

# C 语言教程讲义

Potato

2024 年 7 月 22 日

第零章至第五章

本文件编译于 2024 年 7 月 22 日

## 0 计算机基础知识与计算机语言概述

### 1 标准输出和变量

#### 1.1 printf 函数

打开 visual studio，在“源文件”上右键，点击“添加”->“新建项”，创建一个叫 main.c 的文件，在其中输入

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello World!");
5
6      return 0;
7  }
```

点击“本地 Windows 调试器”，将会出现类似于这样的界面



恭喜你写出了你的第一段代码。计算机领域有一个传统：刚入门编程的人的第一段代码一定要是 Hello World，这是我们对无垠的计算机世界打的招呼。这个黑乎乎的界面叫做终端（这里不做关于终端、shell 等的详细区分），我们以后会常常和它打交道。

我们发现，我们写在 printf 的引号内的内容会被打印出来（顾名思义，printf 就是 print function 嘛<sup>1</sup>），因此我们可以多写几个 printf（不要忘了每行最后的英文分号），如

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello World!");
5      printf("Potato");
6
7      return 0;
8  }
```

运行代码，我们观察到 Hello World 和 Potato 输出在了同一行。我们想要的是这二者各自占一行，那我们就需要在 Hello World!后加一个换行符，也即 \n，代码就变成了

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello World!\n");
5      printf("Potato\n");
6
7      return 0;
8  }
```

---

<sup>1</sup>事实并非如此，实际上此处的 f 指代 format

这次的结果就是我们想要的了。为什么 Potato 也要加换行符？这是因为也许我们在其后还会添加新的 printf，要是到时候忘了给 Potato 加换行符，不就没有实现我们想要的效果吗？为了避免以后出现不必要的问题，我们推荐每个 printf 后都加换行符。

由上面的代码，我们总结出了这些要点：首先，我们要想输出信息，就要将其写在 printf 的引号内；其次，要想换行，就需要在输出的文本最后加一个换行符 \n；另外我们还注意到，我们先输出了 Hello World，再输出了 Potato，也就是说，程序是按照从上到下的顺序执行的。

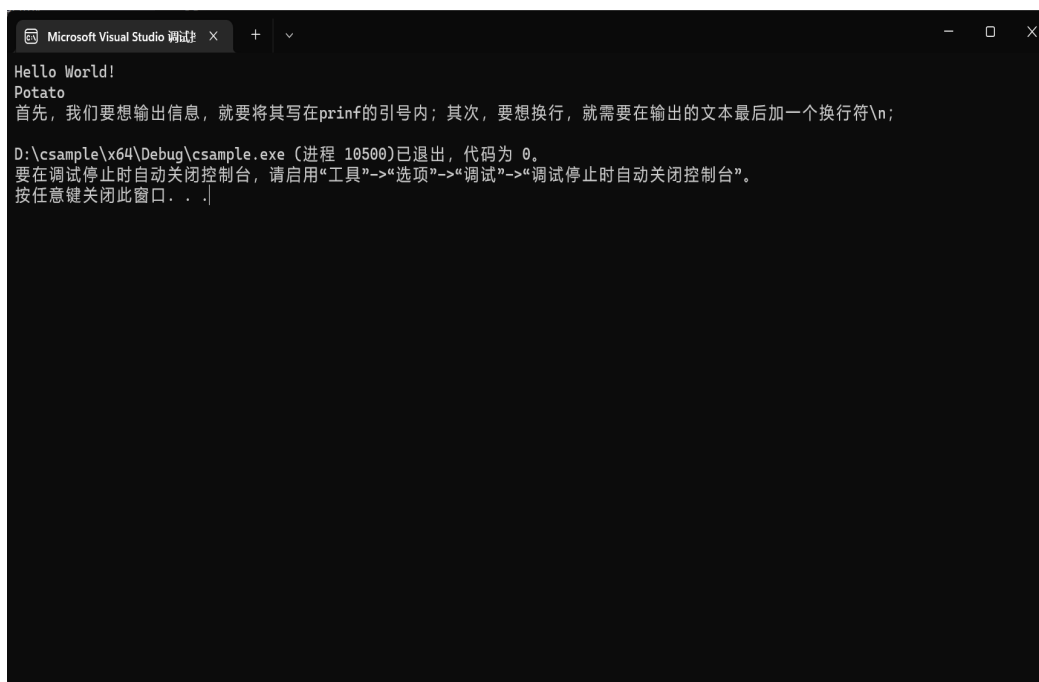
这是很重要的结论，所以我们使用 printf 输出一遍加强印象吧。于是我们写下了：

```
1 printf("首先,我们要想输出信息,就要将其写在printf的引号内;其次,要想换行,就需要在输出的文本最后加一个换行符\n;\n");
```

我们注意到有个问题，就是第一个换行符是我们想要按照文本形式输出的，但是如果按照上面的写法，它会被用于换行而不是直接输出。为了解决这个问题，我们把代码改成这样即可：

```
1 printf("首先,我们要想输出信息,就要将其写在printf的引号内;其次,要想换行,就需要在输出的文本最后加一个换行符\\n;\n");
```

运行结果是：



```
Microsoft Visual Studio 调试
Hello World!
Potato
首先, 我们要想输出信息, 就要将其写在printf的引号内; 其次, 要想换行, 就需要在输出的文本最后加一个换行符\n;
D:\csample\x64\Debug\csample.exe (进程 10500)已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。 . . |
```

为什么会这样呢？实际上，我们输入的 \n 和 \\ 都叫做转义符号，用来实现一些特殊的输出效果。它的语法就是反斜杠 (\) 加一个字母。我们最常用的转义符号就是换行符 \n，实际上还有 \t、\a 等等。因为反斜杠会和下一个字母结合变成转义符号，所以要想输出反斜杠，就需要按照上面的例子那样使用两个反斜杠，这个转义符号的结果就是输出一个反斜杠。

## 1.2 变量

变量 (variable) 顾名思义, 就是可以变的量。一个变量肯定要有名字, 然后要有一个值。这就像是我们的考试成绩, 有一个叫做“数学成绩”的变量, 这次可以是 120, 下次可以是 130。在 c 语言中, 我们这样声明变量: *type name*, 啥意思呢, 比如我们要声明数学成绩 *math\_grade*, 那么就可以这样写:

```
1  int math_grade;
```

这里的 *int* 是整数的意思, 还有其它类型 (如小数) 我们之后会讲到。所以这行代码的意思就是, 我们声明了一个叫做 *math\_grade* 的变量, 它是一个整数。接下来就要给它赋值了, 在大部分计算机语言中, 赋值号都是=, 比如数学成绩是 120 可以这样写:

```
1  math_grade = 120;
```

这段代码的意思就是给 *math\_grade* 赋予 120 这个值。也即, 我们将赋值号右边的值给了左边的变量。我们强调是将右边的值赋给左边, 说明了这不是数学中的等号, 它不意味着左右两边相等, 也不能交换左右两边的位置。

现在我们的 *main.c* 应该是这样的:

```
1  #include <stdio.h>
2
3  int main() {
4      int math_grade;
5      math_grade = 120;
6
7      return 0;
8  }
```

第四行是声明变量, 第五行是给变量赋值。如果我们在声明变量时就知道它要赋什么值, 那么就可以把声明和赋值<sup>2</sup>写在一起:

```
1  int math_grade = 120;
```

如果要同时声明多个类型一样的变量, 那么可以写在一行, 比如同时声明数学、语文和英语成绩:

```
1  int math_grade, chinese_grade, english_grade;
```

不过笔者推荐每个变量单独成行, 因为在最开始写代码时, 我们可能由于考虑不周会选错变量类型 (比如英语成绩会出现 0.5 分的情况, 那么选择整数就不合适了), 如果写在同一行会增大修改的难度, 因此推荐每个变量分开声明:

```
1  int math_grade;
2  int chinese_grade;
3  int english_grade;
```

变量的声明只能有一次, 但是赋值是可以有多次的, 新的赋值会覆盖掉原先的值。比如:

---

<sup>2</sup>按照 c 语言标准, c 语言不存在“赋值语句”, 本书中“赋值语句”均指代赋值表达式语句

```
1  int math_grade = 120;
2  math_grade = 130;
3  math_grade = 140;
```

最后，数学成绩的值就是 140。但是像这样再次声明变量是不允许的：

```
1  int math_grade = 120;
2  math_grade = 130;
3  int math_grade = 140;
```

第三行再次写了 `int math_grade`，再次声明了这个变量，这样的不行的。

那么变量的命名遵循怎样的原则呢？我们这里的命名是 `math_grade`，其实也可以命名成 `MathGrade`，也可以是 `mathGrade`，可以是拼音 `shuxuechengji`，甚至可以是汉语的数学成绩，如果你比较懒，直接使用一个字母来命名变量也可以。c 语言要求，变量名开头不是数字、不包含空格、标点和%、@等符号，不是 c 语言关键字（后面会介绍这个概念）即可。那么我们为什么使用了 `math_grade` 呢？

首先我们说明为什么不用汉语或者汉语拼音。一方面，由于编码不同，在你的电脑上正常显示的中文可能在别人电脑上就变成了乱码（这就是第零章要求大家切换 UTF8 编码的原因），这样严重影响了正常的开发和交流工作。另一方面，大家有没有发现，当你在声明好变量后，你敲下“m”时就会出现一个框，可以直接补全 `math_grade`，这是 IDE 的自动补全功能，如果使用中文就无法自动补全了。zhi yu wei shen mo bu yong pin yin, ge wei jue de zhe yang de ke du xing gao ma?

计算机语言的核心在于其逻辑而不是用了什么自然语言作为支撑，哪怕世界上第一门计算机语言是中国人发明的，计算机语言也会愈发趋向于使用字母来构成，因为这样效率更高。这样大家就可以明白那些被大肆宣扬的“汉语编程语言”的荒谬之处了，不要陷入了虚伪的爱国主义的陷阱。

我们命名的 `math_grade` 有个特点：单词全部小写，单词之间使用下划线\_连接。这种命名方式叫做蛇形命名法 (snake\_case)。至于我们上面提到的 `mathGrade` 也是一种命名法，叫做小驼峰命名法 (camelCase)。命名法和一些其它规则构成了代码风格 (code style)，本讲义遵循 Google 代码规范，也推荐各位使用这套代码规范。各位在阅读此讲义时应该注意示例代码的代码风格，自己写代码时要模仿示例代码的风格。不过，如果各位是参与到他人的项目，自然是要按照他人已有的标准来写代码。

代码风格提示：变量命名使用蛇形命名，即 `connect_every_lowercase_word_by_underscore`

在给变量命名时，我们既不能命名地过于简单以至于难以分辨变量的作用，也不能命名过于复杂以至影响代码的可读性。我们之后学习常量、函数等时也是如此。

不推荐的命名	推荐的命名
s	sum
MathGrade	math_grade
studentId	student_id

另外，计算机科学中有一些常用的简写：

简写	含义
re	result
res	result
res	reference
dest	destination
stat	statement
exp	expression
sys	system
exec	execute
src	source
lib	library
var	variable
func	function
info	information

我们可以这样打印整数变量：

```

1  #include <stdio.h>
2
3  int main() {
4      int math_grade;
5      math_grade = 120;
6
7      printf("%d\n", math_grade);
8
9      return 0;
10 }
```

运行程序，输出了 120。我们目前只需要这样理解：引号内的 %d 是一个占位符，引号后面的 *math\_grade* 就是用来替换占位符的。因此我们就可以实现一些更复杂的输出了，比如：

```

1  printf("数学成绩为:%d\n", math_grade);
```

现在拓展一下，如果我们定义了数学、语文和英语三门课的成绩，也即：

```

1  int math_grade = 120;
2  int chinese_grade = 100;
3  int english_grade = 140;
```

如何输出呢？我们只需要写三个占位符即可，也就是：

```

1  printf("数学成绩为:%d, 语文成绩为:%d, 英语成绩为:%d\n", math_grade, chinese_grade, english_grade);
```

想象一下，假如我们定义了很多变量，有数学期中成绩，数学期末成绩，语文小测成绩，语文期末成绩……我们在代码里看这么多东西就头大，所以我们希望能用自然语言给它们加上注释，便于我们理解代码。在 c 语言中，有两种形式的注释（comment）：单行注释和多行注释。

单行注释是两个斜杠 (//)，这两个斜杠之后就可以随便写字了，比如：

```
1  #include <stdio.h>
2
3  int main() {
4      int math_grade; // 声明数学成绩
5      math_grade = 120; // 给数学成绩赋值120
6
7      printf("%d\n", math_grade); // 打印数学成绩
8      // 单行注释也可以单独成行
9
10     return 0;
11 }
```

多行注释的语法是/\* \*/，在这之间可以随便写字，比如：

```
1  #include <stdio.h>
2
3  int main() {
4      /*
5       该程序的展示了变量的声明与赋值语法，
6       以及使用printf打印整型变量的方法
7      */
8      int math_grade; // 声明数学成绩
9      math_grade = 120; // 给数学成绩赋值120
10
11     printf("%d\n", math_grade); // 打印数学成绩
12
13     return 0;
14 }
```

需要注意的是，多行注释不能这样写：

```
1  int math_grade; /* 声明数学成绩
2  math_grade = 120; 给数学成绩赋值 */
```

这样的话，第二段代码就被包括在了注释内，从而无法执行。

那么什么时候用单行注释，什么时候用多行注释呢？如果写注释的目的是对一行或几行代码做简单的说明，那么就用单行注释；假如注释的目的是要阐释一个复杂的原理（比如一段算法的数学原理或工程背景）或者解释一个函数（接下来会讲函数是什么）的功能，那么就应该使用多行注释。

有了变量，肯定也要有常量（constant）<sup>3</sup>。在c语言中，我们使用#define来声明常量。比如，圆周率 $\pi$ 就是一个常量，我们可以这样在c语言中定义圆周率：

```
1  #define PI 3
```

<sup>3</sup>还有一种使用const定义的“常量”，与我们此处介绍的有区别。const定义的是一种只读的变量，关于它的知识在后续才会学习，现称的常量均为define宏定义

也即，定义常量的语法是 `#define 常量名 常量的值`。

我们定义圆周率是 3，是一个整数，因此可以按照上文的方法将其打印出来：

```
1  #include <stdio.h>
2
3  #define PI 3
4
5  int main() {
6      printf("圆周率的近似值是%d\n", PI);
7
8      return 0;
9  }
```

你可能觉得奇怪，为什么定义常量的语法和变量区别这么大呢？既不需要写赋值号，也不需要标明类型。实际上，`#define` 是一个宏 (macro)，在 c 语言中，程序被编译运行前，会有一个叫做预处理器 (preprocessor) 的程序将所有宏替换为对应的值，也就是说（在只考虑 `#define` 宏的情况下），上面的代码经过预处理器会变成这样

```
1  #include <stdio.h>
2
3  int main() {
4      printf("圆周率的近似值是%d\n", 3);
5
6      return 0;
7  }
```

发现了吗？我们定义的常量 `PI` 被直接替换成了它的值 3，既然是直接替换，就相当于写死在了程序里，没法修改，所以就是常量了。实际上，我们行首的 `#include` 也是一个宏，它的效果和 `#define` 类似。关于宏和预处理的知识我们在之后会学习到。

代码风格提示：命名常量，一般采用大驼峰命名法 (CamelCase)，也即所有单词都连在一起，每个单词的首字母大写。如 `CapitalizeTheFirstLetterOfEachWord`

我们给出一个数字 1，这个数字的值是写在纸面上的，仅仅通过这个数字本身就可以知道。像这样的量就被称为字面量 (literal)。举几个例子：数字 42 是字面量，“hello”是字面量，但是变量 `int a` 不是字面量，当我们给 `a` 赋值时：

```
1  int a = 1; // 赋值号左边的a不是字面量,右边的1是字面量,即使a被赋值它也不是字面量,因为你无法仅仅从a这个名字本身判断它的值
```

那么常量呢？我们上文说明了常量在编译前会被替换为字面量，因此它也是字面量。字面量这个概念目前用处不大，但是在后续学习中会很有用。



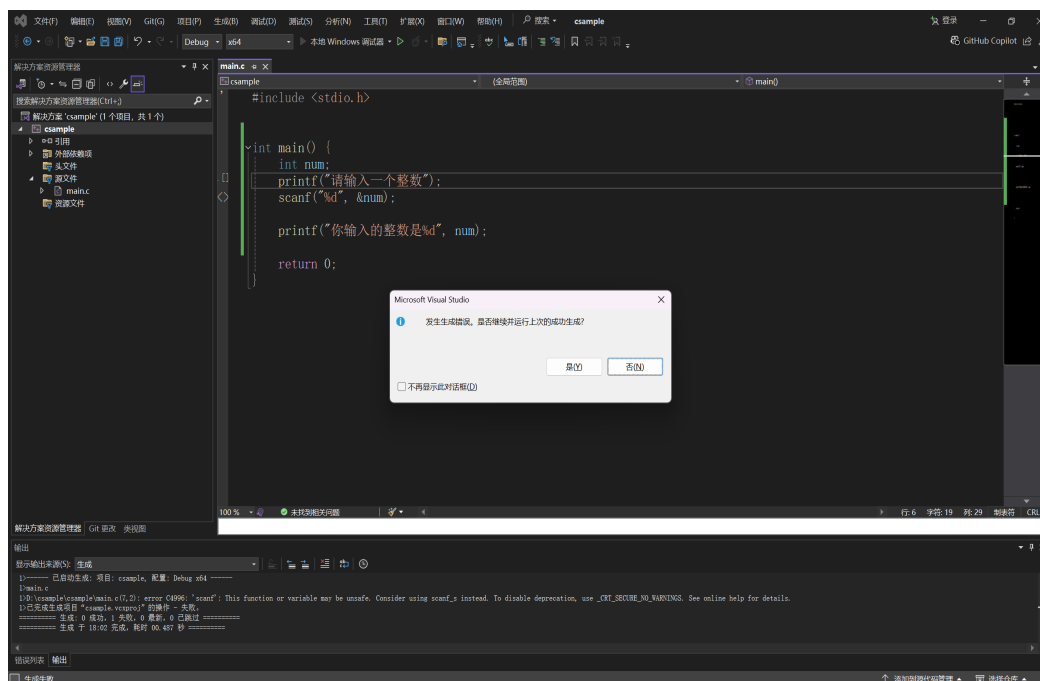
### 1.3 scanf 函数

scanf 函数 (scan format) 用来从终端获取输入的文本。这个东西涉及到一些我们还没有学到的知识，但是我们后续的学习很难离开它，因此需要在这里对其简单介绍，我们目前不需要了解它的原理，会用就可以。

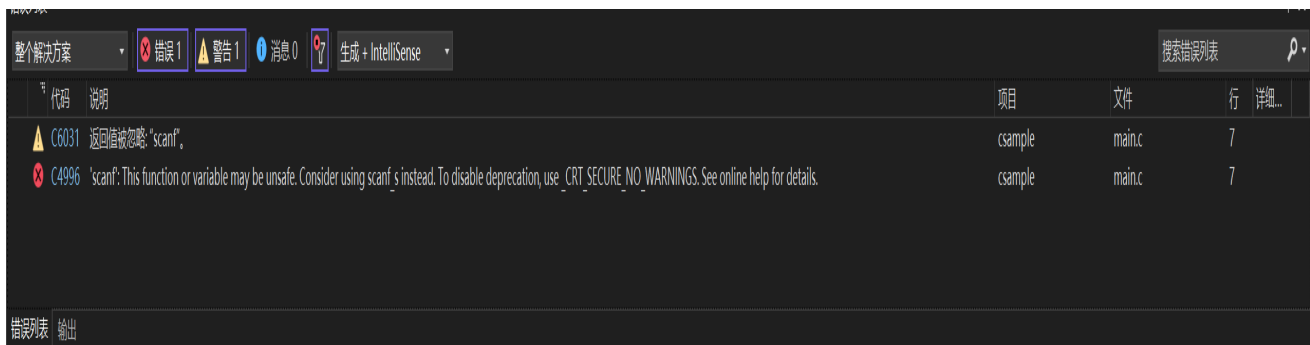
首先，打开 visual studio，在 main.c 中输入一段这样的代码

```
1 #include <stdio.h>
2
3 int main() {
4     int num; // 用于储存用户输入的数字
5     printf("请输入一个整数"); // 提示用户进行输入
6     scanf("%d", &num); // 将输入的数字保存在变量num中,注意不要忘了num前的&
7
8     printf("你输入的整数是%d", num);
9
10    return 0;
11 }
```

点击运行，出现了意料之外的情况



这个窗口的意思是我们的代码有问题，过不了编译。在实际编程开发时，由于我们很难做到一次就尽善尽美，所以常常会遇到这个窗口。还好代码的试错成本是很低的，我们只需要着手去修补自己的代码即可。我们点击否，注意到 visual studio 最下面出现了这样的文本



带有黄色警示标志的叫做警告（warning），它的意义是代码可能有一些不符合规范的地方，需要修改，但是不改也能运行。而红色标志的叫做错误（error），它的意义是这里的代码完全有问题，过不了编译，程序运行不了。我们必须消除代码中所有的 error，也要尽可能消除所有的 warning。

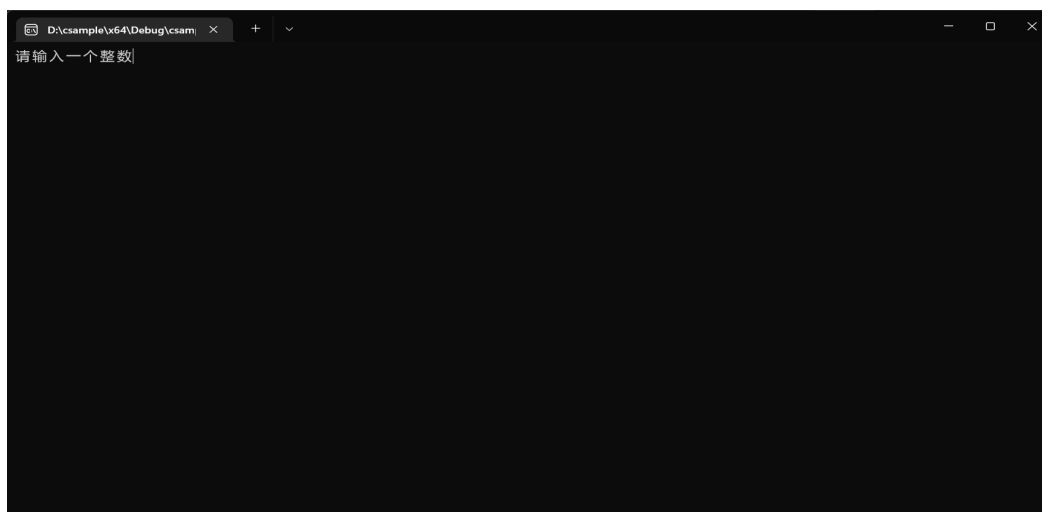
我们开始着手消灭这个 error，它的提示信息说 “This function or variable may be unsafe. Consider using scanf\_s instead. To disable deprecation, use \_CRT\_SECURE\_NO\_WARNINGS. See online help for details.”

翻译成中文，就是“这个函数或者变量可能不安全。考虑使用 scanf\_s 替代。若要取消废弃（检查），使用 \_CRT\_SECURE\_NO\_WARNINGS。详细信息见在线帮助。”。它的意思就是说，我们用的这个 scanf 函数不安全，已经废弃了，要么我们换用 scanf\_s，要么我们加上 \_CRT\_SECURE\_NO\_WARNINGS 来强行使用废弃的函数。

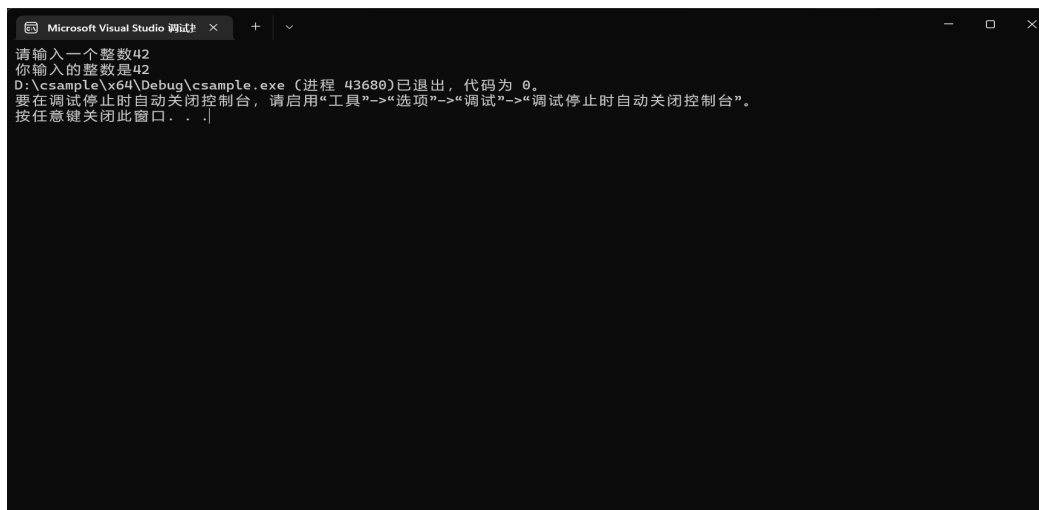
实际上，scanf 由于设计不当，会有缓冲区溢出（buffer overflow）的风险，所以我们确实不应该使用它。但是呢，只有我们使用的 visual studio 会报这个错误，其它 ide 不会强制要求你换，而且大部分的 c 语言试题和 c 语言教程仍在使用 scanf，所以我们采取强制使用废弃函数的方法来消除这个 error。

在 visual studio 的项目右键，点击属性，点击“C/C++”，点击“预处理器”，点击“预处理器定义”，随后点击右边出现的小箭头，点击“<编辑>”，单独一行输入“\_CRT\_SECURE\_NO\_WARNINGS”，点击“确定”即可。

此时我们再运行代码，终端首先变成这样



此时我们输入一个数字，点击回车，出现了这样的效果



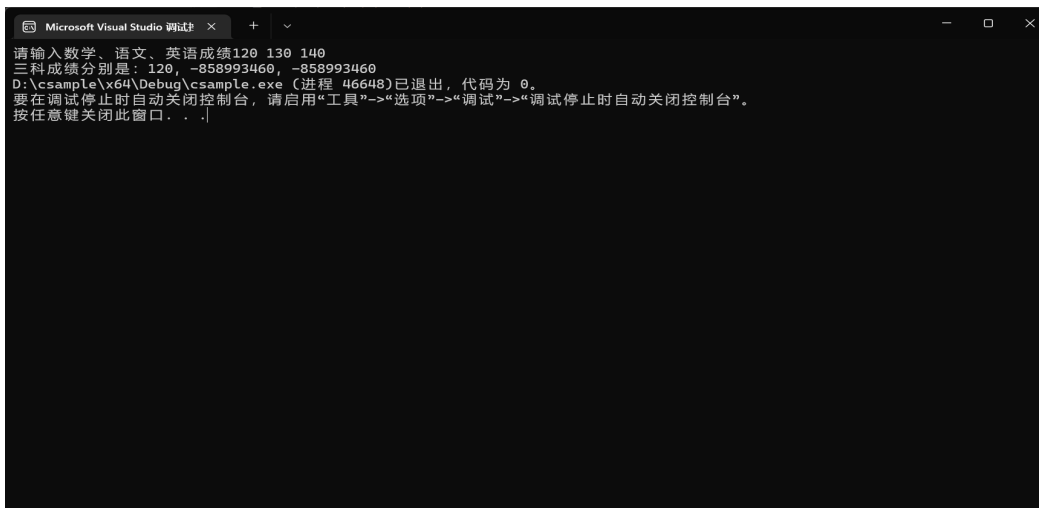
如果我们要进行的是更复杂的输入呢？比如，我们需要数学、语文和英语三科成绩，那么可以这样写

```
1  #include <stdio.h>
2
3  int main() {
4      int math_grade;
5      int chinese_grade;
6      int english_grade;
7
8      printf("请输入数学、语文、英语成绩");
9      scanf("%d %d %d", &math_grade, &chinese_grade, &english_grade);
10
11     printf("三科成绩分别是:%d, %d, %d", math_grade, chinese_grade, english_grade);
12
13     return 0;
14 }
```

我们只需要按照这样输入：120 130 140，就能输出正确的成绩了。如果我们希望输入是 120,130,140，那么就把 scanf 的一行改成

```
1  scanf("%d,%d,%d", &math_grade, &chinese_grade, &english_grade);
```

这样我们就找到了规律，scanf 的引号内的部分就像是一个模板，只有输入与模板对应，函数才正常运行。不信的话，我们可以试试在修改后的代码输入 120 130 140，会得到这样的结果



非常抽象是不是？你知道更抽象的是什么呢？假如我们使用其它编译器，那么会得到不同的结果<sup>4</sup>。这是因为我们的输入没有符合模板的需求，模板需要我们在每两个成绩间使用英语逗号分割，我们用的却是空格，这种情况被称作未定义行为（undefined behavior），这是c语言中比较难以预测的一种的错误，我们后续会详细讲解。不知道各位记不记得，我们在解决那个error时，留了个warning尚未处理？它说我们没有使用scanf的返回值，这个“返回值”就是用来解决这个问题的，我们学习到函数时会专门提到。

## 2 基本数据类型和数值运算

### 2.1 基本数据类型

我们在上一章介绍变量的时候就已经使用了代表整数的int，但是只有整数显然是远远不够用的，c语言还提供了不同取值范围的整数、小数、字符和字符串。这些统称为基本数据类型。

在计算机科学中，整数一般被称作整型。根据取值范围的不同，总共有四种整型：short（短整型），int（整型），long（长整型）和long long，可以这样声明整型变量

```
1 short a = 1;
2 int b = 2;
3 long c = 3;
4 long long d = 4;
```

short 的取值范围是  $[-32768, 32767]$ <sup>5</sup>，int 的取值范围是  $[-327682147483648, 327672147483647]$ ，long 和 long long 的就更大了。实际上还有一类无符号整型，就是只有非负数的整型。在整型前加一个 unsigned 就是对应的无符号整型

```
1 // 无符号类型
2 unsigned short a = 1;
```

<sup>4</sup>类似的一段代码，以 120 130 140 作为输入，64 位电脑，处理器 intel ultra9，在 win11，msvc 编译下运行结果为 120, -858993460, -858993460；ubuntu，gcc 编译下结果为 120, 4096, 0；ubuntu，clang 编译下结果为 120, 0, 4096

<sup>5</sup>以笔者电脑（64 位 win11，MSVC STL）的 limits.h 作为标准

```
3 unsigned int b = 2;
4 unsigned long c = 3;
5 unsigned long long d = 4;
```

short 的范围是  $[-32768, 32767]$ ，而 unsigned short 的范围是  $[0, 65535]$ ，相当于把负数部分都转移给正数了。其它类型对应的无符号类型也是一样。

介绍了这么多，笔者建议大家：除非你有明确的设计要求，对计算机了解比较清晰，非常清楚自己在写什么，否则就一律使用 int。哪怕是明确了需要处理的数据都是非负也不要无符号，哪怕是确定 short 就够包含自己需要的数据也要用 int。

为什么我们这么说？首先，为什么 c 语言设计了这么多种整型和无符号这些东西呢？我们知道，在 c 语言诞生的年代，内存是十分紧张的资源，所以设计多种占用不同内存大小的整型也是便于程序员精准控制内存。但是现代计算机内存大多都是 8GB，16GB 甚至更高，已经没有必要过于珍惜内存了。

如果我们使用推荐外的类型，一方面会增大开发难度，另一方面容易诞生意料以外的问题。为什么呢？首先请考虑一个问题，以 short 为例，它的最大值是 32767，那假如我们给一个 short 变量赋值 32768 会怎么样？我们写一段代码试一下

```
1 #include <stdio.h>
2 #include <limits.h>
3
4 int main() {
5     short a = SHRT_MAX + 1; // SHRT_MAX就是short的最大值,即32767
6     printf("%hd\n", a);
7
8     return 0;
9 }
```

运行结果是-32768，也就是说，如果我们赋的值超过了数据类型取值范围的最大值，它就会从最小值开始赋。一样的，若我们赋的小于范围最小值，就会从最大值开始赋。这就像是把取值范围这根数轴首尾连接在了一起一样。类似的结果也会出现在其它类型，这是 c 语言的特殊设计。

想象一下，假如你在设计一个学生管理软件，为了节省内存，使用 short 来记录学生的编号（因为你觉得学校小，不可能超过三万人），但是使用你开发的系统的学校今年突然人数暴增，现在学校内有了 32768 人，那么最后一个同学的编号就是-32768，这位同学得有多无奈？因此，一般不要使用比 int 小的类型的。

既然使用 short 有这样的危害，为什么不都使用 long 呢？首先，实际场景中很少存在数据大到连 int 都装不下的情况（谁家学校有三百多亿人啊），因此没必要选这么大的；另一方面，如果需要处理的数据都是普遍极大的，一般都是由开发者自行实现适用的记录长数字的类型，使用 long 不划算。所以我们也一般不考虑比 int 大的类型。

那么为什么无论使用场景都不用无符号呢？首先，无符号的最大值和最小值也是连接在一起的，这种情况更刺激：假如一个学校有 65536 名学生，我们使用 unsigned short 来编号，那么最后一位和第一位的编号是一样的，就没办法区分这二人了。其次，我们有时设计的一些复杂算法可能出错，若算法应只输出正值却输出了负值，那么说明算法一定出了问题，假如我们使用无符号类型，就无法这样判断了。

为什么不同整型有不同取值范围呢？这实际上与其占用的比特数有关。c 语言提供了 sizeof 运算符，用于获取一个数据类型占用的比特数（比特数是一个整型）。比如，我们可以这样获取 int 所占用的比特数

```

1  #include <stdio.h>
2
3  int main() {
4      int s = sizeof(int);
5      printf("%zd\n", s);
6
7      return 0;
8  }

```

输出结果是 4。类似的我们可以知道其它整型的大小

```

1  #include <stdio.h>
2
3  int main() {
4      printf("%zd\n", sizeof(short));
5      printf("%zd\n", sizeof(int));
6      printf("%zd\n", sizeof(long));
7      printf("%zd\n", sizeof(long long));
8
9      printf("%zd\n", sizeof(unsigned short));
10     printf("%zd\n", sizeof(unsigned int));
11     printf("%zd\n", sizeof(unsigned long));
12     printf("%zd\n", sizeof(unsigned long long));
13
14     return 0;
15 }

```

我们发现，占用比特数越大，储存数字的范围越大。这是一个定性的结果，我们在更深层次的课程会详细解释储存数据的原理、取值范围如何计算。

小数在计算机科学一般称作浮点型，常用的是 double（双精度浮点型），还有一种叫做 float，这个用法与 double 一致，但是我们推荐使用 double。我们可以这样定义一个浮点型变量

```

1  double a = 1.23; // 小数点是英文句号

```

如果给浮点型赋一个整数（比如赋 1），我们习惯这样赋值

```

1  double b = 1.0; // 也即要加上.0

```

如何打印浮点数呢？打印 double 的占位符是 %lf，打印 float 的占位符是 %f，比如

```

1  #include <stdio.h>
2
3  int main() {
4      double a = 3.14;
5      printf("%lf\n", a);
6
7      return 0;

```

```
8 }

```

输出结果是 3.140000。默认的%lf 为我们保留了六位小数，如果我们把 a 改成 3.1415926，输出结果是 3.141593，不仅保留了六位小数，还做了四舍五入。实际上，%f 也可以用在打印 double 上，但是使用 scanf 时就必须要用%lf 了。因此我们推荐在处理 double 时不要混用，一律使用%lf。

如果小数太长，我们只需要显示前两位，我们就可以这样使用占位符

```
1 printf("%.2lf\n", a);

```

它的意思是要保留小数点后两位数字，还是很形象的吧？假如我们这么写，并且把 a 的值改成 3.14

```
1 printf("%5.2lf\n", a);

```

输出就变成了“ 3.14”，注意多出一个空格。对于%a.bf 和%a.blf，小数点之前的数字代表最少要输出多少位（注意小数点也算一位），之后的数字代表要输出多少位小数。如果要求最少输出位数比数字本身有的位数还多，那就会在最前面输出空格来补齐差距。

还记得字面量吗？数字 42 是字面量，更具体地说，它是整型字面量；数字 3.14 是浮点型字面量；“xyz”是字符串字面量（字符串在后续讲到）。总之，字面量可以更精确地描述为类型+字面量。

## 2.2 数值运算

整型和浮点型都具有加减乘除四种数值运算，整型还有取模运算。一个整型可以与另一个整型做数值运算，也可以与一个整型字面量数值运算，两个整型字面量也可以进行运算。加减乘除的符号分别是+ - \* /，就像数学的写法一样，在运算符两边写参与运算的数字。像是算一加一就是 1 + 1，算二乘二就是 2 \* 2。另外需要特别说明，赋值号只在赋值号右边的运算都完成后才会把右边的值赋给左边。举个例子

```
1 #include <stdio.h>
2
3 int main() {
4     int a = 1; // a和b都是整型变量
5     int b = 2;
6     int c = a + b; // 整型变量进行数值运算,得到的值赋给了c。注意:只有在赋值号右边的运算完成后,赋值号才进行
        赋值
7     printf("%d\n", c); // 输出结果是3
8
9     int d = 1;
10    int e = d * 4; // e由一个整型变量和整型字面量计算得到
11    printf("%d\n", e); // 输出结果是4
12
13    int f = 2 - 2; // f由两个整型字面量计算得到
14    printf("%d\n", f); // 输出结果是0
15
16    return 0;
17 }
```

最需要强调的是除法运算。整型的运算只能算出整型，所以假如参与除法运算的两个数不能整除怎么办？我们试一试

```
1  #include <stdio.h>
2
3  int main() {
4      int a = 2;
5      int b = 3;
6      int c = a / b;
7      printf("%d\n", c);
8
9      return 0;
10 }
```

照理来说， $2/3$  的结果是 0.666 循环，但是这段代码的输出是 0。这说明，整型的除法在无法整除的时候，会截取整数部分作为运算结果。另外考虑一个问题，以 0 作为除数会怎么样？0 作为除数是没有意义的，在 c 语言中以 0 为除数的后果就会导致未定义行为。

除此以外，整型还有取模运算，也就是获取两数相除的余数。取模运算的符号是 %。比如

```
1  #include <stdio.h>
2
3  int main() {
4      int a = 20;
5      int b = 3;
6      int c = a % b;
7      printf("%d\n", c);
8
9      return 0;
10 }
```

输出结果是 2，说明  $20/3$  的结果余 2。这个运算非常重要，比如我们可以通过一个数与 2 的余数是否为 0 来判断这个数是否为偶数：如果余数为 0，说明这个数能被 2 整除，那就是偶数，反之就是奇数。

浮点数的加减乘除运算语法和整型的是一致的，不过浮点数自然没有取模运算。我们刚刚尝试整型的除法运算，发现运算结果是截取了商的整数部分，那浮点型会怎样呢？

```
1  #include <stdio.h>
2
3  int main() {
4      double a = 2.0;
5      double b = 3.0;
6      double c = a / b;
7      printf("%lf\n", c);
8
9      return 0;
10 }
```



运行，发现结果是 0.666657，说明浮点型的运算结果也是浮点型。实际上，整型与浮点型运算的结果也是浮点型，常见的是使用一个浮点型除以一个整型字面量，比如我们这样求三个数的平均值

```
1 double a = 2.0;
2 double b = 3.0;
3 double c = 4.0;
4
5 double avg = (a + b + c) / 3;
```

在 c 语言中，不同的数据类型可以进行转换。比如我们可以这样将一个整型转成浮点型

```
1 int a = 1;
2 double b = (double) a; // b = 1.0, 需要转成什么类型就在括号里写什么类型
```

这样，就将 a 转换成了一个浮点型变量并赋给了 b。注意，转换的意义是生成了一个浮点型变量，而不是将 a 本身变成了浮点型。

一样的，我们可以把浮点型转换成整型。但是这样就有一个问题，比如我们将 4.5 转成整型，结果是多少呢？我们写一段代码试试

```
1 #include <stdio.h>
2
3 int main() {
4     double a = 4.5;
5     int b = (int) a;
6
7     printf("%d\n", b);
8
9     return 0;
10 }
```

结果是 4。由此我们发现，在浮点型转换为整型时，会直接截去小数点。

我们可以想到，浮点型能储存的数字比整型多，因此整型向浮点型转化是一定没有问题的，但是浮点型转整型可能会丢失小数点后的信息。也是由此，整形转浮点型是不需要像上面那样明确写出来的，编译器会自动帮我们进行转换，这叫做隐式转换。而浮点向整型的转换是需要我们按照上面的方法手动进行的，叫做显式转换。

c 语言中算数是有优先级的，与数学一样，即从左到右，先算乘除后算加减，括号可以改变运算顺序。比如

```
1 int a = 4 * 2 + 2;
2 int b = 4 * (2 + 2);
```

a 的值是 10，b 的值是 16。注意，如果你需要套多层括号来改变运算顺序，都使用圆括号（即 ()）而不要使用方括号和花括号。

我们刚刚提到了赋值号是先对右边进行运算再赋给左边，所以我们实际上可以写出这样的东西

```
1 int a = 1;
2 a = a + 1;
```

首先对右边计算，是  $1+1=2$ ，随后将 2 赋给了 a，此时 a 就变成 2 了。当然了，对于其它几种数值运算也能这样使用。

像上面这样的运算是很常用的，但是每次都写成那样岂不是很麻烦？c 语言提供了一种简化语法

```
1  int a = 1;
2  a += 1; // 等价于 a = a + 1;
3  // 类似的, a *= 2就等价 a = a * 2,诸如此类
```

在后续学习到流程控制时，我们需要常常对一个整型变量进行加一或者减一的运算，此时哪怕是上面的简化语法也太累赘了，因此 c 语言设计了自增运算符和自减运算符，用于对整型进行加一或减一的操作（其实对浮点型也适用，不过最常用于整型）。

自增运算符和自减运算符都分为前置和后置两种

```
1  int a = 1;
2  ++a; // 前置自增运算符
3  a++; // 后置自增运算符
4  --a; // 前置自减运算符
5  a--; // 后置自减运算符
```

无论是哪种自增运算符，都能使变量的值加一。无论是哪种自减，都能使值减一。

前置和后置运算符的区别在于将它们与赋值并用的时候

```
1  int a = 1;
2  // 下面两段代码不能放在一起,这里仅作为演示
3  int j = ++a; // j = 2
4  int j = a++; // j = 1
```

当赋值运算和前置运算符并用时，会先自增后赋值，因此 j 的值就是 2。当与后置运算符并用时，会先赋值后自增，所以 j 获得的是原来的值 1。

记不住怎么办？记不住就老老实实写成这样

```
1  int a = 1;
2  ++a;
3  int j = a;
```

另外，虽然使用上面这样的写法就无需考虑使用前置还有后置运算符了，但是笔者推荐养成使用前置运算符的习惯，这样程序的效率更高。这是从 C++ 中获取的经验，不多过解释了。（虽然对于 Java 等语言来说这两种运算符是完全等价的，但是对于 C 和 C++ 来说还是有区别）

最后，我们写一个实例：让用户输入三个整数，计算它们的平均值并返回。

对于这种实战型的题目，我们的一般方法就是先明确设计需求，然后大概确定技术要点，最后进行开发即可。

对于我们这个实例，设计需求就是：一，获取三个整数；二，计算三个数字的平均数并输出。那么这涉及什么技术呢？对于第一个需求，需要掌握 scanf 的用法，对于第二个，需要掌握变量的数值运算和 printf 的用法。我们都学过，所以开始编程吧。

首先，我们创建新的 main.c，输入必要的代码

```

1  #include <stdio.h>
2
3  int main() {
4      return 0;
5  }

```

我们首先处理第一个需求。获取三个整数，这就需要我们把它储存起来，因此先定义三个整型变量

```

1  int a;
2  int b;
3  int c;

```

接下来就是要获取三个数字并进行储存了。我们可以写成这样

```

1  scanf("%d %d %d", &a, &b, &c);

```

这样，用户只需要按照模板进行输入，程序就可以正常工作。可是，用户该怎么知道这个程序是干什么的，他应该怎么输入呢？因此，我们在 scanf 前加一个 printf 来打印基本信息。这样，程序就改成了

```

1  printf("欢迎使用平均数计算器!请按照a b c的格式输入三个整数\n");
2  scanf("%d %d %d", &a, &b, &c);

```

在开发的过程中，应该开发一部分就进行一次测试。因此我们此时可以点击运行试试。发现出现了我们想要的界面，按照模板进行输入，程序以退出代码 0 退出。退出代码 0 是程序正常完成的标志。由此完成了第一个需求。

接下来我们需要计算平均值了，定义一个变量 avg 作为平均值，可是它应该是什么类型呢？我们想，三个整数的平均数不一定是一个整数，所以要用浮点型。因此就定义了

```

1  double avg;

```

接下来就要计算平均数了，于是我们写下这样的代码

```

1  double avg = (a + b + c) / 3;

```

然后输出

```

1  printf("平均数为:%lf\n", avg);

```

大功告成！我们运行程序，试一下效果。首先我们输入 1 2 3，发现输出是 2.000000，是我们预期的结果！那我们再试试 1 2 4，发现结果不对了，怎么还是 2.000000？

这说明，我们的代码出了错。回顾一下，输入部分的代码已经做了测试，输出数字也不会有问题，看来就是计算平均数这步代码出错了。那么为什么呢？注意，abc 三个变量实际都是整型，它们的和是一个整型，除以 3 是整型的数值运算，得到的结果自然也是整型，也就是 2。这个结果隐式转化成了浮点型，因此输出结果是 2.000000。也就是说，我们的问题在于是使用整型进行运算的。如果我们把代码改成这样

```

1  double avg = (double)(a + b + c) / 3;

```

这样，三个变量求和的结果会强制转换成浮点型，使用浮点型的运算算出的才是正确结果。修改后我们再次运行代码，发现这次的输出 2.333333 是正确的。这样我们就完成了整个项目的设计，完整代码如下

```
1  #include <stdio.h>
2
3  int main() {
4      int a;
5      int b;
6      int c;
7
8      printf("欢迎使用平均数计算器!请按照a b c的格式输入三个整数\n");
9      scanf("%d %d %d", &a, &b, &c);
10
11     double avg = (double)(a + b + c) / 3;
12
13     printf("平均数为:%lf\n", avg);
14
15     return 0;
16 }
```

## 3 流程控制 选择语句

### 3.1 布尔代数基础

我们一切命题，最终都会回到“是”与“否”的基本判断上。我们若使用 1 代表真，使用 0 代表假，便很容易得到一些逻辑运算。

首先是和关系。我们说，“我和那位同学都是女生”，这句话成立的条件是我是女生，并且那位同学也是女生。也就是说，和关系意味着两个条件都成立。

$$0 \text{ and } 0 \Rightarrow 0$$

$$1 \text{ and } 0 \Rightarrow 0$$

$$0 \text{ and } 1 \Rightarrow 0$$

$$1 \text{ and } 1 \Rightarrow 1$$

类似地，或关系成立的条件是两个条件成立一个即可。比如“A 或 B 是女生”，只要 A 与 B 之一是女生就成立，若都是女生也成立。

$$0 \text{ or } 0 \Rightarrow 0$$

$$1 \text{ or } 0 \Rightarrow 1$$

$$0 \text{ or } 1 \Rightarrow 1$$

$$1 \text{ or } 1 \Rightarrow 1$$

还有一种基本的就是非关系。“我不是女生”若成立，就说明“我是女生”不成立，因此

$$\text{not } 0 \Rightarrow 1$$

$$\text{not } 1 \Rightarrow 0$$

使用以上三种运算，我们可以结合出很多复杂的运算。比如，“A 是女生或 B 不是女生”，就是 A 是女生或者 B 是男生时成立。总之，这样的逻辑运算一定要能够想明白。

## 3.2 if 语句

if 语句用于条件判断，若判断为真，就进入下属的语句，反之就不进入。它的语法是 if (表达式) 下属语句。如果表达式是真的，那么就会进入下属语句，否则就不进入。

一个常见的判断真假的场景就是判断两个数字的大小关系。c 语言提供了这些运算符用于判断数字（整型和浮点型）的大小关系

关系	操作符
等于	==
不等	!=
小于	<
大于	>
小于等于	<=
大于等于	>=

这些符号可以这样使用

```

1  #include <stdio.h>
2
3  int main() {
4      int a = 5;
5      int b; // 储存输入的数字
6
7      printf("输入一个数,比较其与5的大小关系!\n");
8      scanf("%d", &b);
9
10     // 判断a和b(也即预设的数字和输入的数字)大小关系

```

```

11     if (a <= b) {
12         printf("输入的数大于等于5\n");
13     }
14     if (a > b) {
15         printf("输入的数小于5\n");
16     }
17
18     return 0;
19 }

```

```

1  #include <stdio.h>
2
3  int main() {
4      int a = 5;
5      int b; // 储存输入的数字
6      printf("请输入一个数,猜猜和预设的是否一样\n");
7      scanf("%d", &b);
8
9      // 判断a和b(也即预设的数字和输入的数字)是否相等,如果相等就进入下属语句
10     if (a == b) {
11         printf("你猜对了!");
12     }
13     // 判断a和b(也即预设的数字和输入的数字)是否相等,如果不等就进入下属语句
14     if (a != b) {
15         printf("你猜错了!");
16     }
17
18     return 0;
19 }

```

在讲取模运算符的时候，我们提到，它可以用来判断一个数字的奇偶性，我们来试试设计一个程序，输入一个整数，判断其奇偶性。

开发项目一定要按照我们之前提过的步骤进行。首先我们确定设计需求：一，输入一个整数；二，判断这个整数的奇偶性；三，根据奇偶性分别输出。第一个需求使用 scanf 即可，第二个使用我们学习过的取模运算符进行判断即可，第三个使用 if 语句就能实现。我们来编写代码。

第一个需求是比较简单的，我们直接给出代码

```

1  #include <stdio.h>
2
3  int main() {
4      int a;
5      printf("请输入一个整数\n");
6      scanf("%d", &a);
7
8      return 0;

```

```
9     }
```

接下来就要判断奇偶性了。我们之前提到，一个数字若是偶数，那么它与 2 的模一定为 0（因为偶数一定被 2 整除），因此对于输入的数字，如果与 2 取模的结果是 0，那么就是偶数，反之就是奇数。因此我们首先写出了判断偶数的程序

```
1     if (a % 2 == 0) {
2         printf("你输入了一个偶数\n");
3     }
```

那么判断奇数怎么写？仔细想想，一个数与 2 的余数，除了 0，只能是 1。因此

```
1     if (a % 2 == 1) {
2         printf("你输入了一个奇数\n");
3     }
```

这样我们就完成了整个程序的设计。完整代码如下

```
1     #include <stdio.h>
2
3     int main() {
4         int a;
5         printf("请输入一个整数\n");
6         scanf("%d", &a);
7
8         if (a % 2 == 0) {
9             printf("你输入了一个偶数\n");
10        }
11        if (a % 2 == 1) {
12            printf("你输入了一个奇数\n");
13        }
14
15        return 0;
16    }
```

上面的代码确实完成了我们的需求，但是我们发现，一个数要么是奇数，要么是偶数，我们其实只需要判断一次就能知道数的奇偶性，而不是需要像现在这样费力地构造逻辑上相反的语句。所以如果 if 语句可以有两个从属语句，如果条件满足就转向第一个，反之转向第二个，我们的程序就能大幅简化。c 语言提供了这样的语法，就是 else 语句。else 语句跟在 if 语句后，就像是

```
1     if (a % 2 == 0) {
2         printf("你输入了一个偶数\n");
3     } else {
4         printf("你输入了一个奇数\n");
5     }
```

使用 else 语句改造后的代码就直观简洁多了。

我们之前提到了布尔代数，三种基本的布尔运算在 c 语言中也有对应的语法。

布尔运算	符号
和运算	&&
或运算	
非运算	!

比如说，我们可以这样判断一个数是否包含于 [10, 100] 的区间内。

```
1  if (a >= 10 && a <= 100)
2  // 注意, 不能写成这样: if (10 <= a <= 100)
```

我们可以这样判断一个数是否同时满足不是 3 的倍数或是偶数

```
1  if (a % 3 != 0 || a % 2 == 0)
```

我们之前写了这样的代码来判断一个数是否为偶数

```
1  if (a % 2 == 0)
```

实际上，上面的式子可以改成这样

```
1  if (!(a % 2))
```

这是在干什么？最内层括号计算了 a 与 2 的余数，如果 a 是偶数，那么得到的结果就是 0，我们想要进入下属语句，因此把运算结果使用非运算反过来就可以了。一样的，如果我们输入的是奇数，得到的结果是 1，非运算后就变成了 0，就不会进入下属语句了。因此，我们将判断奇偶数改成了一种更加简洁的语法。注意不要遗漏了取模运算的括号，否则非运算先与 a 结合，就得不到我们期望的结果了。

有了布尔代数的知识，我们可以解决一类更复杂的问题。比如，给定一个成绩，要判断它属于不及格 ([0, 60))，及格 ([60, 80))，良好 ([80, 90)) 还是优秀 ([90, 100])，我们怎么写呢？最开始的想法是这样的

```
1  #include <stdio.h>
2
3  int main() {
4      int a;
5      printf("请输入成绩\n");
6      scanf("%d", &a);
7
8      if (a >= 0 && a < 60) {
9          printf("未及格!\n");
10     }
11     if (a >= 60 && a < 80) {
12         printf("及格!\n");
13     }
14     if (a >= 80 && a < 90) {
15         printf("良好!\n");
16     }
```



```

17     if (a >= 90 && a <= 100) {
18         printf("优秀!\n");
19     }
20
21     return 0;
22 }

```

这样有三个问题：一，假如输入的数字大于 100，或者小于 0 怎么办？我们还需要写一个 if 来捕获这种情况；二，我们的程序是按照顺序执行的，比方说我们输入的数字是 65，在第二个 if 已经捕获了，照理来说就不需要再去判断下面的几个 if 了，但是它还是会执行下去，这样浪费了很多性能；三，一系列的 if 逻辑上是连续的，但是我们的代码表现出来是孤立的，这样增大了阅读难度。

怎么解决呢？c 语言提供了 else if 语句，我们将上面的代码改成

```

1     if (a >= 0 && a < 60) {
2         printf("未及格!\n");
3     }
4     else if (a >= 60 && a < 80) {
5         printf("及格!\n");
6     }
7     else if (a >= 80 && a < 90) {
8         printf("良好!\n");
9     }
10    else if (a >= 90 && a <= 100) {
11        printf("优秀!\n");
12    }
13    else {
14        printf("输入数据有误\n");
15    }

```

这样解决了我们提到的三个问题。else if 按照从上到下的顺序执行，如果第一个 if 没有匹配到，就转到下一个 if，直到匹配到为止。匹配到之后就执行匹配到的 if 的下属语句，而不会继续执行 if 语句。此外，我们还可以在最后缀一个 else 语句来处理缺省情况。

c 语言提供了一个特殊的运算符，叫做三目运算符，用来快速给变量赋值。比如

```

1     int a = b > 0 ? 4 : 5;

```

此处的英文问号和后面的英文冒号就是三目运算符了。它的含义是，当问号前的语句成立时，就使用冒号前的值，反之使用冒号后的。在我们的例子中，如果 b>0 是成立的，就使用冒号前的值，a 的值就是 4，如果 b>0 是不成立的，那么 a 就取 5。这个运算符适用于上面例子中的这种简单场景，不要滥用，否则会增大代码的阅读难度。

流程控制语句是可以嵌套的，比如

```

1     if (a > 0) {
2         if (a % 2 == 0) {
3             printf("a是一个大于0的偶数\n");

```

```

4     } else {
5         printf("a是一个大于0的奇数\n");
6     }
7 }

```

在嵌套时，else 语句总是和与其最近的 if 语句匹配，这点我们观察大括号的嵌套也能看出来。我们在之后的学习和开发实战中会经常用到流程控制语句的嵌套。

实际上，c 语言还提供了一种 switch 语句，类似于 if else，因为这个东西基本没用，我们在学习字符的时候再进行介绍。

我们遇到的有关条件判断的场景不一定是对数字大小关系的判断，对于其它情况要怎么写 if 语句内的条件呢？实际上，if 语句内的条件是一个整型，如果为 0 就判断为假，其它数字都判断为真（不过普遍使用 1）。比如

```

1     if (1) {
2         printf("Hi\n");
3     }

```

就会执行下属语句。

实际上，我们上面使用判断数值大小关系的运算符，就会返回一个整型，比如

```

1     int a = 1;
2     int b = 2;
3     int c = a == b;
4     int d = a != b;

```

那么 c 的值就是 0，d 的值是 1。

## 4 流程控制 循环语句

### 4.1 while 语句

我们经常遇到这样的场景：需要一直做某个任务直到达到某个条件为止。在 c 语言中就有这样的设计，即循环语句。第一种循环是 while 语句，它的语法是 while (条件) 下属语句。如果条件满足，就进入下属语句，下属语句执行完后程序会重新判断条件，如果还满足就再进入下属语句，直到条件不再满足为止。比如，我们若想要打印 1 到 100，就可以这样做

```

1     #include <stdio.h>
2
3     int main() {
4         int a = 1;
5         while (a <= 100) {
6             printf("%d\n", a);
7         }
8
9         return 0;

```

```
10 | }
```

我们首先定义了一个变量 `a` 为 1，随后定义了一个循环语句，它打印 `a` 的值，直到 `a` 为 100 为止。这样，当 `a` 小于等于 100 时，下属语句就会一直执行，输出 `a` 的值，当 `a` 达到 100 后，循环退出，程序结束。看上去很完美不是？但是，假如我们运行这段代码（不建议大家尝试），你会发现电脑一直在输出 1（有些电脑甚至会卡死）。

这是为什么？仔细想想，我们发现在循环的过程中，`a` 的值始终是 1，没有增长过，因此循环一直在执行而不会终止。所以，我们要在循环里加一条语句来使 `a` 增长。`while` 语句改成了

```
1  while (a <= 100) {
2      printf("%d\n", a);
3      ++a;
4  }
```

再次运行，这次结果正确了。

我们说，流程控制语句是可以嵌套的，我们可以把 `if` 和 `while` 语句嵌套在一起，比如，可以这样输出 1 到 100 之间的全部偶数

```
1  #include <stdio.h>
2
3  int main() {
4      int a = 1;
5      while (a <= 100) {
6          if (a % 2 == 0) {
7              printf("%d\n", a);
8          }
9          ++a;
10     }
11
12     return 0;
13 }
```

我们发现，我们为了循环而特意声明的变量 `a` 就像是一个计数器一样。实际上大部分时候这个变量的作用就是充当计数器，控制循环的进行。写计数器的时候一定不要忘记它的自增。这种充当计数器的变量一般命名为 `i`、`j` 和 `k`。如果你在同一段代码内需要超过三个计数器，那么应该考虑是否要简化循环。

`c` 语言提供了 `continue` 和 `break` 关键字，前者可以退出一次循环，后者用于直接退出整个循环。比如，上面的例子可以改写成

```
1  #include <stdio.h>
2
3  int main() {
4      int a = 1;
5      while (a <= 100) {
6          if (a % 2 != 0) {
7              ++a;
```

```

8         continue;
9     }
10    printf("%d\n", a);
11    ++a;
12 }
13
14 return 0;
15 }

```

这是什么意思呢？每次在 while 下属语句时，都会首先判断 a 是不是奇数，如果是就进入 if 的下属语句，a 自增，随后退出本次循环，因此下面的 printf 是不会被执行的。

再比如，假如我们有一个需求：输出一个数字，打印从 1 到这个数字之间的所有数字，如果打印到 9 就不再打印，那么就可以这样写

```

1  #include <stdio.h>
2
3  int main() {
4      int a = 1;
5      int b;
6      printf("请输入一个大于1的整数\n");
7      scanf("%d", &b);
8
9      while (a <= b) {
10         if (a == 9) {
11             break;
12         }
13
14         printf("%d\n", a);
15         ++a;
16     }
17
18     return 0;
19 }

```

在循环的过程中，如果 a 增长到了 9，就会进入 if 的下属语句，从而退出循环。这两个例子其实并没有体现出 continue 和 break 的真正作用，我们在之后的实战学习中会领略到这二者的真正作用。

值得注意的是，如果我们嵌套了多层循环语句，那么 continue 和 break 作用于最内层的循环。我们在学习到复杂嵌套时，各位可以自行验证这个结论。

有了循环的知识，我们就可以试着编写一个程序，它可以不间断地接受输入，直到输入了某个特定的退出符号为止。比如，我们可以编写一个程序来计算若干个正整数的平均数，如果输入非正数就退出循环。

为了实现这个程序，我们还是要遵循设计程序的三个步骤。首先确定需求：一，程序要能够不间断地输入，我们可以考虑使用 while 语句

```

1  while (条件) {
2      scanf("%d", &a);

```

```
3     }
```

现在出现了两个问题：第一，while 的条件应该是什么？第二，如果把输入结果存放在一个变量里，那么下次的结果不就覆盖上一次的了吗？

第一个问题实际上就是我们的第二个需求，也即输入到非正数就退出，我们可以使用 break 语句，把程序写成这样

```
1  while (条件) {
2      scanf("%d", &a);
3      if (a <= 0) {
4          break;
5      }
6  }
```

如果这样，我们就不需要为循环设置条件了，因为我们的退出条件写在了循环内，所以代码就改成了

```
1  while (1) {
2      scanf("%d", &a);
3      if (a <= 0) {
4          break;
5      }
6  }
```

条件是 1，意味着永远成立，因此这个循环如果内层没有退出条件的话，它会永远执行下去（也就是所谓的死循环）。第二个问题的解决其实很简单，反正我们最终的目的是要求和，我们可以这样写

```
1  int sum = 0;
2  int a;
3
4  while (1) {
5      scanf("%d", &a);
6      if (a <= 0) {
7          break;
8      }
9      sum += a;
10 }
```

什么意思？我们每次输入的数字，都会被加到这个变量 sum 中，这样，我们最终得到的就是所有输入的数字的和了。为什么这行求和要写到 if 语句后，而不是写成一个 else 语句呢？也就是这样

```
1  while (1) {
2      scanf("%d", &a);
3      if (a <= 0) {
4          break;
5      } else {
6          sum += a;
7      }
```

```
8      }
```

这样当然也是可以的，它意味着：如果进入 if 的下属语句，就不会进入 else 的，反之如果进入 else 的下属语句就进入不了 if 的，这符合我们的需求。但是我们发现，在修改前的代码，假如进入 if 的下属语句就会退出循环，也执行不到求和的那一行，假如进入了求和的一行，就说明一定没有进入 if，因此效果是一样的。

像上面这样，如果添加 else 语句与否的执行效果相同，我们建议不要使用 else，这样就减少了嵌套的数量，增加了代码的可读性。因为，我们如果阅读的是这样一段代码

```
1      if (a <= 0) {  
2          break;  
3      } else {  
4          sum += a;  
5      }
```

我们在阅读 else 下的代码时就要时刻惦记着：这是 if 语句的另一个情况，它代表了 a 是大于零的。我们每次都要想着 if 语句的条件才能对 else 的代码有正确的理解。但是如果写成这样

```
1      if (a <= 0) {  
2          break;  
3      }  
4      sum += a;
```

我们就可以理解为：a 小于等于零的情况在前面已经处理过了，我们断定下面的代码中 a 是大于零的，这段 if 语句就像是一个卫兵，向我们保证后面代码中 a 的取值范围，因此这样的，在核心业务逻辑代码前用来处理异常情况的代码被称做卫语句。这样就减少了阅读时的负担，这对于修改代码和理解代码都非常重要。实际上，我们一般推荐嵌套不要超过三层，我们在后期会讲到如何减少嵌套的数量。

如果不这么写呢？我们可以通过使用 while 的条件来控制循环的退出。即代码改成

```
1      while (scanf("%d", &a), a > 0) {  
2          sum += a;  
3      }
```

这是什么意思？这段代码的核心是 while 括号内的那一行，这里涉及到了一个叫做逗号运算符的知识，即，在作为流程控制语句条件的语句里，我们可以用逗号连接几个语句，这几个语句依次执行，但是只有最后一个作为条件判断。就拿我们这段代码来说，我们使用逗号连接了 scanf 和 a 的大小判断。首先执行 scanf，将输入的值赋给 a，然后执行大小判断，因为这是最后一个语句，它的判断结果就作为 while 的条件判断了。逗号运算符是 c 语言一个很实用的知识，可以在不失去阅读性的情况下简化我们的代码。当然，我们一般不会在这里连接多个语句，一般就是像上面这样连接一个，否则代码过于臃肿。

现在我们有了所有数的总和，接下来要求平均值，这就需要我们知道具体有几个数，所以我们应该定义一个变量用来储存已经求了几个数的和。那么它应该写在哪里呢？我们记录的数的个数，是用来求和的数的个数，因此应该和求和写在一块，也就是

```
1      int sum = 0;  
2      int count = 0;  
3      int a;
```

```

4
5 while (1) {
6     scanf("%d", &a);
7     if (a <= 0) {
8         break;
9     }
10    sum += a;
11    ++count;
12 }

```

或者使用逗号运算符的写法

```

1 int sum = 0;
2 int count = 0;
3 int a;
4
5 while (scanf("%d", &a), a > 0) {
6     sum += a;
7     ++count;
8 }

```

最后就要计算并打印平均数了。别忘了，整数的平均值可以是小数，所以我们要定义一个浮点型来储存结果。另外，要先把 sum 转成浮点型再计算，这在之前就已经讲过原因了。因此，我们最终的程序是

```

1 #include <stdio.h>
2
3 int main() {
4     int sum = 0;
5     int a;
6     int count = 0;
7
8     while (scanf("%d", &a), a > 0) {
9         sum += a;
10        ++count;
11    }
12
13    double re = (double) sum / count;
14    printf("%lf\n", re);
15
16    return 0;
17 }

```

当然了，如果是我们自己使用，那么可以增添一些提示性的文本，比如

```

1 #include <stdio.h>
2
3 int main() {

```

```

4      int sum = 0;
5      int a;
6      int count = 0;
7
8      printf("请输入连续若干个正整数,每输入一个就按一次回车。输入非正整数表示输入完成\n");
9
10     while (scanf("%d", &a), a > 0) {
11         sum += a;
12         ++count;
13     }
14
15     double re = (double) sum / count;
16     printf("以上数字的平均值是:%lf\n", re);
17
18     return 0;
19 }

```

其实 c 语言提供了一种类似于 while 的语句,叫做 do while, 比如

```

1      do {
2          printf("hi\n");
3      } while (a > 5);

```

与 while

```

1      while (a > 5) {
2          printf("hi\n");
3      }

```

区别在于, do while 要先执行一次 do 内的语句,然后再根据 while 判断是否循环。也就是说, do while 会至少执行一次循环内的语句,而 while 不一定。

我们在使用 while 语句解决一些需要重复若干次循环的问题时,会发现它不太方便,比如我们想要打印 10 次 Hello World

```

1      int i = 1;
2      while (i <= 10) {
3          printf("Hello World\n");
4          ++i;
5      }

```

我们发现,为了循环计数,我们在外层定义了一个变量,还要在循环里时刻想着别忘了它的自增。这样非常麻烦,因此 c 语言提供了 for 语句,它的语法是 for (初始条件; 终止条件; 赋值语句) 下属语句。比如上面的例子,我们可以这样使用 for 语句实现

```

1      for (int i = 0; i < 10; ++i) {
2          printf("Hello World\n");
3      }

```



这是什么意思呢？在执行 for 语句的时候，首先执行初始条件，创建了一个叫做 i 的变量，随后执行终止条件，发现满足，因此执行下属语句。下属语句执行完了，就会执行赋值语句，随后再执行终止条件。如果仍满足条件就继续执行下属语句，然后再执行赋值语句，如此一直循环，直到不满足终止条件为止，此时退出循环。

在我们的示例中，我们首先创建了变量 i，然后到了终止条件，发现满足 i 小于 10，因此执行下属语句，打印了一次，随后执行赋值语句，i 变成了 1，一直这样执行，直到 i 变成了 10，此时不满足终止条件了，退出循环。我们从 0 一直执行到 9，执行了 10 次。

那么，为什么不像 while 的例子那样，写成这样

```
1  for (int i = 1; i <= 10; ++i) {
2      printf("Hello World\n");
3  }
```

从 1 开始，这两段代码效果相同，但是后者不是更好理解吗？其实不是，计算机世界往往从 0 开始计数（学习指针时会解释为什么），我们这样写，还节省了一个等号，这样代码看上去反而更简洁有效。毕竟，区间 [1, 10] 内的整数个数就等于 [0, 9] 的，因此也等于 [0, 10) 的。

就是说，如果我们想要重复 n 次，那么代码就写成

```
1  for (int i = 0; i < n; ++i)
```

这样的话，i 就相当于一个计数器一样。值得注意的是，定义的临时变量只在 for 循环内有效，即

```
1  for (int i = 0; i < n; ++i) {
2      // 这里可以使用 i
3  }
4  // 这里不能用 i
```

在 for 循环的下属语句内，我们也可以改变 i 的值。另外 for 也可以使用 continue 和 break，比如我们在 while 写的输出 1 到 100 内偶数的例子，使用 for 可以改成

```
1  #include <stdio.h>
2
3  int main() {
4      for (int i = 1; i <= 100; ++i) {
5          if (i % 2 != 0) {
6              continue;
7          }
8          printf("%d\n", i);
9      }
10
11     return 0;
12 }
```

为什么 i 又赋成 1 了？我们要灵活变通，i 在这里不只是计数器，它还要被输出，因此这样写更好。

有意思的是，for 的初始条件、终止条件和赋值语句都可以不写，而是使用别的地方的代码替代。比如，如果我们使用的是定义在别的地方的变量作为循环的变量，那么就可以这样写

```

1  int a = 0;
2  for (; a < n; ++a) {}

```

也即, 我们空下了初始条件部分。如果我们退出循环是通过下属语句的 `break`, 那么 `for` 循环就可以写成

```

1  for (int a = 0;; ++a) {
2      if (a >= n) {
3          break;
4      }
5  }

```

再比如, 如果我们对 `a` 的赋值在下属语句, 那么可以不写赋值语句

```

1  for (int a = 0; a < n;) {
2      ++a;
3  }

```

总之, 不需要哪部分就不写哪部分即可, 但是不要忘记使用分号隔开各个部分。当然了, 我们还可以同时不写多个部分, 这都取决于具体情况。另外, 有时我们会见到这样的代码

```

1  for (;;) {}

```

这就是一个死循环, 等价于

```

1  while (1) {}

```

我们在引入 `for` 的时候就解释了它相对于 `while` 的优势, 那么 `while` 的优势是什么? 我们发现, `for` 语句更像是为了确定循环次数的情况而设计的, 在不确定循环次数的情况下, `while` 更加简单。这二者都很重要, 在实际开发时要经过思考后选用合适的循环语句。

## 5 流程控制 复杂实例

### 5.1 变量作用域

考虑一段这样的程序

```

1  #include <stdio.h>
2
3  int main() {
4      int a = 1;
5      if (a) {
6          int b = 1;
7          if (b) {
8              int c = 2;
9              printf("%d\n", c);
10         } else {
11             int c = 3;

```

```

12         printf("%d\n", c);
13     }
14 }
15
16     return 0;
17 }

```

运行结果是 2，如果我们把 b 的值改成 0，那么运行结果就是 3。我们之前说过，变量不可以重复定义，可是这里命名重复定义了变量 c，为什么这段代码可以正常执行呢？

我们把上面的代码抽象一下，提取出核心

```

1  {
2      int c = 2;
3      printf("%d\n", c);
4  }
5  {
6      int c = 3;
7      printf("%d\n", c);
8  }

```

我们发现，实际上两个 c 定义在了两个并列的花括号内。在 c 语言中，变量会在定义变量一级的花括号结束时被销毁。比如上面的代码，第一个变量 c 定义在第一个花括号内，因此当第一个花括号结束时，它就被销毁了，因此在第二个花括号中再定义一个 c 是不冲突的。

再考虑这种情况

```

1  {
2      int a;
3      {
4          int a;
5      }
6  }

```

这样就不行了。因为定义变量 a 的一级的花括号在代码的最后才结束，当我们第二次定义 a 的时候，之前的 a 还没有被销毁，所以这段代码是有错误的。

由于在第二级括号的时候 a 还没有被销毁，因此我们还是可以使用它的

```

1  {
2      int a = 1;
3      {
4          printf("%d\n", a);
5      }
6  }

```

但是内层的变量在外层就不能用了，比如

```

1  {
2      {

```

```
3     int a = 1;
4     }
5     printf("%d\n", a);
6 }
```

这是不行的，因为 a 在内层花括号结束时就被销毁了。

## 5.2 跳转语句

c 语言提供了一种叫做 goto 的语句，用来进行跳转。要想使用 goto，我们需要先定义一个标签，比如

```
1  #include <stdio.h>
2
3  int main() {
4      int n = 1;
5      label:
6          printf("%d\n", n);
7          ++n;
8
9      return 0;
10 }
```

然后使用 goto label 就可以跳转到标签的位置了，比如将上面的代码扩充为

```
1  #include <stdio.h>
2
3  int main() {
4      int n = 1;
5      label:
6          printf("%d\n", n);
7          ++n;
8          if (n < 10) {
9              goto label;
10         }
11
12     return 0;
13 }
```

这段代码会输出 1 到 9。这是因为当 n 小于 10，进入 if 语句的时候，会执行下属的 goto 语句，跳转到 label 所在的地方，又重新开始向下执行代码。

笔者强烈建议永远不要使用 goto 语句，因为它会导致代码的可读性严重降低，产生难以捕获的 bug。并且所有的 label 都只能在同一个函数中跳转，局限性大（非 local 的跳转需要使用 setjump.h，非常复杂）。goto 语句完全可以被使用恰当的 break 替代，因此不要使用 goto。

### 5.3 循环语句的复杂嵌套

考虑这个需求：需要输入行数和列数，打印对应行数和列数的一个由 1 组成的矩形，比如输入 2 3，打印

```
11
11
11
```

怎么实现？我们可以考虑使用一个循环来控制行数，在每一行中用另一个循环来打印对应个数的 1 和换行符。因此我们需要两个计数器，程序首先写成了

```
1  #include <stdio.h>
2
3  int main() {
4      int row;
5      int col;
6      scanf("%d %d", &row, &col);
7
8      int i = 0;
9      int j = 0;
10     while (i < row) {
11         while (j < col) {
12             ++j;
13         }
14         ++i;
15     }
16
17     return 0;
18 }
```

外层循环控制的是循环几次，即输出几行，内层则控制一行输出多少个 1。因此，内层循环的时候要输出 1，外层则要在每次循环结束的时候输出一个换行符。所以代码应该写成

```
1  int i = 0;
2  int j = 0;
3  while (i < row) {
4      while (j < col) {
5          printf("1");
6          ++j;
7      }
8      printf("\n");
9      ++i;
10 }
```

我们试着运行一下，比如输入 3 4，发现运行结果是这样的

```
1111
```

为什么后面没有 1 而是只有换行? 观察我们的代码, 发现在外层第一次循环结束后, `j` 的值等于 `row`, 因此在外层进入第二次循环的时候, 不会进入内层的循环。为了解决这个问题, 我们需要在外面每次循环结束的时候重置 `j` 的值, 即将代码改成

```
1  int i = 0;
2  int j = 0;
3  while (i < row) {
4      while (j < col) {
5          printf("1");
6          ++j;
7      }
8      printf("\n");
9      j = 0;
10     ++i;
11 }
```

这次再次运行结果就对了。

使用 `while` 写太麻烦了, 每次都要考虑定义临时变量和变量的自增与重置, 我们可以使用 `for` 语句重写

```
1  #include <stdio.h>
2
3  int main() {
4      int row;
5      int col;
6      scanf("%d %d", &row, &col);
7
8      for (int i = 0; i < row; ++i) {
9          for (int j = 0; j < col; ++j) {
10             printf("1");
11         }
12         printf("\n");
13     }
14
15     return 0;
16 }
```