重点大学计算机教材

C++高级进阶教程

主编 陈刚

WUHAN UNIVERSITY PRESS
武汉大学出版社

重点大学计算机教材

C++高级进阶教程

主 编 陈 刚

WUHAN UNIVERSITY PRESS
武汉大学出版社

图书在版编目(CIP)数据

C++高级进阶教程/陈刚主编. —武汉:武汉大学出版社,2008.10
重点大学计算机教材
ISBN 978-7-307-06563-5

Ⅰ.C… Ⅱ.陈… Ⅲ.C语言—程序设计—高等学校—教材 Ⅳ.TP312

中国版本图书馆CIP数据核字(2008)第158016号

责任编辑:黄金文 韩光朋 责任校对:黄添生 版式设计:支 笛

出版发行:**武汉大学出版社** (430072 武昌 珞珈山)
(电子邮件:wdp4@whu.edu.cn 网址:www.wdp.whu.edu.cn)
印刷:湖北金海印务公司
开本:787×1092 1/16 印张:27.25 字数:691千字 插页:1
版次:2008年10月第1版 2008年10月第1次印刷
ISBN 978-7-307-06563-5/TP·317 定价:38.00元

重点大学计算机教材

编 委 会（一）

内容提要

本书在假定读者有一定的C++编程能力的基础上，进一步加强了对一些基本概念（如文字常量与常变量、指针与引用、作用域与生命期、分离编译模式、声明与定义、静态联编与动态联编等）的解释，介绍了一些不太常用的关键字（如 volatile、mutable、static_cast、dynamic_cast、const_cast、reinterpret_cast 等）的用法。同时，介绍了一些C++语言机制的底层实现方案，如引用是怎样实现的、对象上的实例成员函数是怎样被调用的、虚函数表是如何存储以及如何被访问的、new 和 delete 的实现过程是怎样的，等等。另外，还进一步加深了对一些常用的C++语言机制的讲解，如 sizeof 的用法、typedef 的用法、命名空间的定义和使用、多维数组与多重指针、各种操作符的重载等，同时也对一些高级话题，如怎样调试程序、为什么需要设计模式、怎样应对C++语言的复杂性等进行了探讨。

希望通过对这些内容的学习，使读者能够在微观和宏观两个方面进一步拓展对C++语言的认识，从而能够更好地利用它进行程序开发。

本书可作为高年级本科生、研究生的程序设计语言教材，也可供相关的工程技术人员参考。

前 言

本书是为学过一遍C++语言的学生和程序员编写的。因此，本书并没有对C++的入门知识作详细的介绍。所谓“学过一遍”的含义是：至少跟着一本教科书从头到尾学习过一遍，对C++的基础知识和基本概念有所了解。对于初学者来说，普遍的情形是：虽然也可以用C++语言进行编程，但是对C++语言的很多地方仍然感到模糊，想有进一步的提高却找不到合适的教材，只能按部就班地在原来的教材上再走一遍。结果是花费了时间，却达不到好的效果，因为学习的针对性太差。

“学过一遍”C++的人，如何才能有进一步的提高呢？本书打算从五个方面入手：

第一，纠正一些模糊不清的概念和观点。由于编程实践的限制，也由于一般的教材讲述重点的限制，某些要点只是匆匆交代了一下就结束了，导致学生留下一些似是而非的观点。例如，main()函数一定是C++程序第一个执行的函数吗？是由对象的构造函数为对象分配内存空间吗？等等。

第二，介绍一些一般的教材上不介绍或没有深入介绍的内容。如关键字mutable的用法，static_cast、dynamic_cast、const_cast、reinterpret_cast的用法，名字空间（namespace）的定义和使用，等等。这些内容往往由于缺乏实际的例子而无法让学生有贴近的观察。而当读到别人的程序中使用了这些机制的时候，会有一种自己不是真正的C++程序员的感觉。

第三，了解一些C++语言的底层实现机制。例如，细心的程序员都会遇到一些难以解释的“怪”现象，如执行cout<<++i<<--i<<i++;语句，程序的执行结果让人感到惊讶不已，却又一时难以解释。再如，引用是变量吗？引用在底层是怎样实现的呢？这就需要了解一些C++语言的底层实现机制。有时，仅仅停留在高级语言的层面很难把一个问题解释清楚。

第四，介绍一些与编程操作相关的内容。没有足够的编程实践是无法学好一门计算机编程语言的。通常的教科书都忽视了编程操作的问题，认为那是与具体开发平台相关的内容，不适合在教材中讲解。实际上，有些问题是在编程实践中带有共性的问题，无论在哪种开发平台下都回避不了。例如，分离编译模式的问题，以及模板在分离编译模式下产生的典型编译错误应如何解决等，都值得仔细讲解，为学生进一步深入学习扫清障碍。

第五，介绍一些有用的C++编程思想。仅仅掌握C++的语法规则是远远不够的，就像一个只懂得下棋规则的棋手，不经过实战的磨练，不会成为一名出色的大师。把一些优秀的编程思想介绍给读者，能使学习者少走弯路，在较短的时间内进行有针对性的训练从而得到快速的提高。

本书的内容就是针对上述五个目标而展开的。为了加强学习效果，绝大部分的内容都是通过具体的程序实例进行讲解，有的还展示了对应的汇编代码以帮助理解。所有的程序实例都在Visual Studio 2005（在书中简称VS 2005）平台下调试通过。每个例子的规模都不大，但都针对一个具体的问题，因而很有说服力。

需要说明的是，C++语言的底层实现机制大多不属于C++标准的一部分，因而不同的编

译器的实现细节可能是不同的。本书展示的汇编语言代码是在VS 2005平台下反汇编的结果，这些代码能够帮助学习者了解一些在高级语言层面之下的实现方案，从而加深对计算机系统（不仅仅是编程语言本身）的理解。

通过本书的学习，能使学生对C++语言的掌握有一个质的飞跃。不仅能够解释程序的各种行为发生的原因，避免一些编程错误的发生，还可以有意识地编写出高效率的代码来。本书适用于对C++语言有一定程度的掌握而想进一步大幅度提高的读者，也适合用做软件及相关专业高年级本科生或研究生教材。

在本书的编写过程中，武汉大学软件学院的胡启平和桂浩老师提出了很多宝贵的建议，武汉大学出版社的黄金文副编审也提供了大力的支持，在此深表感谢！

本书中的例子大多数是编者自己实际教学的案例，部分参考了一些C++教材和专业网站的资料。书中不足甚至是错误的地方，欢迎来信批评指正。联系邮箱是：

chenzuolin@yahoo.cn

如需要配套的资料，也可以直接与武汉大学出版社或编者本人联系。

作　者

2008年6月于珞珈山

目 录

第1章 C++基础知识

1.1 关于C++标准

美国AT&T贝尔实验室的Bjarne Stroustrup博士在20世纪80年代初发明并实现了C++（最初这种语言被称作“C with classes”）。一开始C++是作为C语言的增强版出现的，从给C语言增加类开始，不断地增加新特性。虚函数（virtual function）、运算符重载（operator overloading）、多重继承（multiple inheritance）、模板（template）、异常（exception）、RTTI、名字空间（name space）逐渐被加入标准。1998年国际标准化组织（ISO）颁布了C++程序设计语言的国际标准ISO/IEC 14882-1998。C++是具有国际标准的编程语言，通常称作ANSI/ISO C++。1998年是C++标准委员会成立的第一年，以后每5年视实际需要更新一次标准，下一次标准更新将是在2009年，目前一般称该标准为C++ 0x。遗憾的是，由于C++语言过于复杂，并且经历了长年的演变，直到现在只有少数几个编译器完全符合这个标准。实际上，从严格的角度来说，至今为止没有任何一款编译器完全支持ISO C++。

C++语言发展大概可以分为三个阶段：第一阶段从20世纪80年代到1995年。这一阶段C++语言基本上是传统类型上的面向对象语言，并且凭借着接近C语言的效率，在工业界使用的开发语言中占据了相当大的份额；第二阶段从1995年到2000年，这一阶段由于标准模板库（STL）和后来的Boost等程序库的出现，泛型程序设计在C++中占据了越来越多的比重。同时，由于Java等语言的出现和硬件价格的大规模下降，C++受到了一定的冲击；第三阶段从2000年至今，由于以Loki、MPL等程序库为代表的产生式编程和模板元编程的出现，C++出现了发展历史上又一个新的高峰，这些新技术的出现以及和原有技术的融合，使C++已经成为当今主流程序设计语言中最复杂的一员。

实际上，C语言也有自己的标准。从20世纪70年代初期的早期C语言到后来的K&R C、ANSI C、C89，在将近20年中C语言多次发展演化，一直到1999年C语言又重新定案，成为新的C语言标准，这就是C99标准。

实际上，可以认为C与C++是两门独立的语言。首先，它们有各自的标准和标准委员会；其次，尽管C++支持C语言的几乎全部功能，在语法上与C语言还是有极微妙的差别，编译器对全局变量名和函数名的编译方式也不同；使用C++语言，应该重点使用它的面向对象的特性，而这一点是C所完全做不到的。

对于初学者而言，有C语言的基础对于掌握C++有一定的帮助，但这不是必须的。C++并不依赖于C语言，完全可以直接学习C++。根据《C++编程思想》（Thinking in C++）一书所评述的，C++与C的效率往往相差在正负5%之间。所以，在开发大型程序时C++完全可以取代C语言，仅在非常强调效率、内存空间有限、直接操作硬件等场合使用C语言。

使用过C/C++语言的人都很熟悉下列这些逻辑运算符：&&、||、!、&、|、~、^、&=、|=、

^=、!=。但如果有人在程序中使用 and、or、not、bitand、bitor、compl、xor、and_eq、or_eq、xor_eq、not_eq 来代替上述逻辑运算符，不要认为他们写的不是 C/C++程序。实际上，这是 C99 标准中引入的内容。考察下面的程序。

```
//// Program 1.1-1 ////
#include <iostream>
#include <iso646.h>
using namespace std;
int main(){
    int i=5,j=6,k=7;
    if(i<j and j<k) cout<<"i<j and j<k"<<endl;
    if(i<j or j<k) cout<<"i<j or j<k"<<endl;
}
//// End of Program 1.1-1 ////
```

此程序会顺利输出：

```
i<j and j<k
i<j or j<k
```

其实，像 and、or 等符号并不是真正新加的什么关键字，而是利用宏替换实现的。打开文件 iso646.h，可以看到如下的语句：

```
#define and      &&
#define and_eq   &=
#define bitand   &
#define bitor    |
#define compl    ~
#define not      !
#define not_eq   !=
#define or |     |
#define or_eq    |=
#define xor      ^
#define xor_eq   ^=
```

所以，要想使用这些逻辑运算符的替代符号，要包含文件 iso646.h。假设 Visual Studio 2005 安装在目录 C:\Program Files\Microsoft Visual Studio 8 下，则可以在 C:\Program Files\Microsoft Visual Studio 8\VC\include 目录下找到头文件 iso646.h。有时，直接查看系统头文件可以帮助我们了解一些语言现象的底层实现机制。

1.2 文字常量和常变量

常量可以直观地理解成“值不可改变的量”。在 C++语言中，常量分为两种：文字常量（literal constant）和常变量（constant variable）。概括地说，它们之间的关系是这样的：

①文字常量又称为“符号常量”、“字面常量”，经编译之后写在代码区，是不可寻址的。而常变量同其他变量一样被分配空间，是可以寻址的。

例如，在 Visual C++中，语句 int i=3; 所对应的汇编代码为 mov DWORD PTR _i$[ebp],3，其中_i$表示在一帧数据中，变量 i 距帧指针 ebp 的偏移量。文字常量 3 被直接写在代码区，在数据区无法找到它。利用预编译指令#define 定义的常量也属于文字常量。

整型文字常量前可加 0，表示八进制数，加 0x 表示十六进制数，如 0x14 表示十进制的 20。整型文字常量后可加 L（或 l，但推荐用大写字母，不易和数字混淆）表示 long 类型，加 U（或 u）表示无符号数，如 1024UL。

科学计数法中，指数可写作 e 或 E，如 3e-3 表示 3×10^{-3}。浮点型文字常量缺省为 double 型，其后可加 f 或 F 来表示单精度文字常量；扩展精度由后面跟 l 或 L 来指示。

字符文字前加 L 表示宽字符文字，类型为 wchar_t。在字符串后加“\”表示在下一行继续。字符串前加 L 表示宽字符串文字，以宽空字符结束。两个（宽）字符串在程序中相邻，C++会把它们连接在一起，如“two” “some”，结果为“twosome”。

程序中有些特殊的标识符或表达式，由于同时满足这样两个条件：不可寻址、值不可变，所以可以将它们视为文字常量。它们是：静态数组名、枚举常量、全局（静态变量）首地址。

一个变量，如果限定它在定义的时候就必须同时为它赋初值，且此后它的值就不能再改变，那么这个变量就是一个常变量。常变量由普通变量在前面加 const 关键字来定义。常变量的值在初始化后不能改变，是在高级语言的语义这一层面上定义的，由编译器所做的语法检查进行保障。但由于运行时常变量并不是放在只读内存中，而是和一般变量一样放在数据区，所以在运行时如果能获得常变量的地址，一样可以通过特殊的途径对它们进行修改。如下面的程序。

```
//// Program 1.2-1 ////
#include <iostream>
using namespace std;
void ShowValue(const int& i){
    cout<<i<<endl;
}
int main(){
    const int j=5;
    int *ptr;
    void *p=(void *)&j;
    ptr = (int *)p;
     (*ptr)++;
    ShowValue(j);
}
//// End of Program 1.2-1 ////
```

程序的输出结果是 6，表明常变量 j 的值在运行时的确被修改了。这就证明：常变量是一种加了特殊限制的变量。将常变量理解成“只读”变量会更准确一些。

注意：在上面的程序中，如果将 main()函数中的 ShowValue(j);改成 cout<<j<<endl;，那么输出结果仍然是 5。这并不是说明常变量 j 的值没有改变，而是编译器在代码优化的过程中已经将 j 替换成了文字常量 5 的缘故。

②出于语法规则或代码优化的需要，在某些情况下文字常量和常变量会由编译器进行转

换。例如，考察下面的程序的输出结果。

```
//// Program 1.2-2 ////
#include <iostream>
using namespace std;
void DefineArray(const int n){
    int B[n]={};
    cout<<B[0]<<endl;
}
int main(){
    const int m=5;
    int A[m]={};
    cout<<A[0]<<endl;
}
//// End of Program 1.2-2 ////
```

实际上，在函数 DefineArray()体内，int B[n]={};这条语句出现编译错误。而如果去掉函数 DefineArray()的定义，程序则可以正常运行并输出结果 0。原因是在定义数组 A 的时候，编译器将常变量 m 替换成了文字常量 5；在函数 DefineArray()体内，无法将常变量 n 替换成相应的文字常量，而在 C++中，数组的大小是在编译时决定的，所以出现编译错误。

再看如下的语句：int &r=5;

这条语句会出现编译错误，原因是文字常量是不可寻址的，因而无法为文字常量建立引用。

而下面这条语句又是合法的：const int &r=5;

这里实际上有一个将文字常量转化成常变量的过程。即先在数据区开辟一个值为 5 的无名整型量，然后将引用 r 与这个整型量进行绑定。

1.3 const 的用法

const 是 C++语言引入的一个关键字，是“不变的”、“常量”的意思。用 const 定义一个变量，实际上是定义了一个“常变量”（即只读变量），关于常变量的有关内容请参见 1.2 节。但是，const 的用法远不是这么简单，因为 C++中有指针、引用、函数等多种机制，所以和 const 组合在一起，就会遇到很多实际问题。下面从 4 个方面总结一下 const 的用法。

（1）const 的位置

const 的位置比较灵活。一般说来，除了修饰一个类的成员函数外，const 不会出现在一条语句的最后。以下是一个使用 const 的例子。

```
//// Program 1.3-1 ////
#include <iostream>
using namespace std;
int main(){
    int i=5;
    const int v1=1;
```

```
    int const v2=2;
    const int *p1;
    int const *p2;
    //以下三条语句都会报编译错误，为什么？
    //const * int p3;
    //int * const p3=&v1;
    //int * const p3;
    int * const p3=&i;
    const int * const p4=&v1;
    int const * const p5=&v2;
    const int &r1=v1;
    int const &r2=v2;
    //以下语句会报编译错误，为什么？
    //const   & int r3;
    //以下语句会报警告，并忽略const
    int & const r4=i;
    cout << *p4 << endl;
}
//// End of Program 1.3-1 ////
```

程序的输出结果是：1。当然，读者也可以自行考察一下其他变量的值。以上程序演示的是 const 的位置与它的语义之间的关系。表面上看起来很复杂，但实际上是有规律可循的。也就是说，const 和数据类型结合在一起，形成所谓“常类型”，然后利用常类型来声明或定义变量，这样就产生了常变量。const 用来修饰类型时，既可以放在类型的前面，也可以放在类型的后面；用常类型声明或定义变量时，const 只会出现在变量前面。const 和被修饰类型之间不能有其他标识符存在。

在理解有 const 存在的声明语句时，关键是要搞清楚到底什么是“不可变”的。对一个具体的变量来说，如果 const 直接出现在该变量的前面，则该变量的值一旦被初始化就不能再改变。所以，const int v1;和 int const v1;都是合法的，而且是等价的。由于 const 直接出现在 v1 前，所以 v1 是只读变量（常变量）。而 int const *p 和 int * const p 则是不同的声明语句，原因是前者的 const 修饰的是 int，而后者的 const 修饰的是 int *。前者表示指针 p 指向的是一个只读的整型量（指针所指单元的内容不允许修改），而指针本身可以指向其他的常变量；而后者表示指针 p 本身的值不能修改（const 直接出现在 p 的前面），即一旦指针 p 指向某个整数之后，就不能再指向其他整型量。如果指针被定义为指针常量，那么在定义它的时候必须同时初始化，否则编译器报错。

为了方便叙述，本书中将“常指针”和“指针常量”定义为不同的术语。常指针代表“指向常量的指针”，“指针常量”则代表指针变量本身是只读的。这种区分是为了统一全书的叙述，而在其他很多书籍中二者都是后一种含义。

引用本身可以理解成一个指针常量，所以在引用前使用关键字 const 没有意义。上例中 int & const r4=i;语句的 const 是多余的，编译器在给出警告信息后，自动忽略 const 的存在。常引用是指将被引用对象当做一个常量，也就是不允许通过引用来修改被引用对象的值。本

书中其他地方提到“常引用”，都代表这种含义。常引用的定义有它独特的特点，具体可参阅 1.9 节。要清楚的一点是：普通变量可以当做常变量看待，但反过来不可以。

在很多情况下，为了表达同一种语义，可以将 const 放在不同的位置，如前面的几个例子。但在某些情况下，const 只能放在特定的位置，否则意义就会完全不一样。下面是一个 const 配合二重指针的例子，在这个程序中，const 的位置是不能随意变动的。

```
//// Program 1.3-2 ////
#include <iostream>
using namespace std;
int main(){
    int const **p1;
    int * const *p2;
    int i=5;
    int j=6;
    const int *ptr1=&i;
    int * const ptr2=&j;
    p1=&ptr1;
    p2=&ptr2;
    cout << **p1 << endl;
    cout << **p2 << endl;
}
//// End of Program 1.3-2 ////
```

程序的运行结果是：

```
5
6
```

在程序中定义了两个二重指针 p1 和 p2，它们的声明分别是 int const **p1 和 int * const *p2，但含义完全不同。p1 不是指针常量，它所指向的变量的类型是 int const *（指向整型常量的指针）；p2 也不是指针常量，但它所指向的变量是指针常量（int * const，即指向整型的指针常量）。所以，为 p1 和 p2 赋值是有讲究的。如果在上面的程序中，使用这样的赋值 p1=&ptr2 或 p2=&ptr1，都会产生编译错误。有兴趣的读者可以自行试验一下，并分析产生错误的原因。

（2）const 对象和对象的 const 成员

用 const 来定义一个基本类型的变量，是指不允许显式修改该变量的值。如果用 const 来定义某个类的对象，则情况还要复杂一些，因为对象除了有成员数据之外，还有成员函数。用 const 修饰的对象称为常对象，而用 const 修饰的类的成员函数称为常函数，在常函数中不允许对任何成员变量进行修改。通过常对象，只能调用该对象的常函数。下面是一个关于常对象和常函数的例子。

```
//// Program 1.3-3 ////
#include <iostream>
using namespace std;
class A{
```