第五章　C 程序结构

本章讨论一些与 C 程序整体结构有关的问题，它们对于正确理解 C 语言，正确书写 C 程序都非常重要。有些人使用 C 语言许多年，但仍常犯一些错误，自己也常弄不清楚错在哪里，其原因往往就是一些基本问题不够清楚。本章的讨论方式是希望读者在学习 C 语言编写程序的过程中，也能了解一些更深的原由。在安排这章的材料时，希望能把问题讲得比较透，也能多举出一些例子，帮助读者理解。还有些更有意思的例子出现在后面章节里。本章有些内容比较深入，读者第一次阅读时可能遇到一点困难，未必能完全理解。这也没有关系，建议读者在学了后面章节后，重新回来读读这章的内容。

5.1 数值类型

第二章里已介绍了几种常用数值类型。本节对 C 的所有数值类型做一个全面介绍。

实数类型和整数类型

实数类型共有三个，类型名分别是：

```
float， double， long double
```

这些类型都已在第二章节介绍了。请注意各种实数文字量的书写形式，其中必须包含小数点或指数部分。不加后缀的是 double 类型文字量，float 类型的文字量在数值表示后面加后缀 f 或 F，long double 类型文字量加后缀 l 或 L（建议用大写 L，小写 l 容易与数字 1 混淆）。实数类型内部编码一般采用有关的国际标准（IEEE 标准）。例如：

```
float f1, f2;
double x1, x2;
long double l1, l2;
```

除了实数类型之外的数值类型都是*整数类型*。C 语言将字符类型也看作整数类型，可以作为整数参加运算。各种整数类型都分为*带符号*与*无符号*的两种，带符号类型表示一定范围内的正数和负数，无符号类型的值都不小于 0。在类型名前加 signed 或 unsigned，说明一个整数类型是带符号的或是无符号的。其中 signed 都可以省略，也就是说，不加特别说明的都是带符号类型。字符类型在这方面的情况比较特殊。

字符类型

写简单程序时通常只用字符类型 char。在一般 C 语言系统里，一个字符占一个字节，其中存着字符的编码。字符类型主要用于存储文字信息和输入输出。如果将字符的使用限制在这一范围里，我们将不会遇到任何复杂情况。

如果把字符当作整数参加运算，所用的就是字符的编码（这是一个整数）。由此可见，一个字符与其他整数运算时起什么作用，要看所用的计算机系统里字符的编码方式。在目前使用最广泛的编码系统（如 ASCII 或 EBCDIC 编码系统）里，数字字符和英文字母字符的编码都是顺序地连续排列的。常常可以看到一些 C 程序利用了这一特征。例如，我们可能看到程序里有下面片段：

```
if (c >= 'a' && c <= 'z') ...... /* 判断c中存储的字符是否小写字母 */
n = d - '0';  /* 将变量d里保存的数字字符的“数值”赋给整型变量n */
```

这里假定 c、d 保存着字符的编码，而 d 中存的是某个数字字符。因为数字字符的编码连续排列，假设变量 d 的值为 '3'，d - '0'（字符 0 的编码）的结果正好是 3。

实际上，C 语言还有 signed char 和 unsigned char 两个字符类型，普通 char 类型等价于这两者之一，等价于哪一个要看具体的 C 系统。如果程序里只用普通的可打印

字符（字母、数字、各种标点符号、空白字符等），这方面的情况不会产生任何影响。

把字符区分为“有符号字符”和“无符号字符”，这一点不太好理解，似乎也没有什么道理。问题只出现在用字符类型的数据与其他整数运算时：如何看待字符所表示的数？是把字符数据看成有符号的整数呢，还是看成无符号的整数？这方面的问题在写初级程序的时候完全没有必要去考虑。现在只需要知道这种情况。

整数类型

基本的整数类型有三个：

int， short int， long int

其中 short int 可以简写为 short，而 long int 可以简写为 long。这三个类型都有对应的无符号类型，因此整数类型实际上有六个：

int	short	long
unsigned int	unsigned short	unsigned long

这里的 unsigned int 还可以简写为 unsigned。上表里许多类型名已经是简写形式，例如，unsigned long 的完整形式是 unsigned long int。

C 语言标准没有规定 int、short int、long int 的具体实现方式（二进制编码长度），只规定了一些原则，主要有：short 类型的表示范围不大于 int 类型的表示范围，long 类型的表示范围不小于 int 的表示范围。并规定 short 至少为 16 位，long 至少为 32 位；各 unsigned 类型总采用与对应 signed 类型同样长度的表示。每个类型的具体表示（用多少位表示，用什么编码方式等）由具体 C 语言系统规定。

这里最基本的类型是 int。C 系统里的 int 类型一般采用相应计算机的字长。例如，16 位计算机的 C 语言系统的 int 类型通常采用 16 位表示方式；而在 32 位计算机的 C 语言系统中 int 类型通常用 32 位表示。

PC 机上 C 系统的情况比较复杂。一些老的 C 系统（DOS，Windows 3.1 上的 C 系统）通常采用 16 位的 int 类型，因为 16 位是 8086/8088 CPU 的字长。这时 int 类型的表示范围是 -32768～32767，即 $-2^{15} \sim 2^{15}-1$。unsigned int 用 16 位，表示范围是 0～65535，即 $0 \sim 2^{16}-1$。这些系统里的 long 类型通常用 32 位表示，short 类型通常也用 16 位表示。一些新的 C 系统（如运行在 Windows NT、Windows95/98/2000 等系统上 C 系统）则采用 32 位的 int 类型，long 类型用与 int 一样的表示方式，short 用 16 位表示。

有关具体 C 系统里各种类型的情况，使用前应该查阅系统的有关材料。如语言手册、联机帮助信息或有关书籍。C 标准库里有两个名字分别为“limit.h”和“float.h”的文件，其中列出了与本系统所有数据类型表示有关的信息，读者可以查阅所用 C 系统里这的这两个文件。这方面的进一步细节请参看第 11 章里对标准库情况的介绍。

整数类型的文字量用连续数字序列表示。前面已讲过整型字面量的十进制、八进制和十六进制表示问题。如果需要特别表示写的是长整数，就应加上后缀 l 或 L，short 类型没有字面量写法。无符号整数的后缀是 u 和 U，无符号长整型加后缀 UL 或者 LU 均可。下面是一些无符号整数的字面量的例子：

123U，2987654LU，327LU，32014U

无符号整数类型的另一个特点是算术运算以对应类型的表示范围为模进行。当计算结果超出类型的表示范围时，以取模后的余数作为计算结果。假定 unsigned 用 16 位表示，表示范围是 0～65535。如果计算结果超出这个范围，就以得到的结果除以 65536 的余数作为结果。例如 234+65500 的结果将是 198。其他无符号类型的情况也一样。

由于类型问题，计算中可能出现隐含的类型转换动作。C 语言规定，当各种小整数类型（short、unsigned short 类型，各种 char 类型）的数据出现在表达式之中，计算之

前先将它们转换为 int 类型的值后再参与运算，这一过程称为整数提升。如果某类型的一个值超出了 int 的表示范围(例如 unsigned short 类型的提升时就可能出现这种情况)，那么在整数提升中将其提升为无符号整数类型。

前面讲过，两个不同类型的数值对象进行运算前，要把小数据类型的值转换到大数据类型。现在还要补充一点：在基本类型相同时，C 语言认为无符号类型是比同样有符号类型更大的类型。举例说，如果要做下面计算：

```
2365U + 18764
```

首先要从整型值 18764 转换生成一个无符号整数的对应值，然后用这个新值参与计算。如果需要转换的有符号整数的值为负，转换结果依赖于具体的系统。

如果要求将无符号数转换到有符号数（通过强制转换或者赋值、参数传递等），那么也按前面所说的规则处理：如果原类型的值能在转换的目标类型中表示，那么转换后的值不变，否则转换结果依赖于具体的系统。

基本数据类型的选择

C 语言提供了多个浮点数类型和多个整数类型，目的是使编程者有较多选择机会，满足复杂的系统程序设计中的各种需要。C 语言应用广泛，不同程序或软件中对数值的表示范围和精度的要求会有很大差异。对于某些应用问题而言，选择合适的数值类型可能很重要，在写那些程序时，人们就需要更仔细地考虑，确定每一个变量应该采用哪个数值类型。这是专业 C 程序员的写重要程序时的一项工作。

然而，对于一般程序，特别是对于学习 C 程序设计而言，这种选择就不那么重要了。现在提出如下的类型选择原则，这也是在大多数 C 程序里的最合理选择：

1. 如果没有特殊需要，浮点数总采用 double 类型，因为它的精度和表示范围能满足一般要求（float 的精度常常不够，long double 可能降低效率）。
2. 如果没有特殊需要，整数总采用 int 类型，因为它是每个 C 系统里的最基本类型，必定能得到硬件的基本支持，其效率不会低于任何其他整数类型。
3. 如果没有特殊需要，字符总采用 char 类型。
4. 尽量少用各种 unsigned 类型，除非服务于某些特殊目的。

5.2 函数和标准库函数

随着要处理的问题越来越复杂，程序也会变得越来越长。程序长带来许多问题：长的程序开发困难，牵涉的情况更复杂，写程序的人更难把握。长程序的阅读和理解也更困难，这又影响到程序的开发和维护。如果要修改程序，就必须先理解一项改动对整个程序的影响，防止其破坏了程序的内在一致性。另外，随着程序变大，程序中也常出现一些相同或类似的代码片段，这使程序变得更长，也增加了程序里不同部分间的互相联系。

处理复杂问题的基本方式就是设法把它分解为一些相对简单的部分，分别处理这些部分，然后用各个部分的解去构造整个问题的解。为支持复杂计算过程的描述和程序设计，就需要程序语言提供分解复杂描述的手段，需要有把代码段抽象出来作为整体使用和处理的手段。随着人们对程序设计实践的总结，许多抽象机制被引进了程序语言。这些机制极为重要，人只有借助于它们才可能把握复杂的计算过程，完成复杂的程序或软件系统。C 是 70 年代初研制开发的语言，那时人们在这方面的认识还比较粗浅，所以这里只提供了对计算过程片段的抽象机制，这就是前面已初步介绍过的函数机制。

函数的作用是使人可以把一段计算抽象出来，封装（包装）起来，使之成为程序中的一个独立实体。还有为这样封装起的代码取一个名字，做成一个函数定义。当程序中需要做这段计算时，可以通过一种简洁的形式要求执行这段计算，这种片段称为函数调用。

函数抽象机制带来了许多益处：

1. 重复出现的程序片段被一个唯一的函数定义和一些形式简单的函数调用所取代，这样有可能使程序变得更简短而清晰。
2. 由于整个程序里同样的计算片段仅描述一次，需要改造这部分计算时，就只要修改一个地方：改变函数的定义。程序的其他地方可能完全不需要修改。
3. 函数定义和使用形成对程序复杂性的一种分解，使人在程序设计中可以孤立地考虑函数本身的定义与函数的使用问题，有可能提高程序开发的效率。
4. 把具有独立逻辑意义的适当计算片段定义为函数后，函数可以看成是在更高层次上的程序基本操作。一层一层的函数定义可以使人可以站在一个个抽象层次上去看待和把握程序的意义，这对于开发大的软件系统是非常重要的。

在前面章节里，已经讨论了许多有关的实例。

5.2.1 C 语言的库函数

C 语言是一种比较简洁的语言，其基本部分较小，例如，语言本身甚至没有提供输入输出功能的结构。C 程序所需要的许多东西都是通过函数方式提供的。

每个 C 系统都带有一个相当大的函数库，其中以函数方式提供了许多程序中常用的功能。ANSI C 标准对函数库做了规范化，总结出一批最常用的功能，定义了标准库。今天的每个 C 系统都提供了标准库函数，供人们开发 C 程序时使用。标准库的功能通过一批头文件描述，如果要使用标准库的功能，就需要用 #include 命令引进相应头文件。

此外，具体 C 系统通常还根据其运行环境的情况提供了扩充库，使采用这个 C 系统开发的程序可利用特定硬件或操作系统的功能等。例如，运行在微机 DOS 系统上的 C 系统将提供一批利用 DOS 系统特定功能的函数；运行在 Windows 上的 C 语言系统都提供了一批与 Windows 环境有关的函数；运行在 UNIX 上 C 的系统必定提供一批与 UNIX 系统接口的函数。扩充库的功能也是通过一批头文件描述的，使用它们使也需引入相应头文件。

无论是标准库函数还是扩充库函数，都可看作常用计算过程的抽象。如果写程序时需要，就可以按规定方式直接调用这些函数，不必自己写程序实现这些功能，也不必关心这些函数是如何实现的。这样，开发 C 系统的人只做了一次工作，就使所有使用该系统编程序的人都节省了大量时间和精力。由此可以明显看到函数的意义和作用。

C 标准库函数完成一些最常用的基本功能，包括基本输入和输出、文件操作、存储管理，以及其他一些常用功能函数，如数学函数、数据值的类型转换函数等。对这些函数的介绍散布在本中各个章节里。第 11 章包含对标准库其他重要函数的介绍。至于具体 C 系统的扩充函数库，就需要查阅系统联机帮助材料、系统手册或其他参考书籍。学习本课程时也应学会使用手册和联机帮助材料，学会如何阅读它们。

下面介绍两组简单函数。

5.2.2 字符分类函数

首先介绍标准库文件 `ctype.h` 描述的各种字符分类函数。这些函数很简单，它们对满足条件的字符返回非 0 值，否则返回 0 值。下面是有关函数：

`isalpha(c)`	c 是字母字符
`isdigit(c)`	c 是数字字符
`isalnum(c)`	c 是字母或数字字符
`isspace(c)`	c 是空格、制表符、换行符
`isupper(c)`	c 是大写字母
`islower(c)`	c 是小写字母
`iscntrl(c)`	c 是控制字符
`isprint(c)`	c 是可打印字符，包括空格

isgraph(c)	c 是可打印字符，不包括空格
isxdigit(c)	c 是十六进制数字字符
ispunct(c)	c 是标点符号

要使用这些函数，应当在程序前部用#include 命令包含系统头文件 ctype.h。在这个头文件里还说明了两个字母大小写转换函数：

int tolower(int c)	当 c 是大写字母时返回对应小写字母，否则返回 c 本身
int toupper(int c)	当 c 是小写字母时返回对应大写字母，否则返回 c 本身

例如，下面程序统计文件中数字、小写字母和大写字母的个数，其中使用了标准库的几个字符分类函数。采用标准库函数的做法比自己写条件判断更合适，值得提倡。

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

int main()
{
    int c, cd = 0, cu = 0, cl = 0;
    while ((c = getchar()) != EOF) {
        if (isdigit(c)) ++cd;
        if (isupper(c)) ++cu;
        if (islower(c)) ++cl;
    }
    printf ("digits: %d\n", cd);
    printf ("uppers: %d\n", cu);
    printf ("lowers: %d\n", cl);
    return 0;
}
```

5.2.3 随机数生成函数

计算机程序实现的都是确定性的计算：给一个或者一组初始数据，它总计算出一批确定的结果。然而计算机应用中有时也需要带有随机性的计算。

一个例子是程序调试。在程序调试时，人们需要用各种数据进行程序运行试验，看能否得到预期结果，有时用随机性数据作为试验数据是很合适的。另一个应用领域是计算机模拟，也就是用计算机模拟某种实际情况或者过程，以帮助人认识其中的规律性。客观事物的变化中总有一些随机因素，如果用确定性数据进行模拟，多次模拟得到的结果完全一样，将无法很好地反映客观过程的实际情况。

由于这些情况，人们希望能用计算机生成随机数。实际上，计算机无法生成真正的随机数，通过计算只能生成所谓的伪随机数。如何用计算机生成随机性比较好的随机数仍是人们研究的一个问题。最常用的随机数生成方法是定义一种递推关系，通过这个递推关系生成一个数的序列，还要设法使这个序列中的数看起来比较具有随机性。

最常用的简单递推关系是通过除余法定义的关系：

$$\alpha_0 = A, \quad \alpha_n = (B \times \alpha_{n-1} + C) \bmod D \quad \text{对 } n > 0$$

这里 A, B, C, D 都是整常数，$0 \le A < D$。通过很好地选择常数 B, C，可以产生出值位于 0 到 $D-1$ 范围中的比较好的随机数列。

自己定义随机数的生成函数的问题留到后面讨论，这里介绍 C 标准库提供的随机数功能。要使用标准库的随机数生成功能，程序前部应包含系统文件 stdlib.h，这个文件里描述了与随机数生成有关的函数：

```
int rand(void)
```

这是一个无参函数，每次调用将得到一个新随机整数，其值在 0 和系统定义的符号常量 RAND_MAX 之间。不同系统里的 RAND_MAX 可能不同，一般系统中用 32767。

```
void srand(unsigned seed)
```

这个函数用参数 seed 的值重新设置种子值，即为生成下一个随机数而保存的一个整数值

（由它出发递推，取得下一个随机数）。默认的初始种子值是 1。

举个例子。下面程序先打印出由默认种子值出发的 10 个随机数，而后打印设定了新的种子值 11 后生成的 10 个随机数：

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

int main()
{
    int i;
    for (i = 0; i < 10; ++i) printf ("%d ", rand());
    putchar('\n');
    srand(11);
    for (i = 0; i < 10; ++i) printf ("%d ", rand());
    putchar('\n');
    return 0;
}
```

程序里的 putchar('\n');使输出换一行。在某个系统里这个程序输出：

```
41 18467 6334 26500 19169 15724 11478 29358 26962 24464
74 27648 21136 4989 24011 22223 9834 85 28238 28519
```

5.3 函数定义和程序的函数分解

无论系统提供多少库函数，其数量终归有限，编程时总要考虑定义自己的函数。一个 C 程序主要由一系列函数定义组成。每个函数定义包含一段程序代码，执行时将完成一定工作。定义函数时给定了一个名字，供调用这个函数时使用。函数定义的基本形式是：

返回值类型　函数名　参数表　函数体

*返回值类型*描述函数执行结束时返回的值类型；*函数名*用标识符表示，主要用于调用这个函数；*参数表*描述函数的参数个数和各参数的类型；*函数体*是一个复合语句，描述被这个函数所封装的计算过程。*函数体*之前的部分称为*函数头部*，它描述了函数外部与函数内部的联系，下面要着重讨论这部分的问题。例如，下面是一个我们熟悉的函数：

```
long fact (int n) {
    int fac, i;
    for (fac = 1, i = 1; i < n; ++i)
        fac *= i;
    return fac;
}
```

返回 long 值，函数名为 fact，参数表里只有一对参数描述（一个类型名和一个参数名）。

定义函数时可以不写*返回值类型*，这表示返回 int 类型的值。这种做法不应提倡，因为它容易引起错误（未来的 C 系统将禁止不写类型的形式）。我们也可以定义不返回值的函数，这时用关键字 void 说明“返回值类型”。这种写法很别扭，是 C 语言把两类东西（有返回值和无返回值的函数）用同一形式写出而带来的副作用。

一个函数可以有任意多个参数，各参数描述用逗号分隔。每个参数描述包括一个类型名和一个参数名。函数定义的参数表里给出的参数名称为函数的*形式参数*，简称*形参*。定义无参函数时参数表应写成 () 或者 (void)。后一写法很不自然，是早期 C 语言的遗留问题对新 ANSI C 标准的影响，我们只能接受。没参数的函数又称*无参函数*。

5.3.1 主函数

每个 C 程序里总有一个名为 main 的特殊函数，常称为*主函数*。主函数规定了整个程序执行的起点，专业术语是*程序入口*。程序执行从这个函数开始，一旦它执行结束，整个程序就完成了。程序里不能调用主函数，它将在程序开始执行时被自动调用。

C 语言规定主函数的返回值必须是 int 类型。有些书里的程序示例没描述 main 的返

回值类型，按上面所说，这正说明其返回值为 int。主函数的返回值不会在本程序内部使用（因为 main 不能在程序内调用），这个值将在程序结束时提供给操作系统。在程序外部，例如操作系统，可以检查和使用程序的这个返回值。

写 C 程序时应为主函数定义返回值，一般用返回值 0 表示程序正常结束，用其他值表示执行中出现了非正常情况。按语言规定，在主函数结束时如果没有提供返回值，程序将自动产生一个表示成功完成的返回值（通常就是 0）。一些 C 系统会对主函数返回值的情况产生警告，这是不对的，但我们也不必戒意。这样 main 函数的样子就是：

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

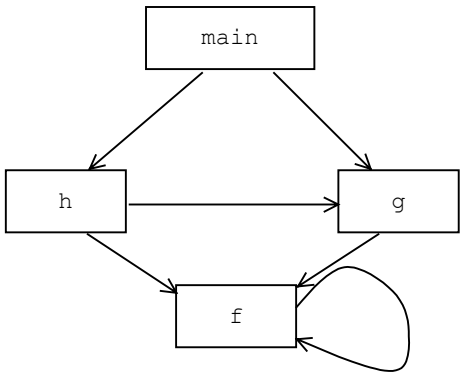
除了主函数外，程序里的其他函数只有在明确调用时才能进入执行状态。所以，一个函数要在程序执行过程中起作用，那么它或是被主函数直接调用的，或是被另外一个能被调用执行的函数调用的。没有被调用的函数在程序执行中不会起任何作用。

5.3.2 程序的函数分解

在编写大些的程序时，应该特别注意程序的功能分解，在这里也就是函数分解。也就是说，应该把程序写成一组较小的函数，通过这些函数的互相调用完成所需要的工作。初学者往往不注意函数分解，一些教学材料或书籍中对这个问题强调得也很不够，给出的程序例子经常是一大片，没有结构性，初学者不良编程习惯的形成往往与此有关。实际上，在学习程序设计的过程中强调函数分解是绝对必要的，没有合理的函数分解，完成规模较大的程序将更困难，要花费更多时间，写出的程序通常也更难理解，出现了错误更难发现和改正。这一点值得读者特别注意。

一般说，一个 C 程序由一组函数构成，图 5.1 显示了一个 C 程序的结构和执行中的调用关系。左边是程序的概貌，其中除主函数外还定义了三个函数，这些函数互相调用。右边图形显示了调用关系，矩形表示函数，箭头表示函数调用。递归调用（函数 f 调用自己）表现为到自身的箭头。

```
#include ...
int f(...) {
    ... f(...) ...
}
int g(...) {
    ... f(...) ...
}
void h(...) {
    ... f(...) ...
    ... g(...) ...
}
int main(void) {
    ... h(...) ...
    ... g(...) ...
}
```



这里的箭头表示函数调用关系

图 5.1　源程序及其确定的函数调用关系示意图

问题是：什么样的程序片段应当定义成函数呢？这并没有万能的准则，程序设计者需要自己分析问题，总结经验。这里提出两条线索，供读者学习时参考：

1. 程序中可能有重复出现的相同或相似的计算片段。可以考虑从中抽取出共同的东西，定义为函数。这将使一项工作只有一个定义，需要时可以多次使用。这样做不但可能缩短程序，也将提高程序的可读性和易修改性。
2. 程序中具有**逻辑独立性**的片段。即使这种片段只出现一次，也有可以考虑把它们定义为独立的函数，在原来需要这段程序的地方写函数调用。这种做法的主要作用是分解程序复杂性，使之更容易理解和把握。例如，许多程序可以分解为三个主要工作阶段：正式工作前的准备阶段，主要工作阶段（通常这里有复杂的循环等），完成前的结束阶段。

把程序分解为相应三个部分，设计好它们之间的信息联系方式后，就可以用独立的函数分别实现了。显然，与整个程序相比，各部分的复杂性都更低了。

很难说什么是一个程序的最佳分解。对一个程序可能有许多种可行分解方式，寻找比较合理或有效的分解方式是需要学习的东西。熟悉程序设计的人们提出的经验准则是：**如果一段计算可以定义为函数，那么就应该将它定义为函数。**

```
    *            *
   * *          ***
  *   *        *****    ***********
 *     *      *******   *         *
*       *     *******   *         *
 *     *      *******   *         *
  *   *        *****    ***********
   * *          ***
    *            *
```

图 5.2　字符图形

5.3.4 对函数的两种观点

一个实例：字符图形

假定要做出一些字符拼出的几何图形，如图 5.2 中那样的菱形、六边形和矩形，应该如何写程序呢？当然可以直接写许多输出语句打印菱形，另写一个程序打印六边形，等等。但如果要输出其他图形，或者要改变图形的大小，原来写的程序几乎就没用了。在实际工作中，程序的需求经常改变和扩充，因此，写程序时必须关心程序的修改和扩充问题。函数在这里能扮演重要角色。另外，重复描述许多类似的输出语句也很烦，既没有意思也容易出错。

现在考虑定义几个函数实现画这类图形的基本功能，而后通过调用这些函数画出所需要的具体图形。为了考虑这些函数，需要首先分析这类图形的性质。这里提出对问题的一种分析（完全可以有其他合理分析，下面另有说明）：图形中每一行有两种情况，一种是从某个位置开始的一段连续字符；另一种是在两个特殊位置输出字符。将这两种情况看着基本操作，可以考虑定义两个函数，其头部分别为：

```
void line(int begin, int end)
void points(int first, int second)
```

第一个函数从位置 begin 到 end 输出一系列星号，第二个函数在 first 和 second 两处输出星号。考虑到字符图形未必总用星号，我们也可以推广定义，引进一个字符参数：

```
void line(char c, int begin, int end)
void points(char c, int first, int second)
```

虽然这两个函数还没有定义，但由于它们的功能已经很清楚，现在就可以利用它们写画图形的函数了。例如，下面语句将画出一个三角形（假定所用变量已有定义）：

```
for (i = 10, j = 10; i > 5; --i, ++j)
    points('*', i, j);
line('*', 5, 15);
```

画其他空心或者实心的规范图形也不困难，留给读者作为练习。由这些可以看出，如果作为函数的使用者，我们只需考虑函数的使用形式和函数的功能，考虑如何基于此完成所需工作。有关功能的具体实现则不是使用函数时需要考虑的问题（也不应该考虑）。

下面转到函数实现者的角度，考虑如何定义这两个函数。这些函数的定义不困难，只是输出适当的空格并在适当位置输出字符 c。下面是这两个函数的定义：

```
void line(char c, int begin, int end) {
    int i;
    for (i = 0; i < begin; ++i) putchar(' ');
    for ( ; i <= end; ++i) putchar(c);
    putchar('\n');
}

void points(char c, int first, int second) {
    int i;
    for (i = 0; i < first; ++i) putchar(' ');
    putchar(c);
    for (++i ; i < second; ++i) putchar(' ');
    if (first < second) putchar(c);
```

```
        putchar('\n');
    }
```

函数里就是一些简单输出语句。这里按照习惯将屏幕的首列作为第 0 列。函数 points 的定义里有一个检查，只在 first 小于 second 时才输出第二个字符。

有了这两个基本函数，我们就很容易做出各种图形了。如果已经做出了一个图形，在必要时也不难修改其大小或者形状。进一步说，我们也可以基于它们定义出一组描画各种基本几何图形的函数。在设计这些函数时，需要选定几个合适的参数。例如，可以定义如下两个画矩形的函数（这里只给出函数头部）：

```
void rect(char c, int begin, int len, int high) { ... ... }
void rect_fill(char c, int begin, int len, int high) { ... ... }
```

也可以通过引入一个附加参数的方式将两个函数合而为一：

```
void rect(char c, int begin, int len, int high, int fill) { ... ... }
```

在参数 fill 为 0 值时画出空心矩形，非 0 时画实心矩形。

不难看出，这些函数又提供了另一层次的功能分解。定义好这样一组函数之后，就可以在另一个层次上写绘制字符图形的程序了。显然，上述分解只是一种可行设计，它有优点也有缺点。我们完全可以考虑其他函数分解。例如，将基本作图函数定义为：

```
void syms(char c, int n) { ... ... }
```

其基本功能就是输出 n 个字符 c。基于这一简单函数同样可以实现各种图形绘制函数。

函数封装和两种观点

函数是封装起来并给以命名的一段程序代码，是程序中具有逻辑独立性的实体。函数需要定义，又能作为整体在程序中调用执行，完成其代码所描述的工作。这些情况引出了对函数的两种观点（两种观察角度）：从函数之外（从函数使用者的角度）看函数，以及在函数内部（以定义者的角度）看函数。看到两者的差异对于认识函数，考虑与函数有关的问题都非常重要。图 5.3 直观地反映这里的问题。

一个函数封装把函数里面和外面分开，形成了两个分离的世界——函数的内部和外部，站在不同的世界里看问题，就形成了对函数“内部观点”和“外部观点”。函数头部的描述规定了函数内外间的交流方式和通道，定义了内部和外部都需要遵守的共同规范。

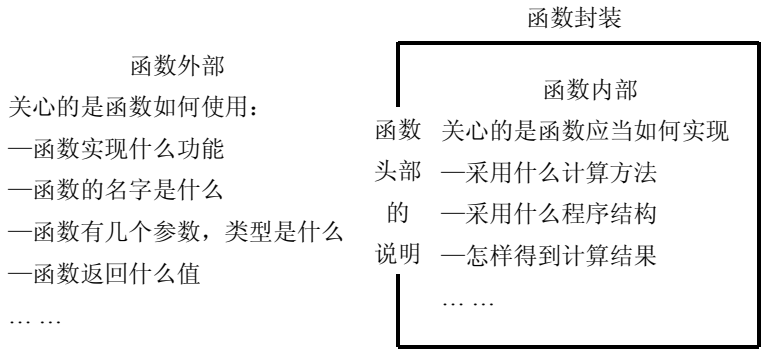


图 5.3　对函数的两种观点

从外部看，一个函数实现了某种功能。只需知道它的名字和类型特征等。调用函数时遵从这些规定，提供数目和类型适当的实参，正确接受返回值，就能得到预期的计算结果。

在函数之外不应该关心函数功能的实现问题。这种超脱很重要，不掌握这种思想方法就无法摆脱琐碎细节的干扰，不能学会处理复杂问题。初学者常犯的一个毛病是事事都想弄清。这种考虑不但常常不必要，有时甚至不可能。例如，对标准库函数，我们不知道它们是用什么语言写的，但这并不妨碍在程序中使用它们。

内部观点所关心的问题当然不同。这时的重要问题包括：函数调用时外部将提供哪些数据，各为什么类型（由参数表规定）；如何用这些参数完成所需计算，得到所需结果（算法问题）；函数应在什么情况下结束？如何产生返回值（返回语句如何写）？在考虑函数实现时，不应去关心那里将调用这个函数等。

函数头部的重要性就在于它构成了函数内部和外部之间的联系界面，函数外部和内部通过这个界面交换信息，达到函数定义和使用之间的沟通。在定义函数之前首先应有全面考虑，据此定义好函数头部，规定好公共规范。此后人的角色就分裂了，应根据是定义函数还是使用函数来观察和解决问题。实际上，一旦弄清了函数功能，描述好函数头部后，函数的定义和使用完全可以由两个或两批人做。只要他们遵循共同规范，对函数功能有共同理解，就不会有问题。大型软件的开发中经常需要做这类功能分解。注意，上面两句话很重要："遵循共同规范"，"对于函数的功能有共同理解"，人经常在这里出现偏差。我们自己写程序时也必须注意，应保证对同一函数的两种观点间的内在一致性。

函数定义

C 程序中，函数的形式参数也是函数的局部变量，在其他局部变量定义之前就有了定义，它们的初值由函数调用时的实参（表达式）取得。形参在函数体内的使用方式与其他局部变量完全一样，也可以重新给它们赋值。

函数被调用执行后，顺序执行函数体内的语句序列。return 语句在函数体中起着特殊的作用，任何 return 语句的执行将导致本函数的本次执行结束。如果本函数由另一函数调用，函数结束将使执行过程返回到那个函数里的调用点，计算从该调用点后面继续下去。如果这个函数就是主函数，函数结束就是程序执行的结束。

return 语句有两种不同形式：

```
return;
return 表达式;
```

分别用在无返回值和有返回值的函数里。无*表达式*的 return 语句简单导致函数结束。有返回值的函数必须用带*表达式*的 return 语句，执行到这里时先求出*表达式*的值，并将这个值转换到函数的返回值类型后作为函数返回值。显然，这就要求*表达式*的类型能转换到函数定义的返回值类型。一个函数里可以有多个 return 语句。所有 return 语句在带不带表达式，所带表达式的类型方面都应当与函数头部一致。

函数结束的一种情况是执行到达函数体的最后。函数以这种方式结束时返回值无定义，所以这种结束方式只能出现在无返回值的函数中。例如前面的 pc_area 就是这种情况。

函数调用

函数调用的形式是函数名后跟圆括号括起、逗号分隔的若干表达式，这些表达式称为*实际参数*，简称*实参*。调用函数时必须提供一组个数正确、类型合适的实参。调用无参函数时需要写一对空括号。无返回值的函数通常用在单独的函数调用语句里，如：

```
pc_area(x + 3);
```

有返回值的函数一般出现在表达式里，用其返回值参加计算。C 语言允许不用函数的返回值（即使它有）。例如，前面多次使用的 printf 实际有一个 int 返回值，工作正常完成时，其返回值是执行中实际输出的字符个数；函数执行中出错时返回负值。在前面例子里从来未用过这个返回值，这时返回值就被简单地丢掉了。

C 语言的参数机制称为*值参数*（简称*值参*）。函数调用时计算各实参表达式的值并分别送给对应形参，而后执行函数体。函数内对形参的赋值与实参无关。即使实参是变量，对形参的赋值也不会改变实参的值。图 5.3 显示了实参与形参的关系，f(m,n)执行时，实参 m 和 n 的值分别送给形参 a 和 b，f 内部对 a 和 b 的操作与 m、n 再也没有关系了。

函数调用中还有一个重要问题必须引起重视。对于多个参数的函数，C 语言没有规定调用时实参的求值顺序，任何依赖实参求值顺序的调用都得不到任何保证（这与对二元算术的运算对象的情况一样）。例如，下面是一个错误的函数调用：

```
n = ...;
m = gcd(n += 15, n);
```

因为在这个调用中，对第一个参数的求值将影响第二个参数的值。下面是另一个错误调用：

```
printf("%d, %d", n++, n);
```

请不要在程序里写这种语句。

另外，实参表达式求值后需要传入函数，这时可能产生隐含的的类型转换动作。为使实参到形参的值传递能进行，就要求所需转换合法，否则编译时将出现类型错误。

5.3.5 函数原型

在 C 程序里，每个有名字的程序对象（变量、函数都是程序对象）都有定义点和使用点。一般说，一个对象只应有一个定义点，可以有多处使用点。为保证使用与定义的一致性，通行的规则是应当“先定义后使用”。以函数的局部变量为例，要求变量定义出现使用变量的语句之前，这就保证了它们的先定义后使用。

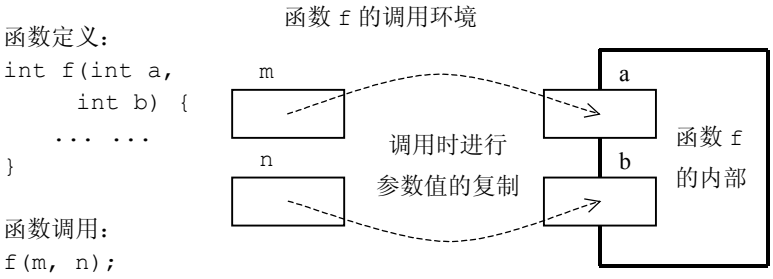


图 5.3　函数调用与参数值的传递

规定“先定义后使用”，是因为对象的使用方式依赖于它们的性质。如果没有定义在先，就难以知道使用是否正确。为保证语言系统能正确处理程序，基本原则是：**保证从每个对象的每个使用点向前看，都能得到与正确使用该对象有关的完备信息。**

在函数的使用点，需要信息就是函数的类型特征，包括函数名，参数个数和类型，函数返回值类型。调用处需要检查参数个数是否正确，各参数的类型是否与函数定义一致，如果不一致能否转换（必要时插入转换动作）等。由于返回值可能参加进一步计算，也要做类似处理。看不到函数的类型特征，就无法正确完成这些检查和处理。

函数原型

前面许多程序实例里都定义了函数。我们一直把主函数写在最后，就是为保证调用点与使用点之间有合适的相对位置。在函数的递归定义中，这一要求仍能满足。例如：

```
long fact(long n) {
    return n <= 0 ? 1 : n * fact(n - 1);
}
```

在函数体里的递归调用点能看到函数头部描述的类型特征。

也存在一些情况，其中无法通过安排函数定义位置的方法解决调用点与使用点间的信息交流关系。典型例子是两个互相调用的函数定义，例如需要写下面两个函数定义：

```
double h(double x) {            double g(double y) {
    ....                            ....
    ... g(...) ...                  ... h(...) ...
    ....                            ....
}                               }
```

无论怎样安排顺序，都无法保证函数的每个使用点都能出现在相应函数的定义点之后。这类情况的存在是 C 语言引进*函数原型说明*的一个重要原因。另外的原因是人们希望能自由地安排函数定义的位置，支持大规模程序的开发，有关情况在后面章节介绍。

函数原型说明（有时也简单地说*原型*或*函数原型*）在形式上与函数头部类似，最后加一个分号。原型说明中参数表里的参数名可缺（可以只写类型）。即使在这里写参数名，所用名字也不必与函数定义用的名字一致。原型说明里的参数名可以起提示作用，也提倡给出有意义的名字，这将有利于函数的正确使用。另外，如果希望写注释来说明函数的作用，有了参数名也更容易描述。下面是前面定义的两个函数的原型说明：

```
void line(char c, int begin, int end);
void points(char c, int first, int second);
```

函数原型可以出现在任何可以写定义的地方。目前人们认为最合理的方式，是把原型说明都放在源程序文件最前面。这样，本程序文件中所有的函数使用点都可以“看到”这里的原型说明。更重要的是，这样做能保证文件里的各处看到的是同一个原型说明，有利于保证同一函数的所有调用之间的一致性。另一方面，这使函数的定义点也能看到这个原型说明，编译程序会检查两者间的一致性，出现不一致时会产生明确的编译错误。这正是我们所希望的：以函数原型作为媒介，保证函数定义和使用之间的一致性。

现在就可以处理上面互相递归定义的例子了。例如可以如下安排（反过来也行）：

```
double h(double);
....
double g(double y) {
    .... h(...) ....
}
double h(double x) {
    .... g(...) ....
}
```

应特别强调：为了保证函数原型真正能起作用，写原型时必须给出完整的类型特征描述。不注意这个忠告可能吃大亏，可能使错误遗留在程序里，到调试和试验运行时遇到更大麻烦。许多材料和书籍里没有这样做，那是很不合适的，希望大家不要效仿。

过时的函数定义形式与原型形式

C 语言是发展的产物，其标准化不得不保留一些过时东西，以保证过去的程序仍能由符合标准的编译系统加工。虽然如此，今天学习 C 语言时还是应该避免使用过时描述方式。ANSI C 标准指出，过时描述方法将逐步淘汰，我们完全没理由去用过时形式。希望读者能始终坚持正确的新的程序书写形式，这也是本书始终强调的。这里介绍过时描述形式只是为了完整。因为读者在参考其他书籍、阅读已有程序时，难免会遇到过时的 C 语言描述方式，因此需要了解这方面的情况。但希望读者绝不要使用这些形式*。

过时的函数定义形式把参数的类型说明另外给出在参数表后面。例如：

```
double f1(x, y, n)
double x, y;
int n;
{
    ... ... /* 函数体 */
}
```

这种方式的重要缺点是参数表和类型说明分离，增加了维护一致性的负担。这种方式在一些早期程序语言中采用，新的程序语言都已抛弃了这种过时形式。

过时原型说明形式的危害更大。过时原型说明在形式上是不描述参数类型，只写函数名和一对括号。有人还常把这种说明与变量定义等写在一起。实际上，这种“简写”原型说明是许多 C 程序错误的根源。请读者务必不要使用这种形式，不要到吃了大亏，自己有了切身体会后才幡然醒悟。

下面是一个例子。假设现在有两个函数，其完整原型分别是：

```
double f(double, int);
int g(double);
```

假设函数在后面（或其他地方）定义，调用它们的片段（注意其中的过时形式）是：

```
int main (void) {
    int g(), n;
    double f(), x;
    ....
    x = g(3);
```

* 现在还能看到许多过去写出的 C 语言的参考书和教科书，以过时形式给出程序实例和说明解释，其中的许多东西早已被 ANSI C 标准否定。请读者阅读参考时注意。

```
        n = f(n);
        ....
    }
```

编译程序将不会发现这个片段有错误，而实际计算结果根本**不可能**正确。这里的错误很多：首先，过时形式的 f 和 g 原型告诉编译系统不检查函数参数，因此 f(3) 中实参 3 不经转换就送给要求 double 参数的 g（不做转换），显然 g 不可能正确完成工作。函数调用 f(n) 的情况更可怕，编译系统甚至根本就没有发现参数个数不对，当然也不会考虑参数类型问题。这样的程序怎么可能得到有意义结果呢？

人很容易犯错误，写复杂的程序时不可能不出错，因此需要编译程序帮助。不良书写方式将阻挠编译程序对程序里错误的检查，最终当然还是写程序的人自己吃到苦头。

还有一点值得提出。为保证编译程序的正确检查，在写无参函数的原型时，参数表必须写成(void)，不能简单写成()，因为这一形式将被看着是过时原型形式。

不写函数原型

为了宽容老程序，C 语言也容许不写原型说明，为此提供了默认处理方式。但不写原型比写过时原型更危险，绝不应该用这种偷懒的方式。编译程序在遇到调用 f(...)时，如果此前没有看到函数 f 的定义或原型说明，那么它就假定有原型说明：

```
int f();
```

出现在包含该调用的最内层复合语句前部，按这个假设处理该函数调用。不难看出，只有当 f 确实返回 int 值，而且.调用的参数个数和所有实参类型都与 f 的要求完全一样（不需要任何转换）时，这种处理方式才能产生正确结果。

举一个使用标准库函数的错误例子，下面程序的编译链接通常都不会出错，但运行时的输出绝不会是正确的，第一个结果不会 1 的正弦函数值，第二个结果可能更奇怪：

```
#include <stdio.h>

int main()
{
    double x, y;
    float u = 1.57;
    x = sin(1);
    y = sin(u);
    printf("Error output: %f, %f\n", x, y);
    return 0;
}
```

纠正的方法就是加上 #include <math.h> 命令行，因为编译程序看不到 sin 的原型，就不会做正确转换。这个例子也说明了一般情况下应该用 double 的另一个原因。

请读者分析，假设前节例子里 main 开始处没给出两个函数的原型说明，编译程序将怎样处理，会出现什么错误。不写函数 f 的原型说明可能引起各种问题，如：

1. 如果编译程序后来遇到了 f 的定义，其返回值类型与它所做的假设不一致，编译程序有可能给出“函数重新定义”的错误信息。
2. 假设 f 在其他源文件定义，或为库函数，而且 f 的类型返回类型与默认假设不符，那么编译不会发现错误，连接时也不检查，产生的可执行程序在执行时则可能出错。
3. 如果函数调用的实参（个数或类型）与函数定义不一致，编译时不会发现错误，也不会自动生成类型转换，连接时也不检查。最终形成可执行程序里的语义错误。

我们应坚持的正确原则是：1）如果使用库函数，那么就必须在源文件前部用#include 命令包含必要的头文件（xxx.h 文件）。2）对所有未能在使用前给出定义的函数（无论它是定义在本文件后面，还是在其他源文件里），都应给出正确完整的原型说明。3）把原型说

明写在源文件最前面（不要写在函数内部），以使函数的定义点和所有使用点都能“看到”同一个原型说明。如果坚持了这些原则，就能避免函数调用与定义不一致的错误。

5.4 C 程序结构与变量

一个 C 语言源文件由一系列外部定义和外部说明构成。*外部定义*、*外部说明*是指写在源程序文件表层的定义和说明（不在函数体内）。函数定义是一种外部定义，写在外层（不在函数体内）的原型说明是一种外部说明。一个外部定义或者说明总从它出现的位置开始起作用，其作用范围一直延续到文件结束。

C 函数只能在外部定义（不允许在函数内定义函数），所有函数都定义在表层，程序的结构比较简单。本节将讨论由各种定义（或说明）的出现位置引起的语义问题，这里以变量定义作为讨论对象，其他定义或说明，如果既允许出现在外部也允许出现在函数内，在起作用的范围方面与变量定义一样。这里将涉及一些重要概念，其中最重要的就是*作用域*和*存在期*。#include 行及其他类似行的地位特殊，有关问题将在下节讨论。

5.4.1 外部定义的变量

以外部定义形式定义的变量称为*外部变量*或*全局变量*，它们的性质与函数内部定义的变量（局部变量）不同。外部变量在可使用性方面与函数类似，原则上可以在整个程序的任何地方使用。由于变量定义从定义点开始起作用，因此人们常把外部变量定义写在源文件里所有函数定义之前。这样，该源文件的任何函数就都可以访问这些外部变量了。

与函数定义类似，外部变量也允许后定义先使用，或在一个源文件里定义而在其他文件里使用。如果有这种需要，就必须在使用这些变量前给出有关的*外部变量说明*。这种说明在形式上与变量定义类似，但要在最前面增加关键词 extern。例如可以写：

```
extern int number, length;
```

这表示：下面要使用在其他地方定义的整型外部变量 member 和 length。这种说明并不是定义，程序里必须有另外的地方定义了这两个变量。人们一般也把这种说明放在源文件最前面*。这样，被说明的外部变量就可以在整个源文件中使用了。这种做法的另一优点是能保证整个源文件参照着同一个外部变量说明，保证各处使用的一致性。

例如，下面程序段里定义了外部变量 num。还有一个外部变量说明，说明要用其他地方定义的整型变量 exnumber。随后的函数里使用了它们：

```
... ...
int num;
extern int exnumber;
int f(int n) {
    int n;
    ... n ... num ... extnumber ...
    ... ...
}
int main (void) {
... f(...) ...
... num ... exnumber ...
    ... ...
}
```

也应注意外部变量引起的一种情况。如果一个函数的定义里访问了某个外部变量，那么这个函数就对该变量就有了依赖性。例如上例里的 f，它已不再是独立的函数了，不能孤立地把它的定义复制到另一个程序里使用。如果想在其他地方使用这个函数，函数的环境里必须有同样的外部变量。

---

* 另一常用方式是把外部变量说明和函数原型说明等写在独立的头文件里，这种文件通常用 h 为扩展名。在需要用这些说明的源程序文件里用#include 命令包含有关头文件。这种方式将在后面讨论。

这里还要提出两点说明：

1. 定义和说明这两个术语的意义不同。定义要求建立被定义对象，而说明仅指明有这个东西存在，被说明的东西必须在其他地方建立（定义），否则这个说明就是无效的、没有意义的。对于变量，定义与说明的差异表现得非常明显。程序执行到变量定义处时将建立被定义的变量，为其分配存储值的位置。而变量说明只是通告“有这样一个变量存在，其性质是……”，这里并不实际建立变量，也没有存储安排问题。
2. 外部变量可以在整个程序里使用，这也就意味着在整个程序内不能有重名的外部变量。否则目标文件连接时就会出现问题。由于连接时还要用到 C 语言的程序库，因此在编写程序时还要注意，所定义程序对象的名字不要与程序库里的名字冲突。

5.4.2 作用域与生存期

一个变量定义实际上同时说了几件事：1）定义了一个具有特定类型的变量，它可保存这种类型的值；2）给变量命名。实际上，这个变量定义还确定了两个问题：

1. 在程序中的哪个范围内该变量定义有效，即在哪个范围里可以用这个变量名去说与该变量有关的事情，包括取值和赋值。每个变量定义都有一个确定的作用范围，这个范围称为该变量定义的作用域，变量的作用域由变量定义的位置确定。
2. 变量的实现基础是内存单元。变量在程序运行中建立（程序结束时其中定义的所有变量都撤消，下一次执行不能去使用上次执行结束时各变量的值），实际上，各种变量存在的时间也可能不同。一个变量在程序执行中存在的那段时期称为这个变量的存在期。有关变量存在期的规定更复杂一些。

作用域和存在期是程序语言中的两个重要概念，弄清楚它们，许多问题就容易理解了。作用域和存在期有联系但又不同。作用域讲变量定义的作用范围，说的是源程序中的一段，可以在代码中划清楚，是静态概念。存在期则完全是动态概念，讲的是程序执行过程中的一段期间。变量的存在期里一直保持着自己的存储单元，只要不赋值，存于这些单元中的值就保持不变。在作用域和存在期这两方面，外部变量和函数内的普通变量（C 语言称为自动变量）的性质截然不同。

5.4.3 外部变量和自动变量

外部变量定义的作用域是整个程序，这意味着可以程序中任何地方通过外部变量名使用它们。函数内定义的变量的作用域是有关定义所在的复合语句，在这个复合语句之外该定义无效。因此说函数内定义的变量是局部的。另外，前面提过，函数形参都看作函数定义的局部变量，作用域就是这个函数的函数体。

这方面的一个常见错误认识与 main 函数有关。初学者知道 main 的特殊地位，有时就误以为 main 里定义的变量可以在其他函数使用。从作用域角度看，main 也是普通函数，其内部定义也是局部的，同样不能在函数体之外使用。

在存在期方面，外部变量的存在期是程序的整个执行期间。也就是说，在程序执行开始时，所有外部变量都已有定义，有了确定存储位置。外部变量的这种有定义状态一直延续到程序结束，它们与对应存储位置的关联也保持保持不变。

全局变量定义在函数之外，一个函数可以把公共数据存入这种变量，另一函数就可以直接使用它。因此，全局变量可以看作函数间交换数据的一种通道，利用它们交换数据，程序写起来直截了当。后面章节里有这方面的例子。

自动变量的情况完全不同。在复合语句里定义的自动变量的存在期就是这个复合语句的执行期间。也就是说，该复合语句开始执行时建立这里定义的所有变量。它们一直存在到该

复合语句结束。复合语句结束时，内部定义的所有变量都撤消。如果执行再进入这一复合语句，那么就再次建立这些变量。新建变量与上次执行建立的变量毫无关系，是另一组变量，正是由于这种变量被自动建立和撤消的性质，C 语言中把它们称作自动变量。

看下面函数定义：

```
int fun (int n) {
    int m, i;
    for (m = i = 0; i < n; ++i) {
        int k = m + 1;
        m = k + i * i;
    }
    return m;
}
```

在函数 fun 被调用时，首先建立 n 并用实参给它初值，而后顺序建立变量 m 和 i。在函数里的每次进入循环体时建立变量 k 并给它初值，循环体执行结束时撤销 k。函数 fun 的局部变量 n，m 和 i 生存到函数返回为止。如果下次再调用 fun，又会重新建立名字为 n，m 和 i 又一组变量，在函数 fun 的新的一次执行中使用。

假设有下面程序段：

```
for (n = 0; n < 10; n++) {
    int m;
    if (n == 0) m = 2;
    /* 循环执行第二次到达这里时m的值无法确定 */
    ....
}
```

按上面解释，每次循环体开始执行时建立一个名为 m 的新变量。第一次循环时，由于 n 值为 0，m 赋值 2。但在第二次及其后的循环执行中条件不成立，相应赋值语句不执行，这样到了写注释的位置，m 的值仍不能确定。注意这里所说“值不能确定”的含义，由于新建立的 m 没有赋过值，我们对它这时的值没有任何保证。

上面讨论实际提出了 C 程序的两种作用域：一种作用域是整个程序，称作全局作用域；另一种作用域由复合语句确定，每个复合语句确定了一个作用域，这是局部作用域。全局作用域是所有外部定义（外部变量定义、外部函数定义等）的作用域，而局部定义的作用域是局部的。C 语言还有一种以源文件为单位的作用域，这将在下面讨论。

此外，一个变量在不在作用域里与能不能使用它并不是同一件事。例如，假设一个外部变量定义出现在程序文件后面某处，在此之前的程序中就无法直接使用这个变量（因为看不到它的定义或说明），如果要用这种变量，那就必须把变量的定义前移，或者在前面另加外部变量说明。

5.4.4 变量定义的嵌套

多种作用域的存在造成了作用域的嵌套。作为函数体的复合语句嵌套在全局作用域里，形成了作用域嵌套。复合语句里还可以有嵌套的复合语句，形成作用域的进一步嵌套。函数定义只能出现在全局作用域里，不会出现嵌套定义的问题。变量则不同，变量定义可以出现在任何复合语句里面，其作用域就有了出现嵌套的可能性。

这方面的基本规定是：同一作用域里不允许定义两个以上同名变量，也就是说，作用域相同的变量的名字不能冲突。否则使用哪个变量的问题就无法确定了。

不同作用域中定义同名变量不但是容许的，也是人们经常做的。例如下面程序段的主函数和函数 f 里都定义了名字为 x 和 n 的变量：

```
int f(...) {
    double x, n;
    ....
}
```

```
main (void) {
    double x, z;
    int n, m;
    ....
}
```

这没有问题，无论同样类型的 x，还是不同类型的 n，两对变量虽然同名但互不相干（它们的作用域不同），只是两个变量恰好用了同样名字。

另一种情况需要特别提出，这就是在嵌套的两个作用域里出现了同名变量定义，这时两个变量之间有什么关系呢？例如，下面程序段里有两个 x 的定义：

```
{   int x, y;
    ....
    while (....) {
        double x;
        ..x..
    }
    ....
}
```

首先，前面的规定仍然有效：在不同作用域里定义的变量，即使重名，无论它们的类型相同或者不同，总是不同的变量，互相之间没有任何关系。

但这里出现了新情况：上面代码段的整个复合语句是外层变量定义的作用域，而作为循环体的复合语句是内层变量定义的作用域。显然，循环体之外是在内层 x 定义的作用域之外，在那里内层的变量定义无效。但内层复合语句出现在外层变量定义的作用域内，按规定，外层变量定义在这里有效。这样，循环中出现的变量 x 应该算哪个 x？是外层的还是内层的？C 语言规定：当内层复合语句出现同名变量定义时，外层同名定义将被内层定义遮蔽。对于上面例子，外层 x 的定义被在循环体里的同名变量定义遮蔽，因此，循环体里的 x 总表示在内层定义的那个名字为 x 的双精度变量。

同样可能出现外部定义的全局变量被函数里面的同名局部变量定义所遮蔽的情况。

5.4.5 静态局部变量

本节从一个小问题出发看另一种变量的意义。假设需要一个函数，它每调用一次就输出一个空格，每调用到第 10 次时输出一个换行。将这个函数命名为 format，它无需参数。我们可以用它改造前面打印完全平方数的程序，使每行输出 10 个数：

```
for (n = 1; n * n <= 200; n++) {
    printf("%d", n * n);
    format();
}
```

现在考虑如何定义这个函数。显然这里需要一个计数变量，每次调用将它加 1，发现变量值达到 10 时就输出换行符并将计数器归 0。第一个想法可能是写出下面定义：

```
void format(void) {
    int m = 0;
    if (++m == 10) {
        putchar('\n');
        m = 0;
    }
    else putchar(' ');
}
```

可惜这个定义不行，无论调用多少次都不输出换行。由前面关于变量性质的讨论，不难发现其中原因（如果读者还没看清，就应当重读前节内容）：m 是局部自动变量，每次函数调用建立新 m 并初始化为 0，所以无论调用 format 多少次，m 也不可能达到 10。

可见，为完成这里所需要的工作，这个函数必须的自己的两次调用之间传递信息，因为函数的一次执行需要知道此前函数已被调用的次数。由于存在期的限制，自动变量无法胜任

这种信息传递工作。

一个解决办法是把 m 定义为外部变量。这样，m 的存在期不再依赖于函数调用，其值也能在函数的调用之间保持不变（只要其间没有另外给它赋值），从而可以完成传递信息的任务。外部变量当然可以在任何函数里使用。下面是修改后的定义：

```
int m = 0;
void format(void) {
    if (++m == 10) {
        putchar('\n');
        m = 0;
    }
    else putchar(' ');
}
```

这一函数已经能正常工作了。

然而这个函数定义有一个重要缺陷。按设想，m 保存的是 format 的私用数据，但是为完成所需工作，这里将它定义为外部变量，因此就可以被任何函数使用了。这样，程序的其他部分就可能不恰当地使用 m（例如将局部变量 n 不慎写成 m），可能导致 format 计数错误，还可能掩盖了其他程序错误。当然，这里的计数正确与否未必重要，因为只是个小例子，但如果这种变量里保存的是重要数据呢？信息隐蔽与合理保护是非常重要的问题，小到程序的组织结构，大到关系国家安全的重要计算机系统，都必须关注这个问题。

这个小问题实际提出了对另一种变量的需求，这种变量的作用域应是局部的，定义在函数体里，从而保证信息的隐蔽性，避免其他函数无意的越权访问；而其存在期应是全局的，因此可以跨越函数的不同调用，在两次调用间传递信息。此外，这种变量的初始化只应进行一次，使变量值能在函数的不同调用间保持。C 语言的*静态局部变量*就是这样的。静态局部变量的定义位置与其他局部变量一样，另用关键字 static 指明其特殊性。静态局部变量的性质就是上面三条：局部作用域、全程存在期、一次初始化。

完成上面工作的更合理函数定义是第一个 format 定义的简单修改：

```
void format(void) {
    static int m = 0;
    if (++m == 10) {
        putchar('\n');
        m = 0;
    }
    else putchar(' ');
}
```

这样就既保证了功能的正确，又保证了局部数据的安全性，表现出静态局部变量的作用。

静态局部变量的另一个典型应用是实现随机数生成函数，每调用一次产生一个随机数。前面介绍了标准库的随机数功能，现在考虑如何自己定义一个随机数生成函数。每次调用时需要前次得到的递推值（*种子值*），需要用一个变量保存它。显然该值不能用自动变量保存，静态局部变量是一种合理选择。下面是一个例子。请读者考察这个定义，看看它产生的序列是否具有较好随机性。如果发现缺陷，请设法改进。

```
int random () {
    static unsigned long seed = 1;
    return seed = (seed * 1103515245 + 12345) % 32768;
}
```

5.4.6 变量的几个问题

C 语言中还有其他变量类，它们各有特点。本节先介绍这些变量的情况。

寄存器变量

局部自动变量可加关键字 register 定义为寄存器变量。寄存器是 CPU 里的临时数据存储单元，其特点是速度最快而数量最少，是最紧缺的存储资源。定义寄存器变量是想告诉编译程序：该变量对程序效率影响较大，最好把它安排在寄存器里。实际安排由编译程序决定。这种说明只是有关优化的提示。寄存器的一个特点是没有地址，因此不能做后面要介绍的与地址有关的操作，即第七章要介绍的指针操作。例如下面函数里的 i 和 n：

```
int fun (register int n) {
    register int i;
    ... ...
}
```

由于现代计算机系统很复杂，各种成熟的编译程序都能针对具体系统和具体程序的情况做许多有价值的优化，而人认为重要的优化则未必真正有效。因此，最好还是将优化的考虑留给编译程序处理，不必过多考虑 register 描述。

外部静态变量

全局变量的一个缺陷是作用域太大，程序大时有可能变成问题。例如，一个大程序可能由几个人或几个小组共同开发，为防止名字冲突，所有外部变量名都需要事先商定，外部变量很多时就很麻烦。为缓解这一问题，C 语言提供了一种作用域局限于一个源程序文件的外部变量，称为静态外部变量。这种变量也用关键字 static 说明（这个静态与局部变量的静态毫无关系）。静态外部变量能（且只能）在其定义所在的源程序文件里使用。

C 语言还允许把函数定义为静态，方式是在返回类型前加 static。这样定义出来的函数也只能在一个源文件内部使用，也是为大系统的开发服务的，可以帮助避免函数名冲突。在不同源程序文件里定义的静态函数，即使名字相同，也被认为是不同函数。

变量的初始化

前面讲过，变量定义时可以直接初始化，方式是加一个初始化部分。例如：

```
int n = 5, k, m = n;
```

这不仅定义了三个整型变量，还为其中的两个指定了初始值。

外部变量、局部静态变量的定义在程序开始执行前完成，其初始化工作也在程序执行前完成。这些变量的存在期一直延续到程序结束，其初始化工作只做一次。这种性质也是保证前面第三个 format 函数能正常工作的一个重要因素。由于这些变量的初始化在程序执行前完成，所以其初始化表达式有严格限制，只能用不需要执行程序就可以求出值的表达式。最常见的是直接写出的文字量，也允许用文字量、符号常量及基本的运算符号构造起来的表达式，但不能包含各种涉及赋值的运算，如增量/减量运算等。

程序执行进入复合语句按定义的顺序逐个建立自动变量，如果变量定义包含初始化（是表达式），就用该部分求出的值设置变量值。基本类型的自动变量对初始化表达式的形式没有限制，可以包含函数调用，引用当时已有值的任何变量。只是表达式类型应符合变量类型的需要（可转换）。函数参数的初始化由函数调用表达式确定，在执行进入函数体之前完成。由于自动变量在每次执行进入其定义域时重新定义，因此也每次重新初始化。

默认初始化

定义变量时可以不写初始化部分。如果定义外部变量和局部静态变量时未写初始化部分，系统自动将其初始化为 0 值。即，C 语言对这两类变量提供了默认的初始化方式。

如果定义自动变量（包括寄存器变量）时不提供初始值，系统将不自动做初始化，定义后的变量处在未初始化状态。直接使用未初始化自动变量的值显然没道理，因为不知道所用的值是什么。这是初学者常犯的一个错误。许多编译系统能对这种错误提出警告。

常变量

常变量指加前缀关键字 const 定义的变量，这种变量的特定是不允许赋值。常变量的值只能通过初始化确定，在其存在期里总代表着同一个值。下面是一个常变量定义：

```
const int num = 10;
```

读者可能要问，常变量有什么用呢？有了字面量还需要常变量吗？常变量还是有一些特点的。首先，常变量有名字，有名字的东西可以共享，可以在不同地方使用。有时人们就在程序的前面定义一批常变量，例如：

```
const double Pi = 3.14159265;
const double E = 2.71828;
```

我们可以定义任何类型的常变量，这是常变量与 enum 定义的枚举常量不同的地方。下面有专门一节对 C 语言里定义“常量”的不同方式进行讨论和比较。

另外，假设某常变量定义出现在局部作用域里，那么这个常变量也在执行中动态建立和初始化，每次初始化的值完全可以不同。也就是说，虽然常变量在存在期里保持一个固定的值，但由同一局部定义建立的常变量的两个存在过程中却可能存着不同值。看下面例子：

```
for (i = 2; i <= 200; i+= 2) {
    const int n = i * i;
    ... ...
}
```

这样，在每次进入循环体将建立一个新的常变量 n，这个常变量的值在循环体的不同执行中是不同的。

与常变量类似，函数也可以有常参数。这种参数同样由实参提供初值，但在函数体里不允许对它们重新赋值。常参数的定义形式也是在类型描述前加 const 关键字：

```
int f(const int n, int m) { ... ...}
```

实际上，在 C 程序里使用更多的常参数是常指针参数，有关情况见讨论指针的一章。

5.4.7 一个实例

求函数的根是数值计算里经常要做的事情，本节将讨论一种求函数根的方法，并以实现这一方法的程序作为函数定义和使用的实例。这里讨论的求根方法称为弦线法。

现在首先介绍弦线法。假设现在需要求根的函数是 $f(x)$，而且给定了一个求根区间 $[x_1,x_2]$。对考查的区间 $[x_1,x_2]$ 的两个端点 x_1 和 x_2，我们可以做一条弦过函数图形在两端点的位置 $(x_1,f(x_1))$ 和点 $(x_2,f(x_2))$，见图 5.4。如果值 $f(x_1)$ 和 $f(x_2)$ 异号（一正一负），上述弦必定与 x 轴有一个交点，交点的 x 坐标可用下面公式求出：

$$x=\frac{x_1\cdot f(x_2)-x_2\cdot f(x_1)}{f(x_2)-f(x_1)}$$

根据 $f(x)$ 的正负情况，下一步可以把考虑问题的区间缩小为 $[x_1,x]$ 或者 $[x,x_2]$，并保证在缩小后区域的两个端点上的函数值仍异号。这样就可以进一步用弦线法逼近函数的根。这个过程的特殊情况是某次的分界点正好是根。即使没这么幸运，由于区间不断缩小，我们也能得到任意接近根的数值结果。

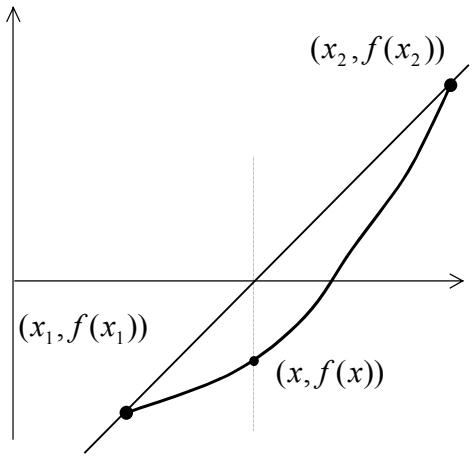


图 5.4 函数图形与弦线

在写解决这个问题的程序时，我们考虑定义几个函数：被求根的数学函数是独立的逻辑整体，其定义依赖于具体需要，但它应该是有一个双精度参数并给出双精度结果的函数。我们在程序前部给出其原型说明，函数定义可以写在程序中任何地方，下面没有具体给出。

求数学函数形成的弦线与 x 轴的交点可看作独立工作，我们定义函数 crossp，它以区间两端点的坐标为参数，算出弦线与坐标轴的交点。

把求根过程也定义为一个函数，这样，这个程序片段就可以用在任何程序里，只要提供具体需要求根的函数，程序中以所希望的值调用求根函数就可得到所需结果。作为试验，读者可以写出适当的主函数，看一看函数求根的实际情况。下面是有关定义：

```
#include <stdio.h>
#include <math.h>
double f (double); /* f的原型，函数定义可以写在任何地方 */

double crossp (double x1, double x2) {
    double y1 = f(x1), y2 = f(x2);
    return (x1 * y2 - x2 * y1) / (y2 - y1);
}

double root (double x1, double x2) {
    double x, y, y1 = f(x1);
    do {
        x = crossp(x1, x2);
        y = f(x);
        if (y * y1 > 0) { /* y与y1符号相同，取新区间为 [x,x2] */
            x1 = x; y1 = y;
        }
        else x2 = x;      /* y与y1符号不同，取新区间为 [x1,x] */
    } while (fabs(y) >= 1E-6); /* y值不够小，继续 */
    return x;
}

int main () {    /* 定义从略 */  }
```

假设我们需要对函数 $f(x) = x\sin x - 2x^2$ 求根，只要在程序中加上下面函数定义：

```
double f (double x) {
    return x * sin(x) - 2 * x * x;
}
```

并给出适当的主函数定义。

这个例子牵涉到前面提出的许多问题：函数原型说明，不同函数中同名的局部参数，局部自动变量的初始化（这里的初始化表达式包含函数调用），程序的函数分解，等等。关于函数 root 能解决所提出的问题，请读者自己进一步分析。

5.5 预处理

用语言处理系统把源程序转变成为可执行程序的过程称为“源程序的加工”。C 语言程序的加工分为三步：预处理、编译和连接，图 5.5 描绘了这个过程。前面介绍过编译和连接两个加工步骤，本节要介绍预处理步骤。预处理的工作最先做，由 C 语言预处理程序完成。预处理程序是任何 C 系统的组成部分，它处理 C 源程序里所有的预处理命令，得到不含预处理命令的源程序。C 语言提供预处理命令的目的是使编程序更加方便。

预处理命令以独立的预处理命令行的形式出现。# 符号是特殊引导符号，如果源程序里某行的第一个非空格符号是 #，那么这行就是预处理命令行。预处理命令的作用是要求预处理程序完成一些操作，下面分别介绍各种预处理命令的情况。

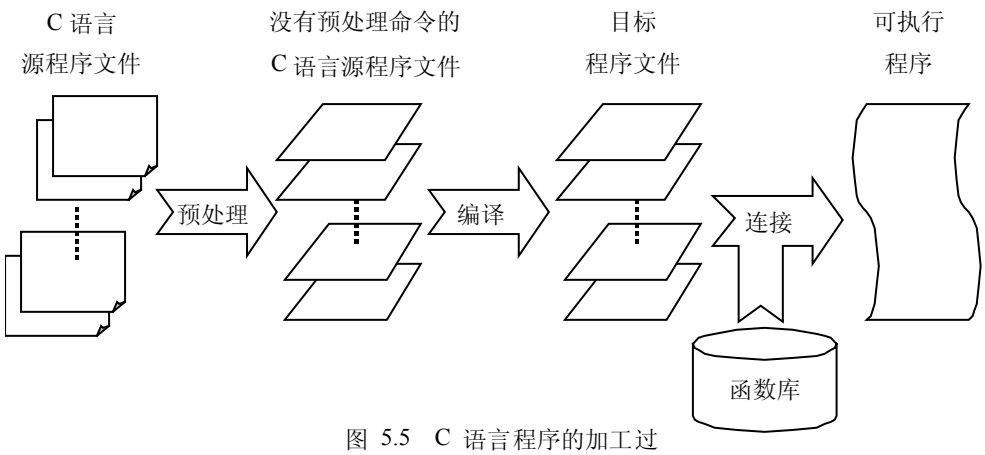


图 5.5 C 语言程序的加工过

5.5.1 文件包含命令

前面例子程序中反复出现了以 #include 开始的行，这就是*文件包含命令*，其作用是把指定文件的内容包含到当前源文件里来。文件包含命令有两种形式：

```
#include <文件名>
#include "文件名"
```

两者的差异在于查找相应文件的方式。对第一种形式，预处理程序直接到某些指定目录中查找所需文件，目录指定方式由具体系统确定，通常指定几个系统目录。对第二种形式，预处理程序先在源文件所在目录中查找，找不到时再到指定目录查找。因此，在包含系统文件（如标准库文件）时一般应该用第一种形式；如果要包含自己定义的文件，该文件存放在与被处理源程序的同一个目录下，显然应该用第二种形式。

文件包含命令的处理过程是：首先查找所需文件，找到后就用该文件的内容取代当前文件里这个包含命令行。替换进来的文件里仍可有预处理命令，它们也将被处理。

前面例子程序里都用了包含命令引进标准头文件（这种文件以 .h 作为扩展名，因此也称为 .h 文件）。标准头文件通常以文本文件形式存在 C 系统目录的某子目录里（子目录名通常为 include 或 h），其内容主要是标准库函数的原型说明、标准符号常量的定义等。用 #include 命令包含这种文件，相当于在源程序文件的前部写这些函数原型等，这对保证编译程序正确处理标准库函数调用是至关重要的。

文件包含命令的最重要用途是用于组织程序的物理结构，使我们能够以多个源程序文件的方式开发比较大的程序，直至非常复杂的软件。有关这方面的讨论见第十章。

5.5.2 宏定义与宏替换

由 #define 开始的行称为*宏定义命令行*。宏定义行有两种形式，下面分别介绍它们。

简单宏定义

简单宏定义的形式是：

```
#define  宏名字  替代正文
```

*宏名字*应是一个标识符，至少一个空格之后的*替代正文*可以是任意一段正文，其中可以包含程序中能出现的任何字符和空格等，一直延续到本行结束。如果需要写多行的替代正文，可以行末写一个符号“\”（注意，反斜线后紧接换行符），这将使下一行继续被当作*替代正文*（反斜线符和换行都丢掉）。宏定义的作用就是为*宏名字*定义替代。

预处理程序记录处理当前文件的过程中遇到的所有*宏名字*及其*替代*。如果在扫描源程序正文中遇到已定义的宏名字，就用对应的替代正文替换它，这种操作称为*宏展开*或*宏替换*。

替代正文里也允许出现宏名字，它们将被继续展开，直到不需要展开为止。注意，在宏展开过程中只检查完整的标识符，字符串里的同样字符序列并不认为宏名字的出现。

当替代正文是数值或者可以静态求值的表达式时，如果相应宏名字在程序中出现，就相当于在那里写了这个数值或者表达式，这也是一种定义符号常量的方式。在许多书籍中可以看到程序前面有如下形式的宏定义：

```
#define START 0
#define END 300
#define STEP 20
```

这样定义的效果与前面用 enum 定义枚举常量类似，但是通过预处理过程中的宏替换实现的。有关它们的比较见下一节。此外，由于宏替换的替代正文可以是任何东西，完全可以用这种方式定义代表整数之外的值的宏名字。

实际上，替代正文可以写任何东西，下面是两个简单例子：

例 1，如果定义了：

```
#define SLD static long double
```

此后，假如程序中出现了：

```
SLD x = 2.4, y = 9.16;
```

经过替换后，这一行将变成：

```
static long double x = 2.4, y = 9.16;
```

和直接写类型一样。这里的替代正文的是几个标识符的序列，它们构成一个类型描述。

例 2，如果定义了：

```
#define NOSTOP while(1)
```

程序中出现的所有

```
NOSTOP { ... ... }
```

宏展开后这段代码成为一个条件为真的循环。这里的替代正文根本不是完整的 C 程序结构。

预处理程序并不检查宏定义中的替代正文是不是 C 语言结构，也不检查替换之后得到的结果是不是正确的 C 程序段，它只简单完成宏替换的工作。给出适当的替换正文，正确使用已定义的宏，都是写程序的人自己的事情。

带参宏定义

带参数宏定义的形式是：

#define 宏名字(参数列表) 替代正文

与前一形式的不同在于宏名字后紧跟一个左括号（这里绝不能有空格），括号里是逗号分隔的标识符，作为宏参数名。这段之后是宏定义的替换正文的任意字符序列。

使用带参宏时不但要给出宏名字，还要用类似函数实参的形式给出各宏参数的替代段，多个替代段之间也用逗号分隔，这种形式称为宏调用。对宏调用的替换分为两步：首先用各实际替代段替换宏定义替代正文里出现的各个宏参数，而后将这个替换的结果（展开后的替代正文）代入程序里出现宏调用的位置，形成宏替换的最后结果。

例 1，定义求两个数据之中较小的一个的宏。它可以如下定义：

```
#define min(A, B) ((A) < (B) ? (A) : (B))
```

如果程序中某处出现语句：

```
z = min(x + y, x * y);
```

在预处理过程中，首先将替代正文段里的 A 都代换为 x + y，B 代换为 x - y，再将代换结果带入程序代码中。这个语句将被展开为：

```
z = ((x + y) < (x * y) ? (x + y) : (x * y));
```

带参宏的使用形式与函数调用类似，其作用似乎也像函数调用，但两者实际上很不同。首先，宏定义和调用并不考虑类型问题。例如上面的宏定义里就可以对任何合理的（数值）

类型使用。在执行方面，宏调用将在程序加工中被在现场展开，不会形成运行中的调用动作。一个定义好的宏在程序中某个地方能否使用、使用后会发生什么现象（例如会不会产生类型转换）、能否得到预期效果等，完全看宏展开之后的情况。

带参宏有几个值得特别注意的问题。首先，有些宏展开后会引起对参数的多次计算，而从宏调用的形式上完全看不到这种情况。例如上面定义的 min，展开之后总会有一个参数表达式被计算两次，有时这种情况会引来奇怪后果。请看下面调用：

```
z = min(n++, m++);
```

展开的结果是：

```
z = ((n++) < (m++) ? (n++) : (m++))
```

无论变量 n 和 m 在语句执行前的情况如何，总会有一个变量做了两次增量操作。这个情况在程序正文中完全看不到，很可能成为程序里难以发现的错误。

此外，人们通常都在带参宏的替代正文中写许多括号，把各参数的出现都括起，也把整个段括起，以防展开后由于运算符的优先级而引起问题。假设定义了下面的求平方宏：

```
#define square(x) x * x
```

表面上看它完全正确，但在特定环境下它却可能出问题。考虑下面调用：

```
z = square(x + y);
```

宏展开后得到的是：

```
z = x + y * x + y;
```

这显然不可能是写程序的人所希望的。

宏定义从定义的位置开始起作用，直到本源程序文件结束。不允许对一个宏名字重复定义。C 还提供了预处理命令#undef，其作用就是取消已有的宏定义，用：

```
#undef 宏名字
```

可以从这个命令出现的位置开始取消对宏名字已有的定义。

标准库的一些常用“函数”实际上是用宏定义实现的。例如前面程序里反复用过的 getchar、putchar 等，以及头文件 ctype.h 中定义的各种字符分类操作。在使用这些东西时，就需要注意由于多次求值而造成程序错误。

有时人们也借助于宏定义来简化程序书写。带参宏在这方面的效果有与函数类似之处，也有不同之处。采用宏定义的方式，由于相应的宏调用在定义的位置上现场展开，可以避免程序执行中由函数调用引起的额外开销，因此可以提高程序执行的效率。另一方面，宏展开也会导致程序变长，可执行程序变大。此外，复杂的宏定义展开之后的情况可能很复杂，从而导致程序错误难以定位和排除。因此，应特别注意宏定义的“正确性”。

初学者写宏定义的一个常见错误是在替代正文后面不适当地多写了分号。按照宏定义的展开规则，宏名字之后的所有东西都被当作替代正文，这样，在替代正文中的分号也会被代入程序里，从而可能造成程序错误。例如，下面可能就是一个不正确的宏定义：

```
#define NUM 10;
```

这样定义之后，程序里的：

```
for (i = 0; i < NUM; ++i) { ... }
```

就会被展开为：

```
for (i = 0; i < 10;; ++i) { ... }
```

造成编译出错，显然不是我们所希望的。

由上面的讨论可以看出，C 语言的宏机制是一种非常简单的正文替换机制。预处理过程中所做的宏替换就是一种简单的正文替换。在这里根本不检查程序的语法形式，更不会处理类型等等问题，只是用一段字符去代替另一段字符。一般说，在写程序时，如果有其他可以用的机制，我们最好还是采用其他机制，例如枚举定义、常变量、函数等等。它们的意义更清晰，也更不容易用错，编译程序也能帮助我们做更多的检查。

5.5.3 条件编译命令

另一组主要的预处理命令前面没有用过，简单程序里也不常用。它们实现条件编译，其作用是划出源程序里的一些片段，使预处理程序可根据条件保留或丢掉某一段，或者从几个片段中选择一段保留下来。实现条件编译的预处理命令有四个，它们是：

```
#if        #else        #elif        #endif
```

其中#if 和#elif 命令要求一个能静态求出整型值的表达式。另外两个没有参数。

条件编译命令的常见使用形式有如下三种：

```
1)
#if  整型表达式
  .... /* 代码片段 */
  .... /* 条件成立时保留 */
#endif

2)
#if  整型表达式
  .... /* 代码片段 */
  .... /* 条件成立时保留 */
#else
  .... /* 代码片段 */
  .... /* 条件不成立时保留 */
#endif
```

```
3)
#if  整型表达式
  .... /* 条件成立时 */
  .... /* 保留 */
#elif  整型表达式
  .... /* elif 部分 */
  .... /* 可以有多个 */
#elif  整型表达式
...
...
#else
  .... /* 条件都不成立 */
  .... /* 时保留 */
#endif
```

其中的整数表达式是条件，值为 0 表示条件不成立，否则就是条件成立。这里常用 ==、!= 做判断，例如判断某个宏定义的符号常量值是不是等于某个值等。

形式 1）用于描述在一定条件下保留或丢掉一段代码（条件成立时保留）；形式 2）用于在两段代码中选择一段；形式 3）根据多个表达式的情况，从若干代码段里选取一段。请注意，这些选取都在预处理阶段完成的，得到的是选择后的结果，选取命令行和应丢掉的片段都没有了。条件命令也允许嵌套。

为方便使用，C 语言还提供了一个特殊谓词 defined，其使用形式有两种：

```
defined 标识符        或        defined(标识符)
```

当标识符是有定义的宏名字时 defined(标识符) 得到 1，否则得 0。这种表达式常被作为条件编译的条件。

此外还有两个预处理命令 #ifdef 和 #ifndef，它们相当于 #if 和 defined 的组合的简写形式：

```
#ifdef 标识符        相当于        #if defined(标识符)
#ifndef 标识符       相当于        #if !defined(标识符)
```

5.6 定义常量

从上面的介绍可以看到，如果在 C 程序里需要定义“常量”，我们可能有三种不同方式：用 enum 定义的枚举常量，用 const 修饰符定义常值的变量，还有用预处理命令的宏定义方式“定义常量”。如果一个程序里确实需要定义一些“常量”，那么应该如何选择呢？下面谈谈目前 C 程序设计界的一般看法及其道理，供希望进一步了解内情的读者参考。这里并不想去分析“什么是常量”，在程序里哪些地方需要常量，需要什么常量。而希望从实践的角度看一些典型情况的选择。

首先，人们认为宏定义是一种缺乏约束的鲁莽的正文代换。代换方式和结果不受任何限制，完全不顾程序的语法和语义。利用它可以将程序里的标识符代换为任何东西，可能导致源程序的意义变得难以理解。因此人们提出的原则是：应该尽可能少用宏定义，在可能用其他方式的地方尽量采用其他更容易把握的方式，尽量用 enum 和 const 代替宏。

在变量定义前加 const 定义出的是 const 变量。这种变量在其他方面与非 const 变量一样，只是不能重新赋值，因此它们的值只能通过初始化给定。我们可以定义各种类型的 const 变量，定义位置决定了它们的定义域和存在期，什么时候建立和初始化等等。但是，const 变量也是变量，因此不能用在“常量表达式”里。例如，不能用作 case 标号的常量，不能用于描述枚举常量的值，不能用于初始化外部变量。（后面还会看到，const 变量不能用在数组定义里说明元素的个数，也不能用于初始化数组和结构、联合变量等。）

枚举常量的值为 int 类型，因此只能用与定义 int 类型的常量。这种常量可用在上面所说的 const 不能使用的各种地方，包括用作 case 标号的常量，用于初始化其他枚举常量，初始化全局变量等等。今后在可能的情况下，我们将尽量使用枚举常量。C 程序设计界的人们也更赞同这种方式。

当然，如果我们需要非整数值的常量，那就应该考虑用 const，例如：

```
const double Pi = 3.14159265;
const double E = 2.71828;
```

只有在这两种方式都不合适，而用宏定义又能带来特别的方便时，才应考虑用宏定义。

5.7 字位运算符*

计算机里的数据以二进制编码形式表示，位（二进制位，bit）是最基本的编码单位，是最小的数据表示单位。实际问题中有些对象的变化情况很简单，只用一个或几个位就能表示。例如前面的单词统计程序，读入状态（state）只有两种不同情况，用一个二进制位就够了。如果程序里这种数据对象很多，将它们都表示为现成类型的对象，就可能造成很大存储浪费。一种解决办法是把多个这类数据对象存入一个基本数据类型的变量或常量。此外，系统程序也常需要直接操作二进制位数据。例如，硬件的工作状态信息常用二进制位串表示，操作它们时就需用位串发命令。为面向复杂的系统程序设计，C 语言特别提供了对二进制位的操作。这里介绍相关的运算符：字位运算符，也称为按位运算符。

先考察基本的位运算。一个位只能取值 0 或 1，位运算从一个或两个 0/1 值出发，得到 0/1 结果。常用位运算共有四个，其中的“否定”为一元运算，其他是二元运算：

位“否定”	一个运算对象，对象为 1 时得到值 0，对象值 0 时得到值 1
位“与”	两个运算对象的值都是 1 时运算结果是 1，其他情况结果是 0
位“或”	两个运算对象的值都是 0 时运算结果是 0，其他情况结果是 1
位“异或”	如果恰有一个运算对象的值为 1 时结果是 1，否则结果是 0

这样，1 和 1“与”的结果是 1，1 和 0“或”结果也是 1，1 和 1“异或”结果是 0。

C 语言基于上述位运算定义了四个字位运算符，这些运算符可用于各种整型对象，得到整型结果。一元字位运算符把整型数据看成二进制位的序列，对每个位分别运算，得到结果的各个位；二元运算符分别对两个运算对象对应的各个位分别运算。这些运算符是：

字位“否定”	~	字位“或”	|
字位“与”	&	字位“异或”	^

现在看一个例子，设变量 x 和 y 都是 16 位整数，它们值分别是：

```
x:    0010,1001,0101,0111
y:    1001,1100,1111,1010
```

对于 x 和 y 做各种字位运算，得到的将是：

```
~x       1101,0110,1010,1000
x & y    0000,1000,0101,0010
x | y    1011,1101,1111,1111
```

*这一节的内容与本章其他部分不同，讨论的是 C 语言一个具体方面。这部分内容比较特殊，与其他部分密切不关系，简单程序里也很少用。读者可根据需要在任何时候学习这一部分，或暂时跳过去。

```
x ^ y    1011,0101,1010,1101
```

请读者根据这个例子，设法进一步弄清楚上面介绍的各种字位运算符的意义。

为讨论字位运算符的使用，需要介绍掩码的概念。掩码（mask）是程序中专门写出或设法构造出的二进制串，其用途就是为了与其他变量做位操作。字位运算中常要构造掩码，或直接用字面量写出来，或利用已有数据通过字位运算做出来。人们写掩码时常用十六进制或八进制写法，因为它们与二进制有直接对应关系，这也是 C 语言里十六进制和八进制表示的主要用途。下面几个例子展示了一些构造掩码的方法。

例，假设 x 是 16 位整变量（16 位二进制序列），现在要求写出一个表达式，判断 x 的第 5 和第 8 位是否都为 0。

表示整数 x 的 16 位二进制序列从低位向高位（从右向左）顺序编号为第 0 到 15 位，下面是一个例子，给出变量 x 的一个假设值，以便读者理解：

	15　　0
x	1 0 1 1 0 1 1 0 1 0 1 0 1 1 0 1
掩码	0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0

用这里给出的掩码与 x 进行“字位与运算"，就可以除了第 5 和第 8 位之外的各个位全都变成 0，而第 5 位和第 8 位的信息会留下来。这样，如果用这个掩码与 x 的运算结果是 0，那么就表示 x 的第 5 位和第 8 位都是 0。所以，题目所要求判断可以写为：

```
x & 0x0120 == 0
```

如果 x 的这两个位不全为 0，字位运算的结果必然不是 0。这正是我们所需要的。这里采用十六进制写掩码，因为这样写起来比较方便。

下面列出采用字位运行的程序中的几个常用操作，实现它们应使用的掩码和运算：

1. 取出被处理数据的某些位：用 & 运算及这些位为 1 其他位都是 0 的掩码；
2. 把被处理数据的某些位设置为 0，其他各位不变（此运算一般称为“复位”或“清 0”）：用 & 运算，以及要设置的那几个位为 0，其他位都是 1 的掩码；
3. 把被处理数据的某些位设置为 1，其他各位不变（此运算一般称为“置位”或“置 1”）：用 | 运算，以及要设置的那几个位为 1，其他位都是 0 的掩码；
4. 使被处理数据的某些位翻转（0 变 1，1 变 0），其他位都不变（此运算一般称为“翻转”）：用 ^ 运算，以及要翻转的那几个位为 1，其他位都是 0 的掩码。

前面例子中用的就是第一种技术。请读者注意，虽然在这里写了“设置”、“改变”等，实际上都是为了说起来方便。字位运算也像算术运算一样，总是从已有的两个整数求出另一个表示结果的整数值，它们都不改变原有的计算对象。

与三个二元字位运算对应，C 语言也提供了三个赋值运算符 &=、|= 和 ^=。用它们可以实现改变左边运算对象的操作（掩码放在运算符右边）。

C 语言还有另外两个字位运算，字位左移和字位右移。它们把整数作为二进制位序列，求出把这个序列左移若干位，或右移若干位得到的序列，其运算符是：

左移：	<<	右移：	>>

左移和右移都是二元运算符，位于运算符左边的是被移位的数据（将它们看作二进制串），右边的整数指明操作中的移多少位。空出位置补 0。左右移运算也是求出新整数值，相应赋值运算符是“<<=”和“>>=”。下面是一个例子：

	15　　0
x	0 0 1 1 0 1 1 0 1 0 1 0 1 1 0 1
x << 3	1 0 1 1 0 1 0 1 0 1 1 0 1 0 0 0
x >> 2	0 0 0 0 1 1 0 1 1 0 1 0 1 0 1 1

例如：

```
x = 23 << 4;
```

x 得到的移位后的值是 368，因为 23 的二进制编码为 10111，如果整数采用 16 位二进制表示，这就是 0000000000010111，左移 4 位得到的是 0000000101110000，这也就是 368。左移的一个特殊用途是实现整数值乘以 2 的幂，1 << 4 的结果是 16。相应地，右移可以实现除以 2 的幂的计算。

例：写函数 getbits，它以一个无符号数为参数，返回这个数从第 p 位开始的 n 个位。这一函数可以写为：

```
unsigned getbits(unsigned x, int p, int n) {
    return (x >> (p + 1 - n)) & ~(~0 << n);
}
```

位的编号仍然是自右向左从第 0 位开始。右移的结果将所需要的那 n 个位移到了最右边（最低位）。掩码的构造：对 0 的字位否定得到全 1 的数，将它左移 n 位使该数的低 n 位变成 0，而所有高位都是 1。对这个结果再求否定，就得到了一个低 n 位都是 1，而高位都是 0 的掩码。这正是我们所需要的。

5.8 编程实例

5.8.1 一个简单猜数游戏

问题：写一个简单交互式游戏。程序自动生成一个位于某范围里的随机数，要求用户猜这个数。用户输入一个数后，程序有三种应答：too big，too small，you win。

设计：我们可以用随机数生成器产生随机数。为了提供更大灵活性，可以在程序开始时要求人提供一个范围（例如 0 到 32767 之间的整数），而后进入游戏循环。每次用户猜出一个数后询问是否继续。这个程序的主要部分是一系列交互式的输入和输出。

首先考虑整个程序的工作流程，可以给出如下基本设计：

```
从用户得到数的生成范围
do {
    生成一个数m
    交互式地要求用户猜数，直至用户猜到
} while(用户希望继续);
结束处理
```

生成数的工作可以直接调用标准库函数 rand。假设我们希望随机数的范围为 0 到 m－1，那么可以采用如下语句得到所需的随机数：

```
unknown = rand() % m;
```

如果将判断用户希望继续的工作定义为函数：

```
int next();
```

将它定义为一个返回 0/1 值的函数，用于控制程序大循环的继续或者结束。另外，也把取得工作范围和取得下一猜数值的工作分别定义为函数：

```
int getrange();
int getnumber(int limit);
```

getrange 要求用户提供一个 2 到 32767 的值，如果用户提供的值超出范围，应该要求重新输入。getnumber 取得的猜测值也应该在给定范围内，否则也应提示用户重新输入。这里我们想给用户几次重新输入的机会，如果超过次数仍然不对，函数就返回一个负值，通知调用处出现了异常情况，使调用函数的程序段可以处理这种情况。

程序的主函数部分已经不难写出来了：

```
int next();
int getrange();
int getnumber(int limit);
```

```
int main() {
    int m, unknown, guess;

    if ((m = getrange()) < 0) return 1; /* 给范围时出错次数太多 */
    ++m; /* m作为取模的数，应该比最大的数大一 */
    do {
        unknown = rand()%m;

        while (1) {
            if ((guess = getnumber(m)) < 0) {
                printf("Too many errors. Stop!");
                return 2; /* 猜数时出错次数太多 */
            }
            if (guess > unknown) printf("Too big!\n");
            else if (guess < unknown) printf("Too small!\n");
            else {
                printf("Congratulation! You win!");
                break;
            }
        }
    } while (next());

    printf("Game over.\n");
    return 0;
}
```

虽然程序中所用的几个函数都还没有定义，我们不但可以编写这段程序，还可以将它送给编译系统去检查错误。当然，在定义好所用的函数之前，还不可能生成可执行程序，更无法实际运行它。

现在考虑读入猜数上界的函数。考虑用一个常量限定用户出错的次数，函数里的读入循环的重复次数用这个数值控制，以免无穷无尽地在这里打转。对于每个输入，都应该检查其合法性：是不是数，是否位于合法范围之内？如果得到一个合适的数就返回它；如果有问题，就要求用户重新输入。当重复次数超过 ERRORNUM 时返回一个负数。下面函数里还加入了适当的输出提示语句：

```
enum { ERRORNUM = 5 };

int getrange() {
    int i, n;
    for (i = 0; i < ERRORNUM; ++i) {
        printf("Choose a range [0, n]. Input n: ");
        if (scanf("%d", &n) != 1 || n < 2 || n > 32767) {
            printf("Wrong number. Need a number in 2~32767.\n");
            while (getchar() != '\n')
                ;
        }
        else return n;
    }
    return -1;
}
```

再考虑读入猜测数的函数。这个函数与前一个类似，也有一些差异。这里需要有一个数值范围参数，函数里的检查也有所不同。但整个函数的结构是一样的：

```
int getnumber(int m) {
    int i, n;
    for (i = 0; i < ERRORNUM; ++i) {
        printf("Your guess: ");
        if (scanf("%d", &n) != 1 || n < 0 || n >= m) {
            printf("Wrong number. Need a number in 0~%d.\n", m-1);
            while (getchar() != '\n')
                ;
```

```
            }
            else return n;
        }
        return -1;
    }
```

我们完全可能将上面两个函数的共同之处抽取出来，定义为一个带检查并允许用户出错的整数输入函数，将上述两个函数的调用都换成对该函数的调用。但这两个函数的输出信息串不同，要将它们统一起来，就需要通过函数的参数传递所需的信息串。有关的参数描述方式需要到下一章才能弄清楚，因此这里就不做这件事了。

写出下述简单函数后，这一程序就完成了：

```
int next() {
    int c;
    printf("Next game? (y/n): ");
    while (isspace(c = getchar()))
        ;
    if (c == 'y') return 1;
    else return 0;
}
```

这里的 while 循环是为了跳过用户输入中前面的空格。这一函数只考虑了用户输入是否为 y 的情况。如果需要，也可以让它处理用户输入出错等等问题。

在这个程序里，我们考虑了函数的定义，并考虑了输入数据检查方面的一些情况。请读者试验这个程序，考虑它在输入处理方面还有什么缺陷：有没有不合理的要求，是否对用户使用提出了不合理的限制，还有哪些值得改进的地方？

5.8.2 加密与解密

加密和解密是许多人非常感兴趣而又觉得有些神秘的事情，这里将它作为一个程序设计的实例。今天，实际应用中的加密和解密工作确实都是借助于程序完成的。

假设有一个文本文件（当然也可以是包含其他信息的文件），我们希望保存它或者传输它，但由于某种原因，又不希望别人了解和使用文本的内容。做到这一点的一种基本方式就是通过加密，改变文件内容的表现形式。为了文件的接受者或者我们自己将来还能看到文件的真实内容，还需要有办法恢复文件的本来面目，这也就是解密。加密和解密的工作历史非常悠久，这些工作可以通过任何一对互逆的动作完成，可以借助于任何辅助信息或者手段。从用计算机完成加密/解密工作的角度看，所需要的就是定义一套系统的改变文件内容编码方案。当然，还要求这一修改是可逆的，以便能完成解密工作。

现在考虑一个简单的文本加密程序，其工作就是读入一个个字符，而后将通过一个函数变换修改过的一个个字符写出去。这一程序的主函数非常简单：

```
#include <stdio.h>

int code(int c) { return c + 13; }

int main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(code(c));
    return 0;
}
```

这里还包括了随便写的一个 code 函数（加密函数）。作为例子，我们可以用这一加密程序对它自身的源程序文本（假定文件名为 code.c）加密，例如在 DOS 的交互状态下输入：

```
code <code.c >magic
```

这一命令将文件 code.c 送入程序 code 的标准输入，并将由标准输出得到的加密结果存入文件 magic。程序完成后，可以在 magic 里看到下面长字符串（为放在这里而截为 3 行）：

```
0v{pyéqr-IÇüqv|;uK↨0v{pyéqr-IÇüqyvo;uK↨↨v{ü-p|qr5v{ü-p6-ê-⌂rüé⌂
{-p-8->@H-è↨↨v{ü-znv{56↨ê↨----v{ü-pH↨----äuvyr-55p-J-trüpun⌂566
-.J-R\S6↨--------}éüpun⌂5p|qr5p66H↨----⌂rüé⌂{-=H↨è↨↨
```

我们再写一个解密程序（decode.c）：

```
#include <stdio.h>

int code(int c) { return c - 13; }

int main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(code(c));
    return 0;
}
```

用命令：

```
decode <magic >original
```

此后又可以在文件 original 里看到原来的信息了（原来的 C 源程序代码）。

用按位异或运算可以写出一个加密函数，它自身同时又是自己的解密函数：

```
#include <stdio.h>

enum { cd = 0xFF };
int code(int c) { return c^cd; }

int main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(code(c));
    return 0;
}
```

请读者自己分析其中的情况。

有关加密方法的研究已经形成了一个成果丰富的领域，有兴趣的读者可以自己找到很多相关的书籍和文章。

本章讨论的重要概念

无符号类型，整数提升，抽象，标准库函数，扩充函数库，程序执行入口，对函数的两种观点，函数封装与接口，形式参数（形参）与实际参数（实参），赋值参数（值参数，值参），函数原型，函数原型说明，过时的形式，外部定义，外部说明，外部变量（全局变量），外部变量定义，外部变量说明，定义与说明，作用域，存在期，自动变量，全局作用域，局部作用域，作用域的嵌套，静态局部变量，寄存器变量，外部静态变量，变量的初始化，常变量，常参数，预处理，文件包含命令，宏定义与宏替换（宏展开），带参数的宏，条件命令，位运算，字位运算符（按位运算符，~、&、|、^、<<、>>），加密和解密

练习

1. 检查你所用的 C 语言系统，弄清其中各种数值类型的表示范围。
2. 利用字符分类函数，写一个程序统计一个文件里出现的十进制整数的个数。另写一个程

序统计文件中出现的十六进制整数的个数。

3. 一对实数可以表示平面上一个点的坐标。一系列实数对，当数对的第一个值为递增时，将它们表示的平面点连接，可得到一条与 x 轴同方向的折线。由一条这种折线、该折线两端引 x 轴的垂线、x 轴本身能够形成一个封闭区域的边界。写一个程序，它接受一系列由标准输入得到的数对，计算出该区域的面积。用一对零（两个 0.0）表示输入结束。假定输入的数对中 x 值总是递增的，y 的值均不为负。如果 y 值可以为负，修改程序使它也能计算正确。
4. 定义函数判断一个点与坐标原点的距离是否小于 1，是否在单位圆内。写一个通过蒙特卡罗方法计算圆周率值的程序：每次计算随机生成两个 0 与 1 之间的实数（利用标准库随机数生成函数产生这种实数），看这两个值形成的点是否在单位圆内。生成一系列随机点，统计单位圆内的点数与总点数，看它们之比的 4 倍是否趋向 π 值。生成 100、200、…、1000 个数据点做试验。
5. 1）利用本章字符图形的基本函数，写出打印实心和空心三角形、正方形、六边形的通用函数（设计适当的参数）。设法写出能打印出空心和实心圆形（或者椭圆形）的函数。

   2）利用本章有关字符图形的基本函数，定义一些画各种字符图形的有用的功能函数，并利用它们画出一些你所感兴趣的图形。

   3）改写上述函数，使它能用某个由参数给定的符号做输出。改造上述函数，使利用它能够用字符 A 在若干输出行中拼出一个大的 A。对其他几个字母、数字做同样的事。
6. 修改正文中的函数 format，使它带一个参数 n，并能以 n 次调用形成一个周期，前 n-1 次调用各输出一个空格，第 n 次调用输出一个换行符。
7. 设计一些程序，设法检查 5.4.5 节定义的 random 函数的随机性是否令人满意。
8. 写出一些数学函数的定义，并用它们和你自己确定的区间去试验 5.4.7 节中定义的求根函数。检查所得到的根是否令人满意。
9. 请实现一积分函数，使它能求出一个定义好的数学函数在某区间的数值积分值。试采用矩形方法、梯形方法，考察它们在不同分割情况下的表现，加细分割对积分值的影响。用不同的数学函数试验程序。如果函数在积分区间出现奇点，程序将出现什么问题？考虑有什么处理办法。
10. 用牛顿叠代法求方程 $f(x)=0$ 的根的迭代公式是：$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$，公式里的 $f'(x)$ 是 $f(x)$ 的微商。由某个初始值 x 出发，反复使用上面公式可以求出方程根的近似值。

    二分法求根的初始情况与弦线法类似，也是从两端点函数值异号的一个区间开始，每次求出区间中点，从中点函数值的符号决定取左边半区间还是右边半区间作为缩短的区间。不断重复这个手续，直到得到满意的结果。

    写出用牛顿叠代法、二分法求函数根的函数。用它们求一些方程（多项式方程或包含超越函数的方程）在某点附近或某对点之间的根。考察方程情况与循环次数的关系（这就需要统计程序循环执行的次数）。
11. 请写出如下程序：

    1）统计输入文件（由标准输入得到）的行数和空格数。

    2）分别统计输入文件里的元音字母和辅音字母。

    3）分别统计输入文件里的非空白字符个数和非空的行数。
12. 写出如下几个程序，完成从标准输入到标准输出的复制：

    1）在遇到的连续空格字符时只输出一个空格字符。

    2）每个词放在一行。

    3）把输入中的每个 tab 字符换成 4 个空格字符。

    4）把输入中的连续 4 个空格字符换成一个 tab 字符。

13. 写出下面的处理 C 语言源程序的程序，它们都由标准输入取得信息（通过重新定向可以把 C 源文件送给这些程序），结果送到标准输出。注意，在被处理程序里可能出现注释、字符串常量（如"/*"）、字符常量（如'"'）、字符或者串字符里的换义字符（例如字符串"Here the \" is not the end of a string"）等，应当考虑如何处理。编译加工各程序转换的结果，以检查程序的正确性。
    1）写程序读入 C 源程序，向标准输出写删除程序中所有的注释后的结果。
    2）写程序删除 C 程序中所有无用字符，使其达到最短并保持程序意义不变。
    3）修改前面完成的统计 C 源程序中标识符总个数的程序，考虑程序中的注释、字符串、字符常量等的影响，保证函数统计结果的正确性。
14. 要取得 0 到 n - 1 的随机数，一个可能的办法是用表达式 rand() % n。这种方法得到的数足够随机吗？取 n 值为 2、3、4 等做试验。如果在某种情况下得到的结果不够随机，请设法找出一种利用能 rand()得到令人满意的随机数的方法。
15. 请写出函数 int setbits(int x, int p, int n, int y)，它求出把整数 x 从左端数第 p 位开始的 n 个位用 y 的最右边 n 个位替换，其他位都不变得到的整数。
16. 请写出函数 int right_rot(int x, int n)，它求出将 x 向右方向循环移 n 位得到的数（右循环移位：总把从右端移出的位值填补到左边空出的位）。
17. 请设计一些加密和解密方法，并定义出相应的加密程序和解密程序。注意：由于加密和解密是将产生出变换后的字节编码，有时你会遇到一些奇怪的现象。例如，你的加密程序可能将某个字符转换成换行符号或者制表符，另一更特殊的情况是将某个字符转换成了文件结束符。这些都可能导致你无法正常显示文件的内容，或者影响你的解密程序的正常运行。要克服文件结束符带来问题，请用 feof 函数判断文件结束（如果需要这样做，请参阅本书后面有关文件处理的讨论）。
18. HTML 网页中有一部分是真正的网页信息内容，另外有大量的形式是由一对“<”和“>”括起的 HTML 标注。(1) 请写一个程序，它能删除由输入得到的 HTML 网页里的所有标注，只留下基本信息。(2) 设法留下原页面里的段落信息，使得到的结果更容易读。
19. 在 HTML 网页里常常有一些对其他网页的链接。请写一个程序，它能抽取出一个网页中所有的链接，一行一个地显示出它们。