

# C/C++深层探索

姚新颜 编著

人 民 邮 电 出 版 社

## 图书在版编目(CIP)数据

C/C++深层探索/姚新颜编著. —北京：人民邮电出版社，2002.12

ISBN 7-115-10915-X

I. C... II. 姚... III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2002)第 092498 号

# 内容提要

本书试图通过近 50 节的内容带领读者从各个方面去把握 C/C++的语法、语义，并通过分析 C/C++编译器生成的汇编代码，使读者明白 C/C++的某些底层实现，从而更加深入地理解 C/C++的概念、规则和不足。

本书没有面面俱到地讲述如何使用 C/C++语言编程，而是深入剖析了 C/C++语言的历史变化、各项特性及底层实现。本书试图引领读者不仅在 C/C++语言的范围内学习，而且更侧重于从汇编语言的角度、从编译程序和链接程序的角度去了解、分析 C/C++语言。通过本书，希望读者不仅能看清 C 语言的现在，还会知道 C 语言的过去，以及把握 C 语言的未来。

本书适合已经初步掌握了 C/C++的语法，希望从一个更深的层次去了解 C/C++的读者。

### C/C++ 深层探索

- 
- ◆ 编 著 姚新颜  
责任编辑 王文娟
  
  - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号  
邮编 100061 电子函件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
读者热线 010-67180876  
北京汉魂图文设计有限公司制作  
北京 印刷厂印刷  
新华书店总店北京发行所经销
  
  - ◆ 开本：800×1000 1/16  
印张：19.5  
字数：332 千字 2002 年 12 月第 1 版  
印数：1— 000 册 2002 年 12 月北京第 1 次印刷

ISBN 7-115-10915- /

---

定价：36.00 元

本书如有印装质量问题，请与本社联系 电话：(010) 67129223

# 前言

如果你已经初步掌握了 C/C++ 的语法，开始渴望从一个更深的层次去了解 C/C++ 的一些底层实现，又苦于国内很少有这方面的参考资料，那么，请翻一下这本书吧，你会有所发现的。

现在国内的计算机图书市场上，讲述 C 语言的书籍数量很多，但其定位主要都是面向初学者，很少能够做到深入剖析语言的历史变化、各项特性及底层实现，更不用说让读者读完之后有一种豁然开朗的感觉。

本书将引领读者不仅仅在 C 语言的范围内学习 C 语言，还要从汇编语言的角度、从编译程序和链接程序的角度去了解、分析 C 语言。通过本书，读者不仅仅会看到 C 语言的现在，还会知道 C 语言的过去，以及把握 C 语言的未来。

C/C++ 已经有 20 多年的历史，在这个过程中，编程语言的设计理念有很大的发展，作为使用最广泛的系统编程语言的 C，以及作为最主要的编译型面向对象编程语言 C++，都在不断变化。如何抓住 C/C++ 的内在脉络，穿过各种表象去把握住语言的真正设计思想，是每一个认真的 C/C++ 程序员将要面对的考验。

这本书是一次尝试。

要达到这个目标，身为作者我必须严肃对待，另外读者也要在阅读过程中不断思考、提问和自己动手寻找答案。这个世界上没有一本书会告诉你所有的事情，让你不用动脑筋光看书就真正学会知识。所以，当你阅读本书碰到困难时，千万不要轻言放弃，那些内容绝对不是你想象中那么难，花时间琢磨一下或许就会柳暗花明。

另外，本书的示例平台是 GNU/Linux 系统，不是我们常见的 Windows 系统；编译器是 GCC 而不是 Visual C++ 或者 C++ Builder。Linux 对于相当一部分读者来说可能比较陌生，但考虑到目前我们已经有很多种途径去学习、使用 Linux，我还是坚持自己的这个决定。

本书分五大部分：

Part I 讲述一些 C 语言的基本概念；

Part II 进一步讲解 C 语言的一些难点；

Part III 为大家分析一些容易被忽略的特性；

Part IV 介绍了 C 语言的最新发展；

Part V 对 C++作了某种程度的讨论。

虽然我建议大家按部就班地从头开始看，但如果读者对某些章节的内容已经相当熟悉，则可以单独挑出感兴趣的章节进行阅读。

大多数章节都会附有充足详细的注释以帮助读者更加容易地理解正文的内容，具体的格式对应关系很简单，譬如第 03 节正文第一处需要注释的地方会用“<sup>[1]</sup>”标记，读者可以在该节后面的“[FN0301]”处找到注释。至于所有建议的参阅资料都可以在附录A中找到对应的条目。

最后，请容许我在这里对下面的人们表示衷心的感谢：

首先是我的父母，在整个写作过程中，他们给予我最多的帮助；其次是曾燕燕女士，她无偿地为我提供了写作过程中需要参考的所有外文资料；易峰、谌贻荣、曹文花、叶红宁、肖颖琳、湛庆延、刘睿和肖涛都在我陷入困境的时候热情地伸出友谊之手。

对本书的所有建议和批评可以发到以下电子邮箱：fred@263.net

谢谢！

编者

2002 年 8 月

# 目 录

|                      |     |
|----------------------|-----|
| Part I .....         | 1   |
| 00 预备知识.....         | 3   |
| 01 C/C++语言的发展简史..... | 9   |
| 02 关于字节顺序.....       | 15  |
| 03 调用函数、栈.....       | 19  |
| 04 变量的可见范围与生存期.....  | 27  |
| 05 变量的声明和定义.....     | 33  |
| 06 编译和链接.....        | 41  |
| 07 外部变量的链接性质.....    | 45  |
| 08 静态内部变量.....       | 51  |
| 09 函数的声明和定义（上） ..... | 55  |
| 10 函数的声明和定义（下） ..... | 65  |
| 11 函数的链接性质.....      | 73  |
| 12 使用头文件.....        | 81  |
| Part II.....         | 85  |
| 13 静态库.....          | 87  |
| 14 动态库.....          | 93  |
| 15 简单类型的转换.....      | 97  |
| 16 复合类型.....         | 103 |
| 17 关于指针（上） .....     | 109 |
| 18 关于指针（中） .....     | 115 |

|                                                    |     |
|----------------------------------------------------|-----|
| 19 关于指针（下） .....                                   | 121 |
| Part III.....                                      | 125 |
| 20 词法分析.....                                       | 127 |
| 21 注释.....                                         | 131 |
| 22 优先级与运算顺序.....                                   | 135 |
| 23 友好的typedef.....                                 | 139 |
| 24 C-V限定词.....                                     | 147 |
| 25 字符串.....                                        | 153 |
| 26 void表示什么 .....                                  | 159 |
| 27 #pragma 与 _Pragma .....                         | 165 |
| Part IV.....                                       | 169 |
| 28 声明内部变量 .....                                    | 171 |
| 29 更严格的类型检查 .....                                  | 175 |
| 30 _Bool 的加入 .....                                 | 177 |
| 31 _Complex 与 _Imaginary .....                     | 181 |
| 32 内联函数 .....                                      | 185 |
| 33 变长数组（上） .....                                   | 199 |
| 34 变长数组（下） .....                                   | 203 |
| 35 可伸缩数组成员 .....                                   | 209 |
| 36 Designated Initializer 和 Compound Literal ..... | 217 |
| 37 Restricted Pointer .....                        | 225 |
| 38 增强的数值运算（上） .....                                | 229 |
| 39 增强的数值运算（中） .....                                | 237 |

|                      |     |
|----------------------|-----|
| 40 增强的数值运算（下） .....  | 245 |
| 41 字符集与字符编码 .....    | 251 |
| Part V .....         | 259 |
| 42 C++的函数 .....      | 261 |
| 43 名字空间 .....        | 265 |
| 44 C 和 C++的标准库 ..... | 271 |
| 45 模板 .....          | 277 |
| 46 外部对象的初始化 .....    | 283 |
| 附录.....              | 293 |
| A 参考资料.....          | 295 |
| B 网络资源.....          | 302 |

## Part I

- 00** 预备知识
- 01** C/C++语言的发展简史
- 02** 关于字节顺序
- 03** 调用函数、栈
- 04** 变量的可见范围与生存期
- 05** 变量的声明和定义
- 06** 编译和链接
- 07** 外部变量的链接性质
- 08** 静态内部变量
- 09** 函数的声明和定义（上）
- 10** 函数的声明和定义（下）
- 11** 函数的链接性质
- 12** 使用头文件



## 00 预备知识

本书的很多部分需要通过C/C++编译器输出的汇编代码来分析C/C++的内部实现，所以读者最好能够初步熟悉i386 平台的汇编指令及相关知识<sup>[1]</sup>。下面是本书中经常出现的几条汇编指令<sup>[2]</sup>：

### **PUSH SRC** <sup>[3]</sup>

例如：

```
push  eax
push  [ebx]
push  1234
```

这条指令的实际动作是：

```
esp ← esp - 4           //esp 的值减 4
[esp] ← SRC             //把 SRC 复制到 esp 指向的内存区域
```

### **POP DEST**

例如：

```
pop  ebp
pop  [ebx]
```

这条指令的实际动作是：

```
DEST ← [esp]           //把 esp 指向的内存区域的内容复制到 DEST
esp ← esp + 4           //esp 的值加 4
```

### **MOV DEST, SRC**

例如：

```
mov  eax, ebx
mov  [ebx], 1234
```

这条指令的实际动作是：

`DEST ← SRC`                      `//把 SRC 复制到 DEST`

## **LEAVE**

执行这条指令相当于连续执行以下两条指令：

```
mov  esp, ebp
pop  ebp
```

## **CALL SRC**

例如：

```
call ebx
call [ebx]
call 1234
```

这条指令的实际动作是：

```
push eip                      //eip 的内容入栈保存
eip ← SRC                    //把 SRC 复制到 eip
                             //然后 CPU 从 eip 指向的新区域读取、执行指令
```

## **RET**

这条指令的实际动作是：

```
pop eip                      //把 esp 指向的内容复制到 eip, esp 的值加 4
                             //然后 CPU 从 eip 指向的新区域读取、执行指令
```

## **ADD DEST, SRC**

例如：

```
add eax, 1234
```

这条指令的实际动作是：

```
DEST ← DEST + SRC            //把 DEST 加上 SRC 的结果放到 DEST
```

## **SUB DEST, SRC**

例如：

```
sub esp, 4
```

这条指令的实际动作是：

```
DEST ← DEST - SRC           //把 DEST 减去 SRC 的结果放到 DEST
```

## **AND DEST, SRC**

例如：

```
and esp, -16
```

这条指令的实际动作是：

```
DEST ← DEST and SRC         //把 DEST 和 SRC 的“与”运算结果放到 DEST
```

而本书用作示例的编译器<sup>[4]</sup>所产生的汇编代码是基于AT&T风格，和Win32 平台常见编译器<sup>[5]</sup>生成的Intel风格的汇编代码有点不同，但其实差别很小，我们只需要注意4点即可：

**#1 汇编指令中带有后缀，指示操作数（寄存器或内存）究竟是 8 位、16 位还是 32 位<sup>[6]</sup>。**

例如：

```
movb  $90, -24(%ebp)           //操作数是 8 位的
movw   %ax, %bx                 //操作数是 16 位的
movl   %ebx, %eax               //操作数是 32 位的
```

**#2 源操作数和目标操作数的位置安排与 Intel 风格刚好相反。**

Intel 风格的指令是目标操作数放在“前面”，源操作数放在“后面”：

```
mov eax, ebx                  //把 ebx 的内容复制到 eax
```

而 AT&T 风格的指令是源操作数放在“前面”，目标操作数放在“后面”：

```
movl %ebx, %eax              //把 ebx 的内容复制到 eax
```

### #3 寄存器名前加上“%”，常数、符号的地址以“\$”开头。

例如：

```
movl  $1234, %eax           //把常数 1234 放到 eax
movl  $0x64, %eax           //把常数 0x64 放到 eax
movl  $var, %ebx [7]        //把符号var的地址放到ebx
```

### #4 间接寻址用圆括号表示，偏移量写在括号外面。

例如：

```
mov  eax, [ebx+4]           // Intel 风格
movl  4(%ebx), %eax         // AT&T 风格
```

至于我们用作实验的软件平台GNU/Linux系统，读者可以从网上下载，或购买光盘，而本书所用的GCC3.2 编译器系列也可以在很多站点下载得到<sup>[8]</sup>。由于我们仅仅使用C/C++编译器，所以只需下载其中的两个文件即可：

```
gcc-core-3.2.tar.gz
gcc-g++-3.2.tar.gz
```

把这两个文件复制到相同的目录，例如“/tmp”下，然后解压：

```
#cd /tmp
#tar zxvf gcc-core-3.2.tar.gz
#tar zxvf gcc-g++-3.2.tar.gz
```

进入源代码的目录，然后进行配置、编译：

```
#cd gcc-3.2
#./configure --prefix=/opt/gcc32 [9]
#make
#make install
#cd /tmp
#rm -rf gcc*
```

如果编译成功，则在目录“/opt/gcc32/bin”下有二进制可执行文件gcc和g++，这时就可以使用 3.2 版本的GNU C/C++编译器<sup>[10]</sup>，具体安装细节请参阅相关的文档。

另外，本书一般在专业术语的第一次出现时给出它的英文原词，以提供给读者参考对照。

---

[FN0001]：如果你对“mov ebp, esp”之类的语句不知其意的话，最好先参考一下相关书籍再来阅读本书。

[FN0002]：这里对这些指令的描述是非常粗略的，仅仅是为了帮助读者理解其中的大概，如果需要最详细、最权威的阐述，请参阅[Intel, 2001]。

[FN0003]：SRC 是 source 的缩写，代表源操作数；DEST 是 destination 的缩写，代表目标操作数。除了个别地方，这些操作数一般都是 32 位（4 个字节）的。

[FN0004]：本书使用 GCC3.2 的 i386 版本，它使用 AT&T 风格的汇编代码。AT&T 风格在 UNIX 世界的汇编语言中被广泛使用，请参阅[Konstantin, 2001]。

[FN0005]：例如 Win32 平台上的 Visual C++。

[FN0006]：如果对操作数的确定不会产生歧义，则后缀其实也可以省略掉，例如：

```
movl %eax, %ebx
```

可以写成：

```
mov %eax, %ebx
```

不过，GCC 为了统一起见，所有汇编指令都加上相应的后缀。

[FN0007]：作为对比，请读者注意：

```
movl $var, %ebx      //把 var 的地址放到 ebx
movl var, %ebx        //把 var 的值放到 ebx
```

[FN0008]：请参阅附录 B。

[FN0009]：假设要把 GCC 3.2 安装在目录“/opt/gcc32”，以此类推。

[FN0010]：由于系统仍然按照旧的 PATH 环境变量搜索可执行文件，所以，必须先把旧的编译器文件改名，然后建立符号连接：

```
#cd /usr/bin
#mv gcc gcc.old
#mv g++ g++.old
#ln -s /opt/gcc32/bin/gcc gcc
#ln -s /opt/gcc32/bin/g++ g++
```

## 01 C/C++语言的发展简史

C 语言最初是由 AT&T 贝尔实验室 (Bell Labs) 的 Dennis Ritchie 在 BCPL 和 B 语言的基础上设计, 并在一台 DEC PDP-11 上首次实现的。简洁、高效和可移植性强使其迅速展现出作为系统级编程语言的强大生命力, 随后不久, UNIX 的内核 (kernel) 及应用程序全部用 C 语言改写, 从此, C 成为语言 UNIX 环境下使用得最广泛的主流编程语言。

1978 年, Dennis Ritchie 和 Brian Kernighan 合作推出 The C Programming Language 的第一版<sup>[1]</sup>, 书末的“参考指南 (Reference Manual)”一节给出当时 C 语言的完整定义, 成为那个时候 C 语言事实上的标准, 人们称之为“K&R C”。

随着 C 语言在多个领域的推广、应用, 一些新的特性不断被各种编译器实现并添加进来, 于是, 建立一个新的“无歧义、与具体平台无关的 C 语言定义”成为越来越重要的事情。1983 年 ASC X3<sup>[2]</sup> 成立了一个专门的技术委员会 J11<sup>[3]</sup>, 负责起草关于 C 语言的标准草案。1989 年草案被 ANSI 正式通过成为美国国家标准<sup>[4]</sup>。为了区别于 K&R C, 人们称之为“C89”。

这时, The C Programming Language (2<sup>nd</sup> Ed)<sup>[5]</sup> 也开始出版发行, 书中的内容根据当时最新的 ANSI C 进行了更新。随后, ISO 在 1990 年批准 ANSI C 成为国际标准<sup>[6]</sup>, 于是“ISO C”(或称为“C90”)诞生了<sup>[7]</sup>。除了标准文档在印刷编排上的某些细节不同外, ANSI C (C89) 和 ISO C (C90) 在技术上是完全一样的。

然后, ISO 分别在 1994、1996 年出版了 C90 的技术勘误文档, 更正一些正式标准中的印刷错误, 并且还在 1995 年通过了一份 C90 的技术补充, 对 C90 作了极微小的扩充, 经过这次扩充后的 ISO C 称为 C95。

最近, ANSI 和 ISO 又于 1999 年通过并印刷了新版本的 C 语言标准 (C99)<sup>[8]</sup> 和对应的技术勘误文档。自然, 这是目前关于 C 语言最新、最权威的定义。

现在, 各种 C 编译器均能提供 C89 (C90) 的完整支持, 对 C99 则提供部分支持, 相信很快 C99 便会成为所有 C 编译器的标准。同时, 为了兼容一些“很古老”的代码,

某些 C 编译器还在某种程度上支持 K&R C，但编写 ANSI C 风格的代码才是规范的做法。

本书中的所有 C 代码都是用 GNU C 编译器<sup>[9]</sup> gcc 编译的，gcc 完全支持 C89，并且还有一些对于特定平台实现上的扩充（称为“GNU C89”）。由于 gcc 在默认情况下采用 GNU C89 作为编译的标准，所以为了保证代码严格符合 C89（C90）标准，可以使用特定的编译选项<sup>[10]</sup>，例如：

```
$gcc -std=c89 -pedantic example.c           或者：
```

```
$gcc -std=c90 -pedantic example.c
```

同时，gcc 支持很多 C99 的特性，我们可以指示其按 C99 标准进行编译：

```
$gcc -std=c99 -pedantic example.c
```

本书的 C 代码究竟依据什么标准进行编译要看具体情况，一般地，如果没有特别的命令行参数，则表示用 C89（C90）标准进行编译，并且暗示对于本例子而言，C89（C90）和 GNU C89 是完全相同的。如果例子中的编译命令带有“-std=cXX -pedantic”，则读者可以根据 XX 判断出编译的标准。

与 C 相比，C++ 的发展更为曲折漫长。C 语言从 K&R C 到 C89 虽然经历了 10 年，但语言特性并没有太多的增改。C++ 就不一样了，从 1979 年 Bjarne Stroustrup 提出“带类的 C（C With Classes）”概念<sup>[11]</sup>开始，到 1998 年 ANSI 和 ISO 正式通过长达 700 多页的 C++ 语言标准<sup>[12]</sup>，前后整整 20 年。在此期间，C++ 的语言特性不断增加，到目前为止，C++ 已经成为功能最复杂、特性最丰富的系统级编程语言。

让我们简要地回顾一下这段不寻常的历史。

1979 年，B. Stroustrup 开始研究如何增强 C 语言的数据抽象能力使其最终能够实现面向对象的编程特性。他借鉴 Simula 语言的“Class”概念，把“类”的思想融入到 C 语言中，从而产生一种具备更强的数据抽象能力的新型编程语言<sup>[13]</sup>。为什么选择 C 呢？原因是 C 语言具有高效、扩展性和移植性强的特点。B. Stroustrup 希望设计出来的新语言能够成为效率非常高的主流系统编程语言，当时只有为数不多的编译型语言可供选择，比如 Fortran、Pascal 又或者 C。Fortran 是面向数值计算的，科学用途居多，显然无法担负通用型系统编程的重任；Pascal 最初是基于教学用途设计出来的，



目的是通过简单的语言特性帮助使用者进入面向过程编程的大门，但也由于此，Pascal 缺乏很多支持大型系统开发（例如编写操作系统）的高级特性<sup>[14]</sup>；最后，符合要求的大概只有C语言了。

B.Stroustrup当时实现新语言的做法是写一个转换程序，名字叫“Cfront”。C++代码首先通过Cfront翻译成普通的C代码，然后再用C编译器编译。采取这种做法是为了尽快地推广C++，使它在各种各样的平台上能立即投入使用，因为那个时候C语言已经得到非常广泛的认同并在很多平台上被使用。当时如果要把Cfront设计为直接产生汇编代码，那就要针对每一种特定的平台设计特定的汇编生成器，这样做的成本显然太高，会阻碍C++的传播。即使在今天，还有一部分C++编译器同Cfront的工作原理是一样的<sup>[15]</sup>。

1986年，B.Stroustrup出版了The C++ Programming Language的第一版<sup>[16]</sup>。这时的C++已经开始受到关注，使用C++的人数达到2000左右。1987年，第一次USENIX C++会议开幕，从此，很多重要的语言特性都是在随后举行的会议上被各方面的代表（例如学术界和商业界等）提出、然后修改并由大会最后决定是否增添到C++里面的。在这个过程中，B.Stroustrup一直起着最重要的作用。所以，尽管C++并不是完全由B.Stroustrup一个人单独设计完成的，但他仍然是人人尊敬的“C++之父（Creator of C++）”。

随着越来越多的商业C++编译器出现在市场上，正如你猜到的那样——ANSI开始行动了。1989年，负责C++标准化的ANSI X3J16挂牌成立。次年，B.Stroustrup出版了他另一部经典著作——The Annotated C++ Reference Manual<sup>[17]</sup>（按照惯例，经典著作一定有简称，该著作简称ARM）。由于当时还没有正式的C++标准，所以ARM就成了事实上的标准。

值得一提的是，同样在1990年，C++增加了两个非常重要的新特性：Template（模板）和 Exception（异常）。前者使C++具备了“泛型编程（Generic Programming）”的基础，后者使C++有了更好的运行期错误处理方式。

接着，在1991年，ISO/IEC JTC1/SC22/WG21<sup>[18]</sup>召开第一次会议，开始进行C++国际化的工作。从此，ANSI和ISO的标准化工作保持同步，两者互相协调。很