第三章　变量、函数和控制结构

学了第二章介绍的内容，我们还只能描述从基本数据出发的简单计算过程，写出的程序描述的也仅是某个特定计算，能算出（输出）一个具体结果。如果需要对其他数据做同样计算，也必须改写原程序，在表达式里改用新的数据，经过重新编译连接，得到新的可执行程序后才能做。这样直接修改程序很不方便，而且还可能改错。

此外，即使程序里有同样的计算，我们也需要把同样程序片段重复写几遍。例如，在前章最后已知三边求三角形面积的程序里，就包含了几个同样的子表达式。这种重复使程序无必要地变复杂了。如果需要程序解决的问题更大更复杂，设法使每个计算片段只描述一次也会变得越来越重要。下面的许多讨论都与此有关。

在前一章里，读者也看到一些有意思的例子。C标准库里提供了一些数学函数，每个数学函数实现一种常用计算。令人感兴趣的是，我们可以方便地将这种函数用于不同数据，得到相应的结果。如果能在程序中定义自己所需的函数，定义后又能像标准库的数学函数那样使用，那么就从一个角度解决了上面提出的某些问题。

本章将讨论 C 语言的一些重要编程机制，介绍如何使用它们去做程序设计。其中的主要问题包括：变量的概念和使用，函数的定义和使用，描述计算流程的若干基本控制结构。读者将看到一些程序实例，还能看到一些关于从问题出发进行程序设计的分析过程的讨论。人们还希望完成的程序更具通用性，能方便地对不同数据实现同样计算。如何为程序提供计算所用的数据是另一个问题，将在后面章节里讨论。

3.1　语句、复合结构

C程序中描述计算过程的基本单位是语句。一个语句是由分号结束的一段字符，例如第一章给出的简单C程序里就包括了下面语句：

```
printf("Good morning!\n");
```

语句必须在形式上符合要求，否则就是非法的。这是语法问题，编译程序能检查出程序里的语法错误。另一方面，每个形式合法的语句都表达了一种含义，表示在程序执行中要做的一个动作，这称为语句的*语义*。上面语句表述了对标准库输出函数 `printf` 的一次调用，程序执行到达该语句时，函数 `printf` 将被执行，它向计算机系统的标准输出送去一系列字符，这些字符通常将显示在计算机屏幕上。C语言里还有许多语句形式，我们将逐渐接触和熟悉它们。

仅有基本语句是不够的。为了描述复杂的计算，还需要一些能把语句组合起来的结构，以实现一系列语句的执行，实现对语句执行过程的控制。C语言里描述计算流程的一种最基本结构是*复合结构*（也称*复合语句*），它实现基本的顺序执行。复合结构的形式就是一对花括号，在括号间可以有多个语句。在复合结构执行时，列在其中的各个语句将顺序执行，直到最后一个语句执行完毕，该复合结构就执行完了，这就是复合结构的语义。允许写不含任何语句的复合结构（空复合结构），执行时它什么也不做，立即结束。

在本书给出的第一个简单C程序里，程序的主要部分是：

```
int main () {
    printf("Good morning!\n");
    return 0;
}
```

这里主要就是一个复合结构，其中包含了两个语句。

根据复合结构的语义和 `printf` 所完成的输出动作，下面程序将产生与第一章的简单

程序相同的输出：

```
#include <stdio.h>

int main () {
    printf("Good ");
    printf("morning!");
    printf("\n");
    return 0;
}
```

这个程序执行时，复合结构里的三个输出语句顺序执行，产生的输出正好与前面程序一样。

> 复合结构实现程序中的顺序控制，一个操作完成后执行下一个操作。这种执行方式对应于计算机硬件中指令执行的最基本方式：一条指令执行完毕之后执行下一条指令。实现顺序控制的硬件基础是计算机 CPU 里的指令计数器。

3.2　变量——概念、定义和使用

*程序变量*简称为*变量*。程序变量是表述数据存储的基本概念，是 C 语言以及各种常规程序设计语言*中的一个重要概念。读者应该知道，在计算机硬件层，程序运行中的数据存储靠内存储器、存储单元、存储地址等一系列机制实现，这些机制在程序语言层的反映就是程序变量的概念。还请读者注意，程序变量与数学中的变量是完全不同的概念。

一个程序变量可以看作一个容器，程序运行中可以将有关的数据存入变量中。程序里的每个变量都有一个名字，在程序中可以通过名字使用相应的变量，进而使用存储在这个变量里的数据。

对变量的基本操作有两个：

1. 将数据值存入变量中。这个操作称作给变量*赋值*。程序语言对于怎样给一个变量赋值，能赋什么值往往有一些限制，具体语言常有具体的规定。
2. 取得变量里当时保存的值，以便在计算过程中使用。这个操作称为“取值”。

变量具有保持值的性质（有的语言里存在不遵守这种性质的变量，现在不讨论这种特殊情况）。也就是说：如果在某时刻给某变量赋了一个值，此后用该变量的值时，每次得到的总是那个值。这种情况一直延续到下次再给这个变量赋值为止。问题也有另一面，由于赋值操作的存在，在程序执行中，一个变量在各个时刻所保存的值可能不同。也就是说，变量的值可以变，这与数学中的变量完全不同。

C 程序中的每个变量都有固定的类型。所谓变量“有固定类型”，指的是每个变量只能保存一种类型的值。例如，可以有只能保存 `int` 值的变量，也可以有只能保存 `double` 值的变量。在讨论程序设计问题中说到变量时，常常要提出其类型。例如人们常说某变量是个整型变量（是 `int` 类型的，只能保存 `int` 值的变量），或是双精度变量（只能保存 `double` 类型的值），或者是字符变量等等。

变量名字是标识符，实际上，C 语言里所有的名字都用标识符表示。

3.2.1 变量的定义

C 语言要求程序里所用的每个变量都必须先定义，然后才能使用。定义变量需要提供两方面信息：变量名和变量类型。定义变量的语言结构称为*变量定义*，变量定义在形式上是依次列出所要求的类型和被定义变量的名字，最后写一个分号（这使得变量定义具有与语句相

* 实际上，确实存在着一些“非常规的”程序设计语言，在那些语言里没有变量的概念。如果读者进一步在计算机科学技术领域中学习，将来也可能接触到它们。

同的形式）。例如下面写出的是两个变量定义：

```
int m;
double x;
```

其中第一行定义了一个整型变量 m，第二行定义了一个双精度变量 x。在一个变量定义里也容许同时定义多个类型相同的变量，形式上是用逗号把这些变量名分隔开。例如，下面是两个变量定义：

```
int k, n, sum, count;
long double y, z;
```

这两行代码定义了四个整型变量和两个长双精度类型变量。

C 语言是自由格式语言，因此也允许把多个变量定义写在同一行。为了程序的清晰性，人们一般不采用这种写法。如果同时定义的变量很多，也可以将一个定义描述延续到几行。除了不能用关键字做变量名外，写程序时可以用任何标识符作为变量名。

人们提倡用能说明变量用途的有意义的名字为变量命名，因为这种名字对写程序和读程序的人都有提示作用，有助于提高程序的可读性。尤其是在程序大、变量多时，这种做法就加更重要。数学里常用简单方式为变量命名，那是因为数学公式中变量很少。程序的情况则不同，大程序可能有成百成千，甚至数以万计的变量，因此命名问题就很重要了。应提醒读者，读自己程序的人首先是自己，不注意变量命名问题，吃苦头也是自己。

在 C 程序中任何复合结构里都可以定义变量。在一个复合结构里定义的变量可以在该复合结构的内部使用。C 语言还规定：变量定义必须出现在同一复合结构里的所有语句之前，不能与语句交替出现。也就是说，复合结构一般具有下面的形式：

{变量定义序列　　语句序列}

变量定义或语句都可以没有。复合结构执行时首先一个个处理变量定义，随后再顺序执行各个语句。

变量的存储实现：C 程序的变量功能怎样实现？当一个变量被定义时，系统将为它确定一个存储值的位置，这个存储位置所占存储器单元的多少由变量的类型确定，也就是说，由变量所要存储的数据值的大小决定。例如，在许多系统里，一个双精度类型变量常被分配一块 8 个字节大小的存储位置，其中正好能放下一个双精度值。一个字符类型的变量通常分配一个 1 字节的存储位置。其他类型也一样，由编译系统根据数据类型的大小为变量分配存储。有时不同系统也可能采用不同分配方式。例如在一些微机 C 语言系统里整型变量被分配 2 个字节的存储位置，在另一些系统里的 int 变量被分配 4 个字节。当程序中给变量赋值的时候，这个值就被存入变量所占的存储单元。当需要使用某个变量的值时，就从相应存储位置中取出值来用。

3.2.2 变量的使用：取值与赋值

使用变量的方式就是直接将它写在表达式里。如果在计算表达式时遇到变量，就会取出该变量当时的值参与计算。显然，如果某个表达式里包含有变量，那么它的计算结果就会依赖于有关变量的值。由于变量的值可能变化，同一表达式在不同时刻就可能求出不同的值，这与上一章那种仅包含基本数据值的表达式有本质区别。例如，下面是一个包含了对两个变量使用的表达式：

```
x + sin(3.2 * y) - pow(x, 2)
```

它的值就要依赖于求值时刻变量 x 和 y 的值。

赋值运算符和赋值表达式

赋值操作由*赋值运算符*描述。C 语言用等于符号“=”作为赋值运算符，赋值运算符常简称为*赋值号*。用赋值号可以构造赋值表达式，计算这种表达式的主要效果就是给指定变量

赋一个新值。例如，下面是一个赋值表达式：

```
x = 5.0
```

假设变量 x 为 double 类型，在计算上面表达式时，双精度数值 5.0 将被赋给这个变量。在此以后，变量 x 就保存着数值 5.0（无论它原来的值是什么）。

一般说，赋值运算符左边的运算对象应该是一个变量的描述，最简单的情况就是一个变量名（后面会看到其他情况）；右边运算对象是任意的表达式。左边的变量是赋值操作的目标，赋值运算的主要效果就是求出其右边表达式的值，赋给左边的那个变量。

赋值运算符的优先级很低，低于所有算术运算符。所以表达式：

```
x = 2 + 3 * y
```

将求出赋值号右边算术表达式之和，而后将这个和赋给变量 x。如果 y 的值为 5，那么 x 将被赋值 17。

赋值语句

在赋值表达式后面写一个分号，就构成了一个*赋值语句*。赋值语句是程序里最基本的语句，代表着程序中最重要的一种操作。C 程序中通常用赋值语句的形式给变量赋值。后面会看到，赋值表达式本身也可以独立存在，这是 C 语言里的特殊情况。

变量在程序里的一个主要用途是用于保存计算的中间结果。看一个熟悉的例子。

例：已知三角形的三条边分别是 3、5、7，求这个三角形的面积。

由于计算中需要反复使用由公式 $s=\frac{1}{2}(a+b+c)$ 确定的半周长值，可以考虑用一个变量保存这个值，后面计算中就可以直接通过该变量使用这个值。这样做不但程序写起来简单了，也避免了完全相同的重复计算。下面是写出的程序：

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

int main () {
    double s;
    s = (3 + 5 + 7) / 2.0;
    printf("Area: %f\n", sqrt(s * (s-3) * (s-5) * (s-7) ) );
    return 0;
}
```

使用变量的结果是使这个程序更简短，也更清楚了。

赋值运算符的值与结合性

赋值运算符的主要作用是给变量赋值。但是，由赋值运算符构造出的是赋值表达式，这也是一种表达式，作为表达式，其计算就应该得到一个值。C 语言规定赋值表达式的值就是赋值号右边的那个表达式的值，也就是赋给左边变量的那个值。举例来说，表达式（注意，这不是一个语句，因为它不包括结束语句的分号）：

```
s = (3 + 5 + 7) / 2.0
```

不仅能求出赋值号右边表达式的值 7.5，并将此值赋给变量 s，整个赋值表达式也将求出一个 double 值 7.5。通常人们并不关心也不使用赋值表达式的值，常常采用赋值语句的形式写赋值操作。这是允许的，也是最常见的。

但是，由于赋值表达式本身也能得到一个值，所以也可以将它写在其他表达式的里面。例如，下面的表达式计算中不但能完成给变量 x 赋值 5 的操作（括号里的赋值表达式完成这件事），最后还会求出一个值 13：

```
(x = 5) + 8
```

例如，我们可以在程序里写下面这样的语句：

```
y = (x = 5) + 8;
```

在这个语句中，括号里的赋值表达式给 x 赋值 5，而后将这个赋值表达式的值 5 与整数 8 相加，得到的结果 13 被用于给变量 y 赋值。虽然这种形式在 C 语言里是合法的，但其意义比较容易被误解，因此人们并不提倡使用这种形式的结构。

有时人们以一种特殊形式使用赋值表达式的值，这就是把一个赋值表达式放在另一个赋值符号的右边。这样做的效果就是用同一个表达式为多个变量赋值。例如，下面的赋值语句里就采用了这种方式：

```
y = (z = (x = 3 * s));
```

这就是用一个算术表达式求出的结果给三个变量赋了值。

C 语言规定，赋值运算符采用从右向左的结合顺序，因此上面语句可以简化为下面形式，其效果完全是一样的：

```
y = z = x = 3.5 * s;
```

这种赋值形式有时也被称为“多重赋值”。

赋值与类型

赋值操作中也有类型问题。被赋值的变量有自己的类型（由变量定义确定）；赋值号右边是个表达式，按照有关表达式计算的规定，其计算结果也有确定的类型。如果这两个类型相同，赋值自然可以顺利进行。两个类型不同就出现了新问题。

对赋值中表达式与被赋值变量类型不一致的情况，C 语言规定：在允许转换的情况下，表达式的值将自动转为被赋值变量的类型的值，然后再赋值（前面讲过，数值类型的值都可以互相转换。后面我们会看到不能转换的情况）。

如果我们把前面程序例子的赋值语句改写成：

```
s = (3 + 5 + 7) / 2;
```

编译后运行时就会发现程序的结果不对。显然这是一个合法的语句，右边表达式算出的值也将先转换为双精度类型的值，而后赋给变量 s。但是，为什么这样改动之后，这个程序就会出现语义错误？请读者自己设法弄清楚。

3.2.3 几个问题

变量定义时的初始化

在定义变量时，可以用类似赋值的写法给被定义变量指定初始值，这种描述方式及其效果称为变量的*初始化*。变量初始化的一些规定在后面讨论。首先，用简单数值或仅由数值构成的表达式对类型合适的变量进行初始化是容许的。在变量定义时进行初始化很方便，而且可以避免由于忘记给变量赋值就去使用的常见错误，因此人们提倡在采用定义变量时直接初始化的方式。下面是一些定义时为变量初始化的例子：

```
int n = 1, m = 1;
double s = (3 + 5 + 7)/2.0;
long double x = 4.5L;
```

定义时初始化不能采用多重赋值的那种形式，即使需要赋给多个变量的初值完全相同，也必须一个一个地分开写。

采用定义变量时初始化的方式，前面的示例程序可以改写为：

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

int main () {
    double s = (3 + 5 + 7) / 2.0;
    printf("Area: %f\n", sqrt(s * (s-3) * (s-5) * (s-7) ) );
    return 0;
}
```

变量的属性

现在已经可以对变量概念有一个更清楚的描述了。图 3.1 是对变量概念的图示。一个变量包含四方面属性：

1. 变量的名字，它提供了在程序里访问变量的基本途径；
2. 变量的类型，它规定了变量的可能使用方式，可能存储的值，可能使用的各种操作；
3. 变量的存储位置，这是变量在计算机里的具体实现；
4. 存储在这里的变量值。

在讨论与变量有关的各种问题时，常常会涉及到这几方面的问题。

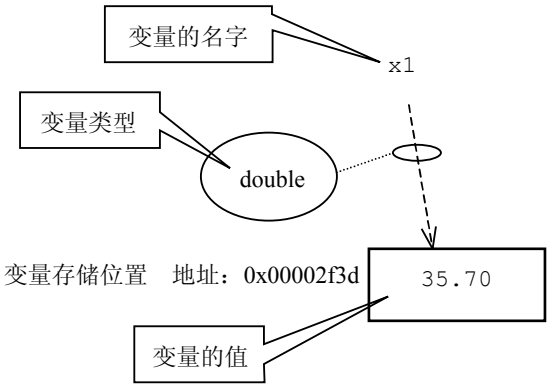

图 3.1　变量的构成

赋值与等于

应该看到，虽然 C 语言采用等于符号作为赋值符号，程序中的赋值与数学中的等于关系是完全不同的两个概念。举一个典型的例子，在数学里等式 $x = x + 1$ 是一个矛盾式，因为没有任何值能够满足这个式子。而在程序里，

```
x = x + 1;
```

是一个很常见的语句，其意义非常清楚，就是取出变量 x 当时的值与 1 相加，然后把得到的结果再赋给变量 x。这个语句的执行效果就是使变量 x 的值增加了 1。这不仅是合法语句，也是程序中经常要做的事情。

对求值顺序敏感的表达式

还有一个值得提出的重要问题。由于赋值是 C 语言里的一种运算，所以我们可能写出下面的语句（假定其中变量已定义，都是 double 类型）：

```
x = 2.0;
y = (x = 3.0) + x;
```

现在想讨论的问题是，在这两个语句执行之后，变量 y 的值究竟是什么？

实际上，这个问题我们无法回答，因为上面的第二个语句是错误的。在执行了第一个语句以后，x 的值是 2.0。对第二个语句的执行要求计算赋值号右边的表达式。由于有了括号所确定的计算顺序，加法运算应当最后做，现在的问题是，被加的两个值到底是什么呢？C 语言没有给出回答，因为赋值号右边表达式的计算结果依赖于加法运算两个运算对象的计算顺序。从道理上说有两种可能情况：

1. 左边运算对象先算，得到值是 3.0，而且在这个计算中变量 x 的值被改变了；随后计算加运算符右边的 x 时也得到值 3.0。这时做加法，计算的结果应当是 6.0；
2. 先计算右边得到 x 的值 2.0，加上左边计算出的 3.0，得到的最后结果是 5.0。

这个分析告诉我们，如果上面语句能执行，y 最后的值也要依赖于加法运算对两运算对象求值的次序。C 语言标准并没有规定加法运算对两个运算对象的求值顺序，它只说在程序中不应该写出这样的表达式，这种表达式可能的结果没有定义。

这个例子是对前面讨论表达式计算方式时的一个遗留问题的回答。本例说明，在 C 语言里确实可以写出依赖特定计算次序的表达式。为了保证我们所写的程序确实能完全我们所希望的计算工作，就要求它们不应该依赖某种特定计算顺序。即使你看到的某本书说某个系统在这方面的情况如何如何，也不要依靠这种说法，因为那样写出的程序将依赖于具体系统，因此不是正确的写程序方式。即使是同一个 C 语言系统，在它的新版本中也完全可能采用其他求值顺序（根据系统开发者的需要）。如果我们的程序具有这种依赖性，在情况变化时

就会遇到许多麻烦。

程序中的注释

为帮助人（包括自己）读程序、理解程序意义，有时我们希望能在程序里写一些说明性文字。这种文字不应对程序的意义（程序的执行）有任何影响，只是希望对人读程序、理解程序起一点帮助作用。程序里具有这种性质的信息称为注释。C 语言的注释形式是：

```
/* 任何字符的序列 */
```

两个组合符号 /* 和 */（组合符号必须连续写，两个字符之间不能有空白字符）起一对括号的作用，被它们括起的是注释内容，其中可以包含任何符号。在编译过程中，注释将被简单丢掉，因此不会影响程序的意义。C 程序里的注释相当于一个空格。在程序中适当地方加入必要的注释是一种良好的编程习惯。对于复杂的大程序，注释的作用更加明显。

简单赋值程序

理解了赋值语句，已经可以更方便地写出许多程序了。只使用到目前为止所学的语言机制，写出的程序大致具有下面的样子：

```
#include <stdio.h>
/* 如果需要用数学函数，这里还要写#include <math.h> */

int main () {
    /* 若干变量定义（以及初始化） */
    /* 若干计算和赋值语句 */
    /* 若干输出语句 */
    return 0;
}
```

许多简单程序都可以按这种模式写出来。

3.3　定义函数（初步）

前一章介绍了如何使用库函数，包括输出函数 printf 和各种数学函数。这些函数可看作 C 语言基本功能的扩充。函数是特定计算过程的抽象，具有一定通用性，可以按规定方式（参数数目、类型等都可以看作是规定）对具体数据使用。对一个（或一组）具体数据，函数执行可以计算出一个结果，这个结果可以在后续计算中使用。

如何看待函数的调用？图 3.2 是一个形象的图示，其中的 *t* 表示时间。当主程序执行函数调用时，它自己的执行暂时中断，执行控制权转到被调用函数，使该函数开始执行。直到函数执行完毕，函数返回使执行控制权回到主程序，主程序才从中断点之后继续下去。

调用其他函数的代码部分称为“主程序”。也有人将两者分别称为“主调函数”和“被调函数”。易见，主程序和被调函数的关系是相对的，因为一个函数里还可能调用另一函数，使调用形成一种层次性。读者应理解在函数调用过程中程序执行的控制转移关系。

实际程序设计中确实有许多对特定函数的需求，这里举一个简单的例子：

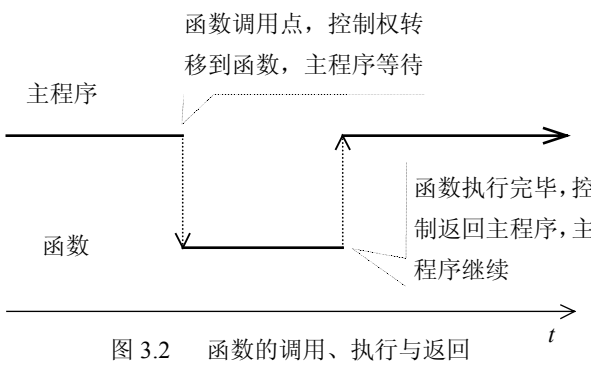

图 3.2　函数的调用、执行与返回

例：假设有一批圆盘，需要求它们的面积。这些圆盘的半径分别是

3.24、2.137、0.865、3.746、12.64、8.421、0.307、7.258

假设圆周率用3.14。依前面程序的样子，我们可能写出下面的程序：

```
#include <stdio.h>
int main () {
    printf("area1 = %f\n", 3.24 * 3.24 * 2.14);
    printf("area2 = %f\n", 2.137 * 2.137 * 3.14);
    printf("area2 = %f\n", 0.885 * 0.865 * 3.14);
    printf("area2 = %f\n", 3.746 * 3.746 * 3.14);
    printf("area2 = %f\n", 12.64 * 12.54 * 3.14);
    printf("area2 = %f\n", 8.421 * 8.421 * 3.14);
    printf("area2 = %f\n", 0.307 * 0.207 * 3.14);
    printf("area2 = %f\n", 7.268 * 7.268 * 3.14);
    return 0;
}
```

这个程序里有许多类似的东西。不仅对各项数据的计算过程相同，同一个数据还要重复出现。这样的程序写起来很讨厌，容易使人看花眼，因此很容易出现错误（例如，上面程序里就有几个错误，请读者设法将它们找出来）。如果函数库里有一个以半径为参数，计算圆盘面积的函数，其类型特征是：

```
double c_area(double)
```

上面程序中的主要部分就可以简化。例如，其中第一个语句可以写为：

```
printf("area1 = %f\n", c_area(3.24));
```

其他语句的情况也类似。如果函数库有以半径为参数，打印圆盘面积的专用函数 `pc_area`，上述程序还可以进一步简化，主要部分只需写下面这样形式的几个语句就够了：

```
pc_area(3.24);
pc_area(2.137);
```

如果有这些函数，程序不但能变短，也会变得更清晰。可惜的是 C 语言函数库里没有这些函数。实际上，由于写程序中的需要是丰富多彩的，任何函数库都不可能提供可能有人需要的所有函数，无法保证程序里需要的东西都能在标准库里找到。C语言提供的解决方式是允许写程序的人自己定义所需函数。

上面这个例子很小，但也说明了编程序的人有自己定义函数的需要。

3.3.1 函数定义

我们可以把一段计算定义成一个函数，并给它取一个名字。有了这样的函数定义后，在程序里需要用这段计算的地方，就可以通过函数的名字调用这个函数，实施有关计算了。下面是定义函数 `c_area` 的程序片段：

```
double c_area (double r) {
    return r * r * 3.14;
}
```

在程序里写了这个定义后，`c_area` 就可以像标准库的各种函数一样使用了。

作为例子，我们先看看如何利用函数定义重写前面提出的程序。要使用自己定义的函数，必须把函数定义的代码段包含在整个程序里，这样的一段代码称为一个“函数定义”。在程序里有了某个函数的定义后，该函数就可以使用了。下面是修改后的完整程序，可以看到其中的语句形式正是我们所希望的：

```
#include <stdio.h>

double c_area (double r) {
    return r * r * 3.14;
}

int main () {
    printf("area1 = %f\n", c_area(3.24));
    printf("area2 = %f\n", c_area(2.137));
```

```
    printf("area3 = %f\n", c_area(0.865));
    printf("area4 = %f\n", c_area(3.746));
    printf("area5 = %f\n", c_area(12.64));
    printf("area6 = %f\n", c_area(8.421));
    printf("area6 = %f\n", c_area(0.307));
    printf("area6 = %f\n", c_area(7.258));
    return 0;
}
```

这个程序包括两个主要部分：前面是函数 c_area 的定义，后面是我们熟悉的 main 部分，其中的语句里调用了前面定义的函数。本书后面的程序大多具有这种形式。

定义好的函数可以在程序里任何需要它的地方使用。上面程序里的 c_area(3.24) 是个函数调用表达式，表示用实参 3.24 去调用函数 c_area。这种表达式也可用于构造更复杂的表达式，形成更复杂的计算，就像用标准数学函数构造复杂表达式一样。例如，假定程序里要计算高 2.4，半径 3.24 的圆锥的体积。有了 c_area 后，表达式就可以写为：

```
2.4 * c_area(3.24) / 3.0
```

将 c_area 定义为函数带来了许多优越性。例如，函数调用一目了然，参数也只需写一次，免去了前面那样的“笔误”。另外，有关计算过程的描述被归纳到一个地方，如果这一计算过程需要修改，只要在一个地方修改就够了。举例来说，如果我们觉得原来选用的圆周率精度不够，希望将它改为 3.14159265，那么就只需要函数的定义：

```
double c_area (double r) {
    return r * r * 3.1459265;
}
```

如果修改原来的程序，事情就麻烦多了。当然，这还是很小的程序，修改再多也没有多少。对于那些数以十万行百万行计的大系统，这种易修改性质的意义就更大了。

还有一点也很重要，定义好的一组有用函数可以作为我们进一步写程序的基本构件，满足其他程序设计的需要。例如上面定义的 c_area，其可能使用并不限于这个小程序，完全可以用到其他需要计算圆盘面积的地方。这一思想的发展就是函数库（C 语言的标准库也是这一想法的一个规范化），由此引出许多关于重复使用软件构件的研究，这方面的想法已经发展现代软件设计实现的一个重要领域。

函数定义的形式

函数定义也有规定的语法形式。一个函数定义包括两部分：函数头部和函数体。

函数头部说明了函数的名字和类型特征，在形式上是顺序写出的几个部分：函数的计算结果类型，函数名字，随后是写在一对括号里表述该函数参数情况的说明。最后这部分也称为函数的参数表，其中说明了本函数要求几个参数，它们各是什么类型的。参数表里还要为每个参数取一个名字，以便在函数体中的表达式和语句里使用这些参数值。在函数 c_area 的定义中，函数头部是：

```
double c_area (double r)
```

这表示本函数的名字是 c_area，其返回值类型是 double；这个函数只有一个参数（参数表里只有一对类型描述和参数名 double r），参数的类型是 double，参数名是 r。

函数体是一个普通复合结构，其中可以包括一些变量定义，而后是一些语句。在函数体里定义的变量称为这个函数的*局部变量*，因为它们只能在本函数体内部使用。参数表里定义的参数也被看作局部变量，可以在函数体里像其他局部变量一样使用。当一个函数被调用时，其函数体的语句将在参数具有特定实参值的情况下开始执行。

函数体里常要用一个有特殊作用的语句：return 语句（返回语句）。return 语句有两种形式，上面所用的形式是：

```
return 表达式;
```

另一种形式在后面讨论。返回语句的基本作用是结束它所在函数的执行，在函数结束前，首先计算该语句中的表达式，由这个表达式求出的值作为本次函数调用的返回值。

例如，在 c_area 的函数体里只包含一个 return 语句。当 c_area 执行时，函数体里的 return 语句被执行，求出其中表达式的值作为返回值，同时 c_area 的执行结束。

定义具有多个参数的函数

要定义具有多个参数的函数，应该在参数表里给出有关各个参数的说明，每个参数都需要说明参数名及其类型。下面是一个例子。

例：定义一个由已知三边长度求三角形面积的函数。

首先考虑被定义函数的类型特征。我们把要定义的函数命名为 t_area（实际上，可以选用我们所喜欢的名字），它应当有三个双精度参数，返回双精度的值。这样，函数的类型特征应当是：

```
double t_area (double, double, double)
```

根据函数定义的形式和本函数所要求的计算过程，这个函数的定义可以写为：

```
double t_area (double a, double b, double c) {
    double s = (a + b + c)/2.0;
    return (sqrt(s * (s-a) * (s-b) * (s-c)));
}
```

这里采用了在定义变量时给初始值的方式，还通过引进变量保存中间值简化了函数定义。如果一个程序里定义了函数 t_area，程序最前面应包括下面的行，因为在这个函数里调用了标准库的数学函数：

```
#include <math.h>
```

下面是一个调用上述函数的语句：

```
x = t_area(3.5, 5.6, 6.8);
```

再如：

```
printf("Area: %f\n", t_area(13, 15, 19));
```

在这些调用里，各实际参数将按顺序送给函数里对应的各参数，参与函数体里的计算。读者很容易写一个程序，把上面函数定义放进去，并通过调用它做对三角形面积的计算。

定义没有返回值的函数

有时我们需要定义一些东西，希望它们为程序做一些事，但却不需要由它们得到返回值。例如，在我们想写的程序里可能需要一个完成某种特殊输出工作的函数，每调用一次，它就为我们输出一些信息。实际中我们还可能遇到许多不需要返回值的函数实例。C 语言对这类问题的解决办法是允许定义没有返回值的函数*。

C 语言规定，将关键字 void 放在函数头部写返回值类型的地方，表示这里定义的是一个不返回值的函数。下面定义前面所提出的以半径为参数，输出圆盘面积的函数：

```
void pc_area (double r) {
    printf("r = %f, S = %f\n", r, 3.14159265 * r * r);
}
```

有了这个函数之后，前面的程序可以写得更简短清晰了：

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

*在许多程序设计语言里，编程者定义的具有独立性、可调用程序段称为子程序，通常将子程序分为两类：执行最后产生返回值的称为函数，只完成一些操作不产生称为过程。并规定函数只能在表达式里使用，过程以过程调用语句的形式使用。C 语言在这个方面有些特殊，将子程序统称为函数。

```
void pc_area (double r) {
    printf("r = %f, S = %f\n", r, 3.14159265 * r * r);
}

int main () {
    pc_area(3.24);
    pc_area(2.137);
    pc_area(0.865);
    pc_area(3.746);
    pc_area(12.64);
    pc_area(8.421);
    pc_area(0.307);
    pc_area(7.258);
    return 0;
}
```

上面给出了完成同样工作的几个程序示例，从它们的对比中，读者应明显感受到函数的意义和作用，还应注意到适当选择所定义函数的价值。定义一组适当的函数，对于大程序的效果将更加明显。实际上，不会定义函数的人根本就不可能写出大的程序。

注意，函数 pc_area 的定义里没有 return 语句，执行完 printf 后就到了作为函数体的复合结构的末尾。在这种情况下函数也正常结束。当然，如果所定义的函数有返回值，就不应该出现这种情况，因为如果函数这样结束，函数的返回值就没有定义。此外，无返回值的函数显然不能放在表达式里使用。

3.3.2 函数和程序

前面给出的所有完整程序例子里都有一段是：

```
int main () {
    ……
    return 0;
}
```

前面一直没说这段代码是什么。实际上这就是定义了一个名为 main 的函数。在一个 C 程序里总需要有名字为 main 的函数，而且只能有一个。

以 main 为名的函数地位很特殊，它表示了一个程序执行的起点和整个过程。在一个 C 程序（被加工为可执行程序后）启动执行时，就从它的 main 函数的体开始执行，一个一个地执行其中语句，直到这个函数结束（语句执行完了或者退出了），整个程序的执行就完成了。因此人们也常把 main 函数称为（C 程序里的）*主函数*。

一个普通函数定义不能构成一个完整程序，只能作为程序中的一部分。函数定义的作用就是定义了这个函数所包含的计算过程，并为该函数确定了一个名字。只有在被调用时函数的体才会真正被执行，从而起作用。这似乎形成一种很奇怪的现象：每个函数都等着被调用，那么程序怎么能开始执行呢？这就是 main 函数的作用，这个函数是在程序开始执行时被自动调用的（它被 C 程序的运行系统调用）。

我们知道，一个函数定义之后可以在程序里多个不同地方调用，程序代码可以为每个调用提供一组不同的实际参数，从而形成不同的计算，得到不同计算结果。main 可以调用其他函数，一个函数还可以调用另一个函数。在这方面 main 函数的情况也特殊，在程序里不允许写对它的调用。main 函数返回值是 int 类型，用返回 0 表示程序正常结束。C 语言还规定，如果 main 函数没有执行 return 语言而结束，系统将自动产生一个表示程序正常结束的值（通常是 0）。

含有一个或多个函数定义的程序通常采用如下形式：

```
#include …
……   /* 函数定义写在这里（可以有一个或几个） */
```

```
int main () {
    …… /* 主程序体，这里通常包含对一些函数的调用 */
    return 0;
}
```

这可以看成是包含几个函数定义的完整程序的模式，请读者在写程序时采用这种形式。

由前面的讨论也可以看出，如果程序里定义的某个函数不被主函数调用（直接或间接被调用），在程序执行中它将不会参与执行，因而对程序完成的工作也不能有任何贡献。这样的函数定义可以删除，不会影响程序的意义。

按照 C 程序的要求和复合结构的规定，最短的而又是完整的程序是：

```
int main () {}
```

这个程序能正常编译连接和执行。由于主函数体里没有语句，函数一执行就结束，读者可以自己试试。读者在编程序时可能发现了一个有趣的问题：上述程序虽然什么都不做，编译连接后也会产生一个不小的可执行程序，其长度因编译系统而异，可能有若干 K 字节或者更多。请读者想一想，这里面的代码究竟是什么。

3.3.3 函数与类型

首先考虑函数定义。函数头部描述返回值类型，函数内部提供返回值需要用返回语句，其中的表达式给出了返回值的计算方法。按规定，任何表达式求出的值都有确定类型，这样，在返回表达式的类型和函数返回值类型之间就可能出现不一致的情况。

C 语言的规定是：返回表达式的类型必须能转换到函数头部定义的返回值类型。执行到返回语句时，其表达式求出的值先转换到函数头部所要求的类型，再将转换结果作为返回值。例如在下面函数定义里，return 语句求出表达式的值之后就需要做类型转换：

```
int fun (int m){
    return 3.14159 * m;
}
```

前面说过，函数调用时的实参个数必须与函数定义要求的参数个数一致，每个实参的类型也必须符合函数要求。要求每个实参应能转换到函数参数表规定的对应类型。当某个实参的类型与函数要求不一致时，首先转换该实参求出的值，而后将得到的结果送给函数。例如，对于上面的函数 fun，在下面调用中也要进行类型转换：

```
x = fun(3.09);
```

假设 x 是双精度变量，请读者分析：在这一语句执行中，哪些地方将发生类型转换，每次是从什么类型转换到什么类型（提示：一共发生了 4 次类型转换）。

3.4 关系表达式、逻辑表达式、条件表达式

只有前面学过的机制，我们写程序的能力还很弱。例如，假设现在要写一个函数

```
double dmax (double, double)
```

求出两个参数中较大的一个。粗粗分析就会发现，现在我们还不能解决这一问题。因为在这个函数中需要比较数据的大小，而后跟据比较结果决定怎样做。这种情况在程序中也很常见，实现这类计算需要更多的运算符和其他功能。

3.4.1 关系表达式和条件表达式

关系运算符用于确定两个数据之间是否存在着某种关系。利用关系运算符可以写出关系表达式，我们可以这种表达式的结果去控制计算的进程。

C 语言的关系运算符共有 6 个，它们是：

> >= <= <	大于、大于等于、小于等于、小于
== !=	等于、不等于

这些运算符可以对各种数值类型使用，通过关系运算符可以构成关系表达式。下面是两个简单的关系表达式，在它们的后面说明了所描述关系的成立与否：

```
3.24 <= 2.98      关系不成立
5 != 3 + 1        关系成立
```

如果进行关系运算的两个数据类型不同，那么就按照与算术运算符一样的规则，先进行数据转换，转到同类型的数据之后再做比较。

关系表达式的计算也应得到一个值。C 语言规定关系运算得到 int 类型的值：当关系成立时，关系表达式求出的值是 1；关系不成立时求出的值是 0。这样，3.24 <= 2.98 的值是 int 类型的 0，而表达式 5 != 3 + 1 的值是 int 类型的 1。

关系运算符的优先级低于算术运算符，高于赋值运算符。其中==和!=的优先级低于另外四个关系运算符。关系运算符也采用自左向右的结合方式，它们也没有明确规定参与比较的两个运算对象的计算顺序。

人们一般不在 C 语言里采用连续写关系运算符的方式，因为这样写出的表达式的计算结果往往使人感到意外。例如，下面是一个合法的表达式：

```
5 >= 3 >= 2
```

要弄清这个表达式描述的是什么则需要特别小心。根据关系运算符的结合方式，上面这个表达式相当于：

```
(5 >= 3) >= 2
```

关系(5 >= 3)成立，所以它的计算结果是 1。这样，上述表达式就相当于：

```
1 >= 2
```

它求出的值是 0，因为所描述的关系不成立！读者一般不会想到这种结果。要在程序里描述 5≥3≥2 一类的数学关系，最好是使用后面讨论的逻辑运算符和逻辑表达式。

逻辑值

关系式只有两种可能结果：它所描述的关系成立，或是该关系不成立。所以说，关系表达式描述的是一种逻辑判断。当一个关系成立时，人们常说这个关系式所表达的关系是“真的”，或说它具有逻辑值“真”；而在其关系不成立时就说该关系是假的，或说表达式具有逻辑值“假”。逻辑判断和逻辑值被用来控制计算的进程。

C 语言里没有专用的逻辑值类型，任何基本类型的值都可以当作逻辑值用，其中：

值等于 0	表示逻辑值“假”
值不等于 0	表示逻辑值“真”

也就是说，任何非 0 值都将被当作“真”（逻辑关系成立），0 值被当作“假”（逻辑关系不成立）。C 程序里有许多需要使用逻辑值的地方，条件表达式就是其中之一。

条件表达式

构造条件表达式需要使用条件运算符，这是 C 语言中唯一的一个具有三个运算对象成分的运算符。条件表达式的形式是：

表达式$_1$? 表达式$_2$: 表达式$_3$

这里的 ? 和 : 总是成对出现。条件表达式的计算方式比较特殊：在它被计算时，首先计算表达式$_1$；如果这个表达式的值非 0（即，条件成立），那么接着计算表达式$_2$，并用它的值作为整个条件表达式的值；如果条件不成立（表达式$_1$的值是 0），就计算表达式$_3$，并用它的值作为整个条件表达式的值。请特别注意，在表达式$_1$非 0 时根本就不计算表达式$_3$；在其值为 0 时不计算表达式$_2$。这个说明非常重要，请读者一定记住。

作为简单例子，现在看条件表达式 x > 0 ? 2 : 3 的计算过程。根据定义，此时先计算 x > 0，而后再根据情况计算另外两个表达式。不难看出，当 x 的值大于 0 时，这个条件表达式求出值 2；而在 x 的值小于 0 时求出的值是 3。可以清楚地看到，这个条件表达式的计算过程依赖于变量 x 当时的值，得到结果的也依赖于 x 当时的值。

有了条件表达式，前面提出的函数 double dmax (double, double) 就不难定义了。函数定义很简单，只包含一个返回语句，其中使用了条件表达式：

```
double dmax (double x, double y) {
    return x > y ? x : y;
}
```

再看另一个例子，定义是常用的符号函数 *sign*。这个函数的数学定义是：

$$sign(x)=\begin{cases}1 & x>0\\ 0 & x=0\\ -1 & x<0\end{cases}$$

也就是说，在参数值大于、等于或者小于 0 时，这一函数将分别得到 1、0 和-1。对应的 C 函数定义也是直截了当的：

```
double sign (double x) {
    return x > 0 ? 1 : (x == 0 ? 0 : -1);
}
```

条件运算符的优先级高于赋值运算符，但低于各种关系运算符。关于条件运算符结合关系的规定是：当多个条件表达式嵌套时，后续的每个 : 符号总与前面最近的没有配对的那个 ? 符号相匹配。根据这些规定，上面函数里的表达式可以简写为下面形式：

```
x > 0 ? 1 : x == 0 ? 0 : -1
```

加了括号后读起来可能更清楚些。

前面已经提出了条件表达式的特殊计算方式。下面是一个例子，当 x 的值不是 0 时，变量 z 将被赋予由 y/x 计算出的值，否则它被赋值 1。由于条件表达式计算方式的特殊规定，这个例子不会有问题：

```
z = x != 0 ? y/x : 1;
```

即使 x 当时值为 0，这里也不会出现除以 0 的情况。如果采用普通运算符的方式，先计算所有运算对象，然后再使用运算符，那么这个例子会怎么样？这个问题请读者考虑。

3.4.3 复杂条件的描述

编程时经常需要描述复杂的关系。例如，可能要说当变量的值 x 在区间 [3, 5] 之内时 z 应取值 2，否则就取值 1。这种情况可以利用关系表达式和条件表达式描述：

```
z = (x >= 3 ? (x <= 5 ? 2 : 1) : 1);
```

可以从理论上证明，对于描述复杂条件而言，有关系表达式和条件表达式就足够了，利用它们已经可以描述所有的复杂条件。但采用这些写法，写出的描述常常不太直观。为了方便人们在程序里描述复杂的条件，C 语言提供了逻辑运算符，利用它们可以描述：多个条件同时成立，多个条件之一成立，某个条件不成立等等。

C 语言提供了三个逻辑运算符 !、&& 和 ||，它们分别表示“否定”、“并且”和“或者”三种逻辑运算。其中的 ! 是一元运算符，另外两个是二元运算符。利用这些运算符可以写出各种逻辑表达式。逻辑表达式的计算结果都是整数类型的 0 或者 1。下表解释了这三个逻辑运算符的意义及其计算方式：

!表达式	把表达式的值看作逻辑值，以该值的否定作为结果：如果表达式的值非 0，则结果为 0；如果表达式值是 0 则结果为 1。

表达式$_1$ && 表达式$_2$	只有两个表达式都非 0 时结果为 1，否则为 0。 计算方式：先求表达式$_1$；若得到 0 则不计算表达式$_2$，以 0 作为整个表达式的结果；否则（表达式$_1$非 0）就计算表达式$_2$，如果它为 0 则整个表达式以 0 为结果，否则以 1 为结果。
表达式$_1$ || 表达式$_2$	只有两个表达式的值都为 0 时结果为 0，否则为 1。 计算方式：先求表达式$_1$；若得到非 0 则不计算表达式$_2$，以 1 作为整个表达式的结果；否则（当表达式$_1$值是 0 时）计算表达式$_2$，如果它为 0 则整个表达式以 0 为结果，否则以 1 为结果。

请注意运算符“&&”和“||”在计算方式上的特点，它们的计算方式与条件运算符类似，在一些情况下不求值第二个运算对象。下面是一个说明其特殊情况的例子：

```
x != 0.0 && y/x > 1.0
```

即使计算这个表达式时 x 的值为 0，计算中也不会出现除以 0 的问题。

逻辑运算符的计算结果也为 int 类型，与关系运算符一样，逻辑表达式得到的结果总是 0 或者 1。否定运算符是一元运算符，其优先级与其他一元运算符相同；两个二元运算符的优先级低于关系运算符，且“&&”的优先级高于“||”（参看本书附录 A 的表）。

例：根据运算符优先级关系，对逻辑表达式：

```
(((x + 3) > (y + z)) && (y < 10)) || (y > 12)
```

可以去掉其中所有括号，简写为如下形式，其意义不变：

```
x + 3 > y + z && y < 10 || y > 12
```

例：判断变量 year 的值是否表示一个闰年的年份。

如果变量 year 的值是闰年年份，那么这个值应当是 4 的倍数但又不是 100 的倍数，或者它是 400 的倍数。在 C 语言里，是某个数的倍数可以用取模运算的结果为 0 表示，所以，year 为闰年的条件可以写为：

```
year%4 == 0 && year%100 != 0 || year%400 == 0
```

由于优先级的规定，这个表达式里完全不需要写括号。当然，为了使人更容易看清楚，也可以适当加一些括号。下面是判断一个年份是否闰年的函数，该函数返回 int 值：

```
int isleapyear (int year) {
    return year%4 == 0 && year%100 != 0 || year%400 == 0;
}
```

这种做判断的函数在程序里很有用，人们常常将这种函数称为谓词。

3.5　语句与控制结构

C 语言里最基本的语句包括赋值语句和函数调用语句等[1]，它们完成一些基本操作。一次基本操作能完成的工作很有限，要实现一个复杂的计算过程，往往需要做许多基本操作，这些操作必须按照某种规定顺序逐个进行，形成一个特定操作执行序列，逐步完成整个工作。为描述各种操作的执行过程（操作流程），语言里必须提供相应的流程描述机制，这种机制一般称为*控制结构*，它们的作用就是控制基本操作的执行。

在机器指令层面上，执行序列的形成由 CPU 硬件直接完成。最基本的控制方式是顺序执行，一条指令完成后执行下一条指令，实现基础是 CPU 的指令计数器。另一种控制方式的代表是分支指令，这种指令的执行导致特定的控制转移，程序转到某指定位置继续下去。

[1] 实际上，按照 C 语言的说法，这些语句都被称为表达式语句，它们都是通过在一个表达式后面加一个表示语句结束的分号构成的。这是是程序里使用最多的基本语句。C 语言还有一些表示控制的基本语句。读者已经看到的 return 语句就是一个控制语句。

通过这两种方式的结合可以形成复杂的程序流程。如果将程序中的流程想象成在程序指令序列里缠绕的线路轨迹，早期程序的控制流出可能形成一团乱线，使人很难把握。

随着程序设计成为越来越多的人的职业，成为一个重要研究对象，人们对程序实践和对编程过程规律性做了许多研究，逐渐认识到，随意的流程控制方式不是一件好事，这种随意性带来许多麻烦，使得程序设计不能变成一种具有科学性的技术工作。

分析了各种情况后，人们提出了程序执行的三种基本流程模式，即顺序执行、选择执行和重复执行模式。在顺序执行中，一个操作完成后接着执行跟随其后的下一操作；选择执行中按照所遇情况，从若干可能做的事情中选出一种去做；重复执行过程则是在某些条件成立的情况下反复做某些事情。图 3.4 描述了这三种基本流程模式的一些典型情况。图 3.4 (a) 是顺序执行。图 3.4 (b) 中画的是选择模式的一种，其中在两种可能性里选出一种执行。还有多中选一等等形式。图 3.4 (c) 是一种重复执行结构，其中条件判断在先而动作在后。还有另外的重复执行模式。

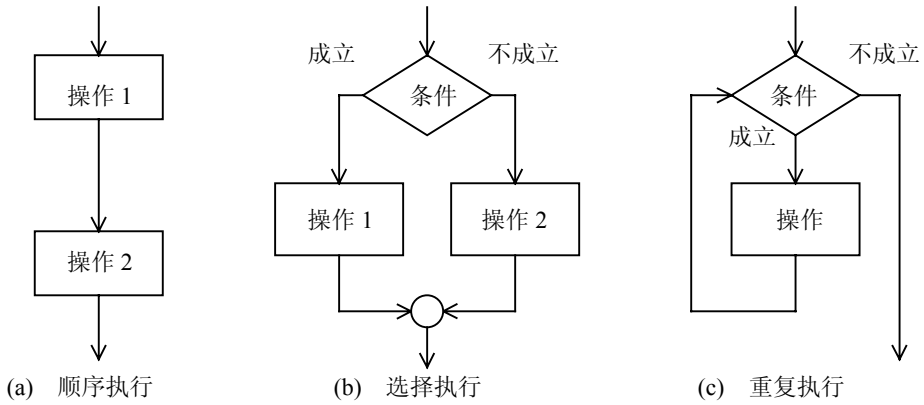

图 3.4　程序控制流程的三种基本模式

应特别指出，上面几种模式有一个共同点：它们都只有一个开始点和一个结束点。这一特点使一个流程模式的整体可以当作一个抽象操作看待，可以把它嵌入到其他不同的（或相同的）流程模式中，构成更复杂的计算流程。这样的流程称为*结构化的流程模式*。

通过结构化流程模式形成的复杂流程具有层次性，具有很好的分解，其意义也比较容易把握。人们已严格证明，上面画出的三种模式对于写任何程序都足够了。也就是说，如果能用其他方式写出一个程序，那么通过这三种模式的嵌套构造也能实现它。

C 语言提供了一组控制机制，包括直接针对上面几种模式的控制结构，这些控制结构也被称作*结构化的控制结构*。在 C 语言里，由一个完整控制结构形成的程序片段也被当作一个语句看待，可以出现在任何可以写语句的地方。这一规定使人可以嵌套地使用这些结构，写出各种复杂的程序。正因为此，控制结构也常常被人们称作*控制语句*。

前面讨论过的复合结构就是一种控制结构，它实现的是顺序执行模式。当一个复合语句执行时，作为其成分的语句被一个个顺序执行，这是程序的最基本执行方式。下面介绍另外几种常用的控制结构，还有几种结果留待下章介绍。

3.5.1 条件语句（if 语句）

条件语句用于描述在一些操作中选择执行。它以一个逻辑条件成立与否为条件，区分两种不同的执行方式，或决定一个操作的做或不做。条件语句有两种不同形式，分别是：

```
if （条件） 语句
if （条件） 语句1 else 语句2
```

请注意包围条件的括号，这是 if 语句的语法要求，写 if 语句时必须有这对括号。这里的条

件可以是任何基本类型的表达式，其值将被当作逻辑值使用，控制作为成分的*语句*的执行。两种条件语句的执行过程分别是：

1. 第一种形式：首先求出*条件*的值，其值非0时执行*语句*，该*语句*的完成也是整个条件语句的完成；否则（*条件*为0）就不执行*语句*，整个条件语句直接结束。
2. 第二种形式：首先求出*条件*的值，其值非0时接着执行*语句*$_1$；否则（值是0）就执行关键词else后面的*语句*$_2$。这两个*语句*之一执行完成时条件语句结束。

条件语句中的*语句*成分常常不是简单语句，而是复合语句或其他结构，也可以是条件语句。

请注意条件语句与条件表达式的不同。条件表达式根据给定条件决定求值方式，其基本目的是算出一个值，以便用于进一步的计算。条件语句则不同，虽然它也有一个条件，但作用是根据条件的成立与否决定做什么，执行什么语句。语句并没有值的概念。

当然，许多情况下两种结构都可以用，这时就应该从程序的简洁明晰等方面考虑和选择。也有些情况下两种写法在各方面的差异都不大，这时可以根据自己的喜好选择。举例说，前面做过的求参数中较大值的函数也可以用条件语句写：

```
double dmax (double x, double y) {
    if (x > y)
        return x;
    else
        return y;
}
```

这与前面定义的函数的功能完全一样。有些人（包括作者）更喜欢写前面简单的一行形式，读者可以根据自己的爱好选择。这里还看到了另一种情况：一个函数里可有多个 return 语句。函数执行中无论遇到哪个 return 语句，函数都会结束。

例：改进求圆盘面积的程序，如果给它参数中出现要求计算负值半径的圆盘面积的情况，它就输出一段信息，说明遇到了错误数据。

在前面给出的几个程序实例里，改造采用函数 pc_area 的那个程序最为简单，因为其中把与计算和输出有关的动作都封装在一个函数 pc_area 里了。修改程序时只需修改这个函数的定义：

```
void pc_area (double r) {
    if (r < 0)
        printf("radius incorrect: %f\n", r);
    else
        printf("radius: %f, area: %f\n", r, 3.14159265*r*r);
}
```

请读者修改有关的主程序并做些运行试验，确认这样修改确实完成了所需工作。另外，也请读者试试修改这个程序的其他版本（采用 c_area 函数的版本和没有定义函数的版本），比较一下完成同样修改所需的工作量。

if 语句的建议书写格式

为使程序清晰，便于阅读，就必须注意对各种结构采用良好书写形式。if 语句的结构比较复杂，下面是这种语句的建议书写形式，无 else 部分的 if 语句可类似处理。

1）当*语句*部分为单个语句时，常用如下书写形式：

```
if (XXXXXX)
    xxxxxxxxxxxx
else
    xxxxxxxxxxxx
```

*语句*部分非常短小时，有时人们也把它直接写在条件之后或者 else 关键字之后。

2）当*语句*部分为复合语句时，常用的写法有两种：

```
if (XXXXXX) {                          if (XXXXXX)
    xxxxxxxxxxx                        {
    xxxxxxxxxxx                            xxxxxxxxxxxx
}                                          xxxxxxxxxxxx
else {                                 }
    xxxxxxxxxxx                        else
    xxxxxxxxxxx                        {
}                                          xxxxxxxxxxxx
                                           xxxxxxxxxxxx
                                       }
```

这里的原则很简单：应把关键字 if 和 else 对齐。复合语句的花括号可看作是本层结构，也作为下一层的范围界定标志，而复合语句内部的成分是下一层结构，应该适当退格并令其互相对齐。这样可以使程序的层次清晰明了。本书后面的示例程序将始终坚持这种形式。也请读者在写程序时效仿这种良好形式。

例：请写函数 void root2(double a, double b, double c)，三个参数分别看作二次方程 $ax^2+bx+c=0$ 的三个系数，函数 root2 分情况求出方程实根并打印输出。

根据数学知识，求二次方程的实根，首先需要求出方程判别式的值，根据判别式可以区分出三种情况：该方程有两个实根，有一个重根，或者没有实根。下面函数采用的就是这种分步解决的方式，其主要部分是一个嵌套的条件语句。

```
void root2 (double a, double b, double c) {
    double tmp, d = b*b - 4*a*c;
    if (d > 0) {
        tmp = sqrt(d);
        printf("Two real roots: %f, %f\n",
               (-b + tmp)/2/a, (-b - tmp)/2/a);
    }
    else if (d == 0)
        printf("One real root: %f\n", -b/2/a);
    else
        printf("No real root\n");
}
```

这个函数里出现了一个条件语句的 else 语句部分又是一个条件语句，通过连续判断完成三种情况的分别处理。这种形式在程序里很常见，有时可能连续出现许多判断。遇到这种情况时，人们常采用上面例子里所用的书写方式。一方面，这种方式并不影响程序的可读性，程序结构仍能得到清晰反映；另一方面，这样做又避免了连续退格引起大片空白给书写造成的麻烦（一再退格可能使有些行退得很远，影响程序的读写）。

请读者考虑，这个函数还有哪些情况没有处理，并将有关修改工作作为练习。

if 语句的嵌套问题

if 语句有两种不同形式，各种语句又可以嵌套出现，这样，有时在写嵌套的 if 语句时就会出现问题。上面那种在 else 部分又是条件语句的情况不会引起任何问题。问题出在条件之后直接出现条件语句的情况。下面是一个例子：

```
if (x > 0)
    if (y > 1) z = 1;
else z = 2;  /*这个 else 部分属于哪个 if? */
```

按条件语句的语法形式，上面例子似乎有两种可能的解释。第一种解释：处在外层是一个没有 else 部分的条件语句，最后的 else 部分属于内层的那一个 if 语句。第二种解释：内层是一个不带 else 部分的条件语句，最后一行的 else 属于外层条件语句。在程序语言里绝不能允许两种不同解释同时存在（这种情况被称为*歧义性*，是语言定义的大忌），对这个问题必须有一个明确的说法。

C语言规定，每个 else 部分总属于前面最近的那个缺少对应的 else 部分的 if 语句。根据这一规定，上面的第一种解释是正确的，第二种解释不正确。进一步说，上面示例里的写法很不合适，因为它的退格方式容易迷惑人，造成误解。如果我们真的需要写出具有第二种意义的嵌套条件语句，那么就应该采用下面的写法：

```
if (x > 0) {
    if (y > 1) z = 1;
}
else z = 2;
```

3.5.2 循环语句（1）：while 语句

循环控制结构实现实现程序的重复执行模式。C 语言有几种循环控制结构，while 语句是其中较简单的一种，使用较多。while 语句也被称为*当型循环语句*，其形式是：

while （条件） 语句

*语句*部分称为*循环体*，常常是一个复合语句，也可以是其他控制结构，也可以是循环结构（这种情况称为*多重循环*）。while 循环的执行方式如图 3.4（c）所述，也就是：

1) 首先求出条件的值；
2) 如果条件的值为 0 则整个 while 语句结束；否则
3) 执行循环体，而后回到 1) 继续。

例：写出求 $\sum_{n=1}^{100} n^2$ 的程序段，假设已经有整型变量 sum 和 n。程序段可以写为：

```
sum = 0;
n = 1;
while (n <= 100) {
    sum = sum + n * n;
    n = n + 1;
}
```

在这段程序执行之后，变量 sum 的值将是我们所需要的结果。

从这个例子可以看到循环程序的一些特点。在进入循环之前，通常需要给循环中使用的各个变量设置循环初值，上面例子里用两个赋值语句完成了这项工作。在每次循环体的执行中，虽然执行的都是同样程序片段，但由于参与循环的一些变量的值改变了，实际做的事情就可能不同。显然，一个循环结构需要有一个继续条件（它的否定就是循环终止的条件），它控制着循环的进行过程。

我们看到，虽然上面的程序片段没有几行，但是由于循环结构的特定执行方式，循环体里的语句可能多次执行，由这个不长的程序片段引起的计算过程可能是很长的。在编写实际程序时，一个重要工作就是确定在计算过程中的重复性动作，并通过适当的循环形式正确地描述这种重复动作。下一章将重点讨论循环的构造问题。

不难将上述程序片段做在一个完整的程序里。例如：

```
#include <stdio.h>

int main() {
    long n, sum;
    sum = 0;
    n = 1;
    while (n <= 100) {
        sum = sum + n * n;
        n = n + 1;
    }
    printf("Sum: %d\n", sum);
    return 0;
}
```

这里采用 long 类型的变量，是因为在一些 C 语言系统里（在采用 16 位整数的系统里），int 类型的变量无法保存算出的和数。

为了节约篇幅，今后我们有时将只给出上面那样的程序片段。把它们放进程序里，以便检测其功能的工作留给读者自己完成。

程序示例

例：写一个程序，要求它从摄氏温度 0 度到 300 度，每隔 20 度为一项，输出一个摄氏温度与华氏温度的对照表。

一种直接而简单的解决方案是在程序里写 16 个类似的输出语句，这样可以解决问题，但却并不理想（如果题目要求输出 10000 项的对照表，我们该怎么办？）。很明显，这里存在着一项重复性的工作，可以考虑用循环结构处理。可以考虑用一个变量保存所处理的摄氏温度值。在循环开始前给这个变量赋初始值 0，每次循环体执行中将它的值加 20，一直加到 300。在每次循环中计算出有关结果，并输出一行信息，其中包括当时的摄氏温度值和华氏温度值。这一分析形成了一套解决问题的方案。

不难找到摄氏与华氏温度对照公式：

$$F = C * 9 / 5 + 32$$

根据上面分析写出程序已经不难了。将临时变量取名为 c，转换结果用双精度类型表示。

```
#include <stdio.h>

int main () {
    int c = 0;
    while (c <= 300) {
        printf("C = %d, F = %f\n", c, c * 9 / 5.0 + 32.0);
        c = c + 20;
    }
    return 0;
}
```

这个程序将显示 16 行输出，其中第一行是：

```
C = 0, F = 32.000000
```

其他各行的输出形式都与此相同。

3.5.3 循环语句（2）：for 语句

从前面程序示例可以看出循环的一种常见模式：开始循环前先做一些准备工作，为循环中用到的一些变量赋循环初值；然后进入循环，其中需要考虑循环继续的条件是否成立；如果条件成立就执行循环体；循环体最后常需要更新一些控制循环进程的变量。这种模式很普遍，C 语言为此提供了专门的 for 循环结构。for 结构的成分较多，其完整形式是：

for （表达式$_1$； 表达式$_2$； 表达式$_3$） 语句

其中表达式$_1$完成初始变量设置（通常写赋值表达式），表达式$_2$是确定循环是否继续的条件，表达式$_3$常用于循环变量更新，语句部分是循环体。其执行方式是：

1) 先求表达式$_1$的值，这只做一次。这里通常写给循环变量初值的赋值表达式；
2) 求表达式$_2$的值，如果它得到 0 则循环结束，否则继续；
3) 执行作为循环体的语句；
4) 求表达式$_3$的值。这里通常写更新循环变量的赋值表达式；
5) 转到 2) 继续执行。

for 结构头部的三个表达式都可以没有，但分号不能少。缺第一或第三个表达式表示不做这部分动作。缺第二个表达式表示条件为 1，是不终止的循环。人们有时需要写不终止的循环。C 语言也提供了从循环里退出的其他机制。例如，在函数定义里可以用 return 语句从循环中直接返回，执行这种 return 也导致退出当时的循环。

程序示例

例 1，用 for 循环重写打印摄氏和华氏温度对照表的程序。

写出来很简单，和前面程序类似，只是将 while 结构换成了 for：

```
#include <stdio.h>

int main () {
    int c;
    for (c = 0; c <= 300; c = c + 20)
        printf("C = %d, F = %f\n", c, c * 9 / 5.0 + 32.0);
    return 0;
}
```

现在循环体里只有一个语句了。for 语句的所有控制信息都出现在语句前部，因此常常更容易读，更容易理解。这种有变量的准备部分，有循环条件，有变量值的更新的重复计算，在 C 语言里特别适合用 for 语句实现。

例 2：写一个求 $\sum_1^n k^2$ 值的函数，假设值 n 由参数得到。

首先确定函数的类型特征，很容易：

```
int sqsum(int n)
```

题目要求重复累加，累加次数是由参数的值 n 确定的。写函数时不知道函数调用的参数值，不同调用中参数的值可能不同，因此必须用循环解决。循环里显然需要有一个变量记录变化中的 k 值，每次循环该变量加 1。还要一个变量记录部分和，每次更新它的值，最后由它可以得到总和。通过这些分析，可以确定循环中需要用两个变量，为它们取名 k 和 sum，采用整型，于是有下面定义：

```
int k, sum;
```

循环开始前可将 sum 设为 0，表示 0 个项的部分和。k 的初值可以设为 1，由它可以算出第一项的值。采用在发现 k 的值大于 n 时终止循环，由此得到循环的继续条件是 k <= n。有了这些分析和考虑，就可以得到如下循环：

```
sum = 0;
for (k = 1; k <= n; k = k + 1)
    sum = sum + k * k;
```

也可以用 while 写：

```
k = 1;
sum = 0;
while (k <= n) {
    sum = sum + k * k;
    k = k + 1;
}
```

很容易把这样的程序段包装成函数（用 while 语句实现也同样简单和方便）：

```
int sqsum (int n) {
    int k, sum = 0;
    for (k = 1; k <= n; k = k + 1)
        sum += k * k;
    return sum;
}
```

3.6　循环程序常用的若干机制

C 语言里有一些常在循环结构里使用的运算符，下面介绍这些运算符。

3.6.1 增量和减量运算符（++、--）

增量和减量运算符用于将变量值加 1 或减 1。两种运算符都有前置写法和后置写法：

将变量 x 的值增加 1		将变量 x 的值减少 1	
++x	x++	--x	x--

以增量运算符为例，上述两种写法在将变量的值增加 1 方面作用相同，但它们作为表达式求出的值不同：前置写法 ++x 求出的值是 x 加一以后的值；后置写法 x++的值是 x 加一操作之前的值。减量操作情况也类似。请看下面语句序列在计算中的情况：

```
x = 2;
y = 2 + ++x;  /* x值变为3，y置为5，因为 ++x 的值是加一之后的值 */
z = 3 + x++;  /* x值变为4，z取得值6，因为 x++ 的值为3，是加一之前的值 */
```

增量和减量运算符常用于循环变量更新。另请注意上面第二个语句的写法，我们在增量运算符和加运算符之间写了空格，这是非常必要的。这里前后出现了三个加符号，插入空格可以保证编译系统对这个表达式的分析不出现错误。

增量（减量）运算符的意义

1）语句“x++;”和语句“x = x + 1;”

当 x 是简单变量时，上面两个语句的意义相同。增量运算也可用于实数类型变量，计算中将做相应的类型转换。后面会看到，对于非简单变量，两种不同写法有可能导致不同计算结果。这两个语句主要差别在于：语句“x++;”的执行中只计算 x 一次，而在语句“x = x + 1;”执行中需要计算 x 两次。减量操作的情况类似。

2）提供增量和减量运算符，一方面为了程序书写的方便；另一方面是考虑作为编译结果的程序执行效率。一般 CPU 都提供了增量和减量指令，增减量运算可直接采用相应指令实现，这样产生的可执行代码效率可能更高些。

3）如果不使用表达式的值，那么写 ++n 和 n++ 并没有什么差异。例如写代码：

```
for (i = 1; i <= 20; ++i)
    printf("%d\n", i*i);
```

将其中 ++i 换成 i++，这段代码的意义不变。人们提倡在一般情况下用前置写法。

增减量运算符的使用

增减量运算符常用于 for 语句头部的变量更新，如上面例子里所示。作为另一个例子，前面求平方和的程序段可以写成：

```
sum = 0;
for (n = 1; n <= 100; ++n)
    sum = sum + n * n;
```

这两个运算符也常用在独立的语句里。

如果将这些运算符写在普通表达式里，那就需要特别注意了。增减量表达式不但能算出一个值，还会修改作为运算对象的变量。因此，如果使用不当，就可能写出依赖于计算顺序的表达式。例如下面几个语句中的意义都没有定义：

```
m = ++n - (n - m);
n = (n * m - n--) + m;
```

因为其计算结果都将依赖于计算的顺序。请读者自己弄清其中情况。这些情况告诉我们，一般最好不将增减量表达式写在复杂表达式的内部，以免因为疏忽造成错误。

3.6.2 逗号运算符

逗号运算符在形式上是一个逗号，它是 C 语言里优先级最低的运算符（其优先级比赋值运算符还低）。逗号运算符采用从左向右的结合方式，其形式是：

表达式$_1$，表达式$_2$

当一个逗号表达式执行时，首先求表达式$_1$的值，然后求表达式$_2$的值，并以表达式$_2$的值作为整个逗号表达式的值（表达式$_1$求出的值就不再关心了）。

逗号表达式主要用在 for 语句头部的变量初置部分和变量更新部分。有了逗号表达式，

这两个部分中就可以赋值或更新多个变量。下面是使用逗号表达式的一个例子：

```
for (sum = 0, n = 1; n <= 100; n++)
    sum = sum + n * n;
```

3.6.3 实现二元运算符操作的赋值运算符

程序里常常需要“sum = sum + n * n;”形式的赋值语句做变量更新，其中用到一个二元运算符，从变量原有值出发，通过与另一表达式运算得到新值再赋给变量。这种操作在程序中很典型。为能更方便地描述这类操作，C语言为许多二元运算符提供了对应的赋值运算符。每个算术运算符都有对应的赋值运算符，分别是：

```
+=      -=      *=      /=      %=
```

这些运算符的优先级与简单赋值运算符相同，同样采用从右向左的结合方式。它们的计算结果就是变量的最后更新值，类型与变量类型相同。写在这些赋值运算符左边的必须是变量，右边可以是任何表达式。

下面是一些例子，每行中左边的语句在效用上与右边语句相同：

```
x += 3.5;                   x = x + 3.5;
sum += n * n;               sum = sum + n * n;
res *= x;                   res = res * x;
x += y += 3;                x = x + (y = y + 3);
```

下面是使用赋值运算符的例子：

```
for (sum = 0, i = 1; i <= 100; i++)
    sum += n * n;
```

这些赋值运算符也有与增量、减量运算符类似的问题。因此上面说的效用等价并不准确。这里也有一次计算或两次计算的问题等，也可能有实现效率问题。

3.6.4 空语句

C语言的另一基本语句是空语句，形式上就是一个分号。空语句执行时什么也不做，其用途就是作为填充，有时需要用它将程序的语法结构补充完整。例如：

```
for (sum = 0, i = 1; i <= 100; i++, sum += n * n)
    ;
```

写在第二行的分号就是一个空语句，在这里表示 for 的循环体。没有这个空语句就不是完整的 for 结构，可能将紧随其后的另一个语句当作循环体，造成程序错误。

利用空语句，前面在讨论 if 语句嵌套时所用的例子现在可以写成：

```
if (x > 0)
    if (y > 1)
        z = 1;
    else
        ;
else z = 2;
```

第一个 else 后的分号表示空语句。

还应该指出在 if 语句使用中的一个常见错误。初学者常常写出下面的 if 片段：

```
if (xxxxx) {
    ......
};
else {
    ......
}
```

编译这段程序会报错，指出 else 的出现位置不对。错误原因是第一个复合语句后的分号。复合语句不需要分号结束，这个分号被认为是个空语句。这样，编译系统认为该 if 语句没有 else 部分（因为 else 部分必须紧随着 if 条件后的第一个语句）。

问题解释：

1）（3.2.2）考虑除法是在哪个类型里进行的。赋值号右边的表达式在整数类型中计算，除法

产生丢掉小数点后面信息的情况，这个例子恰好被除数是奇数，结果不可能正确。
2）（3.3.2）最小程序编译连接后得到的程序包含几千字节代码，这是由连接程序装进去的C程序基本运行系统。回头看一看第一章关于C程序的加工过程。
3）（3.4.2）显然，在x的值是0的时候，程序将出现除0的错误。
4）（3.6.2）n值超出整数范围，肯定导致计算结果出错，另一个可能性是使函数进入无穷无尽的循环之中。由于整数表示范围的限制，对于很大的参数值，这个计算过程不能收敛。

本章出现的有用程序模式

程序模式3.1：简单程序

```
#include <stdio.h>
/* 如果需要用数学函数，这里还要写#include <math.h> */

int main () {
    /* 若干变量定义（以及初始化） */
    /* 若干计算和赋值语句 */
    /* 若干输出语句 */
    return 0;
}
```

程序模式3.2：带函数定义的程序

```
#include …
……  /* 函数定义写在这里（可以有一个或几个） */

int main () {
    …… /* 主程序体，这里通常包含对一些函数的调用 */
    return 0;
}
```

本章讨论的重要概念

语句，语义，复合结构（复合语句），顺序控制，变量，赋值，取值，变量定义，变量命名，赋值运算符（=，赋值号），赋值表达式，赋值语句，多重赋值，变量初始化，注释，主程序与函数，函数定义，函数头部，函数体，参数表，函数参数，返回值，函数的类型，局部变量，返回语句（return语句），不返回值的函数，主函数，关系，关系运算符（<、<=、>、>=、==、!=），关系表达式，逻辑值，条件运算符（?:），条件表达式，逻辑运算符（!、&&、||），逻辑表达式，否定，并且，或者，控制结构，顺序执行，选择执行，重复执行，结构化控制结构，条件语句（if语句），条件语句的嵌套，while语句，for语句，增量运算符（++），减量运算符（--），逗号运算符（,），各种赋值运算符，空语句。

练习

1. 下面的字符序列中哪些不是合法的变量名：

-abc	__aa	for	pp.288	to be
IBM/PC	ms-c	#micro	m%ust	tihs
while	r24_s25	__a__b	a"bc	_345

2. 假设整型变量a的值是1，b的值是2，c的值是3，在这种情况下分别执行下面各个语句，写出执行对应语句后整型变量u的值。
   1）　u = a ? b : c;
   2）　u = (a = 2) ? b + a : c + a;
3. 假设整型变量a的值是1，b的值是2，c的值是0，写出下面各个表达式的值。
   1）　a && !((b || c) && !a)
   2）　!(a && b) || c ? a || b : a && b && c

3）　`!(a + b < c) && b <= c * a - b`

4. 下面程序在执行时，哪些地方将发生类型转换？程序打印的值是什么？

```
int f (int n, float m) {
  return (m + n) / 4;
}

int main () {
  float y = 3;
  printf("%d\n", f(y, y + 1));
  return 0;
}
```

5. 在计算机上试验本章正文中的一些程序。对它们做一些修改，观察程序加工和运行的情况，并对程序的行为做出解释。
6. 定义求圆球的体积、求圆球的表面积、求圆柱体的体积、求圆柱体的表面积的函数。
7. 1）不用函数，直接写一个主程序计算并输出直径为 100 毫米和 150 毫米的金、银、铜、铁、锡球的重量（以 kg 为单位输出）。

   2）重新完成上面程序，先定义一个带有两个参数的函数，它能求出直径为 x 的比重为 y 的圆球的重量，而后在主程序里调用这个函数完成所需工作。将这样得到的解与不用函数的解比较，比较它们的长度、容易出错的程度。假设现在要求修改所用圆周率的精度，考虑用两种方式写程序的修改难度。

   3）请写程序，求出边长为 100 毫米和 150 毫米的金、银、铜、铁、锡立方体的重量。你可以利用前面的程序吗？是否很容易修改前面程序，完成这一计算？比较不用函数的解法和使用函数的解法在易修改和重复使用方面的效用。
8. 如果四边形四个边的长度分别为a、b、c、d，一对对角之和为2α，则其面积为：

$$S=\sqrt{(s-a)(s-b)(s-c)(s-d)-abcd\cos^2\alpha}$$

   其中$s=\frac{1}{2}(a+b+c+d)$。定义一个函数计算任意四边形的面积。设有一个四边形，其四条边边长分别为 3、4、5、5，一对对角之和为145°，写程序计算它的面积。
9. 定义函数：`double tmax(double, double, double)`，它返回三个参数中最大的一个。写一个主函数试验各种参数情况。
10. 写函数，它以两个电阻的值作为参数，求出并联的电阻值。
11. 修改已知四边长求四边形面积的函数，增加对各种参数错误情况的检查和处理（如返回值 0），用各种实例数据检查你的函数否检查出所有可能的错误情况。
12. 分析本章正文中给出的求二次方程根的函数，看它缺乏对哪些特殊情况的处理。补充这些处理，在需要时输出适当的信息，使之成为一个更完整的函数。写一个主函数，用各种特殊情况和一般情况测试所完成的函数。
13. 写一个简单程序，它输出从 1 到 10 的整数。
14. 写一个简单程序，它输出从 10 到－10 的整数。
15. 写一个两个整型参数的简单函数，它输出从第一个整数到第二个整数为止的整数序列。
16. 用定义函数 `double power(double x, int n)`，它求出 `x` 的 `n` 次幂。用主函数试验很大的 `n` 值（例如令 `x` 值为 1），看看会出现什么情况；用大的 `x` 和 `n` 值，看看发生浮点数计算溢出时会出现什么情况。
17. 写一个程序，它在 0～90 度之间每隔 5 度输出一行数据，打印一个表。每行中包括 5 个项目：角度数，以及它所对应的正弦、余弦、正切、余切函数值。
18. 查看有关公式，写求解并输出一元三次方程的根的函数。
19. 写出求等差级数的和 $\sum_{k=1}^{n} ka$ 的函数。两种循环结构给出函数定义，再利用等差级数求和公式给出函数定义。

20. 请到查出银行一年定期存款的利率和 5 年定期存款的利率。假定现在要存入 100 元钱，存款到期后立即将利息与本金一起再次存入。请写出程序，计算按每次存一年和按照每次存 5 年，总共存 50 年后两种存款方式的得款总额。对两种情况都每隔 5 年输出一次当时的总金额。
21. 写一个程序打印出 2 的顺序各次幂。让它打印出 2 的前 30 个幂，看看会出现什么情况。用一个条件为真的循环打印 2 的各次幂，看看会出现什么情况。