，也就是EOF

```
Microsoft Visual Studio 调试控制台
^Z
^Z
^Z
a=0 b=0 f=0.000000
r = -1
```

9.2.3 占位符

`scanf()` 常用的占位符如下，与 `printf()` 的占位符基本一致。

- `%c` ：字符。
- `%d` ：整数。
- `%f` ： `float` 类型浮点数。
- `%lf` ： `double` 类型浮点数。
- `%Lf` ： `long double` 类型浮点数。
- `%s` ：字符串。
- `%[]` ：在方括号中指定一组匹配的字符（比如 `%[0-9]` ），遇到不在集合之中的字符，匹配将会停止。

上面所有占位符之中，**除了 `%c` 以外，都会自动忽略起首的空白字符**。 `%c` 不忽略空白字符，总是返回当前第一个字符，无论该字符是否为空格。

如果要强制跳过字符前的空白字符，可以写成 `scanf(" %c", &ch)` ，即 `%c` 前加上一个空格，表示跳过零个或多个空白字符。

下面要**特别说一下占位符 `%s`** ，它其实不能简单地等同于字符串。它的规则是，从当前第一个非空白字符开始读起，直到遇到空白字符（即空格、换行符、制表符等）为止。

因为 `%s` 不会包含空白字符，所以无法用来读取多个单词，除非多个 `%s` 一起使用。这也意味着， `scanf()` 不适合读取可能包含空格的字符串，比如书名或歌曲名。另外， `scanf()` 遇到 `%s` 占位符，会在字符串变量末尾存储一个空字符 `\0` 。

`scanf()` 将字符串读入字符数组时，不会检测字符串是否超过了数组长度。所以，储存字符串时，很可能会超过数组的边界，导致预想不到的结果。为了防止这种情况，使用 `%s` 占位符时，应该指定读入字符串的最长长度，即写成 `%[m]s` ，其中的 `[m]` 是一个整数，表示读取字符串的最大长度，后面的字符将被丢弃。

```c
#include <stdio.h>

int main()
{
    char name[11];
    scanf("%10s", name);

    return 0;
}
```

上面示例中， `name` 是一个长度为11的字符数组， `scanf()` 的占位符 `%10s` 表示最多读取用户输入的10个字符，后面的字符将被丢弃，这样就不会有数组溢出的风险了。

9.2.4 赋值忽略符

有时，用户的输入可能不符合预定的格式。

```c
#include <stdio.h>

int main()
{
    int year = 0;
    int month = 0;
    int day = 0;
    scanf("%d-%d-%d", &year, &month, &day);
    printf("%d %d %d\n", year, month, day);
    return 0;
}
```

上面示例中，如果用户输入 `2020-01-01` ，就会正确解读出年、月、日。问题是用户可能输入其他格式，比如 `2020/01/01` ，这种情况下， `scanf()` 解析数据就会失败。

为了避免这种情况， `scanf()` 提供了一个**赋值忽略符**（assignment suppression character） `*` 。只要把 `*` 加在任何占位符的百分号后面，该占位符就不会返回值，解析后将被丢弃。

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

```c
int main()
{
    int year = 0;
    int month = 0;
    int day = 0;
    scanf("%d%*c%d%*c%d", &year, &month, &day);
    return 0;
}
```

上面示例中，`%*c` 就是在占位符的百分号后面，加入了赋值忽略符 `*`，表示这个占位符没有对应的变量，解读后不必返回。

完

比特就业课主页：https://m.cctalk.com/inst/s9yewhfr