第6章 函 数

函数是C和C++语言的基本构件，是所有程序活动出现的地方。本章讨论它们的类C特征，包括传递变元、返回值、原型和递归。第二部分讨论C++特有的函数特征，例如函数重载和引用参数。

6.1 函数的一般形式

函数的一般形式是：

```
ret-type function-name(parameter list)
{
body of the function
}
```

ret-type规定了函数返回的数据的类型，函数可以返回除数组以外的任何类型数据。parameter list（参数列表）是用逗号分隔的列表，由变量名和相关的变量类型组成。当函数被调用时，变量根据该类型接收变元的值。一个函数可以没有参数，这时参数列表为空。但即使没有参数，括号仍然是必需的。

在变量声明中，使用一个用逗号分隔的变量名列表，可以声明多个相同类型的变量。相反，所有的函数参数必须单独声明，每个函数参数必须同时具有类型声明符和参数名。函数的参数声明的一般形式是：

```
f(type varname1, type varname2, . . . , type varnameN)
```

例如，下面的两个函数参数声明，一个是正确的，一个是不正确的：

```
f(int i, int k, int j) /* correct */
f(int i, k, float j)   /* incorrect */
```

6.2 函数作用域的规则

“一种语言的作用域规则”是一组确定一段代码是否知道或者可访问另一段代码或数据的规则。

每个函数都是一个独立的代码块。一个函数的代码是属于该函数专有的，除调用这个函数的语句之外，任何其他函数中的任何语句都不能访问这些代码（例如，用goto语句跳转到另一个函数内部是不可能的）。构成一个函数体的代码对程序的其他部分是隐藏的，除非它使用了全局变量或数据，它既不能影响程序的其他部分，也不受程序的其他部分的影响。换句话说，由于两个函数有不同的作用域，定义在一个函数内的代码和数据不能与定义在另一个函数内的代码或数据相互作用。

在函数内部定义的变量称为局部变量。局部变量随着函数的进入而生成，随着函数的退出而销毁。即，局部变量不能在两次函数调用之间保持其值。只有一个例外，就是用存储类型符static声明时。这会使编译器在存储管理方面像对待全局变量那样对待它，但其作用域仍然被限制在该函数的内部（第2章详细讨论了全局变量和局部变量）。

在C/C++语言中，不能把一个函数定义在另一个函数内部。这也是为什么C或者C++语言从理论上讲不是块结构型语言的原因。

6.3 函数变元

如果一个函数要使用变元，就必须声明接受变元值的变量。这些变量称为函数的形式参数，它们就像函数内的其他局部变量那样，在进到函数时创建，在退出函数时销毁。如下面的函数所示，参数声明位于函数名的后面：

```
/* Return 1 if c is part of string s; 0 otherwise. */
int is_in(char *s, char c)
{
  while(*s)
    if(*s==c) return 1;
    else s++;
  return 0;
}
```

函数is_in( )有两个参数：s和c。如果字符c在串s中，函数返回1，否则返回0。

就像使用局部变量那样，可以给函数的形式参数赋值，或者把它们用在任何合法的表达式中。即使这些变量执行接受传给函数的变元的值的特殊任务，用户也可以像使用局部变量那样使用它们。

6.3.1 按值调用与按引用调用

一般来说，有两种方法可以把变元传给子程序。第一种方法叫“按值调用”（call by value）。这种方法是把变元的值复制到子程序的形式参数中，所以子程序中参数的任何改变都不会影响变元。

“按引用调用”（call by reference）是把变元传给子程序的第二种方法。这种方法是把变元的地址复制给参数。在子程序中，这个地址用来访问调用中所使用的实际变元。这意味着，参数的变化会影响调用时所使用的变元。

默认时，C/C++使用按值调用来传递变元。一般来说，这意味着函数的代码不能改变调用函数时使用的变元。看下面的程序：

```
#include <stdio.h>

int sqr(int x);

int main(void)
{
  int t=10;

  printf("%d %d", sqr(t), t);
```

```
  return 0;
}

int sqr(int x)
{
  x = x*x;
  return(x);
}
```

在这个例子中，传递给函数sqr( )的变元值10是复制给参数x的。当执行赋值语句x=x*x时，被修改的仅仅是局部变量x。用于调用sqr( )函数的变量t的值仍然是10，因此，输出为100 10。

记住，传给函数的只是变元值的副本，所有发生在函数内的变化都不会影响调用函数时使用的变量。

6.3.2 创建一个按引用调用

即使C/C++使用按值调用传递参数，用户仍然能够通过将指针传给变元而不是传递变元自身的方法来创建按引用调用。因为变元的地址被传给函数，所以函数内部的代码就可以改变函数外部的变元值了。

指针可以像其他值那样传递给函数。当然必须把参数声明为指针类型。例如，下面的用来交换两个整数变元值的swap( )函数就说明了这一点：

```
void swap(int *x, int *y)
{
  int temp;

  temp = *x; /* save the value at address x */
  *x = *y;   /* put y into x */
  *y = temp; /* put x into y */
}
```

swap( )函数可以用来交换由x和y指针指向的两个变量值，因为传给函数的是它们的地址（而不是值）。因此，在函数中，利用标准的指针操作可以访问变量的内容，也可以交换用于调用函数的变量内容。

记住，函数swap( )或使用指针参数的任何其他函数，必须用变元的地址去调用。下面的程序展示了调用swap( )的正确方法：

```
void swap(int *x, int *y);

int main(void)
{
  int i, j;
  i = 10;
  j = 20;
  printf("%d %d", i, j);
  swap(&i, &j); /* pass the addresses of i and j */
  printf("%d %d", i, j);
  return 0;
}
```

在这个例子中，变量i被赋值10，j被赋值20，然后用i、j的地址调用swap()（一元运算符&用来产生变量的地址）。因此，是i、j而不是它们的值传给了函数swap()。在swap()函数返回后，i和j的值被交换。

注意：C++允许你通过使用引用参数完全自动化一个按引用调用。这个特征将在第二部分描述。

6.3.3 用数组调用函数

数组已在第4章中叙述过。因而，本节讨论将数组作为变元传送给函数的方法，因为这对标准的“按值调用”参数传递来说是个例外。

当把一个数组用做函数变元时，就把数组的地址传给了函数，这是按值调用参数传递惯例的一个例外。这意味着，函数内部的代码可以操作、修改调用函数时所使用的数组的实际内容。例如下面的函数print_upper()，它以大写字母方式打印字符串变元：

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

void print_upper(char *string);

int main(void)
{
  char s[80];

  gets(s);
  print_upper(s);
  printf("\ns is now uppercase: %s", s);
  return 0;
}

/* Print a string in uppercase. */
void print_upper(char *string)
{
  register int t;

  for(t=0; string[t]; ++t) {
    string[t] = toupper(string[t]);
    putchar(string[t]);
  }
}
```

调用完print_upper()后，main()中数组s的内容就变成了大写形式。如果不需要这种改变，程序可以写成：

```
#include <stdio.h>
#include <ctype.h>
void print_upper(char *string);

int main(void)
{
  char s[80];

  gets(s);
  print_upper(s);
```

```
  printf("\ns is unchanged: %s", s);

  return 0;
}

void print_upper(char *string)
{
  register int t;

  for(t=0; string[t]; ++t)
  putchar(toupper(string[t]));
}
```

在这段程序中，由于数组s的值没有改变，所以它的内容保持不变。

标准库函数gets( )是一个向函数传递数组的典型例子。尽管函数gets( )在标准库中是更复杂的，下面给出了一个简化的函数xgets( )，以说明它是如何工作的。

```
/* A simple version of the standard
   gets() library function. */
char *xgets(char *s)
{
  char ch, *p;
  int t;

  p = s; /* gets() returns a pointer to s */

  for(t=0; t<80; ++t){
    ch = getchar();

    switch(ch) {
      case '\n':
        s[t] = '\0'; /* terminate the string */
        return p;
      case '\b':
        if(t>0) t--;
        break;
      default:
        s[t] = ch;
    }
  }
  s[79] = '\0';
  return p;
}
```

函数xgets( )必须用字符指针调用，当然，这可以是一个字符数组名，按照定义，它是一个字符指针。在入口处，xgets( )建立了一个从0到79的for循环，这就防止了输入的字符串过长。如果输入的字符串超过80个字符，函数返回（实际的gets( )函数没有这个限制）。因为C/C++没有内部的边界检查，用户必须确保用于调用xgets( )的数组至少能容纳下80个字符。当你在键盘上键入字符时，它们被放到字符串中。如果键入了退格，计数器t减1，有效地把前一个字符从数组中去除。当键入回车键时，则将一个null字符添加到串的尾部，用来标记串的结束。由于调用xgets( )时所用的数组被修改了，所以，函数返回的内容包含了用户输入的字符。

6.4 传给 main( )的变元 argc 和 argv

有时在运行时向程序传递信息是很有用的。一般的方法是通过命令行变元把信息传递给 main( )。命令行变元是跟在操作系统命令行中程序名之后的信息。例如，当编译一个程序时，在命令提示符后输入如下命令：

```
cc program_name
```

其中，program_name 是一个命令行变元，它指定了你希望编译的程序的名字。

有两个特殊的内嵌变元 argv 和 argc，用于接收命令行变元。argc 参数容纳命令行变元上的变元数且是一个整数。它至少为 1，因为程序名可作为第一个变元。argv 参数是一个指向字符数指针数组的指针。在这个数组中的每个元素指向一个命令行变元。所有的命令行变元都是字符串——任何数将被程序转换成合适的内部格式。例如，下面这个程序在屏幕上打印 Hello 和你的名字（如果你在程序名后面直接键入了它）。

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

int main(int argc, char *argv[])
{
  if(argc!=2) {
    printf("You forgot to type your name.\n");
    exit(1);
  }
  printf("Hello %s", argv[1]);

  return 0;
}
```

假设把程序取名为 name，用户名是 Tom，应键入 name Tom 来运行这个程序，程序的输出是 Hello Tom。

在大多数环境下，每个命令行变元必须用空格或制表符分隔。逗号、分号或其他的符号不能当作分隔符使用。例如：

```
run Spot, run
```

是由三个字符串构成的，而

```
Herb,Rick,Fred
```

则是一个串，因为通常情况下逗号不是合法的分隔符。

在某些环境下，允许用双引号把一个包含空格的字符串引起来，把整个字符串作为单个变元处理。详情请查阅操作系统手册中有关命令行参数的定义。

必须正确地声明 argv，最常用的方法是：

```
char *argv[];
```

空的方括号表示它是一个不定长的数组。可以通过使用下标访问 argv 的单个变元。例如，argv[0]指向第一个字符串，它总是程序名；argv[1]指向第一个变元，等等。

下面是另一个使用命令行变元的例子，程序名是 countdown。它从一个由命令行变元给定

的数值开始倒计数，当计数到0时，发出嘟嘟声。注意，包含此数的第一个变元用标准函数atoi()转换成整数值。如果字符串“display”是第二个命令行变元，计数将显示在屏幕上。

```
/* Countdown program. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int main(int argc, char *argv[])
{
  int disp, count;

  if(argc<2) {
    printf("You must enter the length of the count\n");
    printf("on the command line. Try again.\n");
    exit(1);
  }

  if(argc==3 && !strcmp(argv[2], "display")) disp = 1;
  else disp = 0;

  for(count=atoi(argv[1]); count; --count)
    if(disp) printf("%d\n", count);

  putchar('\a'); /* this will ring the bell */
  printf("Done");

  return 0;
}
```

注意，如果不指定变元，则显示一条错误信息。如果用户不键入适当的信息就试图运行那些含有命令变元的程序，通常都要做出上述反应。

在argv中加上第二个下标就可以访问某个命令行变元中的某个字符。例如，下面的程序在屏幕上一次一个字符地显示调用它时所使用的变元。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int t, i;

  for(t=0; t<argc; ++t) {
    i = 0;

    while(argv[t][i]) {
      putchar(argv[t][i]);
      ++i;
    }
    printf("\n");
  }

  return 0;
}
```

记住，第一个下标访问字符串，第二个下标访问该字符串中的单个字符。

通常，可使用argc和argv给程序下达初始命令。从理论上讲，可以使用多达32767个变元、但大多数操作只允许为数不多的几个。你通常使用这些变元来指示文件名或者选项。使用命令行变元可以使程序显示出专业水平的特征，且便于在批文件中调用该程序。

当程序不使用命令行参数时，通常明确地把main( )声明为无参数的形式。对于C语言程序，这可通过在它的参数列表中使用void关键字做到这一点（这是本书第一部分中几个程序采用的方法）。然而，对于C++程序，可以简单地声明一个空的参数列表。在C++中，用void来表示空参数列表是允许的，但是是冗余的。

argc和argv只是一种传统用法，它们可以是任意的。用户可以用自己喜欢的任何名字作为传给main( )的参数名。有些编译器还支持传给main( )的另外的变元，所以务必要查阅用户手册。

6.5 return语句

return语句在第3章中已讨论过，正如所解释的，return有两个重要的用途。第一，它使得包含它的那个函数立即退出，也就是使程序返回到调用语句的地方继续执行。第二，可以使用它来返回一个值。这一节就讨论这两个用途。

6.5.1 从函数中返回

函数可以用两种方法停止运行并返回到调用它的程序。第一种方法是执行到函数的最后一个语句，即碰到函数结束括号“}”（当然，在程序目标代码中并不存在“}”，但可以假想它的存在）。例如，下面的函数pre_reverse( )在屏幕上按相反顺序显示字符串“I like C”并返回：

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

void pr_reverse(char *s);

int main(void)
{
  pr_reverse("I like C++");

  return 0;
}

void pr_reverse(char *s)
{
  register int t;

  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
}
```

一旦字符串显示完毕，函数pre_reverse( )便无事可做了，这时函数返回到调用处。

实际上，并没有太多的函数使用这种默认方式来终止程序运行。大多数程序依赖return语句来终止运行，因为必须返回一个值，或是为了使函数代码简化和高效。

一个函数可以有多个返回语句。例如，下面的函数find_substr( )返回子串在原串中的位置，不匹配时，返回-1。

```
#include <stdio.h>

int find_substr(char *s1, char *s2);
```

```
int main(void)
{
  if(find_substr("C++ is fun", "is") != -1)
    printf("substring is found");

  return 0;
}

/* Return index of first match of s2 in s1. */
int find_substr(char *s1, char *s2)
{
  register int t;
  char *p, *p2;

  for(t=0; s1[t]; t++) {
    p = &s1[t];
    p2 = s2;

    while(*p2 && *p2==*p) {
      p++;
      p2++;
    }
    if(!*p2) return t; /* 1st return */
  }
   return -1; /* 2nd return */
}
```

6.5.2 返回值

除了那些类型为void的函数外，其他所有函数都返回一个值，这个值是由return语句指定的。在C89中，如果一个非void函数没有明确通过一个return语句返回一个值，那么就返回一个无用（garbage）值。在C++（和C99）中，一个非void函数必须包含一个return语句，该return语句返回一个值。即，在C++中，如果一个函数被指定返回一个值，在它中的任何return语句必须有一个与它关联的值。然而，如果执行到达了非void函数的结尾处，那么将返回一个无用值。尽管这个条件不是一个语法错误，它是一个很大的缺陷，应该避免。

只要函数没有被声明为void，它就可以作为操作数用在任何有效的表达式中。这样，下面的表达式是合法的：

```
x = power(y);
if(max(x,y) > 100) printf("greater");
for(ch=getchar(); isdigit(ch); ) ... ;
```

但是，函数不能作为赋值对象，下面的语句是错误的。

```
swap(x,y) = 100; /* incorrect statement */
```

C/C++编译器认为这个语句是错误的，而且对含有这种语句的程序不予编译（在第二部分将要讨论，C++也允许这一规则有一些例外，它允许某些函数类型出现在赋值语句的左边）。

我们编写的程序中，函数一般具有三种类型。第一种类型是简单计算型，这些函数专门设计成对变元进行计算，并且返回计算值。计算型函数是一个“纯”函数。标准库函数sqrt( )和sin( )就是这类函数，它们分别计算变元的平方根和正弦值。

第二种类型的函数是对信息进行处理，并返回一个值，以此表示处理的成功或失败。标准库函数fclose( )就是一个例子。它关闭一个文件，如果文件关闭成功，则返回0，否则返回EOF。

最后一类函数没有明确的返回值。这种函数严格地讲是过程，不产生返回值。函数exit( )就是一个例子，它结束程序执行。所有不返回值的函数都应被声明为其返回类型为void类型，这样可以防止把这类函数用在表达式中，有助于避免意外的误用。

有时，那些实际上不产生令人感兴趣结果的函数却无论如何也要返回某些东西，例如，函数printf( )返回所写的字符的个数，然而很难发现真正检查这个返回值的程序。换句话说，尽管除了空值函数以外的所有函数都返回一个值，但是用户不必非得去使用这个返回值。有关函数返回值的一个常见问题是，既然这个值是被函数返回的，是不是必须把它赋给某个变量。回答是：不必。如果没有用它赋值，它就被丢弃了。考虑下面的程序，它使用mul( )函数：

```
#include <stdio.h>

int mul(int a, int b);

int main(void)
{
  int x, y, z;

  x = 10; y = 20;
  z = mul(x, y);                /* 1 */
  printf("%d", mul(x,y));       /* 2 */
  mul(x, y);                    /* 3 */

  return 0;
}

int mul(int a, int b)
{
  return a*b;
}
```

在第1行中，把mul( )的返回值赋给了z。在第2行中，返回值并没有赋给谁，而是被printf( )函数所使用。在第3行中，返回值被丢弃了，因为既没有把它赋给某个变量，也没有把它用做表达式的一部分。

6.5.3 返回指针

虽然返回指针的函数同其他类型函数的处理方法相似，但有些概念还需要在此讨论。

指向变量的指针既不是整数也不是无符号整数，而是某个类型数据的存储地址。这种区别在于，指针运算与其基类型有关。例如，如果一个整型数据指针增1，其值就比原来的值大4（假定整数为4字节长）。一般来讲，指针每增1（或减1），它就指向数据的下一项（或前一项）。由于各种数据类型的长度不同，编译器必须了解指针指向的数据类型。因此，返回指针的函数必须明确声明返回指针的类型。例如，不应该使用返回类型int *来返回一个char *指针。

为了返回指针，函数必须声明成带返回指针的类型。下列程序返回指向字符c在串s中第一次出现的地址的指针：

```
/* Return pointer of first occurrence of c in s. */
char *match(char c, char *s)
```

```
{
  while(c!=*s && *s) s++;
  return(s);
}
```

如果不匹配，就返回一个指向null结束符的指针。下面是一个使用match( )的例子：

```
#include <stdio.h>

char *match(char c, char *s); /* prototype */

int main(void)
{
  char s[80], *p, ch;

  gets(s);
  ch = getchar();
  p = match(ch, s);

  if(*p) /* there is a match */
    printf("%s ", p);
  else
   printf("No match found.");

  return 0;
}
```

该程序读一个串和一个字符。如果字符在串中，则从匹配点开始打印串，否则打印“No match found”。

6.5.4 void类型的函数

void的用途之一是声明那些不返回值的函数，这样就可以防止把这种函数用在表达式中，并且避免意外的误用。例如，函数print_vertical( )在屏幕上从上到下垂直打印它的字符串变元，由于不返回值，它被声明为void。

```
void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

下面是一个使用print_vertical( )的例子。

```
#include <stdio.h>
void print_vertical(char *str); /* prototype */

int main(int argc, char *argv[])
{
  if(argc > 1) print_vertical(argv[1]);

  return 0;
}

void print_vertical(char *str)
{
```

```
  while(*str)
    printf("%c\n", *str+-);
}
```

C语言的早期版本没有定义关键字void。因此，没有返回值的函数简单地默认为类型int。所以，在老的C代码中能看到这种例子就不奇怪了。

6.5.5 main( )返回什么

函数main( )把一个整数返回给调用它的进程，这个进程通常是操作系统。从main( )返回一个值和用相同的值调用exit( )是等价的。如果main( )没有显式地返回一个值，那么传给调用它的进程的值从技术上讲是没有定义的。实际上，大多数C/C++编译器自动返回0，但如果要考虑程序的移植性，就不应依赖这一点。

6.6 递归

在C/C++中，函数可以调用自身。如果在函数体内的一条语句调用其自身，则称这个函数为“递归”的。递归是一个自我定义的过程，亦可称为“循环定义”。

递归函数的一个简单例子就是计算整数阶乘的函数factr( )。整数N的阶乘是1到N之间所有整数之积。例如，3的阶乘是1 × 2 × 3，即为6。factr( )函数和其等效的函数fact( )如下所示：

```
/* recursive */
int factr(int n) {
  int answer;

  if(n==1) return(1);
  answer = factr(n-1)*n; /* recursive call */
  return(answer);
}

/* non-recursive */
int fact(int n) {
  int t, answer;

  answer = 1;

  for(t=1; t<=n; t++)
    answer=answer*(t);

  return(answer);
}
```

非递归函数fact( )的执行是易于理解的。它应用一个从1到n的循环，在循环中用变化的乘积依次乘每个数。

递归函数factr( )的执行过程稍微复杂一些。当用变元1调用factr( )时，函数返回1，否则返回factr(n-1)*n这个乘积。为了求这个表达式的值，用n-1去调用factr( )，直到n=1，调用开始返回。

例如，计算2的阶乘时，对factr( )函数的首次调用引起变元1对factr( )的第二次调用。这次调用返回1，然后被2乘（n的初始值），得到2（读者可以自己试试看被3乘的结果。把printf( )

插到factr( )中，显示各级调用及其中间结果）。

当函数调用自身时，它就在堆栈中为一组新的局部变量和参数分配存储空间，函数代码用这些新的变量从顶部开始重新执行，递归调用并不是把程序代码复制一遍，只是被操作的值是新的。每次递归调用返回时，就从堆栈中去除老的局部变量和参数，并从函数内部该函数调用的调用点恢复运行。递归函数被说成是“压入和推出”。

大多数递归例程没有明显地减小代码规模或节省存储空间，另外，由于重复函数调用增加了时间开销，所以几乎所有的递归函数都会比其非递归形式运行速度要慢一些。对函数的多次递归调用可能造成堆栈溢出。因为函数的局部变量和参数都存放在堆栈中，而每次调用都产生这些变量的一个新副本，堆栈可能会消耗完。然而，除非递归函数失控，否则不必为此担心。

递归函数的主要优点是，可以把一些算法写得更加简洁明了。例如，快速排序算法用非递归形式实现是非常困难的。而且，某些问题，特别是与人工智能有关的问题，非常适宜用递归方法。最后，有些人还发现用递归思考问题比非递归更加容易。

编写递归函数时，必须在函数的某些地方使用条件语句，例如if语句，强迫函数在未执行递归调用时返回。如果不这样做，在调用函数后，它就永远无法返回。省略条件语句是在编写递归函数时常犯的错误。在开发过程中广泛使用printf( )和getchar( )，这样就可以查看执行过程，并且可以在发现错误后停止运行。

6.7 函数原型

在C++中，所有的函数在使用前必须声明。这通常是使用函数原型来完成的。函数原型不是最初的C语言的一部分，它们是在C被标准化时增加的。从技术上讲，标准C不要求函数原型，但是却强烈推荐使用它们。C++总是要求原型。在本书中，所有的例子都包括完整的函数原型。使用函数原型可使C语言提供类似Pascal语言那样的有效类型检查，可使编译器发现和报告任何发生在调用时使用的变元类型和函数参数的类型定义不一致所引起的非法类型转换，以及捕获变元个数与函数参数声明中个数不一致等错误。

函数原型的一般形式是：

```
type func_name(type parm_name1, type parm_name2,...,
               type parm_nameN);
```

参数名的使用是可选的。因为使用参数名可使编译器在错误发生时根据名称来标识任何类型的不匹配，所以，包含参数名是一个好主意。

下面的程序演示了函数原型的值，它产生一条错误的信息，因为它试图用一个整型变元调用sqr_it( )函数，而不是所要求的整型指针（把整数转换成指针是非法的）。

```
/* This program uses a function prototype to
   enforce strong type checking. */

void sqr_it(int *i); /* prototype */

int main(void)
{
  int x;

  x = 10;
```

```
  sqr_it(x); /* type mismatch */

  return 0;
}

void sqr_it(int *i)
{
  *i = *i * *i;
}
```

如果函数的定义出现在程序中第一次使用此函数之前,那么这个函数的定义也可用做它的原型。例如，下面是一个合法的程序。

```
#include <stdio.h>

/* This definition will also serve
   as a prototype within this program. */
void f(int a, int b)
{
  printf("%d ", a % b);
}

int main(void)
{
  f(10,3);

  return 0;
}
```

在这个例子中，因为f( )在它在main( )中使用之前定义，所以不要求分开的原型。虽然小程序中可以把函数的定义用做它的原型，在大程序中则很少有这种可能性，尤其是当使用几个文件时。本书的程序中每个函数都包含一个分开的原型，因为那是在实际中C/C++代码书写的方式。

惟一不要求原型的函数是main( )函数，因为它是当程序运行时第一个被调用的函数。

因为要保持与老版本的C语言兼容，所以在C和C++如何处理不带参数的函数的原型化时有一些小的、却重要的区别。在C++中，空参数列表在原型中通过空出任何参数而简单地表示出来。例如，

```
int f(); /* C++ prototype for a function with no parameters */
```

然而，在C语言中，这种原型意味着某些不同的东西。由于历史的原因，一个空参数列表只是说没有给出参数信息。编译器碰到这种情况时，认为函数可能有几个参数，也可能没有参数。在C中，当函数没有参数时，它的原型在参数列表内使用void。下面是一个出现在C程序中的f( )的原型。

```
float f(void);
```

它告诉编译器这个函数是没有参数的，且任何对带有参数的函数的调用都是一个错误。在C++中，在一个空参数列表中使用void仍然是允许的，但却是冗余的。

注意： 在C++中，f()与f(void)是等价的。

使用函数原型不但可以帮助用户在错误发生前捕获它们,还可以禁止调用变元不匹配的函

数，以确保程序的正确运行。

有一点必须牢记，因为在早期的C语言中不支持完整的原型语法，从技术上讲，原型在C语言中是可选的。这对于支持原型前的C代码来说是必需的。如果正在把老的C代码迁移到C++中，在编译之前，需要增加完整的原型。

记住，虽然在C语言中函数原型是可选的，但在C++中却是必需的。也就是说，C++程序中的每个函数必需包含完整的函数原型信息。

6.7.1 标准库函数原型

程序所使用的任何标准库函数必须是原型化的。要做到这一点，需要给每个库函数包括合适的头文件。所有必要的头文件都是由C/C++编译器提供的。在C中，库函数的头文件（通常）都是使用.H扩展名的文件。在C++中，头文件可以是分开的文件，也可以被嵌入到编译器中。无论是哪一种情况，一个头文件都包含两个元素：由库函数使用的任何定义和库函数的原型。例如，stdio.h包含于本书几乎所有的程序中，因为它包含printf( )的原型。标准库函数的头文件在本书第三部分中描述。

6.8 声明变长参数列表

声明参数个数和类型均可变的函数是可能的，最常见的例子是printf( )。为了通知编译器传给函数的变量的个数和类型都是未知的，就必须用三个点号结束它的参数声明。例如，下面的声明指出func( )至少有两个整型参数和紧随其后的数目不定的（包括0个）参数。

```
int func(int a, int b, ...);
```

这种声明形式也可以用于函数的定义。

任何使用可变数量变元的函数必须至少有一个实际变元。例如，下面的例子是不正确的：

```
int func(...); /* illegal */
```

6.9 传统的与现代的函数参数声明

早期的C语言使用不同于标准C或标准C++的参数声明方法。这种早期的方法有时也称为“传统形式”。本书使用的声明方法称为“现代形式”。标准C支持以上两种形式，但特别推崇“现代形式”。标准C++只支持现代参数声明方法，但不管怎样，应该了解传统形式，因为很多老版本的C语言程序仍然使用传统形式。

传统函数参数声明由两部分组成：一个参数列表（位于函数名后面的小括号内）和实际的参数声明（位于右小括号与左大括号之间）。传统参数声明的一般形式是：

```
type func_name(parm1, parm2, . . .parmN)
type parm1;
type parm2;
.
.
.
type parmN;
{
```

function code
}

例如，下面的现代形式声明：

```
float f(int a, int b, char ch)
{
  /* ... */
}
```

其传统声明为：

```
float f(a, b, ch)
int a, b;
char ch;
{
  /* ... */
}
```

在传统形式中，一个类型名后面可跟一个以上的参数声明。

注意：参数声明的传统形式只在C语言中声明，而C++不支持传统形式。