普通高等院校“十二五”规划教材

C++面向对象程序设计双语教程

Object-Oriented Programming in C++

刘嘉敏　马广焜　常燕　朱世铁　编著

国防工业出版社
National Defense Industry Press

普通高等院校“十二五”规划教材

C++面向对象程序设计双语教程

Object-Oriented Programming in C++

刘嘉敏　马广焜　常燕　朱世铁　编著

国防工业出版社

·北京·

图书在版编目(CIP)数据

C++面向对象程序设计双语教程/刘嘉敏等编著. —北京:国防工业出版社,2013.2
普通高等院校“十二五”规划教材
ISBN 978-7-118-08603-4

Ⅰ.①C... Ⅱ.①刘... Ⅲ.①C语言-程序设计-双语教学-教材 Ⅳ.①TP312

中国版本图书馆CIP数据核字(2013)第007900号

※

国防工业出版社出版发行
(北京市海淀区紫竹院南路23号 邮政编码100048)
北京奥鑫印刷厂印刷
新华书店经售

*

开本 787×1092 1/16 **印张** 16 **字数** 363千字
2013年2月第1版第1次印刷 **印数** 1—2500册 **定价** 32.00元

(本书如有印装错误,我社负责调换)

国防书店:(010)88540777 发行邮购:(010)88540776
发行传真:(010)88540755 发行业务:(010)88540717

前　言

面向对象程序设计(Object-Oriented Programming,OOP)是一种以对象为基础,以事件或消息来驱动对象执行相应处理的程序设计方法。它将对象作为程序的基本单元,将数据和对数据的操作封装在一起,以提高软件的重用性、灵活性和扩展性。它是一种为现实世界建立对象模型,利用继承、多态和消息机制设计软件的先进技术,非常适合于大型应用程序与系统程序的开发,是当今计算机程序设计倡导的主流技术。

为使读者快速建立面向对象的概念、循序渐进地理解OOP技术的各知识点内容、系统地建立OOP完整的知识架构及利用OOP方法完整地解决一个实际应用问题,本书以OOP知识节点递进式组成各章节内容,并以实际应用问题作为样例贯穿于全书。本书内容是作者根据十余年担任OOP课程教学、OOP课程设计体会及多年指导学生毕业设计、课程设计和计算机程序竞赛所积累的经验而精心组织的,各章节内容结构清晰,便于学习和掌握。

在多种支持面向对象程序设计语言中,正是由于C++语言以其与C语言兼容、高运行效率等优良特性,使大量C程序员通过C++的帮助迅速掌握了面向对象的概念和方法,全面促进了面向对象技术的应用,从而使C++成为最有影响、最广泛应用的面向对象程序设计语言。因此,本书针对掌握C语言程序设计方法的初学者,以C++语言为基础讲述面向对象程序设计概念和实现方法。

以英文撰写,旨在满足高校提倡的外语教学、双语教学需求,提高学生专业英语阅读能力。由于目前国内鲜见由国内学者自编、符合中国学生思维习惯和学习习惯的OOP英文教材,因此作者倾注了多年心血奉献此书。该书作为校内教材在作者学校计算机科学与技术专业本科生教学已使用4年,曾修订过两版,得到了广大同学的赞许。

在内容组织结构上,为学生提供有:

“教学目标(Objectives)”,告诉学生每章重点知识,让学生学完一章后判定是否达到这些目标,帮助学生建立学习信心。

“重点注释(Notes)”,对重点知识和易混淆知识点加以提示和强调,告诉学生必须清楚这些知识点,并且知道如何正确使用。

“实例(Examples)”,对主要知识点给出通俗易懂、适合初学者的实例,实例都配

有完整的代码和运行结果,使学生可以确定程序预期结果,通过输出结果与程序语句联系在一起,为学生提供学习和巩固概念的方法。

“概念加注框”,将各章节涉及基本概念详解归纳集聚在矩形边框内,供学生一目了然地阅读、学习和掌握。

“思考题(Think Over)”,帮助学生回顾和总结重点知识,检查对知识的学习掌握程度。

“词汇小贴士(Word Tips)”,按字母顺序列出各章节中的生词,给出词性和专业的解释,帮助学生快速、准确理解 OOP 专业知识,提高外语阅读能力。

“练习题 (Exercises)”,提供从概念理解、基本语句运用到简单问题求解的各种类型练习题,便于学生对知识的理解和运用。习题的难度适宜,学生基本能独立完成。较复杂问题和团队合作大型综合训练在课程实验和课程设计中安排。

本书共分 8 章,每章的主要内容有:

第 1 章 Introduction,介绍了什么是程序设计、如何面对实际问题设计相应的程序代码的步骤;简要地介绍了从机器语言到高级语言的程序设计发展历史;介绍了两种主流的程序设计方法,着重介绍了面向对象程序设计的概念和基本特点;讨论了C++程序设计语言的历史和C++ 语言的学习方法。

第 2 章 Basic Facilities,讨论了与 C 语言不同的C++ 的基础知识,以结构化程序设计方法,介绍了基本的输入/输出流、常量、函数、引用和命名空间,可以使学生顺利地从 C 语言过渡到C++ 语言的程序设计。

第 3 章 Classes and Objects(I),从这章开始学习面向对象程序设计的基本概念和方法,本章为类与对象设计的基础部分,介绍了数据抽象和信息隐藏的概念,详细讲述了抽象数据类型(ADT)类的基本设计方法、类中主要成员构造函数和析构函数的定义和对象的生成。

第 4 章 Classes and Objects (II),是进阶使用对象和类的部分,讨论了类中的一些特殊成员的定义和实现,如静态成员、常量成员、对象成员、拷贝成员和友元,以及各种类型的对象声明和使用。

第 5 章 Operator Overloading,介绍了如何对用户定义数据类型使用现有运算符进行运算,主要介绍了运算符重载的基础、运算符的限制和重载成员函数和非成员函数、一元和二元运算符以及类型转换。

第 6 章 Inheritance, 继承是软件重用的一种形式,介绍了面向对象程序设计的主要特性之一——继承概念,重点介绍了派生类的成员函数、访问控制、构造函数和析构函数的定义,讨论了多继承的模糊性,以及如何运用虚基类避免模糊性的设计。

第 7 章 Polymorphism and Virtual Functions,多态是面向对象程序设计的另一个主要特性,从同一基类继承多个类和设计相同的接口的方法,使系统更易于扩展。本章

介绍了两种多态的特点、虚函数的工作原理和设计方法，并介绍了抽象基类的实现过程。

第8章 Templates，模板也是软件重用的一种形式，它使程序员可以利用参数形式描述抽象数据类型的本质，然后用少量代码生成特定的抽象数据类型。本章介绍了函数模板和类模板的定义和实现方法。

书中所有源代码在 Microsoft Visual C++ 6.0 环境中调试和运行通过。

该教材适合于40～54学时教学，配有PPT教学课件，并提供实验指导书、实验题目详解代码、OOP课程设计题目，有需要的任课教师请垂询 jmliu@ sut. edu. com。

本书由刘嘉敏统编，主要编写了第1、2、3、4、5章，参编了第7章；马广焜编写了第6、7、8章；常燕参编了第1章和练习题，并且完成了全书的排版和校对；朱世铁参编了练习题和实验指导书。英国贝德福德大学 Des Stephens 老师花费了他的业余时间，非常耐心地修正了书中英语错误，借此对他的支持和帮助致以衷心的感谢。另外，沈阳工业大学的杨丹、邱辉、靳长旭、闫博、郭云龙、赵天育、王爱新和张荣铭等同学协助完成每章的词汇表，在此致谢。此外，书中引用了其他同行的工作成果，在此一并表示衷心感谢。

本书在编写过程中，作者付出了极大的辛苦。由于英语水平有限，书中难免会有错漏和不妥之处，恳请读者提出宝贵意见，在此衷心地表示感谢。

编 者

2012 年 11 月

目 录

Chapter 1
Introduction

High thoughts must have high language.
—Aristophanes

Objectives

- To understand what a computer program is
- To be able to list the basic stages involved in writing a computer program
- To recognize two programming methodologies
- To know about characteristics of object-oriented programming

1.1 Overview of Programming

1.1.1 What Is Programming?

Much of human behavior and thought is characterized by logical sequences. Since infancy, you have been learning how to act, how to do things. And you have learned to expect certain behavior from other people.

On a broad scale, mathematics never could have been developed without logical sequences of steps for solving problems and proving theorems. Mass production never would have worked without operations taking place in a certain order. Our whole civilization is based on the order of things and actions. We create order, both consciously and unconsciously, through a process we call **programming**.

Programming is planning how to solve a problem. No matter what method is used—pencil and paper, slide rule, adding machine, or computer—problem solving requires programming. Of course, how one program depends on the device one uses in problem solving.

Programming is planning the performance of a task or an event.
Computer is a programmable device that can store, retrieve, and process data.
Computer programming is the process of planning a sequence of steps for a computer to follow.
Computer program is a sequence of instructions to the performed by a computer.

Back to programming—"planning how to solve a problem", note that we are not actually

solving a problem. The computer is going to do that for us. If we could solve the problem ourselves we would have no need to write the program. The premise for a program is that we don't have the time, tenacity or memory capabilities to solve a problem but we do know how to solve it so can instruct a computer to do it for us.

A simple example of this is what is the sum of all integers from 1-10,000. If you wanted to you could sit down with a pencil and paper or a calculator and work this out however the time involved plus the likelihood that at some point you would make a mistake makes that an undesirable option. However you can write and run a program to calculate this sum in less than 5 minutes.

Example 1-1: An example of programming.

```
/* File:example1_1.c
* The program calculates the sum of all integers from 1 to 10,000.
*/
1  #include "stdio.h"
2  #define MAX 10000
3
4  int main()
5  {
6          long sum = 0, number;
7
8          for( number = 1; number <= MAX; number ++)
9       {
10                 sum += number;
11       }
12
13       printf("The sum of all integers from 1 to %ld is : %ld\n", MAX, sum);
14       return 0;
15 }
16
```

This example gives the result 50,005,000. As it happens you can verify this because you know that the sum of integers from 1-N can be calculated as

(N+1)*(N/2)

(10000 + 1)*(10000/2) = 10001*5000 = 50005000

So you have solved the problem of how to calculate the sum of all integers from 1 – 10,000 and the computer has solve the problem of calculating the sum of all integers from 1-10,000.

The computer allows us to do tasks more efficiently, quickly, and accurately than we could by hand—if we could do by hand at all. In order to use this powerful tool, we must specify what we want done and the order in which we want it done. We do this through programming.

However, we have to know the major differences between humans and computers.

Humans have judgment and free will and will not run any instruction they deem not required or nonsensical, whereas a computer will do exactly what it is told with no judgment on the need or sanity of the instruction.

1.1.2 How Do We Write a Program?

To write a sequence of instructions for a computer to follow, we must go through a two-phase process: problem solving and implementation (see Figure 1-1).

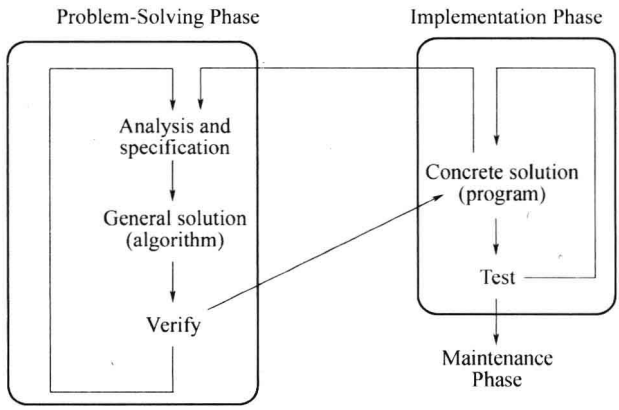

Figure1-1 Programming process

Problem Solving Phase

1. *Analysis and specification.* Understand (define) the problem and what the solution must do.

Every program starts with a specification. This may be a several hundred page document from your latest client or one small paragraph from your professor and pretty much anything in-between.

The specification is very important and specification writing is a whole sub-branch of programming. The important thing is that the specification is correct otherwise, to use a well-known computing adage, garbage in garbage out.

However, specifications are rarely perfect and it is not at all uncommon for the specification to go through several iterations before being finally agreed on.

2. *General solution (algorithm).* Develop a logical sequence of steps that solves the problem.

Having found out how to solve the problem you can then jump in and start coding, right? Wrong. Now is the time to do some program design. Now, how much design is required, and where the design is stored, is rather dependent on the complexity of the program, the experience of the user and the purpose of a program. For instance a simple program being written by an experienced programmer just as a temporary project tool (i.e. will only be in use for a day or 2) probably only requires a little thought about the program design.

The design, once you have it, is not set in stone. It just gives the current ideas about how the program will be written. I do not think I have seen a project where the final code exactly

matches the initial design. Also as the specification changes so will the design. But having it will give a clear place to start coding from and will ultimately lead to better more maintainable code.

3. *Verify*. Follow the steps exactly to see if the solution really does solve the problem.

Implementation Phase

1. *Concrete solution (program).* Translate the algorithm into a programming language.

2. *Test.* Have the computer follow the instructions. Then manually check the results. If you find errors, analyze the program and algorithm to determine the source of the errors, and then make corrections.

Testing does not have to involve running any code. Source review by your peers is a good form of testing too. This is where you sit down round a table and go through the code line by line and examine it for logic errors, conformance to standards and programming errors. This can actually throw up errors that are not made obvious from testing the software by running it.

Once a program has been written, it enters a third phase: maintenance.

Maintenance Phase

1. *Use*. Use the program.

2. *Maintain.* Modify the program to meet changing requirement or to correct any errors that show up in using it.

The programmer begins the programming process by analyzing the problem and developing a general solution called an algorithm. Understanding and analyzing a problem take up much more time than Figure 1-1 implies. They are the heart of the programming process.

Algorithm is a step-by-step procedure for solving a problem in a finite amount of time.

An algorithm is verbal or written description of a logical sequence of actions (or events). After developing a general solution, the programmer tests the algorithm, walking through each step mentally or manually. If the algorithm does not work, the programmer repeats the problem-solving process, analyzing the problem again and coming up with another algorithm.

When the programmer is satisfied with the algorithm, he/she translates it into a programming language. Although a programming language is simple in form, it is not always easy to use. Programming forces you to write very simple, exact instruction. Translating an algorithm into a programming language is called *coding the algorithm.*

Once a program has been put into use, it is often necessary to modify it. Modification may involve fixing an error that is discovered during the use of the program or changing the program in response to changes in the user's requirements. Each time the program is modified, it is necessary to repeat the problem-solving and implementation phase for those aspects of the program that change. This phase of the programming process is known as maintenance and actually accounts for the majority of the effort expended on most programs. Together, the problem-solving, implementation and maintenance phases constitute the *program's life cycle.*

1.2 The Evolution of Programming Language

"As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them — it has created the problem of using its product" (E.W. Dijkstra, Turing Award Lecture, 1972).

Nowadays, there are many programming languages, for example, C, C++, Ada, Pascal, Prolog, FORTRAN, Modula3, Lisp, Java, Scheme. This alphabet soup is the secret power of modern software engineering. As high level computer programming languages, they provide enormous flexibility and abstraction. Programmers are separated from the physical machine allowing complex problem solutions without fretting with the difficulties of ones and zeros. This idea is clarified by an analogy. If we had to think about every phonetic sound made while speaking, communication of abstract ideas would be next to impossible. Much the same way, programming directly with ones and zeros would focus the designer's attention on trivial hardware details instead of on designing abstract solutions. Considering the historical trend that created high level programming, we believe that certain reasonable predictions can be made regarding future advances.

Data Abstraction

One of the keys to successful programming is the concept of abstraction. Abstraction is the crucial to building complex software systems. A good definition of abstraction comes from, and can be summed up as concentrating on relevant aspects of the problem and ignoring those that are not currently important.

The psychological notion of abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low-level details; use of abstraction also permits one to work with concepts and terms that are familiar in the problem environment without having to transform them to an unfamiliar structure.

As the size of our problems grows, the need for abstraction dramatically increases. In simple systems, characteristic of languages used in the 1950s and 1960s, a single programmer could understand the entire problem, and therefore manipulate all program and data structures. Programmers today are unable to understand all of the programs and data — it is just too large. Abstraction is required to allow the programmer to grasp necessary concepts. To understand how abstraction works, it is helpful to show the topology or mapping of a language to the data structures and program modules that the language provides. Once we see the topology of early languages, we can better understand the problems and solutions.

1.2.1 Assembly and Machine Languages

The most basic language of a computer, the machine language, provides program instructions in the bits. Even though most computers perform the same kinds of operations, the

designers of computer may have chosen different sets of binary codes to perform the operation. Therefore, the machine language of one machine is not necessarily the same as the machine language of another machine. The only consistency among computers is that in any modern computer, all data is stored and manipulated as binary codes.

Assembler is a program that translates a program written in assembly language into an equivalent program in machine language.

Early languages had little distinction between programs and data (see Figure 1-2). The data and program co-existed. Because the data and program often were ill-defined, the boundary was irregular. It often was hard to distinguish between data and code. If a true hacker needed to use the number 62, and he or she knew that the machine instruction was coded as a hex 3E — which is equal to decimal 62 — the hacker would reference the instruction elsewhere in the program to refer to decimal 62.

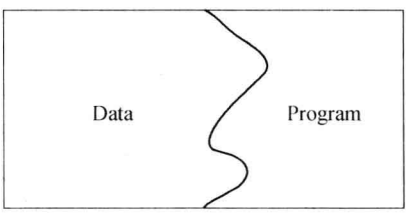

Figure 1-2 The topology of assembly and machines languages

1.2.2 Early Languages

The first programming languages widely used had a clear separation between the data and the program. These languages had a global data structure, but permitted modularization of program structure. Typically, there was only single-level modularization of the program (see Figure 1-3). While this separation of data and program was a good thing, all program segments were at a single level, and typically referenced each other in very complex ways. In software engineering terms, the cohesion was typically very low, and the coupling was quite high. In other words, modules tended to perform many tasks, and there was a lot of dependence on the workings of other modules. In addition, each module had unlimited access to all data because the data was global to all modules. Global data is bad — it makes maintenance extremely difficult, since it is hard to determine which module is 'trashing' the data.

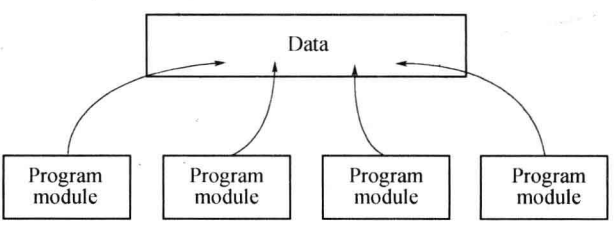

Figure 1-3 The topology of early languages (1950s and 1960s)