第一部分　C++ 基础:类 C 特性

- C 语言概述
- 表达式
- C 语句
- 数组和串
- 指针
- 函数
- 结构、联合、枚举及用户定义的类型
- 控制台 I/O
- ANSI C 语言标准文件 I/O
- C 语言的预处理程序和注释

本书的第一部分讨论 C++语言(以下简称为 C++)的类 C 特性。读者也许知道,C++是 ANSI 标准 C 语言的增强版本。正是基于这一原因,任何 C++编译器都被定义为一个 C 编译器。由于 C++是建立在 C 语言之上的,所以掌握了 C 语言的编程,便能够用 C++编程。C 语言的许多基本概念也是 C++的语言基础,它们将在这部分中一一介绍。由于 C++是 C 语言的扩展,因而本部分的一切描述也都适于 C++。C++的语言特性将在本书的第二部分详述。C++的类 C 特性专门占用一部分内容,这是为了使具有丰富 C 语言编程经验的读者能够方便、迅速地找到 C++的特征信息,而不必重复涉足他们早已掌握的信息。

注意,本书第一部分摘引了《C 语言大全》的部分内容,如果读者对 C 语言特别感兴趣,那么阅读这本书将会很有帮助。它全面介绍了 ANSI C 标准和标准 C 库函数,此外,还有许多 C 语言的应用实例。

第一章　C 语言概述

- C 语言的起源
- C 语言是中级语言
- C 语言是结构化语言
- C 语言是程序员的语言
- C 语言的程序结构
- 库和连接
- 分离编译

本章的目的是介绍 C 语言的概貌、起源、应用情况及构成原理。由于 C++是建立在 C 语言之上的，因此本章也介绍了 C++产生的历史背景。

1.1　C 语言的起源

C 语言是由 Dennis Ritchie 发明并首先在配备 UNIX 操作系统的 DEC PDP-11 计算机上实现的。C 语言是一种比较古老的语言 BCPL 发展过程中的产物。BCPL 是由 Martin Richards 开发的，它影响了由 Ken Thompson 发明的 B 语言，而 B 语言又导致了 C 语言在 70 年代的发展。

多年来，UNIX 操作系统上配备的 C 语言一直被作为 C 语言的公认标准（见 Brian Kernighan 和 Dennis Ritchie 的《程序设计语言 C》，Englewood Cliffs，N. J.：Prentice-Hall，1978）。随着微型机的发展，出现了一大批 C 语言系统。由于当时不存在统一的标准，因而不同的实现系统存在着差异和不相容。为了改变这种局面，ANSI 在 1983 年初夏成立了一个委员会以制定一劳永逸的 C 语言标准。目前，ANSI C 标准已被采纳，并且大多数 C++和 C 语言编译器都实现了这个标准。

1.2　C 语言是中级语言

C 语言通常被称为中级计算机语言。这并不意味着它的功能差，难以使用，或者比 BASIC、Pascal 那样的高级语言原始，也不意味着它象汇编语言那样，会给使用者带来类似的麻烦。C 语言之所以被称为中级语言，是因为它把高级语言的成分同汇编语言的功能结合起来了。下表示出了 C 语言在计算机中所处的地位。

作为中级语言，C 支持对位、字节和地址这些有关计算机功能的基本成分进行操作。C 语言非常容易移植。可移植性表现为可将某种计算机写的软件改编到另一种机器上实现。举例来说，如果为苹果机写的一个程序能够方便地修改并在 IBM PC 机上运行，则这个程序称为可移植的。

所有的高级语言都支持数据类型的概念。一个数据类型定义了一个变量的取值范围和可在其上操作的一组运算。常见的数据类型是整型、字符型和实型。虽然 C 语言有五种基本数据类型，但与 Pascal 和 Ada 相比，它算不上强类型语言。C 语言支持几乎所有的类型转换。例如，字符型和整型数据能够自由地混合在大多数表达式中。C 语言不支持诸如数组越界等运行错误检查，检查运行错误是程序员的责任。

高级	Ada
	Modula-2
	Pascal
	COBOL
	FORTRAN
	BASIC
中级	C
	FORTH
	Macro-assembler
低级	Assembler

C 语言的特色是它支持对位、字(字节)和指针的直接操作。这使得它非常适于经常进行上述操作的系统程序设计。C 语言的另一个重要特点是它仅有 32 个关键字(其中 27 个来源于 Kernighan 和 Ritchie 的公认标准，另 5 个是 ANSI 标准化委员会增加的)。这些关键字构成了 C 语言的命令集。和 IBM PC 用的 BASIC 相比，后者包含的关键字多达 159 个。

1.3 C 语言是结构化语言

尽管把 C 语言叫做块结构语言是不严格的，但它还是常被归为结构化语言。因为它与其它结构化语言，如 ALGOL、Pascal 和 Modula-2 有许多相似之处。(从技术上讲，块结构语言允许在过程和函数中定义过程和函数。用这种方法，全局和局部的概念可以通过“作用域”规则加以扩展，“作用域”管理变量的“可见性”。因为 C 语言不允许在函数中定义函数，所以不能称之为通常意义上的块结构语言。)

结构化语言的一个显著特征是程序和数据的分离。这种语言能够把执行某个特殊任务的指令和所有数据信息与程序的其余部分分离开并隐藏起来。获得隔离的一个方法是调用使用局部(临时)变量的子程序。通过使用局部变量，我们能够写出对程序其它部分没有副作用的子程序。这使得编写共享代码段的 C 程序变得十分简单。如果开发了一些分离得很好的函数，在引用时仅需知道函数做什么，而不必知道它如何去做。切记，过度使用全局变量(可以被全部程序访问的变量)会由于意外的副作用而在程序中引入错误。设计过 BASIC 程序的人对这个问题都深有体会。

结构化语言为设计者提供了大量的程序设计功能。它直接支持某些循环结构，如 WHILE、DO-WHILE 和 FOR。在结构化语言中禁止或不提倡使用 GOTO 语句，不能象 BASIC 和 FORTRAN 中那样把它作为常用的程序控制语句。结构化语言允许将语句缩行，但又不必拘于严格的程序格式(象 FORTRAN 语言那种格式)。

下面是一些结构化语言和非结构化语言的例子。

非结构化语言	结构化语言
FORTRAN	Pascal
BASIC	Ada
COBOL	C
	Modula-2

结构化语言是现代化的，而非结构性正是陈旧语言的标志之一。如今，人们普遍认为结构化语言更易于设计和维护。

C 语言的主要结构成分是函数——C 的独立子程序。在 C 语言中，函数作为程序构件是一种完成程序功能的基本程序块。函数允许一个程序的不同任务被分别定义和编码，使程序模块化，具有良好设计风格的函数能保证它正确工作且不会对程序的其它部分产生副作用。能否在大型项目中设计出独立的函数是至关重要的。在这种项目中，一个程序员的代码不应意外地影响其它人的程序。

在 C 语言中，实现结构化和代码隔离的另一种方法是使用复合语句(或称分程序)。复合语句是程序单位，是一组在逻辑上相互关联的语句。在 C 语言中，复合语句就是处在左右花括号之间的一串语句。例如：

```
if (x < 10) {
  printf("too low, try again\n");
  scanf("%d", &x);
}
```

在上例中，如果 x 小于 10，位于 if 之后、两个花括号之间的两个语句都被执行。这两个句子连同括号表示一个代码块，它们是一个逻辑单位，其中所有语句必须一起执行。注意，每个语句或是一个单语句或是一个复合语句。代码块不仅能使许多算法得到清晰、优美又有效的实现，而且有助于程序员抓住程序的本质。

1.4 C 语言是程序员的语言

一看这节的题目，人们可能会产生疑问："所有的计算机语言不都是程序员使用的吗?"回答是否定的。我们看一下典型的非程序设计员的语言 COBOL 和 BASIC。COBOL 的设计使程序员难以改善所编写代码的可靠性，甚至不能提高代码的编写速度。然而，COBOL 设计者的本意却是打算使非程序人员能读懂程序(这是不大可能的事)。BASIC 的主要目的是允许非专业程序员在计算机上编程解决比较简单的问题。

相反，对于 C++和 C 语言，程序的生成、修改和现场测试自始至终都由真正的程序员进行，其结果是实现了程序员的期望：很少限制、很少强求、块结构、独立的函数以及紧凑的关键字集合。使用 C 语言编程，既可获得 ALGOL 或 Modula-2 的块结构，又具有汇编代码的高效率。这样，C 语言很快就成了第一流专业程序员的最常用语言。

C 语言被程序员广泛使用的一个原因，实际上是可以用它来代替汇编语言。汇编语言使用的汇编指令，是能够在计算机上直接执行的二进制机器的符号表示。汇编语言的每个操作

都映射为计算机执行的单一任务。虽然汇编语言具有使程序员达到最大灵活性和最高效率的潜力，但开发和调试汇编语言程序的困难是难以忍受的。进一步说，由于汇编语言是非结构化的，最后完成的程序肯定象一团意大利面条——纠缠不清的跳转指令、调用指令和下标。非结构性使得汇编语言难于阅读、改进和维护。也许更重要的是，汇编语言程序不能在使用不同的中央处理器的机器之间移植。

最初，C 语言被用于系统程序设计。一个“系统程序”是一大类程序的一部分，这类程序构成了计算机操作系统及其实用程序。通常被称为系统程序的有：

操作系统
翻译系统
编辑系统
汇编系统
编译系统
数据库管理程序

随着 C 语言的普及，许多程序员开始用它设计各类程序。几乎所有的计算机上都有 C 语言编译程序，这使我们很少改动甚至不加改动就能将为一种机器写的 C 语言源程序移植到另一种机器上编译执行。可移植性节省了时间和财力。C 编译程序均可产生非常紧凑、执行快捷的目标码。它比所有的 BASIC 编译程序的目标码更紧凑更快速。

也许 C 语言被用于所有各类程序设计的主要原因是由于程序员喜欢它。C 语言提供了汇编语言的速度和 FORTH 语言的可扩充性，而很少有 Pascal 和 Modula-2 的限制。每个 C 程序员都可以创建并维护一个专门的函数库。这个库可以根据不同的需要进行裁减，以适应各种程序的设计。C 语言支持（更确切地说是鼓励）分别编译，所以可使 C 程序员方便地管理大型项目，最大限度地减少重复劳动。

1.5 C 语言的程序结构

表 1-1 列举了 32 个关键字，它们与标准 C 文法相结合，形成了程序设计语言 C。其中，27 个关键字已在 C 语言的老版本中被定义。另外 5 个，即 enum、const、signed、void 和 volatile 是 ANSI 标准委员会新加的。

表 1-1 ANSI C 标准定义的 32 个关键字表

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

另外，许多 C 和 C++编译程序都增加了几个关键字以充分利用 8088/8086 处理机的

内存组织，支持中间语言程序设计和中断。常用扩展关键字列在表 1-2 中。有的编译器也许还支持其它功能，以充分利用其特殊环境。

表 1-2　几个通用的 AC/C++扩展关键字

asm	_cs	_ds	_es
_ss	cdecl	far	huge
interrupt	near	pascal	

C 语言的关键字都是小写的。C 语言分大写与小写，else 是关键字，“ELSE”则不是。在 C 程序中，关键字不能用于其它目的，即不允许将关键字作为变量名或函数名使用。

所有的 C 程序都是由一个或多个函数构成的。C 语言中至少必须有一个主函数“main()”，它是程序运行开始时被调用的第一个函数。在编制完毕的 C 代码中，main()实质上包含程序要完成的动作的轮廓。这个轮廓由函数调用组成。虽然从技术上讲，主函数不是 C 语言的一个保留词，但它仍被这样看待。如“main”用作变量名，就会给编译程序造成混乱。

C 语言程序的一般形式如下所示。其中 f1()至 f2()代表用户定义的函数。本书的其它部分将详细讨论程序的一般形式。

```
global declarations

return-type main(parameter list)
{
      statement sequence
}

return-type f1(parameter list)
{
      statement sequence
}

return-type f2(parameter list)
{
      statement sequence}
.
.
.
return-type FN(parameter list)
{
      statement sequence
}
```

1.6　库和链接

从技术上讲，纯粹由程序员自己编写的语句构成 C 语言程序是可能的，但却不常见，因为 C 在它的定义中没有执行输入/输出(I/O)操作的任何方法。因此，大多数 C 程序都含有

对 C 语言的各种标准库函数的调用。所有的 C 编译程序都是和标准 C 函数库一起提供的，后者含有完成各种常用任务的函数。ANSI C 标准规定了函数库的最小集合，然而，有的编译器也许含有更多的函数。例如，由于计算环境的不同，ANSI C 标准并不定义图形函数，但有的编译器也许包含这些函数。

C 编译程序的实现者已经编写了大部分常见的通用函数。但调用一个别人编写的函数时，编译程序要“记住”它的名字，随后，“链接程序”就把编写的程序同标准函数库中找到的目标码结合起来。这个过程称为“链接”。某些编译程序带有自己的链接程序，有些则使用操作系统提供的标准链接程序。

保存在函数库中的函数是可重定位的。这意味着，机器码指令的内存地址并未绝对地确定——只有偏移量是确定的。当把程序与标准函数库中的函数相连接时，内存偏移量被用来产生实际地址。有关的技术手册和参考书中更为详细地讲解了这一过程。然而，用户不必对重定位过程做深入的了解。

编写程序时，用户会在标准函数库中找到许多自己需要的函数。它们都是可以简单地组合起来的程序构件。用户编写的可以重复使用的函数，也可放入库中。有些编译程序允许将用户的函数放入标准库中，有些则允许建立附加库。这两种方法都可将代码放入库中，以便经常使用。

记住，ANSI 标准只描述了最小集的标准库。大多数编译器都支持比 ANSI 定义的通用函数更多的标准库。同样，C 语言 UNIX 版本中的一些函数也未在 ANSI 中定义，因为它们是冗余的。这些函数也许会在用户的编译器中被定义。

1.7 分离编译

大多数 C 和 C++程序都可以包含在一个源文件中。然而，程序越长，编译的时间就越长。因此，C 语言允许用户将一个程序划分成不同块并放入不同文件，分别进行编译。编译好所有文件后，再将它们与库函数一并连接，形成所期望的、完整的执行代码。分离编译的好处是，用户改变一个文件的代码后，不必重新编译整个程序。这样就节约了大程序的调试时间。一般的 C++编译器用户手册都介绍了如何编译复杂程序的指令。

第二章　表达式

- 五种基本数据类型
- 修饰基本类型
- 标识符命名
- 变量
- 存取修饰符
- 存储分类符
- 变量初始化
- 常量
- 运算符
- 表达式

本章介绍C语言的最基本要素——表达式。读者将发现，C语言拥有比其它语言更普遍、更高效的表达式。表达式由C语言的原子要素数据和运算符构成，数据可以是变量或常量。与大多数其它语言一样，C语言支持不同的数据类型。由于数据是一切表达式的核心，因此本章将从C语言的基本数据类型开始介绍。

2.1　五种基本数据类型

C语言有五种基本数据类型：字符、整型、浮点、双精度浮点和无值，分别用char、int、float、double和void来表示。读者将看到，C语言中的其它数据类型都是从这些基本类型演变而来的。这些数据的长度和范围因处理器的类型和C语言编译程序的实现而异。一般来讲，一个字符为1个字节，一个整数为2个字节，但不能肯定，用户的程序在移植时不出现字节溢出之类的现象。ANSI C强调的是每种数据类型的最小范围，而不是字节长度(表2-1)。

浮点值的准确范围取决于它们是如何实现的。通常，整型对应于宿主机中一个字的长度。从理论上讲，char类型值受ASCII字符所限，但实际上，对ASCII字符集范围之外的字符，C和C++都以不同的方式进行了处理。

float和double类型的取值范围通常以数字精度给出。它们的大小由浮点数的表示方法而定。然而，不管用什么表示方法，数值都相当大。ANSI标准描述浮点值的最小范围是1e－37～1e＋37。

类型Void有两个有处，它解释说明为无值返回或产生一般指针的函数。这在以后章节中会讲到。

2.2 修饰基本类型

除 void 类型外，基本类型的前面都可以有各种修饰符。修饰符用来改变基本类型的意义，以便更准确地适应各种情况的需求。修饰符有：

signed	（有符号）
unsigned	（无符号）
long	（长型符）
short	（短型符）

这些修饰符适用于字符和整数两种基本类型，其中，long 还适用于 double。

表 2-1 ANSI 标准中的数据类型定义

类型	近似长度(bit)	最小范围
char(字符型)	8	−127～127
unsigned char(无符号字符型)	8	0～255
signed char(有符号字符型)	8	−127～127
int(整型)	16	−32,767～32,767
unsigned int(无符号整型)	16	0～65,535
signed int(有符号整型)	16	同 int
short int(短整型)	16	同 int
unsigned short int(无符号短整型)	16	0～65,535
signed short int(有符号短整型)	16	同 int
long int(长整型)	32	−2,147,483,647～2,147,483,647
signed long int(有符号长整型)	32	同 long int
unsigned long int(无符号长整型)	32	0～4,294,967,295
float(浮点型)	32	约 6 位有效数字
double(双精度型)	64	约 10 位有效数字
long double(长双精度型)	128	约 10 位有效数字

表 2-1 给出了所有根据 ANSI 标准组合的类型、字长和范围。

signed 对整型的修饰是多余的，但仍允许使用，因为整数的缺省定义是有符号数。signed 最重要的用途是用来修饰字符型。

有符号整数和无符号整数的区别是对整数最高位的解释。若指定一个有符号整数，那么 C 编译程序生成代码时将整数最高位作为符号标志。若符号标志是 0，则数值为正；若符号标志是 1，则数值为负。

通常，负数采用 2 的补码形式，即对数的各位(符号标志除外)求反，若对该数加 1，则置符号标志为 1。

有符号整数在大量的算法中是很重要的，但其绝对值仅为无符号数值的一半。例如，32767：

01111111　　11111111

如果高位设置为1，数值将变为－32767。但是，对于无符号整数，高位置1时，数值将变为65535。

2.3 标识符命名

在C/C++中，标识符是对变量、函数、标号和其它各种用户自定义对象的命名。标识符的长度可以是一个或多个字符。标识符的第一个字符必须是字母或下划线，随后的字符必须是字母、数字或下划线。这里列举一些正确的和错误的标识符命名实例。

正确形式	错误形式
count	1count
text23	hi! there
high_balance	high..balance

ANSI C标准规定，标识符可以为任意长度。但是，若标识符用于链接过程中，则必须至少能由前6个字符唯一地区分。这些标识符称为外部名，包括文件间共享的函数名和全局变量名。若标识符不用于链接过程中，则必须至少能由前31个字符唯一地区分，这些标识符称为内部名。然而，在C++中，标识符没有长度限制，所有字符都是可区分的。这种差异在C程序转换成C++程序的过程中变得十分重要。

C语言中的字母是有大小写之分的，因此count、Count、COUNT代表不同的标识符。不过，如果链接程序中不区分大小写字母，则编译可忽略函数名和全局变量名的大小写问题。但是，当前大多数环境都支持可区分大小写字母的链接。

标识符不能和C/C++的关键字相同，也不能和用户编制的函数或C/C++库函数同名。

2.4 变量

变量是内存中一个命名的位置，它可以存储一个被程序修改的值。所有C语言的变量必须在使用之前定义。变量定义的一般形式是：

```
type variable_list;
```

其中，type必须是有效的C数据类型，variable_list(变量表)可由一个标识符名或由逗号分隔符分隔的多个标识符名构成。下面给出一些定义范例：

```
int i, j, l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

注意，在C语言中，变量名与它的类型无关。

2.4.1 变量在哪里说明?

说明变量有三个基本位置：在函数内部、在函数参数的定义中以及在所有函数的外部。

这些变量分别称为局部变量、形式参数和全局变量。

2.4.2 局部变量

在函数内部说明的变量称为局部变量。在有些C语言文献中，局部变量也称为自动变量，这与使用可选择关键字auto定义局部变量是一致的。由于局部变量这一说法更为普遍，所以本书采用了该术语。局部变量仅由说明它的模块的内部语句所访问。换言之，局部变量在自己的代码模块之外是不可见的。切记，模块以左花括号开始，以右花括号结束。

对于局部变量，要了解的最重要的东西是，它们的生命期仅存在于被说明的当前执行代码中，即局部变量在进入模块时生成，在退出模块时消亡。

说明局部变量的最常见代码块是函数。例如，在下面的两个函数中：

```
void func1(void)
{
  int x;
  x = 10;
}

void func2(void)
{
  int x;
  x = -199;
}
```

整型变量x被说明了两次，一次在func1()中，一次在func2()中。func1和func2中的x互不相关。原因是每个x作为局部变量仅在说明它的模块内是可见的。

C语言中包括的关键字auto，可用于说明局部变量。但自从所有的非全局变量的缺省值假定为auto以来，auto就几乎不使用了，因此在本书中见不到这一关键字。(C语言中的auto与其前身B语言相同，也提供了C++中的auto与C相容。)

出于方便和习惯，大多数程序员都在每一函数模块内的开始处说明所需要的变量。其实，局部变量可以在任何代码块内说明。请看下面的实例：

```
void f(void)
{
  int t;

  scanf("%d",&t);

  if(t==1){
     char s[80];     /* this is created only upon
                        entry into this block */
     printf("enter name:");
     gets(s);
     /* do something ... */
  }
```

```
}
```

这里的局部变量 s 就是在 if 块入口处建立的，并在其出口处消亡。因此，s 仅在 if 块中可知，在其它地方均不可访问，甚至在包含它的函数的其它部分也不可见。

在代码块内说明局部变量的一大优点是可防止不期望的边界效应。因为变量在说明它的代码块外不可见，所以它不可能被偶然转换。

C 与 C++之间的一个重要区别，就是在何处说明局部变量。在 C 语言中，必须在程序块的开头先于其它程序语句说明所有的局部变量。例如，下面的函数在 C 编译时就会出现错误。

```
/* This function is in error if compiled using
   a C compiler, but perfectly acceptable to a
   C++ compile.
*/
void f(void)
{
  int i;

  i = 10;

  int j; /* this is an error in C, but not C++ */

  j = 20;
}
```

然而，在 C++中，这个函数是有效的，因为局部变量的定义可以在程序的任何位置。(关于 C++变量说明的问题将在本书的第二部分深入讨论。)

由于局部变量随着定义它们的模块的进出口而建立和释放，因此它们存储的信息也随执行块的结束而丢失。这一点对有关函数的访问特别重要。当访问一函数时，它的局部变量被建立，当函数返回时，局部变量被销毁。也就是说，局部变量的值不能在两次调用之间保持。(然而，可以通过修饰符 static 来控制编译程序，以保存它们的值。)

若不另外指明，局部变量被存储在堆栈中。堆栈是动态可变存储区，这一事实即解释了局部变量值一般不能在函数调用之间保存的原因。

用户可以把一个局部变量初始化为某一已知的值。这个值在每次进入说明局部变量的程序块时赋给该变量。例如，下面的程序十次都打印数值 10。

```
#include "stdio.h"

void f(void)

main()
{
  int  i;

  for (i=0; i<10; i++)  f();
```

```
  return 0;
}

void f(void)
{
  int  j = 10;

  printf("%d",j);

  j++;     /* this line has no lasting effect */
}
```

2.4.3 形式参数

如果一函数必须使用变元，那么它必须说明接受这些变元的值，这些变量称为函数的形式参数。它们同函数内部的局部变量作用相同。通过下面的函数可以看到，形式参数的说明出现在函数名之后、花括号之间。

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

函数 is_in()有 s 和 c 两个参数。当字符 c 包括在串 s 中时，函数值返回 1，否则返回 0。

变量的类型必须如上所示那样通过说明变量来说明。变量类型一旦说明，便可作为一般的局部变量在函数内部使用。切记，作为局部变量的形式参数也是动态的，它们也随函数出口而被释放。

类似于局部变量，用户可以对一函数的形式参数赋值或将其用于任何合法的 C 语言表达式中。即使这些变量执行接受调用参数值这一特殊任务，它们仍可以象任何其它局部变量那样使用。

2.4.4 全局变量

与局部变量不同，全局变量在任何函数之外说明，并且可被任何一个模块使用。它们在整个程序执行期间保持有效。全局变量在任何函数之外说明，可由函数内的任何表达式访问。

在下面的程序中可以看到，变量 count 是在所有函数之外说明的。它可以放置在任何第一次使用之前的地方，只要不在函数内就可以。说明全局变量的最佳位置仍在程序的顶部。

```
#include "stdio.h"

int  count;      /* count is global */
```

```
void  func1(void);
void  func2(void);

main()
{
  count = 100;
  func1();

  return 0;
}

void  func1(void);
{
  int  temp;

  temp = count;
  func2();
  printf("count is %d", count);     /* will printf 100 */
}

void  func2(void);
{
  int  count;

  for(count=1; count<10; count++)
    putchar(".");
}
```

仔细研究这个程序段之后读者会发现，变量 count 既不在 main()中说明，也不在 func1()中说明，但可以在两者中被访问。函数 func2()中也说明了一个同名的局部变量 count，但 func2 访问 count 时，仅访问自己说明的局部变量 count，而不是那个全局变量 count。切记，全局变量和某一函数的局部变量同名时，在函数内对该名的访问仅针对局部变量，对全局变量无任何影响。C 语言的这种特性很方便，然而，如果忘记了这点，程序即使看起来是正确的，也可能导致运行时的奇异行为。

全局变量由 C 编译程序在动态区之外的固定存储区中存储。当程序中有多个函数都使用同一数据时，全局变量将十分有效。然而，由于下列原因，应避免使用不必要的全局变量：(1)不论是否需要，它们在整个程序执行期间均占有执行空间；(2)由于全局变量必须依靠外部说明，所以在使用局部变量就可以达到其功能时使用了全局变量，就会降低函数的通用性，这是因为它要依赖于其本身之外的东西；(3)大量使用全局变量时，不可知或者不期望的副作用将导致程序出错。在编制大型程序时有一个重要问题是：变量值在程序其它地点的偶然改变。在 C 语言程序中，如果使用了过多的全局变量，这种情况就可能发生。

2.5 存取修饰符

有两种修饰符用于控制对变量的存取或修改方法：它们是 const(常数型)和 volatile(暂

态型)。如果出现的话,它们必须在被修饰的类型符和类型名前面。

2.5.1 const

.const 型变量在程序执行期间是不可改变的(然而 const 变量可赋予初始值)。编译程序可以随意将这类变量放到只读存储器(ROM)中。例如:

```
const  int  a = 10;
```

定义了一个初始值为 10 的整型变量,它在程序中就不可修改,但可用在其它类型的表达式中。const 变量通过初始化或者通过某些硬件相关方法获取初值。

const 变量有个非常重要的用途——它们可以阻止参数被函数修改,即当一个指针传送给一个函数后,函数可能修改该指针所指向的变量。然而,如果指针在参数说明段用 const 修饰,函数就无法修改指针所指内容了。例如,函数 sp_to_dash()将把参数中的空格转换成连字符,即串"this is a test"将打印出串"this-is-a-test"。如果参数说明中用了 const 修饰符,函数 sp_to_dash 则无法修改参数所指对象了。

```
#include "stdio.h"

void sp_to_dash(const char *str)

main()
{
  sp_to_dash("this is a test");

  return 0;
}

void sp_to_dash(const char *str)
{
  while(*str) {
     if(*str==' ') printf("%c", '-');
     else  printf("%c", *str);
     str++;
  }
}
```

如果用户按修改串的方式来改写 sp_to_dash(),编译则通不过。例如,下列 sp_to_dash()代码,编译时将报错。

```
/* this is wrong */
void  sp_to_dash(const char *str)
{
  while (*str) {
       if(*str==' ') *str="-";  /* can't do this */
       printf("%c", *str);
       str++;
  }
```

}

C 标准库中的许多函数在它们的参数说明中都使用了 const。例如，strlen()函数的原型如下：

```
int strlen(const char * str);
```

把 str 描述成 const 可确保函数 strlen 不会修改 str 指引的变量。通常，当一个标准库函数不必修改调用参数指引的对象时，参数则用 const 来描述。

用户也可以使用 const 来确保自己的程序不修改变量。切记，const 型变量可以被用户的程序之外的东西所修改。例如，硬件设备可以给 const 变量赋值。然而，通过说明变量为 const 型，用户将发现外部干扰所引起的程序变化。

2.5.2 volatile

修饰符 volatile 通知编译程序：变量值可能由程序中没有显示说明的方式所改变。例如，全局变量的地址可能被传递到操作系统的时钟子程序，用来保存系统实时时间。在这种情况下，变量的值变了，而程序中却没有任何显示的赋值语句对其进行修改。引入修饰符 volatile 的重要原因是，大多数 C 编译程序都假定表达式内部变量的内容不变，并以这一假定来自动优化某些表达式。有些编译程序在编译过程中还改变表达式的计算顺序，修饰符 volatile 可以防止这些改变的发生。

const 和 volatile 可以一起使用。例如，如果假定 0x30 是仅由外部条件改变的端口值，那么下面的说明即可消除用户希望防止的所有偶发副作用。

```
const volatile unsigned char * port = 0x30;
```

2.6 存储分类符

C 语言提供了四个存储分类符。它们是：

extern
static
register
auto

它们告诉编译程序如何存储变量。存储分类符用在变量说明之前，其一般形式是：

```
storage_specifier type var_name;
```

2.6.1 外部的

由于 C 语言允许将分别编译的大程序的各个模块链接在一起，以提高编译速度，协作大项目管理，因此必须将程序需要的所有全局变量通过某种方法告知所有文件。在 C 语言中，一个全局变量可以被多次说明(但不提倡)；在 C++中，一个全局变量仅能说明一次。然而，如何将程序使用的全局变量告知所有文件呢？方法是在一个文件中说明所有全局变量，在另一个文件中用 extern 再次描述它们，例如：

文件 1

```
int x,y;
char ch;
main()
{
   .
   .
   .
}

void func1(void)
{
   x = 123;
}
```

文件 2

```
extern int x,y;
extern char ch;
void func22(void)
{
  x = y/10;
}

void func23(void)
{
  y = 10;
}
```

在文件 2 中，全局变量表是从文件 1 中复制过来的，并将分类符 extern 加在说明之前。extern 告诉编译程序：如下变量的类型和名字已在别处说明过了。换言之，extern 让编译程序知道全局变量的类型和名字，而不必再次为它们分配内存。当将两个模块连到一起时，链接程序将解决对所有外部变量的引用。

关键字 extern 的一般形式是：

```
extern var_list;
```

其中，var_list 是在别处说明过的、用逗号隔开的变量表。

当在函数内部使用一全局变量，而它又是在同一文件内说明时，可以选用 extern，其实这样做很少见。下面的程序段示出了这种用法：

```
int first,last;     /* global declaration of first
                       and last */
main()
{
  extern int first;     /* optional use of the
                           extern declaration
}
```

虽然 extern 变量的定义可以在同一文件中作为全局说明出现，但这是不必要的。C 语言的编译程序发现未说明的变量时，将在全局变量中寻找与其匹配的变量，如果成功，则采用之。

2.6.2 静态变量

静态变量无论在函数中或在文件中都是稳定的变量。与全局变量不同，静态变量在它们的函数或文件外是不可知的，但它们的值在两次调用之间是保持不变的。当编写通用函数和函数库时，这一特点很有用。静态局部变量和静态全局变量的功效不同，下面分别加以介绍。