引　论

本书第一章至第九章介绍了C++语言的全部功能。读者从中可学到传统的C++和面向对象的编程技术，以及类、单一继承、多重继承、磁盘文件、友元、操作符重载、流和C++语言的其他特性。在第十章，读者将可以看到C++库中的150个以上的函数说明。在读者完成全部课程的学习之后，就可以以第十章的函数说明作为指南来编写自己的C++程序了。

本书中每个程序的源代码都由 Walter Bright 和 Zortech 编写。

请阅读本引言中的最低配置说明、如何安装C++编译器、如何从本书获得更多知识等几部分内容，然后，读者就可以打开第一章并准备编译第一个C++程序。

0.1　需　求

以下各节列出了必要的和可选的硬件及软件。如果在安装或使用本书的程序时遇到麻烦，请对照以下各节检查计算机是否满足以下这些最低配置要求。

0.1.1　硬件需求

- IBM PC，XT，AT，PS/2 或 100%兼容系统
- 384K 至 512K 可用 RAM
- 硬盘驱动器(推荐)
- 5.25 英寸软盘驱动器(或 3.5 英寸软盘驱动器)

0.1.2　可选硬件

- 打印机
- 彩色监视器
- 对没有硬盘的系统，至少应有一个容量为 1.2MB 或 1.44MB 的软盘驱动器。最好具备硬盘驱动器

0.1.3　软件需求

- MS-DOS 或 PC-DOS 2.0 或以上版本(有些程序要求 DOS 3.0 或以上版本)
- 所有包含在本书中的其他软件

0.1.4　可选软件

- 读者常用来修改文件的文本编辑器。如果读者没有编辑器，可使用本书提供的共享 VDE 编辑器
- Zortech C++ 2.1。如果读者已拥有完整的开发系统，当然可以编译程序。如果读者

没有并发系统，则不能编译和运行本书中的程序(注：对 Zortech C++以后的版本，在编译程序前可能要修改某些程序，详见第九章)。

0.2 安装 Zortech C++编译器

确保计算机拥有 3MB 可用硬盘空间，否则就只能安装到大存储容量的软盘中。先格式化空盘，然后把#1 磁盘放入驱动器 A:，键入以下两行命令：

```
a:
install
```

这将运行自动安装程序 INSTALL。阅读接着出现在屏幕上的指令，要终止 INSTALL，可以随时按<Esc>。再次安装该软件时需重复以上命令。

安装过程中将要求选择驱动器符，最好选择硬盘驱动器。还将要求提供目录名，\LCPP 是缺省名，也可以选择其他合法的路径名称，但不能把该软件安装在硬盘根目录下。最好是按<Enter>键使用所建议的\LCPP 路径名。

对于软盘安装，可以用反斜线(\)指定一个路径名。这样可保留一定的空间将编译器、编辑器和链接器文件转换到磁盘根目录下。这样做仅适用于在高容量软盘上安装，不适用于硬盘安装。

安装结束时，INSTALL 将运行 READ.EXE 程序，以显示 README.TXT 文本文件。读者也可在 DOS 下运行 READ(此文件在#1 盘上是解压缩文件)，然后在带有 READ.EXE 的当前目录下键入 read，选择 READ.TXT 或其他文本文件来阅读。使用<Cursor>和<Page>键浏览，并按<Esc>键返回目录显示屏或 DOS(如果在 DOS 符下敲入 read readme.txt，READ 将不列出目录而直接显示所选定的文件)。

READ.EXE 的源码被列在第八章，READ 正是读者学习怎样用C++编写许多程序的范例之一。

0.2.1 硬盘安装

在安装到硬盘并阅读了 READ.TXT 文件之后，C:\LCPP 成为当前目录。如果不是，可按<Esc>键，如果必要，可返回 DOS，然后键入 c:和 cd\lcpp 或类似的命令进入新创建的目录下(如果选择了不同的驱动器和路径名，则在这些指令和本书其他地方出现的 C:\LCPP 处使用所选择的盘符和路径名)。

使用 DIR 命令在目录中查找是否有文件 LCPP.EXE，LIB.EXE 和 VDE.EXE。这是三个具有自释放功能的程序，我们需从中提取若干文件。键入命令：

```
lcpp
lib
vde
```

如果读者已有自己的文本编辑器，并且不打算使用本书提供的 VDE 编辑器，则可跳过 VDE 命令。但你可能希望把它拷贝到另一张盘上保存起来，因为 VDE 是一个功能强大的文本编辑器和字处理器(在 VDE.EXE 文件中还包含如何使用该编辑器的完整文档)。

在释放过程中，当自释放程序在当前目录中释放其内容时，屏幕上会出现几行句点和小写的英文字符 o。完成以上过程后，读者可删除以上三个文件(它们已没有用处了)，键入：

```
del lcpp.exe
del lib.exe
del vde.exe
```

如果在提取之前误删除了某个文件，就必须重新运行 INSTALL，再次安装所有的文件(安装程序对安装次数没有任何限制)。

注：跳过以下两节，阅读"测试安装"一节。

0.2.2　大容量软盘安装

在 1.2MB 或 1.44MB 软盘上安装了文件之后，从自释放程序中抽取文件 LCPP.EXE，LIB.EXE 和 VDE.EXE 的命令与硬盘略有不同。首先将安装过的软盘插入驱动器 A:，将一张格式化后的空盘插入驱动器 B:，然后键入下面给出的命令并键入<Enter>键：

```
a:
lcpp /eb:\
lib /eb:\
vde /eb:\
```

这些命令将从 3 个.EXE 文件中提取文件，并存至驱动器 B:中的软盘上。你可以接着将释放后的文件拷贝到其他盘上，以用于编译或编辑程序。最后，从安装过的磁盘上删除文件 LCPP.EXE，LIB.EXE 和 VDE.EXE。

注：跳过下一节，阅读"测试安装"一节。

0.2.3　360K 软盘安装

首先，我们不鼓励读者不用硬盘而仅用两个 360K 的软盘来运行 C++编译器。下面我们给出实现这一过程的步骤。

首先在别的硬盘或大容量软盘上进行安装，然后格式化 3 张软盘，并分别贴上标签 LINKER，COMPILER 和 EDITOR。把以下文件拷贝到 LINKER 盘上：

```
blink.exe
pls.lib
yls.lib
```

把以下文件拷贝到 COMPILER 盘上：

```
*.h(即 page.h 和 emm.h)
*.hpp
ztc.com
ztc2.exe
ztcpp1.exe
```

并把以下文件拷贝到 EDITOR 盘上：

```
examples.vdk
```

```
vde.com
vde.doc
vdel54.upd
vinst.com
vinst.doc
wp.vdf
ws4.vdf
```

当然，还需要一些软盘用于拷贝源程序，即拷贝 SOURCE，ANSWERS 和 LIB 三个目录下的所有子目录和文件。

这里读者在编译和连接过程中分别插入不同的软盘就可以编译程序了。显然，这种配置是很不方便的，因此我们还是推荐读者使用硬盘，而且如果没有硬盘，某些大程序根本无法编译。

0.2.4 测试安装

安装并释放所有文件之后，便可开始使用C++编译器。首先，需要改变 PATH 语句，建立两个环境变量，这将使你可以编译任何目录中的程序，并让编译器和连接器找到编译和连接过程中所需的各类文件。

在软盘根目录下编辑或创建文本文件 AOTOEXEC.BAT。改变或插入一条 PATH 语句，使它成为：

```
path c:\dos;c:\lcpp
```

在 PATH 语句中，还可以增加其他路径，中间用分号隔开即可。另外在 AUTOEXEC.BAT 中加入下面两行语句：

```
set include   c:\lcpp\include;c:\lcpp\lib
set lib=c:\lcpp
```

这些命令产生两个环境变量 INCLUDE 和 LIB，它们能告知编译器和连接器如何找到在连接和编译时所需的各类文件。在 INCLUDE 变量中含有\LIB 目录，在该目录中有类库的源文件，这些将在第六章和第七章中加以说明。需要说明的是，在 INCLUDE 中必须增加该路径名，这样在其他章节中给出的程序才能找到保存在硬盘里的类库。

将修改后的 AUTOEXEC.BAT 存盘后，再重新启动计算机。若启动后出现"Out of environment space error"错误，则应修改根目录里的文件 CONFIG.SYS。把下面一行命令加入 CONFIG.SYS 中：

```
shell=command.com /E:512 /P
```

再次启动计算机后，SHELL 里的 512 就会给扩充的 PATH，INCLUDE 和 LIB 变量分配更多的空间。读者也可根据自己的实际需求来设定这个字节数。

0.3 使用 VDE 共享编辑器

VDE 编辑器是为那些没有文本编辑器的读者准备的。利用 VDE，读者可以修改、浏览

程序和建立新程序。

在使用编辑器前，应在 C:\LCPP 目录下键入命令 vinst，接着按屏幕上的指令将软件安装到计算机中，再键入 S 以将编辑软件配置存入硬盘，返回 DOS 提示符。然后，再转到含有源程序文件的目录下(例如 C:\LCPP\SOURCE\C01)，键入命令：

```
vde welcome.cpp /n
```

文件 WELCOME.CPP 可以是一个现存文件，也可以是创建的新文件。/n 选项告诉 VDE 以 ASCII 形式读或写文件。如果存储一个已编辑的文件而不用/n，可能得不到编译结果。如果只使用 VDE 编辑程序文件，可以使用 VINST.COM 改变缺省文件类型为非文档文件。有关如何改变 VDE 的进一步内容，可阅读文本文件 VDE.DOC 和 VINST.DOC。要打印这两份文件，确定打印机已运行并具备足够的打印纸，然后键入命令：

```
type vinst.doc >prn
type vde.doc >prn
```

注：不要使用 READ 程序检验 VDE 的 DOC 文件，这些文件太大，READ 无法检验。可以尝试修改 READ 以适应大文本文件，参见第八章。

如果读者熟悉 WordStar 或一直使用 Borland Turbo Pascal，Turbo C 或 SideKick，可以正确使用 VDE。VDE 程序指令可以在 WordStar 和 Borland 编辑器中编辑。

要退出 VDE，可键入<Ctrl>-KQ。如果想改变当前文件，按 Y 键以存储改变的文件或按 N 键放弃存盘。存盘而不返回 DOS，按<Ctrl>-KS。要退出编辑器，保存当前文件并返回 DOS，按<Ctrl>-KX。

0.4 编译程序

测试安装效果的最好办法是编译几个程序。这里介绍一下读者在本书的后几章节中如何编译源程序，这些内容将帮助你着手编译：

- 在 DOS 下，确定 PATH、INCLUDE 和 LIB 变量已正确设置。在 DOS 提示符下键入 set，即可查看所列出的变量。在 PATH 中含有 C:\LCPP 目录和其他两个变量列出的目录，这在前面已作过说明。
- 键入 cd\lcpp\source\co1 打开包含第一章程序清单的目录。除了第六章和第七章，用同样方式打开其他各章相应的目录(第六、七章的文件存储在 C:\LCPP\LIB 下)。
- 键入 ztc welcome 以编译和连接 WELCOME.CPP 程序和第一章的程序清单 1.1。读者将在屏幕上看到以下几行：

```
ztcpp1  oztc_APC . tmp welcome
Zortech C++ Demo Compiler
ztc2 ztc. APC . tmp _owelcome. obj
BLINK welcome/noi;
```

- 现在已经完成了在当前目录下的 WELCOME.EXE 文件的编译。键入 welcome 即可运行该程序。
- 键入类似的 ztc 命令，编译本书中的大部分程序。某些更复杂的程序要求特定的命令

去编译，在需要的地方本书将予以说明。

- 如果接收到错误信息，特别是"Error;'ztc1' not found"信息，一定要确保，文件在当前目录下(ZTC1 是 Zortech 的 C 编译器，它不包含在配套磁盘中，读者也不需要。然而，ZTC 在某些时候试着去运行 C 编译器，从而导致错误信息。读者只要修正自己的输入并再试一次就可以了)。

在编译完 WELCOME 程序之后，读者可能想去尝试编译更为高级的程序实例。按照以下步骤编译并运行第五章中的程序：

- 键入 cd\lcpp\source\c05 以改变至第五章的目录。
- 在 DOS 下，键入 zz 运行 ZZ.BAT 文件。此批文件包含所有编译和连接程序所需的指令。
- DOS 提示符出现后，键入 elevsim 运行该程序。键<Esc>退出。

读者可以在 C:\LCPP\SOURCE\C04 下编译和运行 POPUP.CPP 程序。使用 ztc.popup 命令编译该程序，然后运行并键入风次<Spacebar>清除弹出式窗口。读者也可以在 C:\LCPP\LIB 下运行 ZZ.BAT 去编译 WINTOOL 程序。当读者学习了第六章之后，即可以使用 WINTOOL 设计自己的弹出式窗口了。运行 WINTOOL 之后，按<Esc>键返回 DOS。

注：读者阅读各章时，使用 CD 命令去改变 C:\LCPP\SOURCE\C0n 中的 n，n 表示每一章的章号。第六、七章的程序清单存储在 C:\LCPP\LIP 下。在 C:\LCPP\ANSWERS 中可找到附加的程序清单。使用 VDE、编辑器或 READ 去浏览目录中的程序清单。读者可键入 read 并选择一个程序进行浏览。选择一个目录名可改变那个目录。可按照本书给出的指令在屏幕上测试程序源码。

第一章　C++概述

这一章将帮助读者了解什么是C++，使用C++有何方便之处。如果读者熟悉Pascal或C就更好了，可以快捷地阅读本章以了解C++与那些流行的语言有何不同。我们假设读者具备一些基础知识，如位(bit)和字节(byte)的含义、DOS命令、运行程序的步骤等。

我们将从C++程序的各类成分开始，它们是本书所有程序共享的。我们将简单地概括一下基本概念：常量、变量、输入和输出以及操作符等。几乎所有C++程序都使用以上一个或多个概念，所以有必要花时间去阅读并运行一些程序实例。

1.1　C++结构

所有C++程序都是相关的，也即它们的组织结构共享某些相同的"构件"。最好的学习方法是剖析一个像WELCOME.CPP(清单1.1)那样的子程序，以了解这些"构件"是怎样联系在一起的。把这个文件装到硬盘上或键入到编辑器中。然后，在DOS提示符下使用编辑器中的相应指令键入ztc welcome以创建WELCOM.EXE。通过键入该程序的名字来运行它，或使用习惯的手段运行该程序。

注：从现在开始，编译和运行程序实例均使用相同的方法，但要替代像WELCOME.CPP等相应的文件名。除非程序要求特定的编译指令，在以后的讨论中不再重复编译和运行程序的步骤。

程序清单1.1　WELCOME.CPP

```
1:  #include <stream.hpp>
2:
3:  main()
4:  {
5:      cout << "Welcome to C++ programming! \n";
6:  }
```

3—6行是这个小程序的主体，抽掉第5行：

```
main()
{
}
```

被称为函数——C++最重要的一个特性。在程序清单1.1中，只有一个函数名main。在其他C++程序中，可能有几十、上百个以同样形式编写的具有不同名称的函数。不管一个程序有多少函数，它必须仅有一个称为main的函数。运行C++程序时，总是从main开始。

注：不要把函数与数学函数相混淆。C++中的函数是一组指令，这组指令完成一系列的动作。一个函数的作用完全由编程者描述。

函数名后的圆括号告诉编译器，该函数没有从外部接收到信息。下面，读者将学习如何在圆括号中列出参数，这些参数要求函数处理所含的信息。例如，读者可能通过 main 的命令行选项让程序使用一个字母或一个文件名。

函数名和圆括号以及左、右大括号中的内容称为函数体。返回到程序清单 1.1，可以看到 main 的函数体只包括一行：

```
{
    cout<<"Welcome to C++ programming! \n"
}
```

因为，这对大括号，编译器知道所括起的程序行（称为语句）属于函数 main。通常，一条语句表示程序运行时完成的一个动作。其他语句可以是说明、定义、表达式或指令，它们在编译器编译时被执行。特别地，说明语句提供编译器某些诸如新数据类型格式的信息。定义语句是在存储器中为存储的值创建一个空间，可能是一个预先说明的数据类型的变量。表达式（像 1+m）求值后得到一个值，其他指令可能会更改编译器的工作方式。根据不同的条件，它们可能选择不同的程序段来编译。不必担心记不住这些术语。作者和有经验的程序员通常混合使用"说明"和"定义"这两个词，读者不必拘泥这些描述。现在，最重要的是理解C++中大括号的作用，即将一或多条语句、表达式、说明或定义组合到一个单元中。总之，大括号和其中的程序称为块。

1.1.1 流

当运行实例程序时，可以得到信息"Welcome to C++ programming!"。有两种方法输出此信息，但最常用的是输出流。我们可以看看实例程序中的输出流语句：

```
cout << "Welcome to C++ programming ! \n";
```

输出流中最先出现的是 cout，它是"character output"的缩写（顺便提一句，该单词通常读作"see out"，而不是"kout"）。对象 count 是输出的目的地，即我们希望将程序要显示或打印的信息以字符形式发送到此地方。<<流输出符，它是一个符号但却是两个符号构成的。这个符号指向 cout，表示后边的内容流向符号左边的目标对象。在程序清单 1.1 中程序流的源就是串：

```
"Welcome to C++ programming ! \n"
```

双引号表示让编译器逐字取括起来的字符，即不是把括号括起来的字符作为程序语句或指令进行处理。\n 符号称为换行字符。在串中（并不一定代表末尾）插入\n 导致程序在终端或打印机上开始一个新行。

学习输出流的好方法是自己使用一下。把程序清单 1.1 装入编辑器，然后在 main 体中{}之间加以下几行：

```
cout <<"Welcome";
cout <<"to";
cout <<"C++ programming! \n";
```

注意，这三行产生与原来的程序一样的结果。因为前二行没有在末尾加\n，程序以一行显示它们。试着在每一行末尾加\n，看看会产生什么结果？

另一种产生类似结果的方法是使用带有多个成分的一条输出流语句，可以写：

```
cout << "Welcome" << "to\n"
     << "C++ programming\n";
```

请观察在同一程序实例中运行以上两个例子的结果。在第一个例子中，有三条语句末尾带有分号——C++的语句结束符。C++程序中所有语句都必须用分号结束。第二个例子中，有三个串，但只有一条被分成二行的语句。C++忽略文本中的行结尾且不管以一行还是多行写语句。

通常，可以以下列形式写输出流语句：

```
cout <<a<<b<<...<<c;
```

其中 a，b，c 流向输出流对象 cout。如果必要，可以串接许多项，可用一行或几行键入它们：

```
cout <<a
     <<b
     <<...
     <<c;
```

在以上各种情况中，都以一个分号结束一条语句，而不是一行。也可以把长串分成几行键入，例如，在程序清单 1.1 中插入：

```
cout << "There was a young lady named Bright, \
Whose speed was far faster than light; \
She set out one day,\
In a relative way,\
And returned home the previous night.\n";
```

这只是一个输出流语句和一个串。每行末尾以反斜线告诉编译器把带有反斜线的文本行的先前字符联起来。在反斜线后编译器不留出空间。运行该程序时，会看到五行以一行或二行连起来显示，\n 表示换行符，即一行结束。还要说明的是，在这个输出流语句中，仅有两个双引号，一个在开头，另一个在结尾。字符串的长度也没有任何限制。

注：字符串长度无限制是指编译器字符串长度无限制，当然用户也不能在 20MB 的硬盘上输出一个 100MB 的字符串。

我们在后面还将涉及流、字符串和字符，那时读者可以根据程序清单 1.1（WELCOM.CPP）来显示各类字符串，并在字符串中插入换行符\n，以进一步了解\n 的作用。

1.1.2 分号的说明

在学习 C++时，一个较为困难的问题是如何使用分号。在程序清单 1.1 中，只有第 5 行结束时有一个分号，而其他行都没有。原因很简单，第 5 行是一条语句，而语句必须以分号结束。如前所述，分号是指一条语句的结束，也就是说通知 C++编译器一条语句在分号处结束。因而读者可将一条语句写在很多行里，如：

```
cout <<"This is"
     <<"one statement"
     <<"on three lines\n";
```

因为仅有一个分号，C++编译器认为它和以下语句是一样的：

```
cout<<"This is one statement...\n";
```

读者不必刻意去记住大量的关于分号的规则，开始时也可能该加分号的地方没加，而不该加的又加了。学习加分号的技巧是了解C++中的元素，以掌握哪些元素是需要加分号的。另一方面，加分号实际上是很自然和明显的。

注：用错了分号后，编译器会显示出错信息和错误原因。即使有经验的程序员也会有一两次误用分号，因而，若刚开始编程出现很多错误时，不必为此沮丧。

1.1.3 注释说明

程序中的注释是为了让其他程序员和编程者自己能更方便地阅读程序。若需要在程序中插入一段注释，要在开始和结束时加上双字符/*和*/。下面给出几个注释的实例：

```
/* Title:MYPROGRAM by Mr. Software */
/* Revision 1.00B--all bugs fixed(I hope) */
/* Original author skipped town */
```

这类注释还可以扩展成多行注释，例如：

```
/* welcome.cpp by Tom Swan
 * Date:1/1/1991
 * Revision :1.0
 */
```

这里需要说明的是，第二和第三行的*号不起任何作用。编译器工作时，只根据第一行开始的/*和第四行结束时的*/将这四行作为一条注释处理。

同样，读者也可以将注释插在某一条程序语句中，但这样做容易出错。例如在程序清单1.1中插入以下注释：

```
cout << "No comment" /* Oh,yeah? */ << "here! \n";
```

因为编译器会滤掉注释/*Oh,yeah? */，这条语句运行时只显示No Comment here!

用/*和*/括起来的注释称之为C风格的注释，在C和C++中都可以用。除此之外，C++还提供另一类注释(C中不能用)，它们只能以双斜线开始，例如：

```
// Wellcome.cpp by Tom Swan
// Date:1/1/1991
// Revision:1.0
```

C++的这类注释只有起始字符//，因而它只能放在一行程序结束的位置，而且也不能将一条注释写成多行。例如：

```
cout << "Your name?";          // Prompt for user's name
```

该语句显示"Your name?"，语句后的注释用于说明该语句。

C++和C注释能够嵌套使用，这在调试程序时很有用处。我们可以把一条程序语句用C或C++暂时注释起来，使之不产生任何动作。例如：

```
/*
 cout<<"Doesn't display";          // output one string
 cout<<"Neither does this";        // output another string
 */
```

因为第一行有C注释的起始符，最后一行有C注释的结束符，所以编译器把第二行和第三行全部滤去，其中包括//后面的C++注释。

若在调试程序时需要暂时删掉某一行，只需在该行前面插入//即可。例如：

```
// cout << "Your age?";        // Prompt for user's age
```

cout前面的//使这行程序全部成为注释，其中也包括注释//Prompt for user's age。程序清单1.2(COMMENT.CPP)为读者示范了如何使用C和C++注释。其中包括用//建立区分框，以使用/*和*/产生多行注释。

程序清单1.2 COMMENT.CPP

```
1:  // comment.cpp -- Demonstrates C and C++ comment styles
2:
3:  #include <stream.hpp>
4:
5:  //////////////////////////////////////
6:  // Author    : Tom Swan
7:  // Revision  : 1.0   07/13/1990   Time: 03:50 pm
8:  // Purpose   : Demonstrates C++ comment styles
9:  //////////////////////////////////////
10:
11: main()
12: {
13:    cout << "A Brief C++ Commentary\n";   // Display title
14:    cout << "\n";   // Display blank line under title
15:
16: /* This paragraph demonstrates that
17:      C-style comments can occupy more
18:      than one line. */
19:
20:    cout << "// This is not a comment.\n\n";
21:
22:    cout << "/* This also is not a comment. */ \n\n";
23:
24:    cout /* This is a comment. */ << "This text is displayed.\n";
25: }
```

运行COMMENT时，会看到编译器将忽略任何注释符中的串，这是因为串的双引号告诉编译器，不能把串作为程序，而是逐字取文本。这对程序清单同样适用。串的双引号中的注释符同其他字符一样被处理。

从现在起，本书全部程序清单都使用C++注释作开始，类似前面例子的第一行。该行用文件名标识程序并描述程序的功能。

第二个注释例子见程序清单1.3，它使用了一个C注释和一个C++注释，这只是为了解释这个程序什么也不做。

程序清单 1.3 NOTHING.CPP

```
1:  // nothing.cpp -- A mere shell of a program
2:
3:  main()
4:  {
5:      /* This program does nothing! */
6:  }
```

NOTHING 程序是C++中最小的程序(程序忽略注释)。在 DOS 提示符下列出目录，可看到 NOTHING.EXE 文件的大小。该文件占 3650 字节，这表明编译器的效率还是较好的，有些编译器编译这个程序后会产生 8K，10K 或更大的 EXE 文件。

1.1.4 标识符

标识符是可写入程序的唯一符号。函数名称 main 就是一个标识符，它按名标识函数。选好标识符是编写可用程序的最基本训练。

C++识别许多如 main 那样的保留标识符。标识符包括大小字母、数字和下划线，并且可达 127 个字符长。另外，标识符必须以 A－Z 这 26 个大小写字母开始，如 fn1 或 Catch22。标识符 123abc 是非法的，abc123 是合法的。

也可以使用下划线把较长标识符中的名词分开，如 head_count，bottom_of_the_barrel。通常在 C 语言中不能在标识符首位放下划线，然而C++允许如_value 和_overandunder 这样的标识符，只要不与类似开头的系统标识符相冲突。如果读者不使用以下划线开始的标识符，就永远不会产生与系统标识符相冲突的标识符。

1.1.5 标识符大小写

在C++中，大小写是有区别的。例如 keyPress，keypress 和 key_press 是三个不同的标识符。因此在使用键盘键入标识符时，最好全部用大写，或全部用小写，以免出现不必要的麻烦。

大多数 C 和C++程序员用小写字母输入标识符。全部大写的单词，如 XYCOORD 和 PAYMENT 读起来较费劲。而小写字母 xycoord 和 payment 在屏幕上读就更方便一些。还有些程序员采用大小写混合使用，它们不写 array_of_names 或 arrayofnames 而写 arrayOfNames。小写字母用何种方式，一个人的习惯各有不同，但具体使用何种方式一定要考虑是否好用且不易出错。

1.1.6 关键字

关键字是C++为自用而保留的标识符。如关键字包括 auto，float，signed，void 和 while。这些关键字不可作为用户自定义的标识符，它们是有特定含义的并且不能改变。

附录 A 列出了所有C++的保留关键字。读者在学习本书的过程中，会遇到全部关键字，并学会如何使用C++的关键字。

1.1.7 标点符

分号结束C++的语句，它是一个标点符。在前面学过的左、右圆括号也是标点符。

标点符是一类简写的关键字，它对编译器而言，是一种有特定含义的符号(可由一、二或三个字符组成)。C++标点符集包括以下符号：

```
#  ()  []  {}  ,  :  ;  ...
```

正如其他关键字和符号一样，当在本书中遇到它们时，读者应学会使用它们。它们的用法是非常直观的。

1.1.8 分隔符

分隔符有很重要的一点不同于标点符，它们是不可见的。正因为它们在任何地方均不可见，所以用空白描述它们。C++分隔符包括空格、制表符(tabs)、回车以及加进文本中的换行符。

通常，C++忽略空白，只要用户不使用空格分隔标识符、数字和其他应该连在一起的结构形式。所有空白对C++而言其含义是一样的，它可以解释语句为什么能写在多行。对C++，两个语句被软回车或空格分开时，其含义相同。事实上，代替程序清单1.1所表示的"锯齿形"程序风格，我们可写程序：

```
#include <stream.hpp>
main(){cout<<"Hello! \n";}
```

编译器仅关心函数main部分和其中的符号是否被正确地用于分隔和加标点。C++并不关心这些分隔是行终止符还是空格。显然，这种风格使程序更难读和难以调试。

1.1.9 头文件

前面给出的大多数程序和清单的开头都类似于：

```
#include <stream.hpp>
```

当编译器处理此行时，它去读取STREAM.HPP，即一种被用户编译器支持的头文件的内容，其结果与该文件文本就在程序该位置上存在一样。显然，由于每一个所写的程序都没有去拷贝STREAM.HPP，这大大地节省了空间，但也意味着在包含该文件的程序中就自动执行STREAM.HPP文件了。

典型头文件包含声明、定义和其他程序所需使用的指令。如果读者试着从程序清单1.1中删除#include语句，将接收到从编译器发出的错误信息。在这种情况下，因为cout和相关的程序设计被存储在STREAM.HPP文件中，所以程序保持不动。如果在程序中没有包含该文件，编译器不知道cout是什么或怎样去使用输出流。

关键字#include(确切他说是两个符号，标点符#和关键字include)是送给编译器的一个指令，随后跟着一个将要包含到此位置的磁盘文件名。大多数C++程序都是以一条或几条这种include语句开始的。尖括号<和>括起的文件名告诉编译器在存有.HPP和.H文件的标准include目录中寻找该文件，它通常被分配在INCLUDE环境变量指定的路径上。include语句可写成：

```
#include <stdio.h>
#include "newmath.h"
#include "program.h"
```

第一行含有一个存储在 include 目录中的 STDIO.H 文件名。第二和第三行含有名为 NEWMATH.H 和 PROGRAM.H 的文件。由于这两个文件是用双引号而不是用尖括号括起来的，编译器在当前目录而不是在 include 目录路径下查找这两个文件。

对编译器支持的标准头文件使用尖括号，对自己建立的文件使用双引号。读者将会在本书以后的程序清单中看到使用这两种表示的例子。

注：头文件名通常用扩展名.H(C 语言头)或.HPP(C++头)结束。扩展名并不特别重要，读者可以用三个字符的任何名字结束头文件名，但最好以方便为本，用.H 或.HPP 命名头文件，这样会很快地在磁盘文件中找到头文件。

1.2 变 量

在 C++中，变量是内存中被命名的存储单元(见图 1.1)。变量可以存储所有数据类型，即串、数字或多部分的结构。典型的变量都有一个描述其作用的名字，通常用户可使用任何自己喜欢的变量名。作为图示，名字指针指向存储该值的存储单元，但可以忽略这个事实，在程序中认为名字和值是一回事。

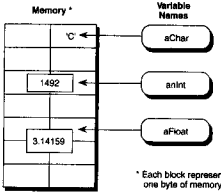

图 1.1 变量是一个在内存中被命名的存有一个值的存储单元

所有变量都有一个共同的特点：一个附加的数据类型。这意味着，对一个变量要求选择一个合适的名字，必须告诉编译器你希望它存储什么类型的信息。表 1.1 列出了 C++通用的数据类型，每种类型给出了例子、类型大小(以字符计)、类型变量值的范围。

表 1.1 通用 C++数据类型

数据类型	实例	大小	最小		最大
char	'c'	1	0	...	255(或 ASCII)
short	-7	2	-128	...	127
int	1024	2	-32,768	...	32,767
long	262144	4	-2,147,483,648	...	2,147,483,637
float	10.5	4	1.5E-45	...	3.4E38(近似)
double	0.00045	8	5.0E-324	...	1.7E308(近似)
long double	1e-8	8	同 double		

注：所有C++版本不一定具有存储相同字节量的数据类型变量。表1.1所列出的是针对本书编译器的信息。其他编译器可能列出相同数据类型的存储内容。然而，char变量总占用一个字节。

在程序中创建一个变量，以数据类型开始并以一个标识符或分号结束。例如，创建一个名为 yesno 的 char 变量，可以键入：

```
char yesno;
```

这称为定义，因为它定义了在存储器中存储一个变量的空间。在此情形下，一个字符占一个字节空间。用户可把该变量定义和其他变量定义插入到任何位置，如函数内部、外部、语句之间、块中等等。然而请注意，存储单元的定义影响编译器创建存储变量空间的方式，并且可能影响程序运行的方式。为此，本书将在函数 main 前和随后的 main 花括号后定义所有变量。

1.2.1 用定义预置变量

有两种预置变量（即给它们赋初值）的方法。第一种方法在大多数情况下是最好的，因为它连接了一个变量的定义和置变量的初值的赋值：

```
char yesno='Y';
```

它创建了 char 类型的 yesno 变量并把字符"Y"赋给该变量。"赋值"即"在存储器中该变量名表示的存储单元中存储"。这个定义和前面所述定义类似，只是以等号结尾，并且值被存在变量中。例如：

```
int counter=1;
float weight=155.5;
```

变量 counter 是 int 类型的并被赋初值 1。变量 weight 是 float 类型并被赋值 155.5。如果把这些定义插入到一个测试程序中，则可用

```
cout << "yesno=" << yesno << '\n';
cout << "couner=" << counter << '\n';
cout << "weight=" << weight << '\n';
```

语句显示它们的值。当编译这些语句时，编译器用从存储器恢复的关联值替代变量名 yesno，counter 和 weight。然后，编译器把这些值转换为文本形式，并用输出流语句显示。由于每一个变量都被限定了一个特定的数据类型，C++知道它将把 yesno 显示为一个字符，把 counter 显示为整数，weight 显示为浮点数。

程序清单1.4(VARIABLE.CPP)演示了如何根据表1.1预置所有数据类型的变量。每个变量定义以适用于每一种数据类型的形式赋每一个初值。输出流语句显示了这些值。

程序清单 1.4 VARIABLE.CPP

```
1:  // variable.cpp -- Common variables
2:
3:  #include <stream.hpp>
4:
5:  main()
6:  {
```

```
7:      char slash  = '/';
8:      short month = 3;
9:      int year  = 1991;
10:     long population = 308700000;
11:     float pi = 3.14159;
12:     double velocity = 186281.7;
13:     long double lightYear = 5.88e12;
14:
15:     cout << "Date = " << month << slash << year << '\n';
16:     cout << "Population of the U.S.A. = " << population << '\n';
17:     cout << "Pi = " << pi << '\n';
18:     cout << "Velocity of light = " << velocity
19:          << " mi./sec." << '\n';
20:     cout << "One light year = " << lightYear << " mi." << '\n';
21: }
```

当运行 VARIABLE 时，它显示了定义在 7—13 行上的每一个变量。15—20 行的输出流语句使用适用于这些变量的数据类型的形式显示这些值：

```
Date=3/1991
Population of the U.S.A=308700000
Pi=3.14159
Velocity of light=186281.7 mi./sec
One light year=5.88e+012 mi.
```

1.2.2 用赋值预置变量

另一种预置变量的方法是在定义变量后使用分离赋值语句。例如，替换程序清单 1.4 中的第 7 行，可以用下列定义创建变量：

```
char slash;
```

然后，在程序清后使用一个赋值语句在 slash 中存储一个值：

```
slash = '/';
```

编译器把等号忽略为一个指令：等号右边的值赋给等号左边的变量。亦即，编译器生成代码，在存储器为 slash 变量保留的存储单元中存储了用 ASCII 斜线字符表示的一个值(参见图 1.2)。

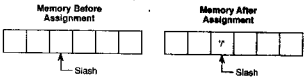

图 1.2 赋值语句 slash='/'；在为 slash 变量保留的存储单元中存储了一个 ASCII 斜线字符

读者也可以给程序清单 1.4 中的其他变量赋值。例如，在第 14 行后增加语句：

```
population = -500;
```

运行这个修改过的程序时，它显示了美国的人口为－500。虽然这个值是错误的，但它说

明了赋值语句可以替换变量的初值。这是变量的一个特点。通常在必要时，程序可以改变变量的值，并且每次程序只赋一个值给一个变量，新值替换了旧值。一个变量一次只接受一个值。

无论读者采用哪一种方法给变量赋值，在程序中预置变量是很重要的。如果忘记了给变量赋初值，变量将使用内存为它们保留的存储单元中的任一值。未预置变量是造成各种程序错误的原因，从错误的银行差额到失败的卫星轨道。为避免用户程序出现这类问题，应确保对所有变量赋初值。

1.2.3 变量的作用域

语句可以引用变量，但仅当与变量在同一作用域中或在同一层上，变量的作用域可扩大到它的定义块界。

图 1.3 给出了一个样本程序(不包含在磁盘中)，它含有在 main 函数中的嵌套块。这个程序是模拟程序，实践中是不能以这个方式写程序的。然而，它却是正确的 C++程序(它可编译)，并且它说明了可见变量的作用域范围。

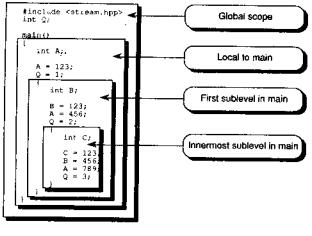

图 1.3 变量的作用域仅可扩展到定义它的块界

图 1.3 例示了几个关于作用域的重要事实。第一，整数 Q 拥有全局使用域，包括整个程序。这意味着所有语句可访问，不管它们出现在程序中何处。Q 是全局变量，它的作用域是整个程序。

在 main 括号中的定义，如整数 A，是局部于 main 的。所以，只有 main 内的语句可以使用 A，也即在由括号限制的 main 块中(同一作用域中)的语句可以使用 A(在图 1.3 给出的程序中没有这种语句)。事实上，在 A 的定义块之外，A 根本不出现在存储器中。

离 main 再远一些的块是嵌套块，在图 1.3 中标为"main 中的第一子层"。由于嵌套块是嵌套在 main 的主块中的，所以嵌套块可以使用它自己定义的 B 和在外层定义的整数 A，也可以使用全局变量 Q。嵌套块总是可以"看到"在它们的外层块中的定义。然而外层的块不可以"看到"嵌在内层块中的定义。这表示，如果在 A=123 后插入一条语句，则程序编译会出错：