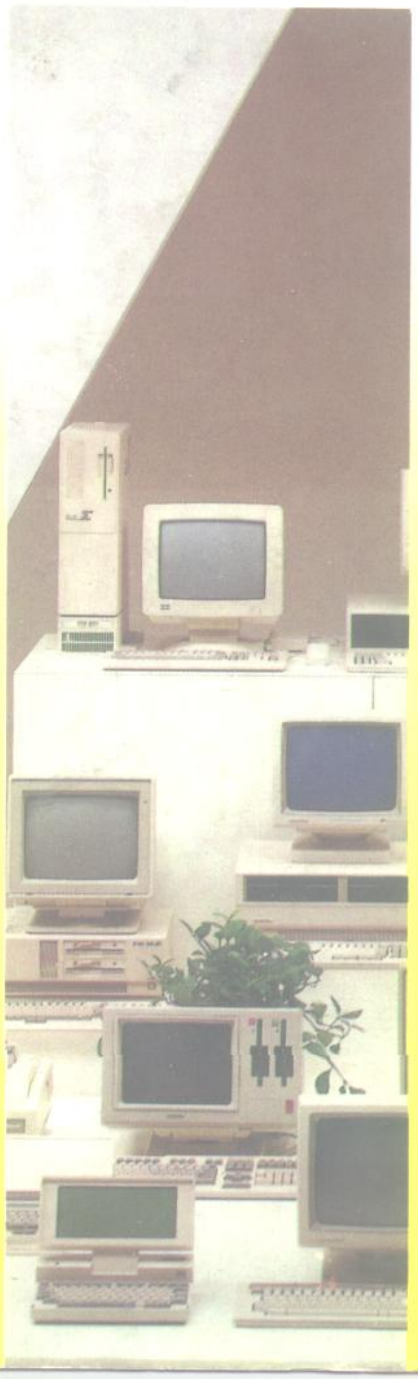
王胜昔　符仲恩　编著

C++

轻松入门



中国科学技术大学出版社

C++轻松入门

C++轻松入门

王胜昔　符仲恩　编著

中国科学技术大学出版社
1997·合肥

内容简介

本书以C语言为基础，用浅显的语言，通过比喻、对比和大量的实例，全面、系统地介绍了C++语言的函数与重载、类、包含和继承、虚拟函数、I/O流和模板等基本概念，以及运用这些基本概念进行程序设计的理论、方法和技巧。最后，作为C++语言的一个综合性练习和具有实用价值的应用，本书还详细地介绍了如何运用C++的类进行数据库管理程序的设计。

本书适合具有一定的C语言背景知识的程序员和大中学校的师生作为从C语言快速转到C++的自学教材，也可供C++程序员和有关计算机软件技术人员作为程序设计的参考资料。

图书在版编目(CIP)数据

C++轻松入门/王胜昔　符仲恩　编著. —合肥：中国科学技术大学出版社，1997年9月
ISBN 7-312-00880-1

Ⅰ　C++轻松入门
Ⅱ　王胜昔　符仲恩
Ⅲ　①计算机　②程序设计　③C语言
Ⅳ　TP312

中国科学技术大学出版社出版发行
(安徽省合肥市金寨路96号，230026)
中国科学技术大学印刷厂印刷
全国新华书店经销

开本：787×1092/16　印张：21.75　字数：557千
1997年9月第一版　1997年9月第一次印刷
数：1—8000册
N 7-312-00880-1/TP·171　定价：21.80元

前　言

在众多的程序设计语言中，C 语言以其极高的代码效率、丰富的数据类型和强大的操作能力，一度成为最受人喜爱、流行最广的语言。90 年代以来，微型计算机的性能日新月异，进一步推动了软件的发展。软件的规模越来越大，程序设计也变得越来越复杂，越来越难以控制，甚至编程的效率也受到了挑战。面向对象的程序设计(Object-Oriented Programming，简称 OOP)开始受到人们的普遍重视。它不仅提高了编程的效率，增强了数据的安全性，提高了程序设计的实用性、可维护性，也缩短了软件开发的周期。

在各种面向对象的程序设计语言中，C++最受人青睐。一方面，C++是 C 的超集，是 C 语言的延伸，而 C 已拥有了最广泛的用户；另一方面，C++除了具备面向对象的特点外，也是生成代码效率最高的语言之一，它的代码效率和 C 语言相当，这使得 C++很快成为 OOP 的主流语言。如今，C 语言程序员纷纷转向 C++，掌握和运用 C++已成为大势所趋。

然而，从 C 到 C++的升级并不像我们想象的那样简单，当我们第一次看到类、重载、继承和虚拟等概念时感到不易理解和接受。这种障碍可能来自两方面的原因，其一是学习的教材，其二是学习的方法。笔者在学习 C++之初，曾看过编译器用户手册式的读物，由于它缺乏对语言本身的详尽讨论，也缺乏让人易于理解、易于接受的例程，每每畏难而退；也买过近百万言的大部头教材，但让人难分主次，总有点胡子眉毛一把抓的感觉，往往为了说明一个简单问题，却列举了一个数十行、甚至数页纸的“例程”，以致于把大部分学习精力都耗费在这些程序的研读上，亦收效甚微。

当然，任何东西的学习都离不开学习的兴趣以及对思维方式和关键问题的把握，程序设计语言的学习更是如此。从 C 语言通向 C++的关键是对“对象”概念的理解和把握，而对象的封装性、继承性和多态性正是面向对象技术的精髓。所有这些概念都是 C 语言所没有的，是阻挡我们通往 C++的“拦路虎”。然而，当越过了这一道屏障之后，我们发现，问题远不是当初想象的那样让人为难。语言本身并不复杂，困难来自我们没有找到一本从语言角度去深入浅出地介绍这些概念的好书，没有把握住领会这些关键概念的方式方法。

能否以一种简单易懂、循序渐进的方式和较少的篇幅来介绍整个 C++语言的核心体系呢？这正是我们写作本书的初衷。我们力求把书写得简洁，把我们的认识、经验和体会写出来，让读者以较快的速度进入 C++的世界。所以，在叙述 C++语言体系时，对于关键概念，我们总是通过相关概念进行逐步引入；对于每一个引申的概念，在含义和用法上尽量和 C 语言做比较；对于每一个新的概念，尽可能结合例程讲解；每个程序都力求简单，一个程序只帮助读者解决一个问题或理解一个概念；在内容编排上力求循序渐进而又简明扼要，紧扣编程实践介绍语言体系。

这种叙述方式使得对C++语言的介绍简单明了，然而过于简单的例程有时对提高初学者编程水平的帮助是有限的。为此，作为C++的应用实例，我们给出了自己制作的一些结构较复杂的类的源程序，并将这些程序化整为零，以避免赘述为原则，尽可能详实地讨论每一个模块的编程思路与使用方法。

作为一个具有一定实用意义的应用，同时也作为C++的一个综合性实例，我们在本书的最后给出了C++数据库类制作技术的较为详尽的分析和讨论。FoxPro和XBase系列是目前国内广泛使用的数据库，它具有对数据库很强的数据表达能力和数据操作能力，是制作各种数据库管理软件的良好工具，为很多程序员所喜爱。笔者也曾使用Clipper和FoxPro编制了大量的应用程序。但是，如果我们采用C或C++，实现对数据库的直接操作，便会充分发挥C或C++的优越性，获得许多意想不到的好处，如可产生较小的代码空间，获得较高的运行效率和良好的软件兼容性，以及任凭程序员自由发挥的余地，等等。所以，使用C++操作XBase的数据库，将具有很大的实用价值。在这本书中，我们用较大的篇幅介绍了如何使用Borland C++制作DBF数据库类。这个类的使用可以使得对数据库的操作变得简单而又方便，即使是对数据库一无所知的读者，通过阅读这部分内容，也能轻松地掌握数据库技术中的一些基本问题和操作方法。

目前，微机上流行的C++版本很多，但占主导地位的还是Borland公司的Borland C++以及Microsoft公司的Microsoft C 7.0和Visual C++，而Borland C++以其优秀的集成开发环境赢得了广大的用户。本书的很多内容，在涉及到编译器的讨论及出错信息方面都是围绕着Borland C++进行的，而且所有例程均在Borland C++ V3.1编译器下调试并通过运行。此外，书中涉及的计算机软硬件资源都是针对x86系列、MS-DOS和MS-Windows系统而言的。我们建议读者在初学时，也不妨以Borland C++ V3.1为基础。

阅读本书需要C语言的初步知识，推荐C语言基础不好的初学者以《C语言大全》作为C语言的参考书。我们自己也曾从这本书的前二百余页中获益匪浅。

为了方便读者使用本书中的程序，我们把所有的源程序整理在软盘中，交由出版社供读者拷贝。我们提供这些程序也出于抛砖引玉的目的，希望能与广大的C++语言和数据库爱好者们相互交流，共同提高。

希望这本书能成为前面提到的那种好书，这曾是我们为之忘我努力的目标。至今虽然几易其稿，也只能说我们自己稍觉满意而已。坦白地讲，这种满意之中多少带些“谁的孩子谁不爱”的成分，书的好坏最终只能由读者评判。书中疏漏在所难免，一些不自量力的浅见敬请有关专家、前辈与读者们谅解和指正。

最后，真诚感谢中国科学技术大学计算机系黄刘生老师仔细审阅了本书，并提出了许多中肯的意见和宝贵的建议；感谢孙萍女士在本书的写作过程中给予的大力支持和鼓励。

符仲恩　王胜昔

1996年12月于合肥

目　　次

第 1 章

愉快的开端

读书宛如交友，第一印象非常重要。第一印象往往取决于是否有一个愉快的开端，本书谨记这一原则。相信本书能使您愉快地踏上 C++之路。

在 C 语言中，我们已习惯于使用.C 作为源程序的扩展名。C++对 C 的第一个改进就是使用.CPP 作为源程序的扩展名，它来自 C Plus Plus 的缩写，用以区分 C 和 C++程序。如果我们使用的语言环境是 Borland C++的 IDE(Integrated Developing Environment)，则在给源程序命名时将自动使用.CPP 作为 C++语言程序的缺省扩展名，而以.C 作为 C 语言程序的缺省扩展名，这给我们的学习带来了很多方便。让我们的学习就从.CPP 开始吧。

1.1 C++的注释

我们在 C 语言中已经掌握了使用"/* */"括起来的注释方法。注释可以是一个段落，也可以是一行中间的一部分。C++继承了 C 的这种注释方法，同时 C++还增加了一种新的注释方法，这就是行注释(相应地，下文中将前一种注释称为段注释)。C++把任何一行中从"//"开始直到该行结束的所有内容皆视为注释。请看下例：

```
/*-------------------------------------------------------------------------------------------*/
    /*
        EXMP0101.CPP
        Showing comment methode in C++.
     */

    #include <stdio.h>
    #include <conio.h>

    void main( )
    {
      //Program checking if a keycode is ESC
      int i, key;
```

```
    while(1) {
      key = getch( );                    //Get a key from console
      if(key == '\x1B') {                //keycode = ESC ?
        printf("\nEscape! ");            //YES, give message
        return;
        }
      else                               //NO, report the keycode
        printf("\nKeycode is %2XH", key);
      }
  }
/*-------------------------------------------------------------------------------------------*/
```

这是一个键码报告程序，在 Borland C++编辑器中编辑这个程序，就会立刻体会到这种新注释的方便之处。3.1 以上版本的 Borland C++编辑器都有语法检查功能，对关键字、注释和错误语句等都以不同颜色显示，注释的部分为暗灰色(在 Windows 环境中用暗蓝色)。在上面的程序中，前 4 行和各行中从"//"开始的所有部分都是暗灰色。

行注释确实提供了某些方便。很多情况下我们可能确实只需要一行注释，比如在上面的程序中多处出现的对程序流程的简单提示，这种时候段注释显得笨拙，因为段注释的最大不便就是始终要记得在注释结束时加上"*/"。通常在程序的调试阶段，我们经常会遇到这种情况，弄不清是否问题真的出在某一句上。这时我们往往希望暂时注掉这一句试试，当然，这种情况下也是行注释来得方便。

另一些情况下我们会倾向于段注释。比如确实需要几行注释才能把问题说清楚，或者在程序调试中需要暂时注掉一个函数，这时你肯定不乐意去为每一行加上行注释。我们乐于使用段注释的另一种情况是为了使注释醒目，比如很多程序员希望在程序头部加上类似的说明(应该说这是一个不错的习惯)：

```
/*************************************
 *  EXMP0102.CPP                     *
 *  Showing comment methode in C++.  *
 *                                   *
 *  Author        某某               *
 *  Date          02/03/1995         *
 *  Last Update   02/03/1995         *
 *  Version       1st                *
 *  Kind          Program            *
 *  Import para   None               *
 *  Out para      None               *
 *************************************/
```

✓Borland C++还支持嵌套注释，只要在命令行使用开关“-C”或在集成开发环境(IDE)中的对话框 Option/Compile/Source 中选通“[ ] Nest Comment”选项。在选通的情况下，允许嵌套注释。对于段注释的嵌套，虽然编辑器有时不能正确地区分(它可能不能对整个注释部分使用正确的颜色)，但编译器却可以正确编译。这意味着，在这种情况下可以任意嵌套地使用“/* */”和“//”注释，这将使得对源程序的注释更加方便和随意。

1.2　变量的定义与声明

我们把下面的语句称为对变量的定义：

```
int i, k;
char ch;
int j = 10;
char *str = "This is a string";
```

即命名一个新的变量，包括指定变量的类型和变量名，甚至为变量赋初值。C 语言的变量在使用之前必须加以定义。编译器遇到变量定义时，为变量分配相应的内存空间，并将所分配内存的起始地址与变量名联系起来。当在程序中用变量名访问变量时，编译器靠与该变量名相关联的地址去操作存储在内存中的变量值。如果不进行变量定义而直接使用变量，编译器就会发现它无法访问这个根本不存在的变量，因而会给出“变量未定义”(undefined symbol)的错误提示。

相对于变量的定义，我们把下面的语句称为对变量的声明：

```
extern int n;
```

因为关键字 extern 仅仅是告诉编译器，它后面的变量已在其它文件内定义，或者说，编译器已经为它分配了内存单元，只不过是允许在这里使用它，所以它仅仅是一个声明(delaration)而已。定义变量时允许给变量赋初值，而声明则不行：

```
double x = 5.5;          //正确，分配变量单元时赋初值 5.5
extern int n = 10;       //错误
```

顺便一提的是，有些文献将变量的“定义”与“声明”统称为“声明”或“定义”，这虽然对于编程并无大碍，但我们坚持认为，分清“定义”和“声明”将有助于对语言的理解。

关于变量定义或声明的位置，C++和 C 稍有不同。C 要求在一个复合语句块(以“{”开始、以“}”结束的一组语句)的开始处定义该语句块中的全部局部变量，这就是说，所有的局部变量定义都必须集中在起始的“{”符号之后。C++没有这种束缚，它允许将变量定义在第一次使用

前的任何一个地方。这意味着，变量的定义可以推迟到即将使用它之时，显得自由自在。下面的程序，使用C++编译器编译不会有问题，而使用C编译器编译时将会给出“syntax error”和“undefined symbol j”等出错信息。

```
/*-------------------------------------------------------------------------------*/
    //EXMP0103.CPP
    //Program to test if it's true that variable can declare anywhere as
    //   long as before it's first used in C++. Here int j is declared
    //   just before it's used in the internal loop. Right!

    #include <stdio.h>

    void main( )
    {
      int i;
      for(i = 0; i < 10; i++)
        for(int j = 0; j < 10; j++)
          printf("OK! i = %d, j = %d\n", i, j);
    }
/*-------------------------------------------------------------------------------*/
```

上面这个程序中，变量i,j是两个循环变量。C++的上述规则允许在for语句中定义所需要的循环变量。上例如想在C环境中编译，必须改成如下形式：

```
/*-------------------------------------------------------------------------------*/
    //EXMP0104.C
    //Corrected version of EXMP0103.CPP. No error.

    #include <stdio.h>

    void main( )
    {
      int i;
      for(i = 0; i < 10; i++) {
        int j;
        for(j = 0; j < 10; j++);
          printf("OK! i = %d, j = %d\n", i, j);
        }
    }
```

/* ---------- */

1.3　C++的强制类型转换

C 语言有 5 种基本数据类型：字符型(char)、整型(int)、浮点型(float)、双精度型(double)和无值型(void)。C 语言对数据类型的要求是严格的，但并不刻板。说它严格，是因为编译器对所用的数据类型作严格检查；而说它不刻板，是因为它几乎允许所有的数据类型之间进行相互转换。许多情况下，数据类型的转换自动进行，并不需要程序员给予太多的关心。比如对于下面的情况：

```
int i;
float f = 10.10;
i = f;
```

这时 i 的值肯定是 10，而不会是 10.10。这里，C 语言自动根据 i 的类型在为它赋值时将浮点数转换为整数。但在另一些情况下，这种自动转换会带来一些问题。比如：

```
/* ---------- */
//EXMP0105.CPP

#include <stdio.h>

void main( )
{
  int i = 10000;
  long l = i * 7;
  printf("l = %ld", l);
}
/* ---------- */
```

乍看起来，程序的输出似乎是 70000，事实上这里的输出却是 4464。诡异！确实有点。但仔细想想，得出这种结果也不难理解：程序在计算变量 l 时将两个乘数都作为整型数处理，这样它首先得到一个整型结果，然后再将它转换为长整型 l。因为实际乘积(70000)已超出 16 位整型值的表达能力，产生了最高位的溢出，使得转换之后面目全非。

解决的办法是运算之前进行类型转换，也就是要求程序按长整型计算 i * 7。如果写成下面的样子，就可以得到想要的结果：

```
(long) i * 7;
```

这也就是将整型的i转换为长整型之后参与运算。但要注意，这里的转换只是运算之中的转换，并不改变i所代表的存储单元中的实际值。因为这样的转换是程序要求的，为区别于编译器的自动转换，通常称其为“强制”类型转换(typecast)。

在传统的C语言中，强制类型转换的语法是：在需要转换的变量前加上包含在括号中的类型说明符。我们再看一个例子：

```
/*-------------------------------------------------------------------------------------------*/
    //EXMP0106.CPP

    #include <stdio.h>

    void main( )
    {
      int i = 1;
      char ch = 0x80;
      printf("Before typecast i * ch is: %d\n", i * ch);
      printf("After typecast i * ch is: %d\n",i * (unsigned) (unsigned char) ch);
    }
/*-------------------------------------------------------------------------------------------*/
```

程序的输出是：

```
Before typecast i * ch is: -128
After typecast i * ch is: 128
```

要特别留心上面的用法：先做一次(unsigned char)，再做一次(unsigned)。这是因为字符在做强制类型转换时，实际上隐含了一个强迫把一个字节单元扩展成字单元的过程。对于有符号的字符，字符型向整型转换的过程还包含了字符型的符号位向字的符号位扩展的过程。在这里，第一次计算i*ch时，编译器自动将ch转换为一个整数，得-128，所得结果在情理当中；第二次计算时，编译器首先将ch强制转换成一个unsigned char，得128，进一步的字扩展和unsigned类型强制转换的结果仍是128。

以下一例是强制类型转换中常犯的错误，也是最典型的错误：

```
/*-------------------------------------------------------------------------------------------*/
    //EXMP0107.CPP

    #include <stdio.h>
```

```
void main( )
{
  char ch = 0x80;
  int i = (unsigned) ch;
  printf("\ni = %d", i);
  long l = (unsigned long) (40000);
  printf("\nl = %ld", l);
}
/*-------------------------------------------------------------------------------------------*/
```

我们希望它的结果是 i = 128, l = 40000, 但实际输出的结果却是:

```
i = -128
l = -25536
```

这样的错误往往发生在多种数据类型做混合运算的场合,而且所发生的错误一般也比较隐蔽,只有小心使用才能得到所期望的结果。

除表达式运算需要强制类型转换的情况以外,有时我们可能更频繁使用的是指针的强制类型转换。因为很多 C 语言库函数中的参数或其返回值都被定义成无值型指针,而实际使用时往往需要将其转换成所需要的类型,这在 Borland C++ for Windows 编程中几乎是随处可见。在 DOS 程序中,最典型的例子是动态内存分配函数 malloc( )。

malloc( )函数的返回值总是一个无值型指针,当我们需要申请一块内存存放其它类型的数组时,我们不得不进行类型转换。对于 C++ 的情况更是如此。比如,我们需要一个整数组 integer[ ]的情况:

```
int *integer = (int *) malloc(elements_number * sizeof(int));
```

这里的 elements_number 是所定义的数组中将存放的元素个数。

C++ 允许传统 C 语言的强制类型转换格式,但它同时提供了另一种作用力更强、形式也更优美的方式,就是将强制类型转换写成函数调用的形式。比如:

```
i = int(f);
i = unsigned(ch);
```

如果仅做单变量的变换,似乎还体现不出这种格式的优越性,但当它作用于整个表达式时,不使用这种新的方式,恐怕很难找到更为直接的办法:

```
float y = 12351.55;
```

```
int r = int(y / 93.3 + 0.5) % 100;
```

这种形式，虽然格式上更像函数调用，但实际上编译器仍然将类型符当成关键字进行处理。

1.4 作用域与作用域分辨符

作用域(scope)在C语言中是一个非常重要的概念。更确切地讲，它应该叫做“有效域”，即一个变量在程序中的有效范围。讲到域，很可能读者会想到“定义域”、“值域”之类。在这里，作用域决不是变量的取值范围，而是在程序的什么部分该变量是可以访问的，但在其它部分则不能访问、不可见，甚至根本不存在。

作用域的概念不仅对C重要，对将来理解C++的面向对象技术也非常关键。理解作用域的概念，首先需要对C语言的所谓“模块”有一个清楚的认识。模块就是我们在前面“变量的定义与声明”一节中提到的“语句块”。模块也叫复合语句，也就是在花括号“{ }”之间的一组语句。C语言的程序无非就是各种模块的堆砌和嵌套。

变量的作用域和它的定义位置有密切关系。如果一个变量定义在某个模块内，它的作用域就仅限于这个模块内。也就是从这个变量定义位置开始到相应模块的结束花括号“}”之前，这个变量才是可见的。这种变量称为局部变量。

相反，如果一个变量被定义在所有的模块之外，那么它对于程序中的所有模块都是可见的，因而被称为全局变量。概括起来，C++的变量的作用域有以下4种：

(1)局部模块作用域：变量的定义在局部块内，只在当前块内有效。一个函数定义体也可视为一个模块；

(2)全局作用域：变量定义在全局范围内，对全程序有效；

(3)文件作用域：变量定义在全局范围内，但只限当前文件存取；

(4)类作用域：变量属于类，在类范围内有效。

下面的例子中，对全局变量和各层模块中的局部变量的作用域给出了简单的演示：

```
/*-------------------------------------------------------------------------------*/
    //EXMP0108.CPP
    //About scope.

    #include <stdio.h>

    int global = 10;
    void main( )
    {
      {
```