ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN TPHCM

KHOA: CÔNG NGHỆ THÔNG TIN

LỚP 06C2

******

BÀI DỊCH

"TEACH YOURSELF C++" – THIRD EDITION

GVHD: Th.s.NGUYỄN TẤN TRẦN MINH KHANG

MÔN : PP LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

NHÓM THỰC HIỆN: **8BIT**

1.Trịnh Văn Long	0612229
2.Đoàn Minh Bảo Long	0612231
3.Hà An Phong	0612330
4.Trần Quang long	0612227
5.Nguyễn Thành Long	0612223
6.Nguyễn Văn Năm	0612326
7.Đỗ Trọng Long	0612232
8. Dương Huỳnh nghĩa	0612285

LỜI MỞ ĐẦU

***

Được sự giúp đỡ, hướng dẫn của thầy các thành viên của nhóm **8BIT** đã cùng nhau thảo luận,nghiên cứu dịch sách "*Teach Yourselt C++, Third Editon*" nghĩa là "*Tự Học C++, ấn bản 3"* của tác giả *Herbert Schildt*. Đây là một cuốn sách rất hay, dễ hiểu và rất hữu ích cho việc học tập bộ môn này cũng như các bộ môn sau này . Đặc biệt là những ai mới bước vào con đường trở thành lập trinh viên quốc tế hay một chuyên viên phần mềm, *Phương Pháp Lập Trình Hướng Đối Tượng* nó định hướng cho người lập trình một cách tổng quan về lập trình. Đối với những sinh viên trong giai đoạn đại cương thì nó đi xuyên suốt bốn năm học.

Các thành viên của nhóm đã cố gắng nhưng cũng không tránh khỏi những sai sót do thiếu kinh nghiệm dịch sách, mong thầy và quí vị độc giả thông cảm. Để cho cuốn sách được hoàn thiên rất mong sự góp ý của các độc giả. Nhóm **8BIT** xin chân thành cảm ơn.

Nhóm 8BIT

MỤC LỤC

CHƯƠNG 1

AN OVERVIEW OF C++ - TỔNG QUAN VỀ C++

Chapter object :

1.1. WHAT IS OBJECT-ORIENTED PROGRAMMING ?.

Lập Trình Hướng Đối Tượng L‡ Gì ?

1.2. TWO VERSIONS OF C++.

Hai Phiên Bản C++

1.3. C++ COMMENTS I/O

Nhập / Xuất C++

1.4. C++ COMMENTS.

Lời Nhận Xét C++

1.5. CLASSES: A FIRST LOOK.

LỚP : Cái Nhìn Đầu Tiên.

1.6. SOME DIFFERENCE BETWEEN C AND C++.

Một Số Khác Biệt Giữa C v‡ C++.

1.7. INTRODUCING FUNCTION OVERLOADING.

Dẫn Nhập Sự Quá Tải H‡m.

1.8. C++ KEYWORDS SHILLS CHECK.

Các Từ Khóa Trong C++.

C++ is an enhanced version of C language. C++ includes everything that is part of C and adds support for object-oriented programming (OOP for short). In addition to, C++ contains many improvements and features that simply make it a "better C", independent of object-oriented programming. With very few, very minor exceptions, C++ is a superset of C. While everything that you know about the C language is fully applicable to C++, understanding its enhanced features will still require a significant investment of time and effort on your part. However, the rewards of programming in C++ will more than justify the effort you put forth.

C++ là một phiên bản của ngôn ngữ C. C++ bao gồm những phần có trong C và thêm vào đó là sự hỗ trợ cho Lập trình hướng đối tượng (viết tắt là OOP). Cộng thêm vào đó, C++ còn chứa những cải tiến và những cải tiến mà đơn giản nhưng đã tạo ra một "phiên bản C tốt hơn", không phụ thuộc vào phương pháp lập trình hướng đối tượng. Với rất ít riêng biệt, C++ có thể xem là tập cha của C. Trong khi những gì bạn biết về C thì hoàn toàn có thể thực hiện trong C++, thì việc hiểu rõ những tính năng của C++ cũng đòi hỏi một sự đầu tư thích đáng về thời gian cũng như nỗ lực của chính bản thân bạn. Tuy nhiên, phần thưởng cho việc lập trình bằng C++ là sẽ ngày càng chứng minh được năng lực của bản thân bạn hơn với những gì bạn đã thể hiện.

The purpose of this chapter is to introduce you to several of the most important features of C++. As you know, the emplements of a computer language do not exist in a void, separate from one another. Instead, they work together to form the complete language. This interrelatedness is especially pronounced in C++. In fact, it is difficult to discuss one aspect of

C++ in isolation because the features of C++ are highly integrated. To help overcome this problem, this chapter provides a brief overview of several C++ features. This overview will enable you to understand the examples discussed later in this book. Keep in mind that most topics will be more thoroughly explored in later chapters.

Mục tiêu của chương này là giới thiệu cho bạn 1 số tính năng quan trọng nhất của C++. Như bạn biết đó, những thành phần của ngôn ngữ máy tính thì không cùng tồn tại trong 1 khoảng không, mà tách biệt với những cái khác, thay vì làm việc cùng nhau để tạo thành một ngôn ngữ hoàn chỉnh. Mối tương quan này được đặc biệt phát biểu trong C++. Sự thật là rất khó để thảo luận về một diện mạo của C++ trong sự cô lập bởi vì những tính năng của C++ đã được tích hợp cao độ. Để giúp vượt qua vấn đề này, chương này cung cấp một cái nhìn ngắn gọn về một số tính năng của C++. Cái nhìn tổng quan này sẽ giúp bạn hiểu hơn về những thí dụ mà chúng ta sẽ thảo luận ở phần sau cuốn sách. Hãy ghi nhớ tất cả những đề mục chúng sẽ giúp bạn hiểu thấu đáo những chương tiếp theo.

Since C++ was invented to support object-oriented programming this chapter begins with a decriptions of OOP. As you will see, many features of C++ are related to OOP in one way or another. In fact, the theory of OOP permeates C++. However, it is important to understand that C++ can be used to write programs that are and are not object oriented. How you use C++ is completely up to you.

Vì C++ được thiết kế ra để hỗ trợ cho việc lập trình hướng đối tượng, nên chương này chúng ta bắt đầu với việc mô tả về lập trình hướng đối tượng (OOP). Rồi bạn sẽ thấy rằng những tính năng khác của C++ cũng lien quan tới OOP bằng cách này hay cách khác. Sự thật là lý thuyết về lập trình hướng đối tượng được trải dàn trong C++. Tuy nhiên, phải hiểu rằng việc dùng C++ để viết chương trình thì có hoặc không có "hướng đối tượng". Việc sử dụng C++ như thế nào là hoàn toàn phụ thuộc vào bạn.

At the time of this writing, the standardization of C++ is being finalized. For this reason, this chapter describes some important differences between versions of C++ that have been in common use during the past several years and the new Standard C++. Since this book teaches Standard C++, this material is especially important if you are using an older compiler.

Vào thời điểm viết cuốn sách này thì việc chuẩn hóa C++ đang được hoàn tất. Vì lý do đó chương này sẽ mô tả những sự khác biệt quan trọng giữa hai phiên bản của C++ mà đã từng được dung phổ biến trong những năm

trước đây và tiêu chuẩn mới trong C++. Vì cuốn sách này dạy về Tiêu chuẩn C++, nên tài liệu này sẽ đặc biệt quan trọng nếu bạn sử dụng những trình biên dịch cũ.

In addition to introducing several important C++ features, this chapter also discusses some differences between C and C++ programming styles. There are several aspects of C++ that allow greater flexibility in the way that you are write programs. While some of these features have little or nothing to do with object-oriented programming, they are found in most C++ programs, so it is appropriate to discuss them early in this book.

Thêm vào việc giới thiệu một số tính năng quan trọng trong C++, chương này cũng thảo luận về những khác biệt giữa hai phong cách lập trình C và C++. Ở đây có vài yếu tố của C++ cho phép nó có tính linh động hơn trong việc viết chương trình. Trong khi những yếu tố này thì rất ít hoặc không có để lập trình hướng đối tượng thì nó được tìm thấy hầu như đầy đủ trong những chương trình của C++, vì thế sẽ thật thỏa đáng nếu thảo luận về chúng trong khởi đầu quyển sách này.

Before you begin, a few general comments about the nature and form of C++ are in order. First, for the most part, C++ programs physically look like C programs. Like a C program, a C++ program begins execution at ***main( )****. To include command-line arguments, C++ uses the same* ***argc, argv*** *convention that C uses. Although C++ defines its own, object-oriented library, it also supports all the functions in the C standard library. C++ uses the same control structures as C. C++ includes all of the built-in data types defined by C.*

Trước khi bạn bắt đầu, một vài chú giải tổng quát về bản chất và biểu mẫu sẽ đưa ra trong trình tự. Đầu tiên, dành cho hầu hết các phần, theo cái nhìn lý tính thì các chương trình C++ trông như các chương trình C. Cũng như C, một chương trình C++ cũng bắt đầu thi hành bởi hàm main(). Để bao gồm những đối số bằng cơ chế dòng lệnh (command-line), C++ cũng sử dụng quy ước **argc, argv** mà C dùng. Mặc dù C++được định nghĩa với những sở hữu bản thân, thư viện hướng đối tượng (object-oriented), nó cũng hỗ trợ những hàm trong thư viện chuẩn của C. C++ sử dụng cấu trúc điều khiển như C. C++ cũng bao gồm nhưng kiểu dữ liệu được xây dựng trong C.

This book assumes that you already know the C programming language. In other words, you must be able to program in C before you can learn to program in C++ by using this book. If you don't knowC, a good starting place is my book Teach Yourseft C, Third Edition (Berkeley: Osborne/McGraw-Hill, 1997). It applies the same systematic approach used in this book and thoroughly covers the entires C language.

Quyển sách này xem như bạn đã biết về ngôn ngữ lập trình C. Nói cách khác bạn phải biết lập trình trong C trước khi bạn học những chương trình C++ được viết trong cuốn sách này. Nếu bạn chưa biết về C, hãy bắt đầu với cuốn sách *Teach yourseft C, Third Edition (Berkeley: Osborne/McGraw-Hill, 1997).*Việc ứng dụng cách tiếp cận có hệ thống được sử dụng trong cuốn sách này sẽ bao phủ toàn vẹn ngôn ngữ C.

Note:

This book assumes that you know how to compile and execute a program using your C++ compiler. If you don't, you will need to refer to your compiler's instructions. (Because of the differences between compilers, it is impossible to give compolation instructions for each in this book). Since programming is best learned by doing, you are strongly urged to enter, compile, and run the examples in the book in the order in which they are presented.

Chú ý:

Cuốn sách này cho rằng bạn đã biết biên dịch và thực thi một chương trình bằng trình biên dịch C++. Nếu chưa, bạn sẽ cần nhiều sự hướng dẫn hơn bởi trình biên dịch của bạn. (Bởi vì những sự khác nhau giữa các trình biên dịch, sẽ là không khả thi khi đưa ra những sự hướng dẫn về sự biên dịch trong cuốn sách này.) Vì chương trình là bài học thực hành tốt nhất, nên bạn cần phải mạnh mẽ tiến đến, biên dịch và chạy thử những ví dụ trong cuốn sách này để biết dược cái gì đang hiện diện đằng sau những ví dụ ấy.

1.1. WHAT IS OBJECT-ORIENTED PROGRAMMING ?- LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG LÀ GÌ ?

Object-oriented programming is a powerful way to approach the task of

programming. Since its early beginnings, programming has been governed by various methodologies. At each critical point in the evolution of programming, a new approach was created to help the programmer handle increasingly complex programs. The first programs were created by toggling switches on the front panel of the computer. Obviously, this approach is suitable for only the smallest programs. Next, assembly language was invented, which allowed longer programs to be written. The next advance happened in the 1950s when the first high-level language (FORTRAN) was invented.

Lập trình hướng đối tượng là con đường mạnh mẽ để tiếp cận nhiệm vụ lập trình. Từ buổi bình minh, lập trình đã bị chi phối bởi nhiều phương pháp khác nhau. Tại mỗi thời điểm cách tân trong sự phát triển của lập trình, một cách tiếp cận mới lại ra đời với mục đích giúp lập trình viên gia tăng khả năng kiểm soát hay điều khiển những chương trình mang tính phức tạp. Những chương trình đầu tiên được tạo ra bởi những công tắc đảo chiều được gắn trên một cái bảng phía trước máy tính. Hiển nhiên là cách tiếp cận này chỉ đáp ứng được cho những chương trình nhỏ nhất. Tiếp theo, ngôn ngữ assembly (hay còn gọi là Hợp ngữ) được ra đời, cho phép viết những chương trình dài hơn. Một ngôn ngữ cấp cao hơn xuất hiện vào những năm của thập niên 1950 khi đó ngôn ngữ cấp cao đầu tiên được ra đời (đó là FORTRAN).

By using a high-level language, a programmer was able to write programs that were several thousands lines long. However,the method of programming used early on was an ad hoc, anything-goes approach. While this is fine relatively short programs, it yields unreadable (and unmanageable) "spaghetti code" when applied to larger programs. The elimination of spaghetti code became feasible with the invention of structured programming languages in the 1960s. These languages include Algol and Pascal. In loose terms, C is a structured language and most likely the type of programming relies on well-defined control structures, code blocks, the absence (or at least minimal use) of GOTO, and stand-alone subroutines that support recursion and local variables. The essence of structured programming is the reduction of a program into its constituent elements. Using structured programming, the average programmer can create and maintain programs that are up to 50,000 lines long.

Bằng việc sử dụng ngôn ngữ cấp cao, các lập trình viên đã có thể viết nên những chương trình có độ dài lên tới vài ngàn dòng. Tuy nhiên, phương pháp lập trình này cũng chỉ được sử dụng như một thứ tạm thời, không gì tiến triển được. Trong khi những mối quan hệ giữa các chương trình ngắn khá khả quan

thì việc thiết kế lại quá khó (và không kiểm soát được) loại "mã spaghetti" này khi ứng dụng cho các chương trình cỡ lớn. Cuộc đấu tranh với loại mã spaghetti này đã tạo đà cho sự phát minh ra *cấu trúc của những ngôn ngữ lập trình* vào thập niên 1960. Những ngôn ngữ này là Algol và Pascal. Trong tương lai không chắc chắn, C là một ngôn ngữ có cấu trúc và gần như là kiểu lập trình mà bạn bạn đang dung với tên gọi là lập trình có cấu trúc. Lập trình cấu trúc dựa trên những cấu trúc điều khiển, những khối mã dễ xác định, sự vắng mặt (hay gần như rất ít sử dụng) của GOTO, và những thủ tục con đứng một mình để hỗ trợ cho phương pháp đệ quy và những biến cục bộ. Bản chất của lập trình cấu trúc là sự giảm thiểu những yếu tố cấu thành bản chất bên trong của một chương trình. Bằng việc sử dụng phương pháp lập trình này, những lập trình viên với năng lục trung bình cũng có thể tạo ra và bảo trì những chương trình mà có khối lượng lên tới 50.000 dòng.

Although structured programming has yielded excellent results when applied to moderately complex programs, even it fails at some point, after a program reaches a certain size. To allow more complex programs to be written, a new approach to the job of programming was needed. Towards this end, object-oriented programming was invented. OOP takes the best of ideas embodied in structured programming and combines them with powerful new concepts that allow you to organize your programs more effectively. Object-oriented programming encourages you to decompose a problem into its constituent parts. Each component becomes a seft-contained object that contains its own instructions and data that relate to that object. In this way, complexity is reduced and the programmer can manage larger programs.

Mặc dù lập trình cấu trúc đã đạt được những kết quả xuất sắc khi ứng dụng vào những chương trình có độ phức tạp trung bình, thậm chí chỉ trục trắc ở vài chỗ sau khi một chương trình đã đạt đến một kích thước chắc chắn. Để cho phép viết được những chương trình phức tạp hơn, một cách tiếp cận mới công việc lập trình đã được đề ra. Cuối cùng, phương pháp lập trình hướng đối tượng đã được phát minh. Lập trình hướng đối tượng (OOP) đã cụ thể hóa những ý tưởng tuyệt vời vào trong cấu trúc lập trình và kết hợp chúng với những khái niệm chặt chẽ mà cho phép bạn tổ chức những chương trình của mình một cách có hiệu quả. OOP khuyến khích bạn phân tích một vấn đề nằm trong bản chất của nó. Mỗi bộ phận đều có thể trở thành một đối tượng độc lập mà chứa đựng những câu lệnh hay dữ liệu lien quan đến đối tượng đó. Bằng cách này sự phức tạp sẽ được giảm xuống và lập trình viên có thể quản lý những chương trình cỡ lớn.

All OOP languages, including C++, share three common defining traits: encapsulation, polymorphism, and inheritance. Let's look at the three concepts now.

Tất cả những ngôn ngữ lập trình hướng đối tượng, bao gồm cả C++, đều mang ba đặc điểm chung sau đây: tính đóng gói, tính đa dạng, và tính kế thừa. Còn chờ gì nữa phải tìm hiểu các khái niệm trên ngay thôi.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keep both safe from outside interference and misuse. In an object-oriented language, code and data can be combined in such a way that a seft-contained "black box" is created. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation.

Tính đóng gói hay tích hợp:

Tính đóng gói là việc ràng buộc những đoạn mã và những dữ liệu lại, điều khiển và giữ chúng an toàn khỏi những ảnh hưởng và những mục đích sai trái. Trong một ngôn ngữ lập trình hướng đối tượng, mã và dữ liệu có thể được kết hợp theo cách mà một cái "hộp đen" độc lập được tạo ra. Khi mã và dữ liệu liên kết với nhau theo kiểu này thì một đối tượng được tạo nên. Nói cách khác, một đối tượng là sản phẩm của sự đóng gói với nguyên liệu là mã và dữ liệu.

With an object, code, data, or both may be private to that object or public. Private code or data is known to and accessible only by another part of the object. That is, private code or data can't be accessed by a piece of the program that exists outside the object. When code and data is public, other parts of your program can access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

Trong phạm vi của một đối tượng, mã, dữ liệu , hay cả hai có thể là phần *private* (phần riêng) của đối tượng hay là *public* (phần chung). Phần mã và dữ liệu trong phần private được biết và truy xuất bởi các thành phần khác của đối

tượng. Nghĩa là mã và dữ liệu trong phần private không thể được truy xuất bởi bất kỳ phần nào trong chương trình mà tồn tại bên ngoài đối tượng đó. Khi mã và dữ liệu là kiểu public thì mọi phần trong chương trình của bạn đều có thể truy xuất đến nó ngay cả khi nó được định nghĩa bên trong một đối tượng. Đặc trưng là những phần có kiểu public của một đối tượng thì được sử dụng để cung cấp một giao diện điều khiển những thành phần kiểu private của một đối tượng.

For all intents and purposes, an object is variable of a user-defined type. It may seem strange that an object that links both code and data can be thought of as a variable. However, in an object-oriented programming, this is precisely the case. Each time you define a new type of object, you are creating new data type. Each specific instance of this data type is a compound variable.

Để phù hợp với nhiều ý định và mục đích, một đối tượng có thể xem là **một biến** theo kiểu người dùng tự định nghĩa. Có thể bạn sẽ lấy làm ngạc nhiên khi một đối tượng có thể liên kết những lệnh(hay mã) và dữ liệu lại mà lại xem như là một biến. Tuy vậy, trong lập trình hướng đối tượng thì điều đó là hoàn toàn chính xác. Mỗi khi bạn định nghĩa một loại đối tượng thì bạn đang tạo ra một kiểu dữ liệu mới. Mỗi một kiểu đối tượng như vậy được coi là một biến phức hợp (hay biến ghép).

Polymorphism

Polymorphism (from the Greek,meaning "many form") is the quality that allows one name to be used for two or more related but technically different purposes. As it relates to OOP, polymorphism allows one name to specify a general class of actions. Within a general class of actions, the specific action are be replied is determined by the type of data. For example, in C, which does not significantly support polymorphism, the absolute value action requires three distinct function names: ***abs( )****,* ***labs( )****, and* ***fabs( )****. Three functions compute and return the absolute value of an integer, a long integer, and float-point value, respectively. However, in C++, which supports polymorphism, each function can be called by the same name, such as* ***abs( )****. (One way this can be accomplished is shown later in the chapter.) The type of the data used to call the function determines which*

specific version of the function is actually executed. As you will see, in C++, it is possible to use ine function name for many different purposes. This is called function overloading.

Tính đa dạng

Polymorphism (Theo tiếng Hy Lạp có nghĩa là "đa thể") hay tính đa dạng là đặc tính cho phép một tên có thể được sử dụng cho hai hay nhiều họ nhưng với nhiều mục đích ngữ nghĩa khác nhau. Vì nó liên quan đến OOP, nên tính đa dạng cho phép một tên để chỉ rõ những hành động chung của một lớp. Bên trong những hành động chung của một lớp, hành động đặc trưng để ứng dụng được định nghĩa bởi một kiểu dữ liệu. Ví dụ, trong C, thì tính đa dạng không được hỗ trợ một cách đáng kể, hành động lấy giá trị tuyệt đối đòi hỏi ba tên hàm riêng biệt là: **abs( ), labs( ), fabs( )**. Những hàm này tính toán và trả lại giá trị tuyệt đối của một số nguyên, một số nguyên dài, và một số thực một cách riêng biệt. Tuy nhiên, trong C++, được hỗ trợ cho tính đa dạng thì chỉ cần mỗi hàm **abs( )**. (Cách này sẽ được nêu đầy đủ trong phần sau của chương này). Kiểu dữ liệu dùng trong hàm sẽ quyết định việc hàm nào sẽ được gọi thực thi. Bạn sẽ thấy, trong C++, hoàn toàn có thể sử dụng một tên hàm cho nhiều mục đích khác nhau. Điều này còn được gọi là ***sự quá tải hàm*** (function overloading).

More generally, the concept of polymorphism is characterized by the idea of "one interface, multiple methods", which means using a generic interface for a group of related activities. The advantage of polymorphism is that it helps to reduce complexity by allowing one interface to specify a general class of action. It is the compiler's job to select the specific action as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the general interface. As the example in the preceeding paragraph illustrates, having three names for the absolute value function instead of just one makes the general activity of obtaining the absolute value of a number more complex than it actually is.

Tổng quát hơn, khái niệm tính đa dạng được đặc trưng bởi ý tưởng " một giao diện nhưng nhiều cách thức ", điều này có nghĩa là sử dụng một giao diện chung cho một nhóm những hành động có liên quan. Sự thuận lợi của tính đa dạng là giúp giảm bớt sự phức tạp bằng cách cho phép một giao diện để chỉ rõ hành động của một lớp tổng quát. Đó là cách mà trình biên dịch chọn hành động đặc trưng để thi hành trong từng tình huống cụ thể. Bạn, một người lập trình, không cần phải làm công việc này một cách thủ công. Bạn cần nhớ và

tận dụng cái giao diện chung. Như thí dụ minh họa trong đoạn văn trước, cần đến ba tên hàm để lấy giá trị tuyệt đối thay vì chỉ cần một tên nhưng chỉ hành động chung là tính giá trị tuyệt đối của một số mà thật sự phức tạp.

Polymorphism can be applied to operators, too. Virtually all programming languages contain a limited application of polymorphism as it relates to the arithmetic operators. For example, in C++, the + sign is used to add integers, long intefer, characters, and floating-point values. In these cases, the compiler automatically knows which type of arithmetic to apply. In C++, you can extend this concept to other types of data that you define. This type of polymorphism is called operator overloading.

Tính đa dạng cũng có thể sử dụng cho các toán tử. Hầu như mọi ngôn ngữ lập trình đều có một sự giới hạn tính đa dạng đối với các toán tử số học. Ví dụ, trong C, dấu + được dung để cộng các giá trị số nguyên, số nguyên dài, ký tự và số thực. Trong những trường hợp này, trình biên dịch tự động hiểu loại số học nào sẽ được áp dụng. Trong C++, bạn có thể mở rộng khái niệm này đối với những kiểu dữ liệu khác mà bạn định nghĩa. Loại đa dạng này được gọi là sử quá tải các toán tử (operator overloading).

The key point to remember about polymorphism is that it allows you to handle greater complexity by allowing the creation of stamdard interfaces to related activities.

Điểm mấu chốt của vấn đề cần nhớ là tính đa dạng cho phép bạn kiểm soát với độ phức tạp rất cao bằng cách cho phép tạo ra một giao diện tiêu chuẩn cho các hành động có liên quan.

Inheritance

Inheritance is the process by which one object can be acquire the properties of another. More specifically, an object can inherit a general set of properties to which it can add those features that are specific only to itself. Inheritance is important because it allows an object to support the concept of hierarchical classification. Most information is made manageable by hierarchical classification. For example, think about the description of a house. A house is part of the general called ***building.*** *In turn,* ***building*** *is*

*part of the more general class **structure**, which is part of the even more general of objects that we call **man-made**. In each case, the child class inherits all those qualities associated with the parent and adds to them its own defining characteristics. However, through inheritance, it is possible to describe an object by stating what general class (or classes) it belongs to along with those specific traits that make it unique. As you will see, inheritace plays a very important role in OOP.*

Tính kế thừa

Tính kế thừa là một quá trình mà một đối tượng có thể sử dụng các đặc tính của một đối tượng khác. Cụ thể hơn, một đối tượng có thể thừa hưởng những đặc tính chung của đối tượng khác để thêm các tính năng khác này vào như một phần của chính bản thân nó. Sự thừa hưởng rất quan trọng bởi vì nó cho phép một đối tượng chấp nhận khái niệm sự phân loại có tôn ti (***hierachial classification)***. Hầu hết các thông tin được tạo ra được quản lý bởi sự sự phân loại có trật tự này. Ví dụ, hãy nghĩ về một ngôi nhà. Một ngôi nhà là một phần của một lớp tổng quát gọi là **công trình**. Cứ như vậy, công trình lại là một phần của một lớp chung gọi là **kiến trúc**, và kiến trúc lại là thành phần của một lớp mà ta gọi là những vật phẩm **nhân tạo**. Trong mỗi trường hợp, lớp con thừa hưởng tất cả những tính chất được liên đới với lớp cha và cộng thêm với những tính chất được định nghĩa bên trong chúng. Nếu không sử dụng sự phân loại có trật tự thì mỗi đối tượng sẽ phải định nghĩa tất cả những đặc tính mà rõ ràng liên quan đến nó. Tuy nhiên, qua sự kế thừa, ta có thể mô tả một đối tượng bằng cách phát biểu những đặc điểm nổi bật thuộc về một lớp tổng quát (hay một họ các lớp) mà tạo nên tính độc nhất của chúng. Bạn sẽ thấy rằng sự thừa hưởng chiếm một vai trò rất quan trọng trong OOP.

CÁC VÍ DỤ :

1.Sự đóng gói không phải là điều gì hoàn toàn mới mẻ trong OOP. Theo một góc độ thì sự đóng gói có thể là một thành tựu khi sử dụng ngôn ngữ C. Ví dụ, khi bạn sử dụng một hàm của thư viện thì thật ra bạn đang sử dụng thủ tục cái "hộp đen", những gì bên trong bạn đâu thể gây ảnh hưởng hay tác động vào chúng (trừ phi, có lẽ là chỉ "bó tay" với những hành động cố tình phá hoại). Hãy xem xét hàm **fopen( )**, khi dùng để mở một file, thì vài biến nội bộ sẽ được khởi tạo và gán giá trị. Càng xa khỏi vùng chương trình liên quan thì càng bị ẩn và khó truy cập. Tuy nhiên, C++ vẫn cung cấp cho bạn nhiều cách tiếp cận sự đóng gói này an toàn hơn.

2.Trong thế giới thực, những ví dụ về sự đa dạng thì khá là chung chung. Ví dụ, hãy xem xét cái vô-lăng lái xe của bạn. Nó làm việc cũng giống như chiếc xe bạn sử dụng thiết bị lái năng lượng, thiết bị lái bằng thanh răng, hay tiêu chuẩn, thiết

bị lái thủ công. Vấn đề là cái bề ngoài hình dáng không quan trọng mà là cái cách các động cơ bánh răng hoạt động (hay cái phương thức).

3.Sự thừa kế các đặc tính và khái niệm tổng quát hơn về lớp là nền tảng để hiểu cách tổ chức. Ví dụ, rau cần tây là một loại(hay thành viên) của lớp rau quả, rau quả lại là một thành phần của lớp cây trồng. Lần lượt như vậy, cây trồng là những sinh vật sống, và cứ thế. Nếu không có sự sự phân loại trong tôn ti thì hệ thống hệ thống kiến thức sẽ không khả thi.

BÀI TẬP:

Hãy nghĩ về vai trò của sự phân loại và tính đa dạng trong cuộc sống hằng ngày của chúng ta.

1.2. TWO VERSIONS OF C++ - HAI PHIÊN BẢN CỦA C++

At the time of this writing, C++ is in the midst of a transformation. As explained in the preface to this book, C++ has been undergoing the process of standardization for the past several years. The goal has been to create a stable, standardized, featured-rich language, that will suit the needs of programmers well into the next century. As a result, there are really two versions of C++. The first is the traditional version that is based upon Bjarne Stroustrup's original designs. This is the version of C++ that has been used by programmers for the past decade. The second is the new Standard C++, which was created by Stroustrup and the ANSI/ISO standardization committee. While these two versions of C++ are very similar at there core, Standard C++ contains several enhancements not found in traditional C++. Thus, Standard C++ is essentially a superset of traditional C++.

Vào thời điểm viết cuốn sách này, C++ là cái chuyển giao của một sự thay đổi. Như đã nói trong phần mở đầu của cuốn sách, C++ phải trải qua một quá trình chuẩn hóa trong vài năm trước đây. Mục tiêu là tạo ra sự ổn định, được chuẩn hóa, một ngôn ngữ giàu tính năng để phục vụ tốt cho các lập trình viên trong thế kỷ tiếp theo. Kết quả là đã tạo ra hai phiên bản C++. Cái đầu tiên là phiên bản truyền thống mà dựa trên nguồn gốc bản phác thảo của Bjarne Stroustrup. Đây là phiên bản C++ được các lập trình viên sử dụng trong thập niên trước đây. Cái thứ hai là "new Standard C++", được tạo ra bởi Stroustrup và ủy ban tiêu chuẩn ANSI/ISO. Trong khi hai phiên bản C++ có hạt nhân khá là giống nhau thì Standard C++(hay C++ chuẩn) chứa vài sự nổi bật mà không

tìm thấy được trong phiên bản C++ truyền thống. Vì vậy, C++ chuẩn một cách đặc biệt là một tập cha của bản C++ truyền thống.

This book teaches Standard C++. This is the version of C++ defined by the ANSI/ISO standardization committee, and it is the version implemented by all modern C++ compilers. The code in this book reflects the contemporary coding style and practices as encouraged by Standard C++. This means that what you learn in this book will be applicable today as well as tomorrow. Put directly, Standard C++ is the future. And, since Standard C++ encompasses all features found in earlier versions of C++, what you learn in this book will enable you work in all C++ programming environments.

Cuốn sách này viết về C++ chuẩn. Phiên bản này được định nghĩa bởi ủy ban tiêu chuẩn ANSI/ISO, và nó là công cụ của tất cả các trình biên dịch C++ hiện nay. Những đoạn mã viết trong cuốn sách này tương ứng với phong cách viết mã đương thời và những bài thực hành theo bản C++ chuẩn. Điều này có nghĩa là những gì bạn học trong cuốn sách này thì đều sự dụng được hiện nay cũng như tương lai. Nói trực tiếp là C++ chuẩn chính là tương lai. Và, vì C++ đã hoàn thiện tất cả các tính năng có trong các bản C++ trước nên tất cả những gì bạn học đều có thể làm việc trong tất cả các môi trường lập trình C++.

However, if you are using an older compiler, it might not accept all the programs in this book. Here's why: During the process of standardization, the ANSI/ISO committee added many new features to the language. As these features were defined, they were implemented by the compiler developers. Of course, there is always a lag time between the addition of a new feature to the language and the availability of the features in commercial compilers. Since features were added to C++ over a period of years, an older compiler might not support one or more of them. This important because two recent additions to the C++ language affect every program that you will write-even the simplest. If you are using an oder compiler that does not accept these new features, don't worry. There is an easy workaround, which is described in the following paragraphs.

Tuy nhiên, nếu bạn sử dụng một trình biên dịch quá cũ, thì có khả năng nó sẽ không chấp nhận những đoạn mã viết trong cuốn sách này. Đây cũng là nguyên nhân mà trong suốt quá trình chuẩn hóa, ủy ban tiêu chuẩn ANSI/ISO đã tăng thêm nhiều tính năng mới cho ngôn ngữ này. Khi những tính năng này được định nghĩa thì chúng cũng được các nhà phát triển trình biên dịch bổ

sung. Tất nhiên đây luôn là một khoảng thời gian chậm chạp giữa việc thêm tính năng vào một ngôn ngữ và việc biến nó khả thi trong các trình biên dịch. Những tính năng này đã được thêm vào trong một thời kỳ nhiều năm nên những trình biên dịch đời cũ thì không được hỗ trợ để biên dịch chúng. Và quan trọng hơn là hai sự bổ sung gần đây vào ngôn ngữ C++ sẽ ảnh hưởng đến mỗi chương trình mà bạn sẽ viết, thậm chí dù là chương trình đơn giản nhất. Nếu bạn sử dụng trình biên dịch cũ hơn thì nó sẽ không chấp nhận những tính năng mới này, nhưng đừng quá lo lắng. Có một cách giải quyết dễ dàng sẽ được mô tả trong những đoạn tiếp theo.

*The differences between old-style and modern code involve two new features: new-style headers and the **namespace** statement. To demonstrate these deffirences we will begin by looking at two versions of a minimal, do-nothing C++ program. The first version, shown here, reflects theway C++ programs were written until recently. (That is, it uses old-style coding.)*

Sự khác biệt giữa phong cách viết mã cũ và hiện đại liên quan đến hai tính năng sau: những kiểu khai báo mới và sự giới thiệu không gian tên **(namespace).** Để chứng minh những sự khác biệt này chúng tôi sẽ bắt đầu bằng việc nhìn lại hai bản C++ ở một khía cạnh nhỏ, không đụng đến chương trình C++. Trong bản đầu tiên, hãy xem sau đây, phản ánh cách những chương trình C++ được viết trong thời gian gần đây. (Đó là viết mã theo phong cách cũ.)

```
/*
Những chương trình với phong cách C++ truyền thống
/*

     #include <iostream.h>

     int main()
     {
          /* vùng mã chương trình */
          return 0;
     }
```

*Since C++ is built on C, this skeleton should be largely familiar, but pay special attention to the **#include** statement. This statement includes the file **iostream.h**, which provides support for C++'s I/O system. (It is to C++ what **stdio.h** is to C.)*

Here is the second version of the skeleton, which uses the modern style:

Vì C++ được xây dựng dựa trên C, nên cái khung rất quen thuộc nhưng hãy chú ý đến dòng phát biểu **#include.** Lời phát biểu này khai báo file **iostream.h** mà được cung cấp cho hệ thống nhập xuất của C++ (trong C là **stdio.h**).

Sau đây là bản thứ hai của cái khung này, sử dụng phong cách hiện đại:

```
/*

Phong cách lập trình C++ hiện đại là sử dụng
lời khai báo theo phong cách mới và vùng không
gian tên

*/

#include <iostream>

using namespace std;


int main()

{

     /* vùng mã chương trình */

     return 0;

}
```

*Notice the two lines in this program immediately after the first comment; this is where the changes occur. First, in the **#include** statement, there is no **.h** after the name **iostream**. And second, the next line, specifying a namespace, in new. Although both the new-style headers and namespaces will be examined in detail later in this book, a brief overview is in order now.*

Chú ý hai dòng trong chương trình này ngay lập tức sau khi đọc xong lời nhận xét đầu tiên; ở đây có sự thay đổi. Đầy tiên, trong lời phát biểu **#include** không có **.h** theo sau tên **iostream**. Và cái thứ hai là dòng tiếp theo, chỉ rõ vùng không gian tên, đây là nét mới. Mặc dù cả hai cái này sẽ được kiểm nghiệm chi tiết trong các phần sau của cuốn sách nhưng chúng ta hãy xem xét ngắn gọn chúng.

The new C++ headers

*As you know from your C programming experience, when you use a library function in a program, you must include its header file.This is done using the **#include** statement. For example, in C, to include the header file for the I/O functions, you include **stdio.h** with a statement like this.*

Những đầu mục mới của C++

Như bạn biết từ những kinh nghiệm lập trình bằng C++, khi bạn sử dụng một hàm thư viện trong một chương trình, bạn phải bao hàm lời khai báo file. Cái này dùng cách phát biểu **#include**. Ví dụ, trong C, để khai báo sử dụng file cho hàm I/O (hàm nhập xuất), bạn sử dụng **stdio.h** với lời phát biểu như sau:

```
#include <stdio.h>
```

*Here **stdio.h** is the name of the file used by the I/O functions, and the preceding statement causes that file to be included in your program. The key point is that the **#include** statement includes a file.*

Ở đây, **stdio.h** là tên file được sử dụng bởi hàm nhập xuất, và việc lời phát biểu này đi trước đã làm cho file đó được bao hàm trong chương trình của bạn. Điểm mấu chốt là **#include** phát biểu việc *includes a file (bao h‡m một file)*.

When C++ was first invented and for several years after that,it used the same style of headers ad did C. In fact, Standard C++ still supports C-style headers for header files that you create and for backward compatibility. However, Standard C++ has introduced a new kind of header that is used by the Standard C++ library. The new-style headers do not specify filenames. Instead, they simply specify standard identifiers that might be mapped to files by the compiler, but they need not be. The new-style C++

headers are abstractions that simply guarantee that the appropriate prototypes and definitions required by the C++ library have been declared.

Khi C++ lần đầu tiên được tạo ra và vài năm sau đó, nó đã sử dụng những kiểu khai báo của C. Thật ra, C++ chuẩn vẫn còn hỗ trợ những khai báo kiểu C cho những file khai báo mà bạn tạo ra và cho sự tương thích ngược. Tuy nhiên, C++ chuẩn đã giới thiệu một kiểu khai báo mới mà được sử dụng trong thư viện C++ chuẩn. Kiểu khai báo mới này *không chỉ rõ* tên những file. Thay vì đơn giản chỉ rõ những định dạng tiêu chuẩn mà chúng được ánh xạ vào các file bởi trình biên dịch nhưng chúng không cần. Kiểu khai báo mới này của C++ là những sự trừu tượng hóa như sự đảm bảo đơn thuần, những vật mẫu hay những định nghĩa thích đáng được yêu cầu bởi thư viện C++.

*Since the new-style header is not a filename, it does not have a **.h** extension. Such a header consist solely of the header name contained between angle brackets supported by Standard C++.*

Vì khai báo mới không phải là một tên file nên đuôi **.h** không cần thiết. Một khai báo gồm có một tên khai báo đơn độc trong dấu ngoặc < >.

Ví dụ, sau đây là vài kiểu khai báo được chấp nhận trong C++ chuẩn:

```
<iostream>

<fstream>

<vector>

<string>
```

*The new-style headers are included using the **#include** statement. The only difference is that the new-style headers do not necessarily represent filenames.*

Những đầu mục kiểu mới này bao gồm việc sử dụng sự trình bày **#include**. Sự khác nhau duy nhất là những đầu mục kiểu mới không tất yếu đại diện cho tên file.

*Becase C++ includes th entire C function library, it still supports the standard C-style header files associated with that library. That is, header files such as **stdio.h** and **ctype.h** are still available. However, Standard C++ also defines new-style headers you can use in place of these header files. The C++ versions of the standard C headers simply add c prefix to the filename and drop the **.h**. For example, the new-style C++ header for **math.h** is **<cmath>**, and the one for*

string.h *is* ***<cstring>****. Although it is currently permissible to include a C-tyle header file when using C library functions, this approach is deprecate by Standart C++.(That is, it is not recommended.) For this reason, this book will use new-style C++ header in all* ***#include*** *statements. If your compiler does not support new-style headers for the C function library, simply substitute the old-style, C-like headers.*

Bởi vì C++ bao gồm toàn bộ thư viện chức năng C, nó vẫn còn hỗ trợ C tiêu chuẩn-những hồ sơ đầu mục kiểu liên quan đến thư viện kia. Điều đó, đầu mục sắp xếp như **stdio.h** và **ctype.h** thì vẫn còn sẵn có. Tuy nhiên, C++ tiêu chuẩn cũng định nghĩa những đầu mục kiểu mới bạn có thể sử dụng thay cho những hồ sơ đầu mục này. Những phiên bản C++ của những đầu mục C tiêu chuẩn đơn giản thêm tiền tố **c** vào đầu tên file và kết thúc bằng **.h** .Chẳng hạn, đầu mục C++ kiểu mới cho **math.h** là **<cmath>**, và cho **string.h** là **< cstring>**. Dù hiện thời thừa nhận bao gồm một hồ sơ đầu mục tyle C khi sử dụng những hàm thư viện C, phương pháp này bị phản đối bởi C++ chuẩn.(Nghĩa là, nó không được khuyến cáo.). Với lý do này, sách này sẽ sử dụng đầu mục C++ kiểu mới trong tất cả các câu lệnh. Nếu trình biên dịch (của) các bạn không hỗ trợ những đầu mục kiểu mới cho thư viện chức năng C, đơn giản thế những đầu mục cũ, giống như C.

Since the new-style header is a recent addition to C++, you will still find many, many older programs that don't use it. These programs instead use C-tyle headers, in which a filename is specified. As the old-style skeletal program shows, the traditional way to include the I/O header is as shown here:

Đầu mục kiểu mới là một sự bổ sung cho C++, bạn sẽ vẫn còn tìm thấy gần đây nhiều chương trình cũ không sử dụng nó. Những chương trình này thay vào đó sử dụng C- những đầu mục tyle, trong đó một tên file được chỉ rõ. Như chương trình lệnh điều khiển kiểu cũ hiện ra, cách truyền thống để bao gồm đầu mục vào/ra được cho thấy như ở đây:

```
#include < iostream.h >
```

This cause the file ***iostream.h*** *to be include in your program. In general, an old-style header will use the same name as its corresponding new-style header with a* ***.h*** *appended.*

Đây là căn nguyên tập tin **iostream.h** được bao gồm trong chương trình của các bạn. Nói chung, một đầu mục kiểu cũ sẽ sử dụng cùng tên với đầu mục kiểu mới tương ứng với nó với **.h** được nối vào.

As of this writing, all C++ compilers support the old-style headers. However, the

old style headers have been declared obsolete, and their use in new programs is not recommended. This is why they are not used in this book.

Kể từ sự ghi này, tất cả các trình biên dịch C++ hỗ trợ những đầu mục kiểu cũ. Tuy nhiên, những đầu mục kiểu cũ được khai báo đã lỗi thời, và sự sử dụng chúng trong những chương trình mới không được khuyến cáo. Điều này là lý do tại sao chúng không được sử dụng trong sách này.

Remember: *While still common in existing C++ code, old-style headers are obsolete.*

Ghi Nhớ : Trong mã C++ phổ biến hiện hữu, những đầu mục kiểu cũ là lỗi thời.

NAMESPACES

When you include a new-style header in your program, the contents of that header are contained in the std namespace. A namespace is simply a declarative region. The purpose of a namespace is to localize the names of library functions and other such items were simply placed into the global namespace(as they are in C). However, the contents of new-style headers are placed in the std namespace. We will look closely at namespace later in this book. For now, you don't need to worry about them because you can use the statement.

Khi bạn bao gồm một đầu mục kiểu mới trong chương trình của các bạn, nội dung của đầu mục kia được chứa đựng trong namespace **std** . Một namespace đơn giản là một vùng tường thuật. Mục đích của một namespace là sẽ địa phương hóa tên của những hàm thư viện và những thư mục như vậy khác thì đơn giản được đặt vào trong toàn cục namespace (giống như trong C). Tuy nhiên, nội dung của những đầu mục kiểu mới được đặt trong namespace **std**. Chúng tôi sẽ có cái nhìn rõ ràng hơn về namespace trong cuốn sách này ở phần sau. Bây giờ, bạn không cần lo lắng họ bởi vì bạn có thể sử dụng câu lệnh

```
using namespace std;
```

to bring the std namespace into visibility (i.e., to put std into the global namespace). After this statement has been complied, there is no difference between working with an old-style header and a new-style one.

để mang đến sự rõ ràng cho namespace **std** (thí dụ, **std** được đặt vào trong namespace toàn cục). Sau khi câu lệnh này được tuân theo, không có sự khác nhau

giữa việc làm việc với một đầu mục kiểu cũ và một đầu mục kiểu mới.

Làm việc Với một Trình biên dịch cũ (WORKING WITH AN OLD COMPILER):

As mentioned, both namespaces and the new-style headers are recent additions to the C++ language. While virtually all new C++ compilers support these features, older compilers might not. If you have one of these older compilers, it will report one or more errors when it tries to compile the first two lines of the sample programs in this book. If this is the case, there is an easy workaround: simply use an old-style header and delete the namespace statement. That is, just replace

Như đã đề cập, cả namespaces lẫn những đầu mục kiểu mới là sự bổ sung cho ngôn ngữ C++ gần đây. Trong khi thực tế tất cả các trình biên dịch C++ mới hỗ trợ những đặc tính này, những trình biên dịch cũ hơn có lẽ không hỗ trợ. Nếu bạn có một trong số những trình biên dịch cũ này, nó sẽ báo cáo một hoặc nhiều lỗi khi nó thử biên tập những dòng đầu tiên trong chương trình mẫu trong cuốn sách này. Nếu đó là nguyên nhân, có một cách giải quyết dễ dàng: Đơn giản sử dụng một đầu mục kiểu cũ và xóa khai báo namespace. Điều đó, thay thế

```
#include <iostream>

using namespace std;
```

bằng

```
#include <iostream.h>
```

This change transforms a modern program into a traditional-style one. Since the old-style header reads all of its contents into the global namespace, there is no need for a namespace statement.

Sự thay đổi này thay đổi một chương trình hiện đại vào trong một kiểu truyền thống. Một khi đầu mục kiểu cũ biên dịch tất cả các nội dung làm bằng lòng namespace toàn cục, thì không có nhu cầu phải khai báo namespace.

One other point: For now and for the next few years, you will see many C++ programs that use the old-style headers and that do not include a namespace statement. Your C++ compiler will be able to compile them just fine. For new programs, however, you should use the modern style because it is the only style of program that compiles with Standard C++. While old-style programs will continue to be supported for many years, they are technically noncompliant.

Một điểm khác: bây giờ và trong những năm tiếp theo, bạn sẽ nhìn thấy nhiều chương trình C++ mà sử dụng những đầu mục kiểu cũ mà không bao gồm câu lệnh namespace. Trình biên dịch C ++ của các bạn sẽ có khả năng để biên tập chúng một cách chính xác. Cho những chương trình mới, tuy nhiên, bạn nên sử dụng kiểu mới bởi vì đó là kiểu duy nhất của chương trình mà sự biên tập phù hợp với C++ tiêu chuẩn. Trong khi những chương trình kiểu cũ sẽ tiếp tục được hỗ trợ trong nhiều năm, thì chúng là kĩ thuật không tương hợp.

EXERCISE

1. *Before proceeding, try compiling the new-style skeleton program shown above. Although it does nothing , compiling it will tell you if your compiler supports the modern C++ syntax. If it does not accept the new-style headers or the namespace statement, substitute the old-style header as described.*

Remember, if your compiler does not accept new-style code, you must make this change for each program in this book.

Bài tập

1. Trước khi tiếp tục, thử biên tập chương trình điều khiển kiểu mới được cho thấy ở trên. Mặc dù nó không là gì, hãy biên tập nó sẽ cho bạn biết nếu trình biên dịch của bạn hỗ trợ cú pháp C++ hiện đại. Nếu nó không chấp nhận những đầu mục kiểu mới hay khai báo namespace, thay thế đầu mục kiểu xưa như đã được mô tả.

Nhớ, nếu trình biên dịch của bạn không chấp nhận mã kiểu mới, bạn phải làm sự thay đổi này cho mỗi chương trình trong sách này.

1.3. C++ CONSOLE I / O - BÀN GIAO TIẾP NHẬP/XUẤT C++

Since C++ is a superset of C, all elements of the C language are also contained in the C++ language. This implies that all C programs are also C++ programs by default.(Actually, there are some very minor exceptions to this rule, which are discussed later in this book.)

Vì C++ là siêu tập hợp của C nên mọi phần tử của ngôn ngữ C đều được chứa trong ngôn ngữ C++. Điều này chỉ ra rằng các chương trình C cũng là các chương trình C++. (Thực sự, có một vài ngoại lệ nhỏ đối với quy tắc sẽ được thảo luận vào cuối sách).

Therefore, it is possible to write C++ programs that look just like C programs. While there is nothing wrong with this per se, it dose mean that you will not be taking full advange of C++. To get the maximum benefit from C++, you must write C++-style programs. This means using a coding style and features that are unique to C++.

Do đó có thể viết các chương trình C++ trông giống như các chương trình C.Hầu hết những người lập trình C++ viết các chương trình sử dụng kiểu và các đặc điểm duy nhất cho C++. Lý do là để giúp bạn bắt đầu làm quen với các thuật ngữ C++ thay vì nghĩ về C. Cũng vậy, bằng cách dùng các đặc điểm C++, bạn sẽ để cho người khác hiểu ngay chương trình của bạn là C++ chứ không phải là chương trình C.

Perhaps the most common C++-specific feature used by C++ programmers is its approach to console I/O. While you may still use functions such as ***printf()*** *anh* ***scanf()****, C++ provides a new, and better, way to perform these types of I/O operations. In C++, I/O is performed using I/O operators instead of I/O function. The output operator is << anh the input operator is>>. As you know, in C, these are the left and right shift operators, respectively. In C++, they still retain their original meanings (left and right shift) but they also take on the expanded role of performing input and output. Consider this C++ statement:*

Có lẽ đặc điểm thông dụng nhất của C++ được những người lập trình C++ sử dụng là bàn giao tiếp nhập/xuất. Trong khi bạn vẫn sử dụng những hàm như **printf()** và **scanf(),** C++ đưa ra một cách mới và tốt hơn để thực hiện các thao tác nhập/xuất này. Trong C++, nhập/xuất được thực hiện bằng cách dùng các *toán tử nhập/xuất* thay vì các hàm nhập/xuất.Toán tử xuất là << và toán tử nhập là >>. Như bạn đã biết, trong C đây là những toán tử dịch chuyển phải và trái. Trong C++, chúng vẫn

còn ý nghĩa gốc (dịch chuyển phải và trái) nhưng chúng cũng có một vai trò mở rộng để thực hiện nhập xuất. Xét câu lệnh C++ sau:

```
cout << "This string is output to the screen.\n";
```

This statement causes the string to be displayed on the computer's screen. ***cout*** *is a predefined stream that is automatically linked to the console when a C++ program begins execution. It is similar to C's* ***stdout****. As in C, C++ console I/O may be redirected, but for the rest of this discussion, it is assumed that the console is being used.*

Câu lệnh này hiển thị một chuỗi lên màn hình máy tính. **cout** là một dòng (stream) được định nghĩa trước, tự động liên kết với bàn giao tiếp khi chương trình C++ bắt đầu chạy. Nó tương tự như **stdout** trong C. Như trong C, bàn giao tiếp Nhập/Xuất C++ có thể được định hướng lại, nhưng trong phần thảo luận này giả thiết là bàn giao tiếp vẫn được dùng.

By using the << output operator, it is possible to output any of C++'s basic types. For example, this statement outputs the value 100.99:

Bằng cách dùng toán tử xuất << có thể xuất bất kỳ loại cơ bản nào của C++. Ví dụ câu lệnh sau xuất giá trị 100,99:

```
cout << 100.99;
```

In general, to output to the console, use this form of the << operator:

cout << expression;

Here expression can be any valid C++ expression-including another output expression.

Nói chung, dùng dạng tổng quát của toán tử << để xuất:

```
cout << expression;
```

Ở đây, *expression* là một biểu thức C++ bất kỳ - kể cả biểu thức xuất khác.

To input a value from the keyboard, use the >> input operator. For example, this fragment inputs an integer value into ***num****:*

Để nhập một giá trị từ bàn phím, dùng toán tử nhập >>. Ví dụ, đoạn sau đây nhập một giá trị nguyên vào **num** :

```
int num;

cin >> num;
```

*Notice that num is not preceded by an **&**. As you know, when you use C's **scanf()** function to input values, variables must have their addresses passed to the function so they can receive the values entered by the user. This is not the case when you are using C++'s input operator.(The reason for this will become clear as you learn more about C++.)*

Chú ý rằng trước **num** không có dấu **&**. Như bạn đã biết, khi nhập một giá trị bằng các cách dùng hàm **scanf()** của C thì các biến có địa chỉ được truyền cho hàm vì thế chúng nhận được giá trị do người sử dụng nhập vào. Khi dùng toán tử nhập của C++ thì lại khác.(Khi học nhiều hơn về C++, bạn sẽ hiểu rõ lý do này).

In general, to input values from the keyboard, use this form of >>.

Tổng quát, để nhập một giá trị từ bàn phím thì dùng dạng >>:

```
cin >> variable;
```

***Note**: The expanded roles of << and >> are examples of operator overloading.*

Chú ý: Các vai trò mở rộng của << và >> là những ví dụ về quá tải toán tử.

*In order to use the C++ I/O operators, you must include the header **<iostream.h>** in your program. As explained earlier, this is one of C++'s standard headers and is supplied by your C++ compiler.*

Để sử dụng các toán tử Nhập/Xuất C++, bạn phải ghi rõ file tiêu đề **iostream.h** trong chương trình của bạn. Đây là một trong những file tiêu đề chuẩn của C++ do trình biên dịch C++ cung cấp.

CÁC VÍ DỤ (EXAMPLES):

1. *This program outputs a string, two integer values, and a double floating-point value:*

1. Chương trình sau xuất một chuỗi, hai giá trị nguyên và một giá trị dấu phẩy động kép:

```
#include <iostream>
```

```
using namespace std;

int main()

{

     int i, j;

     double d;

     i = 10;

   j = 20;

   d = 99.101;

   cout << "Here are some values:";

   cout << i;

   cout << ' ';

   cout < j;

   cout < ' ';

   cout < d;

   return 0;

}
```

The output of this program is shown here.

Chương trình xuất ra như sau.

```
Here are some values: 10 20 99.101
```

Remember: *If you working with an older compiler, it might not accept the new-style headers and the* **namespace** *statements used by this and other programs in this book. If this is the case, substitute the old-style code described in the preceding section.*

Ghi nhớ: *Nếu bạn l‡m việc với một trình biên dịch cũ, nó có thể không chấp nhận những đầu mục kiểu mới v‡ khai báo namespace được sử dụng bởi nó v‡ những chương trình trong cuốn sách n‡y. Nếu gặp trường hợp n‡y, thay thế bằng các mã kiểu cũ đã được mô tả trong phần*

trước.

2. It is possible to output more than one value in a single I/O expression. For example, this version of the program described in Example 1 shows a more efficient way to code the I/O statements:

2. Có thể xuất nhiều hơn một giá trị trong một biểu thức Nhập/Xuất đơn. Ví dụ, chương trình trong ví dụ 1 trình bày một cách có hiệu quả hơn để mã hóa câu lệnh Nhập/Xuất:

```
#include <iostream>

using namespace std;

int main()

{

    int i, j;

    double d;

    i = 10;

    j =20;

        d = 99.101;

        cout << "Here are some values:";

        cout << i << ' ' << j << ' ' << d;

        return 0;

}
```

Here the line

Ở đây, dòng

```
cout << i << ' ' << j << ' ' << d;
```

outputs several items in one expression. In general, you can use a single statement to output as many items as you like. If this seems confusing, simply remember that the << output operator behaves like any other C++ operator and

can be part of an arbitrarily long expression.

xuất nhiều mục trong một biểu thức. Tổng quát, bạn có thể sử dụng một câu lệnh đơn để xuất nhiều mục tùy bạn muốn. Nếu sợ nhầm lẫn thì chỉ cần nhớ rằng toán tử xuất << hoạt động như bất kỳ toán tử nào của C++ và có thể là một phần của một biểu thức dài bất kỳ.

Notice that you must explicitly include spaces between items when needed. If the spaces are left out, the data will run together when displayed on the screen.

Chú ý rằng bạn phải để những khoảng trống giữa các mục khi cần. Nếu có các khoảng trống thì dữ liệu cùng chạy khi được biểu thị trên màn hình.

3. This program prompts the user for an integer value:

3. Chương trình sau nhắc người sử dụng nhập vào một giá trị nguyên:

```
#include <iostream>]

using namespace std;

int main()

{

      int i;

      cout << "Enter a value:";

      cin >> i;

      cout << "Here's your number:" << i << "\n";

      return 0;

}
```

Here is a sample run:

Đây là mẫu được chạy:

```
Enter a value: 100

Here's your number: 100
```

*As you can see, the value entered by the user is put into **i**.*

Như bạn thấy, giá trị nhập vào bởi người sử dụng được đặt vào **i**.

4. The next program prompts the user for an integer value, a floating-point value, and a string. It then uses one input statement to read all three.

4. Chương trình tiếp theo nhắc người sử dụng nhập một giá trị nguyên, một giá trị dấu phẩy động và một chuỗi. Sau đó sử dụng lệnh nhập để đọc cả ba.

```
#include <iostream>

using namespace std;

int main()

{

    int i;

    float f;

    char s[80];

    cout << "Enter an integer, float, and string:";

    cin >> i >> f >> s;

    cout << "Here's your data:";

    cout << i << ' ' << f << ' ' << s;

    return 0;

}
```

As this example illustrates, you can input as many items as you like in one input statement. As in C, individual data items must be separated by whitespace characters (spaces, tabs, or newlines).

Trong ví dụ này, bạn có thể nhập nhiều mục tùy bạn muốn trong một câu lệnh nhập.

Như trong C, các mục dữ liệu riêng lẻ phải cách nhau một khoảng trắng (khoảng trống, tab hoặc dòng mới).

When a string is read, input will stop when the first whitespace character is encountered. For example, if you enter the following the preceding program:

Khi đọc một chuỗi, máy sẽ ngừng đọc khi gặp ký tự khoảng trắng đầu tiên. Ví dụ, nếu bạn nhập dữ liệu cho chương trình trước đây:

```
10 100.12 This is a test
```

the program will display this:

thì chương trình sẽ hiển thị như sau:

```
10 100.12 This
```

*The string is incomplete because the reading of the string stopped with the space after **This.** The remainder of the string is left in the input buffer, awaiting a subsequent input operation.(This is similar to inputting a string by using **scanf()** with the **%s** format.)*

Chuỗi không được đầy đủ do việc đọc chuỗi phải ngừng lại khi gặp khoảng trống sau **This.** Phần còn lại của chuỗi nằm trong bộ đệm, đợi cho đến khi có thao tác nhập tiếp theo. (Điều này tương tự với việc nhập chuỗi bằng hàm **scanf()** có định dạng **%) .**

5. *By default, when you use >>, all input is line buffered. This means that no information is passed to your C++ program until you press ENTER. (In C, the **scanf()** function is line buffered, so this style of input should not be new to you.) To see the effect of line-buffered input, try this program:*

5. Theo mặc định, khi bạn dùng >>, mọi dữ liệu nhập được đưa vào bộ đệm. Nghĩa là không có thông tin được truyền cho chương trình C++ của bạn cho đến khi bạn nhấn ENTER. (Hầu hết các trình biên dịch C cũng sử dụng dữ liệu nhập được đưa vào bộ đệm khi làm việc với hàm scanf(), vì thế dữ liệu nhập đưa vào bộ đệm không phải là điều mới lạ đối với bạn.) Để thấy rõ hiệu quả của dữ liệu được đưa vào bộ đệm, bạn thử chạy chương trình sau đây:

```
#include <iostream>

using namespace std;

int main()
```

```
{
      char ch;
      cout << "Enter keys, x to stop.\n";
      do
      {
            cout <<  ": ";
            cin >> ch;
      }
      while (ch != 'x');
            return 0;
}
```

When you test this program, you will have to press ENTER after each key you type in order for the corresponding character to be sent to the program.

Khi chạy chương trình này, bạn phải nhấn Enter sau mỗi lần bạn đánh một phím tương ứng với kí tự đưa vào chương trình.

BÀI TẬP(EXERCISES)

1. *Write a program that inputs the number of hours that an employee works and the employee's wage. Then display the employee's gross pay.(Be sure to prompt for input.)*

1. Viết chương trình nhập số giờ làm việc của một nhân viên và tiền lương của nhân viên. Sau đó hiển thị tổng tiền lương. (Phải chắc chắn có nhắc nhập dữ liệu.)

2. Write a program that converts feet to inches. Prompt the user for feet and display the equivalent number of inches. Have your program repeat this process until the user enters 0 for the number of feet.

2. Viết chương trình chuyển đổi feet thành inches. Nhắc người sử dụng nhập feet và

hiển thị giá trị tương đương của inches. Lặp lại quá trình cho đến khi người sử dụng nhập số 0 thay cho feet.

3. *Here is a C program. Rewrite it so it uses C++-style I/O statements.*

3. Đây là một chương trình C. Viết lại chương trình này bằng cách sử dụng các câu lệnh Nhập/Xuất theo kiểu C++.

```
/* Convert this C program into C++ style.

   This program computes the lowest common denominator.

*/

#include <stdio.h>

int main(void)

{

     int a, b, d, min;

     printf ("Enter two numbers:");

     scanf ("%d%d", &a, &b);

     min = a > b ? b: a;

     for (d = 2; d < min; d++)

          if (((a%d)==0)&&((b%d)==0)) break;

     if (d==min)

     {

          printf ("No common denominators\n");

          return 0;

     }

     printf ("The lowest common denominator is %d\n",d);

     return 0;

}
```

1.4. C++ COMMENTS – LỜI CHÚ GIẢI TRONG C++

In C++, you can include comments in your program two different ways. First, you can use the standard, C-like comment mechanism. That is, begin a comment with / and end it with */. As with C, this type of comment cannot be nested in C++.*

Trong C++, bạn có thể đưa vào lời chú giải trong các chương trình theo hai cách. Thứ nhất, bạn có thể dùng cơ chế chuẩn về lời chú giải như trong C. Nghĩa là lời chú giải bắt đầu bằng /* và kết thúc bằng */. Như đã biết trong C, loại lời chú giải này không thể lồng nhau trong C++.

The second way that you can add a remark to your C++ program is to use the single-line comment. A single-line comment begins with a // and stops at the end of the line. Other than the physical end of the line (that is, a carriage-return/ linefeed combination), a single-line comment uses no comment terminator symbol.

Cách thứ hai để đưa những lời chú giải vào chương trình C++ là sử dụng *lời chú giải đơn dòng*. Lời chú giải đơn dòng bắt đầu bằng // và kết thúc ở cuối dòng. Khác với kết thúc vật lý của một dòng (như tổ hợp quay trở về đầu dòng đẩy dòng), lời chú giải đơn dòng không sử dụng ký hiệu chấm dứt dòng.

Typically, C++ programmers use C-like comments for multiline commentaries and reserve C++-style single-line comments for short remarks.

Nói chung, người lập trình C++ dùng lời chú giải giống C cho những lời chú giải đa dòng và dùng lời chú giải theo kiểu C++ khi lời chú giải là đơn dòng.

CÁC VÍ DỤ (EXAMPLES):

1. *Here is a program that contains both C and C++-style comments:*

1. Đây là chương trình có các lời chú giải theo kiểu C và C++.

```
/*
     This is a C-like comment.
     This program determines whether an integer is odd
```

```
or even.
*/
#include <iostream>
using namespace std;
int main()
{
     int num; // this is a C++, single-line comment
     // read the number
     cout << "Enter number to be tested:";
     cin >> num;
     // see if even or odd
     if ((num%2)==0)
 cout << "Number is even\n";
     else
          cout << "Number is odd\n";
     return 0;
}
```

2. While multiline comments cannot be nested, it is possible to nest a single-line comment within a multiline comment. For example, this is perfectly valid:

2. Trong khi các lời chú giải theo kiểu C không thể lồng nhau thì lời chú giải đơn dòng C++ có thể lồng vào trong lời chú giải đa dòng theo kiểu C. Ví dụ sau đây là hoàn toàn đúng.

```
/* This is a multiline comment
    inside of which // is need a single-line comment.
    Here is the end of the multiline comment.
*/
```

The fact that single-line comments can be nested within multiline comments makes it easier for you to "comment out" several lines of code for debugging purposes.

Sự việc các lời chú giải đơn dòng có thể được lồng trong các lời chú giải đa dòng dễ dàng làm cho bạn trình bày lời chú giải nhiều dòng mã nhằm mục đích gỡ rối.

BÀI TẬP (EXERCISES):

1. *As an experiment, determine whether this comment (which nests a C-like comment within a C++-style, single-line comment) is valid:*

1. Hãy xác định xem lời chú giải sau đây có hợp lệ không:

```
// This is a strange /* way to do a comment */
```

2. *On your own, add comments to the answers to the exercises in Section 1.3.*

2. Hãy thêm những lời chú giải theo cách riêng của bạn cho phần trả lời trong phần bài tập 1.3.

1.5. CLASSSES: A FIRST LOOK - LỚP : CÁI NHÌN ĐẦU TIÊN

Perhaps the single most important feature of C++ is the class. The class is the merchanism that is used to create objects. As such, the class is at the heart of many C++ freartures. Although th subject of the classes is covered in great detail throughout this book, classes are so fundamental to C++ programming that a brief overview is necessary here.

Có lẽ đặc điểm quan trọng của C++ là lớp. Lớp là cơ cấu được dùng để tạo đối tượng. Như thế lớp là trung tâm của nhiều đặc điểm của C++. Mặc dầu chủ đề tài của lớp được bao phủ chi tiết khắp cả sách này, lớp là đối tượng cơ bản khai báo một chương trình C++ , ngắn gọn tổng quan là sự cần thiết.

A class is declared using the class keywork. The syntax of a class declaration is similar to that of structure. Its general from is shown here:

Một lớp được khai báo sử dụng từ khóa lớp. Cú pháp khai báo của một lớp là cấu trúc. Dạng tổng quát như sau :

```
class  class-name

{

//private functions and variables

public:

// private functions and variables

 };
```

In a class declaration, the object-list is optional. As with a structure, you can declared class object later, as needed. While the class-name is also technically optional, from a practical point of view it is virtually always needed. The reason for this is that the class-name becomes a new type name that is used to decrared object of the class.

Trong khai báo lớp, danh sách đối tượng là tùy chọn. Giống như cấu trúc, bạn có thể khai báo các đối tượng sau này khi cần. Tên lớp là tùy chọn có tính kỹ thuật, từ quan điểm thực hành thực sự nó luôn luôn cần đến. Lý do la tên lớp trở thành một kiểu mới được dùng khai báo các đối tượng của lớp

Function and variables declared inside a class declaration are said to be members of that class. By ddefault, all function and variables declared insinde a class are private to that class. This meas that they are accessible only by other members of that class. To declare public class members, the public keywork is used, followed by a colon. All function and varaibles declared after the public specifter are accessible both by other members of the class and by any other part of the program that contains the class.

Các hàm và biến khai báo bên trong khai báo lớp các thành viên của lớp đó. Theo mặc định, tất cả các hàm và khai báo bên trong của lớp là thuộc bên trong của lớp đó. Để khai báo thành viên chung của lớp, người ta dùng từ khóa **public** ,theo sau hai dấu chấm.Tất cả các hàm và biến được khai báo sau từ khóa **public** được truy cập bởi thành viên khác của chương trình của lớp đó.

Here is a simple class declaration

Đây là ví dụ khai báo lớp:

```
class myclass

{

//private to myclass

int a;

public :

     void set_a(int num);

     int get_a();

};
```

*This class has one private variables called **a,** and two public functions, **set_a()** and **get_a().** Noties are declared within a class using their prototype froms, that are declared to be part of a class are called members functions.*

Lớp này có một biến riêng là **a,** và hai hàm chung là **set_a()** và **get_a()**. Chú ý rằng các hàm được khai báo trong một lớp sử dụng dạng nguyên mẫu của nó.Các hàm được khai báo là một phần của lớp được gọi là các hàm thành viên.

*Since **a** is private, it is not accessible by any code outside **myclass.** However, since **set_a()** and **get_a()** are members of **myclass**, they can access a. Further, **get_a()** and **get_a()** are declared as public program that contains **myclass***

Vì một biến riêng, nó không thể được truy cập bất kì mã nào ngoài myclass. Tuy nhiên, vì **set_a()** và **get_a()** là các thành viên của **myclass** nên có thể truy cập được a. Hơn nữa, **get_a()** và **set_a()** được khai báo là các hàm thành viên chung của **myclass** và có thể được gọi bất kì phần nào của chương trình có chứa **myclass**.

*Although the function **get_a()** and **set_a()** are declared by **myclass**, they are not yet defined. To defined a members function, you must link the type name of the class with the class name followed by two colons. The two colons are called the scope reslution operator. For expamle, here is the way the members function **set_a()** and **get_a()** are defined :*

Mặc dù các hàm **get_a()** và **set_a()** được khai báo bởi myclass, chúng vẫn chưa được định nghĩa. Để định nghĩa một hàm thành viên, bạn phải liên kết tên kiểu của lớp của tên hàm. Thực hiên điều này cách đăt trước tên hàm của lớp tiếp theo là hai dấu chấm. Hai dấu chấm được gọi là toán tử giải phạm vi. Ví dụ, đây là định nghĩa các hàm thành viên **set_a()** và **get_a()** :

```
void myclass ::set_a(int num)

{

     a=num;

}

 int myclass :: get_-a()

{

     return a;

}
```

*Notice that both **set_a()** and **get_a()** have access to **a,** which is private to **myclass**. Because **set_a()** and **get_a()** are members of **myclas**s, they can direclly access is private data.*

Chú ý rằng cả hai hàm set_a() và get_a() có thể truy cập a là biến riêng của myclaas. Vì get_a() và set_a() là các thành viên của myclass nên chúng có thể truy cập trực tiếp các dữ liệu bên trong lớp.

In general, to define a members function you must this from :

Nói chung, khi bạn định nghĩa một hàm thành viên thì dùng:

```
ret-type class-name::func-name(parameter-list)

{

//body of function
}
```

Here class-name is the name of the class to which the function belongs.

Tên ở đây lớp là lớp mà tới đó chức năng thuộc về lớp

*The declaration of **myclass** did not define any object of type **myclass**- it only defines the type of object that will be created when one is actually declared. To create an object, use the class name as a type specifier. Expamle, this line declares two object of type **myclass**:*

Khai báo của lớp không định nghĩa đối tượng nào có kiểu **myclass**- nó chỉ định nghĩa kiểu đối tượng sẽ được tạo ra khi được khai báo thực sự. Để tao ra một đối tượng, hãy sử dụng tên lớp như bộ chỉ định kiểu. Ví dụ, sau đây là khai báo hai đối tượng có kiểu **myclass**:

```
myclass ob1,ob2 // these are object of type myclas
```

Cần nhớ (Remember):

A clas declaration is a logical abstraction that defines a new type. It determines what an object of that type will look like. An object declaration creates a physical antity of that type. That is, an object occupies memory space, but a type definition does not.

Khai báo lớp là một sự trừu tượng logic để định nghĩa một kiểu mới xác định đối tượng có kiểu đó sẽ trông như thế nào.Một khai báo đối tượng tạo ra một thực thể vật lí có kiểu đó..Nghĩa là một đối tượng chiếm không gian bộ nhớ,còn định nghĩa kiểu thì không.

Once an object of a class has been created, your program can reference its public members by using the dot (period) operator in much the same way that structure members are accessed. Assuming the preceding object declaration, the following statement calls set_a() for object ob1 and ob2.

Khi đối tượng tạo ra một lớp, chương trình của bạn có thể có tham chiếu của một lớp tạo ra các thành viên chung của đối tượng bằng những toán tử điểm.Giả sử khai báo đối tượng trước đây, câu lệnh sau gọi hàm cho các đối tượng **ob1** và **ob2**

```
ob1.set_a(10) //sets ob1's version of a to 10
ob2.set_a(99)       // sets ob2's version of a to 99
```

*As the coments indicate, these statements set **ob1's** copy of **a** to 10 and **ob2's** copy to 99. Each object contains its own copy of all data declared within the class. This means that that **ob1's a** is distinct and different from the **a** linked to **ob2***

Như chỉ rõ trong lời chú giải, các lệnh này đặt ra bản sao cua a của ob1 bằng 10 và bản sao của a của ob2 bằng 99. Mỗi đối tượng có bản sao riêng của nó về mọi dữ

liệu được khai báo trong lớp.Nghĩa là a trong ob2**.**

Cần nhớ (Remember):

Each object of a class has its own copy of every variable declared within the class.

Mỗi đối tượng của một lớp có bản sao riêng của các biến được khia báo trong lớp.

Các ví dụ (Expamles):

14.	*As a simple first example, this program demonstrates **myclass,** described in the text. It sets the value of **a** for **ob1** and **ob2** and then display a's value for each object.*

1. Như là một ví dụ đơn giản đầu tiên, chượng trình sau đây dùng **myclass** được khai báo trong văn bản. Để dặt giá trị vủa **a** cho **ob1** và **ob2** và hiển thị giá trị **a** cho mỗi đối tượng :

```
#include "stdafx.h"
#include "iostream.h"
Using namespace std;
class myclass
{
   //private to myclass
   int a;
   public :
        void set_a(int num);
        int get_a();
};

void myclass::set_a(int num)
{
   a=num;
}

int myclass::get_a()
{
   return a;
}

int main()
{
   myclass ob1,ob2;
   ob1.set_a(10);
   ob2.set_a(99);
```

```
   cout<<ob1.get_a()<<"\n";
   cout<<ob2.get_a()<<"\n";
   return 0;
}
```

As you should epect, this program display the values 10 and 99 on the sreen.
Chương trình xuất ra màn hình giá trị 10 và 99 ra màn hình.

2. *In **myclass** from the preceding expamle, a is private. This means that only member function of myclass can access it directly. (This is one reason why the public function **get_a()** is required.) If you try to access a private member of a class from some part of of your program that is not a member of that class, a compile-time error will result. For example, assuming that **myclass** is defined as shown in the preceding example, the following **main()** function will cause an error :*

14. Trong **myclass** ở ví dụ trên, a là biến riêng.Chỉ các hàm thành viên của myclass mới có thể truy cập trực tiếp. (Đây là lí do ví sao phải có hàm **get_a()**)Nếu bạn truy cập một thành viên riêng lẻcủa lớp từ một bộ phận nào đó của chương trình không phải là thành viên của lớp đó, thí sẽ đến lỗi biên dịch. Ví dụm giả sử rằng **myclass** được định nghĩa như trong ví dụ trước đây, hàm main() sau đây sẽ tạo ra lỗi :

```
//This fragment contains an error:

int main()
{
     myclass ob1,ob2;
     ob1.a=10;      // ERROR ! cannot access private
member
     ob2.a=99;      // by non-member functions.
     cout<<ob1.get_a()<<"\n";
     cout<<ob2.get_a()<<"\n";
     return 0;
}
```

*3. Just as there can be public member functions, there can be public member variables as well. For example, if a were declared in the public section of **myclass**, **a** could be referenced by any part of the program, as shown here:*

3. Khi có các hàm thành viên chung, cũng có các biến thành viên chung. Ví dụ, nếu **a** được khai báo la biến chung của **myclass** thì **a** cũng có thể được tham chiếu, như sau đây:

```
#include "stdafx.h"
```

```
#include "iostream.h"

class myclass
{
     //now a is public
     public :
          int a;
     //and there is no need for  set_a() or get_a()
};

int main()
{
     myclass ob1,ob2;
     ob1.a=10;
     ob2.a=99;
     cout<<ob1.a<<"\n";
     cout<<ob2.a<<"\n";
     return 0;
}
```

In this example, since a is declared as a public member of myclass, it is directly accessible from main(). Notice how the dot operator is used to access a. In general, when you are calling a member function or accessing a member variable from outside its class, the object's name following by the dot operator followed by the member you are referring to.

Trong ví dụ này, vì a được khai báo là thanh viên chung của myclass, nó có thể được truy cập trực tiếp từ hàm main(). Chú ý cách sử dụng toán tử điểm để truy cập a.Nói chung, dù bạn gọi hàm thành viên hoặc truy cập biến thành viên, tên đối tượng phải được kèm toán tử điểm, tiếp theo là tên thành viên để chỉ rõ thành viên nào của đối tượng nào bạn cần tham chiếu.

4. *To get a staste of the power of object, let's look at a more practical example. This program creates a class called stack that implements a stack that can be used to store characters :*

4. Để biết rõ năng lực của các đối tượng, háy xét một ví dụ thực tế hơn:Chương trình này tạo ra lớp stack để thi hành một ngăn xếp dùng để lưu trữ ký tự :

```
#include "stdafx.h"
#include "iostream.h"
Using namespace std;
#define SIZE 10
// Declare a stack class for characters
```

```
class stack
{
     char stck[SIZE];  // holds the stack
     int tos ;      // index of top of stack
     public:
          void init();        //intialize stack
          void  push(char  ch);     //push  charater  on
stack
          char pos();     //pop character from stack
};
//Initialize the stack
void stack::init()
{
     tos=0;
}
//Push a charater.
void stack::push(char ch)
{
     if(tos==SIZE)
     {
          cout<<"Day stack";
          return ;
     }
     stck[tos]=ch;
     tos++;
}
//Pop a charater
char stack::pos()
{
     if(tos==0)
     {
          cout<<"Stack is empty ";
          return 0;   //return null on empty stack
     }
     tos--;
     return stck[tos];
}

int main()
{
     stack s1,s2;    //create two stacks
     int i;
     // Initialize the stacks
     s1.init();
     s2.init();

     s1.push('a');
```

```
        s2.push('x');
        s1.push('b');
        s2.push('y');
        s1.push('c');
        s2.push('z');
        for(i=0;i<3;i++)
             cout<<"Pos s1: "<<s1.pos()<<"\n";
        for(i=0;i<3;i++)
             cout<<"Pos s2 : "<< s2.pos()<<"\n";
        return 0;
}
```

This program display the following output:

Kết quả xuất ra màn hình là:

```
pos s1: c
pos s1: b
pos s1: a
pos s2: z
pos s2: x
pos s2: y
```

*Let's take a close look at this program now. The class **stack** contains two private**:** **stack** and **tos**. The array **stck** actually holds the charaters pushed onto the stack, and **tos** contans the index to the top of the stack. The publicc stack function are **init()**, **push()**, and **pop()**, which initialize the stack, push a value, and pop a value, respectively.*

Bây giờ xét chương trình kỹ hơn. Lớp **stack** có hai biến riêng, **stck** và **tos**. Mảng **stck** chứa các ký tự đua vào trong ngăn xếp, và tos chứa các chỉ số trên đỉnh ngăn xếp. Các hàm chung của **init()**, **push()** và **pos()** lần lượt để khỏi động ngăn xếp, đẩy một giá trị vào và kéo một giá trị ra.

*Insider **main(),** two stacks, **s1** and **s2**, are created, and three charaters are pushed onto each stack. It is important to understand that each object is separate from the other. That is, the charater pushed onto **s1** in no way affect the charaters pushed onto **s2**. Each object contians its own copy of stck and tos. This concept is fundamental to undertanding objects. Although all object creates and maintians its own data.*

Trong hàm **main()** hai ngăn xếp **s1** và **s2** được tọa ra và 3 ký tự được đẩy vào mỗi ngăn xếp. Điều quan trọng là hiểu rằng mỗi đối tượng ngăn xếp là cách biệt nhau. Nghĩa là các ký tự đẩy vào trên **s1** không ảnh hưởng đến các ký tự được đẩy vào **s**.

Mỗi đối tượng có một bản sao **stck** và **tos** riêng. Khái niện này là cơ bản để hiểu các đối tượng. Mặc dầu tất cả các đối tượng của một lớp sử dụng chung cho các hàm thành viên, mỗi đối tượng tạo và giữ dữ liệu riêng.

BÀI TẬP (Exrcises):

If you have not done so, enter and run the programs shown in the examples for this section.

1. Nếu bạn không tin, hãy nhập và chạy chương trình trong các phần này.

*Create a class **card** that maintians a library card catalog entry. Have the class store a book's title, author, and number of copies on hand. Store the title and author as string and the munber on hand as an integer. Use a public member function called **store**() to store a book's information and a public member function called **show()** to display the information. Include a short main() function to demonstrate the class.*

2. Tọa lớp **card** để giữ mục nhập catalog của thẻ thư viện. Có lớp chứa tên sách , tác giả, số bản. Lưu trữ tựa đề và tác giả dưới một chuỗivà số bản như một số nguyên.Dùng hàm thành viên chung **store()** để lưu trữ thông tin về sách và hàm thanh viên chung **show( )** để hiển thị thông tin.Có hàm main() để ghi rõ lớp.

*Create a queue class that maintians a circular queue of integers. Make the queue size 100 integers long. Include a short **main()** function that demonstrates its operation.*

3. Tạo một lớp hàm queue để giữ các hàm số nguyên. Tạo một kích thước hang dài 100 số nguyên. Có hàm **main()** chỉ rõ phép toán của nó.

1.6. SOME DIFFERENCE BETWEENCE AND C++ - MỘT SỐ KHÁC BIỆT GIỮA C VÀ C++

Although C++ is a superset of C, there are some small difference between the two, and a few are woth knowing from the star. Before proceeding, let's take time to example them.

Mặc dù C++ là một siêu tập hợp của C, vẫn có một vài điểm khác biệt giữa C và

C++ mà chắn sẽ có ảnh hưởng đến khi bạn viết chương trình C++. Dù mỗ điểm khác nhau này là nhỏ, chúng cũng có trong các chương trình C++. Do đó cần phải xét qua những điểm khác nhau này.

First, in C, when a function takes no paramentes, its prototype has the work void inside its function parameter list. For example, in C, if a function called f1() takes no parameters (and return a char), its prototype will look like this:

Trước hết, trong C, khi một hàm không có tham số, nguyên mẫu của nó có chữ void bên trong danh sách tham số của hàm. Ví dụ, trong C, nếu hàm được gọi là f1() không có tham số thì nguyên mẫu của nó có dạng :

```
char f1(void);
```

However, in C++, the void is optional. Therefore, in C++, the prototype for f1() is usually written like this:

Tuy nhiên trong C++, void là tùy ý. Do đó, trong C++, nguyên mẫu cho **f1()** thường được viết như sau :

```
char f1();
```

*C++ differs from C in the way that an empty parameter list is specifed. If the preceding prototype had occurred in a C program, it would simply means that nothing is said about the parameters. This is the reason that the preceding example did not explicitly use **void** to declare an empty parameter list.(The use of **void** to declare an empty parameter list is not illegal; it is just reduntant. Since most C++ programmers pursue effciency with a nearly religious zeal, you will almost never see **void** used in this way.) Remember, in C++, these two declarations are equivalent.*

C++ khác với C ở chỗ danh sách tham số rỗng được chỉ rõ. Nếu nguyên mẫu trên đây xảy ra trong một chương trình C thì không có gì đề cập đến các tham số của hàm. Trong C++, điều đó có nghĩa là hàm không có tham số. Đây là lí do vì sao nhiều ví dụ trước đây không sử dụng **void** để khai báo trong danh sách các tham số rỗng. (Dùng void để khai báo danh sách các tham số rỗng là không hợp lệ, vì như vậy là thừa. Do hầu hết người lập trình C++ rất cẩn thận như không bao giờ thấy **void** trong trường hợp này). Cần nhớ, trong C++, hai loại khai báo này là tương đương:

```
int f1();
int f1(void);
```

Another subtle differencebetween C and C++ is that in a C++ program, all function must be prototype. Remmember, in C, prototypes are recommended but technically optional. In C++, they are required. As the examples from the previous section show, a member function's prototype contianed in a class also serves as its general prototype, and no othee separate prototupe is required.

Một sự khác biệt rất ít giữa C và C++ là chổ trong một chương trình C++, tất cả các hàm phải có dạng nguyên mẫu, còn trong C để ở dạng nguyên mẫu, nhưng chỉ là tùy chọn. Trong C++ các hàm được đòi hỏi như thế. Các vì dụ trong phần trước chĩ rõ, dạng nguyên mẫu của hàm thành viên ở trong một lớp cũng dùng ở dạng nguyên mẫu của nó và không có nguyên mẫu nào được đòi hỏi.

A third difference between C and C++ is that in C++, if a function is declared as returning a value, it must return a value. That is, if a funciton has return type other than void, any return statement within that function must contian a value. In C, a non- void function is not required to actually return a value. If it doesn't , a garbage value is "returned"

Điểm khác biệt giữa C và C++ là ở điểm khai báo các biến cục bộ. Trong C, các biến cụ bộ chỉ khai báo ở đầu mỗi khối, trước bất kỳ câu lệnh tác động nào. Trong C++, các biên cụ bộ được khai báo bất kỳ chỗ nào.Lợi điểm của cách này là các biến cục bộ có thể được khai báo gần hơn chúng được dùng, điều này giúp tránh được những kết quả ngoài ý muốn.

In C, if you don't explicitly specify the return of a function, an integer return type is assumed. C++ has dropped the " default-to-int" rule. Thus, you must explicitly declare the return type of all functions.

Trong C, nếu bạn không chỉ định rõ sự trở về của một hàm, trả về một số nguyên không có thật.

One other difference between C and C++ that you will commonly encounter in C++ program has to do with where local variables can be declared. In C, local variables can be declared only at the start of a block, prior to any " action " statements. In C++, local variables can be declared anywhere. One advantage of this approach is that local variables can be declared close to where they are first used, thus helping to prevent unwanted side effects.

Một trong những sự khác biệt giữa C và C++ chỉ là bạn có thể bắt gặp phổ biến trong C++ chương trình phải làm việc với các biến số logic có thể là công phai. Trong C, các giá trị có thể công khai trong một khối chương trình,trong câu lệnh hành động. Trong C++, các gái trị có thể công khai bất kì chổ nào. Một sự thuận lợi có thể là phổ biến khi đóng nó và dùng ở bất kì chổ nào, vì thế không làm ảnh hưởng đến các giá trị bên cạnh.

*Finally, C++ define the **bool** data type, which is used to store. Boolean (i.e., true/flase) values. C++ also defines the keywords **true** and **false**, which are the only values that a value of type bool can have. In C++, the outcome of the relational and logical operator is a value of type **bool**,and all conditional statements must evaluate to a **bool** value. Although this might at first seem to be big change from C, it isn't. In fact, it is virtually transparent. Here's why: As you know,in C**, true** is any nonzero value and **fales** is 0. This still holds in C++ because any nonzero value is automatically converted into **true** and any 0 value is uatomatically converted in to **false** when used a boolean expression. The reverse also occurs: true is converted to 1 and false is converted to 0 when a bool values is used in an integer expression. The addition of **bool** allows more thorough type checking and gives you a way to difference between boolean and integer types. Of course, its use is optional; bool is mostly a convenience.*

Kết luận lại, C++ định nghĩa kiểu dữ liệu bool, chúng sẽ cung cấp được dung. Bool trả về giá trị true /flase. C++ đã định nghĩa sẵn giá trị treu flase,sẽ c trả về giá trị khi khai báo kiểu dữ liệu bool. Trong C++, liên hệ của kết quả và toán tử logic của cho giá trị măc định, và tất cả các khai bóa bắt buộc trong kiểu dữ liệu bool. Mặc dù nó là sự thay đổi lớn đối với C, không phải vậy. Sự thật, nó cúng giống như C. Tại vì: nếu bạn hiểu trong C , **true** là khác **0** và **false** là 0. Cũng như vậy trong C++, nếu giá trị khác 0 thì nó tự động cập nhật là **true** còn nếu giá trị là 0 thì nó tự động cập nhật là **false** khi trong biểu thức boolean. Xuất hiện sự đảo ngược : true là 1 còn false là 0 khi trong khai báo kiểu dữ liệu **bool**. Phép cộng trong bool cho nó thực hiên nhiều kiểu kiểm tra và, đêm lại cho bạn sự khác biệt với **boolean** và kiểu số nguyên. Thật vậy, nó được dùng tùy chọn; **bool** thường thuận lợi hơn.

block, prior to any "action" statements. In C++, local variables can be declared anywhere. One advantage of this approach is that local variables can be declared close to where they are first used, thus helping to prevent unwanted side effects.

khối, trước lời gọi hàm. Trong C++, những biến cục bộ có thể được khai báo ở bất cứ đâu. Một trong những lợi thế của cách tiếp cận này là những biến cục bộ có thể được khai báo gần nhất nơi mà chúng được dùng lần đâu, điều này giúp tránh khỏi những tác động phụ không mong muốn.

*Finally, C++ defines the **bool** data type, which is used to store Boolean (i.e, true/false) values. C++ also defines the keywords **true** and **false**, which are the only values of type **bool** can have. In C++, the outcome of the relational and logical operators is a value of type **bool**, and all conditional statements must evaluate to a **bool** value. Although this might at first seem to be a big change from C, it isn't. In fact, it is virtually transparent. Here's why: As you know, in C, **true** is any nonzero value and **false** is 0. This still holds in C++ because any*

*nonzero value is automatically converted into **true** and any 0 value is automatically converted into **false** when used in a Boolean expression. The reverse also occurs: **true** is converted to 1 expression. The addition of **bool** allows more thorough type checking and gives you a way to differentiate between Boolean and integer types. Of course, its use is optional; **bool** is mostly a vonvenience.*

Cuối cùng, C++ định nghĩa kiểu dữ liệu cờ **bool**, được dùng để lưu giá trị cờ (đún/sai). C++ còn định nghĩa những từ khóa **true** và **false**, cái mà chỉ có thể chứa một giá trị **cờ** duy nhất. Trong C++, những toán tử quan hệ và toán tử logic phải trả về giá trị cờ. Mặc dù điều này có vẻ như là một sự thay đối lớn từ C, nhưng không phải vậy. Thực ra, cũng dễ hiểu. Đây là lý do tại sao: Như bạn biết, trong C, **true** là một giá trị khác 0 và **false** là 0. Điều này vẫn còn trong C++ vì bất kỳ giá trị khác 0 nào cũng tự động được chuyển thành **true** và bất kỳ giá trị 0 nào cũng đều tự động được chuyển thành **false** khi một biểu thức so sánh được gọi. Ngược lại **true** được chuyển thành 1. Việc thêm **cờ hiệu** cũng cho phép sự kiểm tra kỹ lưỡng hơn và đưa ra cho bạn một cách khác giữa phép toán Bool và số nguyên. Dĩ nhiên, với sự chọn lựa này, **cờ hiệu** là thuận tiện nhất.

VÍ DỤ (EXAMPLES):

*In a C program, it is common practice to declare **main()** as shown here if it takes no command-line arguments:*

Trong một chương trình C, thông thường khai báo hàm **main()** như sau nếu như không có đối dòng lênh:

```
int main(void)
```

*However, in C++, the use of **void** is redundant and unnecessary.*

Tuy nhiên, trong C++, việc dùng **void** là thừa và không cần thiết.

*This short C++ program will not compile because the function **sum()** is not prototyped:*

Chương trình C++ ngắn sau sẽ không biên dịch vì hàm **sum()** không ở dạng nguyên mẫu.

```
// This program will not compile
```

```
#include <iostream>
using namespace std;

int main()
{
   int a,b,c;
   cout<< "Enter two numbers: ";
   cin>>a>>b;
   c = sum(a,b);
   cout<<" Sum is: "<<c;

   return 0;
}
// This function needs a prototype.
sum(int a,int b)
{
   return a+b;
}
```

Here is a short program that illustrates how local variables can be declared anywhere within a block:

3. Đây là một chương trình ngắn minh họa các biến cục bộ được khai báo bất kì chỗ nào trong khối:

```
#include <iostream>
using namespace std;
int main()
{
```

```
        int i; // local var declared at start of block


        cout<<" Enter number: ";
        cin>> i;


        // copute factorial
        int j,
        fact=1; // vars declared after action statements
        for(j=i;j>=1;j--) fact = fact * j;
        cout<<" Factorial is: "<<fact;
        return 0;
}
```

*The declaration of **j** and **fact** near the point of first use is of little value in this short example: however, in large functions, the ability to declare variables close to the point of their first use can help clarify your code and prevent unintentionla side effects.*

Khai báo **j** và **fact** gần với điểm sử dụng đầu tiên có giá trị nhỏ trong ví dụ ngắn này, tuy nhiên, trong một hàm lớn có thể khai báo biến gần điểm sử dụng nó đầu tiên, giúp làm rõ ràng mã và ngăn ngừa được những hệ quả ngòai ý muốn. Đặc điểm này của C++ thường được dùng trong các chương trình C++ khi có các hàm lớn.

*The following program creates a Boolean variable called **outcom** and assigns it the value **false**. It then uses this variable.*

4. Chương trình sau tạo ra biến Boolean gọi là **outcom** và đặt giá trị cờ là **false**. Sau này nó được dùng như là một biến.

```
#include <iostream>

using namespace std;


int main()
```

```
{
   bool outcome;


   outcome = false;
   if(outcome) cout<<" true";
   else cout<<" false" ;
   return 0;
}
```

*As you should expect, the program displays **false.***

Như bạn thấy, chương trình hiện **false**

BÀI TẬP (EXERCISES):

1. The following program will not compile as a C++ program. Why not?

1. Chương trình sau sẽ không biên dịch như một chương trình C++. Tại sao?

```
// This program has an error.
#include <iostream>
using namespace std;
int main()
{
   f();
   return 0;
}
void f()
```

```
{
   cout<<" this won't work ";
}
```

2. On your own, try declaring local variables at various points in a C++ program. Try the same in a C program, paying attention to which declarations generate errors.

2. Bạn thử khai báo các biến cục bộ tại các điểm khác nhau trong một chương trình C++. Thử làm như thế trong một chương trình C, chú ý khai báo này tạo ra lỗi.

1.7. INTRODUCING FUNCTION OVERLOADING - DẪN NHẬP SỰ NẠP CHỒNG HÀM:

After classes, perhaps the next most important and pervasie C++ feature is function overloading. Not only does function overloading provide the mechanism by which C++ achieves one type of polymorphism, it also forms the basis by which the C++ programming evironment can be dynamically extended. Because of the importance of overloading, a brief introduction is given here.

Sau lớp, có lẽ đặc điểm quan trọng tiếp theo của C++ là sự nạp chồng hàm(quá tải hàm). Không chỉ sự nạp chồng hàm cung cấp cơ chế mà nhờ đó C++ có được tính đa dạng, mà hàm này còn tạo nên cơ sở để mở rộng môi trường lập trình C++. Do tính quan trọng của nạp chồng hàm nên ở đây trình bày ngắn gọn về hàm.

In C++, two or more functions can share the same name as long as either the type of their arguments differs or the number of their arguments differs-or both. When two or more functions share the same name, they are said to be overloaded. Oveerloaded functions can operations to be referred to by the same name.

Trong C++, hai hoặc nhiều hàm có thể dùng chung một tên nhưng có loại đối số hoặc số đối khác nhau – hoặc cả hai khác nhau. Khi hai hay nhiều hàm có cùng tên, chúng được gọi là *quá tải* (hay nạp chồng). Quá tải hàm giúp giảm bớt tính phức tạp của chương trình bằng cách cho phép các toán có liên hệ tham chiếu cùng một tên.

It is very easy to overload a function: simply declare and define all required versions. The compiler will automatically select the correct version based upon the number and/or type of the arguments used to call the function.

Dễ dàng để thực hiện quá tải một hàm: chỉ cần khai báo và định nghĩa các phiên bản cần có của nó. Trình biên dịch sẽ tự động chọn phiên bản đúng để gọi dựa vào số và/ hoặc kiểu đối số được dùng gọi hàm

Note: *It is also possible in C++ to overload operators. However, before you can fully understand operator overloading, you will need to know more about C++.*

Chú ý:

Trong C++ cũng có thể thực hiện quá tải các toán tử. Tuy nhiên, trước khi bạn hiểu đầy đủ về quá tải toán tử, bạn cần biết nhiều hơn về C++.

CÁC VÍ DỤ (EXAMPLES):

One of the main uses for function overloading is to achieve compile-time polymorphism, which embodies the philosophy of one interface, many methods. As you know, in C programming, it is common to have a number of related functions that deffer only by the type of data on which they operate. The classic example of this situation is found in the C standard library. As mentioned earlier in this chapter, the library contains the functions ***abs(),labs()*** *and* ***fabs()****. Which return the absolute value of an integer, a long integer, and a floating-point value, respectively.*

Một trong những cách sử dụng chính đối với sự quá tải hàm là đạt được tính đa dạng thời gian biên dịch, gồm nhiều phương pháp. Như bạn, biết trong lập trình C thường có một số hàm có liên quan với nhau, chỉ khác nhau kiểu dữ liệu mà các hàm hoạt động. Ví dụ cổ điển về trường hợp này được tìm thấy trong thư viện chuẩn của C. Như đã trình bày trong phần đầu chương này, thư viện chứa các hàm **abs()**, **labs()**,**fabs()**, các hàm này lần lượt trả về giá trị tuyệt đối của một số nguyên, một số nguyên dài và một giá trị dấu phẩy động.

However, because three different names are neede due to the three different data types, the situation is more complicated than it needs to be. In all three case, the absolute value is being returned; only the type of the data differs. In C++, you can correct this situation by overloading one name for the three types of data, as this example illustrates:

Tuy nhiên, do có 3 tên khác nhau tương ứng với 3 kiểu dữ liệu nên tình trạng trở nên phức tạp hơn. Trong cả ba trường hợp, giá trị tuyệt đối được trả về, chỉ có kiểu dữ liệu là khác nhau. Trong C++, bạn có thể hiệu chỉnh tình trạng này bằng cách quá tải một tên cho ba kiểu dữ liệu như được trình bày trong ví dụ minh họa sau:

```
#include <iostream>

using namespace std;

//overload abs() three ways

int abs(int n);

long abs(long n);

double abs(double n);

int main()

{

   cout<<" Absolute value of -10: "<<abs(-10)<<"\n\n";

   cout<<"   Absolute   value   of   -10L:   "<<abs(-
10L)<<"\n\n";

   cout<<"   Absolute   value   of   -10.01:   "<<abs(-
10.01)<<"\n\n";

   return 0;

}

// abs() for ints

int abs(int n)

{
```

```
   cout<<" In integer abs()\n";
   return n<0? -1:n;
}

// abs( for longs
long abs(long n)
{
   cout<< " In long abs()\n";
   return n<0?-1:n;
}
//abs() for doubles
double abs (double n)
{
   cout<<" In double abs()\n";
   return n<0? -1: n;
}
```

*As you can see, this program defines three functions called **abs()**-one for each data type. Inside **main()**,**abs()b** is called using three difference types of arguments. The compiler automatically calls the correct version of **abs()** based upon the type of data used as an argument. The program produces the following output:*

Như bạn thấy chương trình này định nghĩa ba hàm gọi là **abs()**-mỗi hàm cho mỗi kiểu dữ liệu. Bên trong hàm **main()**, hàm **abs()** được gọi bằng cách dùng ba kiểu đối số khác nhau. Trình biên dịch tự động gọi 4 phiên bản đúng của hàm **abs()**, dựa vào kiểu dữ liệu của đối số.

```
In integer abs()
Absolute value of -10: 10
```

```
In long abs()

Absolute value of -10L: 10

In double abs()

Absolute value of -10.01: 10.01
```

*Although this example is quite simple, it still illustrates the value of function overloading. Because a single name can be used to describe a general class of action, the artifical complexity caused by three slightly different names-in this case, **abs()**, **fabs()**, and **lab()**-is eliminated. You now must remember only one name-the one that describles the general action. It is left to the compiler to choose the appropriate specific version of the function (that is, the method) to call. This has the net effect of reducing complexity. Thus, through the use of polymorphism, three names have been reduced to one.*

Dù đây là ví dụ đơn giản, nhưng nó vẫn minh họa ý nghĩa của sự quá tải hàm. Do một tên được dùng để mô tả một lớp tác động tổng quát, nên tính phức tạp nhân tạo do 3 tên khác nhau gây ra – trong trường hợp này là **abs()**,**labs()**,**fabs()** đã bị loại bỏ. Bạn phải nhớ rằng chỉ một tên – một tên mô tả tác động *tổng quát.* Trình biên dịch sẽ chọn phiên bản *thích hợp* của hàm để gọi. Chính điều này làm giảm bớt tính phức tạp. Do đó, nhờ sử dụng tính đa dạng, ba tên giảm còn lại một tên.

*While the use of polymorphism in this example is fairly trivia, you should be able to see how in a very large program, the one interface, multiple methods * approach can be quite effective.*

Ta thấy rằng cách sử dụng đa dạng trong ví dụ này rõ ràng là không đáng kể, nhưng khi bạn xem xét cách nó ứng dụng trong một chương trình lớn, phương pháp "một giao diện, đa phương thức" hoàn toàn có thể mang lại nhiều kết quả.

*Here is another example of function overloading. In this case, the function **date()** is overloaded to accept the date either as a string or as three integers. In both case, the function displays the date passed to it.*

Sau đây là một ví dụ khác về việc ứng dụng nạp chồng hàm. Trong trường hợp này, hàm **date()** được nạp chồng để nhận vào ngày tháng ở dạng một chuỗi hay là ở dạng 3 số nguyên. Trong cả 2 trường hợp, hàm đều cho kết quả ngày tương ứng.

```
#include <iostream>
```

```
using namespace std;
void date(char *date);
void date(int month, int day, int year);

int main()
{
      date("8/23/99");
      date(8,23,99);
      return 0;
}
void date(char *date)
{
      cout << "Date: " << date << "\n";
}
void date(int month, int day, int year)
{
      cout << "date: " << month << "/" ;
      cout << day << "/" << year << "\n";
}
```

This example illustrates how function overloading can provide the most natural interface to a function. Since it is very common for the date to be represented as either a string or ass three inegers containing the month, day, and year, you are free to select the most convenient form relative to the situation at hand.

Ví dụ này minh họa được ứng dụng của nạp chồng hàm có thể tạo ra được giao diện tự nhiên nhất cho một hàm. Như vậy, ngày được trình bày rất chung hoặc là bằng một chuỗi hoặc là bằng 3 số nguyên gồm ngày, tháng và năm, tùy theo hoàn cảnh hiện tại mà bạn tự do lựa chọn dạng thích hợp nhất tương ứng.

3. *So far, you have seen overloaded functions that differ in the data types of their argunments. However, overloaded functions can also differ in the number of arguments, as this example illustrates:*

3. Đến đây, bạn đã thấy được định nghĩa chồng các hàm khác nhau về kiểu dữ liệu của đối số tương ứng với từng hàm. Tuy nhiên, định nghĩa chồng các hàm cũng có thể khác nhau về số lượng đối số, như ví dụ minh họa sau:

```
#include <iostream>

using namespace std;


void f1 (int a);

void f1(int a, int b);


int main()

{

   f1(10);

   f1(10,20);

   return 0;

}


void f1( int a)

{

   cout << "In f1(int a)\n";

}


void f1(int a, int b)

{

   cout << "In f1(int a, int b)\n";

}
```

4. *It is important to understand that the return type alone is not a sufficient difference to allow function overloading. If two functions differ only in the type of data they return, the compiler will not always be able to select the proper one to call. For example, this fragment is incorrect because it is inherently ambiguous:*

4. Thật quan trọng để hiểu là chỉ xét kiểu giá trị trả về thôi thì không đủ khác biệt để cho phép định nghĩa chồng hàm. Nếu 2 hàm chỉ khác nhau về kiểu dữ liệu trả về thì trình biên dịch sẽ không thể nào gọi được hàm thích hợp. Chẳng hạn như đoạn chương trình dưới đây sai vì rõ ràng là nó rất nhập nhằng, mơ hồ:

```
int f1(int a);

double f1(int a);.

f1(10); //trinh bien dich se goi ham nao day???
```

*As the comment indicates, the compiler has no way of knowing which version of **f1()** to call.*

Theo như chú thích biểu thị ở trên thì trình biên dịch không có cách nào biết được cần phải gọi phiên bản nào của hàm f1( ).

Bài tập (Exercises):

*1. Create a function called **sroot()** that returns the square root of its argument. Overload sroot() three ways: have it return the square root of an integer, a long integer, and a **double.** (To actually compute the square root, you can use the standard library function sqrt().)*

1. Tạo lập hàm **sroot()** trả về căn bậc hai của đối số đưa vào. Định nghĩa chồng hàm **sroot()** với 3 kiểu: trả về căn bậc hai của một số nguyên, một số nguyên dài và một số thực dài. (Thực ra để tính căn bậc hai, bạn có thể sử dụng hàm thư viện chuẩn **sqrt()**).

2. The C++ standard library contains these three functions:

2. Thư viện chuẩn của C++ chứa 3 hàm sau:

```
double atof(const char *s)

int atoi(const char *s)

long atol(const char *)
```

*These functions return the numeric value contained in the string pointed to by **s.** Specifically, **arof()** return a double. **Atoi()** returns an integer, and atol returns a long. Why is it not possible to overload these functions?*

Những hàm này trả về giá trị số chứa trong một chuỗi trỏ tới địa chỉ biến s. Nói một cách cụ thể, hàm **atof()** trả về một số thực dài, hàm **atoi()** trả về một số nguyên, và hàm **atol()** trả về một số nguyên dài. Hãy cho biết tại sao không thể định nghĩa chồng 3 hàm trên?

*3. Create a function called **min()** that returns the smaller of the two numeric arguments used to call the function. Overload **min()** so it accepts characters, and **doubles** as arguments.*

3. Tạo lập hàm **min()** trả về giá trị nhỏ hơn của 2 đối số khi gọi hàm. Định nghĩa chồng hàm **min()** để nó có thể nhận vào đối số là ký tự, số nguyên và số thực.

*4. Create a function called **sleep()** that pause the computer for the number of seconds specified by its single argument. Overload **sleep()** so it can be called with either an integer or a string representation of an integer. For example, both of these calls to **sleep()** will cause the computer to pause for 10 seconds:*

4. Tạo lập hàm **sleep()** để dừng máy tính trong một số lượng giây nhất định được xác lập bởi một đối số đưa vào. Định nghĩa chồng hàm **sleep()** để nó có thể được gọi với đối số là một số nguyên hoặc một chuỗi số biểu diễn cho một số nguyên. Ví dụ như cả hai lời gọi hàm dưới đây đều thực hiện dừng máy tính trong 10 giây:

```
sleep(10);

sleep("10");
```

Demonstrate that your functions work by including them in a short program. (Fell free to use a delay loop to pause the computer).

Chứng minh rằng các hàm của bạn hoạt động bằng cách đặt chúng trong một chương trình ngắn. (Sử dụng một vòng lặp trì hoãn để dừng máy tính).

1.8. C++ KEYWORDS – TỪ KHÓA TRONG C++

***C++** support all of the keywords defined by C and adds 30 of its own. The entire set of keywords defined by C++ is shown in Table 1-1. Also, early versions of*

C++ defined the overload keyword, but it is now obsolete.

C++ hỗ trợ tất cả các từ khóa được định nghĩa trong C và bổ sung thêm 30 từ khóa riêng. Toàn bộ từ khóa được định nghĩa trong C++ trình bày trong bảng 1-1. Cũng có những phiên bản C++ trước đây định nghĩa từ khóa **overload**, nhưng bây giờ nó đã bị xóa bỏ.

Kiểm tra các kỹ năng (Skills Check) :

At this point you should be able to perform the following exercises and answer the questions.

Give brief descriptions of polymorphism, encapsulation, and inheritance.
How can comments be included in a C++ program?
Write a program that uses C++ style I/O to input two integers from the keyboard and then displays the result of raising the first to the power of the second. (For example, if the user enters 2 and 4, the result is 2^4, or 16)

Ở điểm này, bạn có thể làm các bài tập và trả lời các câu hỏi sau:

1. Mô tả tóm tắt về sự đa dạng, sự đóng gói và sự kế thừa trong C++.
2. Các chú thích có thể được đưa vào một chương trình C++ như thế nào?
3. Viết một chương trình sử dụng dạng nhập xuất của C++ để nhập vào 2 số nguyên từ bàn phím và trình bày kết quả lũy thừa của số đầu với số mũ là số thứ hai. (ví dụ như nêu bạn nhập vào là 2 và 4 thì kết quả xuất ra là 2^4 hay 16).

asm	const_cast	explicit	int	register	switch	union
auto	continue	extern	long	reinterpret_cast	template	unsigned
bool	default	false	mutable	return	this	using
break	delete	float	namspace	short	throw	virtual
case	do	for	new	signed	true	void
catch	double	friend	operator	sizeof	try	volatile
char	dynamic_cast	goto	private	static	typedef	wchar_t
class	else	if	protected	static_cast	typeid	while
const	enum	inline	public	struct	typename	

*Create a function called **rev_str()** that reverses a string. Overload **rev_str()** so it can be called with either one charater array or two. When it is called with one string, have that one string contain the reversal. When it is called with two strings, returns the reversed string tin the second argument. For example:*

4. Tạo lập hàm **rev_str()** đảo ngược một chuỗi. Định nghĩa chồng hàm **rev_str()** để nó có thể được gọi thực hiện với đối số là một mảng ký tự hoặc hai. Khi gọi hàm với đối số là một chuỗi thì nó sau khi thực hiện chuỗi đó sẽ nhận giá trị của chuỗi đảo ngược. Còn khi gọi hàm với đối số là 2 chuỗi thì chuỗi đảo ngược sẽ được trả về cho đối số thứ hai. Ví dụ:

```
char s1[80], s2[80];

strcpy(s1, "hello");

rev_str(s1, s2);   //reversed string goes in s2, s1
untouched

rev_str(s1);  // reversed string is returned ins1
```

Given the following new style C++ program, show how to change it into its old-style form.

5. Cho chương trình C++ kiểu mới sau, hãy đổi nó về dạng cũ:

```
#include <iostream>

using namespace std;

int f(int a);

int main()

{

     cout<< f(10);

     return 0;

}


int f(int a)

{
```

```
        return a * 3.1416
    }
```

*6. What is the **bool** data type?*

Thế nào là dữ liệu kiểu **luận lý?**

CHƯƠNG 2

Giới Thiệu Lớp (Introducing Classes)

2.1 CONSTRUCTOR AND DESTRUCTOR FUNCTION.

Các H‡m Cấu Tạo v‡ Hủy.

2.2 CONSTRUCTORS THAT TAKE PARAMETERS.

Tham Số Của H‡m Tạo.

2.3 INTRODUCING INHERITANCE.

Giới Thiệu Tính Kế Thừa.

2.4 OBJECT POINTERS.

Con Trỏ Đối Tượng.

2.5 CLASSES, STRUCTURES, AND UNIONS ARE RELATED.

Lớp, Cấu Trúc, v‡ Hội Có Liên Hệ.

2.6 IN-LINE FUNCTIONS.

Các H‡m Nội Tuyến.

2.7 AUTOMATIC IN-LINING SKILLS CHECK.

Nội Tuyến Tự Động.

This chapter introduces classes and objects. Several important topics are covered that relate to virtually all aspects of C++ programming , so careful reading is advised.

Chương n‡y giới thiệu về lớp v‡ đối tượng . V‡i chủ đề quan trọng được đề cập đến liên quan đến tất cả diện mạo của chương trình C++, vì thế lời khuyên l‡ nên đọc cẩn thận.

ÔN TẬP: (REVIEW SKILLS CHECK)

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Write a program that uses C++ style I/O to prompt the user for a string and then display its length.
Create a class that holds name and address information. Store all the information in character string that are private members of the class. Include a public function that displays the name and address. (Call these function store() and display().)
Create an overloaded rotate() function that left-rotates the bits in its argument and returns the result. Overload it so it accepts ints and longs. (A rotate is similar to a shift except that the bit shifited off one end is shifted onto the other end.)
What is wrong with the following fragment?

Trước khi tiến hành bạn nên trả lời thật nhanh và chính xác những câu hỏi dưới đây và làm bài tập sau:

Thiết lập chương trình sử dụng C++ style I/O để gợi ý người sử dụng 1 chuỗi ký tự và sau đó trình bày đầy đủ.
Hãy tạo 1 lớp gồm tên và thông tin địa chỉ . Lưu trữ tất cả thông tinh trong 1 chuỗi ký tự mà gồm những thành viên của lớp. Thêm vào phần hàm chung những lưu trữ về tên và địa chỉ. (gọi những hàm store() và display()).
Hãy tạo 1 hàm quá tải mà phần xoay bên trái những mảnh có sự đối kháng và trả về kết quả. Lượng quá tải thì được chấp nhận ints và longs (sự luân phiên thì gần giống như loại bỏ sự dịch chuyển mà mảnh dịch chuyển từ 1 đến cuối cùng thì được dịch chuyển lên trên của cái cuối cùng khác.)
Cái gì sai trong đoạn chương trình dưới đây?

```
#include <iostream>

using namespace std;

class myclass
```

```
{
      int i;
      public:
      .
      .
};
int  main()
{
      myclass ob;
      ob.i=10;
      .
      .
}
```

2.1. CONSTRUCTOR AND DESTRUCTOR FUNCTIONS - PHƯƠNG THỨC THIẾT LẬP VÀ PHƯƠNG THỨC PHÁ HỦY:

If you have been writting programs for very long, you know that it is common for parts of your program to require initialization. The need for initialization is even more common when you are working with objects. In fact, when applied to real problems, virtually every object you create will require some sort of initialization. To address this situation, C++ allows a construction function to be included in a class declaration. A class's constructor is called each time an obj of that class is created. Thus, any initialization that need to be performed on an obj can be done automatically by the constructor function.

A constructor function has the same name as the class of which it is a part and has no return type. For example, here is a short class that containsa constructor function:

Nếu bạn buộc phải thiết kế một chương trình dài, bạn biết rằng thật bình thường cho chương trình của bạn khi cần giá trị ban đầu. Sự cần thiết cho giá trị ban đầu thì

hơn cả sự kết hợp mà bạn đang làm việc với đối tượng. Thực tế thì khi áp dụng cho một vấn đề thực sự, mỗi đối tượng bạn tạo nên sẽ cần một vài sự sắp xếp giá trị ban đầu. Ở tình trạng này, C++ cho phép phương thức thiết lập có thể được bao gồm trong khai báo lớp. Sự thiết lập của lớp thì được gọi khi mà mỗi lần đối tượng của lớp đó được tạo. Vì vậy, những giá trị ban đầu thì cần phải được tiến hành trên một đối tượng có thể làm việc một cách tự động bởi phương thức thiết lập.

Phương thức thiết lập có cùng tên với lớp mà nó là một phần và không có giá trị trả về. Ví dụ như ở dưới đây có một đoạn chương trình ngắn có chứa phương thức thiết lập.

```
#include <iostream>

using namespace std;

class myclass

{

      int a;

      public:

      myclass(); // constructor

      void show();

};

myclass :: myclass()

{

      cout << "In construction\n";

      a=10;

}

void myclass :: show()

{

      cout << a;

}

int main()

{
```

```
    myclass ob;

    ob.show();

    return 0;

}
```

In this simple example, the value of a is initialized by the constructotr my class(). The constructor is called when the obj ob is created. An obj is created when that obj's declaration statement is executed. It is immportant to understand that in C++, a variable declration statement is an "action statement". When you are programming in C, it is easy to think of declaration statement as simple establishing variables. However, in C++, because an obj might have a constructor, a variable declaration statement may, in fact, cause a consideable number of action to occur.

Notice how myclass() is defined. As stated, it has no return type. According to the C++ formal syntax rules, it is illegal for a constructor to have a return type.

For global obj, an obj's constructor is called once, when the program first begins execution. For local obj, the constructor is called each time the declaration statement is executed.

The complement of a construction is the destructor. This function is called when an obj is destroyed. When you are working with obj, it is common to have to perform some actions when an obj is destroyed. For example, an obj that allocates memory when it is created will want to free that memory when it is destroyed. The name of a destruction is the name of its class, preceed by a ~. For example, this class contains a destructor function.

Đây là một ví dụ đơn giản, giá trị của **a** được bắt đầu bởi phương thức thiết lập **myclass()**. Sự thiết lập được gọi khi đối tượng **ob** được tạo. Một đối tượng được tạo khi mà khai báo đối tượng trình bày được thực hiện. Rất là quan trọng để hiểu rằng trong C++, khối khai báo có thể thay thế được là khối hành động. Khi bạn đang lập trình chương trình C thật dễ dàng để nghĩ rằng khối khai báo như là thay đổi chương trình thiết kế một cách giản đơn. Tuy nhiên, trong C++ bởi vì đối tượng có thể có phương thức thiết lập thay đổi trình bày khối khai báo. Thực tế nó là một con số đáng kể.

Làm thế nào để **myclass()** được định nghĩa. Nó không có giá trị trả về. Theo chương trình C++ có nhiều lệnh cú pháp, nó là không hợp lệ nếu phương thức thiết lập có giá trị trả về.

Với một đối tượng tổng thể, phương thức thiết lập của một đối tượng được gọi một lần khi mà chương trình lần đầu tiên được thi hành. Theo định nghĩa từng khu vực, phương thức thiết lập được gọi mỗi lần khối khai báo được thi hành.

Hoàn tất một thiết kế là một phá hủy, hàm được gọi khi đối tượng được phá hủy, khi bạn làm việc với đối tượng thật là bình thường khi thực hiện những hành động mà đối tượng bị phá hủy. Ví dụ: Một đối tượng cấp phát vùng nhớ khi được tạo sẽ muốn giải phóng bộ nhớ khi nó bị phá hủy. Tên của phương thức phá hủy là tên của lớp đặt trước nó bởi dấu ngã. Ví dụ: Lớp này có chứa một phương thức phá hủy:

```
#include <iostream>

using namespace std;

class myclass

{

      int a;

      public:

            myclass(); // constructor

            ~myclass(); // destructor

      void show();

};

myclass :: myclass()

{

      cout << "In construction\n";

      a=10;

}

myclass :: ~myclass()

{

      cout << "Destructing...\n";

}

void myclass :: show()

{
```

```
      cout << a << "\n";
}
int main()
{
      myclass ob;
      ob.show();
      return 0;
}
```

A class's destructor is called when an obj is destroyed. Local obj are destroyed when they go out of scope. Global obj are destroyed when the program ends.

It is not possible to take the address of either a constructor or a destructor.

Note: *Technically, a constructor or a destructor can perform any type of operation. The code within these function does not have to initialize or rest anything related to the class for which they are defined. For example, a construction for the preceding examples could have computed pi to 100 places. However, have a constructor or destrucotor perform actions not directly related to the initialization or orderly destruction of an obj makes for vary poor programming style and should be avoided.*

Phương thức phá hủy của một lớp thì được gọi khi một đối tượng bị phá hủy. Đối tượng xác định bị phá hủy khi nó ra ngoài phạm vi cho phép. Đối tượng tổng thể sẽ bị phá hủy khi chương trình kết thúc.

Không cần thiết để lấy địa chỉ của cả phương thức thiết lập hay phương thức phá hủy.

Lưu ý:

Theo kĩ thuật, phương thức thiết lập hay phương thức phá hủy có thể thực hiện bất cứ loại sự việc. Đoạn code trong những hàm này không phải bắt đầu hoặc thiết lập lại bất cứ thứ gì liên quan đến lớp mà nó được đinh nghĩa. Ví dụ, một phương thức phá hủy cho ở ví dụ trước có thể nhập **pi** to 100 chỗ. Tuy nhiên, có phương thức thiết lập hay phương thức phá hủy thực hiện hành động không trực tiếp liên quan đến giá trị ban đầu hoặc từng bước phá hủy của một đối tượng làm cho cấu trúc chương trình thiếu phong phú và nên tránh điều đó.

Examples:

You should recall that the stack class created in Chapter 1 required an initialization function to set the stack index variable. This is precisely the sort of operation that a constructor function was designed to perform. Here is an improved version of the stack class that uses a construcor to automatically initialize a stack obj when it is created:

VÍ DỤ:

1. Bạn nên làm lại một lần để lớp ngăn xếp đã tạo trong chương 1, hàm giá trị ban đầu được thiết lập liệt kê ngăn xếp có thể thay đổi. Đó là một loạt hoạt động mà phương thức thiết lập đã thiết kế để thi hành. Đây là một phiên bản cải thiện của lớp sắp xếp mà sử dụng phương thức thiết lập đế sử dụng một cách tự đông ngăn xếp đối tượng khi nó được cài đặt:

```
#include <iostream>

using namespace std;

#define SIZE 10

// Declare a stack class for characters.

class stack

{

      char stck[SIZE]; // hold the stack

      int tos; // index of top of stack

      public:

            stack(); // constructor

            void push(char ch); // push character on stack

            char pop(); // pop character from stack

};


// Initialize the stack.

stack :: stack()
```

```
{
      cout << "Constructing a stack\n";
      tos=0;
}
// Push a character.
void stack :: push(char ch)
{
      if(tos==SIZE)
      {
            cout << "Stack is full\n";
            return;
      }
      stck[tos]=ch;
      tos++;
}
// Pop a character
char stack :: pop()
{
      if(tos==0)
      {
            cout<<"Stack is empty\n";
            return 0; // return null on empty stack
      }
      tos--;
      return stck[tos];
}
int main()
```

```
{
      // Create two stacks that are automatically initialized.
      stack s1, s2;
      int i;
      s1.push('a');
      s2.push('x');
      s1.push('b');
      s2.push('y');
      s1.push('c');
      s2.push('z');
      for(i=0;i<3;i++)
            cout <<"Pop s1:"<<s1.pop()<<"\n";
      for(i=0;i<3;i++)
            cout <<"Pop s2:"<<s2.pop()<<"\n";
      return 0;
}
```

As you can see, now the initialiazation task is performed automatically by the constructor function rather than by a separate function that must be explicitly called by the program. This is an important point. When an initialization is performed automatically when an obj is created, it eliminated any prospect that, by error, the initialization will not be performed. This is another way that obj help reduce program complexity. You, as the programmer, don't need to worry about initialization-it is performed automatically when the obj is brought into existence.

Như bạn thấy, giờ đây những yêu cầu giá trị ban đầu được thực hiện tự động bởi phương thức thiết lập hơn là những hàm tách rời mà rõ ràng phải được gọi là chương trình. Đây là một điểm quan trọng. Khi mà giá trị ban đầu được thực hiện một cách tự động khi mà một đối tượng được tạo, ngoại trừ những trường hợp này bị lỗi nên giá trị ban đầu sẽ được thực hiện. Đây là một cách mà đối tượng có thể giảm bớt những chương trình phức tạp. Bạn-một người làm chương trình, không cần quan tâm đến giá trị ban đầu, nó được thực hiện một cách tự động khi đối tượng được thi hành.

2. Here is an example that shows the need for both a constructor and a destructor function. It creates a simple string class, called strtype, that contains a string and its length. When a strtype obj is created, memory is allocated to hold the string and its initial length is set to 0. When a strtype obj is destroyed, that memory is released.

2. Sau đây là một ví dụ để chỉ rõ sự cần thiết của cả phương thức thiết lập và phương thức phá hủy. nó tạo một lớp chuỗi kí tự đơn giản được gọi là strtype, nó chứa đựng chuỗi kí tự và chiều dài của nó. Khi mà đối tượng strtype được tạo bộ nhớ được cấp phát để giữ chuỗi kí tự và chiều dài ban đầu của nó được bắt đầu từ 0. Khi mà đối tượng strtype bị phá hủy thì bộ nhớ sẽ được giải phóng.

```
#include <iostream>

#include <cstring>

#include <cstdlib>

using namespace std;

#define SIZE 255

class strtype

{

      char *p;

      int len;

      public:

            strtype(); // constructor

            ~strtype(); // destructor

            void set(char *ptr);

            void show();

};

// Initialize a string obj.

strtype::strtype()

{

      p=(char *)malloc(SIZE);
```

```
        if(!p)
        {
                cout<<"Allocation error\n";
                exit(1);
        }
        *p='\0';
        len=0;
}
// Free memory when destroying string obj.
strtype::~strtype()
{
        cout<<"Freeing p\n";
        free(p);
}
void strtype::set(char *ptr)
{
        if(strlen(p)>=SIZE)
        {
                cout<<"String too big\n";
                return;
        }
        strcpy(p,ptr);
        len=strlen(p);
}

void strtype::show()
{
```

```
        cout<<p<<"-length:"<<len;
        cout<<"\n";
}

int main()
{
        strtype s1, s2;
        s1.set("This is a test.");
        s2.set("I like C++.");

        s1.show();
        s2.show();
        return 0;
}
```

This program uses malloc() and free() to allocated and free memory. While this is perfectly valid, C++ does provide another way to dynamically manage memory, as you will see later in this book.

Note: *The preceding program uses the new-style headers for the C library functions used by the program. As mentioned in Chapter 1, if your compiler does not support these headers, simply substitute the standard C header files. This applies to other programs in this book in wich C library functions are used.*

Chương trình này dùng malloc() và free() để cấp phát và giải phóng bộ nhớ. Trong khi đó là một căn cứ hoàn hảo, C++ cung cấp một cách khác để quản lí bộ nhớ một cách linh hoạt hơn như bạn có

thể thấy ở phần sau của cuốn sách này.

LƯU Ý: Chương trình trước sử dụng tiêu đề phong cách mới cho hàm thư viện C được sử dụng bởi chương trình. Như đã được nhắc đến ở chương 1. Nếu trình biên dịch của bạn không hỗ trợ những tiêu đề này, có thể thay thế bởi các file tiêu đề chuẩn trong C. Những áp dụng trong các chương trình khác trong cuốn sách này ở những

chỗ sử dụng hàm thư viện trong C.

3. *Here is an interesting way to use an object's constructor and destructor. This program uses an object of the* ***timer*** *class to time the interval between when an object of type* ***timer*** *is created and when it is destroyed. When the object's destructor is called, the elapsed time is displayed. You could use an object like this to time the duration of a program or the length of time a function spends withing a block. Just make sure that the object goes out of scope at the point at which you want the timing interval to end.*

Đây là một cách lý thú khác để dùng các hàm tạo và hàm hủy của một đối tượng. Chương trình này dùng đối tượng của lớp **timer** để xác định khoảng thời gian giữa khi một đối tượng được tạo và khi bị hủy. Khi hàm hủy của một đối tượng được gọi, thời gian trôi qua được hiển thị. Bạn có thể dùng đối tượng như thế này để định khoảng thời gian của một chương trình hay độ dài thời gian ma một hàm làm việc trong một khối. Phải đảm bảo rằng đối tượng đi qua phạm vi tại điểm mà bạn muốn định thời gian cho đến lúc kết thúc.

```
#include <iostream.h>
#include <time.h>

class timer
{
     clock_t start;
public:
     timer();// constructor
     ~timer();// destructor
};

timer::timer()
{
     start=clock();
```

```
}


timer::~timer()
{
      clock_t end;
      end = clock();
      cout<<"Elapsed        time:        "<<(end-
      start)/CLOCKS_PER_SEC<<"\n";
}


main()
{
      timer ob;
      char c;
      // delay...
      cout<<" Press a key followed by ENTER: ";
      cin>>c;


      return 0;
}
```

This program uses the standard library function **clock()**, which returns the number of clock cycles that have taken place since the program started running. Dividing this value by **CLOCKS_PER_SEC** converts the value to seconds.

Chương trình này dùng hàm thư viện chuẩn **clock()** để trả về số chu kỳ đồng hồ xảy ra từ khi chương trình bắt đầu chạy. Chia giá trị này cho **CLOCKS_PER_SEC** để chuyển thành giá trị giây

EXERCISES

1. Rework the **queue** class that you developed as an exercise in Chapter 1 by replacing its initialization function with a constructor.
2. Create a class called **stopwatch** that emulates a stopwatch that keeps track of elapsed time. Use a constructor to initially set the elapsed time to 0. Provide two member functions called **start()** and **stop()** that turn on and off the timer, respectively. Include a member function called **show()** that displays the elapsed time. Also, have the destructor function automatically display elapsed time when a **stopwatch** object is destroyed. (To simplify, report the time in seconds.)
3. What is wrong with the constructor shown in the following fragment?

1. Làm lại lớp **queue** trong bài tập chương 1 để thay hàm khởi đầu bằng hàm tạo.

2. Tạo lớp **stopwatch** để so với đồng hồ bấm giờ trong việc xác định thời gian trôi qua. Dùng hàm tạo để đặt thời gian trôi qua lúc đầu về 0. Đưa vào hàm thành viên **show()** để hiển thị thời gian trôi qua khi đối tượng **stopwatch()** bị hủy.(Để đơn giản thời gian được tính bằng giây).

3. Điều gì sai trong hàm tạo ở đoạn chương trình dưới đây:

```
class sample
{
          double a,b,c;
     public:
          double sample();  // error, why?
};
```

2.2. CONSTRUCTORS THAT TAKE PARAMETERS - THAM SỐ CỦA HÀM TẠO

It is possible to pass arguments to a constructor function. To allow this, simply add the appropriate parameters to the constructor function's declaration and definition. Then, when you declare an object, specify the arguments. To see

how this is accomplished, let's begin with the short example shown here:

Có thể truyền đối số cho hàm tạo. Để làm điều này chỉ cần bổ sung các tham số thích hợp cho khai báo và định nghĩa hàm tạo. Sau đó, khi bạn khai báo một đối tượng, bạn hãy chỉ rõ các tham số này làm đối số. Để thấy rõ điều này, chúng ta hãy bắt đầu với ví dụ ngắn dưới đây:

```
#include <iostream>

using namespace std;

class myclass

{

     int a;

     public:

          myclass(int x);//constructor

          void show();

};


myclass::myclass (int x)

{

     cout<< "In constructor\n";

     a=x;

}


void myclass::show()

{

     cout<<a<<"\n";

}
```

```
int main()
{
     myclass ob(4);
     ob.show();
     return 0;
}
```

*Here the constructor for **myclass** takes one parameter. The value passed to **myclass()** is used to initialize **a**. Pay special attention to how **ob** is declared in **main()**. The value 4, specified in the parentheses following **ob,** is the argument that is passed to **myclass()**'s parameter **x,** which is used to initialize **a.***

Ở đây, hàm tạo đối với **myclass** có một tham số. Giá trị được truyền cho **myclass()** được dùng để khởi tạo **a**. Cần chú ý cách khai báo **ob** trong hàm **main()**. Giá trị 4, được chỉ rõ trong các dấu ngoặc sau **ob**, là đối số được truyền cho tham số x của **myclass**, được dùng để khởi tạo **a**.

Actually, the syntax for passing an argument to a parameterized constructor is shorthand for this longer form:

```
myclass ob = myclass(4);
```

Thực sự các phép truyền một đối số cho một hàm tạo được tham số hóa có dạng dài hơn như sau:

```
myclass ob = myclass(4);
```

However, most C++ programmers use the short form. Actually, there is a slight technical difference between the two forms that relates to copy constructors, which are discussed later in this book. But you don't need to worry about this distinction now.

Tuy nhiên, hầu hết những người lập trình C++ sử dụng ngắn gọn. Thực ra có một sự khác biệt nhỏ giữa hai dạng hàm khởi tạo sẽ được nói rõ trong phần sau của cuốn sách này. Nhưng bạn không cần lo lắng về điều này,

Notes *Unlike constructor functions, destructor functions cannot have parameters. The reason for this is simple enough to understand: there exists no mechanism by which to pass arguments to an objects that is being destroyed.*

CHÚ Ý: Không giống như các hàm khởi tạo, các hàm hủy không có tham số. Lý do thật dễ hiểu: không có cơ chế nào để truyền đối số cho một đối tượng bị hủy.

CÁC VÍ DỤ (EXAMPLES):

1 *It is possible-in fact, quite common-to pass a constructor more than one argument. Here **myclass()** is passed two arguments:*

1 Thực tế, rất thông dụng, có thể truyền nhiều tham số cho một hàm tạo. Sau đây, có hàm tham số truyền cho **my class():**

```
#include <iostream.h>

class myclass

{

     int a,b;

     public:

          myclass(int x,int y);//constructor

          void show();

};


myclass::myclass (int x,int y)

{

     cout<< "In constructor\n";

     a=x;

     b=y;

}


void myclass::show()
```

```
{
     cout<<a<<" "<<b<<"\n";
}

int main()
{
     myclass ob(4,7);
     ob.show();
     return 0;
}
```

*Here 4 is passed to **x** and 7 is passed to **y**. This same general approach is used to pass any number of arguments you like(up to the limit set by the compiler, of course).*

Ở đây, 4 được truyền cho x và 7 cho y. Cách tổng quát như vậy được dùng để truyền một số đối số bất kỳ mà bạn muốn(dĩ nhiên, lên đến giới hạn của trình biên dịch).

2 *Here is another version of the **stack** class that uses a parameterized constructor to pass a "name" to a stack. This single-character name is used to identify the stack that is being referred to when an error occurs.*

Đây là một dạng khác của lớp **stack** dùng hàm tạo được tham số hóa để truyền "tên" cho ngăn xếp. Tên ký tự đơn giản này dùng để nhận ra ngăn xếp nào được tham chiếu khi có lỗi xảy ra.

```
#include <iostream.h>
#define SIZE 10

// Declare a stack class for characters.
class stack
```

```
{
     char stck[SIZE]; // holds the stack
     int tos;// index of top-of-stack
     char who; //identifies stack
     public:
          stack(char c); //constructor
          void push (char ch); //push charater on stack
          char pop(); // pop character from stack
};

// Initialize the stack.
stack::stack(char c)
{
     tos = 0;
     who = c;
     cout<<" constructing stack "<<who<<"\n";
}

//push a character
void stack::push(char ch)
{
     if(tos==SIZE)
     {
          cout<<"Stack "<<who<< " is full\n";
          return ;
```

```
     }
     stck[tos] = ch;
     tos++;
}

//pop a character
char stack::pop()
{
     if(tos==0)
     {
          cout<<" Stack "<<who<<" is empty\n";
          return 0;// return null on empty stack
    }
     tos--;
     return stck[tos];
}

int main()
{
     // Creat two stacks that are automatically
initialized.
     stack s1('A'),s2('B');
     int i;
     s1.push('a');
     s2.push('x');
     s1.push('b');
```

```
        s2.push('y');

        s1.push('c');

        s2.push('z');

        // This will generate some error messages.

        for(i=0;i<5;i++)        cout<<"        Pop        s1:
"<<s1.pop()<<"\n";

        for(i=0;i<5;i++)        cout<<"        Pop        s2:
"<<s2.pop()<<"\n";


        return 0;

}
```

Giving objects a " name", as shown I this example, is especially useful during debugging, when it is important to know which object generates an error.

Khi cho các đối tượng một "tên" như chỉ rõ trong ví dụ, thì điều quan trọng là biết được đối tượng nào tạo ra lỗi, nên rất hữu ích trong quá trình gỡ rối.

*3. Here is a different way to implement the **strtype** class (developed earlier) that uses a parameterized constructor function:*

3. Đây là cách khác để thực hiện lớp **strtype** (đã được trình bày) sử dụng một hàm tạo được tham số hóa:

```
#include <iostream.h>

#include <malloc.h>

#include <string.h>

#include <stdlib.h>


class strtype

{
```

```
        char*p;
        int len;
        public:
                strtype(char*ptr);
                ~strtype();
                void show();
};

strtype::strtype(char*ptr)
{
        len=strlen(ptr);
        p=(char*)malloc(len+1);
        if(!p)
        {
                cout<<" Allocation error\n";
                exit(1);
        }
        strcpy(p,ptr);
}

strtype::~strtype()
{
        cout<<" Freeing p \n";
        free(p);
}
```

```
void strtype::show()
{
     cout<<p<<" - length: "<<len;
     cout<<"\n";
}

main()
{
     strtype s1("This is a test"), s2("i like C++");

     s1.show();
     s2.show();
     return 0;
}
```

*In this version of **strtype**, a string is given an initial value using the constructor function.*

Trong dạng này của **strtype**, một chuỗi được cho một giá trị ban đầu bằng cách dùng hàm tạo.

4. *Although the previous examples have used constants, you can pass an object's constructor any vaild expression, including variables. For example, this program uses input to construct an object:*

4. Mặc dù các ví dụ trước đã dùng các hằng số, bạn cũng có thể truyền cho hàm tạo của đối tượng một biểu thức, kể cả các biến. Ví dụ, chương trình sau dùng dữ liệu nhập của người sử dụng để tạo một đối tượng:

```
#include <iostream.h>
```

```
class myclass
{
          int i,j;
     public:
          myclass(int a,int b);
          void show();
};

myclass::myclass(int a,int b)
{
          i=a;
          j=b;
}

void myclass::show()
{
          cout<<i<<' '<<j<<"\n";
}

main()
{
          int x,y;
          cout<<" Enter two integers: ";
          cin>>x>>y;
```

```
        // use variables to construct ob
        myclass ob(x,y);
        ob.show();
        return 0;
}
```

This program illustrates an important point about objects. They can be constructed as needed to fit the exact situation at the time of their creation. As you learn more about C++, you will see how useful constructing objects "on the fly" is.

Chương trình này minh họa một điểm quan trọng về các đối tượng. Các đối tượng có thể được tạo ra khi cần để làm phù hợp với tình trạng chính xác tại thời điểm tạo ra các đối tượng. Khi biết nhiều hơn nữa về C++, bạn sẽ thấy rất hữu dụng khi tạo ra các đối tượng theo cách "nửa chừng" như thế.

BÀI TẬP (EXERCISES):

1. *Change the **stack** class so it dynamically allocates memory for the stack. Have the size of the stack specified by a parameter to the constructor function. (Don't forget to free this memory with a destructor function).*

1. Thay đổi lớp **stack** để cho nó cấp phát động bộ nhớ cho ngăn xếp. Kích thước ngăn xếp được chỉ rõ bằng một tham số với hàm tạo (Đừng quên giải phóng bộ nhớ bằng một hàm hủy).

2. *Creat a class called **t_and_d** that is passed the current system time and date as a parameter to its constructor when it is created. Have the class include a member function that displays this time and date on the screen. (Hint: Use the standard time and date functions found in the standard library to find and display the time and date).*

2. Hãy tạo lớp **t_and_d** để truyền ngày và giờ hệ thống hiện hành như một tham số cho hàm tạo của nó khi được tạo ra. Lớp gồm có hàm thành viên hiển thị ngày giờ này lên màn hình. (Hướng dẫn: dùng các hàm ngày và giờ chuẩn trong thư viện chuẩn để tìm và hiển thị ngày).

3. *Create a class called **box** whose constructor function is passed three **double** values, each of which represents the length of one side of a box. Have the **box** class compute the volume of the box function called **vol()** that displays the volume of each **box** object.*

3. Hãy tạo lớp **box** có hàm tạo được truyền 3 giá trị **double**, diễn tả độ dài các cạnh của hộp. Hãy cho lớp **box** tính thể tích của hình lập phương và lưu trữ kết quả trong biến **double**. Tạo hàm thành viên **vol()** để hiển thị thể tích của mỗi đối tượng **box**.

2.3. INTRODUCING INHERITANCE - GIỚI THIỆU TÍNH KẾ THỪA:

Although inheritance is discussed more fully in Chapter 7, it needs to be introduced at this time. As it applies to C++, inheritance is the mechanism by which one class can inherit the properties of another. Inheritance allows a hierarchy of classes to be built, moving from the most general to the most specific.

Mặc dù tính kế thừa sẽ được thảo luận đầy đủ hơn trong chương 7, nhưng ở đây cũng cần giới thiệu về tính kế thừa. Khi áp dụng vào C++, tính kế thừa là cơ chế mà nhờ đó một lớp có thể kế thừa các đặc điểm của lớp khác. Tính kế thừa cho phép tạo ra thứ bậc của các lớp, chuyển từ dạng tổng quát nhất đến cụ thể nhất.

To begin, it is necessary to define two terms commonly used when discussing inheritance. When one class is inherited by another, the class that is inherited is called the base class. The inheritance class is called the derived class. In general, the process of inheritance begins with the definition fo a base class. The base class defines all qualities that will be common to any derived classes. In essence, the base class represents the most general description of a set of traits. A derived class inherits those general traits and adds properties that are specific to that class.

Để bắt đầu, cần phải định nghĩa hai thuật ngữ thường được dùng khi thảo luận về tính kế thừa. Khi một lớp được kế thừa bởi một lớp khác thì lớp được kế thừa được gọi là **lớp cơ sở** (base class) và lớp kế thừa được gọi là lớp **dẫn xuất** (derived class). Nói chung, quá trình kế thừa bắt đầu từ định nghĩa lớp cơ sở. Lớp cơ sở xác định các tính chất mà sẽ trởnên thông dụng cho các lớp dẫn xuất. Nghĩa là lớp cơ sở hiển thị mô tả tổng quát nhất một tập hợp các đặc điểm. Một lớp dẫn xuất kế thừa các đặc điểm tổng quát này và bổ sung thêm các tính chất riêng của lớp dẫn

xuất.

To understand how one class can inherit another, let's first begin with an example that, although simple, illustrates many key features of inheritance.

To start, here is the declaration for the base class:

Để hiểu cách một lớp kế thừa lớp khác, trước hết chúng ta hãy bắt đầu bằng một ví dụ, dù đơn giản, nhưng cũng minh họa nhiều đặc điểm chính của tính kế thừa.

Bắt đầu, đây là khai báo cho lớp cơ sở:

```
// Define base class.
Class B
{
          int i;
     public:
          void set_i(int n);
          int get_i();
};
```

Dùng lớp cơ sở này để thấy rõ lớp dẫn xuất kế thừa nó:

```
// Define derived class
Class D: public B
{
          int j;
     public:
          void set_j(int n);
          int mul();
};
```

*Look closely at this declaration. Notice that after the class name D there is a colon followed by the keyword **public** and the class name **B**. This tells the complier that class **D** will inherit all components of class **B**. The keyword **public** tells the compiler that **B** will be inherited such that all public elements of the base class will also be public elements of the derived class. However, all private elements of the base class remain private to it and are not derctly accessible by the derived class.*

*Here is an entire program that uses the **B** and **D** classes:*

Xét kĩ hơn khai báo này, chú ý rắng sau tên lớp D là dấu hai chấm (:) và tiếp đến là từ khóa **public** và tên lớp **B**. Điều này báo cho trình biên dịch biết lớp **D** sẽ kế thừa các thành phần của lớp **B**. Từ khóa **public** báo cho trình biên dịch biết **B** sẽ được kế thừa sao cho mọi phần tử chung của lớp cơ sở cũng sẽ là các phần tử chung của lớp dẫn xuất. Tuy nhiên, mọi phần tử riêng của lớp cơ sở vẫn còn riêng đối với nó và không được truy cập trực tiếp bởi lớp dẫn xuất.

Đây là toàn bộ chương trình dùng các lớp B và D:

```
// Asimple example of inheritance

#include <iostream.h>


// Define base class.

class base

{

     int i;

     public:

          void set_i(int n);

          int get_i();

};

// Define derived class.

class derived: public base

{
```

```
     int j;
     public:
          void set_j(int n);
          int mul();
};

// Set value in base.
void base::set_i(int n)
{
     i=n;
}

// Return value of i in base.
int base::get_i()
{
     return i;
}

// Set value of j in derived.
void derived::set_j(int n)
{
     j=n;
}

// Return value of base's i times derived's j.
```

```
int derived::mul()
{
     // derived class can call base class public member
functions.
     return j* get_i();
}

main()
{
     derived ob;

     ob.set_i(10); // load i in base
     ob.set_j(4); // load j in derived

     cout<<ob.mul(); //display 40
     return 0;
}
```

*Look at the definition of **mul()**. Notice that it calls **get_i(),** which is a member of the base class **B**, not of **D**, without linking it to any specific object. This is possible because the public members of **B** become public members of **D**. However, the reason that **mul()** must call **get_i()** instead of accessing **I** directly is that the private members of a base class (in this case, **i**) remain private to it and not accessible by any derived class. The reason that private members of a class are not accessible to derived classes is to maintain encapsulation. If the private members of a class could be made public simply by inheriting the class, encapsulation could be easily circumvented.*

Xét định nghĩa hàm **mul()**. Chú ý rằng hàm này gọi **get_i()** là thành viên của lớp cơ sở B, không phải của **D**, không liên kết nó với đối tượng cụ thể nào. Điều này có thể được vì các thàn viên chung của B cũng trở thành các thành viên chung của D. Tuy nhiên lý do mà **mul()** phải gọi **get_i()** thay vì truy cập I trực tiếp là vì các thành viên

riêng của lớp cơ sở (trong trường hợp này là i) vẫn còn riêng đối với lớp cơ sở và không thể truy cập bởi lớp dẫn xuất nào. Lý do mà các thành viên riêng của lớp không thể được truy cập bởi các lớp dẫn xuất là để duy trì tính đóng kín. Nếu các thành viên riêng của một lớp có thể trở thành chung cho kế thừa lớp thì tính đóng kín dễ dàng bị phá vỡ.

The general form used to inherit a base class is shown here:

Dạng tổng quát dùng để kế thừa một lớp được trình bày như sau:

```
Class derived-class-name: access-specifier base-class-nam

{

      .

      .

      .

};
```

*Here access-specifier is one of the following three keywords: **public, private,** or **protected.** For now, just use **public** when inheriting a class. A complete description of access specifiers will be given later in this book*

Ở đây <thành phần cần truy cập> là một trong 3 từ khóa sau: **public, private,** hoặc là **protected.** Hiện giờ, chỉ sử dụng **public** khi thừa kế 1 lớp. Sự mô tả hoàn thiện hơn về <thành phần cần truy cập> sẽ được đưa ra vào phần sau của cuốn sách này.

VÍ DỤ (*EXAMPLE*):

*Here is a program that defines a generic base class called **fruit** that describes certain characteristics of fruit. This class is inherited by two derived classes called **Apple** and **Orange.** These classes supply specific information to fruit that are related to these types of fruit.*

Đây là một chương trình định nghĩa lớp cơ sở về giống loài được gọi là **fruit** nó thể hiện những đặc điểm nào đó của trái cây.Lớp này thì được thừa hưởng bởi hai

lớp nhận là **Apple** and **Orange**. Những lớp này cung cấp những thông tin đặc trưng về **fruit**.những thứ liên quan đến nhiều loại trái cây khác.

```
// An example of class inheritance.

#include <iostream>

#include <cstring>

using namespace std;


enum yn

{

     no, yes;

}

enum color

{

     red, yellow, green, orange;

}


void out(enum yn x);


char *c[] = {"red", "yellow", "green", "orange"};


// Generic fruit class.

class fruit

{

     // in this base, all elements are public

     public:

          enum yn annual;
```

```
          enum yn perennial;
          enum yn tree;
          enum yn tropical;
          enum color clr;
          char name[40];
};

// Derive Apple class.
class Apple: public fruit
{
     enum yn cooking;
     enum yn crunchy;
     enum yn eating;
     public:
          void seta(char*n, enum color c, enum yn ck,
enum yn crchy, enum yn e);
          void show();
};

// Derive orange class.
class Orange: public fruit
{
     enum yn juice;
     enum yn sour;
     enum yn eating;
     public:
```

```
          void seto(char*n, enum color c, enum yn j,
enum yn sr,   enum yn e);
          void show();
};

void Apple::seta(char*n, enum color c, enum yn ck,
enum yn crchy, enum yn e)
{
     strcpy(name, n);
     annual = no;
     perennial = yes;

     tree = yes;
     tropical = no;
     clr = c;
     cooking = ck;
     crunchy = crchy;
     eating = e;
}

void Orange::seto(char*n, enum color c, enum yn j,
enum yn sr, enum yn e)
{
     strcpy(name, n);
     annual = no;
     perennial = yes;
```

```
        tree = yes;
        tropical = yes;
        clr = c;
        juice = j;
        sour = sr;
        eating = e;
    }

    void Apple::show()
    {
        cout << name << "apple is:" << "\n";
        cout << " Annual: ";
        out (annual);
        cout << "Perennial: ";
        out (perennial);
        cout << "Tree: ";
        out (tree);
        cout << "Tropical: ";
        out (tropical);
        cout << "Color: " << c[clr] << "\n";
        cout << "Good for cooking: ";
        out (cooking);
        cout << "Crunchy: ";
        out (crunchy);
        cout << "Good for eating: ";
```

```
        out (eating);
        cout << "\n";
    }

    void Orange::show()
    {
        cout << name "orange is:" << "\n";
        cout << " Annual: ";
        out (annual);
        cout << "Perennial: ";
        out (perennial);
        cout << "Tree: ";
        out (tree);
        cout << "Tropical: ";
        out (tropical);
        cout << "Color: " << c[clr] << "\n";
        cout << "Good for juice: ";
        out (juice);
        cout << "Sour: ";
        out (sour);
        cout << "Good for eating: ";
        out (eating);
        cout << "\n";
    }
```

```
void out(enum yn x)
{
     if(x==no)
          cout << "no\n";
     else
          cout << "yes\n";
}

int main()
{
     Apple a1, a2;
     Orange o1, o2;

     a1.seta("Red Delicious", red, no, yes, yes);
     a2.seta("Jonathan", red, yes, no, yes);

     o1.seto("Navel", orange, no, no ,yes);
     o2.seto("Valencia", orange, yes, yes, no);

     a1.show();
     a2.show();

     o1.show();
     o2.show();
```

```
    return 0;
}
```

*As you can see, the base class **fruit** defines several characteristics that are common to all types of fruit. (Of course, in order to keep this example short enough to fit conveniently in the book, the **fruit** class is somewhat simplified.) For example, all fruit grows on either annual perennial plants. All fruit grows on either on trees or on other types of plants, such as vines or bushes. All fruit have a color and a name. This base class is then inherited be the **Apple** and **Orange** classes. Each of these classes supplies information specific to its type of fruit.*

Như các bạn có thể thấy, lớp cơ sở **fruit** định nghĩa những nét phổ biến đối với tất cả những loại trái cây.(Tuy nhiên để giữ cho ví dụ đủ ngắn gọn để phù hợp với cuốn sách, lớp **fruit** có phần được đơn giản hóa). Ví dụ, tấ cả những loại hoa quả phát triển hoặc là cây một năm hoặc là cây lâu năm. Tất cả những loại hoa quả hoặc là ở trên cây hoặc là một dạng khác của thực vật., chẳng hạn như cây leo, hoặc là cây bụi. Tất cả những loại trái cây đều có màu và tên. Lớp cớ sở thì được thừa kế bởi lớp ***Apple*** và ***Orange*.** Mỗi lớp này cấp những nét điển hình của mỗi loại trái cây.

This example illustrates the basic reason for inheritance. Here, a base class is created that defines the general traits associated with all fruit. It is left to the derived classes to supply those traits that are specific to each individual case.

Ví dụ này làm sáng tỏ lí do cơ bản cho việc thừa kế. Ở đây, một lớp cơ sở được tạo ra để định nghĩa những đặc điểm chung liên quan đến tất cả các loại trái cây.Nó thì được đưa đến những lớp nhận để cung cấp những đặc điểm đó, và chúng thì cụ thể đối với mỗi trường hợp riêng biệt.

This program illustrates another important fact about inheritance: A base class is not exclusively "owned" by a derived class. A base class can be inherited by any number of classes.

Chương trình này còn minh họa cho nhiều mặt quan trong khác về sự thừa kế: một lớp cơ sở không chỉ dành riêng cho một lớp nhận. Một lớp cơ sở có thể được thừa kế bởi nhiều lớp.

BÀI TẬP (*Exercise*):

Given the following base class

1. Cho lớp cơ sở sau:

```
class area_cl

{

     public:

          double height;

          double width;

};
```

*Create two derived classes called **rectangle** and **isosceles** that inherit **area_cl.** Have each class include a function called **area()** that returns the area of a rectangle or isosceles triangle, as appropriate. Use parameterized constructors to initialize **height** and **width.***

Tạo 2 lớp con là ***rectangle*** và ***isosceles*** kế thừa từ lớp **area_cl.** Mỗi lớp phải có một hàm ***area()*** và trả về diện tích của hình chữ nhật hay tam giác tương ứng. Sử dụng phương thức thiết lập tham số để đưa giá trị ban đầu vào ***height*** và ***width*.**

1.4. OBJECT POINTERS - CON TRỎ ĐỐI TƯỢNG:

So far, you have been accessing members of an object by using the dot operator. This is the correct method when you are working with an object. However, it is also possible to access a member of an object via a pointer to that object. When a pointer is used, the arrow operator (->) rather than the dot operator is employed. (This is exactly the same way the arrow operator is used when given a pointer to a structure.)

Xa hơn, bạn có thê truy xuất một thành phần của đối tượng bằng cách sử dụng toán tử chấm. Đây là một phương pháp đung khi bạn đang làm việc với một đối tượng. Tuy nhiên, cũng có thể truy xuất đến một thành phần của đối tượng qua một con trỏ chỉ đối tượng đó. Khi một con trỏ được sử dụng, thì toán tử mũi tên (->) được tận dùng nhiều hơn toán tử chấm .(Đây là một cách chính xác tương tự. Toán tử mũi tên được sử dụng khi đụng đến con trỏ chỉ đến một cấu trúc.)

You declare an object pointer just like you declare a pointer to any other type of

variable. Specific its class name, and then precede the variable name with an asterisk. To obtain the address of an object, precede the object with the & operator, just as you do when talking the address of any other type of variable.

Bạn biểu diễn một con trỏ đối tượng giống như là bạn biểu diễn một con trỏ chỉ đến bất kì một loại tham biến nào khác. Chỉ rõ ra lớp tên riêng của nó, và sau đó ghi dấu * trước tên biến. Để sử dụng địa chỉ của một đối tượng, hãy đặt trước đối tượng toán tử &, ngay khi ban thực hiện thì nó sẽ lấy địa chỉ của bất kì kiểu biến nào khác.

Just like pointers to other types, an object pointer, when incremented, will point to the next object of its type.

Giống như con trỏ chỉ đến các loại khác, một con trỏ đối tượng khi tăng lên sẽ chỉ đến đối tượng kế tiếp cùng loại với nó.

VÍ DỤ (*EXAMPLE*):

Here is a simple example that uses an object pointer:

1. Đây là ví dụ đơn giản sử dụng một con trỏ đối tượng:

```
#include "stdafx.h"
#include"iostream"
 using namespace std;
 class myclass
 {
      int a;
     public:
          myclass (int x);//thiet lap
          int get();
 };
```

```
myclass:: myclass(int x)
{
     a=x;
}

int myclass::get()
{
     return a;
}

int main()
{
     myclass ob(120);// tao doi tuong
     myclass*p; //tao con tro chi ve doi tuong
     p=&ob;// dua dia chi cua con tro vao p
     cout<<"Gia tri doi tuong dang giu:"<<ob.get();
     cout<<"\n";

     cout<<"Gia tri cua dia chi con tro dang
giu:"<<p->get();
     return 0;
}
```

Notice how the declaration

Chú ý cách khai báo:

```
myclass *p;
```

*Creates a pointer to an object of **myclass.** It is important to understand that creation of an object pointer does not create an object-it creates just a pointer to one. The address of **ob** is put into **p** by using this statement:*

Tạo một con trỏ chỉ đến đối tượng của lớp **myclass.** Quan trọng là phải hiểu việc tạo một con trỏ đối tượng không phải là việc tạo một đối tượng- nó chỉ là việc tạo một con trỏ chỉ đến đối tương. Địa chỉ của của **ob** được gán cho **p** bằng câu lệnh:

```
p=&ob;
```

Finally, the program shows how the members of an object can be accessed through a pointer.

Cuối cùng chương trình minh họa cách để một con trỏ truy cập đến các thành phần của đối tượng.

We will come back to the subject of object pointers in Chapter 4, once you know more about C++. There are several special features that relate to them.

Chúng ta sẽ trở lại chủ đề con trỏ đối tượng ở chương 4, một khi bạn đã biết thêm về C++. Có vài điểm đặc biệt liên quan đến chúng.

1.5. CLASSES, STRUCTURES, AND UNIONS ARE RELATED - LỚP, CẤU TRÚC, VÀ NHỮNG SỰ KẾT HỢP CÓ LIÊN QUAN:

As you have seen, the class is syntactically similar to the structure. You might be surprised to learn, however that the class and the structure have virtually identical capabilities. In C++, the definition of a structure has been expanded so that it can also include member functions, including constructor and destructor functions, in just the same way that a class can. In fact, the only difference between a structure and a class is that, by default, the members of a class are private but the members of a structure are public. The expanded syntax of a structure is shown here:

Như các bạn hiểu, lớp là những cấu trúc có cú pháp giống nhau. Bạn có thể thấy ngạc nhiên khi học, lớp và cấu trúc hầu như là có các chức năng giống nhau. Trong C++, sự định nghĩa cấu trúc được mở rộng đến mức nó có thể bao gồm

những thành phần hàm: hàm thiết lập và hàm phá hủy, giống như những gì một lớp có thể làm. Trên thực tế, sự khác biệt duy nhất giữa lớp và cấu trúc là: ở sự xác lập mặc định, thành phần cảu lớp là **private** thì thành phần của cấu trúc là **public.** Cú pháp mở rộng của một cấu trúc là được trình bày như sau:

```
struct type-name

{

   //public function and data members

     private:

     //private function and data members

}object-list,
```

*In fact, according to the formal C++ syntax, both **struct** and **class** create new class types. Notice that a new keyword is introduced. It is **private,** and it tells the compiler that the members that follow are private to that class.*

Trên thực tế, theo cú pháp C++ thông thường, cả **srtuct** và **class** đều tạo nên một kiểu lớp mới. Giới thiệu một từ khóa mới. Đó là **private,** và nó chỉ cho trình biên dịch các thành phần theo sau nó là riêng tư đối với lớp đó.

*On the surface, there is a seeming redundancy in the fact that structures and classes have virtually identical capabilities. Many newcomers to C++ wonder why this apparent duplication exists. In fact, it is not uncommon to hear the suggestion that the **class** keyword is unnecessary*

Bên ngoài , trên thực tế có vẻ như có sự dư thừa là lớp và cấu trúc có những khả năng như nhau. Nhiều người mới tiếp xúc với C++ tự hỏi rằng tại sao lại có sự trùng lặp rõ ràng như vậy tồn tại. Nhưng thực tế, không hiếm khi nghe thấy rằng từ khóa **class** thì không cần thiết.

The answer to this line of reasoning has both a "strong" and "weak" form. The "strong" (or compelling) reason concerns maintaining upward compatibility from C. In C++, a C-style structure is also perfectly acceptable in a C++ program. Since in C all structure members are public by default, this convenience is also maintained in C++. Further, because ***class*** *is a syntactically separate entity from* ***struct,*** *the definition of a class is free to evolve in a way that will not be compatible with a C-like structure definition. Since the two are separated, the future direction of C++ is not restricted by compatibility concerns.*

Câu trả lời cho lí do của vấn đề là có cả "mạnh mẽ" va "yếu ớt". Lý do chủ yếu(hoặc là hấp dẫn) là bao gồm việc duy trì các tính năng tương thích của C. Trong C++, cấu trúc theo phong cách của C cũng được chấp nhận hoàn toàn trong một chương trình C++. Kể từ khi trong C tất cả các thành phần cấu trúc được mặc định dùng chung, quy ước này cũng được duy trì trong C++. Hơn thế nữa, bởi vì, **class** là một thực thể có cú pháp riêng biệt với **struct,** định nghĩa của một lớp thì tự do để mở ra mà không là tính năng của C – giống như là định nghĩa của cấu trúc. Kể từ khi hai loại này được tách biệt, thì phương hướng trong tương lai của C++ không bị hạn chế bởi những mối quan tâm về tính năng tương thích.

The "weak" reason for having two similar constructs is that there is no disadvantage to expanding the definition of a structure in C++ to include member functions.

Lí do phụ giải thich cho việc có hai cấu trúc tương tự nhau là không có bất lợi nào để mở rộng định của một cấu trúc trong C++ để chứa thêm các thành phần hàm.

Although structures have the same capabilities as classes, most programmers restrict their use of structures to adhere to their C-like form and do not use them to include function members. Most programmers use the ***class*** *keyword when defining objects that contain both data and code. However, this is a stylistic matter and is subject to your own preference. (After this section, this books reserves the use of* ***struct*** *for objects that have no function members.)*

Mặc dù cấu trúc có những tính năng tương tự như lớp, nhưng hầu hết các lập trình viên hạn chế sử dụng những cấu trúc bám chặt vào C và không sử dụng chúng để chứa các thành phần hàm. Hầu hết những lập trình viên sử dụng từ khóa **class** khi định nghĩa những đối tượng bao gồm cả dữ liệu và đoạn mã. Tuy nhiên, đây chỉ là vấn đề văn phong, và là chủ đề liên quan đến sở thích riêng của mỗi người.(Sau phần này, cuốn sách này sẽ duy trì cách sử dụng cấu trúc cho đối tượng không có thành phần hàm).

*If you find the connection between classes and structures interesting, so will you find this next revelation about C++: classes are also related. In C++, a union defines a class type that can contain both function and data as members. A union like a structure in that, by default, all members are public until the **private** specifier is used. In a union, however, all data members share the same memory location (just in C). Unions can contain constructor and destructor functions. Fortunately, C unions are upwardly compatible with C++ unions.*

Nếu bạn nhận thấy sự thú vị của mối liên quan giữa lớp và cấu trúc thì bạn sẽ thấy được những điều mới về C++ : union và class cũng có mối quan hệ. Trong C++, union định nghĩa một loại lớp, nó có thể bao gồm cả hàm và dữ liệu. Một union thì giống như một cấu trúc, trong đó, ở mặc định, thì tất cả các thành phần là public mãi cho đến khi **private** được sử dụng. Trong một union, tất cả các tất cả các thành phần dữ liệu sử dụng chung một vùng nhớ (chỉ có trong C). May thay, những sự kết hợp trong C tương thích với những sự kết hợp trong C++.

Although structures and classes seem on the surface to be redundant, this is not the case with unions. In an object-oriented language, it is important to preserve encapsulation. Thus, the union's ability to link code and data allows you to create class types in which all data uses a shared location. This is something that you cannot do using a class.

Mặc dù các cấu trúc và lớp dường như có vẻ là dư thừa, điều này thì không xảy ra đối với union. Trong ngôn ngữ lập trình hướng đối tương, điều quan trọng là giữ được sự gói gọn. Theo cách đó, những khả năng của union cho phép chúng ta kết nối các đoạn mã lệnh và dữ liệu để tạo nên các loại class sử dụng các thông tin đã được chia sẻ. Đây là vài thứ mà bạn không thể sử dụng class.

*There are several restrictions that apply to unions as they relate to C++. First, they cannot inherit any other class for any other type. Unions must not have many any **static** members. They also must not contain any object that has a constructor or destructor. The union, itself, can have a constructor and destructor, through. (Virtual functions are described later in this book.)*

Có một vài hạn chế khi ứng dụng union vào trong C++. Đầu tiên, bạn không thể thừa hưởng bất kì lớp nào khác, và chúng không thể được sử dụng như là lớp cơ sở trong bất kì loại nào khác. Union không được có thành phần **static** (tĩnh). Chúng cũng không được có bất kì đối tượng nào có phương thức thiết lập hoặc phá hủy. Mặc dù, tự union có thể có phương thức thiết lập và phá hủy. Cuối cùng, union không thể có hàm thành phần ảo. (Hàm ảo sẽ được trình bày sau trong cuốn sách này).

There is a special type of union in C++ called an type of union in C++ called an anonymous union An anonymous union does not have a type name, and no variables can be declared for this sort of union. Instead, an anonymous union tells the compiler that its members will share the same memory location. However, in all other respects, the members are accessed directly, without the dot operator syntax. For example, examine this fragment:

Có một vài loại union đặc biệt trong C++ được gọi là union ẩn danh (*anonymous union)*. Một union ẩn danh thì không có tên loại, và không một tham chiếu nào có thể được khai báo theo kiểu union này. Thay vào đó, union ẩn danh chỉ cho trình biên dịch rằng những thành phần của nó sẽ sử dụng chung vùng nhớ. Tuy nhiên, trong tất cả các khía cạnh khác, những thành phần hoạt động và được đối xử giống như những tham chiếu bình thường khác. Đó là, những thành phần được truy cập một cách trực tiếp, không cần cú pháp toán tử chấm. Ví dụ, nghiên cứu đoạn chương trình sau:

```
union
{
 //an anonymous union
 int I;
 char ch[4];
 i=10;//access I and ch directly
 ch[0]='X';
}

 i=10;//truy xuat i va ch mot cach truc tiep
 ch[0]='X';
```

*As you can see, **i** and **ch** are accessed directly because they are not part of any object. They do, however, share the same memory space.*

Như bạn có thể thấy, i và ch được truy xuất một cách trực tiếp bởi vì chúng không phải là thành phần của đối tượng. Chúng hoạt động, tuy nhiên, dùng chung một vùng nhớ.

The reason for the anonymous union is that it gives you a simple way to tell the compiler that you want two or more variables to share the same memory location. Aside from this special attribute, members of an anonymous union behave like other variables.

Hàm nặc danh cung cấp cho các bạn một cách đơn giản để báo cho trình biên dịch rằng, bạn muốn hai hay nhiều tham chiếu cùng dùng chung một vùng nhớ. Ngoài tính năng đặc biệt này, tất cả các thành phần của union đều được đối xử giống như các biến khác.

*Anonymous unions have all of the restrictions that apply to normal unions plus these additions. A global anonymous union must be declared **static**. An anonymous union cannot contain private members. The names of the members of anonymous union must not conflict with identifiers within the same scope.*

Hàm nặc danh cũng có tất cả những hạn chế như các hàm nặc danh bình thường khác, và thêm một vài cái khác. Một hàm nặc danh toàn cục phải được khai báo kiểu **static.** Một hàm nặc danh không thể có các thành phần riêng tư. Tên của các thành phần trong hàm nặc danh không được trùng với những dạng khác mà không cùng phạm vi.

VÍ DỤ (*EXAMPLES)*:

*Here is a short program that uses **struct** to create a class:*

Đây là một đoạn chương trình ngắn sử dụng kiểu **struct** để tạo một lớp

```
#include "stdafx.h"

#include"iostream.h"

#include"cstring"

// su dung struct de dinh nghia mot class

struct st_type

{
```

```
     st_type(double b,char *n);
     void show();
     private:
          double balance;
          char name[40];
};
st_type::st_type(double b,char*n)
{
     balance=b;
     strcpy(name,n);
}
void st_type:: show()
{
     cout<<"Name :"<<name;
     cout<<": $"<<balance;
     if(balance<0.0)
     cout <<"***";
     cout<<"\n";
}

int main()
{
     st_type acc1(100.12,"Johnson");
     st_type acc2(-12.34,"Hedricks");
```

```
    acc1.show();
    acc2.show();

    return 0;
}
```

*Notice that, as stated, the members of a structure atr public by default. The **private** keyword must be used to declare private members.*

Chú ý rằng, các thành phần của cấu trúc được mặc định là chung. Từ khóa **private** được dùng để mô tả các thành phần riêng biệt.

*Also, notice one difference between C-like and C++-like structures. In C++, the structure tag-name als becomes a complete type name that can be used to declare objects. In C, the tag-name requires that the keyword **struct** precede it before it becomes a complete type.*

Một điểm khác giữa cấu trúc trong phong cách lập trình C và C++: trong C++, tên của cấu trúc cũng trở thành tên của đối tượng cần mô tả. Trong C, tên cấu trúc cần có từ khóa là **struct** phía trước nó để nó có thể trở thành tên của đối tượng.

Here is the same program, rewritten using a class

Đây là chương trình giống như trên nhưng sử dụng một lớp:

```
#include <iostream>
#include <string>
using namespace std;
class cl_type()
{
    double balance;
    char name[40];
    public:
```

```
            cl_type(double b, char *n);
            void show();
};
cl_type::cl_type(double b, char *n )
{
     balance *b;
     strcpy(name,n);
}
void cl_type::show()
{
     cout<<" Name : "<<name;
     cout<<" : $"<<balance;
     if(balance<0.0) cout<<"***";
     cout<<"\n";
}
int main()
{
     cl_type acc1( 100.12,"JohnSon");
cl_type acc2( -100.12,"Hedricks");

     acc1.show();
     acc2.show();

     return 0;
}
```

2. *Here is an exemple that use a union to display the binary bit pattern, byte by byte, contained within a* ***double*** *value*

Dưới đây là một ví dụ về sử dụng cấu trúc **union** để biểu diễn các bit nhị phân, từng byte một, chứa trong **double**

```
#include <iostream>

using namespace std;

union bits

{

     bits(double n);

     void show_bits();

     double d;

     unsigned char c[sizeof(double)];

};

bits::bits(double n)

{

     d=n;

}

void bits::show_bits()

{

     int i,j;

     for ( j=sizeof(double)-1l;j>=0;j--)

     {

          cout<<"Bit pattern in byte "<<j<<" :";

          for( i=128;i;i>>=1)

               if( i&c[j]) cout<<"1";
```

```
                else cout<<"0";
            cout<<"\n";
        }
}
int main()
{
     bits ob(1991.829);
     ob.show_bits();
     return 0;
}
```

Kết quả xuất ra màn hình là :

```
Bit pattern in bye 7:01000000
Bit pattern in bye 6:10011111
Bit pattern in bye 5:00011111
Bit pattern in bye 4:01010000
Bit pattern in bye 3:11100101
Bit pattern in bye 2:01100000
Bit pattern in bye 1:01000001
Bit pattern in bye 0:10001001
```

*3. Both structures and unions can have constructors and destructors. The following example shows the **strtrype** class reworks as a structure. It contains both a constructor and a destructor function.*

3. Cả cấu trúc **structures và union** có thể có phương thức thiết lập và phá hủy. Ví dụ dưới đây sẽ chỉ ra là lớp **scrtype** như một cấu trúc structure. Nó bao gồm cả

phương thức thiết lập và phương thức phá hủy.

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
struct strtype
{
     strtype(char *ptr);
     ~strtype();
     void show();
private:
     char*p;
     int len;
};
strtype::strtype(char *ptr)
{
     len=strlen(ptr);
     p=(char*)malloc(len+1);
     if(!p)
     {
          cout<<" Allocation error\n";
          exit(1);
     }
     strcpy(p,ptr);
```

```
}
strtype::~strtype()
{
     cout<<"Freeing p\n";
     free(p);
}
void strtype::show()
{
     cout<<p<< "- length= "<<len;
     cout<<"\n";
}
int main()
{
     strtype s1(" This is a test."), s2(" I like C++");
     s1.show();
     s2.show();
     return 0;
}
```

4. *This program uses an anonymous union to display the individual bytes that comprise a* ***double.(*** *This program assumes that* ***double*** *are 8 bytes long.)*

4. Đây là chương trình sử dụng cấu trúc nặc danh **union** để diễn tả các byte riêng biệt của kiểu **double** .

```
#include <iostream>
using namespace std;
```

```
int main()
{
	union
	{
		unsigned char byte[8];
		double value;
	};
	int i;
	value =859655.324;
	for(i=0;i<8;i++)
	{
		cout<<(int)byte[i]<<" ";
	}
	return 0;
}
```

*As you can see, both **value and bytes** are accessed as if they were normal variables and not part of a union. Even though they are at the same scope level as any other local variables declared at the same point. This is why a member of an anonymous union cannot have the same name as any other variable known ti its scope.*

Như các bạn thấy, cả value và bytes được truy cập như là 1 biến bình thường không phải là 1 thành phần của cấu trúc nặc danh **union.** Mặc dù chúng được công bố là 1 thành phần của cấu trúc nặc danh **union,** tên của chúng thì cùng chung 1 phạm vi và chúng dùng chung 1 địa chỉ ô nhớ. Đó là tại sao mà các thành phần của cấu trúc nặc danh **union** không được đặt trùng tên.

BÀI TẬP (*Exercises*):

*Rewrite the **stack** class presented in Section 2.1 so it uses a structure rather than a class.*

Viết lại lớp chồng đã trình bày ở mục 2.1 để nó sử dụng một cấu trúc tốt hơn một lớp.

*Use a **union** class to swap the low and high-order bytes of an integer(assuming 16-bit integers; If yours computer uses 32- bit integer. Swap the bytes of a short **int**.)*

Sử dụng một lớp **union** để hoán vị các byte thấp và cao hơn của một số nguyên (lấy số nguyên 16 bit, nếu máy tính bạn sử dụng số nguyên 32 bit, hoán vị các byte của số nguyên ngắn).

3. Explain wht an anonymous union is and how it differs from a normal union.

Giải thích một lớp kết hợp nặc danh là gì và cách phân biệt nó với một lớp kết hợp bình thường.

1.6. IN-LINE FUNCTION - HÀM NỘI TUYẾN:

Before we continue this examination of classes, a short but related digression is needed. In C++, it is possible to define functions that are not actually called but, rather, are expanded in line, at the point of each call. This is much the same way that a C-like parameterized macro works. The advantage of in-line functions is that they have no overhead associated with the function aca be executed much faster than normal functions. (Remember, the machine instructions that generate the function call and return take time each time a function is called. If there are parameters, even more time overhead is generated.)

Trước khi chúng ta tiếp tục với bài kiểm tra của khóa học, chúng ta sẽ tản mạn 1 chút bên lề.. Trong C++, một hàm có thể thực sự không dc gọi, nhất là chỉ trên 1 dòng, ngay thời điểm mỗi lần gọi. Giống như là cách thức thể hiện của các macro có tham số.

The disadvantage of in-line functions is that if they are too large and called too often, your program grows larger. For this reason, in general only short functions are declared as in-line functions.

Điều thuận lợi của các hàm nội tuyến là chúng không phải có sự kết hợp giữa hàm

gọi và hàm trả về. Điều đó có nghĩa rằng hàm nội tuyến có thể thực thi nhanh hơn các hàm bình thường.(Nhớ rằng,lời chỉ dẫn máy được hàm gọi và trả về mỗi lần hàm dc gọi.

*To declare an in-line function, simply precede the function's definition with the **inline** specifier. For example, this short program shows how to declare an in-line function.*

Điều bất cập của hàm nội tuyến là chúng quá lớn và thường được gọi nhiều lần, chương trình của bạn sẽ lớn hơn. Vì lý do, chỉ những hàm ngắn mới được mô tả như là một hàm nội tuyến.

*In this example, the function **even(),** which returns **true** if its argument is even, is declared as being in-line. This means that the line .*

Để biểu thị một hàm nội tuyến, định nghĩa hàm với từ phía trước là **inline.** Ví dụ một chương trình ngắn có sự dụng hàm nội tuyến.

```
#include <iostream>

using namespace std;

inline int even(int x)

{
     return !(x%2);

}

int main()

{
     if(even(10)) cout<<" 10 is even\n";

     if(even(11)) cout <<"11 is even\n";

return 0;

}
```

*In this example, the function **even(),** which returns **true** if its argument is even, is declared as being in-line. This means that the line*

Trong ví dụ trên, hàm **even()** sẽ trả về giá trị **true** nếu biểu thức xảy ra, dc biểu thị như là hàm nội tuyến. 2 dòng có ý nghĩa tương đương nhau:

```
if(even(10)) cout<<" 10 is even\n";

if(!(10%2)) )) cout<<" 10 is even\n";
```

*This example also points out another important feature of sung **inline:** an in-line function must be defined before it is first called. If it isn't, the compiler has no way to know that it is supposed to be expanded in-line. This is why **even()** was defined before **main().***

Ví dụ này cũng chỉ ra đặc điểm quan trọng khác trong việc sử dụng hàm **inline .** Một hàm nội tuyến cần phải được định nghĩa trước lần gọi đầu tiên. Nếu không, Trình biên dịch sẽ không hiểu. Đó là lý do tại sao hàm **even()** được định nghĩa trước hàm **main().**

The advantage of using rather than parameterized macros is towfold. First, it provides amore structured way to expand short functions in line. For example, when you are creating a parameterized macro, it's easy to forget that extra parentheses are often needed to ensure proper in-line expansion in every case. Using in-line functions prevents such problems.

Việc sử dụng **inline** thì tốt hơn là dùng macro có tham số ở 2 điểm. Thứ nhất, nó cung cấp nhiều cách thiết lập cấu trúc để mở rộng các hàm ngắn.. Ví dụ như, khi chúng ta đang tạo một macro có tham số, chúng ta sẽ dễ dàng quên them dấu ngoặc đơn mở rộng thường cần để chắc chắn sự khai triển nội tuyến dc xảy ra trong mọi trường hợp. Sử dụng hàm nội tuyến sẽ tránh được vấn đề này.

*Second, an in-line function might be able to be optimized more thoroughly by the compiler than a macro expansion. In any event, C++ programmers virtually never use parameterized macros, instead relying on **inline** to avoid the overhead of a function call associated with a short function.*

Thứ 2, một hàm nội tuyến sẽ được trình biên dịch tốt hơn so với macro.. Trong mọi trường hợp, người lập trình C++ thường không bao giờ sử dụng hàm macro có tham số, thay vào đó người ta sử dụng **inline** để tránh trường hợp quá tải khi khởi hàm kết hợp với hàm ngắn.

*It is important to understand that the **inline** spectifier is a request, not a command, to the compiler. If, for various reasons, the compiler is unable to fulfill the request, the function is compiled as a normal function and the **inline***

request is ignored

Cần hiểu rõ **inline** là một yêu cầu chứ không phải là một lệnh đối với trình biên dịch. Với nhiều lý do khác nhau, trình biên dịch sẽ không thể hoàn thành yêu cầu, hàm sẽ được biên dịch như hàm bình thường , và **inline** yêu cầu sẽ bị hủy bỏ.

*Depending upon your compiler, several restrictions to in-line functions may apply. For example, some compilers will not in-line a function if it contains a **static** variable, a loop statement, a **switch** or a **goto,** or if the function is recursive. You should check your compiler's user manual for specific restrictions to in-line functions that might affect you.*

Dựa vào trình biên dịch của bạn, có một vài hạn chế của hàm nội tuyến.ví dụ như, một vài trình biên dịch sẽ không cho hàm nội tuyến là một hàm nếu như trong hàm có kiểu dữ liệu **static,** vòng lặp, lệnh **switch và lệnh goto,** hoặc là các hàm đệ quy. Bạn nên kiểm tra khả năng sử dụng của trình biên dịch của bạn để các hạn chế của hàm nội tuyến không ảnh hưởng tới bạn.

CÁC VÍ DỤ

*Any type of function can be in-line, including function that are members of classes. For example, here the member function **divisible()** is in-lined for fast execution. (The function returns **true** if its argument can be evenly divided by its second.)*

Mọi loại hàm đều có thể là hàm nội tuyến, bao gồm nhưng hàm mà là thành phần của các lớp. Ví dụ, dưới đây là hàm **divisible()** được dùng nội tuyến cho việc thao tác nhanh.(Hàm trả về **true** nếu như biểu thức đầu tiên của nó xảy ra)

```
#include <iostream>

using namespace std;

class samp

{

          int i,j;

     public:

               samp(int a,int b);
```

```
                int divisible();
};
samp::samp(int a,int b)
{
          i=a;
          j=b;
}
inline int samp::divisible()
{
          return !(i%j);
}
      int main()
{
          samp ob1(10,2),ob2(10,3);
          if(ob1.divisible())  cout<<" 10  chia  het  cho
2\n";
          if(ob2.divisible()) cout<<" 10 chia het cho 3
\n";
          return 0;
}
```

2. *It is perfrctly permissible to in-line an overloaded function. For example, this program overloads **min()** three ways. Each way is also declared as **inline.***

Có lẽ sẽ hoàn hảo hơn khi sử dụng hàm nội tuyến để dùng cho các hàm nạp chồng. Đây là ví dụ sử dụng hàm nạp chồng **min()** trong 3 cách. Mỗi cách là một hàm nội tuyến.

```
#include <iostream>
using namespace std;
inline int min(int a,int b)
{
     return a<b?a:b;
}
inline long min(long a,long b)
{
     return a<b?a:b;
}
inline double min(double a,double b)
{
     return a<b?a:b;
}
int main()
{
     cout<<min(-10,10)<<"\n";
     cout<<min(-10.01,100.02)<<"\n";
     cout<<min(-10L,12L)<<"\n";
     return 0;
}
```

BÀI TẬP (*Exercises*):

1. *In Chapter 1 you overloaded the **abs ()** functuion so that it could find the absolute of integers, long integers, and **doubles.** Modify that program so that those functions are expanded.*

Trong chương 1, bạn đã định nghĩa chồng hàm **abs()** để tìm giá trị tuyệt đối của số nguyên, số nguyên dài và số thực dài. Hãy sửa đổi chương trình đó để các hàm được mở rộng nội tuyến.

2. *Why might the following function not be in-lineed by your compiler?*

Tại sao hàm sau không thể dịch nội tuyến bởi trình biên dịch?

```
void f1()
{
          int I;
          for(i=0; i<10; i++)    cout << i;
}
```

1.7. AUTOMATIC IN-LINING - HÀM NỘI TUYẾN TỰ ĐỘNG:

*If a member function's definition is short enough, the definition can be included inside the class declaration. Doing so causes the function to automatically become an i-line function, if possible. When a function is defined within a class declaration, the **inline** keyword is no longer necessary.(However, it is not an error to use it in this situation.) For example, the **divisible ()** function from the preceding section can be automatically in-line as shown here.*

Nếu một thành phần của hàm được định nghĩa quá ngắn, thì định nghĩa hàm đó có thể nằm bên trong của sự định nghĩa 1 lớp. Làm như vậy nhiều khả năng sẽ tự động tạo ra một hàn nội tuyến. Khi một hàm được định nghĩa trong khai báo một lớp, từ khóa **inline** thì không cần thiết. (Tuy nhiêncũng không có lỗi nếu sử dụng

nó trong trường hợp này). Chẳng hạn như hàm **divisible()** ở phần trước có thể tự động nội tuyến như sau:

```
#include <iostream>
using namespace std;
class samp
{
     int i,j;
public:
     samp(int a,int b);
     int divisible()
     {
          Return (!(i%j));
};
samp::samp(int a,int b)
{
     i=a;
     j=b;
}
int main()
{
     samp ob1(10,2),ob2(10,3);

     //cau lenh dung
     if(ob1.divisible()) cout<<" 10 chia het cho 2\n";

     //cau lenh sai
```

```
    if(ob2.divisible()) cout<<" 10 chia het cho 3 \n";


    return 0;
}
```

*As you can see, the code associated with **divisible()** occurs inside the declaration for the class samp. Further notice that no other definition of **divisible( )** is needed-or permitted. Defining **divisible( )** inside **samp** cause it to be made into an in-line function automatically.*

Như bạn thấy đấy, đoạn mã kết hợp với hàm **divisible()** xuất hiện bên trong khai báo lớp **samp**. Hơn nữa, cũng cần chú ý rằng không cho phép cách định nghĩa khác cho hàm **divisible()**. Định nghĩa hàm này bên trong lớp **samp** khiến nó tự động trở thành một hàm nội tuyến.

When a function define inside the class declaration cannot be made into an in-line function (because a restriction has been violated), it is automatically made into a regular function.

Khi một hàm được định nghĩa bên trong một khai báo lớp thì không thể được làm bên trong một hàm nội tuyến (do vi phạm một hạn chế), nó sẽ tự động trở thành một hàm bình thường.

*Notice how **divisible( )** is defined within **samp**, paying particular attention to the body. It occurs all on one line. This format is very common in C++ programs when a function is declared within a class declaration. It allows a declaration to be more compact. However, the samp class could have been written like this:*

Chú ý cách định nghĩa hàm **divisible()** trong lớp **samp,** đặc biệt chú ý đến phần thân hàm. Nó xuất hiện tất cả trên một dòng. Định dạng này rất phổ biến trong các chương trình C++ khi khai báo một hàm bên trong một khai báo lớp. Nó cho phép việc khai báo chặt chẽ hơn. Tuy nhiên, lớp **samp** cũng có thể được viết như sau:

```
class samp
{
```

```
            int i, j;
      public:
            samp(int a, int b);

            //ham divisible() duoc dinh nghia o day va tu
dong noi tuyen
            int invisible()
            {
                  return !(i%j);
            }
};
```

*In this version, the layout of **devisible( )** uses the more-or-less standard indentation style. From the compile's point of view, there is no defference between the compact style and the standard style. However, the compact style is commonly found in C++ programs when short functions are defined inside a class definition.*

Trong phiên bản này, cách bố trí hàm **divisible()** sử dụng nhiều hoặc ít hơn kiểu thụt dòng chuẩn. Từ góc nhìn của trình biên dịch thì không có sự khác biệt nào giữa kiểu kết hợp này với kiểu chuẩn. Tuy nhiên, kiểu kết hợp thường được tìm thấy hơn trong các chương trình C++, khi các hàm ngắn được định nghĩa bên trong một định nghĩa lớp.

The same restrictions that apply to "normal" in-line functions apply to automatic in-line functions apply to automatic in-line functions within a class declaration.

Các hạn chế như nhau này chấp nhận các hàm nội tuyến "bình thường" để tạo thành các hàm nội tuyến tự động trong một khai báo lớp.

CÁC VÍ DỤ:

1. *Perhaps the most common use of in-line functions defined within a class is to define constructor and destructor functions. For example, the samp class can more efficiently be defined like this:*

Có lẽ, cách sử dụng phổ biến nhất của các hàm nội tuyến được định nghĩa bên trong một lớp là định nghĩa các hàm dựng và phá hủy. Ví dụ như lớp **samp** có thể được định nghĩa một cách hiệu quả hơn như sau:

```
#include <iostream.h>
class samp
{
        int i, j;
        public:
          samp(int a, int b)
          {
               i=a;
               j=b;
          }
          int divisible()
          {
               return !(i%j);
          }
};
```

Việc định nghĩa hàm **samp()** bên trong lớp **samp** là tốt nhất không nên định nghĩa theo cách khác.

2. Sometimes a short function will be included in a class declaration even

though the automatic in-lining feature is of little or no value. Consider this class declaration:

Đôi khi một hàm ngắn chứa trong một khai báo lớp dù cho không có giá trị nào nội tuyến. Xem đoạn khai báo lớp sau:

```
class myclass{
     int i;
public:
     myclass(int n)
     {
          i=n;
     }
     void show()
     {
          cout << i;
     }
};
```

Ở đây, hàm **show()** tự động trở thành hàm nội tuyến. Tuy nhiên, như bạn biết đấy, các toán tử nhập xuất kết nối đến CPU hay các toán tử nhớ chậm đến mức ảnh hưởng loại trừ đến hàm được gọi chồng cơ bản đã bị mất. Mặc dù vậy, trong các chương trình C++, ta vẫn thường thấy các hàm nhỏ của loại này được khai báo bên trong một lớp để thuận tiện hơn và không gây sai xót.

BÀI TẬP:

1. Chuyển đổi lớp **stack** ở ví dụ 1, mục 2.1 để nó sử dụng hàm nội tuyến tự động ở vị trí thích hợp.
2. Chuyển đổi lớp **strtype** ở ví dụ 3, mục 2.2 để nó sử dụng hàm nội tuyến tự động.

CÁC KỸ NĂNG KIỂM TRA:

Ở điểm này, có thể bạn nên làm các bài tập và trả lời các câu hỏi sau:

Phương thức thiết lập là gì? Phương thức phá hủy là gì? Và chúng được thực thi khi nào?

Khởi tạo lớp **line** để vẽ một đường trên màn hình. Lưu độ dài của dòng vào một biến kiểu nguyên riêng là **len.** Ta có phương thức thiết lập của lớp **line** với một tham số là độ dài của dòng. Nếu hệ thống của bạn không hỗ trợ về đồ họa thì có thể biểu diễn dòng bằng các dấu *.

Chương trình sau cho kết quả gì?

```
#include <iostream.h>

int main()

{

      int i = 10;

      long l = 1000000;

      double d = -0.0009;


      cout << i << "  " << l << "  " << d;

      cout << "\n";


      return 0;

}
```

Tạo một lớp con khác kế thừa từ lớp **area_cl** ở mục 2.3, bài tập 1. Gọi lớp này là **cylinder** và tính diện tích xung quanh của hình trụ. Gợi ý: diện tích xung quanh của hình trụ là *2 * pi * R^2 + pi * D * height*.

Thế nào là một hàm nội tuyến? Ưu điểm và khuyết điểm của nó?

Chỉnh sủa lại chương trình sau để cho tất cả hàm thành phần đều tự động nội tuyến:

```
#include < iostream.h>
class myclass
{
          int i, j;
     public:
          myclass(int x, int y);
          void show();
};

myclass :: myclass(int x, int y)
{
     i= x;
     j= y;
}

void myclass :: show()
{
     cout << i << "  " << j << "\n";
}

int main()
{
     myclass count(2, 3);
     count.show();
```

```
    return 0;
}
```

Sự khác biệt giữa một lớp và một cấu trúc là gì?
Đoạn khai báo sau có hợp lệ không?

```
union
{
    float f;
    unsigned int bits;
};
```

Cunulative Skiills Check

Phần này kiểm tra bạn đã tích lũy như thế nào trong chương này và chương trước:

1. Tạo một lớp tên là **prompt**. Tạo hàm dựng một chuỗi chỉ dẫn với các lựa chọn. Hàm dựng xuất một chuỗi và sau đó nhập vào một số nguyên. Lưu lại giá trị đó trong một biến cục bộ gọi là **count.** Khi một đối tượng kiểu **prompt** bị phá hủy thì rung chuông ở điểm cuối cùng nhiều bằng số lần bạn nhập vào.

2. Trong chương 1, bạn đã viết một chương trình chuyển đổi từ feet sang inches. Bây giờ hãy tạo một lớp làm công việc giống như vậy. Lớp này lưu trữ giá trị feet và giá trị inch tương ứng. Xây dựng phương thức thiết lập cho lớp với tham số feet và một phương thức thiết lập trình bày giá trị inch tương ứng.

3. Tạo một lớp tên là **dice** chứa một biến nguyên cục bộ. Tạo hàm **roll()** sử dụng bộ sinh số ngẫu nhiên chuẩn, là hàm **rand()**, để phát sinh một số trong khoảng 1 đến 6. Sau đó dùng hàm **roll()** để trình bày giá trị đó.

CHAPTER 3

A CLOSER LOOK AT CLASSES - Xét Kỹ Hơn Về Lớp

CHAPTER OJECT

3.1 ASSIGING OBJECT.

Gán Đối Tượng.

3.2 PASSING OBJECT TO FUNCTION.

Truyền Các Đối Số Cho Các H‡m.

3.3 RETURNING OBJECT FROM FUNCTIONS.

Trả Đối Tượng Về Từ H‡m.

3.4 AN INTRODUCTION TO FRIEND FUNCTIONS SKILLS CHECK.

Giới Thiệu Các H‡m Friend.

In this chapter you continue to explore the class. You will learn about assigning objects, passing objects to functions, and return objects from functions. You will also learn about an important new type of function: the friend

Trong chương này bạn sẽ tiếp tục học về lớp. Bạn sẽ học về việc gán các đối tượng, chuyển các đối tượng qua các hàm, và trả lại các đối tượng từ các hàm. Bạn sẽ được hoc thêm một loại hàm quan trọng nữa đó là : Hàm bạn.

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Trước khi tiếp tục bạn nên trả lời chính xác các câu hỏi sau đây và làm các bài tập sau.

Given the following class, what are the names of its cuntructor and destructor function?

Trong lớp sau, đâu là tên hàm tạo và hàm hủy?

```
Class widget
{
     int x, y;
public:
     //…fill in constructor and destructor functions
};
```

When is a constructor function called? When is a destructor function called?

. Khi nào hàm tạo được gọi? Khi nào hàm hủy được gọi?

Given the following base class, show how it can be inherited by a deliverd class called Mars

Cho lớp cơ bản sau, hãy chỉ ra bằng cách nào nó có thể kế thừa lớp gốc mang tên Sao Hỏa.

```
class planet
{
     int moons;
     double dist_from_sun;
     double dismeter;
     double mass;
public:
 //…
};
```

There are two way to cause a function to be expanded in line. What are they?

Có hai cách để làm cho một hàm được mở rộng ra trong ranh giới. Chúng là hai cách nào?

Give two possible restrictions to in-line functions.

Hãy đưa ra hai cách hạn chế khả thi đối với các hàm trong giới hạn.

*Given the following class, show how an object called **ob** that passes the value 100 to **a** and to **c** would be declared.*

Cho lớp sau, hãy cho biết bằng cách nào đối tượng mang tên **ob** đã gán giá trị 100 cho a và X cho c như đã tuyên bố.

```
class sample
{
     int a;
     char c;
public:
sample(int x, char ch) { a = x; c = ch;}
};
```

1.1. Assigning Objects - Gán đối tượng:

*One object can be assigned to another provided that both objects are of the same type. By default, when one object is assigned to another, a bitwise copy of all data members is made. For example, when an object called **o1**'s is assigned to another object called **o2**, the contents of all of **o1**'s data are copied into the equivalent members of **o2**. This is illustrated by the following program:*

Một đối tượng có thể gán giá trị cho một đối tượng khác khi hai đối tượng này có cùng một kiểu. Mặc định, khi một đối tượng được gán cho đối tượng khác, thì một bản copy tất cả các thành phần của đối tượng đó sẽ được tạo ra. Ví dụ, khi gán đối tượng **o1** cho **o2**, thì nội dung bên trong dữ liệu của **o1** sẽ được copy sang cho dữ liệu của **o2**. Sau đây là chương trình minh họa:

```
// An example of object assignment

#include <iostream>

Using namespace std;


Class myclass

{

       int a, b;

     public:
```

```
            void set(int i, int j) {a = i; b = j; }
            void show() {cout <<a << ” “ << b <<”\n“;}
};

int main()
{
     Myclass o1, o2
     o1.set(10, 4);
     // assign o1 to o2
     o2 = o1;

     o1.show();
     o2.show();

     return 0;
}
```

*Here, object **o1**, has its member variables a and b set to the values 10 and 4, respectively. Next, **o1** is assigned to **o2**. This causes the current value of **o1.a** to be assigned to **o2.a** and **o1.b** to be assigned to **o2.b**. Thus, when run, this program displays*

Ở đây, đối tượng o1, với các biến thành viên là a và b đã lần lượt được gán giá trị 10 và 4. Tiếp đó, o1 gán cho o2. Việc này gây ra việc các giá trị o1.a được gán cho o2.a và o1.b được gán cho o2.b. Vì vậy, khi chạy chương trình, kết quả sẽ là:

```
10  4
10  4
```

*Keep in mind that an assignment between two objects simply makes the data in those objects identical. The two objects are still completely separate. For example, after the assignment, calling **o1.set( )** to set the value of **o1.a** has no effect on **o2** or it's a value.*

Hãy nhớ rằng việc gán giá trị giữa hai đối tượng đơn giản là nhân đôi dữ liệu ra 1 đối tượng khác. Hai đối tượng này là hoàn toàn khác nhau. Ví dụ, sau khi gán, ta gọi o1.set( ) để thao tác trên giá trị o1.a thì hoàn toàn không ảnh hưởng gì đến giá trị của đối tượng o2.

VÍ DỤ (EXAMPLES):

Only objects of the same type can be used in an assignment statement. If the objects are not of the same type, a compile-time error is reported. Further, it is not sufficient that the types just be physically similar-their type names must be the same. For example, this is not a valid program:

Chỉ có những đối tượng cùng kiểu mới sử dụng được lệnh gán. Nếu các đối tượng không cùng một kiểu, thì lỗi trong quá trình biên dịch sẽ xảy ra. Hơn nữa nó không đủ quyền hạn khi chỉ có phần thân là giống nhau mà tên kiểu cũng phải giống nhau. Ví dụ chương trình sau đây là sai:

```
// This program has an error
#include <iostream>
using namespace std;

     class myclass
     {
          int a, b;
     public:
          void set(int i, int j) { a = i; b = j; }
          void show(){cout << a <<" " << b <<"\n"; }
     };
```

```
/* This class is similar to myclass but uses a
different class name and thus appears as a different
type to the compiler.
*/

     Class yourclass
     {
          int a, b;
     public:
          void set( int i, int j) { a = i; b = j; }
          void show( ) {cout<<a<<" "<<b<<"\n";}
     };

     int main()
     {
          myclass o1;
          yourclass o2;

          o1.set(10, 4);

          o2 = o1; //ERROR, objects not of same type

          o1.show();
          o2.show();

          return 0;
```

```
}
```

In this case, even though **myclass** and **yourclass** are physically the same, because they have different type names, they are treated as differing types by the conmpiler.

Trong trường hợp này, mặc dù **myclass** và **yourclass** giống nhau về phần thân, nhưng chúng có tên kiểu khác nhau, và chúng sẽ được trình biên dịch coi như hai kiểu.

*2. It is important to understand that all data members of one object are assigned to another when an assignment is performed. This includes compound data such as arrays. For example, in the following version of the **stack** example, only **s1** has any characters actually pushed onto it. However, because of the assignment, **s2**'s **stck** array will also contain the characters **a, b** and **c**.*

2. Quan trọng là bạn phải hiểu rằng tất cả dữ liệu thành viên của một đối tượng sẽ dược gán cho các thành phần của đối tượng khác khi việc gán được thực hiện. Việc này bao gồm kiểu dữ liệu đa hợp như mảng. Ví dụ, trong chương trình sau của **stack**, chỉ s1 thực sự có các ký tự dược đẩy vào. Tuy nhiên bởi vì phép gán, stck của đối tượng s2 cũng chứa những ký tự như a, b, c.

```
#include <iostream>
using namespace std;
#define SIZE 10
// Declare a stack class for characters
Class stack
{
     char stck[SIZE]; //holds the stack
     int tos; // index of the top stack
public:
```

```
     stack(); // constructor
     void push(char ch); // push character on stack
     char pop(); // pop character from stack
};

// Initialize the stack
stack::stack()
{
     Cout << "Constructor a stack\n";
     tos=0;
}
// Push a character
void stack::push(char ch)
{
     if(tos==SIZE)
     {
          cout << "Stack is full\n";
          return;
     }
     stck[tos] = ch;
     tos++;
}

// Pop a character
char stack::pop()
```

```
{
     if(tos==0)
     {
          cout << "Stack is empty\n";
          return; //Return null on empty stack
     }
     tos--;
     return stck[tos];
}

int main()
{
/*
Create two stacks that are automatically initialized
*/
     stack s1, s2;
     int i;

     s1.push('a');
     s1.push('b');
     s1.push('c');

//clone s1
     s2 = s1; //now s1 and s2 are indentical
```

```
     for(i=0; i<3; i++)
          cout << ”Pop s1: “ << s1.pop() << ”\n”;
     for(i=0; i<3; i++)
          cout << ”Pop s2: “ << s2.pop() << ”\n”;

     return 0;
}
```

*3. You must exercise some case when assigning one object to another. For example, here is the **strtype** class developed in Chapter 2, along with a short **main( )**. See if you can find an error in this program.*

3. Bạn phải thực hiện trong vài trường hợp khi gán đối tượng này cho đối tượng khác. Ví dụ, ở đây có lớp **strtype** đã được nói đến trong chương hai, với hàm main( ) ngắn gọn. Bạn hãy tìm lỗi sai trong chương trình sau

```
// This program contains an error
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype
{
     char *p;
     int len;
public:
```

```
        strtype(char *ptr);
        ~strtype();
        void show();
};

strtype::strtype(char *ptr)
{
        len = strlen(ptr);
        p = (char *) malloc(len+1);
        if(!p)
        {
                cout << "Allocation error\n";
                exit(1);
        }
        strcpy(p, ptr);
}

strtype::~strtype()
{
        cout << "Freeing p\n";
        free(p);
}

void strtype::show()
{
```

```
        cout << p << " - length: " << len;
        cout << "\n";
}

int main()
{
        strtype s1("This is a test"), s2("I like C++");

        s1.show();
        s2.show();

        //assign s1 to s2 - - this generates an error
        s2 = s1;

        s1.show();
        s2.show();

        return 0;
}
```

*The trouble with this program is quite insidious. When **s1** and **s2** are created, both allocate memory to hold their respective strings. A pointer to each object's allocated memory is stored in **p**. When a **strtype** object is destroyed, this memory is released. However, when **s1** is assigned to **s2**, **s2**'s **p** now points to he same memory as **s1**'s **p.** Thus, when these objects are destroyed, the memory pointed to by **s1's p** is freed twice and the memory originally pointed to by **s2's p** is not freed at all.*

Lỗi của chương trình này khá là khó thấy. Khi s1 and s2 được tạo ra, cả hai đều được cấp phát vùng nhớ để lưu lại chuỗi của chúng. Một con trỏ đến ô nhớ đã cấp phát cho đối tượng lưu trong p. Khi đối tượng strtype bị phá hủy, vùng nhớ này cũng được giải phóng. Tuy nhiên, khi s1 được gán cho s2, thì con trỏ p của s2 cũng trỏ về nơi mà con trỏ p của s1 đang nắm giữ. Vì vậy, khi những đối tượng này bị phá hủy, vùng nhớ được trỏ bởi con trỏ p của s1 được giải phóng đến hai lần, và vùng nhớ ban đầu được trỏ bởi con trỏ của s2 thì không được giải phóng hoàn toàn.

While benign in this context, this sort of the problem occurring in a real program will cause the dynamic allocation system to fail, and possibly even cause a program crash. As you can see from the preceding example, when assigning one object to another, you must make sure you aren't destroying information that may be needed later.

Trong tình huống tốt, một loại lỗi sẽ xảy ra trong chương trình thực khiến cho sự cấp phát động của hê thống bị trục trặc, và thậm chí có thể gây ra một sự hư hại chương trình. Như bạn đã thấy ở ví dụ trước, khi gán một đối tượng này cho đối tượng khác thì bạn cần đảm bảo rằng bạn không hủy hoại thông tin mà có thể cần dùng sau đó.

Exercises

What is wrong with the following fragment?

Có gì sai trong đoạn chương trình sau

```
//This program has an error

#include <iostream>

using namespace std;


class c11

{

     int i, j;

public:

     c11(int a, int b) { i = a; j = b; }
```

```
        //……
};

class c12
{
        int i, j;
public:
        c12(int a, int b) { i = a; j = b; }
        //……
};

int main()
{
        c11 x(10, 22);
        c12 y(0, 0);
        x = y;

        //……
}
```

*Using the **queue** class that you created for Chapter 2, Section 2.1, Exercise 1, show how one queue can be assigned to another.*

Sử dụng lớp **queue** mà bạn đã tạo ra trong chương hai, mục 2.1, bài tập 1, để xem coi một đối tượng queue được gán cho đối tượng khác như thế nào.

*If the **queue** class from the preceding question dynamically allocates memory to hold the queue, why, in this situation, can one queue not be assigned to another?*

Nếu như lớp queue trong câu hỏi trước cấp phát bộ nhớ động để giữ queue, thì tại sao trong tình huống này, một đối tượng queue không thể được gán như vậy?

1.2. PASSING OBJECTS TO FUNCTIONS – Truyền các đối tượng cho hàm:

Objects can be passed to functions as a arguments in just the same way that other types of data are passed. Simply declare the function's parameter as a class type and then use an object of that class as an argument when callings the function. As which other of data, by default all objects are passed by value to a function

Các đối tượng có thể được chuyển tiếp qua hàm như một đối số theo cách mà các kiểu dữ liệu khác được chuyển qua. Nói một cách đơn giản là các tham số của hàm là một kiểu lớp và sau đó dùng một đối tượng của lớp mà được coi như tham số khi gọi hàm. Những phần của dữ liệu mặc định sẽ được chuyển qua hàm bởi giá trị.

Example

Here is a short example that passes an object to a function:

Đây là một ví dụ ngắn của việc chuyển một đối tượng đến hàm

```
#include <iostream>

using namespace std;

class samp

{

        int i;

public:

        samp(int n) { i = n; }

        int get_i() { return i; }
```

```
};
//Return square of o.i
int sqr_it(samp o)
{
     return o.get_i() * o.get_i();
}
int main()
{
        samp a(10), b(2);
        cout << sqr_it(a) << "\n";
        cout << sqr_it(b) << "\n";

        return 0;
}
```

*This program creates a class called **samp** that contains one integer variable called **i**. The function **sqr_it( )** takes an argument of type **samp** and returns the square of that object's **i** value. The output from this program is 100 followed by 4.*

Chương trình này tạo ra một lớp mang tên **samp** chứa một biến nguyên mang tên **i**. Hàm **sqr_it( )** mang một đối số kiểu **samp** và trả về giá trị **i** của đối tượng đó. Giá trị xuất ra của chương trình là 100 theo sau 4.

As stated, the default method of parameter passing in C++, including objects, is by value. This means that a bitwise copy of the argument is made and it is this copy that is used by the function. Therefore, changes to the object inside the function do not effect the calling object. This is illustrated by the following example:

Như đã nói, phương pháp mặc định của chuyển tiếp tham số trong C++, bao gồm các đối tượng, như là giá trị. Có nghĩa là một bản copy của đối số sẽ được tạo ra và bản copy này được sử dụng như hàm. Vì vậy, việc thay đổi hàm bên trong đối

tượng thì không ảnh hưởng đến việc gọi đối tượng. Điều này được minh họa bởi ví dụ sau

```
/*
Remember, objects, like other parameters, are passed by value. Thus changes to the parameter inside a function have no effect on the object used in the call.
*/
#include <iostream>
using namespace std;

class samp
{
        int i;
public:
        samp(int n) { i = n; }
        void set_i(int n) { i = n; }
        int get_i() {return i; }
};

/* Set o.i to its square. This has no effect   on the object used to call sqr_it(), however
*/
void sqr_it(samp o)
{
        o.set_i(o.get_i() * o.get_i());

        cout << "Copy of has I value of " << o.get_i();
```

```
        cout << "\n";
}

int main()
{
        samp a(10);

        sqr_it(a); //passed by value

        cout << "But, a.i is unchanged in main: ";
        cout << a.get_i(); //displays 10

        return 0;
}
```

The output displayed by this program is

Kết quả hiển thị của chương trình là

```
Copy of a has i value of 100
But, a.i is unchanged in main: 10
```

As with other types of variables, the address of an object can be passed to a function so that the argument used in the call can be modified by the function. For example, the following version of the program in the preceding example does, indeed, modify the value of the object whose address is used in the call to ***sqr_it( )****.*

Như các kiểu biến khác, địa chỉ của một đối tượng có thể được chuyển đến một hàm như một đối số được dùng trong việc gọi điều chỉnh hàm. Ví dụ, trong chương trình ở ví dụ trước đã thực hiện, thực vậy, điều chỉnh giá trị của đối tượng mà địa chỉ của nó được dùng gọi hàm **sqr_it( )**.

```
/*
Now that the address of an object is passed to
sqr_it(), the function can modify the value of the
argument whose address is used in the call
*/
#include <iostream>
using namespace std;

class samp
{
        int i;
public:
        samp(int n) { i = n; }
        void set_i(int n) { i = n; }
        int get_i() {return i; }
};

/*
Set o.i its square. This affects the calling arhument
*/
void sqr_it(samp *o)
{
        o->set_i(o->get_i() * o->get_i());

        cout<<"Copy of a has i value of "<< o->get_i();
        cout << "\n";
```

```
}

int main()
{
        samp a(10);
        sqr_it(&a); // pass a's address to sqr_it()

        cout << "Now, a in main() has been changed: ";
        cout << a.get_i(); // displays 100

        return 0;
}
```

This program now displays the following output

Chương trình lúc này xuất ra:

Copy of a has i value of 100

Now, a in main() has been changed: 100

When a copy of an object is made when being passed to a function, it means that a new object comes into existence. Also, when the function that the object was passed to terminates, the copy of the argument is destroyed. This raises two questions. First, is the object's constructor called when the copy is made? Second, is the object's destructor called when the copy is destroyed? The answer may, at first, seem surprising.

Khi một bản copy một đối tượng được tạo ra thì được chuyển tiếp đến hàm, nghĩa là một đối tượng đến trong sự tồn tại. Hơn nữa,khi hàm mà đối tượng được chuyển qua thì kết thúc, bản copy của đối số bị phá hủy. Việc này làm nảy sinh ra 2 câu hỏi. Một là, hàm tạo được gọi khi bản copy được tạo ra? Hai là, hàm hủy được gọi khi bản copy bị hủy? Câu trả lời có thể là ngay lúc đầu tiên, dường như

rất ngạc nhiên phải không.

When a copy of an object is made to be used in a function call, the constructor function is not called. The reason for this is simple to understand if you think about it. Since a constructor function is generally used to initialize some aspect of an object, it must not be called when making a copy of an already existing object passed to a function. Doing so would alter the contents if the object. When passing an object to a function, you want the current state of the object, not its initial state.

Khi một bản copy của một đối tượng được tạo ra để sử dụng trong hàm, thì hàm đối tượng không được gọi. Lý do của việc này thật đơn giản để hiểu nếu bạn nghĩ về nó. Từ lúc hàm tạo được dùng để khởi tạo một khía cạnh của đối tượng, nó phải được gọi khi tạo ra một bản copy của một đối tượng tồn tại tùy biến qua hàm. Việc làm này sẽ biến đổi nội dung của một đối tượng. Khi chuyển mốt đối tượng đến hàm, bạn muốn trạng thái hiện tại của đối tượng chứ không phải trạng thái khởi tạo của đối tượng.

However, when the function terminates and the copy is destroyed, the destructor function is called. This is because the object might perform some operation that must be undone when it goes out of scope. For example, the copy may allocate memory that must be released.

Tuy nhiên, khi hàm kết thúc và bản copy bị hủy, hàm hủy sẽ được gọi. Vì đối tượng thi hành một số thuật toán mà nó chưa làm khi nó vượt khỏi tầm vực. Ví dụ, bản copy có thể cấp phát vùng nhớ thì phải được giải phóng.

To summarize, when a copy of an object is created because it is used as an argument to a function, the constructor function is not called. However, when the copy is destroyed (usually by going out of scope when the function returns), the destructor function is called.

Tóm lại, khi một bản copy của một đối tượng được tạo ra bởi vì nó được dùng như một đối số với hàm, hàm tạo không được gọi. Tuy vậy, khi bản copy bị phá hủy (Thông thường là khi hàm trả về vì vượt quá tầm vực), hàm hủy sẽ được gọi.

The following program illustrates the preceding discussion:

Chương trình sau minh họa cho phần đã viết ở trên:

```
#include <iostream>
using namespace std;

class samp
{
     int i;
public:
     samp(int n) {
          i = n;
          cout << "Constructing\n";
     }
     ~samp() { cout << "Destructing\n";}
     it get_i() { return i;}
};

//Return square of oi
int sqr_it(samp o)
{
     return o.get_i() * o.get_i();
}

int main()
{
     samp a(10);
```

```
    cout << sqr_it(a) << "\n";

    return 0;
}
```

This function displays the following

Hàm này sẽ hiển thị kết quả sau

```
Constructing
Destructing
100
Destructing
```

*As you can see, only one call to the constructor function is made. This occurs when a is created. However, two calls to the destruction are made. One is for the copy created when a is passed to **sqr_it( )**. The other is for **a** itself.*

Như bạn thấy, chỉ một lời gọi hàm tạo được tạo ra. Việc này xảy ra khi a được tạo ra. Tuy nhiên, hai lời gọi hàm hủy được tạo ra. Một cho việc tạo ra bản copy khi a được chuyển tiếp đến hàm **sqr_it( )**. Cái kia là cho chính bản thân **a.**

The fact that the destructor for the object that is the copy of the argument is executed when the function terminates can be a source of problems. For example, if the object used as the argument allocates dynamic memory and frees that memory when destroyed, its copy will free the same memory when its destructor is called. This will leave the original object damaged and effectively useless. (See exercise 2, just ahead in this section, for an example.) It is important to guard against this type of error and to make sure that the destructor function of the copy of an object used in an argument does not cause side effects that alter the original argument.

Sự thật, hàm hủy cho một đối tượng mà bản copy được thực thi khi hàm kết thúc có thể là nguồn gốc của những rắc rối này. Ví dụ, nếu đối tượng được dùng như là đối số thì cấp phát bộ nhớ động và giải phóng bô nhớ khi phá hủy, bản copy của

nó cũng sẽ giải phóng cùng một bộ nhớ khi hàm hủy của nó được gọi. Việc này sẽ làm cho đối tượng ban đầu bị hư hại và những ảnh hưởng không tốt. (Hãy nhìn bài tập 2 chỉ phần đầu là một ví dụ.) Nó rất quan trọng trong việc đảm bảo kiểu này khỏi lỗi và chắc rằng hàm hủy của một đối tượng copy được sử dụng trong một đối số không gây ra ảnh hưởng đến đối tượng ban đầu.

As you might guess, one way around the problem of a parameter's destructor function destroying data needed by the calling argumeny is to the address of the object and not the object itself. When an address is passed, no new object is created, and therefore, no destructor is called when the function returns. (As you will see in the next chapter, C++ provides a variation on this theme that offers a very elegant alternative.)

Như bạn đã ước chừng, một thứ xung quanh vấn đề của hàm hủy một tham số mà phá hủy dữ liệu cần thiết bởi việc gọi đối số là địa chỉ của đối tượng chứ không phải bản thân đối tượng.

However, an even better solution exists, which you can use after you have learned about a special type of constructor called a copy constructor. A copy constructor lets you define precisely how copies of objects are made. (Copy constructors are discussed in Chapter 5.)

Tuy nhiên, một biện pháp tốt hơn tồn tại mà bạn có thể dùng sau khi học về kiểu hàm tạo đặc biệt mang tên *h‡m tạo copy*. Hàm tạo copy cho phép bạn định nghĩa chính xác các mà bản copy đối tượng được tạo ra. (Chúng ta sẽ nói về hàm tao copy này trong chương 5.)

Exercises

1. *Using the **stack** example from section 3.1, Example 2, add a function called **showstack**( ) that is passed an object of type **stack**. Have this function display the contents of a stack.*

Dùng ví dụ stack (ngăn xếp) trong phần 3.1, ví dụ 2, thêm hàm có tên **showstack** mà được chuyển tiếp một đối tượng kiểu **stack**. Hàm này biểu thị nội dung của stack.

As you know, when an object is passed to a function, a copy of that object is made. Further, when that function returns, the copy's destructor function is called. Keeping this in mind, what is wrong with the following program?

Như bạn biết, khi một đối tượng được chuyển tiếp đến hàm, một bản copy của đối tượng sẽ được tạo ra. Hơn nữa, khi hàm đó trả lại, thì hàm hủy của bản copy được

gọi. Hãy nhớ điều này, giờ thì xem trong đoạn chương trình sau sai cái gi

```
//This program contains an error
#include <iostream>
#incldue <cstdlib>
using namespace std;

class dyna
{
     int *p;
public:
     dyna(int i);
     ~dyna() { free(p); cout << "freeing \n"; }
     int get() { return *p; }
};

dyna::dyna(int i)
{
     P = (int *) malloc(sizeof(int));
     if(!p)
     {
          cout << "Allocation failure\n";
          exit(1);
     }
```

```
     *p = i;
}

// Return negative value of *ob.p
int neg(dyna ob)
{
     return -ob.get();
}

     int main()
     {
          dyna o(10);

          cout << o.get() << "\n";
          cout << neg(o) << "\n";

          dyna o2(20);
          cout << o2.get() << "\n";
          cout << neg(o2) << "\n";

          cout << o.get() << "\n";
          cout << neg(o) << "\n";

          return 0;
```

```
}
```

1.3. RETURNING OBJECT FROM FUNCTIONS – Trả về đối tượng cho hàm:

Just as you can pass objects to functions, function can return objects. To do so, first declare the function as returning a class type. Second, return an object of that type using to understand about returning objects from function, however: when an object is returned by a function, a temporary object is automatically created which holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction of this temporary object might cause unexpected side effects in some situation, as is illustrated in Example 2 below.

Như bạn có thể chuyển đối tượng qua hàm, thì hàm cũng có thể trả về đối tượng. Để làm vậy, trước tiên phải khai báo hàm có kiểu trả về một lớp. Thứ hai, trả về một đối tượng của kiểu đó dùng khi hiểu việc trả đối tượng về từ hàm, tuy nhiên: khi một đối tượng được trả về bởi một hàm, một đối tượng tạm thời sẽ được tự động tạo ra để nắm giữ giá trị trả về. Nó sẽ là đối tượng này khi thực sự được trả về từ hàm.

Sau khi giá trị được trả về, đối tượng tạm thời này bị phá hủy. Sự phá hủy đối tượng tạm thời này có thể gây ra một vài ảnh hưởng không mong đợi trong một số tình huống, như ví dụ 2 minh họa dưới đây

Example

Here is an example of a function that returns an object:

Sau đây là ví dụ về việc hàm trả về một đối tượng

```
//Returning an object

#include <iostream>

#incldue <cstring>

using namespace std;
```

```
class samp
{
     char sp[80];
public:
     void show() { cout << s << "\n"; }
     void set(char *str) { strcpy(s, str); }
};

//Return an object of type samp
samp input()
{
     char s[80];
     samp str;

     cout << "Enter a string: ";
     cin >> s;

     str.set(s);

     return str;
}

int main()
{
```

```
    samp ob;


    //assign returned objects to ob
    ob = input();
    ob.show();


    return 0;
}
```

*In this example, **input( )** creates a local object called **str** and then reads a string from the keyboard. This string is copied into **str.s,** and then **str** is returned by the function. This object is then assigned to **ob** inside **main( )** when it is returned by the call to **input( )**.*

Trong ví dụ này, hàm **input( )** tạo ra một đối tượng cục bộ mang tên str và sau đó đọc 1 chuỗi từ bàn phím. Chuỗi này được copy vào str.s và sau đó str được trả về bởi hàm. Đối tượng này sau đó được gán cho **ob** trong hàm main( ) khi nó được trả về bởi việc gọi thực hiện hàm input( ).

You must be careful about returning objects from function if those objects contain destructor functions because the returned object goes out of scope as soon as the value is returned to the calling routine. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is assigned the return value is still using it. For example, consider this incorrect version of the preceding program:

Bạn phải cẩn thận với việc trả đối tượng về từ hàm nếu đối tượng chứa hàm hủy bởi vì việc trả đối tượng làm cho vượt khỏi tầm vực ngay khi mà giá trị được gọi chương trình con. Ví dụ, nếu đối tượng trả về bởi hàm có phương thức hủy thì sẽ giải phóng vùng nhớ cấp phát động, vùng nhớ đó sẽ bị giải phóng ngay cả khi đối tượng mà nó được gán trả về giá trị vẫn còn sử dụng nó. Ví dụ, xem xét đoạn chương trình lỗi sau

```
//An error generated by returning an object

#include <iostream>
```

```
#include <cstring>
#include <stdlib>
using namespace std;

class samp
{
     char *s;
public:
     samp() { s = '\0'; }
     ~samp() { if(s) free(s); cout << "Freeing s\n";}
     void show() { cout << s << "\n"; }
     void set(char *str);
};
//Load a string
void samp::set(char *str)
{
     s = (char *) malloc(strlen(str)+1);
     if(!s)
     {
          cout << "Allocation error\n";
          exit(1);
     }

     strcpy(s, str);
}
```

```
//Return an object of type samp
samp input()
{
     char s[80];
     samp str;

     cout << "Enter a string: ";
     cin >> s;

     str.set(s);

     return str;
}
int main()
{
     samp ob;

     //assign returned objects to ob
     ob = input();
     ob.show();

     return 0;

}
```

The output from this program is shown here:

Màn hình hiển thị:

```
Enter a string : Hello

Freeing s

Freeing s

Hello

Freeing s

Null pointer assignment
```

*Notice that **samp**'s destructor function is called three times. First, it is called when the local object **str** goes out of the scope when **input( )** returns. The second time **~samp**( ) is called is when the temporary object returned by **input( )** is destroyed. Remember, when an object is returned from a function, an invisible (to you) temporary object is automatically generated which holds the returne value. In this case, this object is simply a copy of str, which is the return value of the function. Therefore, after the function has returned, the temporary object's destructor is executed. Finally, the destructor for object **ob**, inside **main**( ), is called when the program terminates.*

Chú ý rằng hàm hủy của samp được gọi đến 3 lần. Lần đầu, nó được gọi khi biến cục bộ str vượt khỏi tầm vực khi hàm input() trả về. Lần hai hàm ~samp( ) được gọi khi đối tượng tạm thời trả về bởi hàm input( ) bị phá hủy. Nhớ rằng, khi một đối tượng được trả về từ hàm, coi như là một đối tượng ẩn (với bạn) thì được tự động phát xuất và giữ giá trị trả về. Trong trường hợp này, đối tượng này đơn giản là một bản copy của str là giá trị trả về của hàm. Vì vậy, sau khi hàm trả về, hàm hủy của đối tượng tạm thời này được thực thi. Cuối cùng, hàm hủy của đối tượng **ob** bên trong hàm main( ) được gọi khi chương trình kết thúc.

*The trouble is that in this situation, the first time the destructor executes, the memory allocated to hold the string input by **input( )** is freed. Thus, not only do the other two calls to **samp**'s destructor try to free an already released piece*

of dynamic memory, but they destroy the dynamic allocation system in the process, as evidenced by the run-time message "Null pointer assignment." (Depending upon your compiler, the memory model used for compilation, and the like, you may or may not see this message if you try this program.)

Phiền toái trong tình huống này, lần đầu tiên hàm hủy thực thi, vùng nhớ được cấp phát để lưu chuỗi nhập vào bởi hàm input( ) thì bị giải phóng. Vì vậy, không những hai lời gọi hàm hủy khác của samp cố gắng xóa vùng nhớ đã được giải phóng khỏi bộ nhớ động mà chúng còn phá hủy hệ thống cấp phát vùng nhớ động trong xử lý, bằng chứng là xuất hiện thông báo lỗi trong xử lý "Null pointer assignment." (Tùy thuộc vào trình biên dịch của bạn, loại vùng nhớ dùng cho biên dịch và giống như vậy mà bạn có thể thấy hay không thấy thông báo lỗi này khi bạn chạy thử chương trình này.

The key point to be understood from this example is that when an object is returned form a function, the temporary object used to effect the return will have its destructor function called. Thus, you should avoid returning objects in which this situation is harmful. (As you will learn in Chapter 5, it is possible to use a copy constructor to manage this situation.)

Mấu chốt cần hiểu sau ví dụ này là khi một đối tượng được trả về từ hàm, thì đối tượng tạm thời được dùng để tác động sự trả về sẽ làm cho hàm hủy của nó được gọi. Vì vậy, bạn nên tránh trả về đối tượng trong tình huống có hại như vậy. (Bạn sẽ học trong chương 5 cách xử lý những tình huống như vầy.)

Exercises

*To illustrate exactly when an object is constructed and destructed when returned from a function, create a class called **who**. Have **who**'s constructor take one character argument that will be used to identify an object. Have the constructor display a message similar to this when constructing an object.*

Để minh họa xác thực khi một đối tượng được tạo hay hủy khi trả về từ một hàm, tạo ra một lớp mang tên who. Hàm tạo lấy một đối số kiểu ký tự mà sẽ dùng để nhận dạng đối tượng. Hàm hủy hiển thị tin nhắn giống như trên khi tạo một đối tượng

```
Constructing who #x
```

*Where **x** is the identifying character associated with each object. When an object is destroyed, have a message similar to this displayed:*

Khi x là ký tự nhận dạng kết hợp với mỗi đối tượng. Khi một đối tượng bị hủy, xuất hiện một tin nhắn giống như sau:

```
Destroying who #x
```

Where, again, x is the indentifying character. Finally, create a function called **make_who**( ) that returns a **who** object. Give each object a unique name. Note the output displayed by the program.

Nơi, lần nữa, x là ký tự nhận dạng. Cuối cùng, tạo ra hàm mang tên make_who( ) trả về giá trị của đối tượng who. Mỗi đối tượng một tên độc nhất. Ghi chép những gì hiện ra bởi chương trình.

Other than the incorrect freeing of dynamically allocated memory, think of a situation in which it would be improper to return an object from a function.

Thông qua việc sai lầm trong giải phóng bộ nhớ cấp phát động, hãy nghĩ về tình huống mà nó sẽ không phù hợp để trả một đối tuợng từ hàm.

1.4. AN INTRODUCTION TO FRIEND FUNCTIONS – Tổng quan về hàm friend:

There will be times when you want a function to have access to the private members of a class without that function actually being a member of that class. Toward this end, C++ supports friend function. A friend is not a member of a class but still has access to its private elements.

Sẽ có lúc bạn muốn có một hàm truy cập vào các thành phần riêng của một lớp mà hàm đó không phải là thành viên của lớp đó. Nhằm mục đính này, C++ đưa ra các hàm **friend**. Một hàm friend không phải là thành viên của một lớp nhưng có thể truy cập các thành viên của lớp đó.

Two reasons that friend functions are useful have to do with operator overloading and the creation of certain types of I/O functions. You will have to wait until later to see these uses of a friend in action. However, a third reason for friend functions is that there will be times when you want one function to

have access to the private members of two or more different classes. It is this use that is examined here.

Có hai lý do mà các hàm friend rất hữu ích là khi tạo ra các kiểu hàm nhập/ xuất nào đó hoặc khi làm việc với sự nạp chồng toán tử. Bạn sẽ thấy rõ hơn công dụng của các hàm friend trong các chương sau. Tuy nhiên lý do thứ ba đối với các hàm friend là sẽ có lúc bạn cần một hàm để truy cập đến các thành viên riêng tư của hai hay nhiều lớp khác nhau. Vấn đề này sẽ được trình bày ở đây.

*A friend function is define as a regular, nomember function. However, inside the class declaration for which it will be a friend, its prototype is also included, prefaced by the keyword **friend.** To understand how this works, examine this short program:*

Một hàm friend được định nghĩa như một hàm bình thường, không friend. Tuy nhiên, bên trong khai báo của lớp, hàm sẽ là một friend khi nguyên mẫu của hàm mở đầu bằng từ khóa **friend**. Để hiểu phương thức hoạt động hãy xét chương trình ngắn sau:

```
// An example of a friend function.

#include <iostream>

using namespace std ;


class myclass

{

     int n, d ;

public:

     myclass ( int i, int j)

     {

          n = i ;

          d = j ;

     }
```

```
     // declare a friend of myclass
     friend int isfactor (myclass ob) ;
};

/* Here is friend function definition. It returns true
if d is a factor of n. Notice that the keyword friend
is not used in the definition of isfactor( ).
*/
int isfactor (myclass ob)
{
     if (!(ob.n % ob.d))   return 1;
     else   return 0;
}
int main ( )
{
     myclass ob1 (10, 2), ob2 (13, 3) ;

     if (isfactor (ob1) )
          cout << " 2 is a factor of 10\n" ;
     else
          cout << " 2 is not a factor of 10\n" ;

     if (isfactor (ob2) )
          cout << " 3 is a factor of 13\n" ;
     else
          cout << " 3 is not a factor of 13\n" ;
```

```
    return 0 ;
}
```

*In this example, **myclass** declares its constructor function and the friend **isfactor( )** inside its class declaration. Because **isfactor( )** is a friend of **myclass, isfactor( )** has access to its private members. This is why, within **isfactor( )**, it is possible to directly refer to **ob.n** and **ob.d**.*

Trong ví dụ này, lớp **myclass** khái báo hàm dựng và hàm friend **isfactor( )** của nó bên trong khai báo lớp. Vì **isfactor( )** là một hàm friend của **myclass** nên **isfactor( )** có thể truy cập đến các thành phần riêng của lớp **myclass**. Đây là lý do tại sao trong **isfactor()** có thể tham chiếu trực tiếp đến **ob.n và ob.d.**

It is important to understand that a friend function, is not a member of the class for which it is a friend, Thus, it is not possible to call a friend function by using an object name and a class member access operator (a dot or an arrow). For example, give the preceding example, this statement is wrong:

Điều quan trọng để hiểu là một hàm friend không phải là thành viên của lớp chứa hàm đó. Do đó, không thể gọi hàm friend bằng cách dùng tên đối tượng và toán tử để truy cập thành viên (dấu chấm hoặc mũi tên). Chẳng hạn với ví dụ trên đây, câu lệnh sau là sai :

```
ob1.isfactor(); //wrong; isfactor ( ) is not a member function
```

Instead, friends are called just like regular functions.

Các hàm friend được gọi như các hàm bình thường.

Although a friend function has knowledge of the private elements of the class for which it is a friend, it can only access them through an object of the class. That is, unlike a member function of myclass, which can refer to n or d directly, a friend can access these variables only in conjunction with an object that is declared within or passed to the friend function.

Mặc dù, hàm friend nhận biết các phần tử riêng của lớp mà nó là friend, nhưng hàm friend chỉ có thể truy cập các phần tử riêng này qua một đối tượng của lớp.

Nghĩa là, không giống như hàm thành viên của lớp **myclass**, có thể tham chiếu trực tiếp đến **n** và **d,** một friend có thể truy xuất các biến này thông qua kết nối với một đối tượng được khai báo bên trong hoặc được truyền đến cho hàm friend.

Note:

The preceding paragraph brings up an important side issue. When a member function refers to a private element, it does so directly because a member function is excuted only in conjunction with an object of that class. Thus, when a member function refer to a private element, the compiler knows which object that private element belongs to by the object that is linked to the function when that member function is called. However, a friend function is not linked to any object. It simly is granted access to the private elements of a class. Thus, inside the friend function, it is meaningless to refer to a private member without reference to a specific object.

CHÚ Ý:

Đoạn trên đây đưa đến một vấn đề quan trọng. Khi một hàm thành viên tham chiếu đến một phần tử riêng, thì nó không tham chiếu trực tiếp vì hàm thành viên chỉ được thi hành khi liên kết với một đối tượng của lớp đó. Do đó, khi một hàm thành viên tam chiếu đến một phần tử riêng, trình biên dịch biết đối tượng nào mà phần tử riêng đó thuộc về do đối tượng đó được kết nối với hàm khi hàm thành viên được gọi. Tuy nhiên một hàm friend không được kết nối với bất kỳ đối tượng nào. Ta chỉ được truy cập đến các thành phần riêng của một lớp. Do đó. Ở bên trong hàm friend, sẽ vô nghĩa khi tham chiếu đến một thành viên riêng mà không có tham chiếu đến một đối tượng cụ thể.

Because friends are not members of a class, they will typically be passed one or more objects of the class for which they are friends. This is the case with ***isfactor( ).*** *It is passed an object of myclass, it can access* ***ob's*** *private elements. If* ***isfactor( )*** *had not been made a friend of myclass, it would be able to access* ***ob.d*** *or* ***ob.n*** *since n and d are private members of myclass.*

Do các hàm friend không phải là thành viên của lớp, chúng có thể được truyền nhiều hay chỉ một đối tượng của lớp mà chúng được định nghĩa để hoạt động. Đây là trường hợp của hàm **isfactor( ).** Nó được truyền đến đối tượng **ob** của lớp **myclass.** Tuy nhiên vì **isfactor( )** là một hàm friend của lớp **myclass,** nên nó có thể truy cập đến các phần tử riêng của đối tượng **ob.** Nếu **isfactor( )** không phải là hàm friend của lớp **myclass,** thì nó không thể truy xuất **ob.n** hoặc **ob.d** vì **n** và **d** là các

thành viên riêng của lớp **myclass.**

Remember:

A friend function is not a member and can not be qualified by an object name. It must be called just like a normal function.

A friend function is not inherited. That is, when a base class include a friend function, that friend function is not a friend of a derived class.

One other important point about friend function is that a friend function can be friends with more than one class.

CẦN NHỚ:

Một hàm friend không phải là một thành viên của một lớp và nó không thể được xác định bằng tên của đối tượng. Nó được gọi giống như các hàm bình thường.

Một hàm friend không được kế thừa. Nghĩa là khi một lớp cơ sở gồm một hàm friend thì hàm friend này không phải là hàm friend của lớp dẫn xuất.

Một điểm quan trọng khác về hàm friend là một hàm friend có thể là các hàm friend của hơn một lớp.

Example:

*One common (and good) use of a friend function occurs when two different types of classes have some quantity is common that needs to be compared. For example, consider the following program, which creates a class called **car** and a class called **truck,** each containing, as a private variable, the speed of the vehicle it represents:*

Cách dùng thông thường (và tốt) của hàm friend xảy ra khi hai kiểu lớp khác nhau có một số đại lượng cần được so sánh. Ví dụ, xét chương trình sau đây tạo ra lớp **car** và lớp **truck**. Mỗi lớp có một biến riêng là tốc độ của xe:

```
#include <iostream>

using namespace std ;


class truck ; // a forward declaration
```

```
class car
{
     int passengers ;
     int speed ;
 public :
     car (int p, int s)
     {
          passengers = p ;
          speed = s ;
     }
     friend int sp_greater ( car c, truck t) ;
};

class truck
{
     int weight ;
     int speed ;
public :
     truck (int w, int s)
     {
          weight = w ;
          speed = s ;
     }
     friend int sp_greater (car c, truck t) ;
```

```
};

/* Return possitive if car speed faster than truck.
    Return 0 if speeds are the same.
     Return negative if car speed faster than truck.
*/
int sp_greater (car c, truck t)
{
     return c.speed – t.speed ;
}

int main ( )
{
     int t ;
     car c1(6, 55), c2(2, 120) ;
     truck t1(10000, 55), t2(20000, 72) ;

     cout << "Comparing c1 and t1: \n" ;
          t = sp_greater (c1, t1) ;
     if (t < 0)
          cout << "Truck is faster. \n" ;
     else if (t == 0)
          cout << "Car and truck speed is the same.
\n" ;
     else
```

```
            cout << "Car is faster.\n" ;


      cout << "Comparing c2 and t2: \n" ;
            t = sp_greater (c2, t2) ;
      if (t < 0)
            cout << "Truck is faster. \n" ;
      else if (t == 0)
            cout << "Car and truck speed is the same.
\n" ;
      else
            cout << "Car is faster.\n" ;


      return 0 ;
}
```

*This program contain rhe function **sp_greater( )**, which is a friend function of both the car and truck classes. (As stated a function can be a friend of two or more classes). This function returns positive if the car object is going faster than the truck object, 0 if their speeds are the same, and negative if the truck is going faster.*

Chương trình này có hàm **sp_greater( )** là một hàm friend của cả hai lớp **car** và **truck**. (Như đã nói ở trên, một hàm có thể là hàm friend của hai hoặc nhiều lớp). Hàm này trả về một giá trị dương nếu đối tượng thuộc lớp **car** chạy nhanh hơn đối tượng thuộc lớp **truck**, trả về 0 nếu nếu hai tốc đọ bằng nhau và giá trị âm nếu đối tượng thuộc lớp **truck** nhanh hơn.

*This program illustrates one important C++ syntax element: the forward declaration (also called a forward reference). Because **sp_greater( )** takes parameters of both the car and the the truck classes, it is logically impossible to declare both before including **sp_greater ( )** in either. Therefore, there needs to be some way to tell the compiler about a class name without actually*

declaring it. This is called a forward declaration. In C++, to tell the compiler that an identifier is the name of a class, use a line like this before the class name is first used:

Chương trình này minh họa một yếu tố cú pháp quan trọng trong C++: *tham chiếu hướng tới,* vì hàm **sp_greater( )** nhận tham số của cả hai lớp **car** và **truck**, về mặt luận lý không thể khai báo cả hai trước khi đưa **sp_greater( )** vào. Do đó cần có cách nào đó để báo cho trình biên dịch biết tên lớp mà thực sự không cần phải khai báo. Điều này được gọi là tham chiếu hướng tới. Trong C++, để báo cho trình biên dịch một định danh là tên của lớp, dùng cách như sau trước khi tên lớp được sử dụng :

```
class class-name ;
```

For example, in the preceding program, the forward declaration is

Ví dụ, trong chương trình trước, tham chiếu hướng tới là:

```
class truck ;
```

*Now truck can be used in the friend declaration of **sp_greater( )** without generating a compile-time error.*

Bây giờ **truck** có thể được dùng trong khai báo hàm friend **sp_greater( )** mà không gây ra lỗi thời gian biên dịch.

*A function can be a member of one class and a friend of another. For example, here is the proceding example rewritten so that **sp_greater()** is a member of car and a friend of truck:*

Một hàm có thể là thành viên của một lớp và là một hàm friend của lớp khác. Sau đây là chương trình trước được viết lại để hàm **sp_greater( )** là thành viên của lớp **car** và là hàm friend của lớp **truck**.

```
#include <iostream>

using namespace std ;

class truck ; // a forward declaration

class car

{
```

```
     int passengers ;
     int speed ;
 public :
     car (int p, int s)
     {
          passengers = p ;
          speed = s ;
     }
     int sp_greater (truck t) ;
};

class truck
{
     int weight ;
     int speed ;
public :
     truck (int w, int s)
     {
          weight = w ;
          speed = s ;
     }
     // not new use of the scope resolution operator
     friend int car :: sp_greater (truck t) ;
};
```

```
/* Return possitive if car speed faster than truck.
    Return 0 if speeds are the same.
     Return negative if car speed faster than truck.
*/
int car :: sp_greater (car c, truck t)
{
     return speed – t.speed ;
}

int main ( )
{
     int t ;
     car c1(6, 55), c2(2, 120) ;
     truck t1(10000, 55), t2(20000, 72) ;

     cout << "Comparing c1 and t1: \n" ;
          t = sp_greater (c1, t1) ;
     if (t < 0)
          cout << "Truck is faster. \n" ;
     else if (t == 0)
          cout << "Car and truck speed is the same.
\n" ;
     else
          cout << "Car is faster.\n" ;
```

```
     cout << "Comparing c2 and t2: \n" ;
          t = sp_greater (c2, t2) ;
     if (t < 0)
          cout << "Truck is faster. \n" ;
     else if (t == 0)
          cout << "Car and truck speed is the same.
\n" ;
     else
          cout << "Car is faster.\n" ;

     return 0 ;
}
```

*Notice the new use of the scope resolution operator as it occurs in the friend declaration within the truck class declaration. In this case, it is used to tell the compiler that the function **sp_greater( )** is a member of the car class.*

Chú ý đến cách dùng mới của toán tử phân giải phạm vi (Scope resolution operator) xảy ra trong khai báo friend bên trong khai báo của lớp **truck.** Trong trường hợp này, nó được dùng để báo cho trình biên dịch hàm **sp_greater( )** là một thành viên của lớp **car.**

One easy way to remember how to use the scope resolution operator is that the class name followed by the scope resolution operator followed by the member name fully specifies a class member.

Một cách dễ dàng để nhớ cách sử dụng toán tử phân giải phạm vi là tên lớp theo sau bởi toán tử phân giải phạm vi rồi đến tên thành viên chỉ rõ thành viên của lớp.

In fact, when refering to a member of a class, it is never wrong to fully specify its name. However, when an object is used to call a member function or access a member variable, the full name is redundant and seldom used. For example:

Thực tế, khi tham chiếu đến một thành viên của lớp sẽ không bao giờ sai khi chỉ

rõ tên đầy đủ. Tuy nhiên, khi một đối tượng được dùng để gọi một thành viên hay truy cập một biến thành viên, tên đầy đủ sẽ thừa và ít được dùng. Ví dụ:

t = cl.sp_greater (t1) ;

can be written using the (redundant) scope resolution operator and the class name car like this:

có thể được viết bằng cách dùng toán tử phân giải và tên lớp **car** như sau:

t = c1.car :: sp_greater (t1) ;

*However, since c1 is an object of type car, the compiler already knows that **sp_greater( )** is a member of the car class, making the full class specification unnecessary.*

Tuy nhiên, vì c1 là một đối tượng kiểu **car,** trình biên dịch đã biết **sp_greater( )** là một thành viên của lớp car nên việc chỉ rõ tên đầy đủ là không cần thiết.

Exercise:

*Imagine a situation in which two classes, called **pr1** and **pr2,** show here, share one printer. Further, imagine that other parts of your program need to know when the printer is in use by an object of either of these two classes. Create a function called **inuse( )** that return true when the printer is being used by either and false otherwise. Make this function a friend of both **pr1** and **pr2.***

Hãy tưởng tượng một tình huống trong đó có 2 lớp **pr1** và **pr2** được trình bày dưới đây sử dụng chung một máy in. Xa hơn, hãy tưởng tượng các phần của chương trình của bạn cần biết khi nào thì máy in được dùng của một đối tượng của một trong hai lớp. Hãy tạo hàm **inuse( )** trả về giá trị đúng (true) khi máy in đang được dùng và giá trị sai (false) trong trường hợp ngược lại. Làm cho nó trở thành hàm friend của cả hai lớp **pr1** và **pr2**.

```
class pr1
{
      int printing ;
      //……
public:
```

```
        pr1( )
        {
              printing = 0 ;
        }
        void setprint(int status)
        {
              printing = status ;
        }
        //…..
};

class pr2
{
        int printing ;
public:
        pr2( )
        {
              printing = 0 ;
        }
        void setprint(int status)
        {
              printing = status ;
        }
        //…..
};
```

MASTERY SKILLS CHECK:

Before proceding, you should be able to answer the following question and perform the exercises.

KIỂM TRA KỸ NĂNG TIẾP THU:

Trước khi tiếp tục, bạn hãy trả lời các câu hỏi và làm các bài tập sau:

What single prerequisite must be met in order for one object to be assigned to another?
Give the classfragment.

Để cho một đối tượng được gán vào một đối tượng khác thì trước hết cần có điều kiện gì?

Cho đoạn chương trình sau:

```
class samp
{
     double *p ;
public:
     samp(double d)
     {
          p = (double *) malloc( sizeof (double)) ;
          if (!p)   exit(1) ;   // allocation error
          *p = d ;
     }
     ~ samp ( )
     {
          free (p) ;
     }
```

```
        //….
};


//……
samp ob1 (123, 09), ob2(0, 0) ;
//…….
ob2 = ob1 ;
```

*what problem is caused by the assignment of **ob1** to **ob2**?*

Điều gì xảy ra sau khi gán **ob1** vào **ob2?**

Give the class :

3. Cho lớp sau :

```
class planet
{
        int moons ;
        double dist_from_sun ;  // in miles
        double diameter ;
        double mass ;
public:
        //…….
        double get_miles ( )
        {
                return dist_from_sun ;
        }
};
```

*create a function called **light( )** that takes as an argument an object of type planet and returns the number of seconds that it takes light from the sun to reach the planet. (assume that light travels at 186,000 miles per second and that **dist_from_sun** is specified in miles).*

hãy tạo hàm **light( )** nhận một đối tượng thuộc lớp **planet** làm đối số trả về một số giây cần thiết để ánh sáng đi từ mặt trời đến hành tinh. (Giả thiết ánh sáng di chuyển 186.000 dặm/giây và **dis_from_sun** được tính bằng dặm).

Can the address of an object be passed to a function as an argument?

Địa chỉ một đối tượng có thể được truyền cho hàm như một đối số không?

Using the stack class, write a function called loadstack( ) that returns a stack that is already loaded with the letters of the alphabet (a-z). Assign this stack to another object in the calling routine and prove that it contains the alphabet. Be sure to change the stack size so it is large enough to hold the alphabet.

Dùng lớp **stack,** hãy viết hàm **loadstack( )** để trả về một ngăn xếp đã được nạp vào các chữ cái (a-z). Hãy gán ngăn xếp này cho một đối tượng khác khi gọi một chương trình phụ và chứng tỏ rằng nó chứa các chữ cái alphabet.

Explain why you must be careful when passing objects to a function or returning objects from a function.

Hãy giải thích tại sao bạn phải cẩn thận khi truyền các đối tượng cho hàm hoặc trả về các đối tượng từ một hàm.

What is a friend function?

Hàm friend là gì?

COMMULATIVE SKILLS CHECK:

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

KIỂM TRA KỸ NĂNG TỔNG HỢP:

Phần này kiểm tra xem bạn có thể kết hợp chương này với các chương trước như thế nào.

*Functions can be overloaded as long as the number oer type of their parameters differs. Overload **loadstack( )** from Exercise 5 of the mastery Skills check so that it takes an integer, called**pper**, as a parameter. In the overloaded version, if **upper** is 1, load the stack with the uppercase alphabet. Otherwise, load it with the lowercase alphabet.*

Các hàm có thể được nạp chồng. Hãy nạp chồng hàm **loadstack( )** trong bài tập 5 của phần kiểm tra kỹ năng tiếp thu để cho nó nhận một số nguyên, gọi là **upper,** làm tham số. Trong phiên bản định nghĩa chồng, nếu upper là 1, hãy đưa vào ngăn xếp các mẫu tự in hoa. Ngược lại đưa vào ngăn xếp các mẫu tự thường.

*Using the **strtype** class shown in section 3.1. Example 3, add a friend function that takes as an argument a pointer to an object of type strtype and returns a pointer to the string pointed to by that object. (That is,have the function return p). Call this function **get_string( ).***

Dùng lớp strype trong phần 3.1, ví dụ 3, hãy bổ sung một hàm friend nhận con trỏ về một đối tượng kiểu strype làm tham số và trả về con trỏ cho một chuỗi được trỏ bởi đối tượng đó. (Nghĩa là cho hàm trả về P). Gọi hàm này là get_string( ).

Experiment: When an object of a derived class, is the data associated with the base class also copied? To find out, use the following two class and write a program that demonstrates what happens.

Thực hành :Khi một đối tượng của lớp dẫn xuất được gán cho một đối tượng khác của cùng lớp dẫn xuất thì dữ liệu để kết hợp với lớp cơ sở có được sao chép không? Để biết được, hãy dùng hai lớp sau đây và viết chương trình cho biết những gì xảy ra.

```
class base
{
     int a ;
public:
     void load_a( int n)
     {
          a = n ;
     }
     int get_a( )
     {
```

```
          return a ;
     }


};

class derived : public base
{
     int b ;
public :
     void load_b int n)
     {
          b = n ;
     }
     int get_b )
     {
          return a ;
     }
} ;
```

CHƯƠNG 4

ARRAYS, POITERS, AND REFERENCES - Mảng, con trỏ và tham chiếu

Chapter objectives

Mục lục

This chapter examines several important issues involving arrays of objects and pointers to objects. It concludes with a discussion of one of C++'s most important innovations: the reference.The reference is crucial to many C++ features, so a careful reading is advised.

Chương này xét đến nhiều vấn đề quan trọng gồm mảng, các đối tượng và con trỏ tới đối tượng. Phần kết trình bày về một trong những điểm mới quan trọng nhất của C++: Tham chiếu. Tham chiếu là rất quan trọng đối với nhiều đặc điểm của C++, vì thế bạn nên đọc kỹ.

C++ Review Skill check

C++ Kiểm tra kỹ năng ôn

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Trước khi bắt đầu, bạn nên trả lời đúng các câu hỏi và làm các bài tập sau:

When one object is assigned to another, what precisely takes place?

1. Khi một đối tượng được gán cho một đối tượng khác, điều gì xảy ra?

Can any troubles or side effects occur when one object is assigned to another> (Give an example).

2. Có rắc rối hoặc tác dụng phụ nào xảy ra khi gán một đối tượng cho một đối tượng khác? (cho một ví dụ).

When an object is passed as an argument to a function, a copy of that object is made. Is the copy's constructor function called? Is its destructor called?

3. Khi một đối tượng được truyền như đối số cho một hàm, bản sao của đối tượng đó được thực hiện. Hàm tạo của bản sao đó có được gọi không? Hàm hủy của bản sao có được gọi không?

By default, objects are passed to functions by value, which means that what occurs to the copy inside the function is not supposed to affect the argument used in the call. Can there be a violation of this principle? If so, give an example.

4. Theo mặc định, các đối tượng truyền cho hàm bằng giá trị, nghĩa là những gì xảy ra cho bản sao bên trong hàm được giả thiết là không có ảnh hưởng đến đối số được dùng trong lời gọi… Có thể có sự vi phạm nguyên lý này không? Cho ví dụ.

*Given the following class, create a function called **make_sum()** that returns an object of type **summation**. Have this function prompt the user for a number and then construct an object having this value and return it to the calling procedure. Demonstrate that the function works.*

5. Cho lớp sau đây, hãy tạo hàm **make_sum()** để trả về một đối tượng kiểu **summation**. Hãy làm cho hàm này nhắc người sử dụng nhập một số và sau đó tạo một đối tượng có giá trị này và trả nó và cho thủ tục. Hãy chứng tỏ hàm hoạt động.

```
class summation
{
     int num;
     long sum; // summation of num
public:
     void set_sum(int n);
     void show_sum()
     {
          cout<<num<<" summed is "<< sum<<"\n";
     }
};

void summation::set_sum(int n)
{
     int i;
     num = n;

     sum = 0;
     for(i=1; i<=n;i++)
          sum+=i;
```

```
}
```

*In the preceding question, the function **set_num()** was not defined in line withing the **summation** class declaration. Give a reason why this might be necessary for some compilers.*

6. Trong bài trên, hàm **set_num()** đã không được định nghĩa nội tuyến trong khai báo lớp **summation**. Hãy giải thích tại sao điều này là cần thiết cho một số trình biên dịch.

*Given the following class, show how to add a friend function called **isneg()** that takes one parameter of type **myclass** and returns true if **num** is negative and false otherwisse.*

7. Cho lớp sau đây, hãy bổ sung một hàm friend **isneg()** nhận một tham số loại **myclass** và trả về giá trị đúng nếu **num** là âm và giá trị sai nếu ngược lại.

```
class myclass

{

     int num;

public:

     myclass(int x) { num = x;}

};
```

Can a friend funtion be friends with more than one class?

8. Một hàm friend có thể là những hàm friend của nhiều hơn một lớp được không?

1.1. ARRAYS OF OBJECTS - MẢNG CÁC ĐỐI TƯỢNG

As has been stated several times, object are varialbes and have the same capabilities and attributes as any other type of variable. Therefore, it is perfectly acceptable for objects to be arrayed. The syntax for declaring an array of objects is exactly like that used to declare an array of any other type of variable. Further, arrays of objects are accessed just like arrays of other

types of variables.

Như đã được nói nhiều lần, các đối tượng là các biến và có cùng khả năng và thuộc tính như bất kỳ loại biến nào. Do đó, các đối tượng hoàn toàn có thể được xếp thành mảng. Cú pháp để khai báo một mảng các đối tượng là hoàn toàn giống như cú pháp được dùng để khai báo mảng các loại biến bất kỳ. Hơn nữa, các mảng của các đối tượng được truy cập như các mảng của các loại biến khác.

VÍ DỤ(EXAMPLES):

Here is an example of an array of objects:

Đây là một ví dụ về mảng các đối tượng

```
#include <stdafx.h>
#include <iostream>
using namespace std;

class samp
{
     int a;
public:
     void set_a(int n) {a = n;}
     int get_a() {return a;}
};

int main()
{
     samp ob[4];
     int i;
```

```
    for(i=0;i<4;i++) ob[i].set_a(i);

    for(i=0;i<4;i++) cout<<ob[i].get_a();

    cout<<"\n";

    return 0;
}
```

*This program creates a four-element array of objects of type **samp** and then loads each element's **a** with a value between 0 and 3. Notice how member functions are called relative to each array element. The array name, in this case **ob**, is indexed; then the member access operator is applied, followed by the name of the member function to be called.*

Chương trình này tạo ra mảng đối tượng có 4 phần tử kiểu **samp** và sau đó nạp vào **a** của mỗi phần tử một giá trị giữa 0 và 3. Chý ý cách gọi các hàm thành viên đối với mỗi phần tử mảng. Tên mảng, trong trường hợp này là **ob**, được đánh chỉ số: Sau đó mỗi toán tử truy cập thành viên được áp dụng, theo sau bởi tên của hàm thành viên được gọi.

*If a class type includes a constructor, an array of objects can be initialized. For example, here **ob** is an initialized array:*

1. Nếu một kiểu lớp gồm một hàm tạo, thì mỗi mảng đối tượng có thể được khởi đầu. Ví dụ, ở đây **ob** là mảng được khởi đầu

```
// Inintialize an array

#include <stdafx.h>

#include <iostream>

using namespace std;


class samp

{
```

```
     int a;
public:
     samp(int n) { a = n;}
     int get_a() {return a;}
};
int main()
{
     samp ob[4]= {-1,-2,-3,-4};
     int i;

     for(i=0;i<4;i++) cout<<ob[i].get_a()<<" ";

     cout<<"\n";

     return 0;
}
```

*This program diplays **-1 -2 -3 -4** on the screen. In this example, the values -1 through -4 are passed to the **ob** constructor function.*

Chương trình này hiển thị -1, -2, -3, -4 ra màn hình. Trong ví dụ này, các giá trị từ -1 đến -4 được truyền cho hàm tạo **ob.**

Actually, the syntax shown in the initialization list is shortthand for this longer form (first shown in Chapter 2):

Thực sự, cú pháp trong danh sách khởi đầu là dạng ngắn của dạn dài hơn như sau (được trình bày trong chương hai):

```
samp ob[4] ={ samp(-1),samp(-2),samp(-3),samp(-4) };
```

However, the form used in the program is more common (although, as you will see, this form will work only with arrays whose constructors take only one argument).

Tuy nhiên, khi khởi đầu mảng một chiều, dạng được trình bày trong chương trình trên là thông dụng (mặc dù, như bạn sẽ thấy, dạng này chỉ làm việc với các mảng có hàm tạo chỉ nhận một đối số).

You can also have multidimensional arrays of objects. For example, here is a program that creates a two-dimensional array of objects and initializes them:

2. Bạn cũng có mảng đối tượng nhiều chiều. Ví dụ, đây là chương trình tạo mảng đối tượng hai chiều và khởi đầu chúng:

```
// creat a two-dimensional array of objects.

#include "stdafx.h"

#include <iostream>

using namespace std;

class samp

{

     int a;

public:

     samp(int n) { a = n;}

     int get_a() { return a; }

};

int main()

{

     samp ob[4][2] = { 1, 2, 3, 4, 5, 6, 7, 7};

     int i;


     for(i=0;i<4;i++)

     {
```

```
            cout<<ob[i][0].get_a()<<" ";
            cout<<ob[i][1].get_a()<<"\n";
      }

      cout<<"\n";
      return 0;
}
```

This program displays

Chương trình này hiển thị

```
1   2
3   4
5   6
7   8
```

As you know, a constructor can take more than one argument. When initializing an array of objects whose constructor takes more than one argument, you must use the alternative form of initialization mentioned earlier. Let's begin with an example:

3. Như bạn biết, một hàm tạo có thể nhận nhiều đối số. Khi khởi đầu một mảng đối tượng có hàm tạo nhận nhiều đối số, bạn phải dùng dạng khởi đầu khác đã nói trước đây. Hãy bắt đầu với một ví dụ:

```
#include <stdafx.h>
#include <iostream>
using namespace std;
class samp
{
      int a,b;
```

```
     public:
          samp( int n,int m) { a = n; b = m;}
          int get_a() { return a;}
          int get_b() { return b;}
};
int main()
{
samp ob[4][2] ={ samp(1,2),samp(3,4),
            samp(5,6), samp(7,8),
samp(9,10), samp(11,12),                    samp(13,14),
samp(15,16)              };
     int i;

     for(i=0;i<4;i++)
     {
          cout<<ob[i][0].get_a()<<" ";
          cout<<ob[i][0].get_b()<<"\n";
          cout<<ob[i][1].get_a()<<" ";
          cout<<ob[i][1].get_b()<<"\n";
     }
     cout<<"\n";
     return 0;
}
```

*In this example, **samp's** constructor takes two arguments. Here, the array **ob** is declared and initialized in **main()** by using direct calls to **samp's** constructor. This is necessary because the formal C++ syntax allows only one argument at a time in a comma-separated list. There is no way, for example, to specify two (or more) arguments per entry in the list. Therefore, when you initialize arrays*

of objects that have constructors that take more than one argument, you must use the "long form" initialization syntax rather than the "shorthand form".

Trong ví dụ này, ham tạo của **samp** nhận hai đối số. Ở đây, mảng **ob** được khai báo và được khởi đầu trong **main()** bằng cách gọi trực tiếp hàm tạo của **samp**. Điều này là cần thiết do cú pháp chính thức của C++ cho phép chỉ một đối số một lần trong danh sách phân cách nhau bằng dấu phẩy. Chẳng hạn, không có cách nào để chỉ rõ hai (hay nhiều) đối số cho mỗi mục nhập trong danh sách. Do đó khi bạn khởi đầu các mảng đối tượng có các hàm tạo nhận hơn một đối số, bạn phải dùng cú pháp khởi đầu "dạng dài" hơn là "dạng ngắn".

Note: *you can always use the long form of initialization even if the object takes only one argument. It's just that the short form is more convenient in this case.*

Chú ý:Bạn có thể luôn luôn sử dụng dạng dài để khởi đầu ngay cả khi đối tượng chỉ nhận một đối số. Nhưng chính dạng ngắn lại thuận tiện hơn trong trường hợp này.

The preceding program displays

Chương trình này hiển thị:

```
1   2

3   4

5   6

7   8

9   10

11  12

13  14

15  16
```

EXERCISES

BÀI TẬP

Using the following class declaration, create a ten-element array and initialize the bchb element with the values A through J. Demonstrate that the array does,

indeed, contain these values.

Dùng khai báo lớp sau đây để tạo mảng 10 phần tử và khởi đầu phần tử **ch** với các giá trị từ A đến J. Hãy chứng tỏ mảng chứa các giá trị này.

```
#include <iostream>

using namespace std;


class letters

{

     char ch;

public:

     letters (char c) { ch = c;}

     char get_ch() { return ch; }

};
```

*Using the following class declaration, create a ten-element array, initialize **num** to the values 1 through 10, and initialize **sqr** to **num**'s square*

2. Dùng khai báo lớp sau đây để tạo một mảng 10 phần tử, hãy khởi đầu **num** với các giá trị từ 1 đến 10 và hãy khởi đầu **spr** đối với bình phương của **num**.

```
#include <iostream>

using namespace std;


class squares

{

     int num, sqr;

public:
```

```
    squares (int a, int b) { num = a; sqr = b;}
    void show() { cout<< num<<' '<<sqr<<"\n";}
};
```

Change the initialization in Exercise 1 so it uses the long form. (That is, invoke ***letter's*** *constructor explicitly in the initialization list).*

3. Hãy thay đổi cách khởi đầu **ob** trong bài tập 1 bằng cách dùng dạng dài. (Nghĩa là nhờ đến hàm tạo của **ob** trong danh sách khởi đầu.

1.2. USING POINTERS TO OBJECTS – Sử dụng con trỏ đối tượng:

As discussed in Chapter 2, objects can be accessed via pointers. As you know, when a pointer to an object is used, the object's members are referenced using the arrow (->) operator instead of the dot (.) operator.

Như đã thảo luận trong chương 2, các đối tượng có thể được truy cập thông qua con trỏ. Như bạn biết khi sử dụng con trỏ tới một đối tượng, các thành viên của đối tượng đó được tham chiếu bằng cách dùng toán tử mũi tên (->) thay vì dùng toán tử chấm (.).

Pointer arithmetic using an object pointer is the same as it is for any other data type: it is performed relative to the type of the object. For example, when an object pointer is incremented, it points to the next object. When an object pointer is decremented, it points to the previous object.

Số học về con trỏ đối với một đối tượng giống như đối với một kiểu dữ liệu khác: nó được thực hiện đối với đối tượng. Khi một con trỏ đối tượng giảm xuống, nó trỏ tới đối tượng đứng trước.

EXAMPLE:

1. Here is an example of object pointer arithmetic:

Đây là ví dụ về số học con trỏ đối tượng:

```
// pointers to objects
#include <iostream>
using namespace std ;
class samp
{
     int a, b ;
public:
     samp (int n, int m) { a = n ;  b = m; }
     int get_a( )  {return a;  }
     int get_b( )  {return b;  }
};
int main ( )
{
     samp ob [4] = {samp(1, 2), samp(3, 4), samp(5,
6), samp(7, 8)};
     int i ;
     samp *p;
     p = ob;       // get  starting address of array
     for (I = 0; i < 4; i++)
     {
          cout << p -> get_a( ) << '   ';
          cout << p -> get_b( ) << "\n" ;
          p++ ;   //advance to next object
     }
     cout << "\n" ;
```

```
    return 0;
}
```

This is program displays:

Chương trình hiển thị như sau:

```
1 2

3 4

5 6

7 8
```

*As evidenced by the output, each time **p** is incremented, it points to the next object in the array.*

Như được chỉ rõ trong dữ liệu xuất, mỗi lần p tăng, nó trỏ tới đối tượng kế tiếp trong mảng.

EXERCISE:

1. Rewrite Example 1 so it displays the contents of the ob array in reverse order.

Hãy viết lại ví dụ 1 để nó hiển thị nội dung của mảng **ob** theo thứ tự ngược lại.

2. Change Section 4.1, Example 3 so the two-dimensional array is accessed via a pointer. Hint: In C++, as in C, all arrays are stored contiguously, left to right, low to high.

Hãy thay đổi phần 4.1, ví dụ để cho mảng hai chiều được truy cập thông qua con trỏ. Hướng dẫn: Trong C++, cũng giống như trong C, tất cả các mảng đều được lưu trữ kề nhau, từ trái qua phải, từ thấp lên cao.

1.3. THE THIS POINTER – Con trỏ This:

C++ contains a special pointer that is called this. this is a pointer that is automatically passed to any member function when it is called, and it is a

pointer to the object that generates the call. For example , given this statement,

C++ có một con trỏ đặc biệt là this. Đây là một con trỏ tự động chuyển đến bất kỳ hàm thành phần nào mà nó được gọi, và nó là con trỏ cho đối tượng sinh ra lời gọi. Ví dụ, cho câu lệnh sau,

```
Ob.f1();  // assume that ob is an object
```

The function f1() is automatically passed a pointer to ob-which is the object that invokes the call. This pointer is referred to as this.

It is important to understand that only member functions are passed a this pointer . For example, a friend does not have a this pointer.

Hàm f1() tự động đưa con trỏ tới ob-nơi đố tượng gọi lời gọi. Con trỏ này được dùng là this.

Để hiểu chỉ có một hàm thành phần được chuyển đến con trỏ this là quan trọng. Ví dụ, a firend không có con trỏ this.

EXAMPLE

As you have seen, when a member function refers to another member of a class, it does so directly, without qualifying the member with either a class or an objiect specification. For example, examine this short program , which creates a simple inventory class:

VÍ DỤ:

1. Như bạn thấy, khi mà một hàm thành phần dùng cho những thành phần khác của lớp, nó làm trực tiếp, không nói rõ thành phần với cả lớp và đối tượng xác định. Ví dụ, xem xét đoạn chương trình ngắn này, cái nào tao nên 1 lớp kiểm kê đơn giản:

```
// Demonstrate the this pointer .

#include <iostream>

#include <cstring>

using namespace std;
```

```
class inventory
{
     char item[20];
     double cost;
     int on_hand;
     public:
          inventory(char *I, double c, int o)
          {
               strcpy(item,i);
               cost=c;
               on_hand=o;
          }
     void show();
};

void inventory::show()
{
     cout<<item;
     cout<<":S"<<cost;
     cout<<"On hand:"<<on_hand<<"\n";
}
int main()
{
     inventory ob("wrench",4.95,4);
```

```
     ob.show();

     return 0;

}
```

As you can see , within the constructor inventory() and the member function show(), the member variables item, cost, and on_hand are referred to directly. This is because a member function can be called only in conjunction with an object'data is being referred to.

However, there is an even more subtle explanation. When a member function is called , it is automatically passed a this pointer to the object that invoked the call. Thus, the preceding program could be rewritten as shown here.

Như bạn thấy, trong h‡m thiết lập inventory() v‡ h‡m th‡nh phần show(), th‡nh phần có giá trị item(), cost, v‡ on_hand được dùng trực tiếp. Điều n‡y bởi vì một h‡m th‡nh phần có thể được gọi cùng chung với thông tin của đối tượng đang được dùng.

Tuy nhiên, có nhiều hơn sự giải thích mơ hồ. Khi một hàm thành phần được gọi, nó tự động chuyển con trỏ this đến đối tượng đã gọi lời gọi. Vì vậy, chương trình trước có thể được viết lại như sau:

```
// Demonstrate the this pointer

#include <iostream>

#include <cstring>

using namespace std;


class inventory

{

     char item[20];

     double cost;

     int on_hand;
```

```
     public:
          inventory(char*i, double c, int o)
          {
               strcpy(this->item,i);    //accesss
members
               this->cost=c;  //through the this
               this->on_hand=o;    //pointer
          }
     void show();
};

void inventory::show()
{
     cout<<this->item;   //use this to access members
     cout<<":S"<<this->cost;
     cout<<"On hand:"<<this->on_hand<<"\n";
}
int main()
{
     inventory ob("wrench",4.95,4);
     ob.show();
     return 0;
}
```

Here the member variable are accessed explicitly through the this pointer. Thus, within show(), these two statements are equivalent:

Đây là thành phần có giá trị được thông qua rõ rang con trỏ this. Vì vậy, trong show(), hai câu lệnh dưới đây là tương đương:

```
cost=123.23;

this->cost=123.23;
```

In fact, the first form is, loosely speaking, a shorthand for the second.

While no C++ programmer would use the this pointer to access a class member as just shown, because the shorthand form is much easier, it is important to understand what the shorthand implies.

The this pointer has several uses, including aiding in overloading operators. This use will be detailed in chapter ٧. For now, the important thing to understand is that by default, all member functions are automatically passed a pointer to the invoking object.

Thật sự, dạng đầu tiên, hiển thị dài dòng, cách viết nhanh cho dạng thứ hai.

Trong khi không có lập trình viên C++ nào dùng con trỏ this để truy cập một thành phần của lớp như được trình bày, bởi vì dạng viết nhanh dễ dàng hơn nhiều, thật là quan trọng để hiểu cách viết ngắn gợi ý điều gì.

Con trỏ this có vài cách dùng, bao gồm trợ giúp toán tử quá tải. Cách dùng này sẽ chi tiết hơn trong chương này. Vì lúc này, điều quan trọng để hiểu là do mặc định, tất cả hàm thành phần tự động chuyển con trỏ đến lời gọi đối tượng.

EXERCISE

Given the following program, convert all appropriate references to class members to explicit this pointer references.

BÀI TẬP

1. Cho đoạn chương trình dưới đây, chuyển đổi tất cả sự tham khảo thích hợp qua lớp thành phần để làm rõ ràng sự tham khảo con trỏ.

```
#include <iostream>

using namespace std;
```

```
class myclass
{
     int a,b;
     public:
          myclass(int n, int m)
          {
               a=m;
               b=m;
          }
     int add()
          return a+b;
     void show();
};

void myclass::show()
{
     int t;
     t=add();  //call member function
     cout<<t<<"\n";
}
int main()
{
     myclass ob(10,14);
     ob.show();
     return 0;
```

```
}
```

1.4. USING NEW AND DELETE - Cách dùng toán tử new và delete:

Up to now, when memory needed to be allocated, you have been using malloc(), and you have been freeing allocated memory by using free(). These are, of course, the standard C dynamic allocation functions. While these functions are available in C++, C++ provides a safer and more convenient way to allocate and free memory . In C++, you can allocate memory using new and release it using delete. These operators take these general forms:

Cho đến bây giờ, khi bộ nhớ cần được cấpp phát, bạn phải dùng malloc(), và bạn phải giải phóng bộ nhớ được cấp phát bằng cách sử dụng free(). Những điều này, dĩ nhiên, hàm cấp phát động trong chuẩn C. Trong khi những hàm có giá trị trong C++, C++ cung cấp cách an toàn hơn và tiện lợi hơn.để cấp phát và giải phóng bộ nhớ. Trong C++, bạn có thể cấp phát bộ nhớ sử dụng new và giải phóng nó bằng delete. Những toán tử này mang những dạng chung:

```
p-var=new type;

delete p-var;
```

Here type is the type specifier of the obj for which you want to allocate memory and p-var is a pointer to that type. new is an operator that returns a pointer to dynamically allocated memory that is large enough to hold an obj of type type. delete releases that memory when it is no longer needed. delete can be called only with a pointer previously allocated with new. If you call delete with an invalid pointer, the allocation system will be destroyed , possibly crashing your program.

Loại này là loai rõ ràng của đối tượng mà bạn muốn cấp phát bộ nhớ và p-var là một con trỏ dạng đó. new là một toán tử mà trả về một con trỏ để cấp phát bộ nhớ động đủ lớn để giữ một đối tượng của loại type. delete giải phóng bộ nhớ đó khi nó không cần thiết nữa. delete có thể được gọi chỉ với một con trỏ cấp phát trước đây với new. Nếu bạn gọi delete với một con trỏ không có giá trị, hệ thống cấp phát sẽ được phá hủy, có thể phá hỏng chương trình của bạn.

If there is insufficient available memory to fill an allocation request, one of two actions will occur. Either new will return a null pointer or it will generate an exception. (Exceptions and exception handling are described later in this book; loosely, an exception is a run-time error that can be managed in a structured fashion.) In Standard C++, the default behavior of new is to generate an exception when it can not satisfy an allocation request. If this exception is not handled by your program, your program will be terminated. The trouble is that the precise action that new takes on failure has been changed several times over the past few years. Thus, it is possible that your compiler will not implement new as defined by Standar C++.

Nếu bộ nhớ có giá trị không đủ để điền vào yêu cầu cấp phát, một hay hai hành động sẽ xảy ra. Ngay cả new sẽ trả về một con trỏ null hay là nó sẽ tạo ra một ngoại lệ. (Những sự ngoại lệ và sự điều khiển ngoại lệ được mô tả sau này; một cách dài dòng, một sự ngoại lệ là lỗi run-time mà có thể quản lý trong cấu trúc hình dáng.) Trong chuẩn C++, cách hành động mặc định của new là tạo một ngoai lệ khi nó không thể thỏa mãn yêu cầu cấp phát. Nếu sự ngoại lệ này không được điều khiển bởi chương trình của bạn, chương trình sẽ bị chấm dứt. Vấn đề là hành động chính xác mà new gánh lấy thất bại sẽ được thay đổi vài lần trong vài năm qua. Vì vậy, thật khả thi để trình biên dịch sẽ không thực thi new như được định nghĩa bởi chuẩn C++.

When C++ was first invented, new returned null on failure. Later this was changed such that new caused an exception on failure. Finally, it was decided that a new failure will generate an exception by default, but that a null pointer could be returned instead, as an option. Thus, new has been implemented differently at different times by compiler manufacturers. For example, at the time of this writting, Microsoft's Visual C++ returns a null pointer when new fails. Borland C++ generates an exception. Although all compiler will eventually implement new in compliance with Standard C++, currently the only way to know the precise action of new on failure is to check your compiler's documentation.

Khi C++ lần đầu tiên được phát minh, new trả về null trên thất bại. Sau này nó được thay đổi như là new gây ra sự ngoại lệ trên thất bại. Cuối cùng, nó được quyết định rằng sự thất bại new sẽ sinh ra một sự ngoại lệ bởi mặc định, nhưng con trỏ null đó có thể được trả về thay vì, như là một tùy chọn. Vì vậy, new thi hành khác nhau tại thời điểm khác nhau bởi nhà sản xuất trình biên dịch. Ví dụ, tại thời điểm của bài viết này, Microsoft's Visual C++ trả về một con trỏ null

when new thất bại. Borland C++ sinh ra sự ngoại lệ. Tuy nhiên tất cả trình biên dịch sẽ thậm chí thi hành new theo yêu cầu với chuẩn C++, hiện tại cách duy nhất để biết hành động rõ ràng của new trên thất bại là kiểm tra tài liệu của trình biên dịch.

Since there are two possible ways that new can indicate allocation failure, and since different compilers might do so differently, the code in this book will be in such a way that both contingencies are accommodated. All code in this book will test the pointer returned by new for null. This handles compilers that implement new by returning null on failure, while causing no harm for those compilers for which new throws an exception. If your compiler generates an exception when new fails, the program will simply be terminated. Later, when exception handling is described, new will be re-examined and you will learn how to better handle an allocation failure. You will also learn about an alternative form of new that always returns a null pointer when an error occurs.

Từ khi có hai cách khả thi thì new có thể chỉ ra sự cấp phát thất bại, và từ những trình biên dịch khác có thể làm khác nhau, code trong cuốn sách này sẽ như là một cách mà sự kiện ngẫu nhiên được giúp đỡ. Tất cả code trong cuốn sách này sẽ thử con trỏ trả về bởi new cho null. Sự điều khiển trình biên dịch này thực thi new bởi trả về null trên thất bại, trong khi không gây ra sự tổn hại nào cho những trình biên dịch mà new đưa ra một sự ngoại lệ. Nếu trình biên dịch của bạn sinh ra một ngoại lệ thì new thất bại, chương trình sẽ bị dừng lại. Sau nay, khi sự điều khiển ngoại lệ được diễn tả, new sẽ được xem xét lại và bạn sẽ học cách làm thế nào điều khiển tốt hơn sự cấp phát thất bại. Bạn cũng sẽ học về dạng thay thế của new mà luôn luôn trả về một con trỏ null khi mà một lỗi xuất hiện.

One last point: none of the examples in this book should cause new to fail, since only a handful of bytes are being allocated by any single program.

Althought new anh delete perform function similar to malloc() and free(), they have several advantages. First, new automatically allocates enough memory to hold an obj of the specified type. You do not need to use sizeof, for example, to compute the number of bytes required. This reduces possibility for error. Second, new automatically returns a pointer of the specified type. You do not need to use an explicit type cast the way you did when you allocated memory by using malloc() (see the following note). Third, both new and delete can be overloaded, enabling you to easily implement your own custom allocation system. Fourth, it is possible to initialize a dynamically allocated obj. Finally, you no longer need to include <cstdlib> with your programs.

Một điểm cuối cùng: không có ví dụ nào trong cuốn sách này gây nên new để thất bại, từ khi chỉ có một tí bytes được cấp phát bởi một chương trình độc lập nào đó.

Mặc dù new và delete cho thấy hàm giống như malloc() và free(), nó có vài thuận lợi. Thứ nhất, new tự động cấp phát đủ bộ nhớ để giữ một đối tượng của loại lý thuyết. Bạn không cần phải dùng sizeof, ví dụ , để tính toán số bytes cần thiết. Điều này giảm bớt khả năng có lỗi. Thứ hai, new tự động trả về một con trỏ của loại lý thuyết. Bạn không cần dùng loại rõ ràng tạo ra cách bạn làm khi bạn cấp phát bộ nhớ bằng cách sử dụng malloc() (xem lưu ý bên dưới). Thứ ba , cả new và delete có thể bị quá tải, cho phép bạn có thể dễ dàng thực thi hệ thống cấp phát tùy ý của chính bạn. Thứ tư, khởi tạo một đối tượng cấp phát động. Cuối cùng, bạn có thể không cần chứa <cstdlib> với chương trình của bạn nữa.

Note: *In C, no type cast is required when you are assigning the return value of malloc() to a pointer because the void* returned by malloc() is automatically converted into a pointer compatible with the type of pointer on the left side of the assignment. However, this is not the case in C++, which requires an explicit type cast when you use malloc(). The reason for this different is that it allows C++ to enforce more rigorous type checking.*

Now that new and delete have been introduced, they will be used instead of malloc() and free().

Lưu ý: Trong C, không có loại bố cục nào được cần khi mà bạn gán giá trị trả về của malloc() cho một biến con trỏ bởi vì void* trả về bởi malloc() được tự động chuyển đổi thành con trỏ tương thích với loại con trỏ ở bên trái của sự chỉ định. Tuy nhiên, điều này không phải là trường hợp trong C++, mà cần một loại bố cục rõ ràng khi bạn dùng malloc(). Lý do cho sự khác nhau này đó là nó cho phép C++ làm cho loại kiểm tra nghiêm ngặt hơn.

Bây giờ new và delete đã được giới thiệu, nó sẽ dùng thay thế cho malloc() và free().

EXAMPLES

As a short first example, this program allocates memory to hold an integer:

VÍ DỤ:

Như là một ví dụ ngắn, chương trình này cấp phát bộ nhớ để giữ một số nguyên:

```
// A simple example of new and delete.
#include <iostream>
using namespace std;

int main()
{
     int *p;
     p=new int;     //allocate room for an integer
     if(!p)
     {
          cout<<"Allocation error\n";
          return 1;
     }
     *p=1000;
     cout<<"Here is integer at p:"<<*p<<"\n";
     delete p; //release memory
     return 0;
}
```

Create a class that contains a person's name and telephone number. Using new, dynamically allocate an obj of this class and put your name and phone number into these fields within this obj.
What are the two ways that new might indicate an allocation failure?

Tạo một lớp mà chứa tên và số điện thoại của một người. Dùng new, cấp phát động một đối tượng của lớp này và đặt tên và số điện thoại của nó bên trong những vùng này trong đối tượng này.

Cho biết hai cách mà new có thể biểu thị một cấp phát thất bại?

1.5. MORE ABOUT NEW AND DELETE - Mở Rộng của new và delete:

*This section discusses two additional features of **new and delete.** First, dynamically allocated objects can be given initial values. Second, dyanmically allocated arrays can be created.*

Phần này thảo luận thêm 2 đặc điểm của toán tử **new và delete.** Trước hết, các đối tượng được chia phần linh động có thể được khởi tạo giá trị ban đầu.Thứ hai là, các mảng động có thể được tạo ra.

*You can give a dynamically allocated object an intial value by using this form of the **new** statement:*

Bạn có thể khởi tạo cho đối trượng cấp phát linh động một giá trị ban đầu bằng cách sử dụng khai báo **new** như sau:

```
p-var =new type (intial-value);
```

*To dynamically allocate a one-dimensional array, use this form of **new:***

Để cấp phát linh động cho 1 mạng động, sử dụng hình thức sau của **new:**

```
p-var= new type[size];
```

After this statement has excuted, pvar will point to the start of an array of size element of the type specified. For various technical reasons, it is not possible to initialize an array that is dynamically allocated.

Sau khi câu lệnh được thi hành, pvar sẽ chỉ vào vị trí phần tử đầu tiên của mảng theo định nghĩa. Vì nhiều lý do của công nghệ, sẽ không phù hợp khi khởi tạo một mảng mà được cấp phát động.

*To delete a dynamically allocated array, use this from of **delete** :*

Để xóa một mảng cấp phát động, sử dụng toán tử **delete:**

```
Delete [] p-var;
```

This syntax causes the compiler to call the destructor function for each element in the array. It does not cause p-var to be free multiple times. P-var is still freed only once.

Cấu trúc này là nguyên nhân mà trình biên dịch gọi phương thức phá hủy cho mỗi phần tử trong mảng. Đó không phải là nguyên nhân p-var được giải phóng nhiều lần. p-var vẫn chỉ được giải phóng 1 lần.

Note : *For older compilers, you might need to spectify the size of the array that you are deleting between the squre brackets of the* ***delete*** *statement. This was required by the original definition of C++. However, the size specification is not needed by modern compilers.*

Chú ý: Đối với các trình biên dịch cũ, bạn có thể cần phải xác định kích thước của mảng mà bạn đang xóa giữa dấu ngoặc vuông của câu lệnh **delete.** Đó là một yêu cầu bởi sự định nghĩa gốc trong C++. Tuy nhiên, đối với trình biên dịch mới thì sự đặc tả kích thước không cần phải có.

Examples:

This program allocates memory for an integer and initializes that memory:

Đây là chương trình cấp phát bộ nhớ cho một số integer và khởi tạo giá trị cho địa chỉ đó.

```
#include "iostream.h"
using namespace std;
int main()
{
          int *p;
          p=new int(9);
          if(!p)
          {
               cout<<" Allocation error\n";
               return 1;
```

```
        }
        cout<<" Here is integer at p :"<<"*<<"\n;
        delete p;
        return 0;
}
```

As you should expect, this program displays the value 9, which is the initial value given to the memory point to by ***p.***

Như bạn mong đợi, chương trình sẽ hiện thị giá trị 9, là giá trị của ô nhớ mà được trỏ tới bởi **p.**

The following program passes initial values to a dynamically allocate object:

Chương trình dưới bỏ qua giá trị khởi tạo để cấp phát linh động cho đối tượng

```
#include "iostream.h"
using namespace std;
class samp
{
        int i,j;
        public :
                samp(int a,int b) { i=a;j=b;}
                int get_product () {return i*j;}
};
int main()
{
        samp *p;
        p=new sam(6,5);
        if(!p)
        {
```

```
            cout<<" Allocation error\n";
            return 1;
        }
        cout<<"Product is "<<p->get_product()<<"\n";
        delete p;
        return 0;
}
```

*When the **samp** object is allocated, its constructor is automatically called and is passed the value 6 and 5.*

Khi mà đối tượng **samp** được cấp phát, phương thức khởi tạo của nó tự động được gọi và nhận được giá trị là 6 và 5.

The following program allocates an array of integer.

Chương trình dưới cấp phát cho một mảng số nguyên.

```
#include "iostream.h"
using namespace std;
int main()
{
        samp *p;
        p=new int[5];
        if(!p)
        {
            cout<<" Allocation error\n";
            return 1;
        }
        int I;
        for(i=0;i<5;i++) p[i]=i;
```

```
            for(i=0;i<5;i++){
            cout<<" Here is integer at p["<<i<<""] :";
            cout<<p[i]<<"\n";
            }
            delete []p;
            return 0;
}
```

The program displays the following:

```
Here is integer at p[0] : 0
Here is integer at p[0] : 1
Here is integer at p[0] : 2
Here is integer at p[0] : 3
Here is integer at p[0] : 4
```

The following program creates a dynamic array of objects:

Chương trình khởi tạo một mảng động các đối tượng.

```
#include "iostream.h"
using namespace std;
class samp
{
          int i,j;
          public :
               void set_if(int a,int b) { i=a;j=b;}
               int get_product () {return i*j;}
};
```

```
int main()
{
          samp *p;
          p=new sam[10];
          int i;
          if(!p)
          {
               cout<<" Allocation error\n";
               return 1;
          }
 for(i=0;i<10;i++)
               p[i].set_if(i,i);
          for(i=0;i<10;i++)
          {
               cout<<"Product [ <<"i"<<"]is ";
               cout<<"p[i].get_product()<<"\n";
     }

          delete []p;
          return 0;
}
```

This program displays the following :

```
Product [0] is :0
Product [0] is :1
Product [0] is :4
```

```
Product [0] is :9

Product [0] is :16

Product [0] is :25

Product [0] is :36

Product [0] is :49

Product [0] is :64

Product [0] is :81
```

*The following version of the preceding program gives samp a destructor, and now when **p** is freed. Each element's destructor is called:*

Phiên bản dưới của chương trình trên cung cấp thêm cho lớp samp phương thức phá hủy, và khi **p** được giải phóng là mỗi lần thành phần phá hủy được gọi.

```
#include "iostream.h"
using namespace std;
class samp
{
          int i,j;
          public :
               void set_if(int a,int b) { i=a;j=b;}
               ~samp(){ cout <<" destroying …\n";}
               int get_product () {return i*j;}
};
int main()
{
          samp *p;
          p=new sam[10];
```

```
            int i;
            if(!p)
            {
                  cout<<" Allocation error\n";
                  return 1;
            }
for(i=0;i<10;i++)
                  p[i].set_if(i,i);
            for(i=0;i<10;i++)
            {
                  cout<<"Product <<"i"<<"is ";
                  cout<<"p[i].get_product()<<"\n";
}

            delete []p;
            return 0;
}
```

This program displays the following :

Kết quả chương trình như sau:

```
Product [0] is :0
Product [0] is :1
Product [0] is :4
Product [0] is :9
Product [0] is :16
```

```
Product [0] is :25

Product [0] is :36

Product [0] is :49

Product [0] is :64

Product [0] is :81

Destroying…

Destroying…

Destroying…

Destroying…

Destroying…

Destroying…

Destroying…

Destroying…

Destroying…

Destroying…
```

As you can see, samp's destructor is called ten times=once for each element in the array.

Như các bạn thấy, phương thức phá hủy của **samp** được gọi 10 lần cho mỗi một phần tử trong mảng.

Exercises

Sho**w how to convert the following code into its equivalent that uses **new.

Hãy chỉ ra cách chuyển đoạn code sau sử dụng toán tử **new.**

```
char *p;

p=(char*)malloc(100);
```

```
//…

strcpy(p,”this is the test”);
```

Hint: A string is simply an array of characters.

Ghi chú: Một chuỗi đơn giản là một mảng các ký tự.

*Using **new,** show how to allocate a **double** and give it an initial value of -123.0987*

Sử dụng toán tử new, trình bày cách cấp phát bộ nhớ cho một số nguyên dài và cho nó một giá trị ban đầu là -123.0987.

1.6. REFERENCES - Tham chiếu:

C++ contains a feature that is related to the pointer: the reference. A referance is an implicit pointer that for all intents and purposes acts like another name for a variable. There are three ways that a reference can be used. First, a reference can pass to a function. Second, a reference can be returned by a function. Finally, an independent reference can be created, Each of these application of the reference is examined, beginning with reference parameters.

C++ bao gồm một đặc điểm mà liên quan đến con trỏ : đó là tham chiếu. Tham chiếu là một loại con trỏ ẩn dùng với mục đích biểu diễn một tên khác cho biến. Có 3 cách sử dụng tham chiếu. Một là, tham chiếu co thể là một hàm chức năng.Hai là, tham chiếu có thể được trả về bởi hàm. Cuối cùng, một tham chiếu độc lập co thể được tao. Mỗi ứng dụng của tham chiếu đều được kiểm tra, bắt đầu với thông số của tham biến.

Without a doubt, the most important use if a refernce is as a parameter to a function. To help you understand what a reference parameter is and how it works, let’s first start with a program that uses a pointer (not a reference)as a parameter:

Không nghi ngờ, điều quan trọng nhất là sử dụng tham chiếu như là một tham biến trong hàm. Để giúp bạn hiểu rõ tham chiếu là như thế nào và chúng hoạt động ra sao, chúc ta bắt đầu với 1 chương trình sử dụng con trỏ(không phai là

tham chiếu) như là một tham biến.

```
#include "iostream.h"
using namespace std;
void f(int *n)
int main()
{
        int i=0;
        f(&i);
        cout<<"Here is i's new value :"<<i<<"\n";
        return 0;
}
void f(int*n)
{
        *n=100;
}
```

*Here **f()** loads the value 100 into the integer pointed to by **n.** In this program, **f()** is called with the address of **i in main().** Thus, after **f()** return, **I** contains the value 100.*

Hàm **f()** ở đây nạp giá trị 100 vào con trỏ số nguyên bởi n. Trong chương trình, **f()** được gọi với địa chỉ của i trong hàm main().

This program demonstrates how a pointer is used as a parameter to manually create a call-by-reference parameter-passing mechanism. In a C program, this is the only way to achieve a call-by-reference.

Đó là chương trình mô tả con trỏ hoạt động như là 1 tham biến như thế nào để tạo một biến được gọi bởi tham chiếu. Trong chương trình C, đó là cách duy nhất để đạt được sự "gọi theo tham khảo".

However, IN C++, you can completely automate this process by using a reference parameter.To see how, let's rework the previous program. Here is a version that uses a reference parameter:

Tuy nhiên, trong C++, bạn có thể tự hoàn thành quá trình này bằng cách sử dụng các tham số truyền theo địa chỉ. Làm thế nào để thấy rõ điều đó, hãy trở lại chương trình phía trước. Đây là phiên bản của chương trình mà sử dụng tham chiếu:

```
#include <iostream>

Using namespace std;

void f(int &n)

int main()

{

          int i=0;

          f(i);

          cout<<"Here is i's new value :"<<i<<"\n";

          return 0;

}

void f(int &n)

{

          n=100;

}
```

Examine this program carefully. First, to declare a reference variable or parameter, you precede the variable's name with the &. This is how ***n*** *is declared as parameter to* ***f().*** *Now that* ***n*** *is a reference. It is no longer necessary- or even legal- to apply the operator. Instead, each time* ***n*** *is used within* ***f(),*** *it is automatically treated as a pointer to the argument used to call* ***f().*** *This means that the statement "n=100;" actually puts the value 100 into the variable used to call* ***f(), w****hich, in this case, is* ***i.*** *Further, when* ***f()*** *is called,*

*there is no need to precede the argument with the &. Instead, because **f()** is declared as taking a reference parameter, the address to the argument is automatically passed to **f().***

Kiểm tra chương trình thật kỹ. Trước tiên, biểu thị một tham chiếu là biến số hay tham số, bạn điền phía trước tên của biến số "&". Điều đó có nghĩa là **n** được định nghĩa là một tham số của **f(). Bây** giờ n là một tham chiếu. Không cần thiết hoặc thậm chí đúng luật để cấp một toán tử. Thay vào đó, mỗi lần **n** được sử dụng trong **f(),** nó sẽ được tự động xem là một con trỏ- đối số để gọi **f().** Câu lệnh n=100 có nghĩa thật sự là đặt giá trị 100 vào biến số sử dụng để gọi **f(),** trong trường hợp này là **i.** Hơn nữa, khi hàm **f()** được gọi, không cần có phía trước đối số "&". Thay vào đó, vì hàm **f()** được công bố như là sử dụng các tham số truyền theo địa chỉ, địa chỉ của đối số tự động được gọi vào trong **f().**

*To review, when you use a reference parameter, the compiler automatically passed the address of the variable used as the argument. There is no need no manually generate the address of the argument by preceding it with an & (in fact,it is not allowed). Further, within the function, the compiler automatically uses the variable pointer to by the reference parameter. There is no need to employ the * (and again, it is not allowed). Thus, a reference parameter fully automates the call-by-reference parameter-passing mechanism.*

It is important to understand that you cannot change what a reference is pointing to . For example, if the statement

n++:

*were put inside **f()** in the preceding program, **n would** still be pointing to **I** in **main().** Instead of incrementing **n, this** statement increments the value of the variable being referenced(in this case, **i).***

Để nhìn lại, khi bạn sử dụng một tham số truyền theo địa chỉ, trình biên dịch sẽ tự động điền địa chỉ của biến số vào như là đối số. Không cần phải tạo ra 1 địa chỉ cho đối số bằng cách thêm vào trước nó "&"(sự thật, nó không được cho phép). Hơn nữa, trong hàm, trình biên dịch tự động sử dụng các biến con trỏ như là các tham số truyền theo địa chỉ. Cũng không cần thiết sử dụng toán tử *(nó không được phép). Do đó, tham số truyền theo địa chỉ tự động gọi "call-by-reference parameter-passing mechanism".

Thật quan trọng để hiểu rằng bạn không thể thay đổi cái gì mà tham chiếu trỏ tới. Ví dụ như câu lệnh sau :

n++;

được đặt vào trong hàm **f()** trong chương trình trước, **n** vẫn sẽ chỉ vào **i** trong hàm main(). Thay vì tăng một trị số của n, câu lệnh này tăng giá trị của biến số được tham chiếu tới (trong trương hợp này là i).

Reference parameters offer several advantages over their (more or less) equivalent pointer alternatives. First, from a practical point of view, you no longer need to remember to pass the address of an argument. When a reference parameter is used, the address is automatically passed. Second, in the opinion of many programmers, reference parameters offer a cleaner, more elegant interface than the rather clumsy explicit pointer mechanism. Third, as you will see in the next section, when an object is passed to a function as a reference, no copy is made. This is one way to eliminate the troubles associated with the copy of argument damaging something needed elsewhere in the program when its destructor function is called.

Tham số truyền theo địa chỉ có một vài điểm thuận tiện hơn con trỏ. Trước hết, trên quan điểm thực hành, bạn không cần phải nhớ địa chỉ của đối số. Khi mà tham số truyền theo địa chỉ được dùng, địa chỉ sẽ tự động được thông qua. Thứ hai, theo ý kiến của nhiều lập trình viên, tham số truyền theo địa chỉ thì có giao diện tao nhã rõ ràng hơn. Đó là một cách loại bỏ những phức tạp kết hợp với những đối số giống nhau làm hại nhưng thứ cần khi mà trong chương trình gọi phương thức phá hủy được gọi.

Example

*The classic example of passing arguments by reference is a function that exchanges the value of the two argument with which it is called. Here is an example called **swapargs()** that uses references to swap its two integer arguments.*

Một ví dụ điển hình của việc bỏ qua đối số bằng cách sử dụng tham chiếu là hàm hoán vị 2 giá trị của đối số mỗi khi nó được gọi. Đây là ví dụ gọi hàm **swaparg()** mà sử dụng tham chiếu để hoán vị 2 giá trị số nguyên.

```
#include <iostream>

using namespace std;

void swapargs( int &x, int &y);
```

```
int main()
{
     int i,j;
     i=10;
     j=19;
     cout<<"i:  "<<i<<";";
     cout<<"j:  "<<j<<"\n";
     swapargs(i,j);
     cout<<"After swap :";
     cout<<"i:  "<<i<<";";
     cout<<"j:  "<<"\n";
     return 0;
}
void swapargs(int &x,int &y)
{
     int t;
     t=x;
     x=y;
     y=t;
}
```

*If **swapargs()** had been written using pointer instead of references, it would have looked like this:*

Nếu hàm **swapargs()** được viết bằng con trỏ thay cho tham chiếu, thì nó sẽ như thế này:

```
void swapargs(int *x,int *y)
{
 int t;
 t=*x;
 *x=*y;
 *y=t;
}
```

*As you can see, by using the reference version of swapargs(), the need for the *operator is eliminated.*

Như bạn thấy, bằng cách sử dụng tham chiếu, đã loại bỏ được toán tử *.

2. *Here is program that uses the **round()** function to round a **double value.** The value to kbe rounded is passed by reference.*

Đây là chương trình sử dụng hàm **round()** để làm tròn giá trị **double.** Giá trị được làm tròn được bỏ qua bởi tham chiếu

```
#include <iostream>
#include <cmath>
using namespace std;
void round(double &num);
int main()
{
     double i=100.4;
     cout<<i<<"rounded is ";
     round(i);
     cout<<i<<"\n";
     i=10.9;
```

```
        cout<<i<<"rounded is ";
        round(i);
        cout<<i<<"\n";
        return 0;
}
void round(double&num)
{
        double frac;
        double val;
        frac=modf(num,&val);
        if(frac<0.5) num=val;
        else
              num=val+1.0;
}
```

round() *uses a relatively obscure standard library function called* ***modf()*** *to decompose a number into its whole number and fractional parts. The fractional part is returned; the whole number is put into the variable pointed to by* ***modf()'s*** *second parameter.*

Hàm **round()** sử dụng thư viện chuẩn của hàm gọi lệnh **modf()** để phân tách số thành phần nguyên và phần thập phân. Phần thập phân được trả về, phần nguyên được đặt vào biến số trỏ vào bởi tham số thứ 2 của hàm **modf().**

Exercises

Write a function called ***neg()*** *that reverses the sign of its integer parameter. Write the function two ways-first by using a pointer parameter and then by using a reference parameter. Include a short program to demonstrate their operation.*

Viết hàm **neg()** để đảo dấu của tham số nguyên. Viết hàm bằng hai cách- sử dụng con trỏ và sử dụng tham số truyền theo địa chỉ.Bao gồm cả chương trình nhỏ mô tả phép thực hiện đó.

What is wrong with the following program?

Chỉ ra điểm sai?

```
#include <iostream>

using namespace std;

void triple(double &num);

int main()

    {

           Double d=7.0;

           triple(&d);

           cout<<i;

           return 0;

    }

    void triple(double &num)

    {

           num=3*num;

    }
```

3. Give some advantages of reference parameters.

Điểm thuận lợi của tham số truyền theo địa chỉ.

1.7. PASSING REFERENCES TO OBJECTS – Truyền tham chiếu cho đối tượng:

As you learned in Chapter 3, when an object is passed to a function by use of the default call-by-value parameter-pasing mechnism, a copy of that object is made although the parameter's constructor function is not called. Its destructor function is called when the function returns. As you should recall, this can cause serious problems in some instances-when the destructor frees dynamic memory, for example.

Khi bạn tìm hiểu chương 3, khi mà một đối tượng được chuyển thành hàm bằng cách sử dụng phương thức gọi hàm tham số truyền theo địa chỉ mặc đình, một bản sao của đối tượng được tạo ra mặc dù phương thức thiết lập của tham số không được gọi. Phương thức phá hủy của nó được gọi khi hàm trả về. Khi mà bạn gọi lại hàm, đó là nguyên nhân gây ra lỗi nghiêm trọng trong một vài trường hợp- khi mà phương thức phá hủy giải phóng bộ nhớ động.

One solution to this problem is to pass an object by reference.(The other solution involves the use of copy constructors, which are discussed in Chapter 5).When you pass the object by reference, no copy is made, and therefore its destructor function is not called when the function returns. Remember, however, that changes made to the object inside the function affect the object used as the argument.

Một giải pháp để giải quyết vấn đề là chuyển đối tường thành tham chiếu.(Một giải pháp khác để giải quyết vấn đề là sử dụng bản sao của phương thức phá hủy, mà chúng ta đã thảo luận trong chương 5.Khi bạn chuyển 1 đối tượng bằng tham chiếu, việc tạo bản sao sẽ không được thực hiện, thậm chí phương thức phá hủy của nó cũng không được gọi khi hàm trả về.Nên nhớ rằng, việc thay đổi như thế làm đối tượng bên trong hàm bị ảnh hưởng bởi đối tượng dùng như là một đối số.

Note: *It is critical to understand that a reference is not a pointer. Therefore, when an object is passed by reference, the member access operator remains the dot (.) not the arrow (->).*

Chú ý:

Thật sai lầm khi hiểu rằng tham chiếu không phải là con trỏ. Do đó, khi mà đối tượng được chuyển sang tham chiếu, các thành phần được truy cập bởi toán tử ".". Chứ không phải là toán tử "->".

Example

The following is an example that demonstrates the usefulness of passing an object by reference. First, here is a version of a program that passes an object of ***myclass*** *by value to a function called* ***f():***

Ví dụ dưới đây mô tả sự hữu ý của việc chuyển đối tượng thành tham chiếu. Trước hết, đây là phiên bản của chương trình chuyển đối tượng của lớp myclass bởi giá trị của hàm **f()**

```
#include <iostream>
```

```
using namespace std;
class myclass
{
     int who;
public:
     myclass(int n){
          who=n;
          cout<<"Constructing "<<who<<"\n";
     }
     ~myclass()
     {
          cout<<"Destructing "<<who<<"\n";
     }
     int id(){return who;}
};
void f(myclass o)
{
     cout<<"Receive "<<o.id<<"\n";
}
int main()
{
     myclass x(1);
     f(x);
     return 0;
}
```

This function displays the following:

Kết quả hiện thị:

```
Contructing 1

Receive 1

Destructing 1

Destructing 1
```

*As you can see, the destructor function is called twice-first when the copy of object 1 is destroyed when **f()** terminates and again when the program finishes.*

Như bạn thấy, phương thức phá hủy được gọi 2 lần trước khi bản sao của đối tượng 1 được phá hủy khi mà hàm **f()** kết thúc và khi chương trình kết thúc.

*However, if the program is changed so that **f() uses** a reference parameter, no copy is made and, therefore, no destructor is called when **f()** return:*

Tuy nhiên, nếu chương trình được thay đổi để hàm **f()** dùng tham số truyền theo địa chỉ, sẽ không có bản sao được tao ra, và do đó, không có phương thức phá hủy được gọi khi hàm f() trả về.

```
#include <iostream>

using namespace std;

class myclass

{

     int who;

public:

     myclass(int n){

          who=n;

          cout<<"Constructing "<<who<<"\n";
```

```
        }
        ~myclass()
        {
              cout<<"Destructing "<<who<<"\n";
        }
        int id(){return who;}
};
void f(myclass &o)
{
        cout<<"Receive "<<o.id()<<"\n";
}
int main()
{
        myclass x(1);
        f(x);
        return 0;
}
```

This function displays the following:

Kết quả hiện thị.

```
Contructing 1
Receive 1
Destructing 1
```

Remember

when accessing members of an object by using a reference, use the dot operator; not the arrow.

Nhớ rằng khi mà truy cập các thành phần của đối tượng sử dụng tham chiếu, sử dụng toán tử "." Chứ không phải là toàn tử mũi tên"->"

Exercise

1. *What is wrong with the following program ? Show how it can be fixed by using a reference parameter.*

Tìm chỗ sai trong chương trình? Chỉ ra cách khắc phục sử dụng tham số truyền theo địa chỉ.

```
#include <iostream>

#include <cstring>

#include <cstdlib>

using namespace std;

class strtype{

     char*p;

public:

     strtype(char*s);

     ~strtype(){delete []p;}

     char *get(){return p;}

};

strtype::strtype(char*s)

{

     int l;

     l=strlen(s)-1;

     p=new char[1];

     if(!p)
```

```
        {
            cout<<"Allocation is error\n";
            exit(1);
        }
        strcpy(p,s);
    }
    void show(strtype x)
    {
        char*s;
        s=x.get();
        cout<<s<<"\n";
     }
    int main()
    {
        strtype a("Hello"), b("There");
        show(a);
        show(b);
        return 0;
     }
```

1.8. RETURNING REFERENCES - Trả về tham chiếu:

A function can return a reference. As you will see in Chapter 6, returning a reference can be very useful when you are overloading certain types of operators. However, it also can be employed to allow a function to be used on the left side of an assignment statement. The effect of this is both powerful and startling.

Một hàm có thể trả về một tham chiếu. Như bạn sẽ thấy ở chương 6. việc trả về một tham chiếu có thể rất hữu dụng khi mà bạn nạp chồng vào một số loại toán toán tử nào đó. Tuy nhiên, nó cũng có thể được tận dụng để cho phép một hàm được sử dụng bên trái của một câu lệnh gán. Tác động của nó thì rất lớn và đáng chú ý.

EXAMPLES:

VÍ DỤ:

To begin, here is very simple program that contains a function that returns a reference:

Để bắt đầu, đây là một chương trình đơn giản bao gồm một hàm trả về một tham chiếu:

```
// A simple example of a function returning a reference

#include ”iostream.h”

Using namespace std;

int &f(); //return a reference

int x;


int main()

{

     f()=100; //assign 100 to reference to returned by
f()

     cout<<x<<”\n”;
```

```
     return 0;
}
//return an int reference


int &f()
{
     return x; //return a reference to x;
}
```

*Here function **f()** is declared as returning a reference to an integer. Inside the body of function, the statement.*

Ở đây, hàm **f()** được khai báo có chức năng là trả về một tham chiếu cho một số nguyên. Bên trong thân hàm có câu lệnh:

```
return x;
```

*does not return the value of the global variable **x,** but rather, it automatically return x's address (in the form of a reference). Thus, inside **main(),** the statement.*

mà không trả về một giá trị của biến toàn cục **x,** nhưng dĩ nhiên nó sẽ tự động trả về địa chỉ của x (dưới hình thức là một tham chiếu). Vì vậy, bên trong thân hàm **main(),** câu lệnh

f()=100;

*To review, function **f()** returns a reference. Thus, when **f()** is used on the left side f the assignment statement, it is this reference, returned by **f(),** that is being assigned to. Since f() returns a reference to **x (**in this example), it is **x** that reviews the value 100.*

Để xem lại, hàm f() trả về một tham chiếu. Do đó, khi hàm f() được sử dụng trong câu lệnh gán, thì tham chiếu được trả về bởi hàm f() sẽ được gán. Kể từ khi hàm f() trả về một tham chiếu cho **x** (trong ví dụ này), thì x sẽ nhận giá trị 100.

2. *You must be careful when returning a reference that the object you refer to does not go out of scope. For example, consider this slight reworking of function **f():***

Bạn phải thận trọng khi trả về một tham chiếu rằng đối tượng mà bạn đang đề cập đến không vượt ra tầm vực. Ví dụ, xem lại cách hoạt động của hàm **f():**
//return an int reference

```
int &f()

{

     int x;  //x is now a local variable

     return x;  //returns a reference to x

}
```

*In this case, **x** is now local to **f()** and will do out of scope when **f()** returns. This effectively means that the reference returned by **f()** is useless.*

Trong trường hợp này, x bây giờ thuộc về hàm f() và sẽ hoạt động bên ngoài tầm vực khi mà f() trả về. Trên thực tế, điều này có nghĩa là tham chiếu trả về bởi **f()** là vô nghĩa.

NOTE:

Some C++ compilers will not allow you to return a reference to a local variable. However, this type of problem can manifest itself in the other ways. Such as when objects are allocated dynamically.

One very good use of returning a reference is found when a bounded array type is created. As you know, in C and C++, no array boundary checking occurs. It is therefore possible to overflow or underflow an array. However, in C++, you can create an array class that performs automatic bounds checking. An array class contains two core functions-one that stores information into the array and one that retrieves information. These function can check, at run time, that the array boundaries are not overrun.

Một trong những lợi ích của việc trả về một tham chiếu được phát hiện khi mà một loại mãng có giới hạn được tạo. Như bạn biết, trong C và C++, không có sự giới hạn bởi đường biên của mảng. Vì thế có thể tràn hoặc là thiếu hụt mảng. Tuy nhiên, trong C++, bạn có thể tạo một lớp mảng mà có thể tự động thực thi việc kiểm tra giới hạn. Một lớp mảng bao gồm 2 hàm lõi – một là lưu trữ thông tin bên trong mảng, và một là dùng để lấy thông. Khi chạy, những hàm này có thể kiểm tra, là không vượt qua các đường biên của mảng.

The following program implements a bounds-checking array for characters:

Chương trình sau đây thực hiện việc kiểm tra đường giới hạn của mảng những kí tự:

```
#include "stdafx.h"
#include <iostream>
#include<cstdlib>
using namespace std;
class array
{
     private:
          int size;
          char *p;
     public:
          array(int num);
          ~array() {delete []p;}
          char &put(int i);
          char get (int i);
};

array ::array(int num)
{
     p=new char[num];
     if(!p)
     {
          cout<<"Allocation error";
```

```
            exit(1);
        }
        size=num;
}

//put something into the array
char &array::put (int i)
{
        if(i < 0 || i<=size)
        {
            cout<<"Bonus error!!!\n";
            exit (1);
        }
        return p[i];//return reference to p[i]
}

//Get something from the array
char array ::get(int i)
{
        if(i<0 || i>=size)
        {
            cout <<"Bonus error !!!\n";
            exit(1);
        }
        return p[i];//return character
```

```
}

int main()
{
      array a(10);
      a.put(3)='X';
      a.put(2)='R';

      cout <<a.get(3)<<a.get(2);
      cout<<"\n";

      //now genrate run-timr boundary error
      a.put(11) = '!';

      return 0;
}
```

*This example is a practical use of functions returning references, and you should exam it closely. Notice that the **put()** function returns a reference to the array element specified by parameter **i.** This reference can then be used on the left side of an assignment statement to store something in the array-if the index specified by **i** is not out of bounds. The reverse is **get(),** which returns the value stored at the specified index if that index is within arrange. This approach to maintaining an array is sometimes referred to as a safe array. (You will see a better way to create a safe array later on, in Chapter 6.)*

Ví dụ này là một tiện ích thực dụng của hàm trả về tham chiếu, và bạn nên xem xét nó kĩ lưỡng. Chú ý rằng hàm **put()** trả về một tham chiếu cho phần tử của mảng được chỉ định bởi **i.** Sau đó tham chiếu này có thể được sử dụng bên trái của một lệnh gán để lưu trữ vài thứ trong mảng – Nếu index được chỉ định bởi **i** thì không nằm ngoài đường ranh giới. Trái lại là hàm **get()**, nó trả về một giá trị được.

*One other thing to notice about the preceding program is that the array is allocated dynamically by the use of **new.** This allows arrays of differing lengths to be declared.*

As mentioned, the way that bounds checking is performed in this program is a practical application of C++. If you need to have array boundaries verified at run time, this is one way to do it. However, remember that bounds checking slows access to the array. Therefore, it is best to include bounds checking only when there is a real likelihood that an array boundary will be violated.

Write a program that creates a two-by-three-dimensional safe array of integers. Demonstrate that works.

Viết một chương trình tạo một mảng số nguyên an toàn 2 chiều …? Chứng tỏ nó hoạt động.

Is the following fragment valid? If not, why not?

Đoạn chương trình sau có hợp lệ không? Nếu không, tại sao không?

```
int &f()

{

}

int *x;

x=f();
```

1.9. INDEPENDENT REFERENCES AND RESTRICTIONS – THAM CHIẾU ĐỘC LẬP VÀ NHỮNG HẠN CHẾ :

Although not commonly used, the independent reference is another type of reference that is available in C++. An independent reference is a reference variable that is all effects is simply another name for another variable. Because references cannot be assigned new values, an independent reference must be initialized when it is declared.

Mặc dù không được thường xuyên sử dụng, nhưng *tham chiếu độc lập* là một loại khác của tham chiếu mà có thể được sử dụng trong C++. Một tham chiếu độc lập là một biến tham chiếu mà có những tác động đơn giản đối với những tên biến khác nhau thì cho biến số khác nhau. Bởi vì những tham chiếu không thể được gán giá trị mới, một tham chiếu độc lập phải được khởi tạo ngay khi được khai báo.

NOTE: *because independent references are sometimes used, it is important that you know about them. However, most programmers feel that there is no need for them and that they can add confusion to a program. Further, independent references exist in C++ largely because there was no compelling reason to disallow them. But for the most part, their use should be avoided.*

Chú ý: Thỉnh thoảng tham chiếu độc lập được sử dụng, vì vậy bạn cần phải biết về chúng. Tuy nhiên, hầu hết lập trình viên cảm thấy rằng không cần đến chúng và chúng chỉ đem lại rắc rối cho chương trình. Hơn thế nữa, một tham chiếu độc lập tồn tại trong C++ nhiều hơn bởi vì không có bất cứ một lí do thuyết phục nào không thừa nhận chúng. Nhưng thông thường nên tránh việc sử dụng chúng.

There are a number of restrictions that apply to all types of references. You can not reference another reference. You cannot obtain the address of a reference. You cannot create an arrays of references, and you cannot reference a bit-field. References must be initialized unless they are members of class, are return values, or are function parameters.

Có một số các hạn chế khi áp dụng đến tất cả các loại tham chiếu. Bạn không thể biết được tham chiếu này với tham chiếu khác. Bạn không thể lấy được địa chỉ của tham chiếu độc lập. Bạn không thể tạo một mảng tham chiếu, và bạn không thể tham chiêu đến một trường bit. Những tham chiếu phải được khởi tạo nếu nó không là thành phần của một lớp, và trả về những giá trị, hoặc là những hàm tham số.

EXAMPLES:

Ví dụ:

Here is a program that contains an independent reference:

Đây là một chương trình chứa một tham chiếu độc lập:

```
#include "stdafx.h"
```

```
#include <iostream>
using namespace std;

int main()
{
  int j, k;
  int &i = j;    // independent reference

  j = 10;

  cout << j << " " << i; // outputs 10 10

  k = 121;
   i = k;         // copies k's value into j, not k's
address

  cout << "\n" << j;  // outputs 121

  return 0;
}
```

*In this program, the independent reference **ref** serves as a different name for **x**. Form a practical point of view, **x** and **ref** are equivalent.*

Trong chương trình này, tham chiếu độc lập **ref** được đối xử như là một cái tên khác của x. Hình thành nên một quan điểm thực tế tổng quan là **x** và **ref** thì tương đương nhau.

An independent reference can refer to a constant. For example, this is valid:

Một tham chiếu độc lập có thể tham chiếu đến một hằng số. Ví dụ, đây là hợp lệ:
const int & ref=10;

EXERCISE:

Ví dụ:

One your own, try to think of a good use for an independent reference

Thêm nữa, tham chiếu độc lập còn có một lợi ích nhỏ, mà bạn có thể thấy từ giờ trong những chương trình khác.

Skills check:

Những kĩ năng kiểm tra

At this point, you should be able to perform the following exercises and answer the questions.

Vào thời điểm này, bạn nên thực hành một số bài tập bên dưới và trả lời các câu hỏi sau.

Given the following class, create a two-by-five two-dimensional array and give each object in the array an initial value of your own choosing. Then display the contents of the array.

Hãy thay đổi phương pháp của bạn để giải quyết vấn đề trên, sao cho truy cập mảng bằng cách sử dụng một con trỏ.

```
class a_type
{
     double a,b;
     public:
          a_type(double x, double y)
          {
               a=x;
```

```
            b=y;
        }
        void show() (cout <<a<<' '<<b<<"\n";};
};
```

Modify your solution to the preceding problem so it accesses the array by using a pointer.

Hãy thay đổi phương pháp của bạn để giải quyết vấn đề trên, sao cho truy cập mảng bằng cách sử dụng một con trỏ.

*What is the **this** pointer?*

Con trỏ **this** là gì?

*Show the general forms for **new** and **delete.** What are some advantages of using them instead of **malloc()** and **free()?***

Hãy chỉ ra những dạng thông thường của **new** và **delete.** Vài tiện ích của việc sử dụng chúng thay vì sử dụng **malloc()** và **free()** là gì?

What is a reference? What is one advantage of using a reference parameter?

Tham chiếu là gì? Một tiện ích của việc sử dụng tham chiếu là gì?

*Create a function called **recip()** that takes one **double** reference parameter. Have the function change the value of that parameter into its reciprocal. Write a program to demonstrate that it works.*

Tạo một hàm được gọi là **recip()** nó giữ một tham chiếu kiểu **double.** Hãy dùng hàm này để thay đổi giá trị của tham chiếu đó thành số nghịch đảo của nó. Viết một chương trình để chứng minh nó hoạt động.

CUMULATIVE SKILLS CHECK:

KỸ NĂNG TÍCH LŨY:

This section checks how well you have integrated material in this chapter with that from the preceding chapters.

Phần này kiểm tra cách mà bạn kết hợp tài liệu trong chương này với những chương trước.

Give a pointer to an object, what operator is used to access a member of that object?

Cho một con trỏ trỏ vào một đối tượng, toán tử nào được sử dụng để truy cập một thành phần của đối tượng đó?

*In chapter 2, a **strtype** class was created that dynamically allocated space for a string. Rework the **strtype** class (shown here for your convenience) so it uses **new** and **delete.***

Trong chương 2, một lớp **strtype** được tạo để cấp chỗ trống cho một chuỗi. Hãy xem xét lại lớp **strype** (đã được chỉ ra ở đây để tạo cho ban sự thuận lợi) vì vậy nó sủ dụng **new** và **delete.**

```
#include<cstring>

#include<iostream>

#include<cstdlib>

using namespace std;


class strtype

{

     char *p;

     int len;

     public:

          strtype (char *ptr);

          ~strtype();

          void show();

};


strtype::strtype(char*ptr)

{

     len=strlen(ptr);
```

```
        p=(char*)malloc(len+1);
        if(!p)
        {
              cout<<"Allocation error";
              exit(1);
        }
        strcpy(p,ptr);
}

strtype::~strtype()
{
        cout<<"freeing";
        free(p);
}

void strtype::show()
{
        cout <<p<<len;
        cout <<"\n";
}
int main()
{
        strtype s1("This is the first string");
        strtype s2("This is the second string");
```

```
        s1.show();
        s2.show();
        return 0;
    }
```

On your own, ...,reword so that is uses a refer

Theo bạn,…..khi nào đề cập đến

CHAPTER 5

FUNCTION OVERLOADING – Nạp chồng hàm

Chapter objectives

5.1 OVERLOADING CONSTRUCTOR FUNCTION

Nạp chồng Các H‡m Tạo.

5.2 CREATING AND USING A COPY CONSTRUCTOR

Tạo V‡ Sử Dụng H‡m Bản Sao.

5.3 THE OVERLOAD ANACHRONISM

Sự Lỗi Thời Của Từ Khóa overload.

5.4 USING DEFAULT ARGUMENTS

Sự Dụng Các Đối Số Mặc Định.

5.5 OVERLOADING AND AMBIGUIT

Nạp chồng v‡ Tính Không Chính Xác Định.

5.6 FINDING THE ADDREES OF AN OVERLOADED FUNCTION SKILLS CHECK

Tìm Địa Chỉ Của Một H‡m nạp chồng.

In this chapter you will learn more about overloading functions. Although this topic was introduced early in this book, there are several further aspects of it that need to be covered. Among the topics included are how to overload constructor funcitons, how to create a copy constructor, how to give functions default arguments, and how to avoid ambiguity when overloading.

Trong chương này bạn sẽ học về quá tải các hàm. Mặc dầu chủ đề này đã được giới thiệu trước trong cuốn sách này, nhưng vẫn còn nhiều khía cạnh cần được trình bày. Trong số các chủ đề gồm có cách quá tải các hàm tạo, cách tạo hàm tạo bản sao (conpy constructor), cách cho một hàm các đối số mặc định và cách tránh tính không xác định khi quá tải.

Review skills check (kiểm tra kĩ năng ôn)

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Trước khi bắt đầu, bạn nên trả lời đúng các câu hỏi và làm các bài tập sau đây:

What is a reference? Give two important uses.

Tham chiếu là gì? Hãy trình bày hai cách dùng quan trọng.

*Show how to allocate a **float** and an **int** by using **new**. Also, show how to free them by using **delete**.*

Trình bày cách cấp phát một **float** và một **int** bằng cách dùng **new.** Cũng vậy, trình bày cách giải phóng chúng bằng cách dùng **delete.**

*What is the general form of **new** that is used to initialize a dynamic variable? Give a concrete example.*

Trình bày dạng tổng quát của **new** dùng để khởi đầu một biến động. Cho ví dụ cụ thể.

Given the following class, show how to initialize a ten-element array so that x has the valuses 1 through 10.

Cho lớp sau đây, trình bày cách khởi đầu một mảng 10 phần tử để cho x có giá trị từ 1 đến 10.

```
class samp

{

     int x;

public:

     samp(int n) { x = n; }

     int getx() { return x; }

};
```

Give one advantage of reference parameters. Give one disadvantage.

Trình bày một ưu điểm của các tham số tham chiếu. Trình bày một nhược điểm.

Can dynamically allocated arrays be initialized?

Có thể khởi đầu một mảng được cấp phát động không?

*Create a function called **mag()** using the following prototype that raises **numb** to the order of magnitude specified by **order**:*

Hãy tạo hàm có tên **mag()** bằng cách dùng nguyên mẫu sau đây để nâng **num** lên cấp có độ lớn là **order**:

```
void mag(long &num, long order);
```

*For example, if **num** is 4 and order is 2, when **mag()** returns, **num** will be 400. Demonstrate in a program that the function works.*

Ví dụ nếu **num** là 4và **order** là 2 thì khi **mag()** trả về, **num** sẽ là 400. Hãy viết một chương trình để hàm hoạt động.

5.1. OVERLOADING CONSTRUCTOR FUNCTIONS - QÚA TẢI CÁC HÀM TẠO:

It is possible-indeed, common-to overload a class's constructor function. (It is not possible to overload a destructor, however). There are three main reasons why you will want to overload a constructor function: to gain flexibility, to support arrays, and to creat copy constructors. The first two of these are discussed in this section. Copy constructors are discussed in the next section.

Thông thường có thể quá tải hàm tạo của một lớp. (Tuy nhiên, không thể quá tải hàm hủy). Có 3 lý do chính tại sao bạn cần quá tải hàm tạo: để có tính linh hoạt, để hỗ trợ mảng và để tạo các hàm bản sao. Hai lý do đầu tiên được thảo luận trong phần này. Các hàm tạo bản sao được thảo luận trong phần tiếp theo.

One thing to keep in mind as you study the examples is that there must be a constructor function for each way that an object of a class will be created. If a program attempts to create an object for which no matching constructor is found, a compile-time error occurs. This is why overloaded constructor functions are so common to C++ programs.

Một điều cần nhớ khi bạn nghiêng cứu các ví dụ là hàm tạo của lớp phải phù hợp với cách mà đối tượng của lớp đó được khai báo. Nếu không có sự phù hợp, lỗi thời gian biên dịch sẽ xảy ra. Đây là lý do tại sao các hàm tạo được quá tải là rất thông dụng trong các chương trình C++.

EXAMPLES

VÍ DỤ

*Perhaps the most frequent use of overloaded constructor functions is to provide the option of either giving an object an initialization or not giving it one. For example, in the following program, **o1** is given an initial value, but **o2** is not. If you remove the constructor that has the empty argument list, the program will not compile because there is no constructor that matches a noninitialized object of type **samp.** The reverse is also true: If you remove the parameterized constructor, the program will not compile because there is no match for an*

initialized object. Both are needed for this program to compile correctly.

1. Có lẽ cách sử dụng thường xuyên nhất đối với các hàm tạo được quá tải là hoặc khởi đầu một đối tượng hoặc không khởi đầu một đối tượng. Ví dụ, trong chương trình này, **o1** được cho một giá trị đầu còn **o2** thì không. Nếu bạn loại bỏ hàm tạo có danh sách đối số rỗng thì chương trình sẽ không biên dịch vì không có hàm tạo phù hợp với đối tượng kiểu **samp** không được khởi đầu. Điều ngược lại cũng đúng: Nếu bạn loại bỏ hàm tạo được tham số hóa thì chương trình ssẽ không biên dịch bởi vì không có sự phù hợp với đối tượng được khởi đầu, cả hai đều cần cho chương trình này để biên dich đúng.

```
#include <iostream>

using namespace std;


class myclass

{

     int x;

public:

     // overload constructor two ways

     myclass () { x = 0; } //no initializer

     myclass (int n) { x = n ;} // initializer

     int getx () { return x; }

};


int main()

{

     myclass o1(10); // declare with initial value

     myclass o2; // declare without initializer
```

```
        cout<<" o1: "<< o1.getx()<<'\n';
        cout<<" o2: "<< o2.getx()<<'\n';

        return 0;
    }
```

Another common reason constructor functions are overloaded is to allow both individual objects and arrays of objects to occur within a program. As you porbably know from your own programming experience, it is fairly common to initialize a single variable, but it is not as common to initialize an array. (Quite often array values are assigned using information known only when the program is executing). Thus, to allow noninitialized arrays of objects along with initialized objects, you must include a constructor that supports initialization and one that does not.

2. Một lý do thông dụng khác đối với các hàm tạo được quá tải là để cho các đối tượng riêng lẻ lẫn các mảng đối tượng xảy ra trong chương trình. Có lẽ từ kinh nghiệm lập trình bạn đã biết, rất thông dụng để khởi đầu một biến đơn, nhưng khonog thông dụng để khởi đầu một mảng. (Thường các giá trị mảng được gán bằng cách dùng thông tin được biết chỉ khi chương trình đang thi hành). Do đó, để cho các mảng đối tượng chưa được khởi đầu đi với các đối tượng đã được khởi đầu, bạn phải dùng đến hàm tạo để hỗ trợ sự khởi đầu và một hàm tạo không hỗ trọ sự khởi đầu.

   For instance, assuming the class **myclass** from Example 1, both of

   these declarations are valid:

   Ví dụ, giả sử với lớp **myclass** trong ví dụ 1, cả hai khai báo sau đều đúng:

```
myclass ob(10);
myclass ob[10];
```

By providing both a parameterized and a parameterless constructor, your program allows the creation of objects that are either initialized or not as needed.

Bằng cách dùng hàm tạo cho cả sự khởi đầu và sự khonog khởi đầu, các biến có thể được khởi đầu hoặc không khởi đầu khi cần. Ví dụ, chương trình này khai báo hai mảng có kiểu **myclass**, một mảng được khởi đầu và một mảng không.

*Of course, once you have defined both parameterized and parameterless constructors you can use them to create initialized and noninitialized arrays. For example, the following program declares two arrays of **myclass**; one initialized and the other is not:*

```
#include <iostream>
using namespace std;

class myclass
{
     int x;
     public:
          //overload constructor two days
          myclass() { x = 0; }  //no initializer
          myclass( int n ) { x = n; }  // initializer
          int getx() { return x; }
};

int main()
{
     myclass   o1[10];      //declare   array   without
initialized

     // declare with initializers
```

```
    myclass o2[10] = {1,2,3,4,5,6,7,8,9,10};

    int i;

    for(i=0; i<10; i++)
    {
        cout<<" o1["<< i << "]: "<< o1[i].getx();
        cout<<"\n";
        cout<<" o2["<< i << "]: "<< o2[i].getx();
        cout<<"\n";
    }
    return 0;
}
```

*In this example, all elements of **o1** are set to 0 by the constructor function. The elements of **o2** are initialized as shown in the program.*

Trong ví dụ này, mọi phần tử của **o1** được hàm tạo đặt ở zero. Các phần tử của **o2** được khởi đầu như trong chương trình.

*Another reason for overloading constructor functions is to allow the programmer to select the most convenient method of initializing an object. To see how, first examine the next example, which creates a class that holds a calendar date. It overloads the **date()** constructor two ways. One form accepts the date as a character string. In the other form, the date is passed as three integers*

Một lý do khác để quá tải các hàm tạo là cho phép người lập trình chọn phương pháp thuận lợi nhất để khởi đầu cho một đối tượng. Để thấy rõ điều này như thế nào, trước hết chúng ta hãy xét ví dụ tiếp theo tạo ra một lớp để giữ ngày tháng theo lịch. Nó quá tải hàm tạo **date()** theo hai cách. Một cách nó nhận ngày tháng như một chuỗi ký tự. Cách khác, ngày tháng được truyền như 3 số nguyên.

```
#include <cstdio> // included for sscanf()
```

```
using namespace std;


class date
{
     int day, month, year;
public:
     date( char *str);
     date( int m, int d, int y)
     {
          day = d;
          month = m;
          year = y;
     }
     void show()
     {
          cout<< month<<'/'<<day<<'/';
          cout<< year<<'\n';
     }
};


date::date( char *str)
{
     sscanf(str,     "%d%*c%d%*c%d",&month,     &day,
&year);
}
```

```
int main()
{
      // construct date object using string
      date sdate(" 12/31/99");

      // construct date object using integers
      date idate (12,31,99);

      sdate.show();
      idate.show();
      return 0;
}
```

*The advantage of overloading the **date()** constructor, as shown in this program, is that you are free to use whichever version most conveniently fits the situation in which it is being used. For example, if a **date** object is being created from user input, the string version is the easiest to use. However, if the **date** object is being constructed through some sort of internal computation, the three-integer parameter version probably makes more sense.*

Ưu điểm của việc quá tải hàm tạo **date()**, như được trình bày trong chương trình, là bạn tự do sử dụng phiên bản nào thuân lợi nhất phù hợp với tình huống mà nó được sử dụng. Ví dụ, nếu đối tượng **date** được tạo ra từ dữ liệu nhập của người sử dụng thì sử dụng phiên bản chuỗi là thuận lợi nhất. Tuy nhiên nếu đối tượng date được tạo thông qua một loại tính toán bên trong thì sử dụng phiên bản tham số 3 số nguyên có lẽ sẽ hay hơn.

*Although it is possible to overload a constructor as many times as you want, doing so excessively has a destructuring effect on the class. From a stylistic point of view, it is best to overload a constructor to accommodate only those situations that are likely to occur frequently. For example, overloading **date()** a third time so the date can be entered interms of accept an object of type **time_t** (a type that stores the system date and time) could be very valuable. (See the*

mastery Skills Check exercises at the end of this chapter for an example that does just this).

Mặc dù có thể quá tải hàm tạo nhiều lần như ban muốn, nhưng nếu thực hiện quá tải quá nhiều lần có thể tạo ra tác dụng hủy hoại trên lớp. Từ quan điểm tu từ tốt nhất là nên quá tải một hàm tạo phù hợp vớinhững tình huống thường xảy ra. Ví dụ, việc quá tải hàm **date()** lần thứ ba để cho ngày tháng được đưa vào như ba số nguyên bát phân thì ít có ý nghĩa hơn. Tuy nhiên, việc quá tải hàm để nhận một đối tượng có kiểu **time_t** (la kiểu lưu trữ ngày giờ hệ thống) có thể rất có giá trị. (Xem các bài tập trong phần kiểm tra kỹ năng lĩnh hội ở cuối chương này).

There is one other situation in which you will need to overload a class's constructor function: when a dynamic array of that class will be allocated. As you should recall from the preceding chapter, a dynamic array cannot be initialized. Thus, if the class contains a constructor that takes an initializer, you must include an overloaded version that takes no initializer. For example, here is a program that allocates an object array dynamically:

Có một trường hợp mà bạn cần quá tải một hàm tạo của lớp: khi mảng động của lớp được cấp. Bạn nhớ lại trong chương trước, một mảng động không thể được khởi đầu. Do đó, nếu lớp có hàm tạo nhận một bộ khởi đầu, bạn phải đưa vào phiên bản được quá tải không nhận bộ khởi đầu. Ví dụ, đây là chương trình cấp phát động một mảng đối tượng.

```
#include <iostream> // included for sscanf()

using namespace std;



class myclass

{

     int x;

public:

     // overload constructor two ways

     myclass() { x = 0;} //no initializer

     myclass( int n) { x = n;} //initializer

     int getx () { return x;}
```

```
     void setx(int n) {x = n; }
};

int main()
{
     myclass *p;
     myclass ob(10); // initialize single variable

     p = new myclass[10];//can't use initializers
here
     if(!p)
     {
          cout<<" Allocation error\n";
          return 1;
     }

     int i;
     // intialize all elements to ob
     for( i=0; i<10; i++)
     {
          cout<<" p[ "<<i<<"]"<<p[i].getx();
          cout<<'\n';
     }

     return 0;
}
```

*Without the overloaded version of **myclass()** that has no initializer, the **new** statement would have generated a compile-time error and the program would not have been compiled.*

Không có phiên bản được quá tải của **myclass()** và **myclass** không có bộ khởi đầu thì câu lệnh **new** sẽ sinh ra lỗi thời gian biên dịch và chương trình sẽ không được biên dịch.

EXERCISES

BÀI TẬP

1. Given this partially defined class

1 Cho lớp được xác định riêng phần như sau:

```
class strtype
{
   char*p;
   int len;
   public:
      char *getstring() { return p;}
      int getlength() { return len;}
};
```

*Add two constructor functions. Have the first one take no parameters. Have this one allocate 255 bytes of memory (using **new**), initialize that memory as a null string, and give **len** a value of 255. Have the other constructor take two parameters. The first is the string to use for initialization and the other is the number of bytes to allocate. Have this version allocate the specifiex amount of memory and copy the string to that memory. Perform all necessary boundary checks and demonstrate that your constructors work by including a short program.*

Hãy bổ sung hai hàm tạo. Hãy cho hàm thứ nhất không nhân tham số. Hãy cho hàm này cấp phát 255 byte bộ nhớ (bằng cách dùng **newb**), hãy khởi đầu bộ nhớ đó như một chuỗi rỗng (null) và cho **len** giá trị 255. Hãy cho hàm tạo khác nhận hai tham số. Tham số thứ nhất là chuỗi dùng để khởi đầu và tham số kia là số byte để cấp phát. Cho phiên bản này cấp phát lượng bộ nhớ đã được chỉ rõ và chép chuỗi vào bộ nhớ. Thực hiện việc kiểm tra giới hạn biên cần thiết và chứng tỏ hàm tạo của bạn hoạt động bằng cách đưa nó vào trong một chương trình ngắn.

*In Exercise 2 of Chapter 2, Section 2.1, you created a stopwatch emulation. Expand your solution so that the **stopwatch** class provides both a paramenterless constructor (as it does already) and an overloaded version that accepts the system time in the form returned by the standard function **clock()**. Demonstrate that your improvement works.*

. Trong bài tập 2, chương 2, phần 1, bạn đã tạo ra sự thi đua của đồng hồ bấm giờ. Hãy mở rộng giải đáp của bạn để cho lớp **stopwatch** bao gồm một hàm tạo không tham số và một phiên bản được quá tải để nhận giờ hệ thông dưới dạng được trả về bởi hàm chuẩn **clock()**. Chứng tỏ rằng chương trình cần cải tiến của bạn hoạt động

On your own, think about ways in which an overloaded constructor function can be beneficial to your own programming tasks.

Hãy tìm cách trong đó một hàm tạo được quá tải có thể có lợi ích cho các công việc lập trình của bạn.

5.2. CREATING AND USING A COPY CONSTRUCTOR - TẠO VÀ SỬ DỤNG HÀM TẠO BẢN SAO:

One of the more important forms of an overloaded constructor is the copy constructor. As numerous examples from the preceding chapters have shown, problems can occur when an object is passed to or returned from a function. As you will learn in this section, one way to avoid these problems is to define a copy constructor.

Một trong những dạng quan trọng hơn của hàm tạo quá tải là hàm tạo bản sao (copy constructor). Đã có nhiều ví dụ được trình bày trong những chương trước, vấn đề xảy ra khi có một đối tượng được truyền tới hay được trả về từ một hàm. Như bạn sẽ biết trong phần này, cách để tránh những vấn đề này là định nghĩa một

hàm tạo bản sao, đó là kiểu đặc biệt của hàm tạo được quá tải.

To begin, let's restate the problem that a copy constructor is designed to solve. When an object is passed to a function, a bitwise (i.e.,exact) copy of that object is made and given to the function parameter that receives the object. However, there are cases in which this identical copy is not desirable. For example, if the object contains a pointer to allocated memory, the copy will point to the same memory as does the original object. Therefore, if the copy makes a change to the contents of this memory, it will be changed for the original object too! Also, when the function terminates, the copy will be destroyed, causing its destructor to be called. This minght lead to undesired side effects that further affect the original object.

Để bắt đầu chúng ta hãy xét lại vấn đề mà một hàm tạo bản sao được thiết kế để giải quyết. Khi một đối tượng được truyền cho một hàm, một bản sao từng bit của đối tượng đó được tạo ra và được truyền cho tham số của hàm để nhận đối tượng. Tuy nhiên, có những trường hợp trong đó bản sao đồng nhất là không như mong muốn. Ví dụ, nếu đối tượng có con trỏ tới bộ nhớ được cấp phát, thì bản sao sẽ trỏ tới *cùng* bộ nhớ như đối tượng gốc đã làm. Do đó, nếu bản sao tạo ra sự thay đổi cho nội dung bộ nhớ thì nó cũng sẽ được thay đổi đối với đối tượng gốc. Cũng vậy, khi một hàm kết thúc, bản sao sẽ bị hủy và hàm hủy của nó được gọi. Điều này dẫn đến những tác dụng không mong muốn làm ảnh hưởng đến đối tượng gốc.

A similar situation occurs when an object is returned by a function. The compiler will commonly generate a temporary object that holds a copy of the value returned by the function. (This is done automatically and is beyond you control). This temporary object goes out of scope temporary object's destructor to be called. However, if the destructor frees dynamically allocated memory), trouble will follow.

Tình trạng tương tự này cũng xảy ra khi một đối tượng được trả về từ một hàm. Thông thường, trình biên dịch sẽ tạo ra một đối tượng tạm để giữ bản sao của giá trị do hàm trả về. (Việc này được thực hiện một cách tự động và bạn không điều khiển được). Đối tượng tạm này sẽ ra khỏi phạm vi một khi giá trị được trả về cho thủ tục gọi, khiến hàm hủy của đối tượng tạm được gọi. Tuy nhiên, nếu hàm hủy hủy bỏ thứ gì đó cần cho thủ tục gọi (ví dụ, nó giải phóng bộ nhớ cấp phát động), thì rắc rối sẽ xảy ra.

At the core of these problems is the fact that a bitwise copy of the object is being made. To prevent these problems, you, the programmer, need to define

precisely what occurs when a copy of an object is made so that you can avoid undesired side effects. The way you accomplish this is by creating a copy constructor. By defining a copy constructor, you can fully specify exactly what occurs when a copy of an object is made.

Cốt lõi của vấn đề này là ở chỗ bản sao từng bit cảu đối tượng được thực hiện. Để ngăn chặn những vấn đề này, là người lập trình, bạn cần xác định chính xác những gì xảy ra khi bản sao của một đối tượng được thực hiện để cho bạn tránh được những tác dụng không mong muốn. Thực hiện điều này bằng cách tạo ra một hàm tạo bản sao. Bằng cách định nghĩa hàm tạo bản sao, bạn có thể hoàn toàn chỉ rõ chính xác những gì xảy ra khi bản sao của một đối tượng được thực hiện.

It is important for you to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first situation is assignment. The second situation is initialization, which can occur three ways:

Điều quan trọng là bạn phải hiểu rằng C++ xác định hai trường hợp phân biệt trong đó giá trị của một đối tượng được truyền cho đối tượng khác. Trường hợp thứ nhất là phép gán. Trường hợp thứ hai là sự khởi đầu có thể xảy ra theo 3 cách:

When an object is used to initialize another in a declaration statement.

When an object is passed as a parameter to a function and

When a temporary object is created for use as a return value by a function.

Khi một đối tượng được dùng để khởi đầu một đối tượng khác trong câu lệnh khai báo.

Khi một đối tượng được truyền như tham số cho hàm và

Khi một đối tượng được tạo ra dùng để làm giá trị trả về bởi một hàm.

The copy constructor only applies to initializations. It does not apply to assignments.

Hàm tạo chỉ áp dụng cho sự khởi đầu. Nó không áp dụng cho phép gán.

By default, when an initialization occurs, the compiler will automatically provide a bitwise copy. (That is, C++ automatically provides a default copy constructor that simply duplicates the object). However, it is possible to specify precisely how one object will initialize another by defining a copy constructor. Once defined, the copy constructor is called whenever an object is used to initialize another.

Theo mặc định, khi một sự khởi đầu xảy ra, trình biên dich sẽ tự động cung cấp bản sao từng bit. (Nghĩa là, C++ tự động cung cấp một hàm tạo bản sao mặc định). Tuy nhiên, có thể chỉ rõ chính xác cách mà một đối tượng sẽ khởi đầu một đối tượng khác bằng cách định nghĩa hàm tạo bản sao. Khi đã được định nghĩa, hàm tạo bản sao được gọi bất cứ lúc nào mà một đối tượng được dùng để khởi đầu một đối tượng khác.

Remember: *Copy constructor do not affect assignment operations*

The most common form of copy constructor is shown here:

Cần nhớ: *các h‡m tạo bản sao không có ảnh hưởng đến các phép gán*

Mọi hàm tạo bản sao có dạng tổng quát sau:

```
classname(const classname&obj)
  {
    //body of constructor
  }
```

*Here obj is a reference to an object that is being used to initialize another object. For example, assuming a class called **myclass**, and that y is an object of type **myclass**, the following statements would invoke the **myclass** copy constructor:*

Ở đây, *obj* là một tham chiếu tới một đối tượng được dùng để khởi đầu một đối

tượng khác. Ví dụ, giả sử lớp được gọi là **myclass**, và y là đối tượng cảu **myclass** thì các câu lệnh sau đây sẽ dùng đến hàm tạo bản sao của **myclass.**

```
myclass x = y; // y explicitly initializing x
func1(y);      //  passed as a parameter
y = func2();   // receiving a returned object
```

*In the first two case, a reference to y would be passed to the copy constructor.In the third, a reference to the object returned by **func2()** is passed to the copy constructor.*

Trong hai trường hợp đầu, một tham chiếu tới y sẽ được truyền cho hàm tạo bản sao. Trường hợp thứ ba, một tham chiếu tới đối tượng được trả về bởi **func2()** sẽ được truyền cho hàm tạo bản sao.

EXAMPLES

CÁC VÍ DỤ

*Here is an example that illustrates why an explicit copy constructor function is needed. This program creates a very limited "sage" integer array type thay prevents array boundaries from being overrun. Storage for each array is allocated using **new**, and a pointer to the memory is maintained withing each array object.*

Đây là ví dụ minh họa tại sao cần đến một hàm tạo bản sao cụ thể. Chương trình này tạo ra một mảng nguyên "an toàn" c giới hạn để ngăn chặn sự tràn qua giới hạn biên của mảng. Dùng **new** để cấp phát bộ nhớ lưu trữ mỗi mảng, và một con trỏ tới bộ nhớ được duy trì trong mỗi đối tượng mảng.

```
/* This program creates a "safe" array class, since
space for the array is dynamically allocated, a copy
constructor is provided to allocate memory when one
```

```
array object is used to initialize another.
*/

#include <stdafx.h>
#include <iostream>
#include <cstdlib>
using namespace std;

class array
{
     int *p;
     int size;
public:
     array( int sz)
     {
          //constructor
          p = new int[sz];
          if(!p) exit(1);
          size = sz;
          cout<<" Using 'normal' constructor\n";
     }
     array () {delete [] p;}

     // copy constructor
     array(const array &a);
```

```
     void put(int i, int j)
     {
          if(i>=0&& i<size)
               p[i]=j;
     }
     int get (int i)
     {
          return p[i];
     }
};

/* Copy constructor
In the following, memory is allocated specifically for the copy, and the address of this memory is assigned to p, therefore, p is not pointing to th same dynamically allocated memory as the original object
*/

array::array (const array &a)
{
     int i;

     size = a.size;
     p = new int [ a, size]; // allocate memory for copy
     if(!p) exit(1);
     for(i=0;i<a.size;i++)  p[i]  =  a.p[i];  //  copy
```

```
contents
     cout<<" Using copy constructor\n";
}

int main()
{
     array num(10); // this calls "normal" constructor
     int i;

     //put some values into the array
     for(i=0;i<10;i++)
          num.put(i,i);
     // display num
     for(i=9;i>=0;i--)
          cout<<num.get(i);
     cout<<"\n";

     // create another array and initialize with num
     array x = num; // this invokes copy constructor

     // display x
     for(i=0;i<10;i++) cout<<x.get(i);

     return 0;
}
```

*When **num** is used to initialize x, the copy constructor is called, memory for the new array is allocated and stored in **x.p**, and the contents of **num** are copied to x's array. In this way, **x** and **num** have arrays that have the same values, but each array is separate and distinct. (That is, **num.p** and **x.p** do no point to the same piece of memory). If the copy constructor had not been created, the bitwise initialization **array x = num** would have resulted in **x** and **num** sharing the same memory for their arrays! (That is, **num.p** and **x.p** would have, indeed, pointed to the same location).*

Khi **num** được dùng để khởi đầu x, hàm tạo bản sao được gọi, bộ nhớ được cấp phát cho mảng mới và được lưu trữ trong **x.p**, và nội dung cảu **num** được sao chép vào mảng của x. Trong cách này, x và **num** có các mảng có cùgn những giá trị, nhưng mỗi mảng là hoàn toàn phân biêt (**num.p** và **x.p** không trở về cùng một phần bộ nhớ). Nếu hàm tạo không được tạo ra thì sự khởi động từng bit **array=num** sẽ dẫn đến kết quả là **x** và **num** cùng sử dụng chung bộ nhớ giống nhau đối với các mảng! (Nghĩa là **num.p** và **x.p** cũng trở về một vị trí).

The copy constructor is called only for initializations. For example, the following sequence does not call the copy constructor defined in the preceding program:

Hàm tạo bản sao được gọi chỉ đối với sự khởi đầu. Ví dụ, những câu lệnh sau đây không gọi hàm tạo được định nghĩa trong chương trình trước:

```
array a(10);

array b(10);

b=a; // does not call copy constructor
```

*In this case, **b=a** performs the assignment operation.*

Trong trường hợp này, **b=a** thực hiện phép gán.

To see how the copy constructor helps prevent some of the poblems associated with passing certain types of objects to functions, consider this (incorrect) program:

Để thấy rõ cách hàm tạo bản sao giúp ngăn ngừa một số vấn đề liên quan với việc

truyền các kiểu đối tượng nào đó cho hàm, hãy xét chương trình (sai) sau đây:

```
// This program has an error
#include <stdafx.h>
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype
{
   char *p;
public:
   strtype(char *s);
   ~strtype() {delete [] p;}
   char *get() {return p;}
};

strtype::strtype( char *s)
{
   int l;

   l = strlen(s)+1;

   p = new char[1];
   if(!p)
```

```
    {
         cout<<" Allocation error\n";
         exit(1);
    }

    strcpy(p,s);
}

void show(strtype x)
{
    char *s;

    s = x.get();
    cout<<s<<"\n";
}

int main()
{
    strtype a("Hello"), b("There");

    show(a);
    show(b);

    return 0;
}
```

*In this program, when a **strtype** object is passed to **show(),** a bitwise copy is made (since no copy constructor has been defined) and put into parameter **x**. Thus, when the function returns **x**, goes out of scope and is destroyed. This, of course, causes **x**'s destructor to be called, which frees **x.p**. However, the memory being freed is the same memory that is still being used by the object used to call the function. This results in an error.*

Trong chương trình này khi đối tượng **strtype** được truyền cho **show()**, một bản sao từng bit được thực hiện (do không có hàm tạo bản sao được định nghĩa) và được đưa vào tham số **x**. Do đó, khi hàm trả về, **x** ra khỏi phạm vi và bị huỷ. Dĩ nhiên, điều này khiến cho hàm hủy của x được gọi để giải phóng **x.p**. Tuy nhiên, bộ nhớ được giải phóng cũng là bộ nhớ đang được sử dụng bởi đối tượng dùng để gọi hàm. Điều này gây ra lỗi.

*The solution to the preceding problem is to define a copy constructor for the **strtype** class that allocates memory for the copy when the copy is created. This approach is used by the following, corrected, program:*

Giải pháp để giải quyết vấn đề này là định nghĩa một hàm tạo bản sao cho lớp **strtype** để cấp phát bộ nhớ cho bản sao khi bản sao được tạo ra. Phương pháp này được dùng trong chương trình đã được hiệu chỉnh sau đây:

```
/* This program uses a copy constructor to allow
strtype objects to be passed to functions
*/
#include <stdafx.h>
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype
```

```
{
     char *p;
public:
     strtype( char*s); //constructor
     strtype(const strtype &c); //copy constructor
     ~strtype() { delete [] p;} // destructor
     char *get() {return p;}
};

// "Normal" constructor
strtype::strtype(char *s)
{
     int l;

     l = strlen(s)+1;

     p = new char [1];
     if(!p)
     {
          cout<<" Allocation error\n";
          exit(1);
     }

     strcpy(p,s);
}
```

```
// copy constructor
strtype::strtype(const strtype &o)
{
     int l;

     l = strlen(o.p)+1;

     p = new char [1]; // allocate memory for new copy
     if(!p)
     {
          cout<<" Allocation error\n";
          exit(1);
     }

     strcpy(p,o.p); // copy string into copy
}

void show(strtype x)
{
     char *s;

     s = x.get();
     cout<<s<< "\n";
}
```

```
int main()
{
      strtype a("Hello"), b("There");

      show(a);
      show(b);

      return 0;
}
```

*Now when **show()** terminates and **x** goes out of scope, the memory pointed to by **x.p**(which will be freed) is not the same as the memory still in use by the object passed to the function.*

Bây giờ, khi **show()** kết thúc và **x** ra khỏi phạm vi, bộ nhớ được trỏ tới bởi **x.p** (sẽ được giải phóng) không giống như bộ nhớ đang được sử dụng bởi đối tượng được truyền cho hàm.

EXERCISES

BÀI TẬP

The copy constructor is also invoked when a function generates the temporary object that is used as the function's return value (for those functions that return objects). With this in mind, consider the following output:

Hàm tạo bản sao cũng được dùng đến khi một hàm sinh ra đối tượng tạm thời được dùng như giá trị trả về của hàm (đối với những hàm trả về đối tượng). Với ý nghĩa đó, xét dữ liệu xuất sau:

```
Constructing normally

Constructing normally
```

```
     Constructing copy
```

This output was created by the following program. Explain why, and describe precisely what is occuring.

Dữ liệu xuất này được tạo ra bởi chương trình sau đây. Hãy giải thích tại sao và mô tả chính xác điều gì xảy ra.

```
#include <stdafx.h>
#include <iostream>
using namespace std;

class myclass
{
public:
     myclass();
     myclass (const myclass &o);
     myclass f();
};

// Normal constructor
myclass::myclass()
{
     cout<<" Constructing normally\n";
}

// Copy constructor
```

```
myclass::myclass( const myclass &o)
{
     cout<<" Constructing copy\n";
}

// Return an object.
myclass myclass::f()
{
     myclass temp;

     return temp;
}

int main()
{
     myclass obj;
     obj = obj.f();
     return 0;
}
```

Explain what is wrong with the following program and then fix it.

Hãy giải thích điều gì sai trong chương trình sau và sau đó sửa chương trình lại cho đúng.

```
// This program contains an error.
#include <stdafx.h>
```

```
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass
{
     int *p;
public:
     myclass (int i);
     ~myclass () {delete p;}
     friend int getval (myclass o);
};

myclass::myclass(int i)
{
     p = new int;

     if(!p)
     {
          cout<<" Allocation error\n";
          exit(1);
     }
     *p = i;
}
```

```
int getval(myclass o)
{
     return *o.p; //get value
}

int main()
{
     myclass a(1), b(2);

     cout<<getval(a)<<" "<<getval(b);
     cout<<"\n";
     cout<<getval(a)<<" "<<getval(b);

     return 0;
}
```

In your words, explain the purpose of a copy constructor and how it differs from a normal constructor

Theo cách của bạn, hãy giải thích mục đích của hàm tạo bản sao và nó khác với hàm tạo thông thường thế nào.

5.3. THE OVERLOAD ANACHRONISM - Sự Lỗi Thời Của Tứ khóa Overload:

*When C++ was first invented, the keywork **overload** was required to create an **overloaded** function. Although overload is now obsolete and no longer supported*

by modern C++ compliers, youmay still see overload used in old program, so it is a good idea to understand how it was applied.

Khi C++ mới được phát minh, từ khóa **overload** dùng để tọa một hàm quá tải. Măc dầu **overload** không cần thiết nữa, nhưng để duy trì tính tương thích của C++ và cú pháp của sự quá tải theo kiểu cũ vẫn còn trình biên dịch C++ chấp nhận. Khi bạn tránh dùng từ khóa này,bạn vẫn thấy **overload** được dùng trong chương trình hiện nay, để hiểu nó được ứng dụng như thế nào cũng là một ý kiến hay.

*The general grom of **overload** is shown here.*

Dạng tổng quát của overload như sau:

```
overload func-name;
```

*Where func-name is the name of the function to be overloaded. This statement must precede the overloaded function declarations. For example, this tells the compiler that you will be overloading a function called **timer():***

Ở đó func-name là tên hàm được quá tải. Câu lệnh này phải khai báo trước các hàm được quá tải. Ví dụ, câu lệnh sau báo cho chương trình biên dịch biết là bạn quá tải một hàm có tên **timer()**

overload timer;

***Remmber**: **overload** is obsolete and no longer supported by modern C++ compilers.*

Cần Nhớ: *vì* **overload** *đã trở th‡nh lỗi thời trong các chương trình C++ hiện nay nên người ta thường tránh dùng nó.*

5.4. USING DEFAULT ARGUMENTS - Sử dụng các đối số mặc định:

There is feature of the C++ that is related to function overloading. This featureis called the defaukt argument, and it allows you to give a parameter a default value when no corresponding argurrment is specified when the function is called. As you will see, using default arguments is essentially a shorthand from

of function overloading.

Có một đặc điểm của C++ liên quan đến sự quá tải hàm. Đặc điểm này này gọi là đối số mặc định và cho phép bạn gán một giá trị măc định cho hàm số khi không có đối số tương ứng được chỉ rõ khi hàm được gọi. Như bạn sẽ thấy, sử dụng các đối số mặc định chủ yếu là dạng ngắn của sự quá tải hàm.

To give a parameter a default arguments, simply follow that parameter with an equal sign and the value you want it to defualt to if no corresponding argument is present when the function is called. For example, this function gives its two paramentes default values of 0:

Để gắn một đối số mặc định cho một tham số thì sau tham số có dấu bằng và một giá trị mà bạn muốn mặc định nếu không có đối tương ứng khi hàm được gọi. Ví dụ, hàm sau đây cho hai giá trị măc định 0 đối với các tham số.

```
void f(int a=0, int b=0)
```

Notice that this syntax is smilar to variable initializantion. This function can now be called there difference ways. First, it can be called with both arguments specifief. Second, it can be called with only the first argument specified. In this case, ***b*** *will default to 0. Finally ,* ***f()*** *can be called with no arguments, causing both a and b to default to 0. That is, the following invoctions of f() are all valid:*

Chú ý rằng cú pháp này tượng tự như khởi đầu một biến.Bây giời hàm có thể giọ theo 3 cách khac nhau. Thứ nhất, nó được gọi với hai đối số được chỉ rõ. Thứ hai, nó được gọi với đối số thứ nhất được chỉ rõ. Trong trường hợp này, **b** sẽ mặc định về zero. Nghĩa là các câu lệnh sau đều đúng:

```
f();            // a and b default to 0
f(10);           // a is 10, b defaults to 0
f(10,99);       // a is 10, b defaults to 99
```

In this example, it should be clear that there is no way to default ***a*** *and specify* ***b***

Trong ví dụ này, rõ ràng không là không có cách mặc định **a** và chỉ rõ **b**

When you create a function that has one or more default arguments, those arguments must be specfied noly once: either in the function's prototype or in its definition if the definition precedes the function's first use. The defaults cannot be specified in both the prototype and the definition. This rule applies even if you simply duplicate the same defaults.

Khi bạn tạo hàm có một hay nhiều đối số măc định thì những đối số đó được chỉ rõ một lần : hoặc trong định nghĩa hàm hoặc trong nguyên mẫu của nó chứ không được cả hai. Qui tắc này được dùng nay trong cả khi bạn nhân đôi các ngầm định giống nhau.

As you can probably guess, all default parameters must be to the right of any parameters that don't have defaults. Further, once you begin to define default parameters, you cannot specify any parameters that have the same defaults.

Có lẽ bạn cũng đoán được,tất cả các tham số mặc định phải ở bên trong thân hàm số không có mặc định. Hơn nữa, một khi bạn bắt đầu định nghĩa các tham số măc định, bạn không thể chỉ định các tham số măc định các tham số không có mẫ định.

One other point about default arguments: they must be constants or global variables. They cannot be local variables or other paramenters.

Một điểm khác về đối số mặc định: chúng phải là các hằng hoặc lá các biến toàn cục. Chúng không thể là các biến địa phương hoặc các tham số khác.

Examples

Các Ví Dụ

1. *Here is a program that illustrates the example described in the preceding discussion:*

Đây là chương trình được minh họa ví dụ mô tả trong phân thảo luận trước:

```
// A simple firdt example of default arguments.

#include "stdafx.h"
```

```
#include <iostream>
using namespace std;

void f(int a=0, int b=0)
{
     cout<< "a: " << a << ", b:" << b;
     cout<< '\n';
}

int main()
{
     f();
     f(10);
     f(10,99);
     return 0;
}
```

As you should expect, this program display the following output:

Như bạn chờ đợt, chương trình sẽ hiện thị kết quả như sau:

```
a: 0, b: 0
a: 10, b: 0
a: 10, b: 99
```

Remember that once the first default argument is specfied, all following paraments must have defaults as well. For example, this slightly different version of f() causes a compile-time error:

Nhớ rằng khi đối số măc định đầu tiên được chỉ rõ, tất cả tham số theo sau cũng phải mặc định. Ví dụ, phiên bản nơi khác của f() sau đây tạo ra lỗi thời gian biên

dịch:

```
void f(int a=0, int b)   //wrong! B must have default,
too
{
     cout<< "a: " << a << ", b:" << b;
     cout<< '\n';
}
```

2. *To understand how default arguments are ralated to function overloading, first consider the next program, which overloads the function called rect_area(). This function returns the area of a rectangle.*

Để hiểu các đối số măc định liên quan với sự quá tair hàm, trước hết hãy xét chương trình tiếp đây. Chương trình này quá tải hàm **rect_area()**. Hàm này trả về diện tích hình chữ nhật.

```
//computer area of a rectangle using overloading
functions.
#include "stdafx.h"
#include <iostream>
using namespace std;

//return area of a non-square rectangle
double rect_area (double length, double width)
{
     return length * width;
}

//return area of a square
double rect_area (double length)
```

```
{
     return length * length;
}

int main()
{
     cout<< "10 x 5.8 rectangle has area ";
     cout<< rect_area(10.0, 5.8) << '\n';

     cout<< "10 x 10 square has area : ";
     cout<< rect_area(10.0) << '\n';
     return 0;
}
```

*In this program, **rect_area()** is overloaded two ways. In the first way, both dimensions of a rectangle are passed to the function. This version is used when the rectangleis not a square. However, when the rectangle is a aquare, only one **rect_area()** is called*

Trong chương trình này, **rect_area()** được quá tải theo hai cách trong cách thứ nhất, của hai chiều của hình chữ nhật được truyền cho hàm. Phiên bản này được sử dụng khi hình chữ nhật không phải là hình vuông. Tuy nhiên, khi hình chữ nhật là hình vuông, chỉ cần một đối số được chỉ rõ và phiên bản thứ hai cua **rect_area()** được gọi.

*If you think about it, it is clear than in this situation there is really no need to have two difference function. Instead, the second paramenter can be defaulted to some value that acts as a flag to **rect_area()**. When this value is seen by the function, is uses the **length** paramenter twice. Here is an example of this approach :*

Rõ ràng trong trường hợp này thực sự không cần có hai hàm khác nhau. Thay vào đó, tham số thứ hai có thể được mặc định về giá trị nào đó tác động như còi hiệu cho **box_area()**. Khi hàm thấy giá trị này, nó dùng tham số l**ength** hai lần. Đây là ví

dụ tiếp cận này.

```
//computer area of a rectangle using defualt functions.
#include "stdafx.h"
#include <iostream>
using namespace std;

//return area of a  rectangle
double rect_area (double length, double width = 0)
{
     if(!width)
          width = length ;
     return length * width;
}

int main()
{
     cout<< "10 x 5.8 rectangle has area ";
     cout<< rect_area(10.0, 5.8) << '\n';

     cout<< "10 x 10 square has area : ";
     cout<< rect_area(10.0) << '\n';
     return 0;
}
```

*Here 0 is the default of **width**.This value was chosen because no rectangle will have a width of 0. (Actually, a rectangle width of 0 is a line.) Thus, if default value is senn, **rect_area()** automatically uses the value in length for the value of*

width.

Ở đây, zore là giá trị măc định của with. Giá trị này được chọn vì không có hình hộp nào có cạnh la zero cả. (Thật sự hình chữ nhật có độ rộng zero là đường thẳng). Do đó, nếu nhìn thấy giá trị măc định này **box-area()** tự động dùng giá trị trong **length** cho giá trị của **with**

As this example shows, default agruments often provide a simple alternative to function overloading. (Of cuorse, there are many situations in which function overloading is still required.)

Như ví dụ này chỉ rõ, các đối số ngầm định thường dưa ra cách đơn giản để quá tải hàm. (Dĩ nhiên, có nhiều trường hợp trong đó cũn cần đến sự quá tải hàm)

3. *It is not only legal to give constructor functions default arguments, it is also common. As you saw earlier in this chapter, many times a constructor is overloaded simply to allow both intialized and uninitialized object to be create. In many cases ,you can avoid overloading a constructor by giving it one or more default arguments. For example, examine this program :*

Việc cho các hàm tạo các đối số măc định là đúng đắn và thông dụng. Như bạn đã thấy trước đây chương trình này, một hàm chỉ tạo được quá tải chỉ cho phép tạo ra các đố tượng được khởi đầu vẫn không được khởi đầu. Trong nhiều trường hợp, bạn có thể tránh quá tải của một hàm tạo bằng cách cho nó một hay nhiều hàm định. Ví dụ, xét chương trình sau:

```
#include "stdafx.h"

#include <iostream>

using namespace std;


class myclass

{

     int x;

     public:

          /* Use default argument instead of overloading
```

```
myclass' constructor */
          myclass (int n = 0) { x = n; }
          int getx() { return x; }
};
int main()
{
     myclass o1(10);     //declare with initial value
     myclass o2;         //declare without initializer
     cout<< "o1: " << o1.getx() << '\n';
     cout<< "o2: " << o2.getx() << '\n';
     return 0;
}
```

*As this example shows, by giving **n** the default value of 0, it is possible to create objects that have explicit initial values and those for which the default value is sufficient.*

Như ví dụ này chỉ rõ, bằng cách cho n giá trị măc định zero, có thể tạo ra các măc đối tượng có các giá trị đầu rõ ràng và các đối tượng có giá trị mặc định là đủ.

*Another good application for a default argument is found when a parameter is used to select an option. It is possible to give that parameter a default value that is used as a flag that tells the function to continue to use the previously selected option. For example, in the following program, the function **printf()** display a string on the screen. If its how parameter is set to ignore, the text is display as is. If how is upper, the text is diaplayed in uppercase. If **how** is l**ower**, the text is displayed in **lower** case. When **how** is not specified, it defaults to -1, which tells the function to rease the last how value.*

Một cách áp dụng khác đối với một đối mặc định được tìm thấy khi một hàm số được dùng để chọn một tùy chọn. Có thể cho hàm một giá trị mặc định được dùng như một cờ hiệu dùng để báo cho hàm tiếp tục sữ dụng tùy chọn được cho trước. Ví dụ, trong chương trình dưới đây, hàm **printf** hiển thị một chuỗi lên màn hình.Nếu tham số **how** được đặt cho ignore, văn bản được hiển thị như chính

nó.Nếu how là **upper**, văn bản đươch hiện thị dưới chữ hoa. Nếu **how** là **lower** thì văn bản được hiển thị dưới đây là chữ thường. Khi không chỉ rõ hơ, giá trị mặc định của nó là -1, báo cho hàm dùng lại giá trị **how** sau cùng

```
#include "stdafx.h"
#include <iostream>
#include <cctype>
using namespace std;

const int ignore = 0;
const int upper = 1;
const int lower = 2;

void prinft( char *s, int how = -1);
int main()
{
     printf("Hello There \n",ignore );
     printf("Hello There \n",upper);
     printf("Hello There \n"); //continue in upper
     printf("Hello there \n",lower);
     printf("That's all \n"); //continua in lower
     return 0;
}

/* printf a string in the specified case.
 Use last case specified if none is given.*/
void printf( char * s, int how)
```

```
{
     static int oldcase = ignore;
     //reuse old case if none specified
     if(how < 0)
          how = oldcase;
     while(*s)
     {
          switch(how)
          {
               case upper: cout<< (char) toupper (*s);
                    break;
               case lower : cout<< (char) tolower(*s);
                    break;
               default : cout<< *s;
          }
          s--;
     }
     oldcase = how;
}
```

This function display the following output:

Hàm này hiển thị kết quả

```
Hello There
HELLO THERE
HELLO THERE
hello there
```

```
that's all
```

Earlier in this charpter you saw the general from of a copy constructor. This general from was shown with only one parameter. However, it is possible to create copy constructors that take additional values. For example, the following is also an acceptable from a copy constructor:

Phần đầu trong chương này bạn đã thấy dạng tổng quát của hàm tạo bản sao. Dạng tổng quát này được trình bày chỉ với một tham số. Tuy nhiên có thể tạo các hàm bản sao có các đối số thêm vào miễn là các đối số thêm vào có các giá trị măc định. Ví dụ, dạng sau đây cũng chấp nhập là dạng của hàm tạo của hàm tạo bản sao:

```
myclass(  const mayclass *obj, int x = 0

{

     //body of constructor

}
```

As long as the first argument is a raference to the object being copied, and all other arguments default, the function qualifies as a copy constructors.This flexibiliry you to create copy also contructors that have other uses.

Chỉ cần đối số thứ nhất là tham chiếu tới đối tượng được sao chép, và các đối tượng khác được mặc định, thì hàm sẽ là hàm tạo bản sao. Tính linh hoạt này cho phép bạn bạn tạo tra các hàm bản sao có những công cụ khác.

6. *Although default arguments are powerful and convenient, they can be misused. There is no question that, when used correctly, default arguments allow a function to perfrom its job in an efficient and easy-to-use mamner. However, this is only the case when the default default value given to a parameter makes sense. For example, if the argument is the value wanted nine times out ten, giving a function a default argument to this effect is obviously a good idea. However, in cases in which no one value is more likely to be used than anotherm, or when there is no one benefit to using a default value. Actually, providing a default argument when one is not called for destructures your program and tends to mislead anyone else who has to use that function.*

Mặc dầu các đối số măc định là mạnh và thuận lợi, chúng có thể không dùng đúng cách. Không có vấn đề mà khi được dùng đúng, các đối số măc định cho phép thưc hiện công việc một cách dễ dàng và có hiệu quả. Tuy nhiên, đây chỉ là

trường hợp khi giá trị măc định (được cho vào tham số) có ý nghĩa. Ví dụ. nếu đối số là giá trị cần có 9 lần ngoài thì việc cho hàm một đối số măc định hiểu nhiên là một ý tưởng tốt. Tuy nhiên, trong những trường hợp khi không có ích lợi gì để dùng đối số măc định như một giá trị cờ hiệu thì sữ dụng giá trị mặc định không có nghĩa hơn. Thực sự, việc cung cấp đối số mặc định ít có nghĩa hơn. Thực sự, việc cung cấp đối số mặc định khi không được gọi sẽ làm hỏng chương trình của bạn và sẽ hướng dẫn sai cho người sữ dụng hàm đó.

As with function overloading, part of becoming an excellent C++ programmer is knowing when to use a default argument and when not to.

Với sự quá tải hàm, một người lập trình C++ thành tháo sẽ biết khi nào sữ dụng đối số măc định và khi nào không.

EXERCISES:

Các Ví Dụ:

1. *In the C++ standard library is the function strtol(), which has this prototype:*

Trong thư viện chuẩn C++ có hàm **strtol(),** nguyên dạng sau:

```
long strtol(const char *start, const  **end, int base );
```

The function converts the numeric string pointed to by start into a long integer. The munber base of the numeric string is specified by base. Upon return, end pionts ti the character in the string immediately following the end of the number. The long integer equivalent of the numeric string is returned base must be in the range 2 to 38. However, most commonly, base 10 is used

Hàm chuyển đổi chuỗi số được con trỏ tới **strat** thành số nguyên dài. Cơ sở của chuỗi số được chỉ rõ bởi **base**/ Khi tra về, end trỏ tới ký tự trong chuỗi theo sau số cuối. Tương dương số nguyên dài của chuỗi trả về **base** phải ở trong khoảng từ 2 đến 38. Tuy nhiên, rất thông thường cơ số 10 được dùng.

Create a function called myclass() that works the same as strtol() except that base the default argument of 10.(Feel free to use strtol() to actually perfrom the conversion. It requires the hearder < cstdlib>.) Demonstrate that your version

Hãy tạo hàm **mystrtol()** hoạt động giống như **strtol()** với ngoại lệ là base được

cho một đối số mặc định la 10. (Bạn có thể dùng hàm **strtol()** để thực hiện sử chuyển đổi. Cần phải có file **stdlib.h**)Hãy chứng tỏ phiên bản của bạn hoạt động đúng.

What is wrong with the following function prototype ?

Có gì sai trong nguyên mẫu hàm sau:

```
char *f( char *p, int x = 0, char *q)
```

Most C++ compiles supply nonstandard function that allow cursor positioning and the like. If you compiler supplies such function, create s function called myclreol() that clears the line from the current cursor position to the end of the line. However, give ths this function a parameter that specifies the number of character positions to clear. If the parameter is not specified, automatically clear the entire line. Otherwise, clear only the number of character positions specified by the parameter.

Hầu hết các trinh biên dịch C++ cung cấp các hàm không chuẩn cho phép con chạy (cusor) định. Nếu trình biên dịch của bạn có chức năng như thế, hãy tạo hàm **myclreol()** để xóa dòng từ vị trí con chạy đến cuối dòng. Tuy nhiên, hãy cho hàm này này một tham số vị trí của vị trí được xóa. Ngược lại, chỉ xóa những vị trí ký tự nào được chỉ rõ bởi tham số.

4. *What is wrong with the following prototype, which uses a default argument?*

Có gì sai trong nguyên mẫu dùng đối số mặc định sau đây?

```
int f(int count , int max = count);
```

5.5. OVERLOADING AND AMBIGUITY - SỰ QUÁ TẢI VÀ TÍNH KHÔNG XÁC ĐỊNH:

When you are overloading functions, it is possible to introduce ambiguity into your program. Overloading-caused ambiguity can be introduced through type conversions, reference parameters, and default arguments. Further, some types of ambiguity are caused by the overloaded function themselves. Other types occur in the manner in which an overloaded function is called. Ambiguity must be removed before your program will compile without error.

Khi quá tải các hàm, có thể dẫn đến tính không xác định (ambiguity) trong chương trình của ban. Tính không xác định do quá tải gây ra có thể được đưa vào thông qua các chuyển đổi kiểu, các tham số tham chiếu và các đối số mặc định. Hơn nữa, một số loại không xác định gây ra do chính các hàm được quá tải. Các loại khác xảy ra do cách gọi các hàm được quá tải. Tính không xác định phải được loại bỏ trước khi chương trình của bạn biên dịch không có lỗi.

EXAMPLES

CÁC VÍ DỤ

One of the most common types of ambiguity is caused by C++'s automatic type conversion rules. As you know, when a function is called with an argument that is of a compatible (but not the same) type as the parameter to which it is being passed, the type of the argument is automatically converted to the target type. In fact, it is this sort of type conversion that allows a function such as ***putchar()*** *to be called with a character even though its argument is specified as an* ***int****. However, in some cases, this automatic type conversion will cause an ambiguous situation when a function is overloaded. To see how, examine this program:*

Một trong những loại thông dụng nhất của tính không xác định gây ra bởi các quy tắc chuyển đổi kiểu tự động của C++. Như bạn biết, khi một hàm được gọi có một đối số phải có kiểu tương thích (nhưng không giống) như đối số được truyền cho hàm, kiểu của đối số tự động được chuyển đổi thành kiểu đích. Điều này đôi khi được xem là *sự xúc tiến kiểu* (type promotion), và hoàn toàn đúng. Thực tế, chính loại chuyển đổi kiểu này cho phép một hàm như **putchar()** được gọi với một ký tự mặc dù đối số của nó được chỉ rõ kiểu **int**. Tuy nhiên, trong một số trường hợp, loại chuyển đổi kiểu tự động này sẽ gây ra tình trạng không xác định khi một hàm được quá tải. Để thấy rõ như thế nào, hãy xét chương trình sau:

```
// This program contains an ambiguity error.

 #include <iostream>

 using namespace std;

 float f(float i)

 {
```

```
        return i/2.0;
  }
  double f(double i)
  {
        return i/3.0;
  }
  int main()
  {
float x = 10.09;
double y = 10.09;
cout << f(x); // unambiguous – use f(float)
cout << f(y); // unambiguous – use f(double)
cout << f(10); // ambiguous, convert 10 to double or
     float??
return 0;
  }
```

*As the comments in **main()** indicate, the compiler is able to select the correct version of **f()** when it is called with either a **float** or a **double** variable. However, what happens when it is called with an integer? Does the compiler call **f(float)** or **f(double)**? (Both are valid conversions!). In either case, it is valid to promote an integer into either a **float** or a **double**. Thus, the ambiguous situation is created.*

Như những lời chú giải trong **main()**, trình biên dịch có thể lựa chọn phiên bản đúng của **f()** khi nó được gọi với hoặc một biến **float** hoặc một biến **double**. Tuy nhiên, điều gì xày ra khi nó được gọi với một số nguyên? Trình biên dịch gọi **f(float)** hoặc **f(double)**? (Cả hai là những chuyển đổi có hiệu lực!). Trong mỗi trường hợp, xúc tiến một số nguyên thành hoặc **float** hoặc **double** đều đúng. Do đó, tình trạng không xác định đã xảy ra.

This example also points out that ambiguity can be introduced by the way an

*overloaded function is called. The fact is that there is no inherent ambiguity in the overloaded versions of **f()** as long as each is called with an unambiguous argument.*

Ví dụ này cũng chỉ ra rằng tính không xác định có thể xảy ra khi một hàm quá tải được gọi. Tính không xác định không phải là có sẵn trong các phiên bản quá tải của **f()** miễn là mỗi lần được gọi với đối số xác định.

Here is another example of function overloading that is not ambiguous in and of itself. However, when this function is called with the wrong type of argument, C++'s automatic conversion rules cause an ambiguous situation.

Đây là một ví dụ khác về sự quá tải hàm mà tự nó không có tính không xác định. Tuy nhiên, khi được gọi với kiểu đối số sai, các quy tắc chuyển đổi tự động của C++ gây ra tình trạng không xác định.

```
// This program is ambiguous.

#include <iostream>

using namespace std;

void f(unsigned char c)

{

 cout << c;

}

 void f(char c)

 {

  cout << c;

}

int main()

{

   f('c');

   f(86); // which f() is called???
```

```
    return 0;
}
```

*Here, when **f()** is called with the numeric constant 86, the compiler can not know whether to call **f(unsigned char)** or **f(char)**. Either conversion is equally valid, thus leading to ambiguity.*

Ở đây, khi **f()** được gọi với hằng số 86, trình biên dịch không thể biết là gọi **f(unsigned char)** hay gọi **f(char)**. Mỗi chuyển đổi đều đúng nên dẫn đến tính không xác định.

One type of ambiguity is caused when you try to overload functions in which the only difference is the fact that one uses a reference parameter and the other uses the default call- by-value parameter. Given C++'s formal syntax, there is no way for the compiler to know which function to call. Remember, there is no value parameter and calling a function that takes a reference parameter. For example:

Một loại tính không xác định xảy ra do bạn quá tải các hàm trong đó có sự khác biệt duy nhất là một hàm sử dụng một tham số tham chiếu và hàm kia sử dụng tham số mặc định gọi bằng giá trị. Cho cú pháp chính thức của C++, không có cách để cho trình biên dịch biết hàm nào được gọi. Cần nhớ, không có sự khác biệt về mặt cú pháp giữa cách gọi hàm nhận tham số giá trị với cách gọi hàm nhận tham số tham chiếu. Ví dụ:

```
// An ambiguous program.
#include <iostream>
using namespace std;

int f(int a, int b)
{
     return a+b;
}
```

```
// this is inherently ambiguous
int f(int a, int &b)
{
     return a-b;
}

int main()
{
     int x=1, y=2;
     cout << f(x,y); // which version of f() is called???
     return 0;
}
```

*Here, **f(x,y)** is ambiguous because it could be calling either version of the function. In fact, the compiler will flag an error before this statement is even specified because the overloading of the two functions is inherently ambiguous and no reference to them could be resolved.*

Ở đây, **f(x,y)** là không xác định vì có thể gọi mỗi phiên bản của hàm. Thực tế, trình biên dịch sẽ báo hiệu lỗi khi câu lệnh này được chỉ định vì sự quá tải hai hàm là hoàn toàn không xác định và không có tham chiếu nào tới chúng được giải quyết.

Another type of ambiguity is caused when you are overloading a function in which one or more overloaded functions use a default argument. Consider this program:

Một loại tính không xác định khác xảy ra khi quá tải hàm trong đó có một hay nhiều hàm được quá tải dùng một đối số mặc định; hãy xét chương trình sau.

```
// Ambiguity based on default arguments plus overloading.
#include <iostream>
using namespace std;

int f(int a)
{
     return a*a;
 }

 int f(int a, int b = 0)
 {
      return a*b;
 }

 int main()
 {
      cout << f(10, 2); // calls f(int, int)
      cout << f(10); // ambiguous - call f(int) or f(int, int)???
      return 0;
 }
```

Here the call **f(10,2)** is perfectly acceptable and unambiguous. However, the compiler has no way of knowing whether the call **f(10)** is calling the first version of **f()** or the second version with **b** defaulting.

Lời gọi **f(10,2)** hoàn toàn được chấp nhận và xác định. Tuy nhiên, trình biên dịch không có cách để biết **f(10)** gọi phiên bản thứ nhất của **f()** hay phiên bản thứ hai với **b** là mặc định.

EXERCISE

BÀI TẬP

Try to compile each of the preceding ambiguous programs. Make a mental note of the types of error messages they generate. This will help you recognize ambiguity errors when they creep into your own programs.

Hãy thử biên dịch một chương trình trong các chương trình không xác định trên đây. Hãy chú ý các loại thông báo lỗi. Điều này sẽ giúp bạn nhận ra các lỗi của tính không xác định khi các lỗi này có trong chương trình riêng của bạn.

5.6. FINDING THE ADDRESS OF AN OVERLOADED FUNCTION - TÌM ĐỊA CHỈ CỦA MỘT HÀM QUÁ TẢI:

To conclude this chapter, you will learn how to find the address of an overloaded function. Just as in C, you can assign the address of a function (that is, its entry point) to a pointer and access that function via that pointer. A function's address is obtained by putting its name on the right side of an assignment statement without any parentheses or arguments. For example, it ***zap()*** *is a function, assuming proper declarations, this is a valid way to assign* ***p*** *the address of* ***zap()*** *:*

Để kết thúc chương này, bạn sẽ học cách tìm địa chỉ của một hàm quá tải. Như trong C, bạn có thể gán địa chỉ của một hàm (nghĩa là điểm nhập của nó) cho một con trỏ và truy cập hàm đó thông qua con trỏ. Địa chỉ của một hàm có được bằng cách đặt tên hàm ở bên phải của câu lệnh gán mà không có dấu đóng mở ngoặc hoặc đối số. Ví dụ, nếu **zap()** là hàm, giả sử các khai báo đều đúng, thì cách hợp lệ để gán địa chỉ của **zap()** cho **p** là:

```
p = zap;
```

In C, any type of pointer can be used to point to a function because there is only one function that it can point to. However, in C++ the situation is a bit more complex because a function can be overloaded. Thus, there must be some mechanism that determines which function's address is obtained.

Trong C, một kiểu con trỏ bất kỳ được dùng để trỏ tới một hàm bởi vì chỉ có một hàm mà nó có thể trỏ tới. Tuy nhiên, trong C++, tình trạng hơi phức tạp hơn bởi vì hàm có thể được quá tải. Do đó, phải có một cơ chế nào đó để xác định cần có địa chỉ của hàm nào.

The solution is both elegant and effective. When obtaining the address of an overloaded function, it is the way the pointer is declared that determines which overloaded function's address will be obtained. In essence, the pointer's declaration is matched against those of the overloaded functions. The function whose declaration matches is the one whose address is used.

Giải pháp là rất đẹp và có hiệu quả. Khi thu được địa chỉ của một hàm quá tải, chính *cách khai báo con trỏ* xác định địa chỉ của hàm quá tải nào sẽ thu được. Thực chất, sự khai báo con trỏ phù hợp với cách khai báo của các hàm quá tải. Hàm có sự phù hợp khai báo là hàm có địa chỉ được sử dụng.

EXAMPLE

VÍ DỤ

*Here is a program that contains two versions of a function called **space()**. The first version outputs **count** number of spaces to the screen. The second version outputs **count** number of whatever type of character is passed to **ch**. In **main()**, two function pointers are declared. The first one is specified as a pointer to a function having only one integer parameter. The second is declared as a pointer to a function taking two parameters.*

Đây là chương trình chứa hai phiên bản của hàm **space**. Phiên bản thứ nhất xuất **count** là số khoảng trống trên màn hình. Phiên bản thứ hai xuất **count** là số bất kỳ kiểu ký tự nào được truyền cho **ch**. Trong **main()**, hai con trỏ hàm được khai báo. Con trỏ thứ nhất được chỉ rõ khi trỏ tới hàm chỉ có một tham số nguyên: Con trỏ thứ hai được khai báo như con trỏ tới hàm nhận hai tham số:

```
/*   Illustrate   assigning   function   pointers   to
overloaded functions. */

#include <iostream>

using namespace std;

// Output count number of spaces.

void space (int count)

{

     for( ; count; count--)

cout << ' ';

}

// Output count number of chs.

void space (int count, char ch)

{

     for( ; count; count--)

          cout << ch;

}

int main()

{

     /* Create a pointer to void function with

     one int parameter.*/
```

```
    void (*fp1)(int);
    /* Create a pointer to void function with
    One int parameter and one character parameter.*/
    void (*fp2)(int, char);
fp1 = space; // gets address of space (int)
fp2 = space; // gets address of space (int, char)
fp1(22); // output 22 spaces
cout << "|\n";
fp2(30, 'x'); // output 30 x's
cout << " |\n";
return 0;
  }
```

*As the comments illustrate, the compiler is able to determine which overloaded function to obtain the address of based upon how **fp1** and **fp2** are declared.*

Như các lời chú giải minh họa, trình biên dịch có thể xác định hàm quá tải nào nhờ thu được địa chỉ căn cứ vào cách khai báo của **fp1** và **fp2**.

To review: When you assign the address of an overloaded function to a function pointer, it is the declaration of the pointer that determines which function's address is assigned. Further, the declaration of the function pointer must exactly match one and only one of the overloaded functions. If it does not, ambiguity will be introduced, causing a compile-time error.

Tổng quan: Khi bạn gán địa chỉ của một hàm quá tải cho một con trỏ hàm, chính sự khai báo con trỏ xác định địa chỉ của hàm nào được gán. Hơn nữa, khai báo con trỏ hàm phải phù hợp chính xác với một và chỉ một hàm quá tải. Nếu không, tính không xác định sẽ xảy ra và gây ra lỗi thời gian biên dịch.

EXERCISE

BÀI TẬP

Following are two overloaded functions. Show how to obtain the address of each.

Sau đây là hai hàm quá tải. Hãy trình bày cách để có được địa chỉ của mỗi hàm.

```
int dif(int a, int b)
{
   return a-b;
}
float dif (float a, float b)
{
   return a-b;
}
```

SKILLS CHECK (KIỂM TRA KỸ NĂNG):


Mastery

Skills Check

C++ Kiểm tra kỹ năng lĩnh hội

At this point you should be able to perform the following exercises and answer the questions.

Ở đây bạn có thể làm các bài tập và trả lời các câu hỏi sau:

*Overload the **date()** constructor from Section 5.1, Example 3, so that it accepts a parameter of type **time_t**. (Remember **time_t** is a type defined by the standard time and date functions found in your C++ compiler's library.)*

Hãy quá tải hàm tạo **date()** trong phần 5.1, ví dụ 3 để cho nó nhận một tham số kiểu **time_t**. (Nên nhớ, **time_t** là một kiểu được định nghĩa bởi các hàm về thời gian và ngày tháng chuẩn trong thư viện của trình biên dịch C++ của bạn).

What is wrong with the following fragment?

Điều gì sai trong đoạn chương trình sau?

```
class samp
{
   int a;
   public:
    samp (int i)
    {
```

```
        a = i;
    }
    // …
};
// …
int main()
{
   samp x, y(10);
   // …
}
```

Give two reasons why you might want (or need) to overload a class's constructor.

Hãy trình bày hai lý do tại sao bạn muốn (hoặc cần) quá tải hàm tạo của một lớp.

What is the most common general form of a copy constructor?

Dạng tổng quát của hàm tạo bản sao là gì?

What type of operations will cause the copy constructor to be invoked?

Kiểu phép toán nào sẽ làm cho hàm tạo bản sao được dùng đến?

*Briefly explain what the **overload** keyword does and why it is no longer needed?*

Giải thích ngắn gọn từ khóa **overload** là gì và tại sao không còn cần dùng nữa?

Briefly describe a default argument.

Trình bày ngắn gọn đối số mặc định là gì?

*Create a function called **reverse()** that takes two parameters. The first parameter, called **str**, is a pointer to a string that will be reversed upon return from the function. The second parameter is called **count**, and it specifies how many characters of **str** to reverse. Give **count** a default value that, when present, tells **reverse()** to reverse the entire string.*

Hãy tạo hàm reverse nhận hai tham số. Tham số thứ nhất, gọi là str, là một con trỏ trỏ tới chuỗi mà chuỗi này được đảo ngược khi trả về từ hàm. Tham số thứ hai gọi là count và nó chỉ rõ có bao nhiêu ký tự của chuỗi để đảo ngược. Hãy cho count một giá trị mặc định để báo cho reverse() đảo ngược toàn bộ chuỗi.

What wrong with the following prototype?

Điều gì sai trong nguyên mẫu sau?

```
char *wordwrap (char *str, int size=0, char ch);
```

Explain some ways that ambiguity can be introduced when you are overloading functions.

Hãy giải thích theo nhiều cách xảy ra tính không xác định khi quá tải các hàm.

What is wrong with the following fragment?

Điều gì sai trong đoạn chương trình sau?

```
void compute (double *num, int divisor=1);
void compute (double *num);
```

```
// …

compute(&x);
```

When you are assigning the address of an overloaded function to a pointer, what is it that determines which version of the function is used?

Khi gán địa chỉ của một hàm quá tải cho một con trỏ, chính điều gì xác định phiên bản nào của hàm được dùng?

Skills Check

Kiểm tra kỹ năng tổng hợp

This section checks how well you have integrated material in this chapter with that from the preceding chapters.

Phần này kiểm tra xem bạn kết hợp chương này với những chương trước như thế nào.

*Create a function called **order()** that takes two integer reference parameters. If the first argument is greater than the second argument, reverse the two arguments. Otherwise, take no action. That is, order the two arguments used to call **order()** so that, upon return, the first argument will be less than the second. For example, given*

Hãy tạo hàm order() để nhận hai tham số tham chiếu nguyên. Nếu đối số thứ nhất lớn hơn đối số thứ hai, hãy đảo ngược hai số đối. Ngược lại, không có tác động nào. Nghĩa là, thứ tự của hai đối số được dùng để gọi order() sao cho, khi trả về, đối số thứ nhất sẽ nhỏ hơn đối số thứ hai. Ví dụ, cho:

```
int x=1, y=0;
order (x,y);
```

following the call, x will be 0 and y will be 1.

sau khi gọi, x sẽ là 0 và y sẽ là 1.

Why are the following two overloaded functions inherently ambiguous?

Tại sao hai hàm quá tải sau đây vốn không xác định?

```
int f(int a);
int f(int &a);
```

Explain why using a default argument is related to function overloading.

Hãy giải thích tại sao dùng đối số mặc định có liên quan đến sự quá tải hàm.

*Given the following partial class, add the necessary constructor functions so that both declarations within **main()** are valid. (Hint: You need to overload **samp()** twice.)*

Cho lớp riêng phần sau, hãy bổ sung các hàm tạo cần thiết để cho cả hai khai báo trong **main()** đều đúng. (Hướng dẫn: Bạn cần quá tải **samp()** hai lần).

```
class samp
{
     int a;
     public:
          // add constructor functions
          int get_a()
          {
               return a;
          }
};
int main()
{
     samp ob(88); // int ob's a to 88
     samp  obarray[10];  //  noninitialized  10-element
array
     // …
}
```

Briefly explain why copy constructor.

Hãy giải thích ngắn gọn là tại sao cần các hàm tạo bản sao?

_____

_____

CHƯƠNG 6

INTRODUCING OPERATOR OVERLOADING

GIỚI THIỆU VỀ NẠP CHỒNG TOÁN TỬ

Chapter objectives:

6.1. THE BASICS OF OPERATOR OVERLOADING

Cơ sở của sự nạp chồng toán tử

6.2. OVERLOADING BINARY OPERATORS

Nạp chồng toán tử nhị nguyên

6.3. OVERLOADING THE RELATIONAL AND LOGICAL OPERATORS

Nạp chồng các toán tử quan hệ và luận lý

6.4. OVERLOADING A NUARY OPERATOR

Nạp chồng toán tử đơn nguyên

6.5. USING FRIEND OPERATOR FUNCTIONS

Sử dụng hàm toán tử friend

6.6. A CLOSER LOOK AT THE ASSIGNMENT OPERATOR

Toán tử gán

6.7. OVERLOADING THE [ ] SUBSCRIPT OPERATOR

Nạp chồng toán tử chỉ số dưới []

This chapter introduces another important C++ feature: operator overloading. This feature allows you to define the meaning of the C++ operators relative to classes that you define. By overloading operators, you can seamlessly add new data types to your

program.

Chương này giới thiệu một đặc điểm quan trọng khác của C++: sự Nạp chồng toán tử. Đặc điểm này cho phép bạn xác định ý nghĩa của các toán tử C++ liên quan với các lớp mà bạn định nghĩa. Bằng cách thực hiện Nạp chồng các toán tử đối với lớp, bạn có thể bổ sung các loại dữ liệu mới cho chương trình của bạn

REVIEW SKILLS CHECK (Kiểm tra kĩ năng ôn)

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Trước khi bắt đầu, bạn nên trả lời đúng các câu hỏi và làm các bài tập sau:

*Show how to overload the constructor for the following class to that uninitialized objects can also be created. (When creating uninitialized objects, give **x** and **y** the value 0).*

Hãy thực hiện Nạp chồng cho hàm tạo đối với lớp sau đây sao cho các đối tượng không được khởi đầu cũng được tạo ra. (Khi tạo ra các đối tượng chưa được khởi đầu, cho **x** va **y** giá tri 0).

```
class myclass
{
      int x,y;
      public:
            myclass( int i, int j) { x=i; y=j;}
      //...
};
```

*Using the class from Question 1, show how you can avoid overloading **myclass()** by using default arguments.*

Sử dụng lớp trong câu hỏi 1, hãy chứng tỏ có thể không nạp chồng **myclass()** bằng cách dùng các đối số mặc định.

What is wrong with the following declaration?

Điều gì sai trong khai báo sau đây:

```
int f(int a=0, double balance);
```

What is wrong with these two overloaded functions?

Điều gì sai trong hai hàm nạp chồng sau đây?

```
void f (int a);

void f (int &a);
```

When is it appropriate to use default arguments? When is it probably a bad idea?

Dùng các đối số mặc định khi nào là thích hợp? Khi nào nó là một ý tưởng không hay?

Given the following class definition, is it possible to dynamically allocate an array of these objects?

Cho định nghĩa lớp dưới đây, có thể cấp phát động một mảng các đối tượng này?

```
class test
{
      char *p;
      int *q;
      int count;
      public:
            test(char *x, int *y, int c)
            {
                  p = x;
                  q = y;
                  count = c;
            }
            //...
};
```

What is a copy constructor and under what circumstances is it calles?

Hàm tạo sao chép là gì? Và nó được gọi trong trường hợp nào?

THE BASICS OF OPERATOR OVERLOADING - CƠ SỞ CỦA QUÁ TẢI TOÁN TỬ

Operator overloading resembles function overloading. In fact, operator overloading is really just a type of function overloading. However, some additional rules apply. For example, an operator is always overloaded relative to a user-defined type, such as a class. Other differences will be discussed as needed.

Sự Nạp chồng toán tử giống như sự Nạp chồng hàm. Thật vậy, sự Nạp chồng toán tử chỉ là một loại Nạp chồng hàm. Tuy nhiên, nó có một số quy tắc bổ sung. Ví dụ, một toán tử thường được Nạp chồng đối với một lớp. Những điểm khác biệt khác sẽ được trình bày khi cần thiết.

When an operator is overloaded, that operator loses none of its orginal meaning. Instead, it gains additional meaning relative to the class for which it is defined.

Khi một toán tử được Nạp chồng, toán tử đó không mất ý nghĩa gốc của nó. Thêm nữa, toán tử còn có thêm ý nghĩa bổ sung đối với lớp mà toán tử được định nghĩa.

To overload an operator, you create an operator function. Most often an operator function is a member or a friend of the class for which it is defined. However, there is a slights difference between a member operator function and a friend operator function. The first part of this chapter discusses the creation of member operator functions. Then friend operator functions are discussed.

Để Nạp chồng một toán tử, bạn tạo ra một hàm toán tử (operator function). Thường một hàm toán tử là một thành viên (member) hoặc bạn (friend) của lớp mà toán tử được định nghĩa. Tuy nhiên có sự khác biệt nhỏ giữa một hàm toán tử thành viên và một hàm toán tử friend. Phần đầu của chương trình bày cách tạo ra các hàm toán tử thành viên. Các hàm toán tử friend sẽ được thảo luận sau đó.

The general form of a member operator function is shown here:

Dạng tổng quát của một hàm toán tử thành viên như sau:

```
return-type class-name::operator#(arg-list)
{
      // operator to be performed
```

```
}
```

The return of an operator function is often the class for which it is defined. (However, an operator function is free to return any type). The operator being overloaded is substituted for the #. For example, if the + is being overloaded, the function name would be ***operator+****. The contents of arg-list vary depending upon how the operator function is implemented and the type of operator being overloaded.*

Kiểu trả về của một hàm toán tử là lớp mà toán tử được định nghĩa. (Tuy nhiên, môt hàm toán tử trả về kiểu bất kỳ). Toán tử được Nạp chồng thay thế được cho ký tự #. Ví dụ, nếu toán tử + được Nạp chồng thì tên hàm số là **operator+**. Nội dung của *arg-list* thay đổi phụ thuộc vào cách mà hàm toán tử được thực hiện và kiểu toán tử được Nạp chồng.

There are two important restrictions to remember when you are overloading an operator. First, the precedence of the operator cannot be changed. Second, the number of operands that an operator takes cannot be altered. For example, you cannot overload the / operator so that it takes only one operand.

Có hai hạn chế quan trọng cần nhớ khi Nạp chồng một toán tử. Thứ nhất thứ tự ưu tiên của các toán tử không thay đổi. Thứ hai, số toán hạng của một toán tử không thay đổi. Ví dụ, bạn không thể Nạp chồng toán tử / để cho nó có chỉ một toán hạng.

Most C++ operators can be overloaded. The only operators that you cannot overload are shown here:

Hầu hết các toán hạng trong C++ có thể được Nạp chồng. Các toán tử không thể Nạp chồng được là:

. :: * ?

Also, you cannot overload the preprocessor operators. (the ***** *operator is highly specialized and is beyond the scope of this book).*

Cũng vậy, bạn không thể Nạp chồng các toán tử tiền xử lý. (Toán tử được chuyện dụng cao và ở ngoài phạm vi cuốn sách).

Remember that C++ defines operators very broadly, including such things as the ***[ ]*** *subscript operators, the* ***()*** *function call operators,* ***new*** *and* ***delete****, and the* **.** *(dot) and b* **->** *(arrow) operators. However, this chapter concentrates on overloading the most commonly used operators.*

Nhớ rằng C++ định nghĩa các toán tử rất rộng, gồm những toán tử như toán tử chỉ số dưới **[ ]** và toán tử gọi hàm **()**, **new** và **delete**, và toán tử **.** (chấm) và toán tử **->** (mũi tên).

Tuy nhiên, chương này chỉ tập trung vào sự Nạp chồng các toán tử thường được sử dụng nhất.

Except for the =, operator functions are inherited by any derived class. However, a derived class is free to overload any operator it chooses (including those overloaded by the base class) relative to itself.

Ngoại trừ toán tử =, các hàm toán tử được kế thừa bởi bất kỳ lớp dẫn xuất nào. Tuy nhiên, một lớp dẫn xuất hoàn toàn Nạp chồng được bất kỳ toán tử nào mà nó chọn (gồm cả những toán tử được Nạp chồng bởi lớp cơ sở) đối với nó.

You have been using two overloaded operators: << and >>. These operators have been overloaded to perform console I/O. As mentioned, overloading these operators to perform I/O does not prevent them from performing their traditional jobs of left shift and right shift.

Bạn đã sử dụng hai toán tử được nạp chồng: << và >>. Những toán tử này được Nạp chồng để thực hiện giao tiếp nhập xuất. Như đã nói, sự Nạp chồng các toán tử này để thực hiện nhập xuất / xuất không ngăn trở chúng thực hiện các công việc truyền thống là dịch trái và phải.

While it is permissible for you to have an operator function perform any activity-whether related to the traditional use of the operator or not-it is best to have an overloaded operator's actions stay within the spirit of the operator's traditional use. When you create overloaded operators that stray from this principle, you run the risk of substantially destructuring your program. For example, overloading the / so that the phrase "I like C++" is written to a disk file 300 times is a fundamentally confusing misuse of operator overloading!

Trong khi bạn được phép có một hàm toán tử thực hiện một hoạt động bất kỳ - dù có liên quan đến cách sử dụng các toán tử theo truyền thống hay không – thì tốt nhất nên để cho các tác động của các toán tử được Nạp chồng theo tinh thần sử dụng truyền thống của các toán tử. Khi bạn tạo ra các toán tử Nạp chồng không tuân theo nguyên tắc này, thì có nguy cơ bạn sẽ làm hỏng chương trình của bạn. Ví dụ, nạp chồng toán tử / sao cho cụm từ "I like C++" được viết vào một file đĩa 300 lần thì cơ bản đã tạo ra cách sử dụng sai sự Nạp chồng của toán tử.

The preceding paragraph notwithstanding, there will be times when yo need to use an operator in a way not related to its traditional usage. The two best examples of this are the <<and >> operators, which are overloaded for console I/O. However, even in these cases, the left and right arrows provide a visual "clue" to their meaning. Therefore, if you need to overload an operator in a nonstandard way, make the greatest effort possible to use an appropriate operator.

Mặc dù đoạn trên đây, nhưng có những lúc bạn sẽ cần sử dụng một toán tử không có liên quan gì đến cách sử dụng truyền thống của toán tử. Hai ví dụ tốt nhất về điều này là các toán tử << và >> được Nạp chồng cho giao tiếp nhập / xuất. Tuy nhiên, ngay cả trong những trường hợp này, các mũi tên hướng phải và hướng trái cũng đã nói lên ý nghĩa của các toán tử. Do đó nếu bạn cần Nạp chồng một toán tử theo cách không chuẩn, thì hãy cố gắng sử dụng một toán tử thích hợp. Điểm cuối cùng: các hàm toán tử có thể không có các đối số ngầm định.

One final point: operator functions cannot have default arguments.

Điều lưu ý cuối cùng: các hàm toán tử không có các đối số ngầm định.

OVERLOADING BINARY OPERATORS - QUÁ TẢI TOÁN TỬ NHỊ NGUYÊN

*When a member operator function overloads a binary operator, the function will have only parameter. This parameter will receive the object that is on the right side of the operator. The object on the left side is the object that generates the call to the operator function and is passed implicitly by **this**.*

Khi một hàm toán tử thành viên Nạp chồng một toán tử nhị nguyên, hàm sẽ chỉ có một tham số. Tham số này sẽ nhận đối tượng nằm bên phải toán tử. Đối tượng bên trái là đối tượng tạo lời gọi cho hàm toán tử và được truyền bởi **this**.

It is important to understand that operator functions can be written with many variations. The examples here and elsewhere in this chapter are not exhaustive, but they do illustrate several of the most common techniques.

Điều quan trọng là các hàm toán tử được viết theo nhiều cách khác nhau. Các ví dụ dưới đây và ở những phần khác nhau trong chương này không hẳn là đầy đủ nhưng cũng minh họa được nhiều kỹ thuật thông dụng nhất.

EXAMPLES: (VÍ DỤ)

1. *The following program overloads the + operator relative to the **coord** class. This class is used to maintain X,Y coordinates.*

Chương trình sau thực hiện Nạp chồng cho toán tử + đối với lớp **coord**. Lớp này được dùng để duy trì các tọa độ **X,Y**.

```
// Overload the + relative to coord class.
#include "iostream"
using namespace std;

class coord
{
      int x,y; // coordinate values
      public:
            coord() { x=0; y=0; }
            coord(int i, int j) { x=i; y=j; }
            void get_xy(int &i, int &j) { i=x;j=y;}
            coord operator + (coord ob2);
};

// overload + relative to coord class.
coord coord::operator +( coord ob2)
{
      coord temp;
      temp.x = x + ob2.x;
      temp.y = y + ob2.y;

      return temp;
}

int main()
{
      coord o1 (10, 10) , o2 (5, 3), o3;
```

```
        int x,y;


        o3 = o1 + o2; // add two objects - this
                        // calls operator + ()
        o3.get_xy(x,y);
        cout<<"(o1+o2) X: "<<x<<", y: "<<y<<"\n";


        return 0;
}
```

This program displays the following:

Chương trình này hiển thị:

```
(o1 + o2) X: 15, Y: 13
```

*Let's look closely at this program. The **operator +()** function returns an object of type **coord** that has the sum of each operand's **X** coordinates in **x** and the sum of the **Y** coordinates in **y**. Notice the a temporary object called **temp** is used inside **operator +()** to hold the result, and it is this object that is returned. Notice also that neither operand is modified. The reason for **temp** is easy to understand. In this situation (as in most), the + has been overloaded in a manner consistent with neither operand be changed. For example, when you add 10 + 4, the result is 14, but neither 10 nor the 4 is modified. Thus, a temporary object is needed to hold the result.*

Hãy xét kỹ hơn chương trình này. Hàm **operator +()** trả về một đối tượng có kiểu **coord** là tổng của các tọa độ của mỗi toán hạng theo **x** và tổng của các tọa độ **Y** theo **y**. Chú ý rằng đối tượng tạm thời **temp** được dùng bên trong **operator+()** để giữ kết quả, và đó là đối tượng được trả về. Cũng chú ý rằng toán hạng không được sửa đổi. Lý do đối với **temp** thì dễ hiểu. Trong trường hợp này, toán tử + được Nạp chồng phù hợp với cách sử dụng số học của nó. Do đó, điều quan trọng là toán hạng không được thay đổi. Ví dụ, khi bạn cộng 10+4, kết quả là 14, nhưng 10 và 4 là không thay đổi. Do đó, cần một đối tượng tạm thời để giữ kết quả.

*The reason that the **operator+()** function returns an object of type **coord** is that it allows the result of the addition of **coord** objects to be used in larger expressions. For examples, the statement.*

Lý do mà hàm **operator+()** trả về đối tượng có kiểu **coord** là nó cho phép kết quả của phép cộng các đối tượng **coord** sử dụng trong biểu thức lớn hơn. Ví dụ, câu lệnh:

```
O3 = o1 + o2;
```

*is valid only because the result of **o1 + o2** is a **coord** object that can be assigned to **o3**. if a different type had been returned, this statement would have been invalid. Further, by returning a **coord** object, the addition operator allows a string of additions. For example, this is a valid statement:*

là đúng chỉ vì kết quả của **o1 + o2** là đối tượng **coord** được gán cho o3. Nếu một kiểu khai thác được trả về, câu lệnh này sẽ không đúng. Hơn nữa, khi trả về một đối tượng, toán tử phép cộng cho phép một chuỗi phép cộng. Ví dụ, đây là câu lệnh đúng:

```
o3 = o1 + o2 + o1 + o3;
```

Although there will be situations in which you want an operator function to return something other than an object for which it is defined, most of the time operator functions that you create will return an object of their class. (The major exception to this rule is when the relational an logical operators are overloaded. This situationis examined in Section 6.3. "Overloading the Relational and Logical Operators", later in this chapter).

Mặc dù có nhiều trường hợp bạn muốn một hàm toán tử trả về một đối tượng nào đó thay vì đối tượng mà hàm toán tử định nghĩa, hầu hết các hàm toán tử mà bạn tạo ra sẽ trả về một đối tượng của lớp mà đối với lớp này, các hàm toán tử được định nghĩa (ngoại lệ quan trọng đối với quy tắc này là khi các toán tử quan hệ và các toán tử luận lý được Nạp chồng). Trường hợp này được xét đến trong phần cuối của chương này.

*One final point about this example. Because a **coord** object is returned, the following statement is also perfectly valid:*

Điều cuối cùng của ví dụ này. Vì đối tượng **coord** được trả về, câu lệnh sau đây cũng hoàn toàn đúng.

```
(O1 + o2).get_xy(x,y);
```

*Here the temporary object returned by **operator +()** is used directly. Of course, after*

this statement has executed, the temporary object is destroyed.

Ở đây, đối tượng tạm thời được trả về bởi **operator+()** được sử dụng trực tiếp. Dĩ nhiên, sau khi câu lệnh này được thi hành, đối tượng tạm thời sẽ bị hủy bỏ.

*2. The following version of the preceding program overloads the – and the = operators relative to the **coord** class.*

Dạng sau đây của chương trình trên thực hiện Nạp chồng các toán tử - và = đối với lớp **coord**.

```
#include "iostream"
using namespace std;
class coord
{
      int x,y; // coordinate values
      public:
            coord() { x=0; y=0;}
            coord(int i, int j)
            {
                  x=i; y=j;
            }
            void get_xy(int &i, int &j)
            {
                  i=x; j=y;
            }
            coord operator + (coord ob2);
            coord operator - (coord ob2);
            coord operator = (coord ob2);
};
```

```
// Overload + relative to coord class.
coord coord::operator + (coord ob2)
{
      coord temp;

      temp.x = x + ob2.x;
      temp.y = y + ob2.y;

      return temp;
}

// Overload - relative to coord class.
coord coord::operator - (coord ob2)
{
      coord temp;

      temp.x = x - ob2.x;
      temp.y = y - ob2.y;

      return temp;
}
// Overload + relative to coord class.
coord coord::operator = (coord ob2)
{
      x = ob2.x;
      y = ob2.y;
```

```
        return *this; // return the object that is
                        //assigned
}
int main()
{
        coord o1(10, 10), o2(5, 3), o3;
        int x,y;

        o3 = o1 + o2; // add two objects - this //calls
                       operator +()
        o3.get_xy(x,y);
        cout<<"(o1+o2) X: "<<x<<", Y: "<<y<<"\n";

        o3 = o1 - o2; // subtract two objects
        o3.get_xy(x,y);
        cout<<"(o1-o2) X: "<<x<<", Y: "<<y<<"\n";

        o3 = o1; // assign an object
        o3.get_xy(x,y);
        cout<<"(o1=o2) X: "<<x<<", Y: "<<y<<"\n";

        return 0;
}
```

*The **operator –()** function is implemented similarly to **operator +()**. However, the above example illustrates a crucial point when you are overloading an operator in which the order of the operands is important. When the **operator +()** function was created, it did not matter which order the operands were in. (That is, A+B is the same as B+A). However, the subatraction operation is order dependent. Therefore, to correctly overload the subtraction operator, it is necessary to subtract the operand on the right*

*from the operand on the left. Because it is the left operand that generates the call to **operator –()**, the subtraction must be in this order:*

Hàm **operator-()** được thực hiện tương tự như **operator+()**. Tuy nhiên, nó minh họa một điểm cần chú ý khi Nạp chồng một toán tử trong đó thứ tự của các toán hạng là quan trọng. Khi hàm **operator+()** được tạo ra, nó không chú ý đến thứ tự của các toán hạng như thế nào. (Nghĩa là, A + B cũng giống B + A). Tuy nhiên, phép trừ thì lại phụ thuộc vào thứ tự này. Do đó, để Nạp chồng đúng một toán tử phép trừ thì cần phải lấy toán hạng bên trái trừ toán hạng bên phải. Vì chính toán hạng bên trái tạo ra lời gọi đối với **operator-()** nên phép trừ phải theo thứ tự:

```
x = ob2.x;
```

Remember: *when a binary operator is overloaded, the left operand is passed implicitly to the function and the right operand is passed as an argument.*

Cần nhớ: *khi một toán tử nhị nguyên được Nạp chồng, toán hạng bên trái được truyền cho h‡m v‡ toán hạng bên phải được truyền bên phải được truyền như một đối số.*

*Now look at the assignment operator function. The first thing you should notice is that the left operand (that is, the object being assigned a value) is modified by the operation. This is in keeping with the normal meaning of assignment. The second thing to notice is that the function returns ***this**. That is, the **operator =()** function returns the object that is being assigned to. The reason for this is to allow a series of assignments to be made. As you should know, in C++, the following type of statement is syntactically correct (and, indeed, very common):*

Bây giờ chúng ta xét hàm toán tử gán. Trước hết cần chú ý rằng toán hạng bên trái (nghĩa là đối tượng được gán cho một giá trị) được thay đổi bởi phép toán. Điều này vẫn giữ nguyên ý nghĩa thông thường của phép gán. Điều thứ hai cần chú ý là hàm trả về ***this**. Nghĩa là, hàm **operator=()** trả về đối tượng sẽ được gán. Lý do này là để cho phép thực hiện một chuỗi phép gán. Như bạn biết trong C++, câu lệnh sau đây đúng về mặt ngữ pháp (và thường được sử dụng).

```
a = b =c =d =0;
```

*By returning ***this**, the overloaded assignment operator allows objects of type **coord** to be used in a similar fashion. For example, this is perfectly valid:*

Bằng cách trả về ***this**, toán tử gán được Nạp chồng cho phép các đối tượng kiểu **coord** được dùng theo cách như nhau. Ví dụ câu lệnh này hoàn toàn đúng.

```
ob = o2 = o1;
```

*Keep in mind that there is no rule that requires an overloaded assignment function to return the object that receisves the assignment. However, if you want the overloaded = to behave relative to its class the way it does for the built-in types, it must return ***this**.*

Nhớ rằng không có quy tắc nào đòi hỏi hàm gán Nạp chồng trả về đối tượng mà đối tượng này nhận phép gán. Tuy nhiên nếu bạn muốn Nạp chồng toán tử = để nó hoạt động đối với lớp theo cách như đối với các kiểu định sẵn bên trong thì nó phải trả về ***this**.

*3. It is possible to overload an operator relative to a class so that the operand on the right side is an object of a built-in type, such as integer, instead of the class for which the operator function member. For example, here the + operator is overloaded to add an integer value to a **coord** object:*

Có thể Nạp chồng một toán tử đối với một lớp để cho toán hạng bên phải là một đối tượng có kiểu định sẵn, như một số nguyên chẳng hạn, thay vì lớp mà hàm toán tử đối với nó là một thành viên. Ví dụ, đây là toán tử + được Nạp chồng để cộng một giá trị nguyên và đối tượng **coord**:

```
// Overload + for ob + int as well as ob + ob.
#include "iostream"
using namespace std;


class coord
{
      int x,y; // coordinate values;
      public:
            coord()
            {
                  x = 0;
                  y = 0;
```

```
            }
            coord (int i, int j)
            {
                  x = i;
                  y = j;
            }
            void get_xy( int &i, int &j)
            {
                  i = x;
                  j = y;
            }
            coord operator + (coord ob2); //ob + ob
            coord operator + (int i); // ob + int
};

// Overload + relative to coord class.
coord coord::operator +( coord ob2)
{
      coord temp;

      temp.x = x + ob2.x;
      temp.y = y + ob2.y;

      return temp;
}
```

```
// Overload + for ob + int
coord coord::operator +( int i)
{
      coord temp;

      temp.x = x + i;
      temp.y = y + i;
      return temp;
}

int main()
{
      coord o1(10, 10), o2(5, 3), o3;
      int x,y;

      o3 = o1 + o2; // add two objects - calls
                      //operator+(coord)
      o3.get_xy(x, y);
      cout<<"(o1 + o2) X: "<<x<<", Y: "<<y<<"\n";

      o3 = o1 +100; // add objects + int - calls //
                     operator+(int)
      o3.get_xy(x, y);
      cout<<"(o1 + 100) X: "<<x<<", Y: "<<y<<"\n";

      return 0;
}
```

It is important to remember that when you are overloading a member operator function so that an object can be used in an operation involving a built-in type, the built-in type must be on the right side of the operator. The reason for this is easy to understand. It is the object on the left that generates the call to the operator function. For instance, what happens when the compiler sees the following statement?

Điều quan trọng cần nhớ là khi Nạp chồng một hàm toán tử thành viên để cho một đối tượng có thể được dùng trong một phép toán có kiểu định sẵn, thì kiểu định sẵn phải ở bên phải của toán tử. Lý do này thật dễ hiểu: chính đối tượng ở bên trái tạo ra lời gọi cho hàm toán tử. Tuy nhiên, điều gì xảy ra khi trình biên dịch gặp câu lệnh này?

```
o3 = 19 + o1; // int + ob
```

*There is no built-in operation defined to handle the addtion of an integer to an object. The overloaded **operator+(int i)** function works only when the object is on the left. Therefore, this statement generates a compile-time error. (Soon you will see one way around this restriction).*

Không có phép toán định sẵn được định nghĩa để thực hiện phép cộng một số nguyên vào một đối tượng. Hàm Nạp chồng **operator+(int i)** chỉ hoạt động khi đối tượng ở bên phía trái. Do đó, câu lệnh này tạo ra một lỗi sai về biên dịch.

*4. You can use a reference parameter in an operator function. For example, this is a perfectly acceptable way to overload the + operator relative to the **coord** class:*

Bạn có thể dùng tham số qui chiếu trong một hàm toán tử. Ví dụ, đây là cách hoàn toàn chấp nhận được để Nạp chồng toán tử + đối với lớp **coord**.

```
coord coord::operator +(coord &ob)
{
      coord temp;
      temp.x = x + ob2.x;
      temp.y = y + ob2.y;

      return temp;
}
```

One reason for using a reference parameter in an operator function is efficiency. Passing objects as parameters to functions often incurs a large amount of overhea and consumes a significant number of CPU cycles. However, passing the address of an object is always quick ad efficient. If the operator is going to be used often, using a reference parameter will generally improve performance significantly.

Một lý do để dùng tham số quy chiếu trong hàm toán tử là sự hiệu quả. Truyền các đối tượng như các tham số cho các hàm thường chịu nhiều thủ tục bổ sung và tiêu tốn một số đáng kể các chu kỳ CPU. Tuy nhiên, truyền địa chỉ của một lớp đối tượng thường nhanh và hiệu quả. Nếu toán tử thường được sử dụng thì việc dùng tham số quy chiếu sẽ cải thiện năng suất đáng kể.

Another reason for using a reference parameter is to avoid the trouble caused when a copy of an operand is destroyed. As yo know from previous chapters, when an argument is passed by value, a copy of that argument is made. If that object has a destructor function, when the function terminates, the copy's destructor is called. In some cases it is possible for the destructor to destroy something needed by the calling object. If this is the base, using a reference parameter instead of a value parameter is an easy (and efficient) way around the problem. Of course, you could also define a copy constructor that would prevent this problem in the general case.

Một lý do khác để dùng tham số quy chiếu là tránh những rắc rối gây ra khi bản sao một toán hạng bị hủy. Như bạn biết từ những chương trước, khi một đối số được truyền bởi một giá trị thì bản sao của đối số được thực hiện. Nếu đối tượng có một hàm tạo, thì khi hàm kết thúc, hàm hủy của bản sao được gọi. Trong một số trường hợp, hàm hủy có thể hủy bỏ cái gì đó bằng đối tượng gọi. Trong trường hợp này, việc sử dụng một tham số quy chiếu thay vì một tham số giá trị sẽ là một cách dễ dàng để giải quyết bài toán. Tuy nhiên, cần nhớ rằng bạn cũng có thể định nghĩa một hàm hủy bản sao để ngăn ngừa vấn đề này trong trường hợp tổng quát.

EXERCISES(BÀI TẬP)

*Relative to **coord**, overload the * and / operators. Demonstrat that they work.*

Hãy Nạp chồng toán tử * và / đối với **coord**. Chứng tỏ chúng hoạt động.

Why would the following be an inappopriate use of an overloaded operator?

Tại sao phần dưới đây là cách sử dụng không thích hợp của một toán tử được qúa tải.

```
coord coord::operator%(coord ob)
 {

      double i;
```

```
        cout<<" Enter a number: ";
        cin>>i;
        cout<<"root of "<<i<<" is";
        cout<<sqr(i);
    }
```

*On your own, experiment by changing the return types of the operator functions to something other than **coord**. See what types of errors result.*

Bạn hãy thử thay đổi các kiểu trả về của các hàm toán tử đối với lớp khác **coord**. Xem kiểu gì có kết qủa sai.

OVERLOADING THE RELATIONAL AND LOGICAL OPERATORS - QUÁ TẢI CÁC TOÁN TỬ QUAN HỆ VÀ LUẬN LÝ

It is possible to overload the relational and logical operations. When you overload the relational and logical operators so that they behave in their traditional manner, you will not want the operator functions to return an object of the class for which they are defined. Instead, they will return an integer that indicates either true of false. This not only allows these operator functions to return a true/false value, it also allows the operators to be integrated into larger relational and logical expressions that involve other types of data.

Có thể Nạp chồng các toán tử quan hệ và luận lý. Khi bạn Nạp chồng các toán tử quan hệ và luận lý để chúng hoạt động theo cách truyền thống, bạn sẽ không cần các hàm toán tử trả về một đối tượng của lớp mà các hàm toán tử được định nghĩa đối với lớp này. Thay vì vậy, các hàm toán tử sẽ trả về một số nguyên để chỉ đúng hay sai. Điều này không chỉ cho phép các hàm toán tử trả về giá trị đúng / sai mà nó còn cho phép các toán tử được kết hợp với nhau thành một biểu thức quan hệ và luận lý lớn hơn có chứa các kiểu dữ liệu khác.

Note: *if you are using a modern C++ compiler, you can also have an overloaded relational or logical operator function return a value of type* ***bool****, although there is no advantage to doing so. As explained in Chapter 1, the* ***bool*** *type defines only two values:* ***true*** *and* ***false****. These values are automatically converted into nonzero and 0 values. Integer nonzero and 0 values are automatically converted into* ***true*** *and* ***false.***

Chú ý: Nếu bạn sử dụng trình biên dich C++ mới, bạn cũng có thể Nạp chồng hàm toán tử quan hệ hoặc luận lý và trả về giá trị của cờ **bool**, mặc dù không có lợi khi làm vậy. Như đã giải thích trong chương 1, cờ **bool** chỉ định nghĩa hai giá trị: **đúng** và **sai.** Những giá trị này được tự động chuyển thành số khác không và bằng 0. Số nguyên khác không và bằng 0 được tự động chuyển thành **đúng** và **sai.**

EXAMPLE(VÍ DỤ)

In the following program, the == and && operator are overloaded:

Trong chương trình sau, các toán tử == và && được Nạp chồng.

```
// Overload the == and relative to coord class.
#include "iostream"
using namespace std;

class coord
{
      int x,y; // coordinate values
      public:
            coord()
            {
                  x = 0;
                  y = 0;
            }
            coord (int i, int j)
            {
```

```
            x = i;
            y = j;
        }
        void get_xy(int &i, int&j)
        {
            i = x;
            j = y;
        }
        int operator ==(coord ob2);
        int operator &&(coord ob2);
};

// Overload the == operator for coord
int coord::operator==(coord ob2)
{
    return x==ob2.x && y==ob2.y;
}

// Overload the && operator for coord
int coord::operator &&(coord ob2)
{
    return (x && ob2.x) && (y && ob2.y);
}

int main()
{
    coord o1(10,10),o2(5,3),o3(10,10),o4(0,0);
```

```
        if(o1==o2)
              cout<<" o1 same as o2 \n";
        else
              cout<<" o1 and o2 differ\n";

        if(o1==o3)
              cout<<" o1 same as o3 \n";
        else
              cout<<" o1 and o3 differ\n";

        if(o1&&o2)
              cout<<" o1 && o2 is true \n";
        else
              cout<<" o1 && o2 is false\n";

        if(o1&&o4)
              cout<<" o1 && o4 is true \n";
        else
              cout<<" o1 && o4 is false\n";

        return 0;
  }
```

EXERCISE(BÀI TẬP)

*Overload the < and > operators relative to the **coord** class.*

Qúa tải các toán tử < và > đối với lớp **coord**.

OVERLOADING A UNARY OPERATOR - QUÁ TẢI TOÁN TỬ ĐƠN NGUYÊN

Overloading a unary operator is similar to overloading a binary operator except that there is only one operand to deal with. When you overload a unary operator using a member function, the function has no parameters. Since there is only one operand, it is this operand that generates the call to the operator function. There is no need for another parameter.

Nạp chồng toán tử đơn nguyên tương tự như Nạp chồng toán tử nhị nguyên ngoại trừ chỉ có một toán hạng. Khi bạn Nạp chồng một toán tử đơn nguyên bằng cách dùng hàm thành viên, thì hàm không có tham số. Vì chỉ có một toán hạng, nên chính toán hạng này tạo ra lời gọi cho hàm toán tử. Không cần có một tham số khác.

EXAMPLES(VÍ DỤ)

*The following program overloads the increment operator (++) relative to the **coord** class:*

Chương trình sau Nạp chồng toán tử tăng (++) đối với lớp **coord**:

```
// Overload ++ relative to coord class
#include "iostream"
using namespace std;

class coord
{
      int x,y; // coordinate values
      public:
            coord()
            {
```

```
            x = 0;
            y = 0;
        }
        coord (int i, int j)
        {
            x = i;
            y = j;
        }
        void get_xy(int &i, int&j)
        {
            i = x;
            j = y;
        }
        coord operator ++();
};

// Overload ++ for coord class
coord coord::operator ++()
{
    x++;
    y++;

    return *this;
}

int main()
{
```

```
        coord o1(10, 10);
        int x,y;

        ++o1; // increment an object
        o1.get_xy(x,y);
        cout<<"(++o1) X: "<<x<<", Y: "<<y<<"\n";

        return 0;
}
```

Since the increment operator is designed to increase its operand by 1, the overloaded ++ modifies the object it operates upon. The function also returns the object that it increments. This allows the increment operator to be used as part of a larger statement, such as this:

Vì toán tử tăng được tạo ra để làm tăng toán hạng lên một đơn vị nên Nạp chồng toán tử ++ sẽ làm thay đổi đối tượng mà nó tác dụng. Hàm cũng trả về đối tượng khi nó tăng. Điều này cho phép toán tử tăng được sử dụng như một phần của câu lệnh lớn hơn, chẳng hạn:

```
o2 = ++o1;
```

As with the binary operators, there is no rule that says you must overload a unary operator so that it reflects its normal meaning. However, most of the time this is what you will want to do.

Với các toán tử đơn nguyên, không có quy luật cho rằng bạn phải Nạp chồng toán tử đơn nguyên sao cho nó phản ánh ý nghĩa bình thường của nó. Tuy nhiên, đây chính là những gì bạn cần tìm.

In early versions of C++, when an increment or decrement operator was overloaded, there was no way to determine whether an overloaded ++ or – preceded or followed its operand. That is, assuming the preceding program, these two statements would have been identical:

Trong các ấn bản trước đây của C++, khi Nạp chồng một toán tử tăng hoặc giảm, không có cách để xác định toán tử Nạp chồng ++ hay – đứng trước hay sau toán hạng. Nghĩa là giả sử trong chương trình trước, hai câu lệnh này sẽ giống nhau:

```
o1++;

++o1;
```

*However, the modern specification for C++ has defined a way by which the compiler can distinguish between these two statements. To accomplish this, create two versions of the **operator++()** function. The first is defined as shown int the preceding example. The second is declared like this:*

Tuy nhiên, đặc tả mới đối với C++ đã định nghĩa cách mà nhờ đó trình biên dịch có thể phân biệt hai câu lệnh này. Để thực hiện điều này, hãy tạo ra hai dạng hàm **operator++()**. Dạng thứ nhất được định nghĩa như trong ví dụ trên đây. Dạng thứ hai được khai báo như sau:

```
coord coord::operator++(int notused);
```

*If the ++ precedes its operand, the **operator++()** function is called. However, if the ++ follows its operand, the **operator++(int notused)** function is used. In this case, **notused** will always be passed the value 0. therefore, if the difference between prefix and postfix increment or decrement is important to your class objects, you will need to implement both operator functions.*

Nếu ++ đứng trước toán hạng thì hàm **operator++()** được gọi. Tuy nhiên, nếu ++ đứng sau toán hạng thì hàm **operator++(int notused)** được sử dụng. Trong trường hợp này, **notused** sẽ luôn luôn được truyền giá trị 0. Do đó, nếu sự khác biệt giữa sự tăng (hoặc giảm) trước và sau là quan trọng đối với các đối tượng của lớp, bạn sẽ cần thực hiện cả hai toán tử.

As you know, the minus sign is both a binary and a unary operator in C++. You might be wondering how you can overload it so that it retains both of these uses relative to a class that you create. The solution is actually quite easy: you simply overload it twice, once as a binary operator and once as a unary operator. This program shows how:

Như bạn biết, dấu trừ là toán tử nhị nguyên lẫn đơn nguyên trong C++. Bạn có thể tự hỏi làm cách nào bạn có thể Nạp chồng sao cho nó vẫn giữ nguyên lại cả hai tính chất này đối với lớp do bạn tạo ra. Giải pháp thật sự hoàn toàn dễ: bạn chỉ Nạp chồng nó hai lần, một lần như toán tử nhị nguyên và một lần như toán tử đơn nguyên. Đây là chương trình chỉ rõ điều đó:

```
// Overload - relative to coord class

#include "iostream"
```

```
using namespace std;

class coord
{
      int x,y; // coordinate values
      public:
            coord()
            {
                  x = 0;
                  y = 0;
            }
            coord (int i, int j)
            {
                  x = i;
                  y = j;
            }
            void get_xy(int &i, int&j)
            {
                  i = x;
                  j = y;
            }
      coord operator -(coord ob2); // binary minus
      coord operator -();        // unary minus
};

// Overload - for coord class
coord coord::operator -(coord ob2)
```

```
{
      coord temp;

      temp.x = x - ob2.x;
      temp.y = y - ob2.y;

      return temp;
}

// Overload unary - for coord class
coord coord::operator -()
{


      x = -x;
      y = -y;

      return *this;
}

int main()
{
      coord o1(10, 10), o2(5, 7);
      int x,y;

      o1 = o1 -o2;
```

```
        o1.get_xy(x,y);
        cout<<"(o1 - o2) X: "<<x<<", Y: "<<y<<"\n";

        o1 = -o1; // negation
        o1.get_xy(x,y);
        cout<<"(-o1) X: "<<x<<", Y: "<<y<<"\n";

        return 0;
}
```

*As you can see, when the minus is overloaded as a binary operator, it takes one parameter. When it is overloaded as a unary operator, it takes no parameter. This difference in the number of parameters is what makes it possible for the minus to be overloaded for both operations. As the program indicates, when the minus sign is used as a binary operator, the **operator-(coord ob2)** function is called. When it is used as a unary minus, the **operator-()** function is called.*

Như bạn thấy, khi dấu trừ được Nạp chồng như một toán tử nhị nguyên, nó chọn một tham số. Khi nó được Nạp chồng như toán tử đơn nguyên, nó không chọn tham số. Sự khác biệt này về tham số là điều làm cho dấu trừ được Nạp chồng đối với hai phép toán. Như chương trình chỉ rõ, khi dấu trừ được dùng như toán tử nhị nguyên, hàm **operator-(coord ob)** được gọi. Khi nó được dùng như toán tử đơn nguyên, hàm **operator()** được gọi.

EXERCISES(BÀI TẬP)

*overload the - - operator for the **coord** class. Create both its prefix and postfix forms.*

Nạp chồng toán tử -- đối với lớp **coord**. Tạo cả hai dạng đứng trước và đứng sau.

*Overload the + operator for the **coord** class so that it is both a binary operator (as shown earlier) and a unary operator. When it is used as a unary operator, have the + make any negative coordinate value positive.*

Nạp chồng toán tử + đối với lớp **coord** để cho nó là toán tử nhị nguyên (như

trước đây) lẫn toán tử đơn nguyên. Khi được dùng như một toán tử đơn nguyên, hãy cho toán tử + thực hiện một giá trị tọa độ âm bất kỳ thành dương.

6.5. USING FRIEND OPERATOR FUNCTION - SỬ DỤNG HÀM TOÁN TỬ FRIEND

*As mentional at the start of this chapter, it is possible to overload an operator ralative to a class by using a **friend** rather than a member function. As you know, a friend function does not have a **this** pointer. In the case of a binary operator, this means that a friend operator function is passed both operands explictily. For unary operator , the single operand is passed. All other things being equal, there is no reason to use a friend rather than a member operator function, with one important exception, which is discussed in the examples.*

Như đã nói trong phần đầu của chương này, có thể Nạp chồng của một toán tử đối với lớp bằng cách dùng hàm **friend** hơn lá hàm thành viên member. Như bạn biết, hàm friend không có con trỏ **this**. Trong trường hợp toán tử nhị nguyên, điều này có nghĩa là một hàm toán tử friend được truyền cả hai toán hạng. Đối với các toán tử khác, không có lý do gì để sử dụng hàm toán tử friend hơn là hàm toán tử thành viên, với một ngoại lệ quan trọng, sẽ được thảo luận trong ví dụ.

Remember: *You cannot use a friend to overload the assignment operator. The assignment operator can be overloaded by a member operator function.*

Cần Nhớ:Bạn có thể dùng một hàm friend để Nạp chồng một toán tử gán. Toán tử gán chỉ được Nạp chồng bởi hàm toán tử thành viên.

EXAMPLES:(Các Ví Dụ)

*Here **operator+()** is overloaded for the **coord** class by using a friend function:*

Ở đây, **operator+()** được Nạp chồng hàm đối với lớp **coord** bằng cách dùng hàm friend.

```
//Overload the - relative to coord class using a
```

friend.

```
#include <iostream>
using namespace std;

class coord
{
     int x,y; //coordinate values
     public:
          coord()
          {
               x = 0;
               y = 0;
          }
          coord(int i, int j)
          {
               x = i;
               y = j;
          }
          void get_xy(int &i, int &j)
          {
               i = x;
               j = y;
          }
          friend coord operator+(coord ob1, coord
ob2);
};
```

```
// Overload using a friend.
coord operator+(coord ob1, coord ob2)
{
     coord temp;

     temp.x = ob1.x + ob2.x;
     temp.y = ob1.y + ob2.y;
     return temp;
}
int main()
{
     coord o1(10, 10), o2(5, 3 ), o3;
     int x,y;

     o3 = o1 + o2; // add two object - this calls
operator+()
     o3.get_xy(x, y);
     cout<< "(o1 + o2 ) X: " << x << ", Y: " << y
<<"\n"

     return 0;
}
```

Notice that the left operator os passed to the first parameter and the rigth operator is passed to the second parameter.

Chú ý rằng toán tử hạng bên trái được truyền cho tham số thứ nhất và toán hạng bên phải được truyền cho tham số thứ hai.

Overloading an operator provides one very important feature that member function do not. Using a friend operator function, you can allow object to be use in operations involing built-in types in which the built-in types is on the left side of the operator. As you saw earlier in this chapter, it is possible to overload a binary member operator function such that the left operand is an object and the right operand is a built-in types. But it is not possible to use a member function to allow the built-in type to occur on the left side of the operator. For example, assuming an overload member operator function, the first statement show here is legal; the second is not:

Việc Nạp chồng của một toán tử bằng cách dùng một hàm friend cung cấp một đặc điểm rất quan trọng mà một hàm thành viên không thể có được. Bằng cách dùng toán tử friend, bạn có thể để cho những đối tượng được sử dụng trong các phép toán tử có kiểu định sẵn, ở đó kiểu định sẵn mằn ở bên trái của toán tử. Như bạn đã thấy trong phần đầu của chương này, có thể Nạp chồng một hàm toán tử thành viên nhị nguyên để cho toán hạng bên trái là đối tượng và toán hạng bên phải là kiểu định sẵn. Nhưng không thể sử dụng hàm thành viên để cho kiểu định sẵn nằm ở bên trái toán tử. Ví dụ, giả sử một hàm toán tử thành viên được Nạp chồng, câu lệnh đầu tiên trong phần sau đây là đúng trong lúc thứ hai thì không đúng.

```
ob1 = ob2 + 10; // legal
ob1 = 10 + ob2 ; // illegal
```

While it is possible to organize such statements like the first, always having to make sure that the object is on the left side of the operand and the built-in type on the right can be a cumbersome restriction. The solution to this problem is to make the overloaded operator function friends and define both possible situations.

Trong khi có thể sắp xếp các câu lệnh như câu lệnh thứ nhất để luôn bảo đảm rằng đối tượng ở bên trái toán hạng và kiểu định sẵn ở bên phải toán hạng thì sự sắp xếp như thế gặp trở ngoại cồng kềnh. Giải pháp cho vấn đề này là hàm cho các hàm toán tử được Nạp chồng trở thành các friend.

As you know, a friend operator is explicitly passed both operands. Thus, it is possible to define one overloaded friend function so that the left operand is an object and the right operand is the other type. Then you could overload the operator again with the left operand being the built-in type and the right operand being the object. The following program illustrantes this method:

Như bạn biết , một hàm toán tử friend được truyền cả hai toán hạng. Do đó có thể định nghĩa một hàm friend Nạp chồng sao cho toán hạng bên trái là một đối tượng và toán hạng bên phải là kiểu khác. Sau đó, Nạp chồng toán tử này lần nữa với toán hạng bên trái là kiểu định sẵn và toán hạng bên phải là đối tượng. Chương trình sau minh họa phương pháp này:

```
//Use friend operator function to add flexiblity
#include <iostream>
using namespace std;

class coord
{
     int x,y; //coordinate values
     public:
          coord()
          {
               x = 0;
               y = 0;
```

```
        }
        coord(int i, int j)
        {
            x = i;
            y = j;
        }
        void get_xy(int &i, int &j)
        {
            i = x;
            j = y;
        }
        friend coord operator+(coord ob1, int
i);
        friend coord operator+(int i, coord
ob1);
};

// Overload using a friend.
coord operator+(coord ob1, int i)
{
    coord temp;

    temp.x = ob1.x + i;
    temp.y = ob1.y + i;
    return temp;
}
```

```
// Overload for int + ob1
coord operator+(int i, coord ob1)
{
     coord temp;

     temp.x = ob1.x + i;
     temp.y = ob1.y + i;
     return temp;
}
int main()
{
     coord o1(10, 10);
     int x,y;

     o1 = o1 + 10; // object + integer
     o3.get_xy(x, y);
     cout<< "(o1 + 10 ) X: " << x << ", Y: " << y
     <<"\n";

     o1 = 99 + o1; // integer + object
     o3.get_xy(x, y);
     cout<< "(99 + o1 ) X: " << x << ", Y: " << y
     <<"\n";

     return 0;
}
```

As a result of overloading friend operator function for both situation, both of these statements are now vail:

Là kết quả của việc Nạp chồng hàm toán tử friend đối với các hai trường hợp, cả hai câu lệnh này bây giờ đúng.

```
O1 = o1 + 10;

O1 = 99 + o1;
```

*If you want to use a friend operator to overload either the ++ or - - unary operator, you must pass the operand to the function as a reference parameter. This is because friend function do not have **this** pointers. Remember that the increment and decrement operators imply that the operand will be modified. However, if you overload these operators by using a friend that uses a value parameter, any modification that occur to the parameter inside the friend operator function will not affect to the object that generated the call. And sine no pointer to the object is passed implicitly (that is, there is no **this** pointer) when a friend is used, there is no way for the increment or decrement to affect the operand.*

Nếu bạn muốn dùng một hàm toán tử friend để Nạp chồng đơn nguyên ++ hoặc toán tử đơn nguyên --, bạn phải truyền toán hạng cho hàm như tham số quy chiếu. Diều này do các hàm friend không có con trỏ this. Nhớ rằng các toán tử tăng và giảm chỉ ra rằng toán hạng sẽ được thay đổi. Tuy nhiên nếu bạn Nạp chồng các toán hạng này bằng cách dùng friend thì toán hạng được truyền bởi một giá trị như tham số. Do đó, những thay đổi bất kỳ đối với tham số bên trong toán tử friend sẽ không làm ảnh hưởng đến đối tượng mà đối tượng này tạo ra lời gọi. Và vì không có con trỏ về đối tượng được truyền một cách tường minh(nghĩa là không có con trỏ this) khi một friend được dùng nên không có sự tăng hoặc giảm nào tác động đến toán hạng.

However, if you pass the operand to the friend reference parameter, changes tha occur inside the friend function affect the object that generates

the call. For example, here is a program that overloads the ++ operator by using a friend function:

Tuy nhiên, bằng cách truyền toán hạng cho friend như một tham số tham chiếu, những thay đổi xảy ra bên trong hàm friend có ảnh hưởng đến đối tượng tạo ra lời gọi. ví dụ,đây là chương trình Nạp chồng toán tử ++ bằng cách dùng một hàm friend.

```
//Overload the ++ using a friend
#include <iostream>
using namespace std;

class coord
{
     int x,y; //coordinate values
     public:
          coord()
          {
               x = 0;
               y = 0;
          }
          coord(int i, int j)
          {
               x = i;
               y = j;
          }
          void get_xy(int &i, int &j)
```

```
        {
            i = x;
            j = y;
        }
        // Overload ++ using a friend.
        friend coord operator++(coord &ob);
};

// Overload using a friend.
coord operator++(coord &ob) //Use reference
parameter
{
    ob.x++;
    ob.y++;

    return ob; //return object generating the
call
}

int main()
{
    coord o1(10, 10);
    int x,y;

    ++o1;   //o1 is passed by reference
    o1.get_xy(x, y);
```

```
     cout<< "(++o1) X: " << x << ", Y: " << y
<<"\n";


     return 0;

}
```

If you using a modern compiler, you can also distinguish between the prefix and posfix forms of the increment or decrement operators when using a friend operator in much the same way you did when using member functions. You simply add an integer parameter when defining the postfix version. For example, here are the prototypes for both the prefix and posfix versions of the increment operator relative to the ***coord*** *class:*

Nếu hàm dùng trình viên dịch mới, bạn cũng có thể phân biệt giữa các dạng đứng trước và đứng sau của các toán tăng hay giảm khi bạn dùng hàm toán tử friend như bạn đã dùng với các hàm thành viên. Bạn chỉ bổ sung tham số nguyên khi định nghĩa dạng đứng sau. Ví dụ, đây là các nguyên mẫu cho cả hai dạng đứng trước và đứng sau của toán tử tăng đối với lớp **coord**.

```
coord operator++(coord &ob);  //pradix

coord operator++(coord &ob,int notuse);
```

If the ++ precedes its operand,the ***operator++(coord &ob)*** *function is calles. However, if the ++ follows its operand, the* ***operator ++(coord &ob, int notused)*** *function is used. In this case notused will passed the value 0.*

Nếu ++ đứng trước toán hạng của nó, hàm **operator++(coord &ob)** được gọi.Tuy nhiên, nếu ++ đứng sau toán hạng của nó thì hàm **operator++(coord &ob, int notuse)** được sử dụng. Trong trường hợp này, notuse sẽ được truyền giá trị 0.

EXERCISES: (Bài Tập)

*Overload the – and / operator for the **coord** class using friend function.*

Nạp chồng các toán tử - và / đối lớp coord dùng các hàm friend.

*Overload the **coord** class so it can use **coord** object in operations in which an integer value can be multiplied by each coordinate. Allow the operations to use the either order: ob " int or int " ob.*

Nạp chồng của lớp coord để sử dụng các đối tượng coord trong các phép toán mà giá trị nguyên được nhân với mỗi tọa độ, cho phép các phép toán dùng thứ tự của nó : *ob *int* hoặc *int *ob*

Explain why the solution to Exercise 2 reqires the use of the friend operator functions?

Giải thích tại sao lời đối với bài tập 2 cần dùng các hàm toán tử friend ?

Using a friend, show how to overload the - - relative to the coord class. Define both the prefix and postfix froms.

Sử dụng friend, hãy Nạp chồng toán tử -- đối với lớp coord. Định nghĩa cả hai dạng đứng trước và đừng sau.

6.6. A CLOSER LOOK AT THE ASSIGNMENT OPERATOR - Một Cái Nhìn Về Toán Tử Gán

As you have seem, it is possible to overload the assignment operator relative to a class. By defualt, when the assignment operator is applied to an object, a bitwise copy of the object on the right os put into the object on the left. If this is what you want, there is no reason to provide your own operator=() function. However, there are cases in which a strict bitwise copy is not desirable. You saw some example of this in Chapter 3, in case in which an object allocates memory. In these types of situations, you will want to provide a special assignment operation.

Như bạn thấy, có thể Nạp chồng một toán tử gán đối với một lớp. Theo mặc định, khi một toán tử gán được áp dụng cho một đối tượng thì một bản sao từng bit được đặt vào trong đối tượng bên trái. Nếu đây là điều bạn muốn thì không có lý do gì để đưa ra hàm operator=(). Tuy nhiên, có những trường hợp mà bản sao từng bit chính xác là không cần. Bạn đã gặp một số ví dụ trong chương 3, nhưng trường hợp khi một đối tượng sử dụng bộ nhớ. Trong những trường hợp này bạn muốn có một phép gán đặc biệt.

*Here is another version of the **strtype** class that you have seen in various forms in the preceding chapters. However, this version overloads the = operator so that the pointer **p** is not overwritten by an assignment operator.*

Đây là một phiên bản của lớp strtype đã gặp dưới nhiều dạng khác nhau trong các chương trong các chương trước. Tuy nhiên, phiên bản này Nạp chồng toán tử = để cho con trỏ p không bị viết đè lên bởi một phép gán.

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
```

```
class strtype
{
   char *p;
   int len;
   public:
        strtype(char *s);
        ~strtype()
        {
             cout<< "Freeing " << (unsigned) p
   <<'/n';
             delete [] p;
        }
        char *get()
        {
             return p;
        }
        strtype &operator=(strtype &ob);
};
strtype::strtype(char *s)
{
   int l;
   l = strlen(s) + 1;
   p = new char[1];
   if(!p)
   {
        cout<< "Allocation error \n";
```

```
        exit(1);
    }
    len = 1;
    strcpy(p, s);
}
// Assign an object
strtype &strtype::operator =(strtype &ob)
{
    // see if more is needed
    if(len < ob.len)
    {
        // need to allocate more memory
        delete []p;
        p = new char (ob.len);
        cout<< " Allocation error \n";
        exit(1);
    }
    len = ob.len;
    strcpy (p, ob.p);
    return *this;
}
int main()
{
    strtype a("Hello"), b("There");
```

```
    cout<< a.get() << '\n';
    cout<< b.get() << '\n';

    a = b; // now p is not overwritten

    cout<< a.get() << '\n';
    cout<< b.get() << '\n';

    return 0;
}
```

*As you see, the overloaded assignment operator prevents **p** form being overwritten. If first checks to see if the object on the left has allocated enough memory is freed and another portion is allocated. Then the string is copied to that memory and the length is copied into **len**.*

Như bạn đã thấy, toán tử gán được Nạp chồng ngăn cho **p** không bị viết đè lên. Trước hết nó kiểm tra xem đối tượng bên trái đã cấp đủ bộ nhớ để giữ chuỗi được gán cho nó chưa. Nếu chưa, thì bộ nhớ sẽ được giải thoát và phân khác sẽ được cấp phát. Sau đó chuỗi được phép sao chép vào bộ nhớ và độ dài sao chép vào **len.**

*Notice two other important features about the **operator=()** function. First, it takes a reference parameter. This prevents a copy of object on the right side of the assignment form being made. As you know from previous chapters,when a copy is destroyed when the function terminaters. In this case, destroying the copy would call the destructor function, which would free **p**. However, this is the same **p** still needed by the object used as an argument. Using a reference parameter prevents this problem.*

Chú ý có hai đặc điểm quan trọng về hàm **operator=()**. Thứ nhất, nó có một tham số tham chiếu. Điều này ngăn ngừa sự thực hiện một bản sao của đối tượng ở bên phải của phép gán. Như bạn biết từ chương trước, khi bản sao của một đối tượng được thực hiện khi được truyền một hàm thì bản sao của nó sẽ bị hủy khi hàm kết hàm. Trong trường hợp này bản sao được gọi hàm hủy để giải phóng **p**. Tuy nhiên p vẫn còn cần thiết đối tượng dùng như một đối số. Dùng tham số tham chiếu ngăn ngừa được vấn đề này.

*The second important of the **operator=()** function is that it returns a reference parameter, not an object. The reason for this is the same as the reason it uses a reference parameter. When a function returns an object, a temporary object is created that is destroyed after the return is complete. However,this means that the temporary object's destructor will be called, causing p to be freed, but p (and the memory it points to) is still needed by the object being assigned a value. Therefore, by returning a reference, you prevent a temporary object from being created.*

Đặc điểm quan trọng thứ hai của hàm operator=() là nó trả về một tham chiếu chứ không phải đối tượng. Lý do này giống như lý do để sử dụng tham số tham chiếu. Khi một hàm trả về một đối tượng thì một đối tượng tạm thời được tạo ra và sẽ tự hủy khi sự trả về hoàn tất. Tuy nhiên điều này có nghĩa là hàm hủy của đối tượng tạm thời sẽ được gọi, giải phóng p, nhưng p (và bộ nhớ mà nó trở về) vẫn còn vẫn còn cần thiết do đối tượng được gán giá trị. Do đó, bằng cách trả về một tham chiếu, bạn có thể ngăn được một đối tượng tạm thời tạo ra.

Note: *As you learned in Chapter 5, creating a copy constructor is another way to prevent both of the problems desribed in the precding two paragraghs. But the copy constructor might not be as efficient a solution as using a reference parameter and a reference return type. This is because using a reference prevents the overhead assicitated with copying an object in either circumstance. As you can see, there are often several ways to accomplish the same end in C++. Learning to choose between them is part of becoming an excenllent C++ programmer.*

Lưu ý: Như bạn đã biết trong chương 5, việc lặp một hàm tạo sao chép là một cách khác để ngăn cả hai vấn đề được mô tả trong hai đoạn chương trình trên

đây. Nhưng hàm sao chép không phải là giải pháp có hiệu quả như sử dụng tham số tham chiếu và kiểu trả về một tham chiếu. Điều này do tham chiếu ngăn được các thủ tục bổ sung kết hợp. Như bạn có thể biết, trong C++ thường có nhiều mục đích. Biết chọn tựa cách nào chính là công việc để trở thành người lập trình giỏi.

EXERCISE: (BÀI TẬP)

Given the following class declaration, fill in all details that will create a dynamic array type. That is, allocate memory for the array, strong a pointer to this memory in p. Store the size of the array, in bytes, in size. Have put() return a reference to the specified element. Don't allow the boundaries of the array to be overrun. Also, overload the assignment operator so that the allocated memory of each array is not accidentally destroyed when one array is assigned to another. (In the next section you will see a way to improve your solution to this exercise.)

Cho khai báo lớp dưới đây, hãy thêm vào các chi tiết để tạo nên một kiểu mảng " an toàn". Cũng như vậy, hãy Nạp chồng toán tử gán để cho bộ nhớ được cấp phát của một mảng không bị hủy tình cờ. (Tham khảo chương 4 trở thành không nhớ cách tạo một mảng "an toàn")

```
class dynarray
{
     int *p;
     int size;
     public:
     dynarray(int s);// pass size of array in s
     int &put(int i);//return reference to
     element i
     int get(int i);   // return value of
```

```
        Selement i

            // create operator = () function

};
```

6.7. OVERLOADING THE [ ] SUBSCRIPT OPERATOR - QUÁ TẢI CỦA TOÁN TỬ [ ] CHỈ SỐ DƯỚI

The last operator that we will overload is the [ ] array subscripting operator. In C++, the [ ] is considered a binary operator for the purposes of overloading. The [ ] can be overloaded only by a member function. Therefore, the general form of a member ***operator [ ] ()*** *function is as show here:*

Trước toán tử chỉ số chúng ta có thể Nạp chồng của toán tử [ ] chỉ số dưới trong mảng. Trong C++, toán tử [ ] có thể coi như là toán tử nhị nhân thay cho mục đích của Nạp chồng. Toán tử [ ] có thể là Nạp chồng một hàm chức năng. Bởi vậy, tổng quan chức năng của các hàm thành viên **operator[ ]** là sự trình bầy ở đây.

type class-name::operator [](*int index*)

{

//…

}

Technically, the parameter does not have to be of type ***int*** *, but* ***operator [ ] ( )*** *functions are typically used to provide array subscripting and as an integer value is generally used.*

Kỹ thuật, tham số không phải là kiểu dữ liệu **int,** nhưng toán tử hàm operator [ ] () chức năng điển hình điển hình dùng để cung cấp mảng chỉ số dưới và một giá trị số nguyên được sử dụng.

*To understand how the [ ] operator works, assume that an object called **O** is indexed as shown here:*

Để biết toán tử [ ] operator được dùng như thế nào, giả thiết một đối tượng gọi **O** một là một chỉ số hóa thì trình bày dưới đây:

```
o[9]
```

*This index will translate into the following call to the **operator [ ] ( )** function:*

Chỉ số có thể thông qua trong lời gọi hàm toán tử operator [] ():

```
o.operator [](9)
```

*That is, the value of the expression within the subscripting operator is passed to the **operator [] ()** function in its explicit parameter. The this pointer will point to **o**, the object that generated the call.*

Nghĩa là, giá trị của biểu thức bên trong toán tử chỉ số dưới được chuyển cho hàm operator [ ]() với tham số rõ ràng. Con trỏ có thể chỉ tới **O** , đối tượng mà khi được gọi phát sinh.

EXAMPLES (Ví Dụ)

In the following program, arraytype declares an array of five integers. Its constructor function initializes each member of the array. The overload operator [ ] ( ) function returns the value of the element specified by its parameter.

Trong chương trinh sau đây, mảng được khai báo với chiều dài 5. Mảng xây dựng để chứa các con số của mảng. Hàm operaptor [ ]( ) có chức năng trả lại các giá trị phần tử được chỉ bởi các tham số của nó.

```
#include <iostream>
using namespace std;

const int SIZE = 5;

class arraytype
{
     int a[SIZE];
     int i;
     public:
          arraytype()
          {
               for(i = 0; i < SIZE; i++)
                    a[i]  = i;
          }
          int operator [](int i)
          {
```

```
                return a[i];
            }
};

int main()
{
      arraytype ob;
      int i;

      for( i = 0; i < SIZE ; i++)
            cout<< ob[i] << " ";

      return 0;
}
```

The program display the following output:

Kết quả sau khi chạy chương trình ;

```
0 1 2 3 4
```

The initialization of the array a by the constructor in this and the following programs is for the sake of illustration only. It is not required.

Sự khởi tạo của một mảng bởi việc xây dựng trong đó và chương trình sau minh họa không rõ mục đích .Nó không đúng yêu cầu.

*It is possible to design the **operator [ ] ( )** function in such a way that the **[]** can be on both the left and right sides of an assignment statement. To do this, return a reference to the element being indexed. For example, this program makes this change and illustrates its use:*

Chương trình có thể thực hiện thiết kế bởi hàm operator [ ] () trong một [] có thể cả bên trái và bên phải của bên cạnh một câu lệnh chỉ định. Ví dụ, chương trình dưới làm thay đổi và minh họa về cách dùng nó.

```
#include <iostream>
using namespace std;

const int SIZE = 5;

class arraytype
{
     int a[SIZE];
     int i;
     public:
          arraytype()
          {
               for(i = 0; i < SIZE; i++)
                    a[i]  = i;
          }
          int &operator [](int i)
          {
```

```
            return a[i];
        }
};

int main()
{
    arraytype ob;
    int i;

    for( i = 0; i < SIZE ; i++)
        cout<< ob[i] << " ";

    cout<< "\n";

    //add 10 to each element in the array
    for( i = 0;i < SIZE; i++)
        ob[i] = ob[i] + 10;  // on left of =

    for( i = 0; i < SIZE ; i++)
        cout<< ob[i] << " ";

    return 0;
}
```

This program display the following output:

Kết quả sau khi chạy chương trình :

```
0 1 2 3 4
10 11 12 13 14
```

*Because the **operator [ ] ( )** function now returns a reference to the array element indexed by i, it can be used on the side of an assignment to modify an element of the array. (Of course, it can still be used on the right side as well). As you can see, this makes object of **arraytype** act normal arrays.*

Bởi vì hàm operator []() có chức năng trả về giá trị phần tử trong mảng tại vị trí I, Nó có thể được sử dụng bên cạnh một chỉ định việc sữa đổi một phần tử của mãng.(Tất nhiên, nó thể được sử dụng bên cạnh vẫn còn đúng). Trong khi bạn có thể thấy, lúc này cấu tạo của đối tượng **arraytype** hành động bình thường của mảng.

One advantage of being able to overload the [ ] operator is that it allows a better means of implementing safe array indexing. Either in this book you saw a simplified way to implement a safe array that relied upon functions such as get() and put() to access the elements of the array. Here you will see a better way to create a safe array that utilizes an overloaded [ ] operator. Recall that a safe array that is an array that is encapsulated within a class that performs bounds checking,this approach prevents the array boundaries form being overrun. By overload the [ ] operator for such an array, you allow it to be accessed just like a regular array.

Một lợi thế của hiên thân có khả năng Nạp chồng của [] operator là nó cho phép những phương thức tốt hơn thực hiên việc chỉ số của mảng.Cái này trong sách bạn thấy một cách đơn giản hóa để thực thi một cách chắc chắc trong mảng đựa vào những chức năng của get() và put() truy cập những phần

tử trong mảng. Ở đây bạn có thể thấy tốt hơn tạo ra mảng với toán tử Nạp chồng [ ] operator. Sư gọi lại một mảng là một mảng đó khai báo trong một lớp mà thi hành giới hạn kiểm tra, cách tiếp cận này ngăn cản việc tràn mảng. Bởi toán tử Nạp chồng [] operator được dùng trong mảng, bạn cho phép nó truy cập như mảng bình thường.

To create a safe array, simply add bounds checking to the operator [ ] () function. The operator [ ] ( ) must also return a reference to the element being indexed. For example, this program adds a range check to the previous array program and proves that it works by generating a boundary error:

Để tạo ra một mảng an toàn, đơn giản là thêm vào việc kiểm tra tràn mảng bên trong hàm toán tử operator []. Hàm operator []() phải trả lại tham chiếu cho phần tử đang được chỉ số hóa. Ví dụ, chương trình này thêm một kiểm tra phạm vi vào chương trình trước và chứng minh nó làm việc bởi việc phát sinh một lỗi.

```
// A safe array example
#include <iostream>
#include < cstdlib>
using namespace std;



const int SIZE = 5;


class arraytype
{
     int a[SIZE];
     public:
```

```
        arraytype()
        {
            int i;
            for(i = 0; i < SIZE; i++)
                a[i]  = i;
        }
        int &operator [](int i);
};
// Provide range checking for arraytype
int &arraytype::operator [](int i)
{
    if( i < 0 || i < SIZE -1)
    {
        cout<< "\nIndex value of ";
        cout<< i << " is out of bounds.\n";
        exit(1);
    }
    return a[i];
}
int main()
{
    arraytype ob;
    int i;

    //this is OK
```

```
        for( i = 0; i < SIZE ; i++)
              cout<< ob[i] << " ";

        /* this generates a run-time error because
             SIZE +100 is out of range */
        ob[SIZE+ 100] = 99; //error!

        return 0;
    }
```

In this program when the statement

Trong chương trình này khi khai báo

```
Ob[SIZE + 100 ] = 99;
```

*executes, the boundary error is intercepted by **operator [ ] ( )** and the program is terminated before any damage can be done.*

chạy, lỗi độ dài thì đừng lại bởi **operator [ ]()** và chương trình được hoàn thành trước bị hỏng có thể chạy được.

Because the overloading of the [ ] operator allows you to create safe arrays that look and act just like regular arrays, they can be seamlessly integrated into your programming environment. But be careful. A safe array adds overhead that might not be acceptable in all situations. In fact, the added overhead is why C++ does not perform boundary checking on arrays in the first place. However, in applications in which you want to be sure that a boundary error does not take place, a safe array will be worth the effort.

Bởi vì hàm Nạp chồng của [ ] operator cho phép bạn tạo ra một mảng an toàn mà cái nhìn và hành động cũng như những mảng bình thường, họ có thể không có đường nối vào môi trương bên trong chương trình của bạn. Nhưng phải thận trọng. Khi thêm một phần tử vào đầu của mảng mà không chấp được trong tất cả vị trí. Thật ra, việc thêm vào đầu là tại sao C++ không thực hiện kiểm tra độ dài của mảng trong vị trí đầu tiên.Tuy nhiên, trong ứng dụng trên bạn muốn thực hiên đúng mà không có sự báo lỗi khi đặt rõ độ dài, một mảng an toàn sẽ đáng giá nỗ lực .

EXERCISES: (BÀI TẬP)

6. *Modify Example 1 in Setion 6.6 so that **strtype** overloads the [ ] operator. Have this operator return the character at the specified index. Also, allow the [ ] to be used on the left side of the assignment. Demonstrate its use.*

Hãy khai báo lại phương thức trong ví dụ 1 ở mục 6.6 **strtype** thành hàm Nạp chồng [] operator. Hãy trả về cho phương thức một kí tự. Ngoài ra, toán tử [ ] cũng được dùng như hàm phụ chỉ định. Giải thích các cách dùng hàm.

7. *Modify your answer to Exercise 1 form Section 6.6 so that it uses [ ] to index the dynamic array. That is, replace the **get()** and **put ()** functions with the [ ] operator.*

Hãy khai báo lại các phương thức trong ví dụ 1 của mục 6.6 má sử dụng toán tử [ ] để chỉ số hóa mảng động.Khi đó, thay thế sử dụng chức năng **get()** và **put()** cùng với toán tử [ ] operator.

SKILLS CHECK

KIỂM TRA KĨ NĂNG

MASTERY SKILLS CHECK (Kiểm tra kỹ năng lĩnh hội):

At this point you should be able to perform the following exercises and answer the questions.

Đến đây bạn có thể thực hiện các bài tập và trả lời những câu hỏi sau:

*Overload the >> and << shift operators relative to the **coord** class so that the following types of operations are allowed:*

Nạp chồng của các toán tử >> và << đối với lớp **coord** để cho các kiểu phép toán sau đây:

```
ob<< integer
ob>> integer
```

Make sure your operators that shift the x and y values by the amount specified.

Hãy làm cho các toán tử của bạn nâng các giá trị x và y lên một lượng được chỉ rõ.

Given the class

Cho lớp

```
class three_d
{
 int x,y,z;
 public:
     three_d(int i, int j, int x)
     {
          x = i;
          y = j;
          z = x;
     }
     three_d()
     {
          x = 0;
          y = 0;
          z = 0;
     }
     void get(int &i, int &j, int &k)
     {
          i = x;
          j = y;
          k = z;
     }
};
```

overload the +, -, ++, and - - operator for this class (For the incrememt and deccrement operator, overload only the prefix form)

hãy nạp chồng toán tử +, -, ++ và – đối với lớp này (Đối với các toán tử tăng và giảm, chỉ nạp chồng theo dạng đứng trước)

Rewrite your answer to Question 2 si that it uses reference parameters instead of value parameters to the operator functions. (Hint : you will need to use friend functions for the increment and operators.)

Viết lại phần trả lời cho bài tập 2 để nó dùng các tham số tham chiếu thay vì các tham số giá trị đối với các hàm toán tử.(Hướng dẫn : Bạn cần dùng các hàm friend cho các toán tử tăng và giảm)

How do friend operator function differ from member operator function ?

Các hàm toán tử friend khác các hàm toán tử thành viên như thế nào ?

Explain why you might need to overload the assignment operator.

Hãy giải thích tại sao bạn cần phải Nạp chồng toán tử gán.

*Can **operator=()** be a friend function?*

Operator = () có thể là một hàm friend ?

*Overload the + for the **three_d** class in Question 2 so that it accepts the*

following types of operator:

Nạp chồng toán tử + đối với lớp three_d để cho nó nhận các kiểu phép toán sau:

```
ob + int ;
int + ib;
```

*Overload the ==, !=, and || **operator** relative to the **three_d** class from Question 2.*

Nạp chồng các toán tử ==, !=, và **operator** đối với lớp **three_d**

*Explain the mian reason for overloading the **[ ]** operator.*

Giải thích lý do khai báo toán tử [ ] operator trong main .

Cumulative Skills Ckeck

Kiểm tra kỹ năng tổng hợp:

This section checks how well you have integrated materrial in this chapter with that from the preceding chapters.

Phần này kiểm tra xem khả năng của bạn kết hợp chương này với chương trước như thế nào.

*Create a **strtype** class that allows the following types of operator:*

Tạo lớp **strtype** để cho phép các kiểu toán tử sau:

▼ string concatenation using the + operator.

Ghép chuỗi bằng cách dùng toán tử +.

▼ string assignment using the = operator.

Gán chuỗi bằng cách dùng toán tử =.

▼ string comparisons using < , >, and = =.

So sanh chuỗi bằng cách dùng các toán tử, và ==.

Feel free to use fixed-length string. This is a challenging assignment, but with some thought (and experimentation), you should be able to accomplish.

Có thể dùng chuỗi có độ dài cố định. Đây là công việc có tính thử thách, nhưng với suy nghĩ (và kinh nghiệm thực hành), bạn có thể thực hiện bài kiểm tra này.

CHƯƠNG 7

INHERITANCE - TÍNH KẾ THỪA

Chapter objectives

7.1. BASE CLASS ACCESS CONTROL

Điều khiển truy cập lớp cơ sở.

7.2. USING PROTECTED MEMBER

Sử dụng các thành viên bảo vệ

7.3. CONSTRUCTOR, DESTRUCTORS, AND INHERITANCE

Hàm tạo, hàm hủy, và tính kế thừa

7.4. MULTIPLE INHERITANCE

Tính đa kế thừa

7.5. VIRTUAL BASE CLASSES

Các lớp nên cơ sở ảo

You were introduced to the concept of inheritance earlier in thisbook. Now it is expplore it more thoroughly.Inheritance is one of the three principles of OOP and, as such, it is an import feature of C++. Inheritance does more than just support the concept of hierarchical classification; in Chapter 10 you will learn how inheritance provides support for polymorphism, another principal feature of OOP.

Trong phần đầu của cuốn sách, chúng tôi đã giới thiệu khái niệm tính đa thừa. Bây giờ chúng tôi trình bày đầy đủ hơn. Tính kế thừa là một trong ba nguyên lý của việc lập trình hướng đối tượng (OOP) và là đặc điểm quan trọng của C++. Trong C++, tính đa kế thừa không chỉ hổ trợ khái niệm phân loại theo cấp bậc ,mà

trong chương 10 bạn sẽ biết tính đa kế thừa còn hỗ trợ tính đa dạng, một đặc điểm quan trọng khác của OOP.

The topics covered in this chapter include base class access control and the protected access specifier, inheriting multiple base classes, passing arguments to base class constructors, and virtual base classes.

Các chủ đề bao trùm trong chương này gồm điều kiển truy cập lớp cơ sở và chỉ định truy cập protected, kế thừa đa lớp cơ sở truyền đối số cho các hàm tạo lớp cơ sở và các lớp cơ sở ảo.

Review Skills Check

Kiểm tra kỹ năng ôn tập

Before proceeding, you should be able to correctly answer the following questions and do the exericeses.

Trước hết khi bắt đầu, bạn nên trả lời chính xác các câu hỏi và làm các bài tập sau:

When an operator is overloaded, does it lose any of its otiginal functionality?

Khi nào một toán tử được gọi Nạp chồng, toán tử có mất chức năng gốc của nó không ?

Must an operator be overloaded relative to a user-defined type, such as a class?

Một toán tử phải được Nạp chồng đối với lớp?

Can the precedence of an overloaded operator be changed? Can the number of operands be altered?

Thứ tự ưu tiên của các tử được Nạp chồng có thay đổi không? Số toán hạng có thay đổi không?

Given the following partilly completed program, fill in the needed operator functions:

Cho chương trình thực hiện một phần sau đây, hãy bổ sung các toán hạng cần thiết:

```
#include <iostream>
using namespace std;

class array
{
     int nums [10];
     public:
          array();
          void set(int n[10]);
          void show();
          array operator+(array ob2);
```

```
        array operator-(array ob2);
        int operator==(array ob2);
};
array::array()
{
   int i;
   for(i = 0; i < 10; i++)
        nums[i] =0;
}

void array::set(int *n)
{
     int i;
     for(i = 0; i < 10; i++ )
          nums[i] = n[i];
}

void array::show()
{
     int i;
     for( i = 0; i < 10; i++)
          cout<< nums[i] << " ";

     cout<< "\n";
}
```

```
//Fill in operator function


int main()
{
     array o1, o2, o3;
     int i[10]= { 1, 2, 3, 4, 5, 6, 7, 8, 9,
10};


     o1.set(i);
     o2.set(i);


     o3 = o1 + o2;
     o3.show();


     o3 = o1 - o3;
     o3.show();


     if(o1 == o2)
          cout<< " o1 equals o2 \n";
     else
          cout<< "o1 does not equals o2\n";


     if(o1 == o3)
          cout<< " o1 equals o3 \n";
     else
```

```
            cout<< "o1 does not equals o3\n";


        return 0;

    }
```

Have the overload + add each element of each operand. Have the overloaded - subtract each element of the right operand from the left. Have the overloaded == return true if each element of each operand is the same and return false otherwise.

Hãy cho toán tử Nạp chồng + cộng mỗi phần tử của toán hạng.Hãy cho toán tử - trừ phần tử toán hạng bên trái với mỗi phần tử của toán hạng bên phải. Hãy cho toán tử Nạp chồng == trả về giá trị đúng nếu mỗi phần tử của mỗi toán hạng là giống nhau và trả về giá trị sai nếu ngược lại.

Convert the solution to Exercise 4 so it overloads the operators by using friend functions.

Chuyển đổi lời giải đối với bài tập 4 để cho nó Nạp chồng các toán tử bằng cách dùng các hàm friend.

Using the class and support function from Exercise 4, overload the ++ operator by using a member function and overload the – operator by using a friend. (Overload only the prefix forms of ++ and --.)

Dùng lớp và hàm hỗ trợ từ bài tập 4, hãy Nạp chồng hàm toán tử ++ bằng cách dùng hàm thanh viên và Nạp chồng toán tử -- bằng cách dùng friend.

Can the assignment operator be overloaded by using a friend function ?

Có thể Nạp chồng toán tử gán bằng cách dùng các hàm friend không?

1.1. BASE CLASS ACCESS CONTROL – ĐIỀU KHIỂN TRUY CẬP LỚP CƠ SỞ

When one class inherits another, it uses this general form:

Khi một lớp kế thừa là một lớp khác, nó sử dụng các dạng tổng quát:

class derived-class-name:*access base-class-name*

{

//….

}

Here access is one three keywords: ***public****,* ***private****, or* ***protected.*** *A discussion of the* ***protected*** *access specifier is deferred until the next section of this chapter. The other two are discussed here.*

Ở đây access là một trong ba lớp từ khóa: **public**, **private**, or **protected.**phần thảo luận về chỉ định truy cập **protected** sẽ được trình bày trong phần tiếp theo của chương này. Hai chỉ định còn lại được trình bày ở đây.

The access dpecifier determines how element of the base class are inherited by

*the derived class. When the access specifier for the inherited base class is **public**, all public members of the base become public members of the derived class. If the access specifier is **private**, all public members of the base become private members of the derived class. In either case, any private members of the base remain private to it and are inaccessible by the derived class.*

Chỉ định truy cập xác định cách mà các phần tử của lớp cơ sở được kế thừa bởi lớp dẫn xuất. Khi chỉ định truy cập đối với lớp cơ sở được kế thừa là public, thì mọi thành viên chung của lớp cơ sở trở thành các thành viên chung của lớp dẫn xuất.Ngược lại, các thành viên riêngcủa lớp co sở vẫn còn thành viên riêng của nó và không thể truy cập lớp dẫn xuất.

*It is improtant to undertand that if the access specifier is **private**, public members of the base become private members of the derived class, but these members are still accessible by member function of the derived class.*

Điều quan trọng cần hiểu là nếu chỉ định truy cập la private, thì các thanh viên chung của lớp cơ sở thành các thành viên riêng của lớp dẫn xuất nhưng những thành viên này vẫn được truy cập bởi các hàm thành viên của lớp dẫn xuất.

*Technically, access is optional. If the specifier is not present, it is **private** by defualt if the derived **class**. If the derived class is a **struct**, **public** is the default in the absence of an explicit access specofier. Frankly, most programmers explicitly specify access for the sake of clarity.*

Về mặt kỹ thuật, thì sự truy cập phải lưa chọn. Nếu không được chỉ rõ, đó là **private** bởi lớp dẫn xuất là **class**. Nếu lớp được dẫn xuất ra một **struct, public** là sự măc định thiếu trong viêc truy xuất đến.Thực sự,một chương trình rõ ràng chỉ định sự truy xuất đến đích.

EXAMPLES: (VÍ DỤ)

Here is a short base class and a derived class that inherits it (as public):

Đây là một cơ sở ngắn và một lớp dẫn xuất kế thừa nó (như một lớp chung)

```
#include <iostream>
using namespace std;

class base
{
 int x;
 public:
     void setx(int n)
     {
          x = n;
     }
     void showx()
     {
          cout<< x << '\n';
     }

};

//Inherit as public
class derived: public base
{
 int y;
```

```
 public:
     void sety(int n)
     {
           y = n;
     }
     void showy()
     {
          cout<<  y << '\n';
     }
};

int main()
{
 derived ob;

 ob.setx(10); // access member of base class
 ob.sety(20); // access member of derived class

 ob.showx(); // access member of base class
 ob.showy();  // access member of derived class

 return 0;
}
```

*As the program illustrates, because **base** is inherited as public, the public members of **base_set()** and **showx()** – become public members of derived*

*and are, therefore accessible by any other part of the program. Specifically, they are legally called within **main().***

Như chương trình minh họa, vì base được kế thừa như một lớp chung nên các thành viê chung của **base_set()** và **showx()** trở thành các thành viên chung của derived và do đó có thể được truy cập bởi bất kỳ phần nào của chương trình. Đặc biệt, chúng được gọi một cách hợp lệ trong **main().**

*It is important to understand that just because a derived class inherist a base as public, it does not mean that the derived class has access to the base's private members. For example, this addition to **derived** from the preceding example is incorrect.*

Điều quan trọng cần biết là do một lớp dẫn xuất kế thừa một lớp cơ sở như là chung, không có nghĩa là lớp dẫn xuất truy cập được đến các thành viên riêng của lớp cơ sở. Ví dụ, phần bổ sung này đối với **derived** trong ví dụ trên đây là không đúng:

```
#include <iostream>
using namespace std;

class base
{
     int x;
     public:
          void setx(int n)
          {
               x = n;
          }
```

```
          void showx()
          {
               cout<< x << '\n';
          }

};

//Inherit as public - this has an error!
class derived: public base
{
     int y;
     public:
          void sety(int n)
          {
               y = n;
          }
        /* Cannot access private member of base
        class
        x is a private member of base and not
        available
          within derived */
          void show_sun()
          {
               cout<< x + y << '\n '; //Error !
          }
          void showy()
```

```
        {
            cout<<  y << '\n';
        }
};
```

*In this example, the **derived** class attemts to access x, which is a private member of **base**. This is an error because the private parts of c base class remain private to it no matter how it is inheried.*

Trong ví dụ này,lớp derived thử truy cập x, là thành viên riêng base. Tuy nhiên đây là lỗi sai ví phần riêng của lớp cơ sở vẫn là riêng của nó cho dù nó được kế thừa.

Here is a vartion of the program shown in Example 1; this time derived inherits base as private. This change causes the program to be in error, as indicated in the comments.

Đây là chương trình đã được nêu trong ví dụ 1 để kế thừa base như một lớp riêng. Sự thay đổi này làm cho chương trình có lỗi. Như đã được rõ trong các lời chú giải.

```
#include <iostream>
using namespace std;

class base
{
   int x;
   public:
```

```
        void setx(int n)
        {
             x = n;
        }
        void showx()
        {
             cout<< x << '\n';
        }

};

//Inherit base as private
class derived: public base
{
   int y;
   public:
        void sety(int n)
        {
              y = n;
        }
        void showy()
        {
             cout<<  y << '\n';
        }
};
```

```
int main()

{

   derived ob;


   ob.setx(10); // ERROR - now private to
   derived class

   ob.sety(20); // access member of derived
   class -Ok


   ob.showx(); // ERROR - now private to
   derived class

   ob.showy();  // access member of derived
   class


   return 0;

}
```

*As the comments in this (incorrect) program illustrate , both **showy()** and **setx()** become private to derived and are not accessible outside of it.*

Các lời chú giải trong chương trình (sai) này đã chỉ ra rằng cả hai hàm **showx()** và **setx()** đều trở thành riêng đối với derived và không được truy cập từ bên ngoài.

*Keep in mind that **showx()** and **setx()** are still public within base no mater how they are inherited by some derived class. This means that an object of type **base** could access these functions anywhere. However, relative to objects of type **derived**, they become private.*

Điều quan trọng để hiểu là **showx()** và **setx()** vẫn là chung trong base cho dù chúng được kế thừa bởi lớp dẫn xuất nào đó. Điều này có nghĩa là đối tượng có kiểu **base** có thể truy cập các hàm này bất kỳ ở đây. Tuy nhiên, đối với các đối tượng kiểu **derived**, các hàm trở thành riêng.

*As stated, even though public members of a base class become **private** members of a derived class when inherited using the private specifier, they are still accessible within the derived class. For example, here is a "fixed " version of the preceding program :*

For example. Given this fragment:

Ví dụ . trong đoạn chương trình sau:

```
base base_ob;


base_ob.setx(1); // is legal because base_ob is
of type base
```

the call **setx()** is legal because **setx()** is public within **base**.

Lời gọi đến **setx()** là hợp lệ vì **setx()** la f chung trong **base**.

Như đã nói, măc dù các thành viên chung của lớp cơ sở trở thành các thành viên riêng của lớp dẫn xuất khi được kế thừa bằng chỉ định **private**, các thành viên vẫn còn được truy cập bên trong lớp dẫn xuất. ví dụ,đây là phiên bản "cố định" của chương trình trên.

```
// This program is fixed.

#include <iostream>
```

```
using namespace std;

class base
{
 int x;
 public:
        void setx(int n)
        {
             x = n;
        }
        void showx()
        {
             cout<< x << '\n';
        }

};

//Inherit base as private
class derived: public base
{
 int y;
 public:
   // setx is accessible from within derived
   void setxy(int n, int m)
   {
```

```
        setx(n);
         y = m;
    }
    // show is accessible from within derived
    void showxy()
    {
        showx();
        cout<<  y << '\n';
    }
};

int main()
{
        derived ob;

        ob.setxy(10, 20);

        ob.showxy();

        return 0;
}
```

*In this case, the function **setx()** and **showx()** are accessed inside the derived class, which is perfectly legal because they are private members of that class.*

Trong trường hợp này , các hàm **setx()** và **showx()** được truy cập bên trong lớp dẫn xuất, điều này thì hoàn toàn hợp lệ vì chúng là thành viên riêng của lớp đó.

EXERCISES (Bài Tập)

Examine this skeleton:

Xét đoạn chương trình:

```
#include <iostream>
using namespace std;

class mybase
{
 int a,b;
 public:
     int c;
     void setab(int i,int j)
     {
          a = i;
          b = j;
     }
     void getab(int &i, int &j )
     {
          i = a;
          j = b;
```

```
    }

};

class derived1: public mybase
{
 //...
}

class derived2: private mybase
{
 //...
}

int main()
{
 derived1 ob;
 derived1 ob;
 int i,j ;

 //......
 return 0;
}
```

*Within **main(),** which of the following statements are legal ?*

Trong **mian()** ,các câu lệnh nào sau đây là hợp lệ?

```
A.o1.getab(i, j);

B.o2.getab(i, j);

C.o1.c = 10;

C.o2.c = 10;
```

What happens when a public member is inherited as public? What happens when it is inherited as private?

Điều gì xảy ra khi một thành viên riêng được kế thừa như một thành viên chung? Điều gì xảy ra khi nó được kế thừa như một thành viên riêng?

If you have not done so, try all the examples presented in this section. On your own, try various changes relative to the access specifiers and observe the results.

Hãy chạy ví dụ trong phần này,tùy theo bạn hãy thực hiện các thay đổi khác nhau đối với các chỉ định truy cập và xem các kết quả

1.2. USING PROTECTED MEMBERS - SỬ DỤNG CÁC THÀNH VIÊN ĐƯỢC BẢO VỆ

*As you know from the preceding section, a derived class does not have access to the private members of the base class. This means that if the derived class needs access to some member of the base, that member must be public. However, there will be times when you want to keep a member of a base class private but still allow a derived class access to it. To accomplish this goal, C++ includes the **protected** access specifier.*

Như bạn đã biết từ phần trên đây, một lớp dẫn xuất không truy cập được các thành viên riêng

của lớp cơ sở. Nghĩa là nếu lớp dẫn xuất muốn truy cập một thành viên nào đó của lớp cơ sở thì thành viên đó phải là chung. Tuy nhiên, sẽ có những lúc bạn muốn một thành viên của lớp cơ sở vẫn là riêng nhưng vẫn cho phép lớp dẫn xuất truy cập tới nó. Để thực hiện mục đích này, C++ có chỉ định truy cập **protected**.

*The **protected** access specifier is equivalent to the **private** specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base. Outside the base or derived classes, protected members are not accessible.*

Chỉ định truy cập **protected** tương đương với chỉ định **private** với một ngoại lệ duy nhất là các thành viên được bảo vệ (protected) của lớp cơ sở có thể được truy cập đối với các thành viên của một lớp dẫn xuất từ lớp cơ sở đó. Bên ngoài lớp cơ sở hoặc lớp dẫn xuất, các thành viên được bảo vệ không thể được truy cập.

*The **protected** access specifier can occur anywhere in the class declaration, although typically it occurs after the (default) private members are declared and before the public members. The full general form of a class declaration is shown here:*

Chỉ định truy cập **protected** nằm bất kỳ ở đâu trong khai báo lớp, mặc dù nó thường nằm sau (theo mặc định) các thành viên riêng được khai báo và trước các thành viên chung. Dạng đầy đủ tổng quát của khai báo lớp như sau:

```
class class-name
{
        //private members
        protected:
                // optional
                // protected members
        public:
                // public members
};
```

When a protected member of a base is inherited as public by the derived class, it becomes a protected member of the derived class. If the base is inherited as private, a protected member of the base becomes a private member of the derived class.

Khi một thành viên được bảo vệ của một lớp cơ sở được kế thừa bởi lớp dẫn xuất với chỉ định **public**, nó trở thành thành viên được bảo vệ của lớp dẫn xuất. Nếu lớp dẫn xuất được kế thừa với chỉ định **private**, thì một thành viên được bảo vệ của lớp cơ sở trở thành thành viên riêng của lớp dẫn xuất.

A base class can also be inherited as protected by a derived class. When this is the case, public and protected members of the base class become protected members of the derived class. (Of course, private members of the base class remain private to it and are not accessible by the derived class.)

Lớp cơ sở có thể được kế thừa với chỉ định **protected** bởi lớp dẫn xuất. Trong trường hợp này, các thành viên chung và được bảo vệ của lớp cơ sở trở thành các thành viên được bảo vệ của lớp dẫn xuất. (Dĩ nhiên các thành viên riêng của lớp cơ sở vẫn còn riêng đối với lớp cơ sở và không được truy cập bởi lớp dẫn xuất).

*The **protected** access specifier can also be used with structures.*

Chỉ thị truy cập **protected** cũng được dùng với cấu trúc và hội.

EXAMPLES(CÁC VÍ DỤ)

This program illustrates how public, private, and protected members of a class can be accessed:

Chương trình này minh họa cách mà các thành viên chung, riêng và được bảo vệ của một lớp có thể được truy cập:

```
#include <iostream>
using namespace std;

class samp
{
     // private by default
     int a;
     protected:
          // still private relative to samp
          int b;
```

```
    public:
        int c;

        samp (int n, int m)
        {
            a = n;
            b = m;
        }
        int geta()
        {
            return a;
        }
        int getb()
        {
            return b;
        }
};

int main()
{
    samp ob (10, 20);

    // ob.b = 99; Error! b is protected and thus private
    ob.c = 30; // OK, c is public
```

```
        cout << ob.geta () << ' ';
        cout << ob.getb () << ' ' << ob.c << '\n';

        return 0;
}
```

*As you can see, the commented-out line is not permissible in **main()** because b is protected and is thus still private to **samp**.*

Như bạn có thể thấy, dòng được chú giải là không được phép trong **main()** và **b** được bảo vệ và do đó vẫn còn riêng đối với **samp**.

The following program illustrates what occurs when protected members are inherited as public:

Chương trình sau minh họa điều gì xảy ra khi các thành viên được bảo vệ được kế thừa như các thành viên chung:

```
#include <iostream>
using namespace std;
class base
{
        protected:
                // private to base
                int a, b; // but still accessible by derived
        public:
                void  setab(int n, int m)
                {
```

```
                a = n;
                b = m;
            }
};

class derived:public base
{
      int c;
      public:
            void setc(int n)
            {
                  c = n;
            }
            // this function has access to a and b from base
            void showabc()
            {
                  cout << a << ' ' << b << ' ' << c << '\n';
            }
};
   int main()
{
      derived ob;
      /* a and b are not accessible here because they are
private   to both base and derived. */
      ob.setab(1, 2);
      ob.setc(3);
```

```
        ob.showabc ();

        return 0;
}
```

*Because **a** and **b** are protected in **base** and inherited as public by **derived**, they are available for use by member functions of **derived**. However, outside of these two classes, **a** and **b** are effectively private and unaccessible.*

Vì **a** và **b** được bảo vệ trong **base** và được kế thừa như thành viên chung bởi **derived**, nên chúng có hiệu lực cho sử dụng bởi các hàm thành viên của **derived**. Tuy nhiên, bên ngoài hai lớp này, **a** và **b** hoàn toàn là riêng và không thể truy cập được.

*As mentioned earlier, when a base class is inherited as protected, public and protected members of the base class become protected members of the derived class. For example, here the preceding program is changed slightly, inheriting **base** as protected instead of public:*

Như đã nói trước đây, khi một lớp cơ sở được kế thừa như **protected**, các thành viên được bảo vệ và chung của lớp cơ sở trở thành các thành viên được bảo vệ của lớp dẫn xuất. Ví dụ, sau đây là chương trình có sửa đổi lại chút ít từ chương trình trên đây:

```
// This program will not compile.
#include <iostream>
using namespace std;

class base
{
     protected:
     // private to base
     int a, b; // but still accessible by derived
```

```
     public:
          void setab(int n, int m)
          {
               a = n;
               b = m;
          }
};

class derived : protected base
{
     // inherit as protected
     int c;
     public:
          void setc (int n)
          {
               c = n;
          }
          // this function has access to a and b from
base
          void showabc()
          {
               cout << a << ' ' << b << ' '<< c << '\n';
          }
};

int main()
```

```
{
      derived ob;

      // ERROR: setab() is now a protected member of base.
      ob.setab(1, 2); // setab() is not accessible here.

      ob.setc(3);

      ob.showabc();

      return 0;
}
```

*As the comments now describe, because **base** is inherited as protected, its public and protected elements become protected members of **derived** and are therefore inaccessible within **main()**.*

Như mô tả trong các lời chú giải, do **base** được kế thừa như **protected** nên các thành viên chung và được bảo vệ của nó trở thành các thành viên được bảo vệ của **derived** và do đó không thể truy cập được trong **main()**.

EXERCISES(BÀI TẬP)

What happens when a protected member is inherited as public? What happens when it is inherited as private? What happens when it is inherited as protected?

Điều gì xảy ra khi một thành viên được bảo vệ lại được kế thừa như thành viên chung? Điều gì xảy ra khi nó được kế thừa như một thành viên riêng? Điều gì xảy ra khi nó được kế thừa như một thành viên được bảo vệ?

Explain why the protected category is needed?

Giải thích tại sao cần đến phạm trù được bảo vệ?

*In Exercise 1 from Section 7.1, if the **a** and **b** inside **myclass** were made into protected instead of private (by default) members, would any of your answers to that exercise change? If so, how?*

Trong bài tập 1 của phần 7.1, nếu **a** và **b** bên trong **myclass** được thực hiện trở thành những thành viên được bảo vệ thay vì các thành viên riêng (theo mặc định) thì phần giải đáp của bạn có thay đổi không? Tại sao?

1.3. CONSTRUCTORS, DESTRUCTORS, AND INHERITANCE - HÀM TẠO, HÀM HỦY VÀ TÍNH KẾ THỪA

It is possible for the base class, the derived class, or both to have constructor and or destructor functions. Several issues that relate to these situations are examined in this section.

Lớp cơ sở, lớp dẫn xuất hoặc cả hai có thể có các hàm tạo và/hoặc hàm hủy. Nhiều vấn đề có liên quan đến các trường hợp này được khảo sát trong phần này.

When a base class and a derived class both have constructor and destructor functions, the constructor functions are executed in order of derivation. The destructor functions are executed in reverse order. That is, the base class constructor is executed before the constructor in the derived class. The reverse is true for destructor functions: the derived class's destructor is executed before the base class's destructor.

Khi cả lớp cơ sở lẫn lớp dẫn xuất có các hàm hủy và tạo, các hàm tạo được thi hành theo thứ tự dẫn xuất. Các hàm hủy được thi hành theo thứ tự ngược lại. Nghĩa là, hàm tạo của lớp cở sở được thi hành trước hàm tạo của lớp dẫn xuất. Điều ngược lại thì đúng cho các hàm hủy: hàm hủy của lớp dẫn xuất được thi hành trước hàm hủy của lớp cơ sở.

If you think about it, it makes sense that constructor functions are executed in order of derivation. Because a base class has no knowledge of any derived class, any initialization it performs is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first.

Nếu bạn nghĩ về điều này, các hàm tạo được thi hành theo thứ tự dẫn xuất. Bởi vì lớp cơ sở không nhận biết lớp dẫn xuất, bất kỳ khởi đầu nào do lớp cơ sở thực hiện là khác biệt với khởi đầu do lớp dẫn xuất thực hiện. Do đó hàm tạo của lớp cơ sở phải được thực hiện trước.

On the other hand, a derived class's destructor must be executed before the destructor of the base class because the base class underlies the derived class. If the base class's destructor were executed first, it would imply the destruction of the derived class. Thus, the derived class's destructor must be called before the object goes out of existence.

Ngược lại, hàm hủy của lớp dẫn xuất phải được thi hành trước hàm hủy của lớp cơ sở bởi vì lớp cở sở nằm dưới lớp dẫn xuất. Nếu hàm hủy của lớp cơ sở được thi hành trước thì sẽ đến lớp dẫn xuất bị hủy. Do đó hàm hủy của lớp dẫn xuất phải được gọi trước khi đối tượng không còn tồn tại.

So far, none of the preceding examples have passed arguments to either a derived or base class constructor. However, it is possible to do this. When only the derived class takes an initialization, arguments are passed to the derived class's constructor in the normal fashion. However, if you need to pass an argument to the constructor of the base class, a little more effort is needed. To accomplish this, a chain of argument passing is established. First, all necessary arguments to both the base class and the derived class are passed to the derived class's constructor. Using an expanded form of the derived class's constructor declaration, you then pass the

appropriate arguments along to the base class. The syntax for passing along an argument from the derived class to the base class is shown here:

Không có ví dụ nào trên đây truyền đối số cho hoặc hàm tạo của lớp dẫn xuất hoặc hàm tạo của lớp cơ sở. Tuy nhiên, có thể thực hiện được điều này. Khi chỉ có lớp dẫn xuất thực hiện sự khởi đầu, đối số được truyền cho hàm tạo của lớp dẫn xuất theo cách bình thường. Tuy nhiên, nếu cần truyền đối số cho hàm tạo của lớp cơ sở thì phải thực hiện khác đi một ít. Để làm điều này thì cần lập một chuỗi truyền đối số. Trước hết, tất cả các đối số cần thiết cho cả lớp cơ sở lẫn lớp dẫn xuất được truyền cho hàm tạo của lớp dẫn xuất. Sử dụng dạng mở rộng của khai báo hàm tạo của lớp dẫn xuất, các đối số thích hợp sẽ được truyền cho lớp cơ sở. Cú pháp để truyền đối số từ lớp dẫn xuất đến lớp cơ sở như sau:

```
derived-constructor (arg-list): base (arg-list)
    {
        // body of derived class constructor
    }
```

Here base is the name of the base class. It is permissible for both the derived class and the base class to use the same argument. It is also possible for the derived class to ignore all arguments and just pass them along to the base.

Ở đây *base* là tên của lớp cơ sở. Cả lớp dẫn xuất lẫn lớp cơ sở được phép sử dụng đối số giống nhau. Lớp dẫn xuất cũng có thể bỏ qua mọi đối số và truyền chúng cho lớp cơ sở.

EXAMPLES(CÁC VÍ DỤ)

Here is a very short program that illustrates when base class and derived class constructor and destructor functions are executed:

Đây là một chương trình rất ngắn minh họa khi nào thì các hàm tạo và hàm hủy của lớp cơ sở và lớp dẫn xuất được thi hành:

```
#include <iostream>
using namespace std;

class base
{
     public:
          base ()
          {
               cout << "Constructing base class \n";
          }
          ~base ()
          {
               cout << "Destructing base class \n";
          }
};

class derived : public base
{
     public:
          derived ()
          {
               cout << "Constructing derived class \n";
          }
          ~derived ()
```

```
        {
            cout << "Destructing derived class \n";
        }
};

int main()
{
    derived o;

    return 0;
}
```

This program displays the following output:

Chương trình này hiển thị kết quả sau:

```
Constructing base class
Constructing derived class
Destructing derived class
Destructing base class
```

As you can see, the constructors are executed in order of derivation and the destructors are executed in reverse order.

Như bạn có thể thấy các hàm tạo được thi hành theo thứ tự dẫn xuất và các hàm hủy được thi hành theo thứ tự ngược lại.

This program shows how to pass an argument to a derived class's constructor:

Chương trình sau cho biết cách truyền một đối số cho hàm tạo của lớp dẫn xuất:

```
#include <iostream>
using namespace std;

class base
{
     public:
          base()
          {
               cout << "Constructing base class \n";
          }
          ~base()
          {
               cout << "Destructing base class \n";
          }
};

class derived : public base
{
     int j;
     public:
```

```
        derived (int n)
        {
            cout << "Constructing derived class \n";
            j = n;
        }
        ~derived ()
        {
            cout << "Destructing derived class \n";
        }
        void showj()
        {
            cout << j << '\n';
        }
};

int main()
{
    derived o(10);

    o.showj();

    return 0;
}
```

Notice that the argument is passed to the derived class's constructor in the normal fashion.

Chú ý rằng đối số được truyền cho hàm tạo của lớp dẫn xuất theo cách bình thường.

In the following example, both the derived class and the base class constructors take arguments. It this specific class, both use the same argument, and the derived class simply passes along the argument to the base.

Trong ví dụ sau, hàm tạo của lớp dẫn xuất lẫn của lớp cơ sở nhận một đối số. Trong trường hợp cụ thể này, cả hai sử dụng đối số giống nhau, và lớp dẫn xuất chỉ truyền đối số cho lớp cơ sở.

```
#include <iostream>
using namespace std;

class base
{
     int i;
     public:
          base (int n)
          {
               cout << "Constructing base class \n";
               i = n;
          }
          ~base()
          {
               cout << "Destructing base class \n";
          }
```

```
            void showi()
            {
                  cout << i << '\n';
            }
};
class derived : public base
{
      int j;
      public:
            derived (int n) : base(n)
            {
                  // pass arg to base class
                  cout << "Constructing derived class \n";
                  j = n;
            }
            ~derived ()
            {
                  cout << "Destructing derived class \n";
            }
            void showj()
            {
                  cout << j << '\n';
            }
};
```

```
int main()
{
      derived o(10);

      o.showi();
      o.showj();

      return 0;
}
```

*Pay special attention to the declaration of **derived**'s constructor. Notice how the parameter **n** (which receives the initialization argument) is both used by **derived()** and passed to **base()**.*

Chú ý đến khai báo hàm tạo của **derived**. Chú ý cách tham số **n** (nhận đối số khởi đầu) được dùng bởi **derived()** và được truyền cho **base()**.

In most cases, the constructor functions for the base and derived classes will not use the same argument. When this is the case and you need to pass one or more arguments to each, you must pass to the derived class's constructor all arguments needed by both the derived class and the base class. Then the derived class simply passes along to the base those arguments required by it. For example, this program shows how to pass an argument to the derived class's constructor and another one to the base class:

Trong hầu hết các trường hợp, các hàm tạo đối với lớp cơ sở và lớp dẫn xuất sẽ *không* dùng đối số giống nhau. Khi trong trường hợp bạn cần truyền một hay nhiều đối số cho mỗi lớp, bạn phải truyền cho hàm tạo của lớp dẫn xuất *tất cả* các đối số mà *cả hai* lớp dẫn xuất và cơ sở cần đến. Sau đó lớp dẫn xuất chỉ truyền cho lớp cơ sở những đối số nào mà lớp cơ sở cần. Ví dụ, chương trình này chỉ ra cách truyền một

đối số cho hàm tạo của lớp dẫn xuất và một đối số khác cho lớp cơ sở.

```
#include <iostream>
using namespace std;

class base
{
     int i;
     public:
          base (int n)
          {
               cout << "Constructing base class \n";
               i = n;
          }
          ~base()
          {
               cout << "Destructing base class \n";
          }
          void showi()
          {
               cout << i << '\n';
          }
};

class derived : public base
```

```
{
     int j;
     public:
          derived (int n, int m) : base(m)
          {
               // pass arg to base class
               cout << "Constructing derived class \n";
               j = n;
          }
          ~derived ()
          {
               cout << "Destructing derived class \n";
          }
          void showj()
          {
               cout << j << '\n';
          }
};

int main()
{
     derived o(10, 20);

     o.showi();
     o.showj();
```

```
        return 0;
}
```

*It is not necessary for the derived class's constructor to actually use an argument in order to pass one to the base class. If the derived class does not need an argument, it ignores the argument and simply passes it along. For example, in this fragment, parameter **n** is not used by **derived()**. Instead, it is simply passed to **base()**:*

Điều quan trọng cần hiểu là đối với hàm tạo của lớp dẫn xuất không cần phải nhận một số để truyền cho lớp cơ sở. Nếu lớp dẫn xuất không cần đối số, nó bỏ qua đối số và chỉ truyền cho lớp cơ sở. Ví dụ, trong đoạn chương trình sau, tham số **n** không được dùng bởi **derived()**. Thay vì vậy, nó chỉ truyền cho **base()**:

```
class base
{
        int i;
        public:
                base (int n)
                {
                        cout << "Constructing base class \n";
                        i = n;
                }
                ~base()
                {
                        cout << "Destructing base class \n";
                }
```

```
        void showi()
        {
            cout << i << '\n';
        }
};

class derived : public base
{
    int j;
    public:
        derived (int n) : base(n)
        {
            // pass arg to base class
            cout << "Constructing derived class \n";
            j = 0; // n not used here
        }
        ~derived ()
        {
            cout << "Destructing derived class \n";
        }
        void showj()
        {
            cout << j << '\n';
        }
};
```

EXERCISES(BÀI TẬP)

*Given the following skeleton, fill in the constructor function for **myderived**. Have it pass along a pointer to an initialization string to **mybase**. Also, have **myderived()** initialize **len** to the length of the string.*

Cho chương trình sau, hãy bổ sung hàm tạo cho **myderived**. Hãy cho **myderived** truyền cho **mybase** một con trỏ về một chuỗi khởi đầu. Cũng vậy, hãy cho **myderived()** khởi đầu **len** với đồ dài của chuỗi.

```
#include <iostream>
#include <cstring>
using namespace std;

class mybase
{
     char str [80];
     public:
          mybase (char *s)
          {
               strcpy(str, s);
          }
          char *get()
          {
               return str;
```

```
        }
};

class myderived : public mybase
{
     int len;
     public:
          // add myderived() here
          int getlen()
          {
               return len;
          }
          void show()
          {
               cout << get() << '\n';
          }
};

int main()
{
     myderived ob("hello");

     ob.show();
     cout << ob.getlen() << '\n';
```

```
        return 0;
}
```

*Using the following skeleton, create appropriate **car()** and **truck()** constructor functions. Have each pass along appropriate arguments to **vehicle**. In addition, have **car()** initialize **passengers** as specified when an object is created. Have **truck()** initialize **loadlimit** as specified when an object is created.*

Sử dụng dàn chương trình sau để lập các hàm tạo **car()** và **truck()** thích hợp. Mỗi hàm truyền các đối số thích hợp cho **vehicle**. Ngoài ra, cho **car()** khởi đầu **passengers** như được chỉ rõ khi đối tượng được tạo ra. Cho **truck()** khởi đầu **loadlimit** như được chỉ rõ khi một đối tượng được tạo ra.

```
#include <iostream>
using namespace std;

// A base class for various types of vehicles.
class vehicle
{
     int num_wheels;
     int range;
     public:
          vehicle (int w, int r)
          {
               num_wheels = w;
               range = r;
          }
```

```
          void showv()
          {
               cout << "Wheels: " << num_wheels << '\n';
               cout << "Range: " << range << '\n';
          }
};

class car : public vehicle
{
     int passengers;
     public:
          // insert car() constructor here
          void show()
          {
               showv();
               cout << "Passengers: " << passengers <<
'\n';
          }
};

class truck : public vehicle
{
     int loadlimit;
     public:
          // insert truck() constructor here
          void show()
```

```
            {
                showv();
                cout << "loadlimit" << Loadlimit << '\n';
            }
};

int main()
{
     car c(5,4,500);
     truck t(30000, 12, 1200);

     cout << "Car: \n";
     c.show();
     cout << "\n Truck: \n";
     t.show();

     return 0;
}
```

*Have **car()** and **truck()** declare objects like this:*

Cho **car()** và **truck()** khai báo các đối tượng như thế này:

```
car ob(passengers, wheels, range);
truck ob(loadlimit, wheels, range);
```

1.4. MULTIPLE INHERITANCE - TÍNH ĐA KẾ THỪA

There are two ways that a derived class can inherit more than one base class. First, a derived class can be used as a base class for another derived class, creating a multilevel class hierarchy. In this case, the original base class is said to be an indirect base class of the second derived class. (Keep in mind that any class-no matter how it is created-can be used as a base class.) Second, a derived class can more base classes are combined to help create the derived class. There are several issues that arise when multiple base classes are involved, and these issues are examined in this section.

Có hai cách để một lớp dẫn xuất kế thừa hơn một lớp. Thứ nhất, lớp dẫn xuất được dùng như một lớp cơ sở cho một lớp dẫn xuất khác, tạo ra thứ bậc lớp nhiều cấp. Trong trường hợp này, cấp cơ sở gốc được gọi là lớp cơ sở *gián tiếp* của lớp dẫn xuất thứ hai. (Nhớ rằng, bất kỳ lớp nào-cho dù được tạo ra-có thể được dùng như một lớp cơ sở). Thứ hai, lớp dẫn xuất có thể kế thừa trực tiếp hơn một lớp cơ sở. Trong trường hợp này, hai hay nhiều lớp cơ sở được kết hợp để ra tạo ra lớp dẫn xuất. Có nhiều vấn đề nảy sinh khi nhiều lớp cơ sở được tính đến, và những vấn đề này được xét đến trong phần này.

When a base class is used to derive a class that is used as a base class for another derived class, the constructor function of all three classes are called in order of derivation. (This is a generalization of the principle you learned earlier in this chapter.) Also, destructor functions are called in reverse order. Thus, if class B1 is inherited by D1, and D1 is inherited by D2, B1's constructor is called first, followed by D1's, followed by D2's. The destructors are called in reverse order.

Khi một lớp cơ sở được dùng để dẫn ra một lớp mà lớp này lại được dùng làm lớp cơ sở cho một lớp dẫn xuất khác thì các hàm tạo của cả ba lớp được gọi theo thứ tự dẫn xuất. (Đây là sự mở rộng của nguyên lý mà bạn đã biết trước đây trong chương trình này). Cũng vậy, tất cả các hàm hủy được gọi ra theo thứ tự ngược lại. Do đó, nếu lớp B1 được kế thừa bởi lớp D1 và D1 được kế thừa bởi lớp D2, thì hàm tạo của B1 được gọi đầu tiên, rồi đến hàm tạo của D1, tiếp đến là hàm tạo của D2. Hàm hủy được gọi theo thứ tự ngược lại.

When a derived class directly inherits multiple base classes, it uses this expanded declaration:

Khi một lớp dẫn xuất kế thừa trực tiếp nhiều lớp cơ sở, nó dùng cách khai báo mở rộng sau:

```
   class derived-class-name : access base1, access base2,
…, access baseN
     {
          // … body of class
     }
```

Here base1 through baseN are the base class names and access is the access specifier, which can be different for each base class. When multiple base classes are inherited, constructors are executed in the order, left to right, that the base classes are specified. Destructors are executed in the opposite order

Ở đây, *base1* đến *baseN* là tên các lớp cơ sở và *access* là chỉ định truy cập có thể khác nhau đối với mỗi lớp. Khi nhiều lớp cơ sở được kế thừa, các hàm tạo được thi hành theo thứ tự từ trái qua phải mà các lớp cở sở đã được chỉ rõ. Các hàm hủy được thi hành theo thứ tự ngược lại.

When a class inherits multiple base classes that have constructors that require arguments, the derived class passes the necessary arguments to them by using this expanded form of the derived class' constructor function:

Khi một lớp kế thừa nhiều lớp cơ sở có các hàm cấu tạo cần nhiều đối số, lớp dẫn xuất sẽ truyền các đối số cần thiết cho các hàm cấu tạo này bằng cách dùng dạng mở rộng của hàm cấu tạo của lớp dẫn xuất:

```
     derived-constructor (arg-list) : base1 (arg-list),
base2(arg-list), …, baseN (arg-list)
     {
          // body of derived class constructor
     }
```

Here base1 though baseN are the names of the base classes.

Ở đây, *base1* đến *baseN* là tên các lớp cơ sở.

When a derived class inherits a hierarchy of classes, each derived class in the chain must pass back to its preceding base any arguments it needs.

Khi một lớp dẫn xuất kế thừa một hệ thống thứ bậc các lớp, mỗi lớp dẫn xuất trong chuỗi các lớp này phải truyền cho lớp cơ sở đứng trước các tham số mà lớp dẫn xuất này cần.

EXAMPLES(CÁC VÍ DỤ)

*Here is an example of a derived class that inherits a class derived from another class. Notice how arguments are passed along the chain from **D2** to **B1**.*

Đây là ví dụ về lớp dẫn xuất kế thừa một lớp dẫn xuất khác của một lớp khác. Chú ý cách các đối số được truyền theo chuỗi từ **D2** đến **B1**.

```
// Multiple Inheritance
#include <iostream>
using namespace std;

class B1
{
     int a;
     public:
```

```
          B1(int x)
          {
               a = x;
          }
          int geta()
          {
               return a;
          }
};

// Inherit direct base class.
class D1 : public B1
{
     int b;
     public:
          D1(int x, int y) : B1(y) // pass y to B1
          {
               b = x;
          }
          int getb()
          {
               return b;
          }
};
```

```
// Inherit a derived class and an indirect base.
class D2 : public D1
{
     int c;
     public:
          D2(int x, int y, int z) : D1(y, z) // pass args
to D1
          {
               c = x;
          }
          /* Because bases inherited as public, D2 has
access to              public elements of both B1 and D1.
*/
          void show()
          {
               cout << geta() << ' ' << getb() << ' ';
               cout << c << '\n';
          }
};

int main()
{
     D2 ob(1, 2, 3);

     ob.show();
     // geta() and getb() are still public here
```

```
    cout << ob.geta() << ' ' << ob.getb << '\n';

    return 0;
}
```

*The call to **ob.show()** displays **3 2 1**. In this example, **B1** is an indirect base class of **D2**. Notice that **D2** has access to the public members of both **D1** and **B1**. As you should remember, when public members of a base class are inherited as public, they become public members of the derived class. Therefore, when **D1** inherits **B1**, **geta()** becomes a public member of **D1**, which becomes a public member of **D2**.*

Lời gọi đối với **ob.show()** hiển thị **321**. Trong ví dụ này **B1** là lớp cơ sở gián tiếp của **D2**. Chú ý rằng **D2** truy cập đến các thành viên chung của **D1** và **B1**. Bạn nhớ lại, khi các thành viên chung của một lớp cơ sở được kế thừa, chúng trở thành các thành viên chung của lớp dẫn xuất. Do đó, khi **D1** kế thừa **B1**, **geta()** trở thành thành viên chung của **D1**, và trở thành thành viên chung của **D2**.

As the program illustrates, each class in a class hierarchy must pass all arguments required by each preceding base class. Failure to do so will generate a compile-time error.

Như chương trình minh họa, mỗi lớp ở trong một hệ thống thứ bậc lớp phải truyền tất cả mọi đối số mà lớp cơ sở trước cần đến. Sai sót khi thực hiện sẽ sinh ra lỗi thời gian biên dịch.

The class hierarchy created in this program is illustrated here:

Hệ thống thứ bậc lớp tạo ra trong chương trình này được minh hoạ như sau:

B1

D1

D2

Before we move on, a short discussion about how to draw C++-style inheritance graphs is in order. In the preceding graph, notice that the arrows point up instead of down. Traditionally, C++ programmers usually draw inheritance charts as directed graphs in which the arrow points from the derived class to the base class. While newcomers sometimes find this approach counter-intuitive, it is nevertheless the way inheritance charts are usually depicted in C++.

Trước khi chúng ta tiếp tục, một thảo luận ngắn để làm cách nào dựng nên biểu đồ kế thừa kiểu C++ một cách có thứ tự. Trong biểu đồ có trước , chú ý rằng mũi tên đi lên thay vì đi xuống. Theo truyền thống, những người lập trình C++ thường vẽ những biểu đồ kế thừa theo định hướng với những mũi tên từ lớp được dẫn xuất ra đến lớp cơ sở. Trong khi những người mới đến đôi khi tìm thấy cách tiếp cận này trái với mong đợi, tuy vậy đó là cách mà những biểu đồ kế thừa thường được miêu tả trong C++.

Here is a reworked version of the preceding program, in which a derived class directly inherits two base classes:

Đây là phiên bản được viết lại của chương trình trước trong đó lớp dẫn xuất kế thừa trực tiếp hai lớp cơ sở.

```
#include <iostream>
```

```
using namespace std;

// Create first base class.
class B1
{
     int a;
     public:
          B1(int x)
          {
               a = x;
          }
          int geta()
          {
               return a;
          }
};

// Create second base class.
class B2
{
     int b;
     public:
          B2(int x)
          {
               b = x;
```

```
            }
            int getb()
            {
                 return b;
            }
};
// Directly inherit two base classes.

class D : public B1, public B2
{
     int c;
     public:
    // here, z and y are passed directly to B1 and B2
            D(int x, int y, int z) : B1(z), B2(y)
            {
                 c = x;
            }
/* Because bases inherited as public, D has access to public
   elements of both B1 and B2. */
            void show()
            {
                 cout << geta() << ' ' << getb() << ' ';
                 cout << c << '\n';
            }
};
```

```
int main()
{
      D ob(1, 2, 3);

      ob.show();

      return 0;
}
```

In this version, the arguments to **B1** and **B2** are passed individually to these classes by **D**. This program creates a class that looks like this:

Trong phiên bản này, các đối tượng được truyền riêng lẻ cho các lớp **B1** và **B2** bởi **D**. Chương trình này tạo ra một lớp mà trông như thế này:

B1 B2

D1

The following program illustrates the order in which constructor and destructor functions are called when a derived class directly inherits multiple base classes:

Chương trình sau đây minh họa thứ tự gọi các hàm tạo và hàm hủy khi lớp dẫn xuất kế thừa trực tiếp nhiều lớp cơ sở:

```
#include <iostream>
```

```
using namespace std;

class B1
{
     public:
          B1()
          {
               cout << "Constructing B1 \n";
          }
          ~B1()
          {
               cout << "Destructing B1 \n";
          }
};

class B2
{
     int b;
     public:
          B2()
          {
               cout << "Constructing B2 \n";
          }
          ~B2()
          {
```

```
                cout << "Destructing B2 \n";
            }
};

// Inherit two base classes.
class D : public B1, public B2
{
     public:
          D()
          {
               cout << "Constructing D \n";
          }
          ~D()
          {
               cout << "Destructing D \n";
          }
};

int main()
{
     D ob;

     return 0;
}
```

This program displays the following:

Chương trình này hiển thị:

```
Constructing B1
Constructing B2
Constructing D
Destructing D
Destructing B2
Destructing B1
```

As you have learned, when multiple direct base classes are inherited, constructors are called in order, left to right, as specified in the inheritance list. Destructors are called in reverse order.

Như bạn đã biết, khi nhiều lớp cơ sở trực tiếp được kế thừa, các hàm tạo được gọi theo thứ tự từ trái qua phải, như được chỉ rõ trong danh sách kế thừa. Các hàm hủy được gọi theo thứ tự ngược lại.

EXERCISES(BÀI TẬP)

What does the following program display? (Try to determine this without actually running the program.)

Chương trình sau hiển thị? (Thử xác định mà không chạy chương trình)

```
#include <iostream>
```

```
using namespace std;

class A
{
     public:
          A()
          {
               cout << "Constructing A\n";
          }
          ~A()
          {
               cout << "Destructing A\n";
          }
};

class B
{
     public:
          B()
          {
               cout << "Constructing B\n";
          }
          ~B()
          {
               cout << "Destructing B\n";
```

```
        }
};

class C : public A, public B
{
    public:
        C()
        {
            cout << "Constructing C\n";
        }
        ~C()
        {
            cout << "Destructing C\n";
        }
};

int main()
{
    C ob;

    return 0;
}
```

*Using the following class hierarchy, create **C**'s constructor so that it initializes **k** and passes on arguments to **A()** and **B()**.*

Sử dụng hệ thống thứ bậc lớp sau đây, hãy lập hàm tạo của lớp **C** để cho nó khởi đầu **k** và truyền trên các đối số cho **A()** và **B()**.

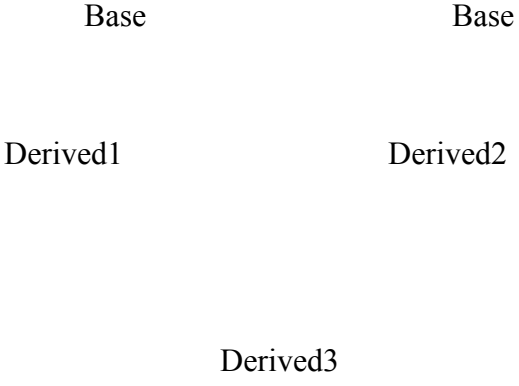
```
#include <iostream>
using namespace std;

class A
{
     int i;
     public:
          A(int a)
     {
          i = a;
     }
};

class B
{
     int j;
     public:
          B(int a)
          {
               j = a;
          }
};
```

```
class C : public A, public B
{
     int k;
     public:
/* Create C() so that it initializes k and passes arguments
   to both A() and B() */
};
```

1.5. VIRTUAL BASE CLASSES - CÁC LỚP CƠ SỞ ẢO

A potential problem exists when multiple base classes are directly inherited by a derived class. To understand what this problem is, consider the following class hierarchy:

Có một vấn đề tồn tại khi nhiều lớp cơ sở được kế thừa trực tiếp bởi một lớp dẫn xuất. Để hiểu vấn đề này, hãy xét hệ thống thứ bậc lớp sau:



Here the base class Base is inherited by both Derived1 and Derived2. Derived3 directly inherits both Derived1 and Derived2. However, this implies that Base is actually inherited twice by Derived3-first it is inherited through Derived1, and then again through Derived2. This causes ambiguity when a member of Base if used by Derived3. Since two copies of Base are included in Derived3, is a reference to a

member of Base referring to the Base inherited indirectly through Derived1 or to the Base inherited indirectly through Derived2? To resolve this ambiguity, C++ includes a mechanism by which only once copy of Base will be included in Derived3. This feature is called a virtual base class.

Ở đây, lớp *Base* được kế thừa bởi hai lớp *Derived1* và *Derived2*. *Derived3* kế thừa trực tiếp cả hai *Derived1* và *Derived2*. Tuy nhiên điều này chỉ ra rằng thực sự *Base* được kế thừa hai lần bởi *Derived3* – lần thứ nhất nó được kế thừa thông qua *Derived1*, và lần thứ hai được kế thừa thông qua *Derived2*. Bởi vì có hai bản sao của *Base* có trong *Derived3*, nên một tham chiếu đến một thành viên của **Base** sẽ tham chiếu về *Base* được kế thừa gián tiếp thông qua *Derived1* hay tham chiếu về *Base* được kế thừa gián tiếp thông qua *Derived2*? Để giải thích tính không rõ rang này, C++ có một cơ chế mà nhờ đó chỉ có một bản sao của *Base* ở trong *Derived3*. Đặc điểm này được gọi là *lớp cơ sở ảo* (*virtual base class*).

In situations like the one just described, in which a derived class indirectly inherits the same base class more than once, it is possible to prevent two copies of the base from being present in the derived object by having that base class inherited as virtual by any derived classes. Doing this prevents two (of more) copies of the base from being present in any subsequent derived class that inherits the base class indirectly. The virtual keyword precedes the base class access specifier when it is inherited by a derived class.

Trong những trường hợp như vừa mô tả trong đó một lớp dẫn xuất kế thừa gián tiếp cùng một lớp cơ sở hơn một lần thì nó có thể ngăn chặn được 2 bản sao của lớp cơ sở cùng hiện diện trong đối tượng dẫn xuất bằng cách cho lớp cơ sở đó được kế thừa như **virtual** bởi bất kỳ các lớp dẫn xuất nào. Thực hiện việc này sẽ ngăn chặn được 2 bản sao của lớp cơ sở hiện diện trong lớp dẫn xuất bất kỳ tiếp theo mà lớp này kế thừa gián tiếp lớp cơ sở. Từ khóa **virtual** đứng trước chỉ định truy cập lớp cơ sở khi nó được kế thừa bởi một lớp dẫn xuất.

EXAMPLES(CÁC VÍ DỤ)

*Here is an example that uses a virtual base class to prevent two copies of **base** from being present in **derived3**.*

Đây là ví dụ dùng một lớp cơ sở ảo để ngăn ngừa 2 bản sao của **base** có mặt trong

derived3:

```
// This program uses a virtual base class.
#include <iostream>
using namespace std;

class base
{
     public:
          int i;
};

// Inherit base as virtual.
class derived1 : virtual public base
{
     public:
          int j;
};

// Inherit base as virtual here, too.
class derived2 : virtual public base
{
     public:
          int k;
};
```

```
/* Here, derived3 inherits both derived1 and derived2.
   However, only one copy of base is present.
   */
   class derived3: public derived1, public derived2
   {
        public:
             int product()
             {
                  return i * j * k;
             }
   };

   int main()
   {
        derived3 ob;

        ob.i = 10; // unambigous because only one copy
   present
        ob.j = 3;
        ob.k = 5;

        cout <<"Product is " << ob.product() << '\n';

        return 0;
   }
```

*If **derived1** and **derived2** had not inherited **base** as virtual, the statement*

Nếu **derived1** và **derived2** không kế thừa **base** như một lớp ảo, thì câu lệnh:

```
ob.i = 10;
```

would have been ambiguous and a compile-time error would have resulted. (See Exercise 1, below.)

sẽ không rõ ràng và sẽ tạo ra lỗi thời gian biên dịch. (Xem bài tập 1 dưới đây).

It is important to understand that when a base class is inherited as virtual by a derived class, that base still exists within that derived class. For example, assuming the preceding program, this fragment is perfectly valid:

Quan trọng cần hiểu là khi một lớp cơ sở được kế thừa như một lớp ảo bởi một lớp dẫn xuất thì lớp cơ sở đó vẫn còn tồn tại trong lớp dẫn xuất đó. Ví dụ, giả sử với chương trình trên đây, đoạn chương trình này hoàn toàn đúng:

```
derived1 ob;
ob.i = 100;
```

The only difference between a normal base class and a virtual one occurs when an object inherits the base more than once. If virtual base classes are used, only one base class is present in the object. Otherwise, multiple copies will be found.

Sự khác biệt duy nhất giữa lớp cơ sở thường và lớp cơ sở ảo xảy ra khi một đối tượng kế thừa lớp cơ sở hơn một lần. Nếu các lớp cơ sở ảo được sử dụng thì chỉ có một lớp cơ sở hiện diện trong đối tượng. Ngược lại, nhiều bản sao sẽ được tìm thấy.

EXERCISES (BÀI TẬP)

*Using the program in Example 1, remove the **virtual** keyword and try to compile the program. See what types of errors result.*

Dùng chương trình trong ví dụ 1, bỏ từ khóa **virtual** và thử biên dịch chương trình. Xem các lỗi.

Explain why a virtual base class might be necessary.

Giải thích tại sao cần lớp cơ sở ảo.

SKILLS CHECK(KIỂM TRA KỸ NĂNG)

Mastery Skills Check

Kiểm tra kỹ năng lĩnh hội

At this point you should be able to perform the following exercises and answer the questions.

Đến đây bạn có thể thực hiện các bài tập và trả lời các câu hỏi sau:

*Create a generic base class called **building** that stores the number of floors a building has, the number of rooms, and its total square footage. Create a derived class*

*called **house** that inherits **building** and also stores the number of bedrooms and the number of bathrooms. Next, create a derived class called **office** that inherits **building** and also stores the number of fire extinguishers and the number of telephones. Note: Your solution may differ from the answer given in the back of this book. However, if it is functionally the same, count it as correct.*

Hãy tạo lớp cở sở chung **building** để lưu trữ số tầng nhà mà một tòa nhà có số phòng và số tổng diện tích. Hãy tạo lớp dẫn xuất **house** kế thừa **building** và lưu trữ. Số phòng ngủ và phòng tắm. Cũng vậy, tạo lớp dẫn xuất **office** kể thừa **building** và cũng lưu trữ số bình cứu hỏa và số mày điện thoại. Chú ý: lời giải của bạn phải khác với lời giải ở cuối cuốn sách này. Tuy nhiên, nếu về mặt chức năng giống nhau thì đếm nó sẽ đúng.

When a base class is inherited as public by the derived class, what happens to its public members? What happens to its private members? If the base is inherited as private by the derived class, what happens to its public and private members?

Khi một thành viên chung của lớp cơ sở được kế thừa như một thành viên chung bởi lớp dẫn xuất, điều gì sẽ xảy ra cho các thành viên chung? Điều gì sẽ xảy ra cho các thành viên riêng? Nếu lớp cơ sở được kế thừa theo cách riêng bởi lớp dẫn xuất, điều gì sẽ xảy ra cho các thành viên chung và riêng?

*Explain what **protected** means. (Be sure to explain what it means both when referring to members of a class and when it is used as an inheritance access specifier.)*

Hãy giải thích ý nghĩa của **protected**. (Hãy giải thích theo hai nghĩa khi tham chiếu đến các thành viên của một lớp và khi được dùng làm chỉ định truy cập tính kế thừa).

When one class inherits another, when are the classes' constructors called? When are their destructors called?

Khi một lớp kế thừa một lớp khác, khi nào các hàm tạo của các lớp được gọi? Khi nào các hàm hủy của các lớp được gọi?

Given this skeleton , fill in the details as indicated in the comments:

Cho dàn chương trình sau, hãy bổ sung các chi tiết theo chỉ dẫn trong các lời chú giải.

```
#include <iostream>
using namespace std;

class planet
{
     protected:
          double distance; // miles from the sun
          int revolve; // in days
     public:
          planet(double d, int r)
          {
               distance = d;
               revolve = r;
          }
};

class earth : public planet
{
```

```
     double circumference; // circumference of orbit

     public:

          /* Create earth (double d, int r). Have it pass
               the distance and days of revolution back
to planet. Have it compute the circumference of the
orbit. (Hint: circumference = 2r*3.1416.)

        */

          /* Create a function called show() that
displays the        information. */

};


int main()

{

     earth ob(93000000, 365);


     ob.show();


     return 0;

}
```

Fix the following program:

Hãy sửa lỗi chương trình sau:

```
/* A variation on the vehicle hierarchy. Butthis program
contains an error. Fix it. Hint: try compiling it as is
and observe the error messages.
```

```
*/
#include <iostream>
using namespace std;

// A base class for various types of vehicle.
class vehicle
{
     int num_wheels;
     int range;
     public:
          vehicle (int w, int r)
          {
               num_wheels = w;
               range = r;
          }
          void showv()
          {
               cout << "Wheels: " << num_wheels << "\n";
               cout << "Range: " << range << "\n";
          }
};

enum motor
{
     gas, electric, diesel
```

```
};

class motorized : public vehicle
{
     enum motor mtr;
     public:
          motorized(enum motor m, int w, int r) :
               vehicle (w, r)
          {
               mtr = m;
          }
          void showm()
          {
               cout << "Motor:";
               switch (mtr)
               {
                    case gas : cout << "Gas\n";
                         break;
                    case electric : cout << "Electric\n";
                         break;
                    case diesel : cout << "Diesel\n";
                         break;
               }
          }
};
```

```
class road_use : public vehicle
{
     int passengers;
     public:
          road_use(int p, int w, int r) : vehicle (w, r)
          {
               passengers = p;
          }
          void showr()
          {
               cout << "Passengers: " << passengers
<<'\n';
          }
};

enum steering
{
     power, rack_pinion, manual
};

class car : public motorized, public road_use
{
     enum steering strng;
     public:
          car (enum steering s, enum motor m, int w, int
r,        int p) : road_use(p, w, r), motorized(m, w, r),
```

```
            vehicle (w, r)
        {
            strng = s;
        }
        void show()
        {
            showv(); showr(); showm();
            cout << "Steering:";
            switch (strng)
            {
                case power : cout << "Power\n";
                    break;
                case rack_pinion : cout "Rack and
                        Panion\n";
                    break;
                case manual : cout << "Manual\n";
                    break;
            }
        }
};

int main()
{
    car c (power, gas, 4, 500, 5);

    c.show();
```

```
        return 0;
}
```

CUMULATIVE SKILLS CHECK

Kiểm tra kỹ năng tổng hợp

This section checks how well you have integrated material in this chapter with that from the preceding chapters.

Phần này kiểm tra xem bạn kết hợp chương này với các chương trước như thế nào.

*In Exercise 6 from the preceding Mastery Skills Check section, you might have seen a warning message (or perhaps an error message) concerning the use of the use of the **switch** statement within **car** and **motorized**. Why?*

Trong bài tập 6 của phần Kiểm Tra Kỹ Năng Lĩnh Hội trên đây, bạn thấy một lời cảnh báo (hoặc có thể là một thông báo lỗi) liên quan đến sử dụng câu lệnh **switch** trong **car()** và **motor()**. Tại sao?

As you know from the preceding chapter, most operators overloaded in a base class are available for use in a derived class. Which one or ones are not? Can you offer a reason why this is the case?

Như bạn biết từ chương trước, hầu hết các toán tử được quá tải trong một lớp cơ sở thì có thể được sử dụng trong lớp dẫn xuất. Có một hay nhiều toán tử nào thì không được như thế? Bạn có thể trình bày lý do cho trường hợp này?

Following is reworked version of the coord class from the previous chapter. This time it is used as a base for another class point is in. On your own, run this program and try to understand its output.

Sau đây là phiên bản sửa đổi của lớp **coord** trong chương trước. Lần này, nó được dùng làm lớp cơ sở cho một lớp khác gọi là **quad** cũng duy trì góc phần tư – trong đó có một điểm cụ thể. Bạn hãy thay chương trình này và tìm hiểu dữ liệu xuất.

```
/* Overload the +, -, and = relative to coord class. Then
   use coord as a base for quad. */

   #include <iostream>

   using namespace std;


   class coord

   {

        public:

             int x, y; // coordinate values

             coord()

             {

                  x = 0;

                  y = 0;

             }

             coord(int i, int j)

             {

                  x = i;

                  y = j;

             }
```

```
        void get_xy(int &i, int &j)
        {
             i = x;
             j = y;
        }
        coord operator + (coord ob2);
        coord operator - (coord ob2);
        coord operator = (coord ob2);
};

// Overload + relative to coord class.
coord coord::operator + (coord ob2)
{
     coord temp;

     cout << "Using coord operator + () \n";

     temp.x = x + ob2.x;
     temp.y = y + ob2.y;

     return temp;
}

// Overload - relative to coord class.
coord coord::operator - (coord ob2)
```

```
{
     coord temp;

     cout << "Using coord operator - () \n";

     temp.x = x - ob2.x;
     temp.y = y - ob2.y;

     return temp;
}

// Overload = relative to coord.
coord coord::operator = (coord ob2)
{
     cout << "Using coord operator = () \n";

     x = ob2.x;
     y = ob2.y;

     return *this; // return the object that is assigned
to
}

class quad : public coord
{
     int quadrant;
```

```
        public:
             quard ()
             {
                  x = 0;
                  y = 0;
                  quadrant = 0;
             }
             quard (int x, int y) : coord (x, y)
             {
                  if(x>=0 && y>=0)
                       quadrant = 1;
                  else
                       if(x<0 && y>=0)
                            quadrant = 2;
                       else
                            if(x<0 && y<0)
                                 quadrant = 3;
                            else quadrant = 4;
             }
             void showq()
             {
                  cout << "Point in Quadrant:" << quadrant
                       <<'\n';
             }
             quad operator = (coord ob2);
   };
```

```
quad quad::operator = (coord ob2)
{
     cout << "Using quad operator=()\n";

     x = ob2.x;
     y = ob2.y;
     if(x>=0 && y>=0)
          quadrant = 1;
     else
          if(x<0 && y>=0)
               quadrant = 2;
          else
               if(x<0 && y<0)
                    quadrant = 3;
               else quadrant = 4;
     return *this;
}

int main()
{
     quad o1(10, 10), o2(15, 3), o3;
     int x, y;

     o3 = o1 + o2; //add two objects-this calls
operator+()
```

```
        o3.get_xy(x, y);
        o3.showq();
        cout << "(o1 + o2)X:"<<x<<", Y:" <<y<< "\n";

        o3 = o1 - o2; // subtract two objects
        o3.get_xy(x, y);
        o3.showq();
        cout << "(o1 - o2)X:"<<x<<", Y:" <<y<< "\n";

        o3 = o1; // assign an object
        o3.get_xy(x, y);
        o3.showq();
        cout << "(o3=o1)X:"<<x<<", Y:" <<y<< "\n";

        return 0;
}
```

Again on your own, convert the program shown in Exercise 3 so that it uses friend operator functions.

Lần nữa, hãy chuyển đổi chương trình trong bài tập 3 sao cho nó sử dụng các hàm toán tử friend.

CHƯƠNG 8

INTRODUCING THE C++ I/O SYSTEM - DẪN NHẬP HỆ THỐNG NHẬP/XUẤT C++

Chapter objectives

8.1. SOME C++ I/O BASICS

Vài cơ sở nhập/ xuất trong C++

8.2. FORMAT I/O

Nhập xuất có định dạng

8.3. USING **WIDTH(), PRECISION(), AND FILL()**

Sử dụng các hàm **width(), precision(), fill()**

8.4. USING I/O MANIPULATORS

Sử dụng bộ thao tác nhập xuất

8.5. CREATING YOUR OWN INSERTERS

Tạo bộ chèn riêng của bạn

8.6. CREATING EXTRACTORS

Tạo bộ chiết

Although you have been using C++ style I/O since the first chapter of this book, it is time to explore it more fully. Like its predecessor, C, the C++ language includes a rich I/O system that is both flexible and powerful. It is important to understand that C++ still supports the entire C I/O system. However, C++ supplies a complete set of object-oriented I/O routines. The major advantage of the C++ I/O system is that it can be overload relative to class that you create. Put differently, the C++ I/O system allows you seamlessly integrate new types that you create.

Mặc dù các bạn đã được sử dụng Nhập/Xuất của C++ kể từ đầu của cuốn sách này, vấn đề sẽ được trình bày lại một cách cặn kẽ. Giống như trong ngôn ngữ C, C++ có một hệ thống Nhập/Xuất linh động và hữu hiệu. Một điều quan trọng mà bạn cần biết đến là C++ vẫn hỗ trợ hoàn toàn hệ thống Nhập/Xuất định hướng đối tượng. Ưu điểm chính của hệ thống Nhập/Xuất của C++ là nó có thể được quả tải lên các lớp do bạn tạo ra. Hay nói cách khác, hệ thống Nhập/Xuất của C++ còn cho phép tích hợp vào ó các kiểu tự tạo một cách êm thấm.

Like the C I/O system, the C++ object-oriented I/O system makes little distinction between console and file I/O. File and console I/O are really just different perspectives on the same mechanism.The examples in this chapter use console I/O, but the information presented is applicable to file I/O as well. (File I/O is examined in detail in Chapter 9.)

Giống như C, hệ thống Nhập/Xuất của C++ hầu như không phân biệt giữa thiết bị và tập tin Nhập/Xuất trên thiết bị chuẩn và trên tập tin chỉ là những dáng vẻ khác nhau của

cùng một cơ chế mà thôi. Trong chương này, những chương trình mẫu được viết cho các thiết bị Nhập/Xuất, nhưng nó cũng hoàn toàn có thể áp dụng cho Nhập/ xuất tập tin (Vấn đề Nhập/Xuất tập tin sẽ được đề cập chi tiết trong chương 9).

At the time of this writing, there are two versions of the I/O library in use: the older one that is based on the original specifications for C++ and the newer one defined by Standard C++. For the most part the two libraries appear the same to the programmer. This is because the new I/O library is, in essence, simply an updated and improve version of the old one. In fact, the vast majority of the differences between the two occur beneath the surface, in the way that the libraries are implemented-not in the way that they are used. From the programmer's perspective, the main difference is that is that the new I/O library contains a few additional features and defines some new data types. Thus, the new I/O library essentially a superset of the old one. Nearly all programs originally written for the old library will compile without substantive changes when the new library is used. Since the old-style I/O library is now obsolete, this book describes only the new I/O library as defined by Standard C++. But most of the information is applicable to the old I/O library as well.

Tại thời điểm bài viết này, có hai phiên bản thư viện nhập xuất được sử dụng: cái cũ hơn dựa trên cơ sở những đặc tính gốc của C++ và phiên bản mới hơn được định nghĩa bởi C++ chuẩn. Đối với người lập trình trình thì 2 thư viện này hầu hết đều giống nhau. Đơn giản là vì thư viện nhập xuất mới là phiên bản cập nhật và cải tiến của phien bản cũ trước nó. Thực tế, phần lớn sự khác biệt giữa 2 thư viện này xảy ra dưới bề mặt, ở cách mà các thư viện này thực hiện không như cách mà chúng được sử dụng. Đứng từ góc độ của người lập trình thì khác biệt chính đó là thư viện nhập xuất mới chứa đựng một vài đặc tính bổ sung và định nghĩa một số kiểu dữ liệu mới. Vì vậy, về bản chất thì thư viện nhập xuất mới là một tập hợp lớn hơn bao gồm cả cái cũ. Gần đây, tất cả các chương trình được viết nguyên gốc với thư viện cũ được biên dịch không có thay đổi gì khi sử dụng thư viện mới. Vì rằng thư viện kiểu cũ này hiện giờ đã lỗi thời, nên cuốn sách này chỉ mô tả thư viện nhập xuất kiểu mới được định nghĩa bởi C++ chuẩn. Tuy nhiên hầu hết thông tin được áp dụng ở đây cũng phù hợp với thư viện cũ.

This chapter covers several aspects of C++'s I/O system, including formatted I/O, I/O manipulators, and creating your own I/O inserters and extractors. As you will see, the C++ system shares many features with the C I/O system.

Nội dung của chương này liên quan đến các khía cạnh đặc trưng của các hệ thống Nhập/Xuất C++ bao gồm Nhập/Xuất có định dạng, bộ thao tác Nhập/Xuất, tạo lập bộ chèn (inserter) và bộ chiết (extractor). Qua đó, bạn sẽ thấy được nhiều ưu điểm giống nhau giữa hệ thống Nhập/Xuất của C và C++.

REVIEW SKILLS CHECK: (kiểm tra kỹ năng ôn tập)

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Trước khi bắt đầu chương mới, bạn hãy trả lời các câu hỏi và làm các bài tập sau đây:

Create a class hierarchy that stores information about airships. Start with a general base class called ***airship*** *that stores the number of passengers and the amount of cargo (in pounds) that can be carried. Then create two derived classes call* ***airplane*** *and* ***balloon*** *from* ***airship.*** *Have* ***airplane*** *store the type of engine used (propeller or jet) and range, in miles. Have* ***balloon*** *store information about the type of gas used to lift the balloon (hydrogen or helium) and its maximum altitude (in feet). Create a short program that demonstrates this class hierarchy. (Your solution will, no doubt, differ from the answer shown in the back of this book. If it is functionally similar, count it as correct).*

Hãy tạo ra một phân cấp lớp để lưu trữ thông tin về các tàu bay. Hãy bắt đầu một lớp cơ sở tên là airship chứa thông tin về số lượng hành khách tối đa và trọng lượng hàng hóa tối đa (đơn vị tính là pound) mà tàu bay có thể chở được. Sau đó, từ lớp cơ sở airship, hãy tạo hai lớp dẫn xuất (derived class) mang tên là airplane và balloon. Lớp airplane lưu kiểu của động cơ (gồm động cơ cánh quạt và động cơ phản lực), tầm xa (đơn vị tính là mile). Lớp balloon lưu thông tin về loại nhiên liệu sử dụng ch khí cầu (gồm hai loại là hyrogen và helium), độ cao tối đa (đơn vị tính là feet). Háy viết một chương trình ngắn minh họa cho phân cấp lớp trên.(Dĩ nhiên là bài giải của bạn phải khác chút ít so với bài giải ở phần sau quyển sách này. Nếu giải thuật của chúng tương tự nhau, tức là bạn đã giải đúng.)

What is ***protected*** *used for?*

Công dụng của bộ đặc tả thâm nhập có bảo vệ (protected) là gì?

Give the following class hierarchy, in what order are the constructor functions called? In what order are the destructor functions called?

Trong phân cấp lớp sau đây, hàm tạo (constructor) được gọi đến như thế nào? Hàm hủy (destructor) được gọi đến như thế nào?

```
#include<iostream.h>

        class A{

             public:
```

```
                    A(){cout<<"Constructing A\n";}
                    ~A(){cout<<"Destructing A\n";}
          };

class B:publicA{
              public:
                    B(){cout<<"Constructing B\n";}
                    ~ B(){cout<<"Destructing B\n";}

};

class C: publicB{
              public:
                     C(){cout<<"Constructing C\n";}
              ~ C(){cout<<"Destructing C\n";}
};

Main()
{
C ob;
return 0;
}
```

Give the following fragment, in what order are the constructor functions called?

Trong đoạn chương trình sau đây, hàm tạo được gọi đến như thế nào?

class myclass: public A, public B, public C {…

Fill in the missing constructor function in this program:

Điền vào chương trình sau đây những hàm tạo còn thiếu:

```
#include<iostream.h>
        class base{
        int l, j;
        public:
            //need constructor
            Void showij() {cout <<l<<''<<j<<'\n';}
        };
        class derived : public base {
            int k;
        public;
           //need constructor
           void show() {cout <<k<<''; showij();}
        };
        main()
        {
             dereved ob(1, 2, 3);
             ob.show();
             return 0;
        }
```

In general, when you define a class hierarchy, you begin with the most _____ class and move to the most _____ class. (Fill in the missing words).

Nói chung, khi định nghĩa một phân cấp lớp, người ta bắt đầu từ lớp ……. nhất,

và dần đến lớp ……. nhất. (Điền vào khoảng trống).

1.1. SOME C++ I/O BASICS - Cơ sở Nhập/Xuất C++

Before we begin our examination of C++ I/O, a few general comments are in order. The C++ I/O system, like the C I/O system, operates through streams. Because of your C programming experience, you should already know what a stream is, but here is a summary. A stream is a logical device that either produces or consumes information. A stream is linked to a physical device by the C++ I/O system. All streams behave in the same manner. Even if the actual the physical devices they are linked to differ. Because all streams act the same, the I/O system presents the programmerwith a consistent interface, even though it operates on devices with differing capabilities. For example, the same function that you use to write to the screen can be used to write to a disk file or to the printer.

Trước hết, giống như ở C, hệ thống Nhập/Xuất của C++ cũng điều khiển các luồng (stream). Đối với các bạn đã tưng lập trình C,chắc hẳn đã biết luồng là gì, tuy nhiên ở đây chúng ta sẽ tóm tắt khái niệm này. Luồng là một thiết bị luận lý có khả năng tạo ra hoặc sử dụng thông tin. Nhờ hệ thống Nhập/Xuất của C++, mỗi luồng được liên kết với thiết bị vật lý. Tuy nhiên, cho dù có nhiều loại thiết bị vật lý khác nhau, nhưng các luồng đều được xử lý như nhau. Chính vị vậy mà hệ thống Nhập/Xuất có thể vận hành trên bất kỳ loại thiết bị nào. Lấy ví dụ, cách mà bạn có thể xuất thông tin ra màn hình có thể sử dụng cho việc ghi thông tin lên tập tin trên đĩa hay xuất ra máy in.

As you know, when a C program begins execution, three predefined streams are automatically opened: ***stdin, stdout,*** *and* ***stderr****. A similar thing happens when a C++ program starts running. When a C++ program begins, these four streams are automatically opened:*

Stream	Meaning	Default device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error	Screen
clog	Buffered version of cerr	Screen

Như chúng ta đã biết, khi một chương trình C được thực thi, có ba luồng đã định sẵn được mở một cách tự động là : stdin, stdout, stderr. Tương tự như vậy, khi một chương trình C++ được thực thi, sẽ có bốn luồng được mở một cách tự động. Đó là:

Luồng	**Ý nghĩa**	**Thiết bị mặc định**
cin	Thiết bị nhập chuẩn	Bàn phím
cout	Thiết bị xuất chuẩn	Màn hình
cerr	Thiết bị báo lỗi chuẩn	Màn hình
clog	Phiên bản của cerr	Màn hình

*As you have probably guessed, the stream **cin, cout,** and **cerr** correspond to C's **stdin, stdout,** and **stderr.** You have already been using **cin** and **cout.** The stream **clog** is simply a buffered versions of **cerr.** Standard C++ also opens wide (16-bit) character versions of these streams called **wcin, wcout, wcerr,** and **wclog,** but we won't be using them in this book. The wide character streams exist to support languages, such as Chinese, that require large character sets.*

Bạn có thể đoán được rằng các từ cin, cout và stdin, stdout và stderr của C. Chúng ta đã sử dụng cin và cout. Luồng clog là một phiên bản nằm ở vùng đệm của **cerr**. C++ chuẩn cũng mở rộng các phiên bản của các luồng này gọi là **wcin, wcout, wcerr,** and **wclog,** nhưng chúng ta sẽ không sử dụng chúng trong cuốn sách này. Các luồng ký tự mở rộng này hỗ trợ cho các ngôn ngữ khác, như tiếng Trung Quốc chẳng hạn, ngôn ngữ này đòi hỏi một bộ ký tự rộng hơn.

By default, the standard streams are used to communicate with the console. However, in environments that support I/O redirection, these streams can be redirected to others devices.

Do mặc định, các luồng chuẩn được liên kết với thiết bị xuất nhập chuẩn. Tuy nhiên, chúng ta có thể định lai cho các luồng để liên kết gắn với các thiết bị xuất nhập khác.

*As you learned in chapter 1, C++ provides support for its I/O system in the header file <**iostream**>. In this file, a rather complicated set of class hierarchies is defined that supports I/O operations. The I/O classes begin with a system of template classes. Template classes, also called generic classes, will be discussed more fully in Chapter 11: briefly, a template class defines the form of a class without fully specifying the*

data upon which it will operate. Once a template class has been defined, specific instances of it can be created. As it relates to the I/O library, Standard C++ creates two specific versions of the I/O template classes: one for 8-bit characters and another for wide characters. This book will discuss only the 8-bit characters classes, since they are by far the most frequently used.

Trong chương 1, C++ cung cấp các hỗ trợ cho hệ thống Nhập/Xuất trong file đề mục <iostream>. Nội dung của tập tin này là các phân cấp lớp hỗ trợ các thao tác nhập/xuất. Các lớp nhập xuất bắt đầu với một hệ thống các lớp mẫu. Các lớp mẫu này, còn được gọi là các lớp tổng quát, sẽ được trao đổi đầy đủ hơn trong chương 11: nói một cách ngắn gọn, một lớp mẫu định nghĩa dạng của lớp không định rõ đầy đủ dữ liệu mà lớp đó vận dụng. Lớp mẫu định nghĩa trong một lần và các phiên bản riêng của nó có thể được tạo ra sau đó. Vì nó liên quan đến thư viện nhập xuất nên C++ chuẩn tạo ra 2 phiên bản riêng cho các lớp nhập/ xuất mẫu này: một cái cho các ký tự 8 bit và cái kia cho các ký tự lớn hơn. Cuốn sách này chỉ trao đổi về các lớp ký tự 8 bit, vì chúng thì thường xuyên được sử dụng.

The C++ I/O system is built upon two related, but different, template class hierarchies. The first is derived from the low-level I/O class called ***basic_streambuf*** *directly. The class hierarchy that you will most commonly be working with is derived from* ***basic_ios.*** *This is a high-level I/O class that provides formatting, error-checking, and status information related to stream I/O.* ***basic_ios*** *is used as a base for several derived classes, including* ***basic_istream, basic_ostream*** *and* ***basic_iostream.*** *These classes are used to create streams capble of input, output and input/output, respectively.*

C++ có hai phân cấp lớp Nhập/Xuất, chúng có liên hệ cới nhau nhưng chúng không giống nhau. Phân cấp lớp Nhập/Xuất thứ nhất được suy dẫn từ lớp Nhập/Xuất cấp thấp, tên là **basic_streambuf**. Lớp này cung cấp các thao tác cơ bản của Nhập/Xuất của C++. Bạn không cần phải sử dụng trực tiếp lớp streambuf này, trừ phi bạn đang lập trình nhập/xuất ở trình độ cao. Thông thường, bạn sẽ sử dụng một phân cấp lớp nhập xuất khác, có tên là **basic_ios**. Đó là lớp nhập/xuất cấp cao, nó cung cấp các thao tác về định dạng, kiểm lỗi, thông tin trạng thái của các luồng nhập/xuất. lớp **basic_ios** là lớp cơ sở bao gồm các lớp istream, ostream, iostream. Ba lớp xuất này được sử dụng để tạo ra các luồng nhập, xuất, và nhập/xuất.

As explained earlier, the I/O library creates two specific versions of the class hierarchies just described: one for 8-bit characters and one for wide characters. The following table shows the mapping of the template class names to their 8-bit character-based versions (including some that will be used in Chapter 9):

Template Class	8-Bit Character-Based Class
basic_streambuf	streambuf
basic_ios	ios
basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_fstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream

Như đã được giải thích ở trên, thư viện nhập/xuất tạo ra 2 phiên bản riêng cho các phân cấp lớp cụ thể là: một phiên bản cho các ký tự 8 bit và một cho các ký tự rộng hơn. Bảng sau cho biết ánh xạ của các tên lớp mẫu cho các phiên bản cơ sở của ký tự 8 bit (bao gồm một số được sử dụng trong chương 9):

Lớp mẫu	Lớp cơ sở ký tự 8 bit
basic_streambuf	streambuf
basic_ios	ios
basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_fstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream

The character-based names will be used throughout the remainder of this book, since they are the names that you will you in your programs. They are also the same names that were used by the old I/O library. This is why the old and the new I/O

libraries are compatible at the source code level.

Các tên của ký tự cơ sở sẽ được sử dụng trong phần còn lại của cuốn này, vì chúng được sử dụng trong các chương trình của bạn. Các tên này cũng giống như tên được sử dụng trong thư viện nhập/xuất cũ. Đó là lý do tại sao các thư viện cũ và mới tương thích với nhau ở cấp độ của mã nguồn này.

*One last point: The **ios** class contains many member functions and variables thar control or monitor the fundamental operation of a stream. It will be referred to frequently. Just remember that if you include **<iostream>** in your program, you will have access to this important class.*

Điểm lưu ý cuối cùng: Lớp **ios** chứa nhiều hàm và biến dùng để điều khiển, và kiểm soát các thao tác cơ bản của một luồng. Lớp này thường được tham khảo đến. Nếu bạn muốn chương trình ứng dụng của mình có thể sử dụng được lớp ios này, hãy nạp tập tin tiêu đề <iostream>.

1.2. FORMATTED I/O - Nhập/Xuất Có Định Dạng

*Until now, all examples in this book displayed information to the screen using C++'s default formats. However, is is possible to output information in a wide variety of forms. In fact, you can format data using C++'s I/O system in much the same way that you do using C's **printf( )** function. Also, you can alter certain aspects of the way information is input.*

Cho đến bây giờ, tất cả các chương trình mẫu trong quyển sách này đều hiển thị thông tin ra màn hình theo dạng mặc định của C++. Tuy nhiên, chúng ta hoàn toàn có thể xuất thông tin theo nhiều hình thức . Nói tóm lai, bạn có thể định dạng dữ liệu bằng hệ thống nhập/xuất của C++ bằng cách giống như sử dụng hàm printf() của C chuẩn. Ngoài ra, bạn có thể thay đổi một số hình thức nhập dữ liệu.

*Each stream has associated with it a set of format flags that control the way information is formatted. The I/O class declares a bitmask enumeration called **fmtflags,** in which the following values are defined:*

Mỗi luồng của C++ được đi kèm với một tập hợp các cờ định dạng. các cờ đinh dạng này xác định cách thức thể hiện của thông tin. Lớp nhập/xuất khai báo một kiểu liệt kê dữ liệu gọi là **fmtflags** được định nghĩa bởi các giá trị sau:

adjustfield	floatfield	right	skipws
basefield	hex	scientific	unitbuf
boolalpha	internal	showbase	uppercase
dec	left	showpoint	
fixed	oct	showpos	

*These values are used to set or clear the format flags and are defined within **ios.** If you are usin an order, nonstandard compiler, it may not define the **fmtflags** enumeration type. In this case, the format flags will be encoded into a long integer.*

Các giá trị này được sử dụng để thiết lập hay xóa bỏ các cờ định dạng và được định nghĩa trong **ios.** Nếu bạn sử dụng một trình biên dịch không đạt chuẩn, nó có thể định nghĩa kiểu liệt kê **fmtflags.** Trong trường hợp này, các cờ định dạng sẽ được mã hóa thành một số nguyên dài.

*When the **skipws** flags is set, leading whitespace characters (spaces, tabs, and newlines) are discarded when input is being perform on a stream. When **skipws** is cleared, whitespace characters are not discarded.*

Khi các cờ **skipws** được thiết lập, các ký tự khoảng trắng (gồm các ký tự khoảng cách, tab, và xuống dòng) được bỏ đi khi đọc một luồng. Khi cờ này bị xóa, các ký tự khoảng trắng sẽ không bị bỏ đi.

*When the **left** flag is set, output is left justified. When right is set, output is **right** justified. When the **internal** flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character. If none of these flags is set, output is right justified by default.*

Khi cờ **left** được đặt, kết xuất sẽ được canh biên trái. Khi bạn thiết lập cờ **right**, kết xuất được canh phải. khi đặt cờ **internal**, một giá trị số được thêm vào để điền vào một trường bằng các chèn vào các khoảng giữa ký tự cơ sở hoặc dấu. Nếu không có cờ nào được thiết lập, theo mặc định, kết xuất sẽ được canh biên phải.

*By default, numeric values are output is decimal. However, it is possible to change the number base. Seting the **oct** flag causes ouput to be displaced in octal. Seting the hex flag causes ouput to be displaced in hexadecimal. To return output to decimal, set the dec flag.*

Mặc nhiên, các giá trị số được trình bày dưới dạng số thập phân. Tuy nhiên, chúng ta

có thể thay đổi cơ số của kết xuất. Thiết lập cờ **oct** sẽ làm cho kết xuất được trình bày ở dạng số hệ bát phân. Khi đặt cờ hẽ kết xuất là số hệ thập lục phân. Để trả lại kiểu số thập phân, ta thiết lập cờ **dec.**

*Setting **showbase** causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal the value 1F will be displaced as 0x1F.*

Khi đặt cờ **showbase**, cơ số của giá trị số được trình bày. Ví dụ, nếu số được trình bày ở hệ thập lục phân, giá trị 1F sẽ được thay bằng 0x1F.

*By default, when scientific notation is displaced, the e is lowercase. Also, when a hexadecimal value is displaced, the x is lowercase. When **uppercase** is set, these characters are displaced uppercase.*

*Setting **showpos** causes a leading plus sign to be displaced before positive values.*

Theo mặc đinh, khi một con số kiểu số mũ được in ra, chữ "e" được trình bày ở kiểu chữ thường. thương tự như vậy, khi in một giá trị số ở hệ thập lục phân, ký tự "x" dùng chỉ hệ thập lục được trình bày ở kiểu chữ thường. khi cờ upease được thiết lập, các ký tự nói trên sẽ được trình bày bằng kiểu chữ in hoa.

Việc thiết lập cờ **showpos** làm cho xuất hiện dấu cộng phía trước các giá trị số dương.

*Setting **showpoint** causes a decimal point and trailing zeros to be displaced for all floating-point output – whether needed or not.*

Thiết lập cờ **showpoint** cho phép in dấu chấm thập phân và các số không đi sau kèm theo các giá trị kiểu chấm động.

*If the **scientific** flag is set, floating-point numeric values are displaced using scientific notation. When **fixed** is set, floating-point values are displaced using normal notation. When neither flag is set, the compiler chooses an approriate method.*

Cờ **scientific** được thiết lập làm các giá trị số kiểu chấm động được trình bày dưới dạng mũ. Nhưng khi đặt cờ **fixed**, các giá trị số kiểu dấu chấm động sẽ được thể hiện theo dạng bình thường. Nếu bạn không lập một cờ nào trong hai cờ nói trên, trình biên dịch sẽ chọn cách thích hợp.

*When **unitbuf** is set, the buffer is flushed after each insertion operation.*

*When **boolalpha** is set, Booleans can be input or output using the keywords **true** and **false.***

Khi cờ **unitbuf** được cài đặt, bộ nhớ đệm được xóa sạch sau mỗi hành động chèn vào.

Khi cờ **boolalpha** được cài đặt, các luận lý có thể được nhập và xuất bằng việc sử dụng từ khóa **true** và **false**.

*Since it is common to refer to the oct, dec, and hex fields, they can be collectively referred to as **basefield.** Similarly, the left, right, and internal fields can be referred to as adjustfield. Finally, the scientific and fixed fields can be referenced as **floatfield**.*

Kể từ bây giờ, đây là chuyện bình thường để tham chiếu tới cờ **oct**, cờ **dec** và cờ **hex**, chúng có thể được tập hợp tham chiếu như basefield. Tương tự, cờ **left**, cờ **right** và cờ **internal** có thể tham chiếu như **adjustfield**. Cuối cùng, cờ **scientific** và cờ **fixed** có thể tham chiếu như **floatfield**.

*To set a format flag, use the **setf( )** function. This function is a member of ios. Its most common form is shown here:*

Hàm **setf()** dùng để thiết lập cờ định dạng. hàm này thuộc lớp **ios**. Dạng sử dụng thường gặp ở hàm này là:

```
fmtflags setf(fmtflags flags);
```

This function returns the previous settings of the format flags and turns on those flags specified by flags. (All other flags are unaffected). For example, to turn on the showpos flag, you can use this statement:

Hàm này trả về giá trị được thiết lập trước đó của cờ và thiết lập các cờ có tên trong hàm. (Các cờ khác không có tên trong mệnh đề gọi hàm sẽ không bị ảnh hưởng.). Ví dụ, để thiết lập cờ showpos, ta sử dụng mệnh đề sau:

```
stream.setf(ios::showpos);
```

*Here stream is the stream you wish to affect. Notice the use of the scope resolution operator. Remember, **showpos** is an enumerated constant within the **ios** class. Therefore, it is necessary to tell the compiler this fact by preceding **showpos** with the class name and thwe scope resolution operator. If you don't, the constant **showpos** will simply not be recognized.*

Ở đây, *stream* là tên luồng bạn muốn tác động**.** Bạn hãy lưu ý đến việc sử dụng toán tử phạm vi ở đây. Vì cờ **showpos** là môt hằng kiểu liệt kê thuộc lớp **ios**. Vì vậy, chúng

ta cần phải thông báo cho trình biên dịch biết điều này bằng cách viết tên lớp ios và toán tử phạm vi :: trước tên cờ **showpos**. Nếu không trình biên dịch sẽ không nhận biết được từ **showpos**.

*It is important to understand that **setf( )** is a member function of the **ios** class and affects streams created by that class. Therefore, any call to **setf( )** is done relatve to a specific stream. There is no concept of calling **setf( )** by itself. Put differently, there is concept in C++ of global format status. Each stream maintains its own format status information individually.*

Một điều quan trọng nữa là hàm **setf()** là một thành phần của lớp **ios**, và nó sẽ ảnh hưởng đến các luồng tạo ra bởi lớp **ios** này. Cho nên , tất cả các mệnh đề gọi hàm seft() đều sử dungj cho một luồng cụ thể, và hàm seft() không thể gọi một cách chung chung được. hay nói khác đi, trong C++, chúng ta không thể đặt cờ định dạng ở phạm vi toàn cục chung cho ất cả các luồng. Mỗi luồng sẽ mang một thông tin trạng thái các định dạng riêng của nó và chúng độc lập với nhau.

*It is possible to set more than one flag is a single call to **setf( )**, rater than making multiple calls. To do this, OR together the values of the flags you want to set. For example, this call sets the **showbase** and hex flags for cout:*

Chúng ta có thể gọi hàm **setf( )** để lập nhiều cờ cùng lúc thay vì gọi hàm này nhiều lần cho mỗi cờ. Để thực hiện điều này, chúng ta sử dụng toán tử OR. Ví dụ để thiết lập 2 cờ **showbase** và **hex,** chúng ta dùng mệnh đề sau:

```
cout.setf(ios::showbase | ios::hex);
```

*REMEMBER: Because the format flags are defined within the **ios** class, you must access their values by using **ios** and the scope resolution operator. For example, **showbase** by itself will not be recognized; you must specify **ios::showbase.***

Cần nhớ: Vì các cờ định dạng được định nghĩa trong lớp **ios**, bạn phải truy xuất đến giá trị của các cờ này bằng cách sử dụng tên lớp ios cùng với toán tử phạm vi. Ví dụ, chúng ta phải viết là **ios::showbase** thay vì chỉ viết là **showbase** sẽ không được nhận biết.

*The complement of **setf( )** is **unsetf( ).** This member function of **ios** clears one or more format flags. Its most common prototype form is shown here:*

Cùng với hàm thiết lập cờ định dạng **setf( ),** còn có hàm xóa cờ **unsetf( ).** Hàm này cũng là một thành phần của lớp **ios,** dùng để xóa một hay nhiều cờ định dạng. Dạng sử

dụng thường gặp của hàm này là:

```
Void unsetf(fmtflags flags);
```

The flags specified by flags are cleared. (All other flags are unaffected).

There will be times when you want to know, but not alter, the current format settings. Since both ***setf( )*** *and* ***unsetf( )*** *alter the setting of one or more flags,* ***ios*** *also includes the member function flags( ), which simply returns the current setting of each format flag. Its prototype is shown here:*

Các cờ định dạng được chỉ rõ bởi các cờ của hàm sẽ bị xóa (các cờ khác thì không bị ảnh hưởng).

Có nhiều lần khi chúng ta biết, nhưng không có sự thay đổi, cách thiết lập định dạng hiện thời. Bởi vì cả hai hàm **setf()** và **unsetf()** thay đổi thiết lập của một hoặc nhiều cờ, nên ios cũng bao gồm hàm thành phần **flags()**, đơn giản trả về sự thiết lập hiện thời của mỗi cờ định dạng. Mẫu của nó được trình bày như sau :

```
Fmtflags flags( );
```

The flags() function has a second form that allows you to set all format flags associated with a stream to those specified in the argument to ***flags( ).*** *The prototype for this version of flags( ) is shown here:*

Hàm **flags()** có một dạng thứ hai cho phép chúng ta thiết lập tất cả các cờ định dạng liên kết với một luồng và được chỉ tới đối số **flags()**. Mẫu của phiên bản này của hàm **flags()** được trình bày như sau :

```
Fmtflags flags(fmtflags f);
```

When you use this version, the bit pattern found in f is copied to the variable used to hold the format flags associated with the stream, thus overwriting all previous flag settings. The function returns the previous setings.

Khi bạn sử dụng dạng này, từng bit của giá trị f sẽ được gán cho biến lưu trị cờ định dạng, và sẽ xóa tình trạng cờ định dạng trước đó. Hàm sẽ trả về thiết lập trước đó.

EXAMPLES:

Here is an example that shows how to set several of the format flags:

Đây là ví dụ minh họa cách thiết lập một vài cờ định dạng:

```
include <iostream>
using namespace std ;

int main( )
{
      // display using default settings
      cout << 123.23 << "hello" << 100 << '\n' ;
     cout << 100.0 << "\n\n" ;

     // now change formats
     cout.unsetf (ios: :dec) ;
     cout.setf (ios: :hex | ios: :scientific) ;
     cout << 123.23 << "hello" << 100 << '\n' ;

     cout.setf (ios: :showpos) ;
     cout << 10 << '  ' << -10 << '\n' ;
     cout.setf (ios: :showpoint | ios: :fixed) ;
     cout << 100.0 ;

      return 0 ;
}
```

This program displays the following output:

123.23 hello 100

10 -10

100

1.232300e-02 hello 64

a ffffffff6

+100.000000

*Notice that the **showpos** flag affects only decimal output. It does not affect the value 10 when output in hexadecimal. Also notice the **unsetf( )** call that turns off the dec flag (which is on by default). This call is not needed by all compilers. But for some compilers, the dec flag overrides the other flags, so it is necessary to turn it off when turning on either **hex** or **oct.** In general, for maximum portability, it is better to set only the number base that you want to use and clear the others.S*

Lưu ý rằng cờ **showpos** chỉ ảnh hưởng đến kết xuất của giá trị số thập phân. Nó không ảnh hưởng đến kết xuất của giá trị 10 khi giá trị này được trình bày ở số hệ thập lục phân. Cũng chú ý đến lời gọi hàm **unsetf( )** sẽ tắt cờ **dec** (được thiết lập như mặc định). Lời gọi này thì không cần thiết đối với mọi trình biên dịch. Nhưng trong một vài trình biên dịch, cờ **dec** được viết chồng lên các cờ khác, do vậy nó cần được tắt đi khi bật cờ **hex** hoặc cờ **oct.** Nói chung, để tiện lợi tối đa, sẽ tốt hơn khi chỉ thiết lập cơ sở số mà bạn muốn sử dụng và xóa những cái khác.

*The following program illustrates the effect of the **uppercase** flag. It first sets the **uppercase, showbase,** and hex flags. It then outputs 88 in hexadecimal. In this case, the X used in the hexadecimal notation is uppercase. Next, it clears the **uppercase** flag by using **unsetf( )** and again outputs 88 in hexadecimal. This time, the x is lowercase.*

Chương trình sau minh họa tác dụng của cờ **uppercase.** Cờ được thiết lập đầu tiên là **uppercase, showbase,** và **hex.** Tiếp theo chương trình sẽ in ra giá trị 88 ở dạng số mũ, và ký tự X dùng để thông báo cơ số 16 được in dưới kiểu chữ in hoa. Sau đó, chương trình dùng hàm unsetf( ) để xóa cờ **uppercase,** kết quả là xuất ra 88 ở dạng thập lục phân và ký tự x ở dạng kiểu chữ thường.

```
#include <iostream>
```

```
using namespace std ;

int main( )
{
cout.unsetf( ios: :dec) ;
cout.setf( ios: :uppercase | ios: :showbase | ios: :hex)
;

cout << 88 << '\n' ;
cout.unsetf(ios: :uppercase) ;
cout << 88 << '\n' ;
return 0 ;
}
```

*The following program use **flags( )** to display the settings of the format flags relative to cout. Pay special attention to the **showflags( )** function. You might find it useful in programs you write.*

Chương trình sau sử dụng hàm **flag( )** để hiện thông tin trạng thái của các cờ định dạng thuộc luồng cout. Hãy đặc biệt chú trọng đến hàm **showflag(),** bạn sẽ thấy được tính hữu dụng của nó:

```
#include <iostream>
using namespace std ;

void showflag( ) ;
int main( )
{
showflags( ) ;
cout.setf( ios: :oct | ios: :showbase | ios: :fixed) ;
```

```
showflags( ) ;
return 0 ;
}

void showflags( )
{
ios: :fmtflags f ;
f = cout.flags( ) ;

if( f & ios: :skipws)
    cout << "skipws on\n" ;
else
    cout << "skipws off\n" ;

if( f & ios: :left)
    cout << "left on\n" ;
else
    cout << "left off\n" ;

if( f & ios: :right)
    cout << "right on\n" ;
else
    cout << "right off\n" ;

if( f & ios: :internal)
```

```
        cout << "internal on\n" ;
    else
        cout << "internal off\n" ;

    if( f & ios: :dec)
        cout << "dec on\n" ;
    else
        cout << "dec off\n" ;
    if( f & ios: :oct)
        cout << "oct on\n" ;
    else
        cout << "oct off\n" ;

    if( f & ios: :hex)
        cout << "hex on\n" ;
    else
        cout << "hex off\n" ;

    if( f & ios: :showbase)
        cout << "showbase on\n" ;
    else
        cout << "showbase off\n" ;

    if( f & ios: :showpoint)
        cout << "showpoint on\n" ;
```

```
else
    cout << "showpoint off\n" ;

if( f & ios: :showpos)
    cout << "showpos on\n" ;
else
    cout << "showpos off\n" ;

if( f & ios: :uppercase)
    cout << "uppercase on\n" ;
else
    cout << "uppercase off\n" ;

if( f & ios: :scientific)
    cout << "scientific on\n" ;
else
    cout << "scientific off\n" ;

if( f & ios: :fixed)
    cout << "fixed on\n" ;
else
    cout << "fixed off\n" ;

if( f & ios: :unitbuf)
    cout << "unitbuf on\n" ;
```

```
else
    cout << "unitbuf off\n" ;

if( f & ios: :boolalpha)
    cout << "boolalpha on\n" ;
else
    cout << "boolalpha off\n" ;

cout << '\n' ;
}
```

*Inside **showflags( ),** the local variable f is declared to be of type **fmtflags.** If your compiler does not define **fmtflags,** declare this variable as long instead. The output from the program is shown here:*

Bên trong hàm **showflags( ),** biến cục bộ f được khai báo kiểu **fmtflags.** Nếu trình biên dịch không định nghĩa **fmtflags,** thì khai báo biến này theo kiểu **long.** Kết quả xuất ra của chương trình trên như sau:

```
Skipws on
Left off
Right off
Internal off
Dec on
oct off
hex off
showbase off
showpoint off
showpos off
```

```
uppercase off
scientific off
fixed off
unitbuf off
boolalpha off

skipws on
left off
right off
internal off
dec on
oct on
hex off
showbase on
showpoint off
showpos off
uppercase off
scientific off
fixed on
unitbuf off
boolalpha off
```

*The next program illustrates the second version of **flags( ).** It first constructs a flag mask that turn on **showpos, showbase, oct,** and **right.** It then uses **flags( )** to set the flag variable associated with cout to these settings. The function **showflags( )** verifies that the flags are set as indicated. (This is the same function used in the previous program).*

4. Chương trình tiếp sau đây minh họa cho cách sử dụng thứ hai của hàm flags(). Trước tiên chương trình sẽ tạo ra một mặt nạ cờ dùng để đặt các cờ **showpos, showbase, oct** và **right.** Sau đó chương trình sẽ thiết lập các biến cờ kết hợp với hàm cout cho các thiết đặt này. Hàm **showflags( )** xác định các cờ đã được bật. (Đây là hàm sử dụng như trong chương trình trước).

```
#include <iostream>

using namespace std;


void showflags( ) ;

int main( )

{

    // show default condition of format flags

    showflags( ) ;


    // showpos, showbase, oct, right are on, others off

    ios :: fmtflags f = ios :: showpos | ios :: showbase
|

      ios :: oct | ios :: right ;


        cout.flags(f) ;    //set flags

        showflags( ) ;

        return 0 ;

}
```

EXERCISES:

1. write a program that sets cout's flags so that integers display a + sign when positive values are displayed. Demonstrate that you have set the format flags correctly.

Viết chương trình thiết lập trạng thái các cờ định dạng sao cho các giá trị nguyên dương được in ra có mang dấu +. Hãy in ra cho biết các cờ định dạng đã được thiết lập đúng.

2. write a program that sets cout's flags so that the decimal point is always shown when floating-point values are displayed. Also, display all floating-point values in scientific notation with an uppercase E.

Hãy viết chương trình để thiết lập trạng thái các cờ định dạng của luồng cout sao cho các trị số kiểu chấm động được trình bày với kiểu chấm thập phân. Ngoài ra các trị số kiểu chấm động được in ra dưới dạng số mũ với ký tự 'E' chỉ phần mũ được trình bày bằng kiểu in hoa.

*3. write a program that saves the current state of the format flags, sets **showbase** and **hex,** and displays the value 100. Then reset the flags to their previous values.*

Hãy viết chương trình thực hiện việc lưu lại trang thái hiện tại của các cờ định dạng, thiết lập 2 cờ **showbase** và **hex** và xuất ra giá trị 100. Sau đó thiết lập các cờ trở lại như trạng thái trước đó.

1.3. USING WIDTH( ), PRECISION( ), AND FILL( ) – SỬ DỤNG HÀM WIDTH(), PRECISION( ), VÀ FILL( ):

*In addition to the formatting flags, there are three member functions defined by **ios** that set these format parameters: the field width, the precision, and the fill character. These are **width( ), precision( ),** and **fill( ),** respectively.*

*By default, when the value is output, it occupies only as much space as the number of characters it takes to display it. However, you can specify a minimum field width by using the **width( )** function. It prototype is shown here:*

Cùng với các cờ định dạng, còn có ba hàm thành phần nữa của lớp **ios.** Các hàm này thiết lập các tham số định dạng gồm có: độ rộng của trường, độ chính xác và ký tự điền vào. Các hàm này là **width( ), precision( ),** và **fill( ).**

Theo mặc định, việc xuất một trị số cần một số khoảng trống bằng với số lượng ký tự cần thiết để thể hiện con số đó. Tuy nhiên khi sử dụng hàm **width( ),** bạn có thể quy định một độ rộng tối thiểu của trường. Dạng thể hiện của hàm này là:

```
Streamsize width(streamsize w) ;
```

*Here w becomes the field width, and the previous field width is returned. The streamsize type is defined by <**iostream**> as some form of integer. In some implementations, each time an output operation is performed, the field width returns to its default setting, so it might be necessary to set the minimum field width before each output statement.*

Ở đây w là độ rộng của trường, hàm sẽ trả về độ rộng của trường trước đó. Kiểu **streamsize** được định nghĩa bằng <iostream> như một vài dạng số nguyên. Trong khi ứng dụng, sau mỗi lần một thao tác xuất được thực hiện, độ rộng của trường được trả lại giá trị mặc định của nó, cho nên chúng ta cần phải đặt lại độ rộng của trường trước mỗi lệnh xuất.

After you set a minimum field width, when a value uses less than the specified width, the field is padded with the current fill character (the space by default) so that the field width is reached. However, keep in mind that if the size of the output value exceeds the minimum field width, the field will be overrun. No values are truncated.

*By default, six digits of precision are used. You can set this number by using the **precision( )** function. Its prototype is shown here:*

Sau khi thiết lập độ rộng trường tối thiểu, nếu giá trị được in ra có chiều dài ngắn hơn độ rộng tối thiểu, trường sẽ được thêm vào một hay nhiều ký tự điền vào để xuất ra có chiều dài bằng đúng với độ rộng tối thiểu. Nhưng nếu như giá trị được in ra cần số khoảng trắng vượt quá độ rộng tối thiểu, trường sẽ bị tràn vì giá trị đó không bị cắt bớt.

Độ chính xác gồm 6 chữ số sau dấu chấm thập phân là thiết lập mặc định. Tuy nhiên, chúng ta có thể quy định lại độ chính xác bằng hàm **precision( ).** Dạng hàm như sau:

```
Streamsize precision(streamsize p);
```

Here the precision is set to p and the old value is returned.

*By default, when a field needs to be filled, it is filled with spaces. You can specify the fill character by using the **fill( )** function. Its prototype is shown here:*

Với p là độ chính xác cần thiết lập và trả về giá trị là độ chính xác trước đó.

Theo mặc định, nếu có một trược cần được lấp đầy, nó sẽ được lấp bằng các ký tự khoảng cách. Nhưng chúng ta vẫn có thể thay đổi ký tự khoảng trắng dùng để lấp vào bằng ký tự khác nhờ vào hàm **fill( ).** Dạng hàm như sau:

```
Char fill(char ch) ;
```

*After a call to **fill( ),** ch becomes the new field character, and the old one is returned.*

Sau khi gọi hàm **fill**( ), ch trở thành ký tự lấp vào mới và trả về cho hàm ký tự trước đó.

EXAMPLES (Các ví dụ):

Here is a program that illusrates the format functions:

Chương trình sau minh họa cho các hàm định dạng nói trên:

```
#include <iostream>

using namespace std ;


int main( )

{

    cout.width(10) ;

    cout << "hello" << '\n' ;  // right justify by
default

    cout.fill('%') ;          // set fill character

    cout.width(10) ;          // set width

    cout << "hello" << '\n' ;  // right justify by
default

    cout.setf(ios: : left) ;    // left- justify

    cout.width(10) ;        //set width

    cout << "hello" << '\n' ;  // output left justified


    cout.width(10) ;         // set width
```

```
    cout. precision(10);  // set 10 digits of precision
    cout << 123.234567 < '\n' ;
    cout.precision(6) ;
    cout << 123.234567 << '\n' ;
    return 0;
}
```

This program displays the following output:

Chương trình xuất ra như sau:

```
     hello
%%%%%hello
Hello%%%%%
123.234567
123.235%%%
```

Notice that the field width is set before each output statement.

Chú ý là độ rộng của trường được định lại trước mỗi lệnh xuất.

The following program shows how to use the C++ I/O format functions to create an aligned table of numbers:

Chương trình sau cho biết cách sử dụng các hàm định dạng nhập/xuất của C++ để in ra một bảng số:

```
// create a table of square roots and squares.
#include <iostream>
#include <cmath>
using namespace std ;
```

```
int main( )
{
   double x ;

   cout.precision(4) ;
cout << "    x    sqrt(x)        x^2\n\n" ;

for (x = 2.0 ; x <= 20.0 ; x++)
{
   cout.width(7) ;
   cout << x << "   ;
   cout.width(7) ;
   cout << sqrt(x) << "    " ;
   cout.width(7) ;
   cout << x*x << '\n' ;
}

return 0 ;
}
```

This program creates the following table:

Chương trình in ra bảng số như sau:

```
  x sqrt(x)        x^2
    2    1.414          4
    3    1.732          9
```

```
4        2          16
5    2.236          25
6    2.449          36
7    2.646          49
8    2.828          64
9        3          81
10   3.162          100
11   3.317          121
12   3.464          144
13   3.606          169
14   3.742          196
15   3.873          225
16       4          256
17   4.123          289
18   4.243          324
19   4.359          361
20   4.472          400
```

EXERCISES:

Create a program that prints the natural log and base 10 log of the numbers from 2 to 100. Format the table so the numbers are right justified within a field width of 10, using a precision of five decimal places.

Hãy viết chương trình in ra logarit tự nhiên và logarit cơ số 10 của các số từ 2 đến 100. Định dạng bảng số sao cho các số được canh biên phải với độ rộng trường là 10. Sử dụng độ chính xác là 5.

*Create a function called **center( )** that has this prototype:*

Tạo ra một hàm gọi là **center( )** có định dạng như sau:

```
Void center(char *s) ;
```

*Have this function center the specified string on the screen. To accomplish this, use the **width( )** function. Assume that the screen is 80 characters wide. (For simplicity, you may assume that no string exceeds 80 characters). Write a program that demonstrates that your function works.*

Chức năng của hàm này là canh giữa một chuỗi ký tự trên màn hình. Để thực hiện hàm này, hãy sử dụng hàm **width( ).** Giả sử là màn hình có độ rộng 80 cột (và để cho đơn giản, giả sử chuỗi ký tự có chiều dài không quá 80 ký tự).

On your own, experiment with the format flags and the format functions. Once you become familiar with the C++ I/O system, you wil have no trouble using it to format output any way you like.

Bạn hãy tự thực hành các nội dung liên quan đến các cờ và hàm định dạng. Một khi bạn đã quen thuộc với hệ thống nhập xuất của C++, bạn sẽ loại bỏ được các trục trặc khi sử dụng nó để định dạng kết xuất theo ý muốn.

1.4. USING I/O MANIPULATORS – SỬ DỤNG BỘ THAO TÁC NHẬP XUẤT

*There is a second way that you can format information using C++'s I/O system. This method uses special functions called I/O manipulators. As you will see, I/O manipulators are, in some situations, easier to use than the **ios** format flags and functions.*

Có một cách thứ 2 mà bạn có dùng để định dạng thong tin là sử dụng hệ thống I/O trong C++. Phương thức này dùng như 1 hàm đặc biệt để gọi thao tác I/O.Như bạn thấy, thao tác I/O, trong một vài trường hợp, dễ sử dụng hơn định dạng cờ và hàm **ios**.

I/O manipulators are special I/O format functions that can occur within an I/O statement, instead of separate from it the way the ios member functions must. The

standard manipulators are shown in Table 8.1. As you will see, Many of the I/O manipulators parallel member functions of the ios class. Many of the I/O manipulators are shown in Table 8.1 were added recently to Standard C++ and will be supported only by modern compilers.

Thao tác I/O thường là các hàm mẫu nhập xuất mà có thể xuất hiện trong câu lệnh I/O, thay vì phải chia ra như là các hàm thành viên của ios . Các thao tác nhập xuất chuẩn được thể hiện trong bảng 8.1. Như bạn thấy, nhiều thao tác nhập xuất I/O thì tương đương với các hàm trong lớp ios.Nhiều thao tác I/O trong bảng 8.1 mới được bổ sung vào thư viện chuẩn của C++ và chỉ được hỗ trợ bởi các trình biên dịch hiện đại.

*To access manipulators that take parameters, such as **setw(),** you must include <**iomanip**> in your program. This is not necessary when you are using a manipulators that does not require an argument.*

Để truy xuất đến các thao tác nhập xuất mà có tham số, chẳng hạn như setw(), bạn phải thêm vào thư viện <**iomanip**> trong chương trình của bạn. Điều này không cần thiết khi bạn sử dụng thao tác nhập xuất mà không có đối số.

As stated above, the manipulators can occur in the chain of I/O operations. For example:

Các thao tác nhập xuất có thể xuất hiện trong một chuỗi các toán tử I/O. Ví dụ như:

```
Cout<< oct<<100<<hex<<100;

Cout<<setw(10)<<100;
```

The first statement tells cout to display integers in octal and then outputs 100 in octal. It then tells the stream to display integers in hexadecimal and then outputs 100 in hexadecimal format. The second statement sets the field to 10 and then displays 100 in hexadecimal format again. Notice that when a manipulator does not take an argument, such as oct in the example, it is not followed by parentheses. This is because it is the address of the manipulators that is passed to the overloaded << operator.

Câu lệnh thứ nhất dùng cout để hiện thị số nguyên trong hệ thập phần và sao đó xuất 100 ở hệ thập phân. Sau đó tiếp tục hiện thị số nguyên trong hệ thập lục phân và xuất 100 ra màn hình trong định dạng thập lục phân. Dòng lệnh thứ 2 thì đặt lại trường 10 và xuất ra 100 ra trong định dạng thập lục phân. Chú ý rằng khi mà một thao tác nhập

xuất không có đối số, chẳng hạn như là oct trong ví dụ trên, nó thường không nằm trong dấu ngoặc đơn. Vì vậy địa chỉ của thao tác được bỏ qua để nạp chồng toán tử <<.

Manipulator	Purpose	Input/Output
boolalpha	Turn on boolalpha flag	Input/output
dec	Turns on dec flag	Input/output
endl	Outputs a newline character and flushes the stream	Output
ends	Output a null	Output
fixed	Tuens on fixed flag	Output
flush	Flush a stream	Output
hex	Turns on hex flag	Input/output
internal	Turns on internal flag	Output
left	Turns on left flag	Output
noboolalpha	Turns off boolalpha flag	Input/output
noshowbase	Turns off showbase flag	Output
noshowpoint	Turns off showpoint flag	Output
noshowpos	Turn off showpos flag	Output
noskipws	Turns off skipws flag	Input
nounitbuf	Turn off unitbuf flag	Output
nouppercase	Turn off uppercase	Output
oct	Turns on oct flag	Input/output
resetiosflags(fmtflags f)	Turns off the flags specified in f	Input/output
right	Turns on right flag	Output
scientific	Turn on scientific flag	Output
setbase(int base)	Sets the number base to base	Input/output

setfill(int ch)	Sets the fill character to ch	Output
setiosflags(fmtflags f)	Turn on the flags specified in f	Input/output
setprecision(int p)	Sets the number of digits of precision	Output
setw(int w)	Sets the field width to w	Output
showbase	Turns on showbase flag	Output
showpoint	Turns on showpoint flag	Output
showpos	Turns on showpoint flag	Output
skipws	Turns on skipws flag	Output
unibuf	Turns on unibuf flag	Output
uppercase	Turns on uppercase	Output
ws	Skips leadung white space	Input

Keep in mind that an I/O manipulator affects only the stream of which the I/O expression is a part. I/O manipulators do not affect all streams currently opened for use.

As the preceding example suggests, the main advantages of using manipulators over the ios member functions is that they are often easier to use and allow more compact code to be written.

Hãy nhớ rằng một thao tác I/O chỉ ảnh hưởng trên dòng mà biểu thức I/O nằm trong đó. Thao tác I/O không ảnh hưởng đến tất cả các dòng hiện thời đang mở.

Như ví dụ trên, tác dụng chính của việc dùng thao tác nhập xuất hơn là các hàm của ios là nó thường dễ dùng và cho phép nhiều thao tác trên 1 dòng.

*If you wish to set specific format flags manually by using a manipulator, use **setiosflags().** This manipulator performs the same function as the member function **setf**(). To turn off flags, use the **resectiosflags()** manipulator**. This** manipulator is equivalent to **unsetf().***

Nếu bạn muốn đặt 1 định dạng cờ cụ thể thường dùng bằng cách sử dụng thao tác nhập

xuất, sử dụng hàm setiosflags(). Thao tác này biểu diễn giống như là hàm setf(). Để tắt cờ, sử dụng thao tác resectiosflags() . Thao tác này tương đương với unsetf().

EXAMPLE:

This program demonstrates several of the I/O manipulators:

Chương trình mô ra một vài thao tác I/O:

```
#include <iostream>

#include <iomanip>

using namespace std;

int main()

{

     cout<<hex<<100<<endl;

     cout<<oct<<10<<endl;

     cout<<setfill('X')<<setw(10);

     cout<<100<<"hi "<<endl;

     return 0;

}
```

This program display the following:

```
64

12

XXXXXXX144hi
```

Here is another version of the program that displays a table of the squares and square roots of the numbers 2 through 20. This version uses I/O manipulators instead of member functions and format flags.

Đây là một phiên bản khác của chương trình hiện thị bảng bình phương và căn bậc 2 của một số từ 2 đến 20. Trong phiên bản này thao tác I/O dùng thay cho hàm và định dạng cờ.

```
#include <iostream>

#include <iomanip>

#include <cmath>

using namespace std;

int main()

{

      double x;

      cout<<setprecision(4);

      cout<<"    x sqrt(x)  x^2\n\n";

      for ( x=2.0; x<=20.0;x++)

      {

            cout<<setw(7)<<x<<"  ;  ";

            cout<<setw(7)<<sqrt(x)<< " ";

            cout<<setw(7)<<x*x;'\n';

      }

      return 0;

}
```

*One of the most interesting format flags added by the new I/O library is **boolalpha** . This flag can be set either directly or by using the new manipulator **boolalpha or noboolalpha.** What makes **boolalpha** so interesting is that setting it allows you to input and output Boolean values using the keywords true and false. Normally you must enter 1 for true and 0 for false. For example, consider the following program:*

Một trong những điều thú vị của cờ chuẩn được thêm vào bởi thư viện I/O mới là **boolalpha.** Cờ này có thể được đặt ko trực tiếp hoặc được sử dụng bới thao tác

boolalpha hoặc noboolalpha. Cái gì làm boolalpha trở nên thú vị? đó là thiết lập nó cho phép bạn nhập và xuất giá trị luận lý bằng cách sử dụng bàn phím true or false. Thường bạn phải nhập 1 cho true và 0 cho false. Xem một ví dụ dưới đây:

```
#include <iostream>

using namespace std;

int main()

{

     bool b;

     cout<<" Before setting boolalpha flag: ";

     b=true;

     cout<<b<<" ";

     b=false;

     cout<<b<<endl;

     cout<<"After setting boolalpha flag: ";

     b=true;

     cout<<boolalpha<<b<<" ";

     b=false;

     cout<<b<<endl;

     cout<<"Enter a Boolean value";

     cin>>boolalpha>>b;//you can enter true or false

     cout<<"You entered "<<b;

     return 0;

}
```

Here is sample run:

```
Before setting boolalpha flag: 1 0
```

```
After setting boolalpha flag: true false

Enter a Boolean value :true

You entered true
```

*As you can see, once the **boolalpha** flag has been set, Boolean values are input and output using the words **true or false.** Notice that you must set the **boolalpha** flags for **cin and cout** separately. As with all format flags, setting **boolalpha** for own stream does not imply it is also set for another.*

Như bạn thấy, một lần cờ **boolalpha** được thiết lập.Giá trị luận lý được nhập và xuất bằng cách sử dụng từ **true** hay **false**. Chú ý rằng bạn phải thiết lập cờ **boolalpha** cho hàm **cin** và **cout.** Vì với tất cả các cờ định dạng, thiết lập cờ **boolalpha** cho các luồng riêng không kéo theo việc thiết lập cho các luồng khác.

EXERCISES

Redo Exercise 1 and 2 form section 8.3, this time using I/O manipulators instead of member functions and format flags.

Làm lại bài 1 và 2 trong phần 8.3, lần này dùng thao tác I/O thay vì dùng hàm thành viên và cờ định chuẩn.

*Show the I/O statement that outputs the value 100 in hexadecimal with the base indicator (the 0x) shown. Use the **setiosflags()** manipulator to accomplish this.*

Chỉ ra câu lệnh I/O dùng xuất giá trị 100 trong hệ thập lục phân với từ chỉ dẫn là 0x . Dùng thao tác **setiosflags()** để thực hiện điều này.

*Explain the effect of setting the **boolalpha flag.***

Giải thích tác dụng của việc đặt cờ boolalpha.

1.5. CREATING YOUR OWN INSERTERS – TẠO BỘ CHÈN VÀO:

As stated earlier, one of the advantages of the C++I/O system is that you can overload the I/O operators for classes that you create. By doing so, you can seamlessly incorporate your classes into your C++ programs, In this section you learn how to overload C++'s output operator<<.

Giống như các phần trước , một trong những thuận lợi của hệ thống I/O trong C++ là bạn có thể nạp chống các toán tử cho lớp của bạn tạo. Bằng cách làm như thế, ban có thể kết nối chặt chẽ lớp của bạn vào chương trình C++ của bạn. Trong phần này bạn sẽ học được cách làm như thế nào để nạp chồng toán tử << trong C++.

In the language of C++, the output operation is called an insertion and the << is called the insertion opertator. When you overload the << for output, you are creating an inserter function, or inserter for short. The rationale for these terms comes from the fact that an output operator inserts information into a stream.

Trong ngôn ngữ C++, thao tác xuất được gọi là một " bộ chèn" và << được gọi là toán tử chèn. Khi bạn nạp chồng << cho việc xuất dữ liệu, bạn đang tạo một hàm chèn hoặc là một bộ chèn ngắn. Lý do cơ bản cho việc này đến từ sự thật là một thao tác xuất sẽ đưa thông tin vào một "dòng"

All inserter functions have this general form:

Mọi hàm chèn đều có cấu trúc chung như sau:

```
     Ostream & operator<<( ostream& istream, class name
ob)

     {

          //body of inserter

     return stream;

     }
```

*The first parameter is a reference to an object of type **ostream. This** means that stream must be an output stream. (Remember, **ostream** is derived from the ios class). The second parameter receives the object that will be output. (This can also be reference parameter, if that is more suitable to your application.) Notice that the inserter function return if the overload << is going to be used in a series of I/O expressions, such as*

cout<<ob1<<ob2<<ob3;

Tham số đầu tiên là một tham chiếu của đối tượng của loại **ostream.** Điều này có nghĩa

là dòng phải là một dòng xuất. (Nhớ rằng, **ostream** được xuất phát từ lớp ios). Tham số thứ 2 là một đối tượng được xuất ra.(nó có thể là tham số truyền theo địa chỉ, nếu điều là phù hợp cho ứng dụng của bạn). Chú ý rằng hàm chèn trả về nếu sự nạp chồng << được dùng trong một dãy các lệnh I/O, chẳng hạn như cout<<ob1<<ob2<<ob3

Within an inserter you can perform any type of procedure. What an inserter does is completely up to you. However, for the inserter to be consistent with good programming practices, you should limit its operations to outputting information to a stream.

Trong một bộ chèn bạn có thể biểu diễn bất kì một loại thủ tục nào. Một bộ chèn làm gì để hoàn thiện . Tuy nhiên, một bộ phải nhất quán với chương trình, bạn nên hạn chế thao tác của nó để xuất thông tin trong dòng

*Although you might find this surprising at first, an inserter cannot be a member of the class on which it is designed to operate. Here is why: When an operator function of any type is a member of a class, the left operand, which is passed implicitly through the **this** pointer, is the object that generates the calls to the operator function. This implies that the left operand is an object of that class. Therefore, if an overloaded operator function is a number of a class, the left operand must be an object of that class. However, when you create an inserter, the left operand is a stream and the right operand is the object that you want to output. Therefore, an inserter cannot be a member function.*

Vậy mà bạn có thể ngạc nhiên với điều này, một bộ chèn không thể là một thành phần của lớp. Lý do là : khi mà hàm toán tử của bất kì loại nào là một thành viên của lớp, toán hạng bên trái, có thể ngầm định bỏ qua thông qua con trỏ **this**, là một đối tượng gọi hàm toán tử. Một gợi ý là toán hạng bên trái là một đối tượng của lớp. Bởi vậy, nếu một toán tử được nạp chồng cho một số lớp. toán hạng bên trái phải là một đối tượng của lớp.Tuy nhiên khi bạn tao một bộ cài, toán hạng bên trái là dòng và toán hạng bên phải là một đối tượng của lớp mà bạn muốn xuất ra. Vì vậy, một bộ chèn không thể là một thành phần của lớp .

The fact that an inserter cannot be a member function might appear to be a serious flaw in C++ because it seems to imply that all data of a class that will be output using an inserter will need to be this is not the case. Even though inserters cannot be members of the class upon which they are designed to operate, they can be friends of the class. In fact, in most programming situations you will encounter, an overloaded inserter will be a friend of the class for which it was create.

Sự thật một bộ chèn không thể là một hàm thành viên có thể xuất hiện là một lổ hổng nghiêm trọng của C++ bởi vì nó dường như là việc đưa tất cả dữ liệu của lớp sẽ được

xuất ra ngoài sử dụng bộ chèn sẽ trờ nên cần thiết ko phài trong trường hợp này. Mặc dù bộ chèn ko thể là thành phần của một lớp được thiết kế cho thao tác, chúng có thể là bạn của lớp. Sự thật, trong hầu hết chương trình bạn sẽ gặp việc này, việc nạp chồng một bộ chèn sẽ là một hàm bạn của lớp.

EXAMPLES

*As a simple first example, this program contains an inserter for the **coord** class, developed in a previous chapter:*

Đây là một ví dụ đơn giản đầu tiên, một chương trình chứa một bộ chèn cho lớp coord, phát triển từ chương trước:

```
// Use a friend inserter for objects of type coord

#include <iostream>

using namespace std;


class coord

{

     int x,y;

public:

     coord(){ x=0;y=0;}

     coord ( int i , int j){ x=i;y=j;}

     friend ostream& operator<<(ostream & stream, coord
ib);

};

ostream& operator<<(ostream & stream, coord ob)

{

     stream<<ob.x<<","<<ob.y<<"\n";

     return stream;
```

```
}
int main()
{
     coord a(1,1),b(10,23);
     cout<<a<<b;
     return 0;
}
```

This program displays the following”

```
1 ,1
10,23
```

The inserter in this program illustrates one very important point about creating your own inserters: make them as general as possible. In this case, the I/O statement inside the inserter outputs the values of x and y to stream, which is whatever stream is passed to the function. As you will see in the following chapter, when written correctly the same inserter that outputs to the screen can be used to output any stream. Sometimes beginners are tempted to write the coord inserter like this:

Bộ chèn trong chương trình minh họa một điều rất quan trọng về việc tao một bộ chèn cho chính bạn:làm chúng một cách tổng quát. Trong trường hôp này, câu lệnh nhập xuất bên trong bộ chèn xuất một giá trị của x và y vào dòng, mà bất cứ cái gì trên dòng đều vượt qua hàm. Như bạn thấy ở chương trước, khi mà viết nhiều bộ chèn thì việc xuất ra màn có thể được dùng để xuất bất cứ dòng nào. Một vài người mới lập trình sẽ bị lôi cuốn viết bộ chèn cho coord như thế này:

```
Ostream & operator << ( ostream& stream, coord ob)
{
     cout<<ob.x<<ob.y<<”\n”;
     Return stream;
}
```

In this case, the output statement is hard-coded to display information on the standard output device linked to cout. However, this prevents the inserter from being used by other streams. The point is that you should make your inserters as general as possible because there is no disadvantage to doing so.

Trong trường hợp này, câu lệnh xuất là một câu lệnh tối nghỉa để hiện thị thông tin trên thiết bị xuất chuẩn lien kết với cout. Tuy nhiên, việc ngăn cản 1 bộ chèn bằng cách dùng một dòng khác. Điểm này là điểm mà bạn co thể làm cho bộ chèn của mình tổng quát nhất bởi vì không có bất kỳ 1 sự bất thuận lợi nào.

For the sake of illustration, here is the preceding program revised so that the inserter is not a friend of the coord class. Because the inserter does not have access to the private part of coord, the variables x and y have to be made public.

Để minh họa, đây là một chương trình có trước được viết lại mà bộ chèn không phải là bạn của lớp coord. Bởi vì bộ chèn không được phép truy xuất đến thành phần riêng của lớp coord, giá trị của x và y phải là public.

```
// Use a friend inserter for objects of type coord

#include <iostream>

using namespace std;


class coord

{

public:

     int x,y;

     coord(){ x=0;y=0;}

     coord ( int i , int j){ x=i;y=j;}

     ostream& operator<<(ostream & stream, coord ib);

};

ostream& operator<<(ostream & stream, coord ob)
```

```
{
      stream<<ob.x<<","<<ob.y<<"\n";
      return stream;
}
int main()
{
      coord a(1,1),b(10,23);
      cout<<a<<b;
      return 0;
}
```

An inserter is not limited to displaying only textual information. An inserter can perform any operation or conversion necessary to output information in a form needed by a particular device or situation. For example, it is perfectly valid to create an inserter will need to send appropriate plotter codes in addition to the information. To allow you to taste the flavor of this type of inserter, the following program creates a class called ***triangle,*** *which stores the width and height of a right triangle. The inserter for this class displays the triangle on the screen.*

Một bộ chèn không chỉ hạn chế trong việc hiện thị thông tin. Một bộ chèn có thể biểu diễn bất kỳ thao tác nào hoặc việc chuyển đổi cần thiết để xuất thông tin trong định dạng cần bằng một thiết bị đặc biết hay hoàn cảnh.Ví dụ như, thật là hoàn hảo để tạo một bộ chèn để gởi đến mã in phù hợp bổ sung vào thông tin. Cho phép bạn nếm điều thú vị của loại bộ chèn này, chương trình dưới đây tạo ra lớp triangle, mà chứ chiều rộng và cao của một tam giác vuông. Bộ chèn của lớp này hiện thị tam giác này ra màn hình.

```
//This program draw right triangle
#include <iostream>
using namespace std;
class triangle
{
      int height,base;
```

```
public:
     triangle ( int h,int b){ height=h;base=b;}
     friend ostream& operator<<(ostream&stream,triangle
ob);
};
//Draw a triangle.
ostream &operator<<(ostream & stream , triangle ob)
{
     int i,j,h,k;
      i=j=ob.base-1;
      for ( h=ob.height-1;h;h--)
      {
            for (k=i;k;k--)
                  stream<<' ';
            stream<<'*';
            if (j!=i)
            {
                  for (k=j-i-1;k;k--)
                        stream<< ' ';
                  stream<<'*';
            }
            i--;
            stream<<'\n';
      }
      for (k=0;k<ob.base;k--)
            stream<<'*';
```

```
       return stream;
}
int main()
{
     triangle t1(5,5),t2(10,10),t3(12,12);
     cout<<t1;
     cout<<endl<<t2<<endl<<t3;
     return 0;
}
```

Notice that this program illustrate how a properly designed inserter can be fully integrated into a "normal" I/O expression. This program displays the following.

Chú ý rằng chương trình này minh họa làm thế để thiết kế đúng cách một hàm chèn có thể kết hợp một cách tốt nhất với lệnh nhập xuất cơ bản. Chương trình dưới đây sẽ biểu hiện điều này.

EXERCISES

*given the following **strtype class** and partial program, create an inserter that displays a string*

Tạo hàm chèn cho lớp strtype và 1 chương trình nhỏ để hiện thị một chuỗi

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype
{
```

```
     char *p;
     int len;
public:
     strtype(char *ptr);
     ~strtype(){delete []p;}
     friend ostream& operator<<(ostream & stream,strtype&
obj);
};
strtype::strtype(char *ptr)
{
     len =strlen(ptr)+1;
     p=new char[len];
     if(!p)
     {
          cout<<"Allocation error\n";
          exit(1);
     }
     strcpy(p,ptr);
}
//Create operator << inserter function here
int main()
{
     strtype s1("This is a test "),s2( "I like C++");
     cout<<s1<<'\n'<<s2;
     return 0;
}
```

Replace the show() function in the following program with an inserter function

Thay thế hàm show() trong chương trình dưới bằng hàm chèn.

```
#include <iostream>
using namespace std;
class planet
{
protected:
     double distance;
     int revolve;
public:
     planet( double d, int r){distance =d; revolve = r;}
};
class earth:public planet{
     double  circumference;
public:
     earth ( double d,int r): planet(d,r)
     {
          circumference= 2*distance*3.1416;
     }
     void show()
     {
          cout<<"Distance from sun : <<distance <<'\n';
          cout<< " Days in obit : "<<revolve<<'\n';
     }
```

```
};
int main()
{
      earth ob(9300000,365);
      cout<<ob;
      return 0;
}
```

Explain why an inserter cannot be member function

Giải thích tại sao một bộ chèn không thể là một hàm thành viên trong class

1.6. CREATING EXTRACTORS – TẠO BỘ CHIẾT:

Just you can overload the << output operator, you can overload the >> input operator. In C++, the >> is referred to as the extraction operator and a function that overloads it is called an extractor. The reason for this term is that the act of inputting information from a stream removes (that is, extracts) data from it.

Không chỉ với toán tử xuất <<, bạn còn có thể nạp chồng đối với toán tử nhập >>. Trong C++, toán tử >> được dùng như là một toán tử nguồn và một hàm nạp chồng được gọi từ hàm này. Đó là lý do giới hạn trong thao tác nhập thông tin từ một dòng sự kiện loại bỏ dữ liệu từ nó.

The general form of an extractor function is shown here:

Hình thức chung của một hàm nhập được thể hiện dưới đây:

```
istream & operator>>( istream & stream, class-name &ob)
{
      // Body of extractor
```

```
    Return stream;
}
```

*Extractor return a reference to **istream,** which is an input stream. The first parameter must be a reference to an input stream. The second parameter is a reference to the object that is receiving input.*

Hàm nhập trả về một tham số là istream,nằm trong "dòng" nhập. Tham số đầu tiên phải là tham chiếu để nhập từ "dòng". Tham số thứ 2 là một tham số của đối tượng mà sẽ nhận thao tác nhập.

For the same reason that an inserter cannot be a member function, any operation within an extractor, it is best to limit its activity to inputting information.

Cũng giống như hàm chèn , mọi thao tác trong hàm nhập không thể là thành viên của một lớp, đó là hạn chết trong hoạt động nhập thông tin.

EXAMPLES

This program adds an extractor to the coord class:

Chương trình này thêm một thao tác nhập vào lớp coord

```
#include <iostream>
using namespace std;


class coord
{
    int x,y;
public:
    coord(){ x=0;y=0;}
    coord ( int i , int j){ x=i;y=j;}
    friend ostream& operator<<(ostream & stream, coord
ib);
```

```
     friend istream& operator>>(istream & stream, coord
ib);
};
ostream& operator<<(ostream & stream, coord ob)
{
     stream<<ob.x<<","<<ob.y<<"\n";
     return stream;
}
istream& operator>>(istream & stream, coord &ob)
{
     stream<<ob.x<<","<<ob.y<<"\n";
     return stream;
}
int main()
{
     coord a(1,1),b(10,23);
     cout<<a<<b;
     cin>>a;
     return 0;
}
```

Notice how the extractor also prompts the user for input. Although such prompting is not required (or even desired for most situations), this function shows how a customized extractor can simplify coding when a prompting message is needed.

Để ý làm cách nào mà hàm nhập cũng nhắc nhở người dùng nhập. Mặc dù không cần thiết (….), hàm này chỉ ra làm cách nào để chọn hàm nhập có làm đơn giản mã lệnh khi mà thông báo nhập thật sự cần thiết.

Here an inventory class is created that stores the name of an item, the number on hand, and its cost. The program includes both an inserter and an extractor for this class.

Đây là một lớp kiểm kê được tạo để chứa tên của một món đồ, số lượng trên tay, và giá của nó. Chương trình bao gồm cả hàm chèn và hàm nhập trong lớp.

```
#include <iostream>

#include <cstring>

using namespace std;

class inventory

{

     char item[40];

     int onhand;

     double cost;

public:

     inventory(char *i,int o, double c)

     {

          strcpy(item,i);

          onhand=o;

          cost =c;

     }

     friend ostream& operator<<(ostream &stream, inventory
ob);

     friend istream& operator<<(istream &stream, inventory
&ob);

};

friend ostream& operator<<(ostream &stream, inventory ob)

{

     stream<<ob.item<<" : "<<ob.onhand;
```

```
        stream<<"on hand at S" << ob.cost<<"\n";
        return stream;
}
friend istream& operator<<(istream &stream, inventory
&ob);
{
        cout<< "Enter item name : ";
        stream>>ob.item;
        cout<<" Enter number on hand : ";
        stream>>ob.onhand;
        cout<<"Enter cost :";
        stream >> ob.cost;
        return stream;
}
int main()
{
        inventory ob("hammer",4,12,55);
        cout<<ob;
        cin>>ob;
        cout<<ob;
        return 0;
}
```

EXERCISES

*Add an extractor to the **strtype** class from exercise 1 in the preceding section*

Thêm vào hàm nhập cho lớp **strtype** trong bài 1 trong phần trước

Create a class that stores an integer value and its lowest factor. Create both an inserter and an extractor for this class.

Tạo một lớp chứ giá trị một số nguyên và thừa số nhỏ nhất. Tạo cả hàm chèn và hàm nhập cho lớp này.

SKILL CHECK

Mastery Skill Check (Kiểm tra kỹ năng lĩnh hội)

At this point you should be able to perform the following exercises and answer the questions.

Tại thời điểm này bạn có thể thực hiên các bài tập dưới đây và trả lời các câu hỏi:

Write a program that displays the number 100 in decimal, hexadecimal, and octal. (Use the ios format flags)

Viết chương trình biểu diễn số 100 trong hệ thập phân, hệ thập lục phân và hệ bát phân (sử dụng định dạng cờ ios).

Write a program that displays the value 1000.5364 in 20 character field, left justified, with two decimal places, using as a fill character. (Use the ios format flags).

Viết chương trình hiện thị giá trị 100.5364 trong vùng 20 ký tự, canh trái, với 2 phần thập phân, sử dụng để điền đầy vào các ký tự. (Dùng định dạng cờ ios).

Rewrite your answers to Exercises 1 and 2 so that they use I/O manipulators.

Viết lại câu trả lời cho bài 1 và 2 sử dụng thao tác I/O

Show how to save the format flags for cout and how to restore them. Use either member functions or manipulators.

Chỉ ra làm cách nào để lưu lại cờ chuẩn choc out và làm sao khôi phục lại chúng. Sử dụng các hàm thành viên khác hay là thao tác nhập xuất.

Create an inserter and an extractor for this class:

Tạo bộ chèn và hàm nhập cho lớp này

```
Class pwr

{

      Int base;

      Int exponent;
```

```
      Double result;// base to the exponent power
      Public:
            Pwr(int b,int e);
};
Pwr::pwr( int b,int e)
{
      Base =b;
      Exponent =e;
      Result =1;
      For (; e;e--) result =result *base;
}
```

Create a class called box that stores the dimensions of a square. Create an inserter that displays a square box on the screen. (Use any method you like to display the box).

Tạo một lớp gọi 1 cái hộp chứa kích thước của hình vuông. Tạo bộ chèn để hiện thị hình lập phương ra màn hình. (Sử dụng bất kỳ phương thức nào để hiện thị hộp)

Cumulative Skills checks- Kiểm tra kỹ năng tích lũy

This section checks how well you have integrated material in this chapter with that from the preceding chapters.

Trong phần kiểm tra này sẽ chỉ cho bạn cách kết hợp dữ liệu từ chương này với các chương khác trước nó.

Using the Stack class shown here, create an inserter that displays the contents of the stack/ Demonstrate that your inserter works.

Sử dụng lớp stack , tạo bộ chèn để hiện thị các giá trị trong stack .

```
#include <iostream>
using namespace std;
```

```
#define SIZE 10
class stack
{
     char stck[SIZE];
     int tos;
public:
     stack();
     void push( char ch);
     char pop();
};
stack::stack()
{
     tos=0;
}
void stack::push(char ch)
{
     if(tos==SIZE)
     {
          cout<<"Stack is full \n";
          return;
     }
     stck[tos]=ch;
     tos++;
}
char stack::pop()
```

```
{
      if (tos==0)
      {
            cout<<"stack is empty\n";
            return;
      }
      tos--;
      return stck[tos];
}
```

write a program that contains a class called watch. Using the standard time functions, have this class's constructor read the system time and store it. Create an inserter that displays the time.

Viết chương trinh chứa lớp đồng hồ. Sử dụng hàm thời gian chuẩn, có một lớp dựng để đọc và chứa giờ hệ thống. Tao một hàm chèn để hiện thị thời gian.

Using the following class, which converts feet to inches, create an extractor that prompts the user for feet. Also, create an inserter that displays the number of feet and inches. Include a program that demonstrates that your inserter and extractor work.

Sử dụng lớp dưới đây,lớp chuyển tử feet sang inches, tạo một hàm nhập nhắc nhở người dùng nhập dạng feet. Và tạo một bộ chèn để hiện thị số dạng feet và inches. Bao gồm cả chương trình mô tả hoạt động của hàm nhập và bộ chèn .

```
Class ft_to_inches
{
      double feet;
      Double inches;
Public:
      Void set ( double f)
      {
            feet=f;
```

```
            Inches= f*12;
      }
};
```

CHAPTER 9

ADVANCE C++ I/O – NHẬP XUẤT NÂNG CAO CỦA C++

Chapter objective

9.1 CREATING YOUR OWN MANIPULATORS
Tạo các thao tác riêng

9.2 FILE I/O BASICS
File I/O cơ bản

9.3 UNFORMATTED BINARY I/O
I/O nhị phân không định dạng

9.4 MORE UNFORMATTED I/O FUNCTIONS

Những hàm I/O không định dạng tiếp theo

9.5 RANDOM ACCESS
Truy xuất ngẫu nhiên

9.6 CHECKING I+N THE I/O STATUS
Kiểm tra trạng thái I/O

9.7 CUSTOMIZED I/O AND FILES
Tùy biến I/O và file

This chapter continues the examination of the C++ I/O system. In it you will learn to create your own I/O manipulators and work with files. Keep in mind that the C++ I/O system is both rich and flexible and contains many features. While it is beyond the scope of this book to include all of those features, the most important ones are discussed here. A complete description of C++ I/O system can be found in my book C++. The Complete Reference (Berkely: Osborne/McGraw-Hill)

Chương này tiếp tục xem xét hệ thống nhập/xuất của C++. Ở đây bạn sẽ được học về cách tạo ra các bộ thao tác nhập/xuất của chính mình và cách thực hiện nhập xuất tập tin. Hãy nhớ là hệ thống nhập/xuất của C++ có thật nhiều hàm và chúng rất linh động. Nhưng trong phạm vi có hạn của quyển sách này chúng ta không thể trình bày các hàm nói trên, thay vào đó, một số hàm quan trọng nhất sẽ được đề cập đến. Để được biết đầy đủ về hệ thống này, bạn có thể tìm thấy trong cuốn sách sau đây của tôi: The Complete Reference.

The C++ I/O system described in this chapter reflects the one defined by Standard C++ and is compatible with all major C++ compilers. If you have an older of nonconforming compiler, its I/O system will not have all the capabilities described

here.

Chú ý: Hệ thống nhập/xuất của C++ phản ánh chuẩn trù định của C++ chuẩn và tương thích phần lớn các trình biên dịch C++. Nếu bạn đang sử dụng một phiên bản cũ hay không tương thích thì hệ thống nhập/xuất của phiên bản này sẽ không thể thực hiện được các tính năng sẽ được nói đến dưới đây.

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Trước khi đi vào nội dụng chính của chương, bạn hãy trả lời đúng các câu hỏi và làm các bài tập sau.

Write a program that displays the sentence C++ is fun in a 40-character-wide field using a colon (:) as the fill character.

1. Viết chương trình in câu "C++ is fun" với độ rộng của trường là 40 và sử sử dụng dấu 2 chấm (:) làm ký tự lấp vào.

Write a program that displays the outcome of 10/3 to three decimal places. Use ios member functions to do this.

2. Viết chương trình trình bày kết quả của biểu thức 10/3 với độ chính xác là 3 chữ số sau dấu thập phân. Hãy sử dụng các hàm thành phần của **ios.**

Redo the preceding program using I/O manipulators.

Làm lại bài tập 2 sử dụng bộ thao tác nhập xuất thay cho hàm.

What is an inserter? What is an extractor?

Bộ chèn là gì, bộ chiết là gì?

Given the following class, create an inserter and extractor for it.

Cho lớp như dưới đây hãy tạo bộ chèn và bộ chiết cho nó:

```
class date

{
```

```
        char date[9] ;
public:
        // add inserter and extractor
};
```

What header must be included if your program is to use I/O manipulators that take parameters?

Ta phải nạp tập tin tiêu đề nào để sử dụng được bộ thao tác nhập xuất có tham số?

What predefined streams are created when a C++ program begins execution?

Các luồng nào được tự động mở ra khi thực thi một chương trình C++?

9.1. CREATING YOUR OWN MANIPULATORS – TẠO CÁC THAO TÁC RIÊNG

In addition to overloading the insertion and extraction operators, you can further customize C++'s I/O system by creating your own manipulator functions. Custom manipulators are important for two main reasons. First, a manipulator can consolidate a sequence of several separate I/O operators. For example, it is not uncommon to have situations in which the same sequence of I/O operators occurs frequently within a program. In these cases you can use a custom manipulator to perform these actions, thus simplifying your source code and preventing accidental errors. Second, a custom manipulator can be important when you need to perform I/O operators on a nonstandard device. For example, you could use a manipulator to send control codes to a special type of printer or an optical recognition system.

Ngoài việc quá tải toán tử chèn và trích lược, bạn còn có thể thiết kế hệ thống nhập xuất của C++ theo các hàm thao tác của chính bạn. Việc tạo các thủ tục rất quan trọng bởi hai nguyên do sau. Thứ nhất, 1 thủ tục có thể củng cố một dãy các toán tử khác nhau. Ví dụ, nó sẽ không có gì lạ khi có một số tình huống mà giống nhau xảy ra với các toán tử bên trong chương trình. Trong những trường hợp này bạn có thể dung một thủ tục tùy biến để thực hiện các hành động này, vì đơn giản hóa đoạn mã của bạn và tránh

những lỗi nguy hại. Thứ hai, một thủ tục tùy biến có thể quan trọng khi bạn cần sử dụng các toán tử trên một thiết bị không chuyên dụng. Ví dụ, ban có thể dung thủ tục để gửi một mã điều khiển đến một chiếc máy in loại đặc biệt nào đó hay một hệ thống nhận dạng quang học.

Custom manipulators are a feature of C++ that supports OOP, but they can also benefit programs that aren't object-oriented. As you will see, custom manipulators can help make any I/O-instensive program clearer and more efficient.

As you know, there are two basic types of manipulators: those that operate on input streams and those that operate on output streams. In addition to these two broad categories, there is a secondary division: those manipulators that take an argument and those that don't. There are some significant differences between the way a parameterless manipulator and a parameterized manipulator are created. Further, creating parameterized manipulator is substantially more difficult than creating parameterless ones and is beyond the scope of this book. However, writing your own parameterless manipulators is quite easy and is examined here.

All parameterless manipulator output functions have this skeleton:

Các thủ tục tùy biến là một tính năng của C++ mà hỗ trợ cho OOP, nhưng chúng cũng có lợi cho những chương trình mà không hướng đối tượng. Như bạn sẽ thấy, các thủ tục tùy biến có thể giúp bạn tạo ra bất kỳ chương trình mạnh mẽ một cách rõ ràng hơn và hiệu quả hơn.

Như bạn biết, có hai loại thủ tục cơ bản : thủ tục điều khiển những luồng vào và thủ tục điều khiển những luồng ra. Ngoài hai loại này, còn có một cách chia thứ hai: các thủ tục mà mang một đối số và không mang đối số. Có hai sự khác nhau ở thủ tục không có tham số và có tham số là bị tạo. Hơn nữa việc tạo thủ tục có tham số thì khó hơn tạo ra thủ tục không tham số và nằm ngoài phạm vi đề cập của sách. Tuy nhiên, tự viết các thủ tục không tham số khá dễ và được đề cập ở đây.

Tất cả hàm thủ tục khuyết tham số đều có khung sườn như sau:

```
ostream &manip-name(ostream &stream)
{
    // your code here
    Return stream;
}
```

Here mani-name is the name of the manipulator and stream is a reference to the

invoking stream. A preference to the stream is returned. This is necessary if a manipulator is used as part of a larger I/O expression. It is important to understand that even though the manipulator has as its single argument a preference to the stream upon which it is operating, no argument is used when the manipulator is called in an output operation.

All parameterless input manipulator functions have this skeleton:

Ở đây many-name là tên của thủ tục và stream là sự tham khảo cho dẫn chứng luồng. Một sự ưu tiên cho luồng là trả về. Thật cần thiết nếu một toán tử được dùng như một phần của việc biểu diễn nhập xuất lớn hơn. Quan trọng là phải hiểu rằng mặc dù thủ tục có đối số ưu tiên cho luồng mà nó điều khiển, thì không đối số được dùng khi thủ tục gọi việc xuất.

Tất cả thục tục không đối số có dạng sau:

```
istream &manip-name(istream &stream)
{
        // your code here
        Return stream;
}
```

An input manipulator receives a reference to the stream on which it was invoked. This stream must be returned by the manipulator.

Một bộ thao tác nhập nhận một tham chiếu đến luồng mà nó thao tác trên đó. Luồng này phải được trả về bởi bộ thao tác.

It is crucial that your manipulators return a reference to the involking stream. If this is not done, your manipulators cannot be used in sequence of input or output operations.

EXAMPLES

*As a simple first example, the following program creates a manipulator called setup( ) that sets the field width to 10, the precision to 4, and the fill character to *.*

Trong ví này, chương trình tạo ra một bộ thao tác mới tên là setup(). Khi được gọi bộ thao tác này sẽ thiết lập độ rộng trường là 10, độ chính xác là 4, và ký tự lấp đầy là *.

```
#include <iostream>

using namespace std;


ostream &setup( ostream &stream)

{

     Stream.width(10);

     Stream.precision(4);

     Stream.fill('*');


     Return stream;

}


int main()

{

     Cout << setup << 123.123456;


     Return 0;

}
```

As you see, setup is used as part of an I/O expression in the same way that any of the built –in manipulators would be used.

Bạn thấy đó, setup được dùng như một phần của nhập xuất theo cách mà các bộ thao

tác xây dựng được sử dụng.

Custom manipulators need not be complex to be useful. For example, the simple manipulators atn() and note(), shown here, provide a shorter way to output frequently use words or phrases.

Tạo thủ tục không cần phải phức tạp mà phải hữu dụng. Ví dụ, thủ tục atn( ) và note() trình bày ở đây, đơn giản là cung cấp một cách ngắn để xuất mà dùng dùng các từ hay cụm từ.

```
#include <iostream>

using namespace std;


// Attention:

ostream &atn:ostream &stream

{

     Stream << "Attention: ";

     Return stream;

}


// Please note:

ostream &note(ostream &stream)

{

     stream << "Please note: ";

     return stream;

}


int main()

{

     cout << atn << "High voltage circuit \n";
```

```
     cout << note << "Turn off all lights\n";


     return 0;
}
```

Even though they are simple, if used frequently, these manipulators save you from some tedious typing.

Mặc dù chúng đơn giản nhưng nếu sử dụng nhuần nhuyễn thì chúng sẽ giúp bạn tránh được nhiều phiền toái.

This program creates the get pass() input manipulator, which rings the bell and then prompts for a password:

Chương trình này tạo ra thủ tục nhập getpass(), mà nó sẽ rung chuông báo và nhắc password

```
#include <iostream>
#icnlude <cstring>
using namespace std;


// a simple input manipulator
istream &getpass(istream &stream)
{
     cout << '\a' ; // sound bell
     cout << "enter password: ";


     return stream;
}


int main()
```

```
{
      char pw[80];



      do
      {
            cin >> getpass >> pw;
      }while( strcmp(pw, "password"));


      cout <<  "log on complete\n";


      return 0;
}
```

EXERCISES

Create an output manipulator that displays the current system time and date. Call this manipulator td()

Tạo thủ tục xuất ra hệ thống ngày giờ hiện hành, mang tên td()

Create an output manipulator called sethex() that sets output to hexadecimal and turns on the uppercase and showbase flags. Also, create an output manipulator called reset() that undoes the changes made by sethex().

Tạo một thủ tục xuất mang tên sethex() mà xuất ra hệ thập lục phân và chuyển chữ thành hoa và xem cờ. Đồng thời tạo một thủ tục xuất mang tên reset mà không làm thay đổi sethex()

Create an input manipulator called skipchar() that read and ignores the next ten characters from the input stream.

Tạo một thủ tục nhập mang tên skipchar() mà đọc và bỏ qua 10 ký tự tiếp theo của chuỗi nhập.

9.2. FILE I/O BASICS – NHẬP XUẤT FILE CƠ BẢN

It is now time to turn our attention to file I/O. As mentioned in the preceding chapter, file I/O and console I/O are closely related. In fact, the same class hierarchy that supports console I/O also supports file I/O. Thus, most of what you have already learned about applies to files as well. Of course, file handling makes use of several new features.

Bây giờ hãy hướng sự quan tâm đến nhập xuất file. Như đã đề cập trong chương trước, nhập xuất file và bàn điều khiển nhập xuất gần giống nhau. Thực ra, lớp phân cấp hỗ trợ cho bàn điều khiển thì cũng hỗ trợ cho file. Vì vậy, hầu như những gì bạn đã học được đều áp dụng tốt cho file. Tất nhiên, kiểm soát file cần sử dụng them một vài tính năng mới.

To perform file I/O, you must include the header <fstream> in your program. It defines several classes, including ifstream, ofstream, and fstream. These classes are derived from istream and ostream. Remember, istream and ostream are derived from ios, so ifstream, ofstream, and fstream also have access to all operations defined by ios (disccused in the preceding chapter).

Để xử lý trên file bạn cần thêm vào header <fstream> trong chương trình của mình. Nó định nghĩa một vài lớp, bao gồm cả ifstream, ofstream, and fstream. Những lớp này xuất phát từ istream và ostream. . Nhớ là, istream và ostream xuất phát từ ios, như vậy ifstream, ofstream, and fstream cũng có thể truy xuất đến bất kỳ phương thức nào được định nghĩa bởi ios (đã bàn trong chương trước).

In C++, a file is opened by linking it to a stream. There are three types of streams: input, output, and input/output. Before you can open a file, you must first obtain a stream. To create an input stream declare an object of type ifstream. To create an output stream declare an object of type ofstream. Streams that will be performing both input and output operations must be declared objects of type fstream. For example, this fragment creates one input stream, one output stream, and one stream capable of both input and output.

Trong C++, một file được mở bằng liên kết đến 1 luồng. Có ba loại luồng là: nhập, xuất hay cả hai. Trước khi bạn có thể mở 1 file, bạn phải quan sát 1 luồng trước tiên. Để tạo 1 luồng nhập thì đặt 1 đối tượng kiểu ifstream. Để tạo 1 luồng xuất thì đặt 1 đối tượng kiểu ofstream. Những luồng này sẽ thực thi trên cả hai loại phương thức phải được định nghĩa như một đối tượng kiểu fstream. Ví dụ, đoạn sau tạo ra 1 luồng nhập và 1 luồng xuất và một cái tích hợp cả hai loại nhập và xuất.

```
Istream in;     // input

Ostream out;    // output

Fstream io;     // input and output
```

Once you have created a stream, one way to associate it with a file is by using the function open(). This function is a member of each of the three file stream classes. The prototype for each is shown here:

Một khi bạn tạo ra một luồng, một cách để tiếp cận file băng hàm open(). Hàm này là một thành viên của 1 trong 3 lớp file luồng. Cách thức làm như sau:

```
void ifstream::open(const char* filename, openmode
mode=ios::In;

void ofstream::open(const char* filename, openmode
mode=ios::out | ios::trunc);

void fstream::open(const char* filename, openmode mode=
ios::in | ios::out);
```

Here filename is the name of the file, which can include a path specifier. The value of mode determines how the file is opened. It must be a value of type openmode, which is an enumeration defined by that contains the following values:

Ở đây, filename là tên file, bao gồm đường dẫn phải được chỉ rõ. Giá trị của mode quyết định file được mở ra sao. Nó phải là giá trị dạng openmode, mà được liệt kê theo sau đây:

```
ios::app

ios::ate

ios::binary

ios::in

ios::out

ios::trunc
```

You can combine two or more of these values by ORing them together. Let's see what each of these values means.

Bạn có thể kết hợp 2 hay nhiều giá trị này với nhau. Hãy xem ý nghĩa của các giá trị này.

Including ios::app causes all output to that file to be appended the end. This value can be used only with files capable of output. Including ios::ate causes a seek to the end of the file to occur when the file is opened. Although ios::ate causes a seek to end-of-file, I/O operations can still occur anywhere within the file.

Việc bao hàm ios::app làm cho việc xuất vào 1 file sẽ được thêm vào cuối file. Giá trị này có thể dùng khi có khả năng xuất file. Bao hàm ios::ate gây ra một sư tìm kết thúc file ngay khi mở file. Mặc dù ios::ate gây ra 1 sự tìm kết thúc file nhưng các thao tác xuất nhập vẫn có thể thực thi tại bất kỳ đâu bên trong file.

The ios::in value specifies that the file is capable of input. The ios::out value specifies that the file is capable of output.

Giá trị ios::in chỉ ra file có khả năng nhập. Giá trị ios::out chỉ ra file có khả năng xuất.

The ios::binary value causes a file to be opened in binary mode default, all files are opened in text mode. In text mode, various character translations might take place, such as carriage return/linefeed sequences being converted into newlines. However when a file is opened in binary mode, no such character translation will occur. Keep in mind that any file, whether it contains formatted text or raw data, can be opened in either binary or text mode. The only difference is whether character translations take place.

Giá trị ios::binary khiến cho 1 file bị mở trong chế độ nhị phân, mặc định tất cả các file được mở ở dạng text. Trong dạng text, nhiều sự chuyển mã ký tự có thể xảy ra, như dãy các dòng trả về sẽ bị chuyển thành dòng mới. Tuy nhiên khi 1 file được mở trong dạng nhị phân thì chuyện đó không xảy ra. Hãy nhớ rằng bất kỳ file nếu chứa dạng văn bản hay dữ liệu thô, có thể mở được ở cả hai dạng nhị phân và text. Sự khác biệt duy nhất là sự chuyển mã xảy ra.

The ios::trunc value causes the contents of preexisting file by the same name to be destroyed and the file to be truncated to zero length. When you create an output stream using ofstream, any preexisting file with the same name is automatically truncated.

Giá trị ios::trunc làm cho nội dung của file tồn tại trước đó mang cùng tên bị phá hủy và cắt cụt chiều dài thành 0. Khi bạn tạo 1 luồng xuất dùng ofstream, bất kỳ file nào bạn tạo ra trước đó tự động bị cắt cụt.

The following fragment opens an output file called test:

Đoạn sau mở 1 file xuất mang tên là test

```
ofstream mystream;

mystream.open("test");
```

Since the mode parameter to open() defaults to a value appropriate the type of stream being opened, there is no need to specify its value in the preceding example.

Vì dạng tham số hàm open() mặc định là 1 giá trị tương ứng với kiểu luồng được mở, nên không cần chỉ rõ giá trị trong ví dụ trước.

If open() fails, the stream will evaluate to false when used in a Boolean expression. You can make use of this fact to confirm that the open operation succeeded by using a statement like this:

Nếu open() thất bại, luồng sẽ đánh giá sai khi dùng 1 biểu thức Boolean. Bạn có thể dùng sự thực này để kiểm chứng việc mở file thành công bằng câu lệnh sau:

```
if(mystream) {
     cout << "Cannot open file.\n";
     // handle error
}
```

In general, you should always check the result of a call to open() before attempting to access the file.

You can also check to see if you have successfully opened a file by using the is_open() function, which is a member of fstream, ifstream, and ofstream. It has this protptype:

Tóm lại bạn nên kiểm tra xem có mở được file không trước khi thử xử lý file.

Bạn cũng có thể kiểm tra xem có mở được file không bằng hàm is_open(), đây là hàm thành viên của of fstream, ifstream, and ofstream. Nó có dạng sau:

```
bool is_open();
```

It returns true if the stream is linked to an open file and false otherwise. For example, the following checks if mystream is currently open:

Nó trả về true nếu liên kết được đến file và false khi ngược lại. Ví dụ, sau khi kiểm tra

nếu mystream hiện được mở:

```
If ( !mystream.is_open()) {
     Cout << "File is not open.\n";
//…
}
```

Although it is entirely proper to open a file by using the open() function, most of the time you'll not do so because the ifstream, ofstream, and fstream classes have constructor functions that automatically open the file. The constructor functions have the same parameters and defaults as the open() function. Therefore, the most common way you will see a file opened is shown in this example:

Mặc dù hoàn toàn có thể mở 1 file bằng hàm open(), nhưng hầu hết thời gian bạn sẽ không làm vậy vì các lớp ifstream, ofstream, and fstream đều có hàm tạo tự động mở file. Hàm tạo này có những tham số và mặc định giống với hàm open(). Vì vậy, cách chung là :

```
ifstream mystream("my file"); //open file for input
```

As stated, if for some reason the file can not be opened, the stream variable will evaluate as false when used in a conditional statement. Therefore, whether you use a constructor function to open the file or an explicit call to open(), you will want to confirm that the file has actually been opened by testing the value of the stream.

Như đã nói, nếu vì lý do nào đó file không mở được, thì biến của luồng sẽ mang giá trị false khi dùng trong lời phát biểu điều kiện. Vì vậy, nếu bạn dùng 1 hàm tạo để mở file thay cho open(), bạn sẽ muốn xác nhận rằng file thực sự đã mở được bằng việc kiểm tra giá trị của luồng.

To close a file, use the member function close(). For example, to close the file linked to a stream called mystream, use this statement:

Để đóng một file, dùng hàm thành viên close(). Ví dụ, đóng liên kết từ đến luồng gọi mystream, theo cách sau

```
mystream.close();
```

The close() function takes no parameters and returns no value.

You can detect when the end of an input file has been reached by using the eof() member function of ios. It has this prototype:

Hàm close() không có tham số và giá trị trả về. Bạn có thể kiểm tra kết thúc file nhập bằng việc dùng hàm thành viên eof(). Nó có dạng sau:

```
bool eof();
```

It returns true when the end of file has been encountered and false otherwise.

Nó sẽ trả về true khi gặp kết thúc file và false trong trường hợp còn lại.

Once a file has been opened, it is very easy to read textual data from it or write formatted, textual data to it. Simply use the << and >> operators the same way you do when performing console I/O, except that instead of using cin and cout, substitute a stream that is linked to a file. In a way, reading and writing files by using >> and << is like using C's fprintf() and fscanf() functions. All information is stored in the file in the same format it would be in if displayed on the screen. Therefore, a file produced by using << is a formatted text file, and any file ready by >> must be a formatted text file. Typically, files that contain formatted text that you operate on using >> and << operators should be opened for text rather than binary mode. Binary mode is best used on unformatted files, which are described later in this chapter.

Một khi file được mở, thì dễ dàng đọc được dữ liệu nguyên văn trong nó hay viết dữ liệu được định dạng vào nó. Đon giản với toán tử >> và << giống như cách mà bạn thao tác với bảng điều khiển nhập xuất, ngoại trừ thay vì dùng cin và cout, thay thế cho luồng mà liên kết 1 file. Theo cách này, đọc và viết files dùng >> hay << giống như trong các hàm của C là fprintf() và fscanf().

Tất cả dữ liệu lưu trữ trong file giống định dạng mà nó được in ra màn hình. Vì vậy, viết vào file bằng << thì phải là file dạng text chuẩn, và bất kỳ file nào dùng >> thì phải là dạng file chuẩn. Điển hình là, các file mà chứa văn bản được định dạng mà bạn thao tác bằng các toán tử >> và << nên được mở ở dạng text tốt hơn mở bằng dạng nhị phân. Dạng nhị phân chỉ dùng tối ưu cho các file chưa rõ định dạng sẽ được nói đến trong chương sau.

EXAMPLES

Here is a program that creates an output file, writes information to it, closes the file and opens it again as an input file, and reads in the information:

Đây là một chương trình tạo một file xuất, ghi thông tin lên đó, đóng file lại rồi mở lại file một lần nữa như là một file để nhập, và đọc dữ liệu trong file này.

```
#include <iostream>
```

```
#include <fstream>
using namespace std;

int main()
{
     ofstream fout("test") ; //create output file

     if (!fout)     {
          cout << "Can not open output file. \n";
          return 1;
     }
     fout << "Hello! \n";
     fout << 100 << ' ' << hex << 100 << endl;

     fout.close();

     ifstream fin("test"); // open input file

     if (!fin) {
          cout << "Can not open input file.\n";
          return 1;
     }

     char str[80];
     int i, j;
```

```
    fin >> str >> i >> j;
    cout << str << ' ' << i << ' ' << j << endl;

    fin.close();

    return 0;
}
```

After you run this program, examine the content of test. It will contain the following:

Sau khi bạn chạy chương trình, kiểm tra nội dung của việc chạy này. Nó sẽ có kết quả như sau:

```
Hello!
100 64
```

As stated earlier, when the << and >> operators are used to perform file I/O, information is formatted exactly as it would appear on the screen.

Trong trạng thái trước, khi mà toán tử << và >> được dùng để thực hiện file I/O, thông tin được định dạng chính xác như là khi chúng xuất hiện trên màn hình sau đó.

Following is another example of disk I/O. This program reads strings entered at the keyboard and writes them to disk. The program stops when the user enters as $ as the first character in a string. To use the program, specify the name of the output file on the command line.

Dưới đây là một ví dụ khác của i/O. Chương trình này đọc những chuỗi nhập vào từ bàn phím và ghi chúng vào ổ dữ liệu. Chương trình dừng khi người dùng nhấn $ như là kí tự đầu tiên trong một chuỗi. Khi sử dụng chương trình, chỉ rõ tên của file xuất trên dòng lệnh bắt đầu.

```
#include <iostream>

#include <fstream>

using namespace std;
```

```
int main(int argc, char *argv[])
{
     if (argc!=1)   {
          cout << "Usage: WRITE <filename> \n";
          return 1;
     }

     ofstream out(argv[0]) ; // output file

     if (!out) {
          cout << "Can not open output file. \n";
          return 1;
     }

     char str[80];
     cout << "Write strings to disk, '$' to stop\n";

     do   {
          cout << ": ";
          cin >> str;
          out << str << endl;
     }while(*str != '$');

     out.close();
```

```
    return 0;
}
```

Following is a program that copies a text file and, in the process, converts all spaces into | symbols. Notice how eof() is used to check for the end of the input file. Notice also how the input stream fin has its skipws flag turned off. This prevents leading spaces from being skipped.

Chương trình dưới đây sao chép một file văn bản, và trong quá trình thực thi, chuyển đổi tất cả các chỗ trống thành ký tự |. Ghi chú: hàm eof() dùng để kiểm tra xem có đọc hết file chưa. Để ý them là cờ fin và skipws trong dòng thực thi được tắt như thế nào. Sự ngăn cách bởi khoảng trắng được bỏ qua.

```
// Convert space to |s.
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
     if (argc!=3)   {
          cout <<"Usage: CONVERT <input> <output> \n";
          return 1;
     }

     ifstream fin(argv[1]) ; // open input file
     ofstream fout(argv[2]) ; // create output file

     if (!fout)     {
          cout << "Can not open output file. \n";
          return 1;
```

```
    }

    if (!fin) {
        cout << "Can not open input file. \n";
        return 1;
    }

    char ch;

    fin.unsetf(ios::skipws); // do not skip spaces

    while (!fin.eof())  {
        fin >> ch;
        if(ch == ' ')  ch = '|';
        if( !fin.eof()) fout << ch;
    }

    fin.close();
    fout.close();

    return 0;
}
```

There are a few differences between C++'s original I/O binary and the modern Standard C++ library that you should be aware of, especially if you are converting older code. First, in the original I/O library, open() allowed a third parameter, which specified the file's protection mode. This parameter defaulted to a normal

file. The modern I/O library does not support this parameter.

Có một vài sự khác biệt giữa thư viện I/O nhị phân gốc và thư viện chuẩn hiện đại trong C++ mà bạn nên chú ý, đặc biệt là khi bạn chuyển đổi các đoạn mã viết theo phong cách cũ. Trước hết, trong thư viên I/O gốc, hàm open() phụ thuộc vào tham số thứ 3, tham số này sẽ chỉ rõ trạng thái bảo vệ của file(trạng thái thực thi). Tham số này được xác lập mặc định cho file bình thường. Còn trong thư viện I/O mới thì hoàn toàn không hỗ trợ cho tham số này.

Second, when you are using the old library to open a stream for input and output using fstream, you must explicitly specify both the ios::in and the ios::out mode values. No default value for mode supplied. This applies to both the fstream constructor and to its open() function. For example, using the old I/O library you must use a call to open() as shown here to open a file for input and output:

Điều thứ 2 là khi bạn bạn sử dụng thư viện cũ để mở một dòng thực thi để nhập và xuất sử dụng dstream, bạn phải chĩ rõ giá trị mô tả của cả ios::in và ios::out. Không có giá trị mặc định nào được cung cấp. Áp dụng này được dùng cho cả phương thức khởi tạo fstream và hàm open() của nó. Ví dụ như, sử dụng thư viện I/O cũ bạn phải gọi hàm open() để chỉ rõ đâu là file nhập đâu là file xuất.

```
fstream mystream;

mystream.open("test", ios::in | ios::put);
```

In the modern I/O library, an obj of type fstream automatically opens files for input and output when the mode parameter is not supplied.

Trong thư viện I/O mới, một đối tượng của fstream tự động mở file để nhập và xuất khi mà đối số "trạng thái " không được hỗ trợ.

Finally, in the old I/O system, the mode parameter could also include either ios::nocreate, which causes the open() function to fail if the file does not already exist, or ios::noreplace, which causes the open() function to fail if the file does already exist. These values are not supported by Standard C++.

Cuối cùng, trong hệ thống I/O cũ, tham số trạng thái cũng chỉ có thể bao gồm ios::nocreate- là nguyên nhân làm cho hàm open() báo lỗi khi mà file không thực sự tồn tại- hoặc là ios::noreplace- là nguyên nhân làm cho hàm open() báo lỗi khi mà file thực sự tồn tại. Những giá trị này không được hỗ trợ trong thư viện C++ chuẩn.

EXERCISE

Write a program that will copy a text file. Have this program count the number of characters copied and display this result. Why does the number displayed differ from thet shown when you list the output file in the directory?

Viết chương trình sao chéo nội dung của file văn bản. Đếm số lượng các ký tự đã sao chép và hiện thị kết quả. Giải thích tại sao số lượng hiện thị lại khác so với khi bạn duyệt file xuất trong thư mục.

Write a program that writes the following table of information to a file called phone:

Viết một chương trình ghi dữ liệu trong bảng dưới vào file các cuộc đã gọi.

```
Isaac Newton, 415 555-3423

Robert Goddard, 213 555-2312

Enrico Fermi, 202 555-1111
```

Write a program that counts the number of words in a file. For simplicity, assume that anything surrounded by whitespace is a word.

Viết chương trình đến số lượng từ trong file. Để đơn giản, giả sử rằng mỗi từ cách nhau một khoảng trắng.

What does is_open() do?

Hàm is_open() thực thi cái gi?

9.3. UNFORMATTED, BINARY I/O - NHẬP XUẤT NHỊ PHÂN KHÔNG ĐỊNH DẠNG

Although formatted text files such as those produced by the preceding examples are useful in a variety of situations, they do not have the flexibility of unformatted, binary files. Unformatted files contain the same binary representation of the data as that used internally by your program rather than the human-readable text that data is translated into by the << and >> operators. Thus, unformatted I/O is also referred to as " raw " I/O. C++ supports a wide range of unformatted file I/O functions. The unformatted functions give you detailed control over how files are written and read.

Mặc dù file văn bạn đã được định dạng trong có vẻ dài trong những ví dụ trước thì rất hiệu dụng trong nhiều trường hợp, nhưng nó lại không có sự uyển chuyển khi mà văn

bản không được định dạng, chẳng hạn như file nhị phân.File không định dạng chứa dữ liệu được biểu diễn dạng nhị phân giống như dữ liệu được đọc bởi chương trình khó hơn là cách người đọc văn bản mà sử dụng toán tử >> và << để dịch dữ liệu. Vì vậy, Việc I/O không định dạng cũng được quy vào như là "nguyên gốc". C++ hỗ trợ cách thức để giải quyết các file không định dạng I/O bằng nhiều hàm. Hàm không định dạng sẽ cho bạn sự điều khiển chi tiết để biết cách đọc và ghi một file.

*The lowest-level unformatted I/O function are **get()** and **put()**. You can read a byte by using **get()** and write a byte by using **put()**. These functions are members of all input and output stream classes, respectively. The **get()** function has many form, but the most commonly used version is shown here, along with **put()**:*

Cấp thấp nhất của các hàm không định dạng I/O là get() và put(). Bạn có thể đọc một byte bằng cách dùng hàm get() và ghi một byte bằng hàm put().Những hàm là thành phần của các lớp nhập xuất theo dòng, theo thứ tự định sẵn. Hàm get() có nhiều hình thức thể hiện, nhưng có một hình thức được dùng phổ biến nhất được thể hiện dưới đây, đi kèm là hàm put():

```
istream &get(char &ch);

 ostream &put(char ch);
```

*The **get()** function reads a single charater from the associated stream and puts that value in ch. It returns a reference to the stream. If a read is attempted at end-of-file, on return the invoking stream will evaluate to false when used in an expression. The **put()** function writes ch to the stream and returns a reference to the stream.*

Hàm get() đọc một ký tự từ dòng thực thi và đặt giá trị đó vào ch. Hàm trả về một tham chiếu của dòng thực thi. Nếu nó đọc đến hết file, nó sẽ yêu cầu dòng thực thi trả về giá trị sai khi nằm trong một biểu thức. Hàm put() ghi ch vào dòng thực thi và trả về một tham chiếu cho dòng thực thi.

*To read and write blocks of data, use the **read()** and **write()** function, which are also members of the input and output stream classes, respectively. Their prototypes are shown here:*

Để đọc và ghi một khối dữ liệu, dùng hàm read() và write(), là hàm thành viên của lớp nhập xuất theo dòng. Ví dụ như:

```
istream &read(char *buf, streamsize num);

ostream &write(const char *buf, streamsize num);
```

*The **read** function reads num bytes from the invoking stream and puts them in the*

*buffer pointed to by buf. The **write()** function writes num bytes to the associated stream from the buffer pointed to by buf. The **streamsize** type is some form of interger. An obj of type **streamsize** is capable of holding the largest number of bytes that will be transferred in any one I/O operation.*

Hàm read() đọc một số byte từ dòng thực thi và đặt chúng vào vùng đệm được trỏ đến bởi buf. Hàm write() ghi một số byte vào dòng thực thi kết hợp từ vùng đệm được trỏ đến bởi buf. Loại streamsize thì trả về một giá trị nguyên, một đối tượng của streamsize có khả năng lưu giữ một lượng lớn các byte mà được vận chuyển trong bất kỳ thao tác I/O nào.

*If the end of the file is reached before num characters have been read, **read()** simply stops, and the buffer contains as many characters as were available. You can find out how many charaters have been read by using the member function **gcount()**, which has this prototype:*

Khi đến cuối file trước khi số lượng ký tự được đọc, hàm read() sẽ dừng lại, và vùng đệm sẽ chứ các ký tự có nghĩa. Bạn có thể tìm ra nhiều ký tự được đọc bằng cách sử dụng hàm thành viên gcount, được minh họa dưới đây:

```
streamsize gcount();
```

It returns the number of characters read by the last unformatted input operation.

Nó trả về số lượng các ký tự được đọc bởi thao tác đọc không định dạng.

*When you are using the unformatted file functions, most often you will open a file for binary rather than text operations. The reason for this is easy to understand: specifying **ios::binary** prevents any character translations from occurring. This is important when the binary representations of data such as integers, floats, and pointers are stored in the file. However, it is perfectly acceptable to use the unformatted functions on a file opened in text mode-as long as that file actually contains only text. But remember, some character traslations may occur.*

Khi mà bạn sử dụng hàm đọc file không định dạng, hầu hết là dùng để mở file nhị phân hơn là mở file văn bản. Lí do thật là dễ hiểu: chỉ rõ **ios::binary** ngăn sự xuất hiện của bất kì kí tự dịch nào. Điều này là quan trọng khi dạng nhị phân của dữ liệu như là số nguyên, số thực, và con trỏ được lưu trữ trong tập tin. Tuy nhiên, có thể chấp nhận được khi dùng hàm không định dạng trong một tập tin ở dạng văn bản-đủ dài để tập tin đó chứa duy nhất văn bản. Nhưng nhớ rằng, một số kí tự dịch có thể xuất hiện.

EXAMPLES

*The next program will display the contents of any file on the screen. It uses the **get()** function.*

Chương trình tiếp theo sẽ trình bày nội dung của bất kì tập tin trên màn hình. Nó dùng hàm **get()**.

```
#include <iostream>

#include <fsream>

using namespace std;


int main(int argc, char *argv[])

{

     char ch;


     if(argc!=2)

     {

          cout<<”Usage: PR <filename>\n”;

          return 1;

     }


     ifstream in(argv[1], ios::in | ios::binary);

     if(!in)

     {

          cout<<”Cannot open file.\n”;

          return 1;

     }
```

```
        while(!in.eof())
        {
                in.get(ch);
                cout<<ch;
        }

        in.close();

        return 0;
}
```

*This program uses **put()** to write characters to a file until the user enters a dollar sign:*

Chương trình này sử dụng **put()** để ghi kí tự cho tập tin cho đến khi gặp kí hiệu đôla:

```
        #include <iostream>
        #include <fstream>
        using namespace std;

        int main(int argc, char *argv[])
        {
                char ch;

                if(argc!=2)
                {
                        cout<<"Usage:WRITE <filename>\n";
                        return 1;
                }
```

```
        ofstream out(argv[1], ios::out | ios::binary);

        if(!out)
        {
             cout<<"Cannot open file.\n";
             return 1;
        }

        cout<<"Enter a $ to stop.\n";
        do
        {
             cout<<";";
             cin.get(ch);
             out.put(ch);
        }while(ch!='$');

        out.close();

        return 0;
   }
```

*Notice that the program uses **get()** to read characters from **cin**. This prevents leading spaces from being discarded.*

Chú ý rằng chương trình này dùng **get()** để đọc kí tự từ **cin**. Điều này ngăn khoảng đầu bị loại bỏ.

*Here is a program that uses **write()** to write a **double** and a string to a file called **test**:*

Đây là một chương trình sử dụng **write()** để ghi **double** và một chuỗi vào một file gọi là **test:**

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

int main()
{
     ofstream out("test", ios::out | ios::binary);

     if(!out)
     {
          cout<<"Cannot open output file.\n";
          return 1;
     }

     double num=100.45;
     char str[]="This is a test";

     out.write((char *) &num, sizeof(double));
     out.write(str, strlen(str));

     out.close();
```

```
    return 0;
}
```

NOTE: *The type cast to* ***(char *)*** *inside the call to* ***write()*** *is necessary when outputting a buffer that is not defined as a character array. Because of C++'s strong type checking, a pointer of one type will not automatically be converted into a pointer of another type.*

Loại phù hợp (char *) trong lời gọi **write()** là cần thiết khi xuất ra một bộ nhớ trung gian không được định nghĩa như là một mảng kí tự. Bởi vì loại kiểm tra mạnh của C++, một con trỏ của một loại sẽ không tự động chuyển thành con trỏ của loại khác.

This program uses ***read()*** *to read the file created by the program in Example 3:*

Chương trình này sử dụng **read()** để ghi file được tạo bởi chương trình trong ví dụ 3:

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream in("test", ios::in | ios::binary);

    if(!in)
    {
        cout<<"Cannot open input file.\n";
        return 1;
    }
```

```
    double num;
    char str[80];

    in.read((char *)&num, sizeof(double));
    in.read(str, 14);
    str[14]='\0'; // null terminate str

    cout<<num<<' '<<str;

    in.close();

    return 0;
}
```

*As is the case with the program in the preceeding example, the type cast inside **read()** is necessary because C++ will not automatically convert a pointer of one type to another.*

Khi trường hợp với chương trình ở ví dụ trước, loại phù hợp bên trong **read()** là cần thiết vì C++ sẽ không tự động chuyển đổi một con trỏ của một loại sang cái khác.

*The following program first writes an array of **double** values to a file and then reads them back. It also reports the number of characters read.*

Chương trình dưới đây đầu tiên sẽ ghi một mảng của giá trị **double** vào một tập tin và đọc nó trở lại. Nó cũng báo cáo số kí tự đọc.

```
// Demonstrate gcount().

#include <iostream>

#include <fstream>

using namespace std;
```

```
int main()
{
     ofstream out("test", ios::out | ios::binary);

     if(!out)
     {
          cout<<"Cannot open output file.\n";
          return 1;
     }

     double num[4]={1.1, 2.2, 3.3, 4.4};

     out.write((char *) nums, sizeof(nums));
     out.close();

     ifstream.in("test", ios::in | ios::binary);

     if(!in)
     {
          cout<<"Cannot open input file.\n";
          return 1;
     }

     in.read((char *) &nums, sizeof(nums));
```

```
        int i;
        for(i=0; i<4; i++)
              cout<<nums[i]<<' ';
        cout<<'\n';

        cout<<in.gcount()<<"characters read\n";
        in.close();

        return 0;
}
```

EXERCISES (BÀI TẬP)

*Rewrite your answers to exercises 1 and 3 in the preceeding section (Section 9.2) so that they use **get()**, **put(), read()**, and/or **write()**.(Use whichever of these function you deem most appropriate.)*

Viết lại câu trả lời của bài tập 1 và 3 trong phần trước(9.2) để dùng **get(), put(), read(),** và/hay **write()**.(Dùng bất cứ hàm nào bạn cho là thích hợp nhất.)

Given the following class, write a program that outputs the contents of the class to a file. Create an inserter function for this purpose.

Cho lớp dưới đây, viết một chương trình mà xuất ra nội dung của lớp ra tập tin. Tạo một hàm chèn cho mục đích này.

```
class account
{
        int custnum;
        char name[80];
        double balance;
```

```
    public
        account(intc, char *n, double b)
        {
            custnum=c;
            strcpy(name, n);
            balance=b;
        }
    // create inserter here
};
```

9.4. MORE UNFORMATTED I/O FUNCTIONS - THÊM MỘT SỐ HÀM NHẬP/XUẤT KHÔNG ĐƯỢC ĐỊNH DẠNG

*In addition to get the form shown earlier, there are several different ways in which the **gets()** function is overloaded. The prototypes for the three most commonly used overloaded forms are shown here:*

Thêm vào một số hình thức đã được đưa ra trước đây, có một vài cách khác trong đó, chức năng **gets()** được nạp chồng. Đầu tiên là 3 dạng nạp chồng phổ biến được đưa ra ở đây:

```
istream&get(char*buf, streamsize num);
istream &get(char* buf, streamsize num, char delim);
int get();
```

*The first form reads characters into the array pointed to by buf until either num-l characters have been read, a new line is found, or the end of the file has been encountered. The array pointed to by buf will be null terminated by **get().** If the new line character is encountered in the input stream, it is **not** extracted. Instead, it is remains in the stream until the next input operation.*

Dạng đầu tiên đọc những kí tự vào trong một mảng được chỉ định bởi *buf* cho đến khi

num-l đã được đọc, hoặc gặp một dong mới hay là kết thúc một file. Mảng được trỏ tới bởi *buf* sẽ được vạch giới hạn kết thúc bởi **get()**. Nếu như phát hiện kí tự của dòng mới trong stream input (tra cứu trong MSDN để hiểu thêm), thì nó không được trích ra. Thay vào đó, nó được lưu lại cho đến khi gặp thao tác nhập tiếp theo.

*The second form reads characters into the array pointed to by buf until either num-I characters have been read the character specified by delim has been found, or the end of the file has been encountered. The array pointed to by buf will be null terminated by **get().** If the delimiter character is encountered in the input stream, it is not extracted. Instead, it remains in the stream until the next input operation.*

Dạng thứ 2 đọc kí tự vào trong một mảng được trỏ bởi buf cho đến khi num-l kí tự được đọc cho đến khi gặp kí tự được xác định bởi delim, hoặc là kết thúc một file. Mảng được trỏ đến bởi buf sẽ được vạch giới hạn kết thúc bởi **get().** Nếu như bắt gặp kí tự phân cách trong stream input, thì nó không được trích xuất nữa. Thay vào đó, nó được duy trì trong chuỗi cho đến khi gặp thao tác input kế tiếp.

*The third overloaded form of **get()** returns the next character from the stream. It returns EOF if the end of the file is encountered. This form of **get()** is similar to C's **getc()** function.*

Dạng thứ 3 của **get(),** trả về kí tự kế tiếp từ trong stream. Nó trả về EOF nếu như nó gặp kết thúc một file. Dạng này của **get()** thì giống như hàm **getc()** trong C.

*Another function that performs input is **getline().** It is a member of each input stream class. Its prototypes are shown here:*

Hàm khác mà thực thi input là **getline().** Nó là một thành phần của mỗi lớp stream input. Nguyên mẫu của nó thì được đưa ra ở đây:

```
istream &getline(char *buf, streamsize num);

istream &getline(char*buf,streamsize num, char
delim);
```

*The first form reads characters into the array pointed to by buf until either num-i characters have been read, a new line character has been found, or the end of the file has been encouraged. The array pointed to by buf until be null terminated **getline().** If the new line character is encountered in the input stream, it is extracted, but it is not put into buf.*

Dạng đầu tiên đọc những kí tự vào trong mảng được trỏ bởi *buf* cho đến khi hoặc là *num-i* kí tự đã được đọc, hoặc là gặp một kí tự xuống dòng hay là kết thúc một file.

Mảng được trỏ bởi buf được vạch giới hạn kết thúc bởi **getline()**. Nếu như bắt gặp kí tự xuống dòng trong stream input thì nó được trích xuất nhưng nó không được đưa vào trong *buf.*

The second form reads characters into the array pointed to by buf until either num-I characters have been read, the character specified by delim has been found, or the end of the file has been encountered. The array pointed to by buf will be null terminated be ***getline().*** *If the delimiter character is encountered in the input stream, it is extracted, but is not put into buf.*

Dạng thứ 2 đọc các kí tự vào trong mảng được trỏ đến bởi *buf* cho đến khi hoặc là đọc *num-i* kí tự đã được đọc, hoặc là gặp kí tự được xác định bởi *delim*, hoặc là gặp kí tự kết thúc file. Mảng được trỏ đến bởi *buf* nó sẽ được vạch giới hạn kết thúc bởi **getline()**. Nếu như bắt gặp kí tự phân cách trong stream input thì nó được trích xuất, tuy nhiên không được đưa vào trong mảng *buf.*

As you can see, the two versions of ***getline()*** *are virtually identical to the* ***get(****buf, num) and* ***get(****buf, num, delim) versions of* ***get().*** *Both read characters from input and put them into the array pointed to by buf until either num-I characters have been read or until the delimiter character or the end of the file is encountered. The difference between* ***get()*** *and* ***getline()*** *is that* ***getline()*** *reads and removes the delimiter from the input stream;* ***get()*** *does not.*

Như bạn có thể thấy, 2 phiên bản của **getline()** là hầu như giống hệt với **get(***buf, num)* và **get(***buf, num, delim)* của **get().** Cả hai đều đọc kí tự từ trong input và đưa chúng vào trong mảng được trỏ bởi *buf* cho đến khi *num-i* kí tự được đọc, hoặc là cho đến khi bắt gặp kí tự phân cách, hoặc là khi kết thúc file. Sự khác biệt giữa **get()** và **getline()** là **getline()** đọc và di chuyển dấu tách ra khỏi stream input; còn **get()** thì không.

It returns the next character in the input stream without removing it from that stream by using ***peek()****. This function is a member of the input stream classes and has this prototype:*

int ***peek();***

It returns the next character in the stream; it returns EOF if the end of the file is encountered.

Nó trả về một kí tự kế tiếp trong stream input nhưng không di dời chúng ra khỏi stream đó nếu sử dụng **peek()**. Hàm này là một thành phần trong lớp stream input và có dạng nguyên mẫu:

```
int peek();
```

Nó trả về một kí tự kế tiếp trong stream; nó sẽ trả về EOF nếu như gặp kí tự kết thúc file.

*You can return the last character read from a stream to that stream by using **putback(),** which is a member of the input stream classes. Its prototype is shown here:*

istream &putback(char c);

where c is the last character read.

Bạn có thể trả về kí tự cuối cùng đọc từ một stream đến stream đó nếu sử dụng **putback(),** nó là một thành phần của lớp stream input. Nguyên mẫu của nó là:

```
istream &putback(char c);
```

Nơi mà c là kí tự cuối cùng được đọc.

*When output is performed, data is not immediately written to the physical device linked to the stream. Instead, information is stored in an internal buffer until the buffer is full. Only then are the contents of that buffer written to disk. However, you can force the information to be physically written to disk before the buffer is full by calling **flush(). Flush()** is a member if the out put stream classes and has this prototype:*

ostream &flush();

*Calls to **flush()** might be warranted when a program is going to be used in adverse environments(in situations where power outages occur frequently, for example).*

Khi mà output được thực thi, dữ liệu không được viết ngay lập tức đến link thiết bị vật lí. Mà thay vào đó, thông tin được lưu trữ trong một bộ nhớ đệm bên trong cho đến khi bộ nhớ đến khi bộ nhớ đệm đầy. Ngay sau đó, nội dung của bộ nhớ đệm mới được viêt lên disk. Tuy nhiên, bạn có thể tác động đến thông tin được viết lên disk một cách ngẫu nhiên trước khi bộ nhớ đệm đầy được gọi là **fflush().Flush()** là một thành phần trong lớp stream output và có dạng nguyên mẫu:

```
ostram &fflush();
```

Việc gọi **fflush()** có thể được xác nhận, khi mà một chương trình dự định được sử dụng trong môi trường bất lợi. (trong tình huống nơi cung cấp lực xuất hiện tình trạng thiếu điện thường xuyên, ví dụ)

EXAMPLES (Ví dụ)

*As you know, when you use >> to read a string, it stops reading when the first whitespace character is encountered. This makes is useless for reading a string containing spaces. However, you can overcome this problem by using **getline()**, as this program illustrates:*
*In this program, the delimiter used by **getline()** is the newline. This makes **getline()** act much like the standard **gets()** function.*

1. Như bạn biết, khi mà bạn sử dụng >> để đọc một chuỗi, nó dừng đọc khi gặp kí tự trắng đầu tiên. Phương pháp này thì vô dụng cho việc đọc một chuỗi bao gồm khoảng trắng. Tuy nhiên, bạn có thể khắc phục lỗi này bằng cách sử dụng **getline()**, chương trình sau đây sẽ chứng tỏ nó:

```
//use getline() to read a string that contains spaces

#include <iostream>

#include <fstream>

Using namespace std;

int main()

{
     char str[80];


   cut <<”Enter your name:”;

   cin.getline(str, 79);

   cut <<str<<’\n’;

   return 0;

}
```

*2.In real programming situations, the functions **peek()** and **putback()** are especially useful because they let you more easily handle situations in which you do not know what type of information is being input at any point in time. The following program gives the flavor of this. It reads either strings or integers from a file. The strings*

and integers can occur in any order.

2. Trong những tình huống trong chương trình thật sự, hàm **peek()** và **putback()** thì đặc biệt hữu dụng bởi vì chúng giúp bạn dễ dàng trong các tình huống xử lí mà bạn không biết loại thông tin đang được đưa vào tại một thời điểm bất kì. Chương trình sau minh họa cho điều này. Nó đọc một chuỗi hoặc những số nguyên từ một file. Những chuỗi, và những số nguyên có thể xuất hiện với thứ tự bất kì.

```
//demonstrate peek()

#include <iostream>

#include <fstream>

#include <cctype>

using namespace std;

int main(int argc, char* argv[])

{

      char ch;

      ofstream out ("test", ios::out | ios::binary);

      if(!out)

      {

                cout <<"cannot open output file.\n";

                return 1;

      }

      char str[80], *p;

      out <<123<<"This is a test" <<23;

      out << "Hello there!" << 99<< "sdf" <<endl;


      ifstream in ("test", ios::in | ios::binary);


      if (!in)
```

```
      {
            cout <<"Cannot open input file";
            return 1;
      }

      do
      {
           p= str;
           ch = in.peek(); //see what type of char is text
           if (isdigit(ch))
           {
                if (isdigit (*p=in.get())) p++; //read
integer
                in.putback (*p); //return char to stream
                *p= '\0'; //Null-terminate the string
                cout <<"integer: "<<atoi(str);
           }
           else
                if (isalpha (ch)) //read a string
                {
                     while (isalpha (*p=in.get())) p++;
                     in.putback(*p); //return char to
stream
                     *p= '\0'; //NUll-terminate the string
                     in.get(); //ignore
                     cout << "String: "<<str;
```

```
                }
                else
                        in.get(); //ignore
                        cout <<'\n';
        }
        while (!in.eof());
        in.close();


        return 0;
}
```

EXERCISES: (Bài tập)

*Rewrite the program in Example 1 so it uses **get()** instead of **getline().** Does the program function differently?*
*Write a program that reads a text file one line at a time and displays each line on the screen. Use **getline().***
*On your own, think about why there may be cases in which a call to **flush()** is appropriate.*

Viết lại chương trình ở ví dụ 1 mà sử dụng **get()** thay vì **getline().** Chức năng của chương trình khác nhau phải không?
Viết một chương trình đọc 1 file text một lần một dòng, và xuất mỗi dòng ra màn hình. Sử dụng **getline().**
Theo bạn, nghĩ tại sao có thể có trường hợp mà một lệnh gọi **fflush()** là có thể.

9.5. RANDOM ACCESS - TRUY XUẤT NGẪU NHIÊN

*In C++'s I/O system, you perform random access by using the **seekg()** and **seekp()** functions, which are members of the input and out put stream classes, respectively. Their most common forms are shown here:*

istream &seekg(off_type offset, seekdir origin);

ostream &seekp(off_type offset, seekdir origin);

*Here **off_type** is an integer type defined by **ios** that is capable of containing the largest valid value that offset can have. **seekdir** is an enumeration defined by ios that has these values.*

Value	***Meaning***
ios::beg	*seek from beginning*
ios::cur	*seek from current location*
ios::end	*seek from end*

Trong hệ thống I/O của C++, bạn có thể thực hiện việc truy cập ngẫu nhiên bằng hàm **seekg(),** và **seekp()**, chúng là thành phần của lớp stream input, output, theo một thứ tự đã định sẵn. Những dạng phổ biến của chúng được biểu diễn ở dưới đây:

```
istream &seekg(off_type offset, seekdir origin);

ostream &seekp(off_type offset, seedier origin);
```

Ở đây, **off_type** là một số nguyên được định nghĩa bởi ios, nó thì có những giá trị này.

Giá trị	**Ý nghĩa:**
ios::beg	seek from beginning
ios::cur	seek from current location
ios::end	seek from end

*The C++ I/O system manages two pointers associated with a file. One is the get pointer, which specifies where in the file the next input operation will occur. The other is the put pointer, which specifies where in the file the next output operation will occur. Each time an input or output operation takes place, the appropriate pointer is automatically sequentially advanced. However, by using the **seekg()** and **seekp()** functions, it is possible to access the file in a nonsequential fashion.*

Hệ thống I/O trong C++ quản lí 2 con trỏ kết hợp với 1 file. Một là *con trỏ get,* nó xác

định nơi trong file mà thao tác input tiếp theo sẽ xảy ra. Một loại khác là *con trỏ put,* nó xác định vị trí trong file xảy ra thao tác output kế tiếp xảy ra. Mỗi thời điểm thì một thao tác output hay input diễn ra, con trỏ thích hợp thì tự động di chuyển liên tục. Tuy nhiên, bằng cách sử dụng hàm **seekg()** và **seekp(),** nó có thể truy cập file một cách không liên tục.

*The **seekg()** fuction moves the associated file's current get pointer offset number of bytes from the specified origin. The **seekp()** function moves the associated file's current put pointer offset number of bytes from specified origin.*

*In general, files that will be accessed via **seekg()** and **seekp()** should be opened for binary file operations. This prevents character translations from occurring which may affect the apparent position of an item within a life.*

You can determine the current position if each file pointer by using these member functions:

pos_type tellg();

pos_type tellp();

Hàm **seekg()** di chuyển con trỏ *get* hiện thời của file *offset* byte bắt đầu từ vị trí được chỉ định bởi *origin.* Hàm **seekp()** di chuyển con trỏ put hiện thời trong file offset byte bắt đầu từ vị trí được xác định bởi *origin.*

Nhìn chung, những file mà được truy xuất qua **seekg()** và seekp() nên được mở cho thao tác tệp nhị phân.

Bạn có thể xác định vị trí hiện tại nếu như con trỏ mỗi file sử dụng nhưng hàm thành phần này:

```
Pos_type tellg();

Pos_type tellp();
```

*Here **pos_type** is an integer type defined by **ios** is capable of holding the largest value that defines a file position.*

*There are overloaded versions of **seekg()** and **seekp()** that move the file pointers to the location specified by the return values of **tellg()** and **tellp()**. Their prototypes are shown here:*

istream &seekg(pos_type position)

ostream &seekp(pos_type position)

Ở đây **pos_type** là một số nguyên được xác định bởi **ios** thì có khả năng giữ giá trị lớn nhất xác định vị trí của một file.

Có những phiên bản trùng của **seekg()** và **seekp()** di chuyển con trỏ file đến vị trí được xác định bởi giá tri trả về của **tellg()** và **tellp().** Dạng nguyên mẫu của nó như sau:

```
istream &seekg(pos_type position)

ostream &seekp(pos_type position)
```

EXAMPLES: (Bài tập)

*The following program demonstrate the **seekp()** function. It allows you to change a specific character in a file. Specific a file name on the command line, followed by the number of the byte in the file you want to change, followed by the new character. Notice that the file is opened for read/write operations.*

1. Chương trình sau đây chứng minh hàm **seekp().** Nó cho phép bạn chuyển đổi một kí tự trong file. Xác định một tên file trên một dòng lệnh, sau đó là số byte mà ban muốn chuyển đổi, sau đó là kí tự mới. Chú ý rằng, file được mở cho thao tác đọc hoặc là viết.

```
#include <iostream>

#include <fstream>

#include <cstdlib>

using namespace std;

int main(int argc, char *argv[])

{

     if (argc!=4)

     {

          cout << "Usage: CHANGE<filename><byte><char>\n";

          return 1;

     }
```

```
        fstream out (argv[1], ios::in | ios::out | ios::binary);
        if(!out)
        {
             cout <<"cannot openfile";
             return 1;
        }
        out.seekp(atoi(argv[2]), ios::beg);
        out.close();

        return 0;
    }
```

*The next program uses **seekg()** to position the get pointer into the middle of a file and then displays the contents of that file from that point. The name of the file and the location to begin reading from are specified on the command line.*

2. Chương trình tiếp theo sử dụng hàm **seekg()** để đặt con trỏ get vào giữa một file và sau đó xuất nội dung của file đó ra màn hình từ vị trí con trỏ. Tên của file và vị trí bắt đầu đọc được xác định trên một dòng lệnh.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[])
{
    char ch;
```

```
    if(argc !=3)
    {
        cout << "Usage: LOCATE <filename> <loc>\n";
        return 1;
    }
    ifstream in(argv[1], ios::in |ios::binary);

    if (!in)
    {
        cout <<"Cannot open input file.\n";
        return 1;
    }

    in.seekg(atoi (argv[2]), ios::beg);

    while (!in.eof())
    {
        in.get(ch);
        cout <<ch;
    }
    in.close();

    return 0;
}
```

EXERCISES:

Write program that displays a text file backwards. Hint: think about this before creating your program. The solution is easier than you might imagine.
Write a program that swaps each character pair in a text file. For example, if the file contains "1234", then after the program is run, the file will contain "2143". (for simplicity, you may assume that the file contains an even number of characters.)

1. Viết một chương trình biểu diễn một file text ngược. Gợi ý: Nghĩ về điều này trước khi tạo chương trình của bạn. Phương án giải quyết thì dễ dang hơn bạn tưởng tượng.

2. Viết một chương trình sắp xếp mỗi cặp kí tự trong một file text. Ví dụ, nếu file gồm có "1234", sau khi chương trình chạy, thì file bao gồm "2143", (để đơn giản, bạn có thể cho rằng file bao gồm số các kí tự).

9.6. CHECKING THE I/O STATUS - KIỂM TRA TRẠNG THÁI I/O

The C++ I/O system maintains status information about the outcome of each I/O operation. The current status of an I/O stream is described in an object of type ***iostate,*** *which is an enumeration defined by* ***ios*** *that includes these members:*

Name	***Meaning***
goodbit	*no errors occurred*
eofbit	*End –of-file has been encountered.*
failbit	*a nonfatal I/O error has occurred*
badbit	*a fatal I/O error has been occurred*

Hệ thống I/O trong C++ duy trì thông tin trạng thái về kết quả của mỗi thao tác I/O. Trạng thái hiện tại của một stream I/O được thể hiện trong một đối tượng loại **iostate**, nó là một bảng liệt kê được định nghĩa bởi **ios** bao gồm các thành phần sau đây:

Tên	**Nghĩa**
goodbit	Không có lỗi xuất hiện
eofbit	Kết thúc một file.
failbit	Xuất hiện một lỗi không I/O không nghiêm trọng.
badbit	Xuất hiện một lỗi nghiêm trọng.

*For older compliers, the I/O status flags are held in an **int** rather than an object of type **iostate.***

*There are two ways in which you can obtain I/O status information. First, you can call the **rdstate()** function, which is a member of **ios.** It has this prototype:*

Đối với một chương trình biên dịch cũ, trạng thái cờ được giữ như là một **int** thì tốt hơn là một đối tượng **iostate.**

Có 2 cách mà bạn có thể thu được các thông tin trạng thái. Đầu tiên, bạn có thể gọi hàm **rdstate(),** nó là một thành phần của **ios.** Nó có dạng nguyên mẫu là:

```
Iostate rdstate();
```

*It return the current status of the error flags. As you can probably guess from looking at the preceding list of flags, **sdstate()** returns **goodbit** when no error has occurred. Otherwise, an error flag is returned.*

*The other way you can determine whether an error has occurred is by using one or more of these **ios** member functions.*

bool bad();

bool eof();

bool fail();

bool good();

Nó trả về một trạng thái hiện thời của những cờ báo lỗi. Nhưng hàu như chắc chắn rằng bạn có thể đoán được từ việc quan sát danh sách các cờ có trước, **rdstate()** trả về **goodbit** khi mà không có lỗi nào xuất hiện. Ngược lại, một cờ báo lỗi được trả về.

Cách khác mà bạn có thể xác định có lỗi hay không là sử dụng một hoặc nhiều hơn một

hàm thành phần của **ios.**

```
bool bad();

bool eof();

bool fail();

bool good();
```

*The **eof()** function was discussed earlier. The **bad()** function returns true **badbit** is set. The **good()** function returns true if there are no errors. Otherwise they return false.*

*Once an error has occurred it might need to be cleared before your program continues. To do this, use the **ios** member function **lear(),** whose prototype is shown here:*

void clear(iostate flags =ios::goodbit);

*If flags is **goodbit** (as it is by default), all error flags are cleared. Otherwise, set flags to the settings you desire.*

Hàm **eof()** đã được thảo luận trước đây. Hàm **bad()** trả về true nếu như **badbit** được thiết lập. Hàm **good()** trả về giá trị true nếu như không có lỗi nào. Ngược lại trả về false.

Một khi mà có lỗi xuất hiện, có thể nó cần được xóa trước khi chương trình của bạn tiếp tục. Để làm điều này, sử dụng hàm thành phần **ios** là **clear(),** dạng nguyên mẫu của nó là:

```
void clear(iostate flags=ios::goodbit);
```

Nếu *cờ l‡ **goodbit*** (như mặc định), tất cả những cờ lỗi đều được xóa. Ngược lại, thiết lập *cờ* để bạn chỉnh sữa theo ý muốn.

EXAMPLES: (VÍ DỤ)

*The following program illustrates **rdstate().** it displays the contents of a text file. If an error occurs, the function reports it by using **checkstatus().***

Chương trình sau đây chứng minh **rdstata().** Nó xuất ra màn hình nội dung của một file text. Nếu có một lỗi, thì hàm thông báo nó bằng cách sử dụng **checkstatus().**

```
#include <iostream>
```

```
#include <fstream>
using namespace std;
void checkstatus(ifstream &in);

int main(int argc, char* argv[])
{
     if (argc !=2)
     {
          cout << "Usage: DISPLAY <filename>\n";
          return 1;
     }

     ifstream in(argv[1]);
     if(!in)
     {
          cout <<"Cannot open input file.\n";
          return 1;
     }

     char c;
     while (in.get(c))
     {
          cout <<c;
          checkstatus (in);
     }
```

```
        checkstatus (in); //Check final status
        in.close();

        return 0;
}

void checkstatus (ifstream &in)
{
        ios::iostate i;
        i=in.rdstate();
        if(i&ios::eofbit)
                cout << "EOF encountered\n";
        else
                if (i&ios::failbit)
                        cout << "Non-Fatal I/O error\n";
                else
                        if (i&ios::badbit)
                                cout <<"Fatal I/O error \n";
}
```

*The preceding program will always report at least one "error". After the **while** loop ends, the final call to **checkstatus()** reports, as expected, that an **EOF** has been encountered.*

Chương trình bên trên sẽ luôn thông báo ít nhất một "lỗi". Sau khi kết thúc lặp, cuối cùng gọi **checkstatus()** thông báo là cuối file.

*This program displays a text file. It uses **good()** to detect a file error:*

Chương trình này xuất một file text ra ngoài màn hình. Nó sử dụng **good()** để phát hiện lỗi file:

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char* argv[])
{
      char ch;
      if(argc!=2)
      {
            cout <<"PR:<filename>\n";
            return 1;
      }
      ifstream in(argv[1]);
      if (!in)
      {
            cout <<"Cannot open input file.\n";
            return 1;
      }

      while (!in.eof())
      {
            in.get(ch);
```

```
            //check for error
            if(!in.good() && !in.eof())
            {
                  cout <<"I/O Error ... terminating\n";
                  return 1;
            }
            cout <<ch;
      }
      in.close();

      return 0;
}
```

EXERCISES:

Add error checking to your answers to the exercises from the preceding section.
Thêm vào phần kiểm tra lỗi trong câu trả lời của bạn ở phần trước.

9.7. CUSTOMIZED I/O AND FILES - FILE VÀ I/O THEO YÊU CẦU

In the preceding chapter you learned how to overload the insertion and extraction operators relative to your own classes. In that chapter, only console I/O was performed. However, because all C++ streams are the same, the same overloaded inserter function, for example, can be used to output to the screen or to a file with no changes whatsoever. This is one of the most important and useful features of C++'s approach to I/O.

Ở chương trước bạn đã học cách để nạp chồng toán cử chèn và rút trích liên quan đến những bài học của bạn. Trong chương đó, chỉ có phần I/O được biểu diễn. Tuy nhiên, bởi vì tất cả các stream trong C++ thì tương tự nhau, hàm nạp chồng tương tự lồng vào nhau, ví dụ, có thể được sử dụng để xuất ra ngoài màn hình hoặc đến file mà không thay đổi nó. Điều này là một trong những đặc điểm quan trọng nhất và hữu dụng nhất phương pháp C++ đối với I/O.

As stated in the previous chapter, overloaded inserters and extractors, as well as I/O manipulators, can be used with any stream as long as they are written in a general manner. If you "hard-code" a specific stream into an I/O function, its uses is,of sourse, limited to only that stream. This is why you were urged to generalize your I/O functions whenever possible.

Như trạng thái của chương trước, nạp chồng chèn và trích xuất, là công cụ I/O tốt nhất có thể, có thể được sử dụng với bất kì stream nào có độ dài giống như chúng được viết cách thông thường. Nếu bạn viết một stream cụ thể vào trong một hàm I/O, thì nó được sử dụng, tất nhiên nó chỉ giới hạn trong stream đó. Đây là lí do tại sao bạn được đề xuất khái quát hóa hàm I/O bất cứ khi nào có thể.

EXAMPLES:

*In the following program, the **coord** class overloads the << and >> operators. Notice that you can use the operator functions to write both to the screen and to a file.*

Trong chương trình sau đây, lớp **coord** nạp chồng toán tử << và >>. Chú ý rằng bạn có thể sử dụng hàm toán tử để để viết cả ra màn hình và file.

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;


class coord
{
```

```
     int x, y;
     public:
          coord (int i, int j)
          {
               x= i;
               y= j;
          }
          friend ostream &operator <<(ostream &os, coord
ob);
          friend istream &operator >>(istream &is, coord
&ob);
};



ostream &operator <<(ostream &os, coord ob)
{
     os <<ob.x<<' '<<ob.y<<'\n';
     return os;
}

istream &operator >>(istream &is, coord &ob)
{
     is >>ob.x >>ob.y;
     return is;
}
```

```
int main()
{
     coord o1(1,2), o2(3, 4);
     ofstream out ("test");

     if(!out)
     {
          cout <<"Cannot open output file.\n";
          return 1;
     }
     out <<o1 <<o2;
     out.close();

     ifstream in("test");
     if(!in)
     {
          cout <<"Cannot openfile input file.\n";
          return 1;
     }
     coord o3(0,0), o4 (0,0);
     in>>o3 >>o4;

     cout <<o3 <<o4;
     in.close();
     return 0;
```

```
}
```

All of the I/O manipulators can be used with files. For example, in this reworked version of a program presented earlier in this chapter, the same manipulator that writes to the screen will also write to a file:

2. Tất cả các bộ phận I/O đều có thể được sử dụng với file. Ví dụ, trong bài làm lại của chương trình được biểu diễn ở chương trước, đoạn điều khiển tương tự để viết ra màn hình cũng như viết ra một file:

```
#include <iostream>

#include <fstream>

#include <iomanip>

using namespace std;


//Attention:

ostream &atn (ostream &stream)

{

      stream <<"Attention:";

      return stream;

}


//Please note:

ostream &note(ostream &stream)

{

      stream <<"Please Note:";

      return stream;

}
```

```
int main()
{
      ofstream out ("Test.txt");
      if (!out)
      {
            cout <<"Cannot open output file.\n";
            return 1;
      }

      //Write to screen
      cout <<atn <<"High voltage circuit.\n";
      cout <<note <<"Turn off all lights\n";

      //write to file
      out <<atn <<"High voltage circuit.\n";
      out <<note <<"Turn off all lights\n";

      out.close();
      return 0;
}
```

EXERCISES:

On your own, experiment with the programs from the preceding chapter, trying each on a disk file.

Tự bạn làm thí nghiệm với những chương trình từ các chương trước, thử tạo mỗi cái

với một file trên ổ disk.

SKILLS CHECK: (KIỂM TRA CÁC KĨ NĂNG)

At this point you should be able to perform the following exercises and answer the questions.

Create an output manipulator that outputs three tabs and then sets the field width to 20. demonstrate that your manipulator works.
Create an input manipulator that reads and discards all nonalphabetical characters. When the first alphabetical character is read, have the manipulator return it to the input stream and return. Call this manipulator ***findalpha.***
Write a program that copies a text file. In the process, reverse the case of all letters.
write a program that reads a text file and then reports the number of times each letter in the alphabet occurs in the file.
if you have not done so, add complete error checking to your solutions to Exercises 3 and 4 above.
What function positions the get pointer? What function positions the put pointer?

Vào thời điểm này, bạn nên làm những bài tập sau đây và trả lời các câu hỏi:

Tạo một đoạn chương trình xuất 3 dấu tab, và sau đó mở rộng lên 20. Chứng minh rằng chương trình của bạn hoạt động.
Tạo một đoạn chương trình input đọc và loại bỏ tất cả các kí tự không nằm trong bảng chữ cái alpha. Khi những kí tự alpha đầu tiên được đọc, có một đoạn chương trình trả nó về stream input và trả về. Gọi đoạn chương trình đó là **findalpha.**
Viết một chương trình copy một file text. Trong quá trình thực hiện, đảo ngược tất cả các kí tự.
Viết một chương trình đọc một file text và sau đó thông báo số lần xuất hiện của mỗi kí tự (trong bảng chữ cái alpha) trong file.
Nếu ban chưa làm, thì hãy thêm "hàm kiểm tra lỗi" vào bài 3 và bài 4 bên trên.
Hàm nào dùng để xác định vị trí của một con trỏ get? Hàm nào để xác định vị trí của con trỏ put.

CHAPTER 10

VIRTUAL FUNCTIONS - HÀM ẢO

Chapter objective

10.1. POINTER TO DERIVED CLASSES.

Con trỏ trỏ tới lớp dẫn xuất

10.2. INTRODUCE TO VIRTUAL FUNCTIONS.

Tổng quan về hàm ảo

10.3. MORE ABOUT VIRTUAL FUNCTIONS.

Nói thêm về hàm ảo.

10.4. APPLYING POLYMORPHISM.

Áp dụng đa hình

This chapter examines another important aspect of C++: the virtual function. What makes virtual function important is that they are used to support run–time polymorphism. Polymorphism is supported by C++ in two ways. First, it is supported at compile time, through the uses of overloaded operators and function. Second, it is supported at run time, through the use of virtual function. As you will learn, run-time polymorphism provides the greatest flexibility.

At the foundation of virtual function and run-time polymorphism are pointers to derived classes. For this reason this chapter begins with a discussion of such pointers.

Chương này nghiên cứu vấn đề quan trọng khác của C++: hàm ảo. Cái mà làm cho hàm ảo trở nên quan trọng do nó được dùng để hỗ trợ chạy ở chế độ đa hình. Đa hình được C++ hỗ trợ bằng 2 cách. Thứ nhất, nó được dùng để biên dịch thời gian, xuyên suốt cách dùng toán tử nạp chồng và hàm. Thứ hai, nó được hỗ trợ trong môi trường chạy, thông qua cách dùng của hàm ảo. Khi bạn học, chạy ở chế độ đa hình cung cấp sự uyển chuyển nhất. Tại sự thiết lập của hàm ảo và chạy ở chế độ đa hình là con trỏ trỏ tới lớp dẫn xuất.

Đó là lý do chương này bắt đầu với sự thảo luận về con trỏ.

REVIEW SKILL CHECK (Kiểm tra kĩ năng ôn tập)

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Create a manipulator that causes numbers to be displayed in scientific notation, using a capital E.
Write a program that copies a text file. During the copy process, convert all tabs into the correct number of spaces.
Write a program that searches a file for a word specified on the command line. Have the program display how many times anything surrounded by whitespace is a word.
Show the statement that sets the put pointer to the 234th byte in a file linked to a stream called out.
What function report status information about the C++ I/O system?
Give one advantage of using the C++ I/O functions instead of the C-like I/O system.

Trước khi tiếp tục, bạn nên trả lời chính xác những câu hỏi dưới đây và làm những bài tập sau.

1. Thực hiện một thao tác tạo nên những con số được hiển thị trong kí hiệu khoa học, dùng tiếng Anh chủ yếu.

2. Viết 1 chương trình sao chép một tập tin văn bản. Trong suốt quá trình sao chép, chuyển đổi tất cả tab thành con số chính xác của khoảng trống.

3. Viết một chương trình tìm kiếm một tập tin cho một từ theo lý thuyết trên dòng lệnh. Có một chương trình hiển thị bao nhiêu lần bất cứ cái gì bao quanh bởi khoảng trắng là một từ.

4. Chỉ ra câu lệnh mà thiết lập con trỏ trỏ tới byte thứ 234 trong một tập tin liên kết với một luồng gọi là out.

5. Hàm nào báo cáo trạng thái thông tin về hệ thống nhập xuất C++.

6. Nêu ra sự thuận lợi của việc sử dụng hàm nhập xuất trong C++ thay cho C-giống như hệ thống nhập xuất.

10.1. POINTERS TO DERIVED CLASSES - CON TRỎ TRỎ TỚI LỚP DẪN XUẤT

Although Chapter 4 discussed C++ pointers at some length, one special aspect was deferred until now because it relates specifically to virtual functions. The feature is this: A pointer declared as a pointer to a base class can be used to point to any class derived from that base. For example, assume two classes called base and derived, where derived inherits base. Given this situation, the following statements are correct:

Mặc dù chương 4 thảo luận con trỏ C++ tại một vài độ dài, một diện mạo đặc biệt bị hoãn lại cho đến bây giờ bởi vì nó liên quan rõ ràng tới hàm ảo. Đặc trưng là điều này: một con trỏ hiển thị như một con trỏ trỏ tới lớp cơ sở có thể được dùng trỏ tới bất kì lớp dẫn xuất nào từ cơ sở đó. Ví dụ, thừa nhân 2 lớp gọi là base và derived, nơi mà derived thừa kế base. Trong tình huống này, những câu lệnh dưới đây là đúng:

```
base *p; // base class pointer

base base_ob; // obj of type base

derived derived_ob; // obj of type


// p can, of course, point to base obj

p=&base_ob; // p points to base obj


// p can also point to derived obj without error

p=&derived_ob; // p points to derived obj
```

As the comments suggest, a base pointer can point to an obj of any class derived from that base without generating a type mismatch error.

Although you can use a base pointer to point to a derived obj, you can access only those members of the derived obj that were inherited from the base. This is because the base pointer has knowledge only of the base class. It knows nothing about the members added by the derived class.

While it is permissible for a base pointer to point to a derived obj, the reverse is not true. A pointer of the derived type cannot be used to access an obj of the base class. (A type cast can be used to overcome this restriction, but its use is not recommended

practice.)

One final point: Remember that pointer arithmetic is relative to the data type the pointer is declared as pointing to. Thus, if you point a base pointer to a derived obj and then increment that pointe, it will not be pointing to the next derived obj. It will be pointing to(what it thinks is) the next base obj. Be careful about this.

Như lời nhận xét gợi ý, một con trỏ cơ sở có thể trỏ tới một đối tượng của lớp dẫn xuất bất kì nào từ cơ sở đó không có phát sinh loại lỗi không khớp.

Tuy bạn có thể dùng con trỏ cơ sở để trỏ đến đối tượng dẫn xuất, bạn chỉ có thể truy xuất những thành phần của đối tượng dẫn xuất mà thừa kế từ cơ sở. Điều này bởi vì con trỏ cơ sở chỉ có thông tin của lớp cơ sở. Nó không biết về những thành viên thêm vào bởi lớp dẫn xuất.

Trong khi nó dùng được cho con trỏ cơ sở trỏ tới đối tượng dẫn xuất, sự đảo ngược thì không đúng. Một con trỏ của loại dẫn xuất không thể được dùng để truy xuất một lớp đối tượng của lớp cơ sở. (Một loại phù hợp có thể được dùng để vượt qua sự hạn chế này, nhưng cách dùng của nó không đề cập tới lệ thường.)

Một điểm cuối cùng: nhớ rằng con trỏ số học loại dữ liệu mà con trỏ hiển thị như là trỏ đến. Như vậy, nếu nó không trỏ đến đối tượng dẫn xuất tiếp theo. Nó sẽ trỏ đến (điều nó nghĩ là) đối tượng tiếp theo. Hãy cẩn thận về điều này.

EXAMPLE (VÍ DỤ)

1. *Here is a short program that illustrates how a base class pointer can be used to access a derived class:*

1. Đây là chương trình ngắn minh họa một lớp cơ sở có thể được dùng để truy xuất lớp gốc như thế nào:

```
// Demonstrate pointer to derrived class.

#include <iostream>

using namespace std;


class base

{
```

```
     int x;
     public:
          void setx(int i)
               x=i;
          int getx()
               return x;
};

class derived:public base
{
     int y;
     public:
          void sety(int y)
               y=i;
          int gety()
               return y;
};

int main()
{
     base *y; // pointer to bse type
     base b_ob; // obj of base
     derived d_ob; // obj of derived

     // use p to access base obj
```

```
    p=&b_ob;
    p->setx(10); //access base obj
    cout<<"Base obj x:"<<p->getx()<<'\n';

    // use p to access derived obj
    p=&d_ob; // point to derived obj
    p->setx(99); // access derived obj

    // can't use p to set y, so do it directly
    d_ob.sety(88);
    cout<<"Derived obj x:"<<p->getx()<<'\n';
    cout<<"Derived obj y:"<<d_ob.gety()<<'\n';

    return 0;
}
```

Aside from illustrating pointers to derived classes, there is no value in using a base class pointer in the way shown in this example. However, in the next section you will see base class pointers to derived obj are so important.

Ngoài việc minh họa con trỏ trỏ tới lớp dẫn xuất, không có giá trị nào trong cách sử dụng con trỏ lớp cơ sở ở cách được trình bày trong ví dụ này. Tuy nhiên, trong đoạn tiếp theo bạn sẽ thấy lớp con trỏ cơ sở trỏ tới đối tượng dẫn xuất quan trọng như thế nào.

EXERCISE (BÀI TẬP)

On your own, try the preceding example and experiment with it. For example, try declaring a derived pointer and having it access an obj of the base class.

Bạn hãy tự thử ví dụ trước and học hỏi kinh nghiệm từ nó. Ví dụ, thử trình bày một con

trỏ gốc và yêu cầu nó truy xuất một đối tượng của lớp cơ sở.

10.2. INTRODUCTION TO VIRTUAL FUNCTIONS – TỔNG QUAN VỀ HÀM ẢO

A virtual function is a member function that is declared within a base class redefined by a derived class. To create a virtual function, precede the function's declaration with the keyword virtual. When a class containing a virtual function is inherited, the derived class redefines the virtual function relative to the derived class. In essence, virtual function implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method. When a virtual is redefined by a derived class, the keyword virtual is not needed.

Một hàm ảo là một hàm thành phần mà được hiển thị trong vòng một lớp cơ sở được định nghĩa lại bởi một lớp dẫn xuất. Để tạo một hàm ảo, đến trước sự khai báo của hàm ảo với một từ khóa ảo. Khi mà một lớp chứa một hàm ảo được thừa kế, lớp dẫn xuất định nghĩa lại hàm ảo liên quan tới lớp dẫn xuất. Thực chất, hàm ảo thực thi triết lý "một giao diện, nhiều phương pháp" tồn tại dưới dạng đa hình. Hàm ảo trong vòng lớp cơ sở định nghĩa dạng của giao diện đối với hàm đó. Mỗi sự định nghĩa lại của hàm ảo bởi một lớp dẫn xuất thực thi thao tác của nó như là nó liên quan đặc biệt tới lớp dẫn xuất. Điều đó là, sự định nghĩa lại tạo một phương pháp rõ ràng. Khi mà hàm ảo được định nghĩa lại bởi một lớp dẫn xuất, từ khóa ảo là không cần thiết.

A virtual function can be called just like any other member function. However, what makes a virtual function interesting-and capable of supporting run-time polymorphism-is what happens when a virtual function is called through a pointer. From the preceding section you know that a base class pointer can be used to point to a derived class obj. When a base pointer points to a derived obj that contains a virtual function and that virtual functions is called through that pointer, C++ determines which version of that function will be executed based upon the type of obj being pointed to by the pointer. And, this determination is made at run time. Put differently, it is the type of the obj pointed to at the time when the call occurs that determines which version of the virtual function will be executed. Therefore, if two or more different classes are derived from a base class that contains a virtual function, then when different obj are pointed to by a base pointer, different versions of the virtual function are executed. This processs is the

way that run-time polymorphism is achieved. In fact, a class that contains a virtual function is referred to as a polymorphic class.

Một hàm ảo có thể được gọi như là bất kì hàm thành phần nào khác. Tuy nhiên, điều làm hàm ảo thú vị-và có khả năng hỗ trợ chạy ở chế độ đa hình-là điều xảy ra khi hàm ảo được gọi thông qua con trỏ. Từ phần trước bạn biết rằng một con trỏ lớp cơ sở có thể trỏ đến đối tượng lớp dẫn xuất. Khi mà một con trỏ cơ sở trỏ đến đối tượng dẫn xuất chứa một hàm ảo và hàm ảo đó được gọi thông qua con trỏ đó, C++ xác định rõ phiên bản nào của hàm đó sẽ được chạy dựa trên loại của đối tượng đang được trỏ đến bởi con trỏ. Và sự xác định này được làm trên môi trường chạy. Đặt khác nhau, nó là loại của đối tượng trỏ đến tại thời điểm mà lời gọi xuất hiện xác định phiên bản nào của hàm ảo sẽ được thực hiện. Vì thế, nếu 2 hay nhiều lớp khác nhau được dẫn xuất từ một lớp cơ sở, phiên bản khác nhau của hàm ảo được thực hiện. Quá trình này là cách chạy ở chế độ đa hình đạt được. Thật sự, một lớp chứa một hàm ảo được cho là như lớp đa hình.,

EXAMPLES (VÍ DỤ)

Here is a short example that uses a virtual function:

Đây là một ví dụ ngắn mà sử dụng hàm ảo:

```
// A simple example using a virtual function.

#include <iostream>

using namespace std;


class base

{

     public:

          int i;

          base(int x)

               i=x;

          virtual void func()

          {
```

```
                cout<<"Using base version of func():";
                cout<<"i<<'\n';
          }
};

class derived1:public base
{
     public:
          derived1(int x):base(x)
     void func()
     {
          cout<<"Using derived1's version of func():";
          cout<<i*i<<'\n';
     }
};

class derived2:public base
{
     public:
          derived2(int x):base(x)
          void func()
          {
                cout<<"Using derived2's version of func():";
                cout<<i+i<<'\n';
          }
```

```
};

int main()
{
     base *p;
     base ob(10);
     derived1 d_ob1(10);
     derived2 d_ob2(10);

     p=&ob;
     p->func(); // use base's func()

     p=&d_ob1;
     p->func(); // use derived1's func()

     p=&d_ob2;
     p->func(); // use derived2's func()

     return 0;
}
```

This program displays the following output:

Chương trình này trình bày xuất:

```
Using base version of func():10
Using derived1's version of func():100
```

```
Using derived2's version of func():20
```

The redefinition of a virtual function inside a derived class might, at first, seem somewhat similar to function overloading. However, the two processes are distinctly different. First, an overloaded function must differ in type and/or number of parameters, while a redefined vrtual function must have precisely the same type and number of parameters and the same return type. (In fact, if you change either the number or type of parameters when redefining and its virtual nature is lost.) Further, virtual functions must be class members. This is not the case for overloaded functions. Also, while destructor functions can be virtual, constructors cannnot. Because of the differences between overloaded functions are redefined virtual functions, the term overriding is used to describe virtual function redefinition.

Sự định nghĩa lại của hàm ảo bên trong lớp dẫn xuất có thể, trước tiên, dường như điều gì đó giống với nạp chồng hàm. Tuy nhiên, hai quá trình khác nhau rõ rệt. Thứ nhất, nạp chồng hàm phải khác loai và/hay số của tham số, trong khi hàm ảo được định nghĩa lại phải có cùng loại và số của tham số và cùng loại trả về. (Thật sự, nếu bạn thay đổi cả số hay loại của tham số khi định nghĩa lại và ảo tự nhiên mất đi.) Rộng hơn, hàm ảo phải là lớp thành phần. Đây không là trường hợp cho nạp chồng hàm. Cũng vậy, trong khi hàm hủy có thể là ảo, hàm thiết lập thì không. Bởi vì sự khác nhau giữa nạp chồng hàm và hàm ảo định nghĩa lại, số hạng ghi đè được dùng để diễn tả hàm ảo đươc định nghĩa lại.

As you can see, the example program creates three classes. The base class defines the virtual function func(). This class is then inherited by both derived1 and derived2. Each of these classes overrides func() with its individual implementation. Inside main(), the base class pointer p is declared along with obj of type base, devired1, and devired2. First, p is assigned the address of ob (an obj of type base). When func() is called by using p, it is the version in base that is used. Next, p is assigned the address of d_ob1 and func() is called again. Because it is the type of the obj pointed to that determines which virtual function will be called, this time it is the overridden version in derived1 that is executed. Finally, p is assigned the address of d_ob2 and func() defined inside derived2 that is executed.

Như bạn thấy, chương trình ví dụ tạo ra 3 lớp. Lớp cơ sở định nghĩa hàm ảo func(). Lớp này được thừa kế bởi cả hai derived1 và derived2. Mỗi lớp này ghi đè func()sự thực thi riêng lẻ của nó. Trong hàm main(), con trỏ lớp cơ sở p được hiển thị song song với đối tương của loại cơ sở, derived1, và derived2. Thứ nhất, p được gán địa chỉ của ob (một đối tượng của loại cơ sở). Khi func() được gọi bởi dùng p, nó là phiên bản cơ sở được dùng. Tiếp theo, p được gán địa chỉ của d_ob1 và func() và được gọi lần nữa. Bởi vì nó là loại của đối tượng trỏ tới xác định rằng hàm ảo nào sẽ được gọi, lần này nó sẽ được ghi đè phiên bản derived1 mà được thi hành. Cuối cùng, p được gán địa chỉ của d_ob2 và func() định nghĩa trong derived2 mà được thi hành.

The key points to understand from the preceding example are that the type of the obj being pointed to determines which version of an overridden virtual function will be executed when accessed via a base class pointer, and that this decision is made at run time.

Từ khóa trỏ tới hiểu từ ví dụ trước là loại của đối tượng được trỏ tới xác định phiên bản nào của hàm ảo ghi đè sẽ thực thi khi truy xuất qua con trỏ lớp cơ sở, và quyết định này được làm trong môi trường chạy.

Virtual functions are hierarchical in order of inheritance. Futher, when a derived class does not override a virtual function, the function defined within its base class is used. For example, here is a slightly different version of the preceding program:

Hàm ảo có thứ tự thừa kế. Rộng hơn, khi một lớp dẫn xuất không ghi đè lên hàm ảo, hàm định nghĩa trong vòng lớp cơ sở của nó được dùng. Ví dụ, đây là phiên bản khác nhau nhỏ của chương trình trước:

```
// Những hàm ảo thì có thứ bậc.

#include <iostream>

using namespace std;


class base {

public:

     int i;

     base(int x) { i = x; }

     virtual void func()

     {

     cout << "Using base version of func(): ";

     cout << i << "\n";

     }

};
```

```
class derived1 : public base {
public:
     derived1(int x) : base(x) {}
     void func()
     {
     cout << "Using derived1's version of func(): ";
     cout << i * i << "\n";
     }
};

class derived2 : public base {
public:
     derived2(int x) : base(x) {}
     // derived2 không nạp chồng hàm func()
};

int main()
{
     base *p;
     base ob(10);
     derived1 d_ob1(10);
     derived2 d_ob2(10);
     p = &ob;
     p->func(); // sử dụng hàm func() của lớp base
```

```
        p->&d_ob1;
        p->func(); // sử dụng hàm func() của lớp derived1
        p->&d_ob2;
        p->func(); // sử dụng hàm func() của lớp derived2

        return 0;
}
```

This program displays the following output:

Chương trình này trình bày xuất:

```
Using base version of func(): 10
Using derived1 version of func(): 100
Using derived2 version of func(): 20
```

In this version, derived2 does not override func(). When p is assigned d_ob2 and func() is called, base's version is used because it is next up in the class hierarchy. In general, when a derived class does not override a virtual function, the base class version is used.

Trong phiên bản này, derived2 không ghi đè lên func(). Khi p được gán d_ob2 và func() được gọi, phiên bản cơ sở được dùng vì nó tiếp theo trong lớp có thứ tự. Nói chung, khi lớp dẫn xuất không ghi đè lên hàm ảo, phiên bản lớp cơ sở được dùng.

The next example shows how a virtual function can respond to random events that occur at run time. This program selects between d_ob1 and d_ob2 based upon the value returned by the standard random number generator rand(). Keep in mind that the version of func() executed is resolved at run time. (Indeed, it is impossible to resolve the calls to func() at compile time.)

Ví dụ tiếp theo cho thấy hàm ảo có thể phản hồi sự kiện ngẫu nhiên mà xảy ra tại môi trường chạy như thế nào. Chương trình này chọn giữa d_ob1 và d_ob2 dựa trên giá trị trả về bởi số khởi xướng ngẫu nhiên chuẩn tại môi trường chạy. (Thực vậy, bất khả thi để giải quyết lời gọi func() tại thời điểm biên dịch.)

```
/* This example illustrates how a virtual function can be
used to respond to random events occurring at run time.
*/

/* Ví dụ này là điển hình như thế nào là một hàm ảo có
thể được sử dụng để đáp trả lại những sự kiện ngẫu nhiên
xảy ra lúc thực thi.
*/

#include <iostream>
using namespace std;

class base {
public:
     int i;
     base(int x) { i = x; }
     virtual void func()
     {
     cout << "Using base version of func(): ";
     cout << i << "\n";
     }
};

class derived1 : public base {
public:
     derived1(int x) : base(x) {}
```

```
        void func()
        {
        cout << "Using derived1's version of func(): ";
        cout << i*i << "\n";
        }
};

class derived2 : public base {
public:
        derived2(int x) : base(x) {}
        void func()
        {
        cout << "Using derived2's version of func(): ";
        cout << i+i << "\n";
}};

int main()
{
        base *p;
        base ob(10);
        derived1 d_ob1(10);
        derived2 d_ob2(10);
        int i, j;

        for(i=0; i<10; i++) {
```

```
        j = rand();
        if((j%2)) p = &d_ob1; // nếu d_ob1 lẻ
        else p = &d_ob2; // nếu d_ob2 chẵn
        p->func(); // gọi hàm thích hợp
    }

    return 0;
}
```

Here is a more practical example of show a virtual function can be used. This program creates a generic base class called area that holds two dimensions of a figure. It also declares a virtual function called getarea() that, when overridden by derived classes, returns the area of the type of figure defined by the derived class. In this case, the declaration of getarea() inside the base class determines the nature of the interface. The actual implementation is left to the classes that inherit it. In this example, the area of a triangle and a rectangle are computed.

Thêm ví dụ thực tập cho thấy hàm ảo có thể được sử dụng như thế nào. Chương trình này tạo một lớp cơ sở có đặc điểm chung gọi là khu vực giữ hai kích thước của hình dạng. Nó cũng hiển thị một hàm ảo gọi là getarea(), khi ghi đè bởi lớp dẫn xuất, trả về cùng của loại của getarea() bên trong lớp cơ sở xác định rõ sự tự nhiên của giao diện. Sự thi hành thật sự mà rời khỏi lớp mà thừa kế nó. Trong ví dụ này, vùng của một tam giác và một hình chữ nhật và một tam giác được tính toán.

```
// Sử dụng hàm ảo để định nghĩa giao diện
#include <iostream>
using namespace std;

class area {
    double dim1, dim2; // những kích thước hình
public:
    void setarea(double d1, double d2)
```

```
        {
            dim1 = d1;
            dim2 = d2;
        }
        void getdim(double &d1, double &d2)
        {
        d1 = dim1;
        d2 = dim2;
        }
        virtual double getarea()
        {
        cout << "You must override this fuction\n";
        return 0.0;
        }
};

class rectangle : public area {
public:
        double getarea()
        {
            double d1, d2;
        getdim(d1, d2);
        return d1 * d2;
        }
};
```

```
class triangle : public area {
public:
     double getarea()
     {
          double d1, d2;

     getdim(d1, d2);
     return 0.5 * d1 * d2;
     }
};

int main()
{
     area *p;
     rectangle r;
     trisngle t;

     r.setarea(3.3, 4.5);
     r.setarea(4.0, 5.0);

     p = &r;
     cout << “Rectangle has area: ”
     << p->getarea() << “\n”;
```

```
        p = &t;
        cout << "Triangle has area: "
        << p->getarea() << "\n";

        return 0;
}
```

Notice that the definition of getarea() inside area is just a placeholder and perform no real function. Because area is not linked to any specific type of figure, there is no meaningful definition that can be given to getarea() inside area. In fact, getarea() must be overridden by a derived class in order to be useful. In the next section, you will see a way to enforce this.

Chú ý rằng sự định nghĩa của getarea() bên trong vùng là chỉ là trình giữ chỗ và không thực hiện hàm thực sự. Bởi vì vùng không liên kết tới bất kì loại rõ ràng nào của hình dạng, không có định nghĩa đầy đủ mà có thể được getarea() bên trong vùng. Thật sự, getarea() phải được ghi đè bởi lớp dẫn xuất để hữu dụng. Trong phần tiếp theo, bạn sẽ thấy cách khác để thi hành điều này.

EXERCISES (BÀI TẬP)

1. *Write a program that creates a base class called num. Have this class hold an integer value and contain a virtual function called shownum(). Create two derived classes called outhex and outoct that inherit num. Have the derived classed override shownum() so that it displays the value in hexadecimal and octal, respectively.*

Viết một chương trình tạo một lớp cơ sở gọi là num. Có lớp này giữ một giá trị nguyên và chứa một hàm ảo gọi là shownum(). Tạo 2 lớp dẫn xuất gọi là outhex và outoct mà thừa kế num. Có lớp dẫn xuất ghi đè shownum() để nó trình bày giá trị trong hệ thập lục phân và hệ bát phân, theo thứ tự định sẵn.

2. *Write a program that creates base class called dist that stores the distance between two points in a double variable. In dist, create a virtual function called trav_time() that outputs the time it takes to travel that distance, assuming that the distance is in miles and the speed is 60 miles per hour. In a derived class called metric, override trav_time() so that it outputs the travel time assuming that the distance is in*

kilometers and the speed is 100 kilometers per hour.

Viết một chương trình mà tạo lớp cơ sở gọi là dist mà lưu trữ khoảng cách giữa 2 điểm trong một cặp giá trị. Trong dist, tạo một hàm ảo gọi là trav_time() mà xuất thời gian nó lấy để vượt qua khoảng cách đó, cho rằng khoảng cách là dặm và tốc độ là 60 dặm/giờ. Trong lớp dẫn xuất gọi là hệ mét, ghi đè trav_time() vì thế nó xuất thời gian di chuyển cho rằng khoảng cách là km và tốc độ là 100km/h.

10.3. MORE ABOUT VIRTUAL FUNCTIONS - NƠI THÊM VỀ HÀM ẢO

As Example 4 from the preceding section illustrates, sometimes when a virtual function is declared in the base class there is no meaningful operation for it to perform. This situation is common because often a base class does not define a complete class by itself. Instead of, it simply supplies a core set of number functions and variables to which the derived class supplies the remainder. When there is no which the derived class supplies the remainder. When there is no meaningful action for derived class virtual function to perform. The implication is that any derived class must override this function. To ensure that this will occur, C++ supports pure virtual functions.

A pure virtual function has no definition relative to the base class. Only the function's prototype is included. To make a pure virtual function, use this general form:

Như ví dụ 4 từ phần trước đã minh họa, thỉnh thoảng khi một hàm ảo được minh họa trong lớp cơ sở không có thao tác nào có ý nghĩa để thực hiện. Tình huống này là phổ biến vì thường một lớp cơ sở không định nghĩa một lớp đầy đủ bởi chính nó. Thay vào đó, nó chỉ đơn giản cung cấp một lõi thiết lập của hàm số và biến số cho lớp dẫn xuất cung cấp cho phần còn lại. Khi không có hành động đầy đủ cho hàm ảo lớp dẫn xuất thực hiện. Sự liên quan là bất kì lớp dẫn xuất nào phải ghi đè lên hàm này. Chắc chắn rằng điều này sẽ xuất hiện, C++ hỗ trợ hàm ảo nguyên mẫu.

Một hàm ảo nguyên mẫu không có định nghĩa liên quan tới lớp cơ sở. Chỉ có mẫu ban đầu của hàm là bao gồm cả. Để tạo một hàm ảo nguyên mẫu, dùng dạng chung này:

```
virtual type func-name(parameter-list) = 0
```

The key part of this declaration is the setting of the function equal to 0. This tells the compiler that no body exists for this function relative to the base class. When a virtual function is made pure, it forces any derived class to override it. If a derived class does

not, a compile-time error results. Thus, making a virtual function pure is a way to guarantee that a derived class will provide its own redefinition.

Từ khóa của khai báo này là thiết lập hàm bằng 0. Điều này bảo trình biên dịch rằng không có cái nào tồn tại cho hàm này liên quan tới lớp cơ sở. Khi một hàm ảo được làm cho nguyên mẫu, nó phá bất kì lớp dẫn xuất nào để ghi đè lên nó. Nếu lớp dẫn xuất không như vậy, chạy biên dịch cho kết quả lỗi. Vì vậy, tạo một hàm ảo nguyên mẫu là cách đảm bảo rằng lớp dẫn xuất sẽ cung cấp định nghĩa lại của chính nó.

When a class contains at least one pure virtual function, it is referred to as an abstract class. Since an abstract class contains at least one function for which no body exists, it is, technically, an incomplete type, and no objects of that class can be created. Thus, abstract classes exist only to be inherited. They are neither intended nor able to stand alone. It is important to understand, however, that you can still create a pointer to an abstract class, since it is through the use of base class pointers that run-time polymorphism is achieved. (It is also permissible to have a reference to an abstract class.)

Khi một lớp chứa ít nhất một hàm nguyên mẫu, nó xem như là một lớp trừu tượng. Khi một lớp trừu tượng chứa ít nhất một hàm mà không có cái nào tồn tại, nó là, một cách kĩ thuật, một loại không hoàn thành, và không có đối tượng của lớp có thể được tạo. Vì vậy, lớp trừu tượng tồn tại chỉ để thừa kế. Nó không dự định không thể đứng một mình. Thật là quan trọng để hiểu, tuy nhiên, bạn có thể cũng tạo một con trỏ cho lớp trừu tượng , từ khi nó thông qua cách dùng của con trỏ lớp cơ sở chạy ở chế độ đa hình được hoàn thành. (Nó cũng chấp nhận được khi có một tham chiếu cho một lớp trừu tượng.)

When a virtual function is inherited, so is its virtual nature. This means that when a derived class inherits a virtual function from a base class and then the derived class is used as base for yet another derived class, the virtual function can be overridden by the final derived class (as well as the first derived class). For example, if base class B contains a virtual function called f(). And D1 inherits B and D2 inherits D1, both D1 and D2 can override f() relative to their respective classes.

Khi một hàm ảo được thừa kế, vì thế là ảo tự nhiên của nó. Điều này nghĩa là khi một lớp dẫn xuất thừa kế một hàm ảo từ một lớp cơ sở và sau đó lớp dẫn xuất được dùng như cơ sở cho lớp dẫn xuất khác, hàm ảo có thể ghi đè bằng lớp dẫn xuất cuối cùng (tốt như là lớp dẫn xuất đầu tiên). Ví dụ, nếu lớp cơ sở B chứa một hàm ảo gọi là f(). Và D1 thừa kế B và D2 thừa kế D1, cả hai D1 và D2 có thể ghi đè f() liên quan từng lớp tương ứng.

EXAMPLES (VÍ DỤ)

Here is an improved version of the program shown in Example 4 in the preceding section. In this version, the function getarea() is declared as pure in the base class area.

Đây là phiên bản cải tiến của chương trình đã trình bày ở ví dụ 4 trong phần trước. Trong phiên bản này, hàm getarea() được hiển thị như là nguyên mẫu trong vùng lớp cơ sở.

```
// Tạo ra một lớp trừu tượng

#include <iostream>

using namespace std;


class area {

     double dim1, dim2; // Nơi chứa kích thước các hình

public:

     void setarea(double d1, double d2)

     {

          dim1 = d1;

          dim2 = d2;

     }

     void getdim(double &d1, double &d2)

     {

     d1 = dim1;

     d2 = dim2;

     }

     virtual double getarea() = 0; // Hàm ảo thuần túy

};


class rectangle : public area {
```

```
public:
     double getarea()
     {
          double d1, d2;
     getdim(d1, d2);
     return d1 * d2;
     }
};

class triangle : public area {
public:
     double getarea()
     {
          double d1, d2;

     getdim(d1, d2);
     return 0.5 * d1 * d2;
     }
};

int main()
{
     area *p;
     rectangle r;
     trisngle t;
```

```
        r.setarea(3.3, 4.5);
        r.setarea(4.0, 5.0);

        p = &r;
        cout << "Rectangle has area: "
        << p->getarea() << "\n";

        p = &t;
        cout << "Triangle has area: "
        << p->getarea() << "\n";

        return 0;
    }
```

Now that getarea() is pure, it ensures that each derived class will override it.

Bây giờ getarea() là nguyên mẫu, bảo đảm rằng mỗi lớp dẫn xuất sẽ ghi đè lên nó.

The following program illustrates how a function's virtual nature is preserved when it is inherited:

Chương trình dưới đây minh họa hàm ảo tự nhiên được giữ như thế nào khi nó được thừa kế:

```
// Những hàm ảo giữ lại bản chất ảo khi được thừa kế.
#include <iostream>
using namespace std;

class base {
```

```
public:
     virtual void func()
     {
          cout << "Using base version of func(): ";
     }
};

class derived1 : public base {
public:
     void func()
     {
          cout << "Using derived1's version of func(): ";
     }
};

// derived2 thừa kế derived1.
class derived2 : public derived1 {
public:
     void func()
     {
          cout << "Using derived2's version of func(): ";
     }
};

int main()
```

```
{
      base *p;
      base ob;
      derived1 d_ob1;
      derived2 d_ob2;

      p = &ob;
      p->func(); // sử dụng hàm func() của lớp base

      p->&d_ob1;
      p->func(); // sử dụng hàm func() của lớp derived1

      p->&d_ob2;
      p->func(); // sử dụng hàm func() của lớp derived2

      return 0;
}
```

In this program, the virtual function func() is first inherited by derived1, which overrides it relative to itself. Next, derived2 inherits derived1. In derived2, func() is again overridden.

Trong chương trình này, hàm ảo func() được thừa kế đầu tiên bởi derived1, ghi đè lên nó liên quan tới chính nó. Tiếp theo, derived2 thừa kế derived1. Trong derived2, func() được ghi đè lần nữa.

Because virtual functions are hierarchical, if derived2 did not override func(), when d_ob2 was accessed, derived1's func() would have been used. If either derived1 nor derived2 had overridden func(), all references to it would have been routed to the one defined in base.

Bởi vì hàm ảo có thứ bậc, nếu derived2 không ghi đè func(), khi d_ob2 được truy xuất, func() của derived1 sẽ được dùng. Nếu cả hai derived1 và derived2 ghi đè func(), tất cả sự tham khảo cho nó sẽđược gửi thẳng tới một định nghĩa cơ sở.

EXERCISES (BÀI TẬP)

1. *On your own, experiment with the two example programs. Specifically, try creating an object by using area from Example 1 and observe the error message. In Example 2, try removing the redefinition of func() within derived2. Confirm that, indeed, the version inside derived1 is used.*

Bạn hãy tự rút ra kinh nghiệm với hai chương trình ví dụ. Một cách rõ ràn, thử tạo một đối tượng bằng cách sử dụng vùng từ ví dụ 1 và quan sát lỗi. Trong ví dụ 2, thử bỏ sự định nghĩa lại của func() trong vòng derived2. Xác nhận rằng, thật vậy, phiên bản bên trong derived1 được dùng.

2. *Why can't an object be created by using an abstract class?*

Tại sao một đối tượng có thể được tạo bằng cách sử dụng lớp trừu tượng?

3. *In Example 2, what happens if you remove only the redefinition of func() inside derived1? Does the program still compile and run? If so, why?*

Trong ví dụ 2, điều gì xảy ra khi bạn bỏ duy nhất sự định nghĩa lại của func() bên trong derived1? Có phải chương trình tiếp tục biên dịch và chạy? Nếu có, tại sao?

10.4. APPLYING POLYMORPHISM - ÁP DỤNG ĐA HÌNH

Now that you know how to use a virtual function to achieve run-time polymorphism, it is time to consider how and why to use it. As has been stated many time in this book, polymorphism is the process by which a common interface is applied to two or more similar (but technically different) situation, thus implementing the "one interface, multiple methods" philosophy. Polymorphism is important because it can greatly simplify complex systems. A single, well-defined interface is used to access a number of different but related actions, and artificial complexity is removed. In essence, polymorphism allows the logical relationship of similar actions to become apparent.

Thus, the program is easier to understand and maintain. When related actions are accessed through a common interface, you have less to remember.

Bây giờ bạn đã biết sử dụng hàm ảo để chạy ở chế độ đa hình như thế nào, đó là thời điểm nhận ra dùng nó như thế nào và tại sao phải dùng nó. Như đã phát biểu nhiều lần trong cuốn sách này, đa hình là quá trình bởi một diện mạo chung được áp dụng hai hay nhiều tình huống giống nhau (nhưng kĩ thuật thì khác nhau), vì vậy thi hành triết lý "một giao diện, nhiều phương pháp". Đa hình là quan trọng vì nó có thể đơn giản hóa hệ thống phức tạp. Một đơn thể, giao diện định nghĩa tốt được dùng để truy xuất số của sự khác nhau nhưng hành động liên quan nhau, và sự phức tạp nhân tạo được bỏ. Thực chất, đa hình cho phép mối quan hệ logic của hành động giống nhau để trở nên rõ ràng. Vì vậy, chương trình đơn giản hơn để hiểu và duy trì. Khi hành động liên quan được truy xuất thông qua giao diện chung, bạn ít phải nhớ.

There are two terms that are often linked to OOP in general and to C++ specifically. They are early binding and late binding essentially refers to those events that can be known at compile time. Specifically, it refers to those function calls that can be resolved during compilation. Early bound entities include "normal" functions, overloaded functions, and non-virtual member and friend functions. When these types of functions are compiled, all address information necessary to call them is known at compile time. The main advantage of early binding (and the reason that it is so widely used) is that it is very efficient. Calls to function bound at compile time are the faster types of function calls. The maintain disadvantage is lack of flexibility.

Có hai dạng mà thường liên kết với hướng đối tượng nói chung và C++ nói riêng. Nó sớm kết nối và kết nối sau cùng về cơ bản xem như những sự kiện có thể được biết ở thời gian biên dịch. Một cách rõ ràng, nó xem những hàm đó gọi rằng có thể giải quyết trong suốt sự biên dịch. Đầu tiên giới hạn thực thể bao gồm hàm "thông thường", nạp chồng hàm, và thành phần không ảo và hàm bạn. Khi những loại của hàm được biên dịch, tất cả những thông tin địa chỉ cần thiết để gọi nó được biết tại thời điểm biên dịch. Sự thuận lợi chính của nối kết sớm này (và lý do mà nó được sử dụng rộng rãi) là nó rất có hiệu quả. Những giới hạn của lời gọi hàm tại thời điểm biên dịch là loại nhanh hơn của lời gọi hàm. Sự không thuận tiện là thiếu sự uyển chuyển.

Late binding refers to events that must occur at run time. A late bound function call is one in which the address of the function to be called is not known until the program runs. In C++, a virtual function is a late bound object. When a virtual function is accessed via a base class pointer, the program must determine at run time what type of object is being pointed to and then select which version of the overridden function to execute. The main advantage of late binding is flexibility at run time. Your program is free to respond to random events without having to contain large amounts of "contingency code". Its primary disadvantage is that there is more overhead

associated with a function call. This generally makes such calls slower than those that occur with early binding.

Kỹ thuật kết nối trễ là là kỹ thuật cho phép sự kiện chỉ xảy ra khi chương trình thực thi. Một hàm kết nối trễ sẽ gọi một địa chỉ của hàm được gọi- hàm này chỉ được nhận dạng trong khi chương trình chạy. Trong C++, một hàm ảo là một đối tượng như thế. Khi hàm ảo được truy xuất qua con trỏ lớp cơ sở, chương trình sẽ quyết định loại đối tượng nào được trỏ đến và lựa chọn nó đẽ thực thi. Điểm thuận chính của kỹ thuật kết nối trễ là sự uyển chuyển trong khi chạy. Chương trình của bạn sẽ có thời gian để đáp ứng các sự kiện ngẫu nhiên mà không phải chứa một lượng lớn các đoạn mã thực thi không cần thiết. Điểm bất tiện nhất đó là có nhiều kết nối hơn khi hàm này được gọi. Chính vì điều này mà làm cho việc gọi trở nên chậm hơn so với việc kết nối sớm.

EXAMPLES

Here is a program that illustrates "one interface, multiple methods". It defines an abstract list class for integer values. The interface to the list is defined by the pure virtual functions store() and retrieve(). To store a value, call the store() function. To retrieve a value from the list, call retrieve(). The base class list does not define any default methods for these actions. Instead, each derived class defines exactly what type of list will be maintained. In the program, two lists operate completely differently, each is accessed using the same interface. You should study this program carefully.

Đây là một chương trình minh họa" một hàm, nhiều chức năng". Nó định nghĩa một lớp danh sách trừu tượng cho số nguyên. Bề mặt của danh sách được định nghĩa bởi các hàm ảo nguyên gốc là store() và retrieve(). Để chứ dữ liệu, gọi hàm store(). Để lấy dữ liệu gọi hàm retrieve(). Lớp cơ sở của danh sách không định nghĩa bất kỳ một phương thức mặc định nào cho những hành động này. Thay vào đó, mỗi lớp dẫn xuất sẽ định nghĩa chính xác loại danh sách sẽ được duy trì. Trong chương trình, có 2 danh sách các thao tác hoàn toàn khác nhau, mỗi cái được truy xuất giống nhau về hình thức. Bạn hãy xem xét kỹ chương trình.

```
// Giải thích các ham ảo.

#include <iostream>

#include <cstdlib>

#include <cctype>

using namespace std;
```

```
class list {
public:
     list *head; // con trỏ để bắt đầu danh sách.
     list *tail; // con trỏ để kết thúc danh sách.
     list *next; // con trỏ để chỉ mục tếp theo.
     list num; // giá trị được lưu trữ.

     list() { head = tail = next = NULL; }
     virtual void store(int i) = 0;
     virtual int retrieve() = 0;
};

// Tạo một danh sách kiểu hàng đợi.
class queue : public list {
public:
     void store(int i);
     int retrieve();
};

void queue::store(int i)
{
     list *item;

     item = new queue;
     if(!item) {
```

```
            cout << "Allovation error.\n";
            exit(1);
        }
        item->num = i;

        // Đặt kết thúc danh sách.
        if(tail) tail->next = item;
        tail = item;
        item->next = NULL;
        if(!head) head = tail;
    }

    int queue::retrieve()
    {
        int i;
        list *p;

        if(!head) {
            cout << "List empty.\n";
            return 0;
        }

        // Lấy ra từ vị trí bắt đầu của danh sách.
        i = head->num;
        p = head;
```

```
        head = head->next;
        delete p;

        return i;
}

// Tạo ra một danh sách ngăn xếp.
class stack : public list {
public:
        void store(int i);
        int retrieve();
};

void stack::store(int i)
{
        list *item;

        item = new stack;
        if(!item) {
                cout << "Allovation error.\n";
                exit(1);
        }
        item->num = i;

        // Đặt trên phía trước danh sách tính toán ngăn xếp.
```

```
    if(head) tail->next = item;
    head = item;
    if(!tail) tail = head;
}

int stack::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "List empty.\n";
        return 0;
    }

    // Lấy ra từ vị trí bắt đầu của danh sách.
    i = head->num;
    p = head;
    head = head->next;
    delete p;

    return i;
}

int main()
```

```
{
    list *p;

    // Khai báo hàng đợi.
    queue q_ob;
    p = &q_ob; // Trỏ đến hàng đợi.

    p->store(1);
    p->store(2);
    p->store(3);

    cout << "Queue: ";
    cout << p->retrieve();
    cout << p->retrieve();
    cout << p->retrieve();

    cout << '\n';

    // Khai báo ngăn xếp.
    stack s_ob;
    p = &s_ob; // Trỏ đến ngăn xếp.

    p->store(1);
    p->store(2);
    p->store(3);
```

```
        cout << "Stack: ";
        cout << p->retrieve();
        cout << p->retrieve();
        cout << p->retrieve();

        cout << '\n';



        return 0;
}
```

The main() function in the list program just shown simply illustrate that the list classes do, indeed, work. However, to begin to see why run-time polymorphism is so powerful, try using this main() instead:

Hàm main() trong chương trình chỉ minh họa đơn giản lớp các danh sách làm gì. Tuy nhiên, bạn cũng có thể thấy được tại sao sự kết nối đa hình lại rất mạnh, thử sử dụng hàm main này thay thế:

```
int main()
{
        list *p;
        stack s_ob;
        queue q_ob;

        char ch;
        int i;
```

```
        for(i=0; i<10; i++) {
             cout << "Stack or Queue? (S/Q): ";
             cin >> ch;
             ch = tolower(ch);
             if(ch=='q') p = &q_ob;
             else p = &s_ob;
             p->store(i);
        }

        cout << "Enter T to terminate\n";
        for(;;)
        {
             cout << "Remove from Stack or Queue? (S/Q): ";
             cin >> ch;
             ch = tolower(ch);
        if(ch=='t') break;
             if(ch=='q') p = &q_ob;
             else p = &s_ob;
             cout << p->retrieve() << '\n';
        }

        cout << '\n';
        return 0;
  }
```

This main() illustrate how random events that occur at run time can be easily handled by using virtual functions and run-time polymorphism. The program executes a for loop

running from 0 to 9. Each iteration through the loop, you are asked to choose into which type of list – the stack or the queue – you want to put a value. According to your answer, the base pointer p is set to point to the correct object and the current value of i is stored. Once the loop is finished, another loop begins that prompts you to indicate from which list to remove a value. Once again, it is your response that determines which list is selected.

Hàm main này minh họa làm thế nào mà các sự kiện ngẫu nhiên xảy ra trong lúc chạy có thể dễ dàng được quản lý bằng cách sử dụng hàm ảo và tính đa hình trong khi chạy. Chương trình này thực thi một vòng lặp từ 0 đến 9. Mỗi lần lặp, bạn được yêu cầu là chon loại danh sách nào- ngăn xếp hay là hàng đợi- mà bạn muốn đặt một giá trị vào. Tùy vào câu trả lời của bạn, con trỏ cơ sở p sẽ chĩ vào đúng đối tượng và giá trị hiện thời của I được lưu dữ. mỗi khi một vòng lặp kết thúc,một vòng lặp khác bắt đầu yêu cầu bạn chỉ ra giá trị nào được loại bỏ ra khỏi loại danh sách nào. Một lần nữa, bạn sẽ thấy được chương trình lựa chọn loại danh sách như thế nào.

While this example is trivial, you should be able to see how run-time polymorphism can simply a program that must respond to random event. For instance, the Windows operating system interfaces to a program by sending it messages. As far as the program is concerned, these messages are generated at random, and your program must respond to each one as it is received. One way to respond to these messages is through the use of virtual functions.

Khi mà ví dụ đơn giản, bạn co thể dễ dàng xem xét làm thế nào mà tính đa hình trong khi chạy có thể dễ dàng cho một chương trình thực thi các sự kiện ngẫu nhiên. Mỗi trường hợp, hệ thống thao tác windows sẽ gởi đến chương trình một thông điệp khác nhau. Khi mà chương trình được kết nối, nhưng thông điệp này được phát sinh ngẫu nhiên, và chương trình của bạn sẽ phải hồi đáp mỗi lần mà nó nhận được thông điệp. Một cách để thực hiện điều này là sử dụng hàm ảo.

EXERCISES

1. Add another type of list to the program in Example 1.Have this version maintain a sorted list (in ascending order). Call this list sorted.

Thêm vào một loại danh sách khác vào trong chương trình ở ví dụ 1. có một hàm sắp sếp danh sách (theo chiều tăng). Gọi hàm sắp xếp này.

2. On your own,think about ways in which you can apply run-time polymorphism to simplify the solutions to certain types of problems.

Theo bạn, hãy nghĩ cách để bạn có thể cung cấp cách thức giải quyết vấn đề đa hình trong khi chạy cho mỗi loại vấn đề.

SKILL CHECK (KIỂM TRA KĨ NĂNG)

Mastery

At this point you should be able to perform the following exercises and answer the questions.

1. What is a virtual function?

2. What types of functions cannot be made virtual?

3. How does a virtual function help achieve run-time polymorphism? Be specific

4. What is a pure virtual function?

5. What is an abstract class? What is a polymorphism class?

6. Is the following fragment correct? If not, why not?

Nắm vững

Tại thời điểm này bạn có thể làm những bài tập dưới đây và trả lời một số câu hỏi.

1. Hàm ảo là gì?

2. Loại hàm nào không thể làm hàm ảo?

3. Làm thế nào mà hàm ảo có thể thực thi trong chế độ đa hình? Giải thích rõ

4. Thế nào là hàm ảo cơ bản?

5. Thế nào là lớp trừu tượng (lớp ảo)? thế nào là lớp đa kế thừa?
6. Đoạn nào sau đây đúng? Nếu không đúng, giải thích?

```
class base {

public:

     virtual int f(int a) = 0;
```

```
    // …
};


class derived : public base {
public:
    int f(int a, int b) { return a*b; }
    // …
};
```

7. Is the virtual quality inherited?

8. On your own, experiment with virtual functions at this time. This an important concept and you should master the technique.

7. Thế nào là kế thừa ảo?

8. Theo bạn, kinh nghiệm xử lý hàm ảo tại thời điểm này. Nó là khái niệm quan trọng và bạn nên làm chủ kỹ thuật này.

Cumulative

This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. Enhance the list example from Section 10.4, Example 1, so that it overloads the + anf – operators. Have the + store an element and the – retrieve an element.

2. How do virtual functions differ from overloaded functions?

3. On your own, reexamine some of the function overloading examples presented earlier in this book. Determine which can be converted to virtual functions. Also, think about ways in which a virtual function can solve some of your own programming problems.

Nâng cao

Phần này kiểm tra xem bạn có thực sự nắm vững và kết hợp nội dung chương này với những chương trước hay không

1. Nâng cao thêm danh sách ví dụ trong phần 10.4 ví dụ 1, làm thế nào để nạp chồng + và -. Toán tử + để chứa thành phần và – dung để loại trừ thành phần.

2. Hàm ảo khác với hàm nạp chồng như thế nào?

3. Theo bạn, kiểm tra lại xem một vài hàm nạp chồng được giới thiệu trước đó trong cuốn sách này. Quyết định xem có nên chuyển sang hàm ảo hay không? Nghĩ cách dùng hàm ảo để giải quyết một số vấn đề trong chương trình của bạn.

This section checks how well you have integrated material in this chapter with that form the preceding chapters.

*Following is a reworked version of the **inventory** class presented in the preceding chapter. Write a program that fills in the functions **store()** and **retrieve().** Next, create a small inventory file in disk containing a few entries. Then, using random I/O, allow the user to display the information about any item by specifying its record number.*

Phần này dùng để kiểm tra cách tốt nhất để kết hợp những thứ có trong chương này với các chương trước.

1. Sau đây là một bản được làm lại của lớp **inventory** được giới thiệu ở chương trước. Viết một chương trình điền đầy vào hàm **store()** và **retrieve().** Kế tiếp, tạo một file kiểm tra nhỏ trên disk bao gồm một vài sự ghi chép. Sau đó, sử dụng I/O ngẫu nhiên cho phép người sử dụng xuất thông tin về bất cứ một vật nào được xác định bởi số mẫu tin của nó.

```
#include <fstream>

#include<cstring>

using namespace std;

#define size 40


class inventory

{

      char item[size]; //Name of item

      int onhand; //number on hand

      double cost; //cost of item
```

```
public:
     inventory (char*i, int o, double c)
     {
          strcpy (item, i);
          onhand = o;
          cost = c;
     }

     void store (fstream &stream);
     void retrieve (fstream &stream);
     friend ostream &operator<<(ostream &stream,
inventory ob);
     friend istream &operator >>(istream &stream,
inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
     stream <<ob.item <<":" <<ob.onhand;
     stream <<"on hand at $" <<ob.cost <<'\n';
     return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
```

```
        cout << "Enter item name:";
        stream >>ob.item;
        cout <<"Enter number on hand:";
        stream>>ob.onhand;
        cout <<"Enter cost:";
        stream >>ob.cost;

        return stream;
}

int main(int argc, char* argv[])
{
        return 0;
}
```

*As a special challenge, on your own, create a **stack** class for characters that stores them in a disk file rather than in an art in memory.*

2. Khi gặp một thử thách đặc biệt, thì tạo một lớp **stack** cho những kí tự để lưu chúng lên disk file thì tốt hơn là trình bày trong bộ nhớ.

CHAPTER 11

TEMPLATES AND EXCEPTION HANDLING - NHỮNG BIỂU MẪU VÀ TRÌNH ĐIỀU KHIỂN BIỆT LỆ

Chapter objectives

11.1. GENERIC FUNCTIONS

Những hàm tổng quát

11.2. GENERIC CLASSES

Những lớp tổng quát

11.3. EXCEPTION HANDLING

Điều khiển ngoại lệ

11.4. MORE ABOUT EXCEPTION HANDLING

Điều khiển ngoại lệ (tt)

11.5. HANDLING EXCEPTIONS THROWN BY NEW

Những điều khiển ném bằng new

This chapter discusses two of C++'s most important high-level features: templates and exception handling. While neither was part of the original specification for C++, both were added several years ago and are defined by Standard C++. They are supported by all modern C++ compilers. These two features help you achieve two of the most elusive goals in programming: the creation of reusable and resilient code.

Chương này thảo luận về 2 tính năng quan trọng của C++ là: những biểu mẫu và điều khiển ngoại lệ. Không nằm trong C++ ban đầu và cả hai đã được phát triển thêm cách đây vài năm và được định nghĩa bởi C++ chuẩn. Chúng đươc hỗ trợ bởi tất cả các trình biên dịch C++ hiện đại. Hai tính năng này giúp bạn đạt được những thắng lợi trong lập trình: bằng việc tạo ra các đoạn mã mang tính linh hoạt và có khả năng tái sử dụng.

Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data that operated upon is specified as a parameter. This allows you to use one function or class with several different types of data without having to explicitly recode a specific version for each different data type. Thus, templates allow you to create reusable code. Both generic functions and generic classes are discussed here.

Bằng việc dung biểu mẫu, hoàn toàn tạo được những hàm và lớp tổng quát. Trong một hàm hay lớp tổng quát, loại dữ liệu mà được dung thì được chỉ định như một tham số. Điều này cho phép bạn dùng một hàm hay lớp với vài kiểu dữ liệu khác nhau mà không phải viết lại code cho mỗi loại dữ liệu. Vì vậy, các biểu mẫu cho phép bạn tạo ra đoạn code tái sử dụng. Cả hàm và lớp tổng quát sẽ được nói ở đây.

Exception handling is the subsystem of C++ that allows you to handle errors that occur at run time in a structured and controlled manner. With C++ exception handling your program can automatically invoke an error handling routine when an error occurs. The principle advantage of exception handling is that it automates much of the error handling code that previously had to be coded "by hand" in any large program. The proper use of exception handling helps you to create resilient code

Điều khiển ngoại lệ là hệ thống phụ của C++ mà cho phép bạn điều khiển các lỗi xảy ra trong khi chạy trong một cấu trúc hay một loại kiểm soát. Với điều khiển ngoại lệ chương trình của bạn có thể tự động gọi một thủ tục điều khiển lỗi khi lỗi xảy ra. Thuận lợi chính của điều khiển ngoại lệ là tự động hóa mã điều khiển lỗi mà đã được code "bằng tay" trong bất kỳ một chương trình lớn nào trước đó

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

What is a virtual function?
What is a pure virtual function? If a class declaration contains a pure virtual function, what is that class called, and what restrictions apply to its usage?
Run-time polymorphism is achieved through the use of _____ functions and _____ class pointers. (Fill in the missing words.)
If, in a class hierarchy, a derived class neglects to override a (non-pure) virtual function, what happens when an object of that derived class calls that function?

What is the main advantage of run-time polymorphism? What is its potential disadvantage?

Trước khi tiếp tục bạn nên trả lời đúng các câu hỏi và làm các bài tập sau:

Hàm ảo là gì?

Hàm nguyên ảo là gì? Nếu một khai báo lớp chứa một hàm ảo, lớp đó được gọi cái gì và sự giới hạn khi ứng dụng

Đa xạ được tạo ra qua việc sử dụng của_____ các hàm và _____ các con trỏ lớp. (Điền vào chỗ trống)

Nếu trong một lớp có tôn ti, Sự bỏ qua một lớp được xuất phát để vượt qua một hàm ảo, điều gì xảy ra khi một đối tượng mà lớp xuất phát gọi hàm đó?

5. Thuận lợi chính của đa xạ là gì? Những bất lợi tiềm ẩn là gì?

11.1. GENERIC FUNCTIONS – NHỮNG HÀM TỔNG QUÁT

*A generic function defines a general set of operations that will be applied to various types of data. A generic function has the type of data that it will operate upon passed to it as a parameter. Using this mechanism, the same general procedure can be applied to a wide range of data. As you know, many algorithms are logically the same no matter what type of data is being operated upon. For example, the Quicksort algorithm is the same whether it is applied to an array of **integers** or an array of **floats**. It is just that the type of data is being sorted is different. By creating a generic function, you can define, independent of any data, the nature if the algorithm. Once this is done, the compiler automatically generates the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself.*

Hàm tổng quát định nghĩa một bộ các xử lý mà sẽ dung cho nhiều kiểu dữ liệu. Một hàm tổng quát có dạng dữ liệu mà xử lý thông qua một tham số. Sử dụng kỹ thuật này, thủ tục tổng quát có thể dùng cho một vùng dữ liệu rộng. Như bạn biết, nhiều giải thuật thì logic mà không được xử lý trên loại dữ liệu. Ví dụ, QuickSort cũng dùng như nhau đối với mảng số nguyên hay số thực. Nó chỉ khác nhau kiểu dữ liệu. Bằng cách tạo ra các hàm chung, bạn có thể định nghĩa, không phụ thuộc vào kiểu dữ liệu, tự nhiên với giải thuật. Một khi được dùng, trình biên dịch tự động sinh ra mã chuẩn cho kiểu dữ liệu thực sự được sử dụng khi bạn thực thi chương trình. Thực chất, khi bạn tạo ra hàm chung

thi bạn đang tạo ra hàm tự động hóa quá tải hàm chính bản thân nó.

*A generic function is created using the keyword **template**. The normal meaning of the word template accurately reflects the keyword's use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed. The general form of a template function definition is shown here:*

Một hàm chung được tạo bằng từ khóa **template**. Nghĩa thường của nó là biểu mẫu word một cách rất xát nghĩa với từ khóa được dùng trong C++. Nó dùng để tạo ra một biểu mẫu(hay một khung sườn) mà mô tả một hàm thực hiện cái gì, trình biên dịch sẽ điền các chi tiêt cần. Hình dạng tổng quát của một biểu mẫu chung được định nghĩa như sau:

Template <class *Ttype*> *ret-type func-name(parameter list)*

{

// body of function

}

Here Ttype is a placeholder name for a data type used by the function. This name can be used within the function definition. However, it is only a placeholder: the compiler will automatically replace this placeholder with an actual data type when it creates a specific version of the function.

*Although the use of the keyword **class** to specify a generic type in a template declaration is traditional, you can also use the keyword **typename**.*

Ở đây Ttype thế cho tên kiểu dữ liệu dùng cho hàm. Tên này có thể dùng mà không có định nghĩa hàm. Tuy vậy, chỉ là thay thế, trình biên dịch sẽ tự động thay vi trí này với kiểu dữ liệu thực sự mà nó được định rõ trong từng hàm. Mặc dù sử dụng từ khóa **class** để chỉ rõ loại tổng quát trong một khai báo biểu mẫu là truyền thống, bạn cũng có thể dùng từ khóa **typename**.

Examples

The following program creates a generic function that swaps the values of the two variables it is called with. Because the general process of exchanging two values is independent of the type of the variables, this process is good choice to

be made into a generic function.

Chương trình sau tạo một hàm chung mà hoán chuyển giá trị của 2 biến mà nó được gọi cùng. Bởi vì tiến trình chung của sự trao đổi 2 giá trị này là không phụ thuộc vào kiểu dữ liệu của biến, tiến trình này là sự chọn lựa tốt để làm một hàm chung.

```
// Function on temple example

#include <iostream>

using namespace std;


// this is a function template

template <class X> void swapargs(X &a, X &b)

{

     X temp;


     temp = a;

     a = b;

     b = temp;

}


int main()

{

     int i = 10, j = 20;

     float x = 10.1, y = 23.3;


     cout << "Original i, j: " << i << "  " << j <<
endl;

     cout << "Original x, y: " << x << "  " << y <<
endl;
```

```
     swapargs(i, j); // swap integers
     swapargs(x, y); // swap floats

     cout << "Swapped i, j: " << i << "  " << j <<
endl;
     cout << "Swapped x, y: " << x << "  " << y <<
endl;

     return 0;
}
```

*The keyword **template** is used to define a generic function. The line*

template <class X> void swapargs(X &a, X &b)

tells the compiler two things: that a template is being created and that a generic definition is beginning. Here X is a generic type that is used as a placeholder. After the template portion, the function swapargs() is declared, using X as the data type of the value that will be swapped. In main(), the swapargs() function is called using two different types of data: integers and floats. Because swapargs()-one that will exchange integer values and one that will exchange floating-point values. You should compile and try this program now.

Từ khóa **template** dùng để định nghĩa hàm chung. Dòng trên nói với trình biên dịch 2 việc: một biểu mẫu đang được tạo ra và hàm chung đó đang bắt đầu. Ở đây X là kiểu chung mà dùng để thay thế. Phần biểu mẫu sau, hàm swaparg() được khai báo, dùng X như kiểu dữ liệu mà sẽ đươc hoán chuyển. Trong main(), hàm swaparg() được gọi với hai kiểu dùng là số nguyên và số thực. Bởi vì swaparg()- một mặt sẽ thay đổi giá trị nguyên và mặt khác sẽ thay đổi các giá trị số chấm động. Bạn nên biên dịch và thử chương trình ngay.

Here are some other terms that are sometimes used when templates are discussed and that you might encounter in other C++ literature. First, a generic function (that is, a function definition preceded by a template statement)

is also called a template function. When the compiler creates a specific version of this function, it is said to have created a general function. The act of generating a function is referred to as instantiating it. Put differently, a generated function is a specific instance of a templates function.

Ở đây có một số thuật ngữ mà thỉnh thoảng được dùng khi các biểu mẫu được thảo luận và bạn có thể gặp trong các văn phong C++ khác. Đầu tiên, một hàm chung(là một hàm có sự định nghĩa bằng một cậu phát biểu template) cũng còn được gọi là hàm biểu mẫu(template). Khi trình biên dịch tạo ra một phiên bản được định rõ của hàm này, nó sẽ tạo ra một hàm chung. Hàm động tổng quát hóa một hàm được quy vào việc như thể thuyết minh nó. Sự khác biệt là một hàm chung là một ví dụ xác định của một hàm biểu mẫu.

The template portion of a generic function definition does not have to be on the same line as the function's name. For example, the following is also a common way to format the swapargs() function:

Phần biểu mẫu của một định nghĩa hàm chung không phải nằm trên một dòng đối với tên hàm. Ví dụ, như sau:

```
template <class X>

void swapargs(X &a, X &b)

{

     X temp;

     temp = a;

     a=b

     b=temp;

}
```

If you use this form, it is important to understand that no other statements can occur between the templates statement and the start of the generic function definition. For example, the following fragment will not compile:

Nếu bạn sử dụng theo kiểu này, thì ở giữa câu template và nơi bắt đầu định nghĩa hàm không được có bất kỳ câu khai báo nào hết. Ví dụ, đoạn sau sẽ không được biên dịch

```
//This will not compile
```

```
template <class X>
int i;    // this is an error
void swapargs(X &a, X &b)
{
     X  temp;

     temp = a;
     a = b;
     b = temp;
}
```

As the comments imply, the template specification must directly precede the rest of the function definition.

Như chú thích gợi ý, sự chỉ rõ biểu mẫu phải trực tiếp để trước phần định nghĩa hàm

*As mentioned, instead of using the keyword **class**, you can use the keyword **typename** to specify a generic type in a template definition. For example, here is another way to declare the swapargs() function.*

Như đã đề cập, thay vì dùng từ khóa **class**, bạn có thể dùng từ khóa **typename** để chỉ ra kiểu chung trong định nghĩa biểu mẫu. Ví dụ, ở đây có cách khác để khai báo hàm swapargs().

```
// Use typename
template <typename X> void swapargs(X &a, X &b)
{
     X temp;

     temp = a;
     a = b;
     b = temp;
```

```
}
```

*The **typename** keyword can also be used to specify an unknown type within a template, but this use is beyond the scope of this book.*

Từ khóa **typename** cũng có thể dùng để chỉ rõ kiểu chưa biết bên trong một biểu mẫu, nhưng cách dùng này nằm ngoài phạm vi cuốn sách.

You can define more than one generic data type with the template statement, using a comma-seperated list. For example, this program creates a generic function that has two generic types:

Bạn có thể định nghĩa nhiều hơn một kiểu dữ liệu chung với phát biểu template, việc dùng một danh sách phân cách bởi dấu phẩy. Ví dụ, chương trình sau tạo ra một hàm chung mà có hai kiểu dữ liệu chung:

```
#include <iostream>

using namespace std;


template <class type1, class type2>

void myfunc(type1 x, type2 y)

{

     cout << x << ' ' << y << endl;

}

int main()

{

     myfunc(10, "hi");


     myfunc(0.23, 10L);


     return 0;

}
```

In this example, the placeholder, types type1 and type2 are replaced by the compiler with the data types int and char and double and long, respectively, when the compiler generates the specific instances of myfunc().

When you create a generic function, you are, in essence, allowing the compiler to generate as many different versions of that function as necessary to handle the various ways that your program calls that function.

Trong ví dụ này, phần thay thế, type1 và type2 được trình biên dịch thay bằng kiểu dữ liệu int và char, double và long, khi trình biên dịch tổng quát hóa những thí dụ xác định của hàm myfunc().

Nhớ rằng khi bạn tạo ra hàm chung, thì bản chất của nó là bạn cho phép trình biên dịch tổng quát hóa nhiều hàm cùng loại lại như cần thiết để kiểm soát nhiều cách khác nhau khi chương trình gọi hàm.

Generic functions are similar to overloaded functions except that they are more restrictive. When functions are overloaded you can have different actions performed within the body of each function. But a generic function must perform the same general action for all versions. For example, the following overloaded functions cannot be replaced by a generic function because they do not do the same thing:

Hàm chung cũng giống như các quá tải hàm ngoại trừ chúng giới hạn hơn. Khi các hàm được quá tải bạn có thể có nhiều hành động khác nhau được thi hành bên trong thân hàm. Nhưng hàm tổng quát thì chỉ có những hành động chung cho tất cả các kiểu. Ví dụ, các hàm quá tải sau không thể thay thế bởi hàm chung bởi vì chúng không thực hiện công việc giống nhau:

```
void outdata(int i)

{

     cout << i;

}


void outdata(double d)

{
```

```
        cout << setprecision(10) << setfill('#');
        cout << d;
        cout << setprecision(6) << setfill(' ');
}
```

Even though a template function overloads itself as needed, you can explicitly overload one, too. If you overload a generic function, that overloaded function overrides (or "hides") the generic function relative to that specific version. For example, consider this version of Example 1:

Mặc dù một hàm chung quá tải chính bản thân nó, bạn cũng có thể quá tải rõ rang một hàm thôi. Nếu bạn quá tải hàm chung, hàm quá tải sẽ ghi đè (hay "ẩn") hàm chung có cùng quan hệ. Ví dụ, :

```
//Overloading a template function
#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
        X temp;

        temp = a;
        a = b;
        b = temp;
}

// This overrides the generic version of swapargs()
void swapargs(int a, int b)
{
```

```
     cout << "This is inside swapargs(int, int) \n";
}

int main()
{
     int i = 10, j = 20;
     float x = 10.1, y = 23.3;

     cout << "Original i, j: " << i << ' ' << j <<
endl;
     cout << "Original x, y: " << x << ' ' <<y << endl;

     swapargs(i, j); // calls overloaded swapargs()
     swapargs(x, y); // swap floats

     cout << "Swapped i, j: "  << i << ' ' << j <<
endl;
     cout << "Swapped x, y: "  << x << ' ' << y <<
endl;

     return 0;
}
```

As the comments indicate, when swapargs(i, j) is called, it invokes the explicitly overloaded version of swapargs() defined in the program. Thus, the compiler does not generate this version of the generic swapargs() function because the generic function is overridden by the explicit overloading.

Như chú thích đã chỉ ra, khi swapargs(i, j) được gọi, chúng hàm swapargs() đã được quá tải trong chương trình. Vì vậy, trình biên dịch không tổng quát hóa hàm swapargs() bởi vì chúng đã được chép đè bởi hàm quá tải.

Manual overloading of a template, as shown in this example, allows you to tailor a version of a generic function to accommodate a special situation. However, in general, if you need to have different version of a function for different data types, you should use overloaded functions rather than templates.

Việc quá tải thủ công của một biểu mẫu, trong ví dụ này, cho phép biến đổi một hàm chung cho phù hợp với tình huống đặc biệt. Tuy nhiên, trong tổng quát nếu bạn cần một hàm cho nhiều kiểu dữ liệu khác nhau, bạn nên dùng quá tải hàm hơn là biểu mẫu.

Exercises

If you have not done so, try each of the preceding examples.

Nếu bạn chưa làm, thử với những ví dụ trước.

Write a generic function, called min(), that returns the lesser of its two arguments. For example, min(3, 4) will return 3 and min('c', 'a') will return a. Demonstrate your function in a program.

Viết một hàm chung, tên là min(), mà trả về giá trị thấp hơn. Ví dụ, min(3, 4) sẽ trả về 3 và min('c', 'a') sẽ trả về a. Thử nghiệm hàm của bạn trong một chương trình

A good candidate for a template function is called find(). This function searches an array for an object. It returns either the index of the matching object (if one is found) or -1 if no match is found. Here is the prototype for a specific version of find(). Convert find() into a generic function and demonstrate your solution within a program. (The size parameter specifies the number of elements in the array.)

Một ứng cử tốt cho biểu mẫu là hàm mang tên find(). Hàm này tìm trong mảng một đối tượng. Nó trả về chỉ số của đối tượng (nếu có) và -1 nếu không. Ở đây là cách thiết kế hàm find(). Chuyển find() thành hàm chung và thi triển giải pháp của bạn trong 1 chương trình. (Tham số size chỉ ra số lượng phần tử trong mảng).

```
int find(int object, int *list, int size)

{
```

```
    // ...
}
```

4. *In your own word, explain why generic functions are valuable and may help simplify the source code to programs that you create.*

Theo bạn, giải thích tại sao hàm chung có giá trị và có thể giúp đơn giản hóa mã chương trình.

11.2. GENERIC CLASSES – LỚP TỔNG QUÁT

In addition to defining generic functions, you can also define generic classes. When you do this, you create a class that defines all algorithms used by that class, but the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

Thêm vào định nghĩa các hàm tổng quát, bạn cũng có thể định nghĩa các lớp tổng quát. Khi bạn làm điều này, bạn tạo ra một lớp mà định nghĩa tất cả các giải thuật được dùng bởi lớp đó, nhưng kiểu dữ liệu thực sự được thao tác sẽ được chỉ rõ các tham số khi đối tượng của lớp đó được tạo ra.

Generic classes are useful when a class contains generalizable logic. For example, the same algorithm that maintains a queue of integers will also work for a queue of characters. Also, the same mechanism that maintains a linked list of mailing addresses will also maintain a linked list of auto part information. By using a generic class, you can create a class that will maintain a queue, a linked list, and so on for any type of data. The compiler will automatically generate the correct type of object based upon the type you specify when the object created.

Các lớp chung thì rất hữu dụng khi một lớp chứa các đặc tính logic chung. Ví dụ, giải thuật giống nhau duy trì hang đợi các số nguyên sẽ hoạt động tương tự với hang đợi các ký tự. Cũng như vậy, các kỹ thuật mà duy trì một danh sách liên kết đia chỉ email cũng duy trì một danh sách liên kết tự động các thông tin. Bằng việc dùng các hàm tổng quát, bạn có thể tạo ra một lớp mà sẽ duy trì hàng đợi, một danh sách liên kết và v.v… cho bất kỳ kiểu dữ liệu nào. Trình biên dịch sẽ tự động tổng quát hóa chính xác kiểu dữ liệu dựa trên kiểu đối tượng khi đối tượng được tạo ra.

The general form of a generic class declaration is shown here: (dạng chung của lớp

tổng quát được khai báo như sau)

```
template <class Ttype> class class-name{
        ….
}
```

Here Ttype is the placeholder type name that will be specified when a class is instantiated. If necessary, you can define more than one generic data type by using a comma-seperated list.

Ở đây, Ttype là thay thế cho tên kiểu mà sẽ được chỉ rõ khi một đối tượng được thuyết minh. Nếu cần thiết bạn có thể định nghĩa nhiều hơn một kiểu dữ liệu chung bằng cách dùng danh sách lien kết bởi dấu phẩy,

Once you have created a generic class, you create a specific instance of that class by using the following general form: (Một khi bạn tạo ra một lớp chung bạn tạo ra một ví dụ xác định của lớp đó bằng cách dùng dạng sau)

```
class-name <type> ob;
```

Here type is the type name of the data that the class will be operating upon.

Member functions of a generic class are themselves, automatically generic. They need not be explicitly specified as such using template.

Ở đây, type là tên kiểu của dữ liệu mà lớp xử lý trên đó. Hàm thành viên của lớp chung là chính bản thân chúng, tổng quát hóa tự động. Chúng không cần chỉ rõ như việc dùng template.

As you will see in Chapter 14, C++ provides a library that is built upon template classes. This library is usually referred to as the Standard Template Library, or STL for short. It provides generic versions of the most commonly used algorithms and data structures. If you want to use the STL effectively, you'll need a solid understanding of template classes and their syntax.

Bạn sẽ gặp trong chương 14, C++ cung cấp một thư viện mà xây dựng trên các lớp biểu mẫu. Thư viện này thường dùng tham khảo như thư viện biểu mẫu chuẩn, hay STL ngắn gọn. Nó cung cấp các bản tổng quát cho hầu hết các cấu trúc dữ liệu và giải thuật thông dụng. Nếu bạn muốn dùng STL tác động, bạn sẽ cần hiểu về các lớp biểu mẫu và cú pháp của chúng

This program creates a very simply generic singly linked list class. It then demonstrates the class by creating a linked list that stores characters.

Chương trình này tạo ra một danh sách liên kết đơn tổng quát. Nó minh họa cho lớp mà tạo ra một danh sách liên kết các ký tự.

```
// A simple generic linked list.

#include <iostream>

using namespace std;


template <class data_t> class list {

     data_t data;

     list  *next;

public:

     list(data_t d);

     void add(list *node) {node->next = this; next =
0; }

     list *getnext()     { return next; }

     data_t getdata()    { return data; }

};


template <class data_t> list<data_t>::list(data_t d)

{

     data = d;

     next = 0;

}
```

```
int main()
{
     list<char> start('a');
     list<char> *p, *last;
     int i;

     // build a list
     last = &start;
     for(i = 1; i < 26; i++) {
          p = new list<char> ('a' + i);
          p->add(last);
          last = p;
     }

     // follow the list
     p = &start;
     while(p) {
          cout << p->getdata();
          p = p->getnext();
     }

     return 0;
}
```

As you can see, the declaration of a generic class is similar to that of a generic function. The actual type of data stored by the list is generic in the class declaration. It is not until an object of the list is declared that the actual data type is determined. In this example, objects and pointers are created inside main() that specify that the data type of the list will be char. Pay special attention to this declaration:

Như bạn thấy đó, sự khai báo lớp tổng quát cũng giống như hàm tổng quát. Kiểu dữ liệu thực sự được lưu trữ bởi danh sách là chung trong sự định nghĩa lớp. Nó không còn là một đối tượng của danh sách khai báo mà kiểu dữ liệu thực sự mới quyết định. Trong ví dụ này, các đối tượng và con trỏ được tạo ra bên trong thân main() mà chỉ rõ kiểu đối tượng của danh sách là ký tự. Chú ý vào sự khai báo này:

```
list<char> start('a');
```

Notice: *how the desired data type is passed inside the angle brackets.*

Chú ý: làm sao để ra lệnh kiểu dữ liệu được đặt trong dấu <>

You should enter and execute this program. It builds a linked list that contains the characters of the alphabet and then displays them. However, by simply changing specified when list objects are created, you can change the type of data stored by the list. For example, you could create another object that stores integers by using this declaration:

Bạn nên gõ và thực thi chương trình. Nó tạo ra một danh sách liên kết mà chứa các ký tự của bảng alphabet và in ra màn hình chúng. Tuy nhiên, bằng sự thay đổi đơn giản khi đối tượng danh sách được tạo, bạn có thể thay đổi kiểu dữ liệu chứa bởi danh sách. Ví dụ, bạn có thể tạo ra đối tượng khác mà chứa số nguyên bằng khai báo sau:

```
list<int> int_start(1);
```

You can also use list to store data types that you create. For example, if you want to store address information, use this structure:

Bạn cũng có thể dùng danh sách để liên kết kiểu dữ liệu mà bạn tạo ra. Ví dụ, nếu bạn muốn lưu lại thông tin địa chỉ, dùng cấu trúc sau:

```
struct addr {

     char name[40];
```

```
    char street[40];
    char city[30];
    char state[3];
    char zip[12];
}
```

Then, to use list to generate objects that will store objects of type addr, use a declaration like this (assuming that structvar contains a valid addr structure):

Khi đó, dùng danh sách để tổng quát hóa các đối tượng mà sẽ chứa kiểu đối tượng addr, dùng khai báo như thế này (giả định rằng biến cấu trúc chứa cấu trúc địa chỉ hợp lệ)

```
list<addr> obj(structvar);
```

Here is another example of generic class. It is a reworking of the stack class first introduced in Chapter1. However, in this case, stack has been made into a template class. Thus, it can be used to store any type of object. In the example shown here, a character stack and a floating-point stack are created:

Ở đây là một ví dụ khác của lớp chung. Nó làm lại lớp ngăn xếp mà đã giới thiệu trong chương đầu. Tuy vậy, trong trường hợp này, ngăn xếp sẽ chuyển thành lớp chung. Vì vậy, nó có thể được dùng để chứa bất kỳ kiểu đối tượng nào. Trong ví dụ sau, ngăn xếp ký tự và ngăn xếp số chấm động sẽ được tạo ra:

```
// This function demonstrates a generic stack
#include <iostream>
using namespace std;

#define SIZE 10

// Create a generic stack class
template <class StackType> class stack {
    StackType stck[SIZE]; //holds the stack
    int tos;  //index of top of stack
```

```
public:
     void init()    { tos = 0; } //initialize stack
     void push (StackType ch);     // push object on
stack
     StackType pop();          // pop object from stack
};

//Push an object
template <class StackType>
void stack<StackType>::push(StackType ob)
{
     if(tos == SIZE)     {
          cout << "Stack is full. \n";
          return;
     }
     stck[tos] == ob;
     tos ++;
}

// Pop an object.
template <class StackType>
StackType stack<StackType>::pop()
{
     if(tos == 0)   {
          cout << "Stack is empty.\n";
```

```
            return 0; // Return Null or empty stack
        }
        tos--;
        return stck[tos];
    }

    int main()
    {
        // Demonstrate character stacks.
        stack<char> s1, s2; //Create two stacks
        int i;

        // Initialize the stacks
        s1.init();
        s2.init();

        s1.push('a');
        s2.push('x');
        s1.push('b');
        s2.push('y');
        s1.push('c');
        s2.push('z');

        for(i = 0; i < 3; i++)
        {
```

```
     cout << "Pop s1: " << s1.pop() << "\n";
}
for(i = 0; i < 3; i++)
{
     cout << "Pop s2: " << s2.pop() << "\n";
}

//Demonstrate double stacks
stack<double> ds1, ds2;  //create two stacks

// initialize the stacks
ds1.init();
ds2.init();

ds1.push(1.1);
ds2.push(2.2);
ds1.push(3.3);
ds2.push(4.4);
ds1.push(5.5);
ds2.push(6.6);

for(i = 0; i < 3; i++)
{
     cout << "Pop ds1: " << ds1.pop() << "\n";
}
```

```
        for(i = 0; i < 3; i++)
        {
              cout << "Pop ds2: " << ds2.pop() << "\n";
        }

        return 0;
}
```

As the stack class (and the preceding list class) illustrates, generic functions and classes provide a powerful tool that you can use to maximize your programming time because they allow you to define the general form of an algorithm that can be used with any type of data. You are saved from the tedium of creating separate implementations for each data type that you want the algorithm to work with.

Như lớp ngăn xếp (và lớp danh sách trước đó) minh họa, hàm tổng quát và lớp tổng quát cung cấp một công cụ hiệu quả mà bạn có thể dùng để tăng thời gian chương trình bởi vì chúng cho phép định nghĩa dạng chung của một giải thuật mà có thể dùng cho bất kỳ loại dữ liệu nào. Như thể bạn bổ sung kiểu dữ liệu mà bạn muốn dùng bằng chung giải thuật.

A template class can have more than one generic data type. Simply declare all the data types required by the class in a comma-separated list within the template specification. For example, the following short example creates a class that uses two generic data types:

Một lớp biểu mẫu có thể có hơn một kiểu dữ liệu chung. Thật đơn giản khai báo tất cả kiểu dữ liệu yêu cầu bởi lớp bằng danh sách phân cách bởi dấu phẩy bên trong sự chỉ định biểu mẫu. Ví dụ, ví dụ ngắn sau tạo ra một lớp mà dùng hai kiểu dữ liệu:

```
// This example uses two generic data types in a class
definition.

#include <iostream>

using namespace std;
```

```
template <class Type1, class Type2> class myclass
{
     Type1;
     Type2;
public:
     myclass(Type1 a, Type2 b) { i = a; j = b; }
     void show() {cout << i << ' ' << j << '\n'; }
};

int main()
{
     myclass<int, double> ob1(10, 0.23);
     myclass<char, char *> ob2('X', "This is a test");

     ob1.show();    //show int, double
     ob2.show();    //show char, char *

     return 0;
}
```

This program produces the following output:

```
10        0.23
 x        This is a test
```

The program declares two types of objects. ob1 uses integer and double data.

ob2 uses a character and a character pointer. For both cases, the compiler automatically generals the appropriate data and functions to accommodate the way the objects are created.

Chương trình khai báo 2 kiểu đối tượng ob1 dùng dữ liệu là số nguyên và số thực, ob2 dùng một ký tự và một con trỏ chuỗi ký tự. Cả hai trường hợp, trình biên dịch tự động tổng quát hóa dữ liệu và hàm thích hợp để cung cấp cách thức mà đối tượng được tạo ra.

Exercises

If you have not yet done so, compile and run the two generic class examples. Try declaring lists and/or stacks of different data types.

Nếu bạn chưa làm, hãy biên dịch và chạy thử ví dụ hai lớp trên. Thử khai báo các danh sách và/hay các ngăn xếp của nhiều kiểu dữ liệu khác nhau.

Create and demonstrate a generic queue class.

Tạo và minh họa lớp chung hàng đợi.

*Create a generic class, called **input**, that does the following when its constructor is called:*

- *prompts the user for input.*
- *inputs the data entered by the user, and*
- *reprompts if the data is not within a predetermined range.*

Tạo một lớp chung, mang tên **input**, mà có những phương thức thiết lập sau:

- ♣ Nhắc người dùng nhập
- ▪ Nhập dữ liệu bởi người dùng và
- ▪ Nhắc lại rằng dữ liệu không nằm trong vùng định nghĩa lại

Objects of type input should be declared like this: (Hàm có dạng sau:)

```
input ob("prompt message", min-value, max-value)
```

Here prompt message is the message that prompts for input. The minimum and maximum acceptable values are specified by min-value and max-value, respectively. (Note: the type of the data entered by the user will be the same as the type of min-value and max-value.)

Ở đây, lời nhắc là nhắc nhập. Các giá trị chấp nhận tối thiểu và cực đại là min-value và max-value. (Chú ý: kiểu dữ liệu mà nhập bởi người dùng sẽ cùng kiểu với min-value và max-value.

11.3. EXCEPTION HANDLING- ĐIỀU KHIỂN NGOẠI LỆ

*C++ provides a built-in error handling mechanism that is called exception handling. Using exception handling, you can more easily manage and respond to run-time errors. C++ exception handling is built upon three keywords: **try**, **catch** and throw. In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (i.e., an error) occurs within the **try** block, it is thrown (using throw). The exception is caught, using **catch**, and processed. The following elaborates upon this general description.*

C++ cung cấp một kỹ thuật điều kiểm lỗi gắn liền mà được gọi là điều khiển ngoại lệ. Dùng điều khiển này, bạn dễ dàng kiểm soát và phản ứng lại các lỗi trong khi chạy. Điều khiển ngoại lệ C++ được xây dựng trên ba từ khóa: **try, catch** and **throw**. Trong hầu hết các trường hợp tổng quát, lời phát biểu chương trình mà bạn muốn giám sát các ngoại lệ thì chứa trong một khóa **try**. Nếu một ngoại lệ(ví dụ như một lỗi) xảy ra bên trong khóa **try**, nó được ném đi (dùng **throw**). Ngoại lệ là bắt, dùng **catch**, và được xử lý. Những chi tiết sau dựa trên một sự mô tả tổng quát.

*As stated, any statement that throws an exception must have been executed from within a **try** block. (A function called from within a **try** block can also throw an exception.) Any exception must be caught by a **catch** statement that immediately follows the **try** statement that throws the exception. The general from of **try** and **catch** are shown here:*

Như đã nói, bất kỳ phát biểu nào mà ném một ngoại lệ phải được thực thi trong một khóa **try**. (Một hàm gọi từ trong một khóa **try** cũng có thể ném một ngoại lệ.) Bất kỳ một ngoại lệ nào phải bị bắt bởi phát biểu **catch** ngay sau lời phát biểu **try** mà ném một ngoại lệ. Dạng chung của **try** và **catch** được thể hiện dưới đây.

```
try{
     // try block
}
catch (type1 arg) {
     // catch block
}
catch (type2 arg) {
```

```
        // catch block
    }
    catch (type3 arg) {
        // catch block
    }
    .
    .
    .
    catch (typeN arg) {
        // catch block
    }
```

*The **try** block must contain the portion of your program that you want to monitor for errors. This can be as specific as monitoring a few statements within one function or as all-encompassing as enclosing the main() function code within a **try** block (which effectively causes the entire program to be monitored).*

Một khóa **try** phải chứa phần chương trình mà bạn muốn kiểm soát các lỗi. Nó có thể được chỉ định như sự kiểm soát vài phát biểu bên trong một hàm hay như hang rào bao bọc hàm main() bên trong một khóa **try** (mà gây ảnh hưởng cho chương trình được kiểm soát)

*When an exception is thrown, it is caught by its corresponding **catch** statement, which processes the exception. There can be more than one **catch** statement associated with a **try**. The **catch** statement that is used is determined by the type of the exception. That is, if the data type specified by a **catch** matches that of the exception, that **catch** statement is executed (and all others are bypassed). When an exception is caught, arg will receive its value. If you don't need access to the exception itself, specify only type in the **catch** clause-arg is optional. Any type of data can be caught, including classes that you create. In fact, class types are frequently used as exceptions.*

Khi một ngoại lệ được ném đi, nó bị bắt lại bởi lời phát biểu **catch** tương ứng mà xử lý ngoại lệ. Có thể có nhiều phát biểu **catch** kết hợp với một **try**. Lời phát biểu **catch** dùng quyết định kiểu ngoại lệ. Nếu kiểu đối tượng được chỉ rõ bởi một **catch** ứng với ngoại lệ

đó, thì lời phát biểu **catch** đó được thi hành (và tất cả cái khác sẽ bị bỏ qua). Khi một ngoại lệ bị bắt, arg sẽ nhận được giá trị của nó. Nếu bạn không cần truy xuất vào nó, thì việc chỉ rõ kiểu mệnh đề **catch** là không bắt buộc. Bất kỳ kiểu dữ liệu nào đều có thể bị bắt, bao gồm các lớp mà bạn tạo ra. Thực tế, kiểu lớp thì được dùng như những ngoại lệ.

The general form of the throw statement is shown here: (Dạng chung của câu lênh ném được trình bày ở đây)

throw exception;

*throw must be executed either from within the **try** block proper or from any function that the code within the block calls (directly or indirectly), exception is the value thrown.*

Ném phải được thi hành bên trong một khóa **try** đúng cách hay từ bất kỳ hàm nào mà mã bên trong khóa gọi (trực tiếp hay gián tiếp), ngoại lệ là giá trị được ném đi.

*If you throw an exception for which there is no applicable **catch** statement, an abnormal program termination might occur. If your compiler compiles with Standard C++, throwing an unhandled exception causes the standard library function **terminate()** to be invoked. By default, **terminate()** calls **abort()** to stop your program, but you can specify your own termination handler, if you like. You will need to refer to your compiler's library reference for details.*

Nếu bạn ném đi một ngoại lệ mà không có một phát biểu **catch** nào phù hợp, sự kết thúc chương trình bất thường có thể xảy ra. Nếu trình biên dịch của bạn biên dịch với C++ chuẩn, việc ném một ngoại lệ vô kiểm soát làm cho hàm **terminate()** trong thư viện hàm chuẩn sẽ được triệu hồi. Mặc định, hàm **terminate()** gọi hàm **abort()** để kết thúc chương trình của bạn, nhưng bạn có thể chỉ ra điều khiển kết thúc của chính mình, nếu muốn. Bạn sẽ cần tham khảo thêm chi tiết hướng dẫn trong thư viện trình biên dịch của mình.

Here is a simple example that shows the way C++ exception handling operates:

Ở đây là ví dụ đơn giản về cách mà điều khiển ngoại lệ C++ làm việc:

```
// A simple exception handling example.

#include <iostream>
```

```
using namespace std;

int main()
{
  cout << "start\n";

  try { // start a try block
        cout << "Inside try block\n";
        throw 10; // throw an error
        cout << "This will not execute";
  }
  catch (int i) {     // catch an error
        cout << "Caught One! Number is: ";
        cout << i << "\n";
        cout << "end";
  }

  return 0;
}
```

This program displays the following output: [Đây l‡ nội dung xuất ra]

```
start
Inside try block
Caught One! Number is: 10
```

```
end
```

*Look carefully at this program. As you can see, there is a **try** block containing three statements and a **catch(int i)** statement that processes an integer exception. Within the **try** block, only two of the three statements will execute: the first **cout** statement and the **throw**. Once an exception has been thrown, control passes to the **catch** expression and the **try** block is terminated. That is, **catch** is not called. Rather, program execution is transferred to it. (The stack is automatically reset as needed to accomplish this.) Thus, the **cout** statement following the **throw** will never execute.*

Nhìn thật kỹ chương trình. Bạn có thể thấy là, có một khóa **try** chứa ba câu lệnh và một phát biểu **catch(int i)** mà xử lý một ngoại lệ số nguyên. Bên trong một khóa **try**, chỉ có hai trong ba dòng phát biểu được thi hành: đầu tiên là lệnh **cout** và **throw**. Một khi một ngoại lệ đã được ném đi, điều khiển nhảy đến biểu thức **catch** và khóa **try** kết thúc. Vì lúc đó không có **catch** nào được gọi thế là chương trình thực thi đi qua. (Ngăn xếp sẽ tự động khởi động lại để hoàn tất nó.) Vì vậy, cậu lệnh **cout** tiếp theo **throw** sẽ không bao giờ được thực thi.

*After the **catch** statement executes, program control continues with the statements following the **catch**. Often, however, a **catch** block will and with a call to **exit()**, **abort(),** or some other function that causes program termination because exception handling is frequently used to handle catastrophic errors.*

Sau một phát biểu **catch** thi hành, điều khiển chương trình tiếp tục với các phát biểu sau **catch**. Tuy nhiên, thường là một khóa **catch** sẽ và gọi hàm **exit(), abort()** hay một hàm nào đó mà gọi cơ chế ngắt bởi điều khiển ngoại lệ thường dùng cho các lỗi nguy hiểm.

*As mentioned, the typed of the exception must match the type specified in a **catch** statement. For example, in the preceding example, if you change the type in the **catch** statement to **double**, the exception will not be caught and abnormal termination will occur. This change is shown here:*

Như đã đề cập, kiểu ngoại lệ phải gắn với kiểu của phát biểu **catch**. Ví dụ, ví dụ trước, nếu bạn thay đổi phát biểu **catch** thành kiểu **double**, thì ngoại lệ sẽ không được bắt và kết thúc bất thường sẽ xảy ra. Sự thay đổi như sau:

```
// This example will not work

#include <iostream>

using namespace std;
```

```
#include <iostream>
using namespace std;

int main()
{
    cout << "start\n";

    try { // start a try block
      cout << "Inside try block\n";
      throw 10; // throw an error
      cout << "This will not execute";
    }
    catch(double i){//won't work for an int exception
      cout << "Caught One! Number is: ";
      cout << i << "\n";
    }
    cout << "end";

    return 0;
}
```

*This program procedures the following output because the integer exception will not be caught by a **double catch** statement.*

Chương trình xuất ra kết quả sau bởi vì một ngoại lệ số nguyên (integer) không thể dùng phát biểu **catch** với số dạng **double.**

```
start
Inside try block
```

```
Abnormal program terminataion

//Báo lỗi kết thúc bất thường
```

*An exception can be thrown from a statement that is outside the **try** block as long as the statement is within a function that is called from within the **try** block. For example, this is a valid program:*

Một ngoại lệ có thể được ném từ một phát biểu bên ngoài khóa **try** như lời phát biểu bên trong một hàm mà đươc gọi bên trong một khóa **try**. Ví dụ, chương trình sau là hợp lệ:

```
     /* Throwing an exception from a function outside
the try block */

     #include <iostream>

    using namespace std;


    void Xtest( int test)

    {

     cout <<"Inside Xtest, test is: " << test <<
"\n";

     if(test) throw test;

    }


    int main()

    {

     cout << "start\n";


     try { // start a try block

          cout << "Inside try block\n";

          Xtest(0);
```

```
        Xtest(1);
        Xtest(2);
    }
    catch(int i){//catch an error
        cout << "Caught One! Number is: ";
        cout << i << "\n";
    }

    cout << "end";

    return 0;
}
```

This program produces the following output: [Xuất ra như sau]

```
start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught One! Number is: 1
end
```

Vừa ném 1 là lập tức thoát ra hàm **try** bởi không có phát biểu **catch** nào tiếp sau -> Xtest(3) không được thi hành

*A **try** block can be localized to a function. When this is the case, each time the function is entered the exception handling relative to that function is reset. For example, examine this program:*

Một khóa **try** có thể cục bộ hóa trong một hàm. Khi là trường hợp sau, mỗi lần hàm đi vào, điều khiển ngoại lệ liên quan với hàm bị khởi động lại. Ví du như sau:

```
#include <iostream>
using namespace std;

// A try catch can be inside a function other than
main()
void Xhandler( int test)
{
    try {
     if(test) throw test;
    }
    catch(int i) {
     cout << "Caught One! Ex. #: " << i << "\n";
    }
}

int main()
{
    cout << "start\n";

    Xhandler(0);
    Xhandler(1);
    Xhandler(2);
    Xhandler(3);

    cout << "end";
```

```
    return 0;
}
```

This program displays this input:

```
start
Caught One! Ex. #: 1
Caught One! Ex. #: 2
Caught One! Ex. #: 3
end
```

As you can see, three exceptions are thrown. After each exception, the function returns. When the function is called again, the exception handling is reset.

Như bạn thấy đó, ba ngoại lệ đều được ném. Sau mỗi ngoại lệ hàm trả về. Khi hàm gọi lại, điều khiển ngoại lệ bị khởi động lại.

*As stated earlier, you can have more than one **catch** associated with a **try**. In fact, it is common to do so. However, each **catch** must catch a different type of exception. For example, the following program catches both integers and strings:*

Như đã nói từ sớm, bạn có thể có nhiều **catch** tương ứng với một **try**. Trong thực tế, làm vậy là phổ biến. Tuy thế, mỗi **catch** phải bắt các ngoại lệ khác kiểu. Ví dụ, chương trình sau bắt cả số nguyên và chuỗi:

```
#include <iostream>
using namespace std;

// Different types of exceptions can be caught.
void Xhandler( int test)
{
    try {
      if(test) throw test;
```

```
    else throw "Value is zero";
  }
  catch(int i) {
    cout << "Caught One! Ex. #: " << i << "\n";
  }
  catch(char *str) {
    cout << "Caught a string:  " << str << "\n";
  }

}

int main()
{
  cout << "start\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);
  Xhandler(3);

  cout << "end";

  return 0;
}
```

*This program displays this input:[*Xuất ra như sau]

```
start

Caught One! Ex. #: 1

Caught One! Ex. #: 2

Caught a string:  Value is zero

Caught One! Ex. #: 3

end
```

*As you can see, each **catch** statement responds only to its own type. In general, **catch** expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other **catch** blocks are ignored.*

Bạn thấy đấy, mỗi phát biểu **catch** chỉ đáp lại với cùng lại. Trong tổng quát, biểu thức **catch** thì kiểm tra theo trật tự xảy ra. Chỉ phát biểu tương ứng được thi hành. Tất cả khóa **catch** thì bị lờ qua.

Exercises - Bài tập

1. *By far, the best way to understand how C++ exception handling works is to play with it. Enter, compile, and run the preceding example programs. Then experiment with them, altering pieces of them and observing the results.*

Cách hiểu tốt nhất điều khiển ngoại lệ C++ là chơi với nó. Đừng lười hãy thử gõ vào, biên dịch và chạy thử là hiểu thôi. Sau đó nâng cao lên làm với các kiểu khác và xem kết quả và rút ra suy nghĩ.

2. *What is wrong with this fragment?*

Có gì sai trong đoạn sau

```
int main()

{

      throw 12.23;

}
```

3. What is wrong with this fragment?

Có gì sai trong đoạn sau

```
try {
      // ….
      throw 'a';
      // …
}
catch(char *)  {
//….
}
```

*4. What will happen if an exception is thrown for which there is no corresponding **catch** statement?*

Điều gì sẽ xảy ra khi một ngoại lệ được ném mà không có phát biểu **catch** nào đáp ứng lại

11.4. MORE ABOUT EXCEPTION HANDLING - TRÌNH BÀY THÊM VỀ ĐIỀU KHIỂN NGOẠI LỆ

There are several additional features and nuances to C++ exception handling that can make it easier and more convenient to use.

Trong việc điều khiển ngoại lệ C++ còn nhiều đặc tính và nét tinh tế giúp cho việc sử dụng nó được dễ dàng và tiện lợi hơn.

*In some circumtances you will an exception handler to catch all exceptions instead of just a certain type. This is easy to accomplish. Simply use this form of **catch**:*

Trong một số trường hợp, bạn sẽ cần một bộ điều khiển ngoại lệ có thể bắt được tất cả

các kiểu của ngoại lệ thay vì chỉ một kiểu nào đó. Điều này được thực hiện dễ dàng bằng cách sử dụng mệnh đề **catch** theo dạng như sau:

```
catch(…)
{
      //process all exceptions
}
```

Here the ellipsis matches any type of data.

Ở đây, dấu ba chấm … bắt được tất cả các kiểu.

*You can restrict the type of exceptions that a function can throw back to its caller. Put differently, you can control what type of exceptions a function can throw outside of itself. In fact, you can also prevent a function from throwing any exceptions whatsoever. To apply these restrictions, you must add a **throw** clause to the function definition. The general form of this is shown here:*

Khi một hàm được gọi từ bên trong khối **block**, bạn có thể hạn chế kiểu dữ liệu nào của ngoại lệ mà hàm có thể ném được. Thực vậy, bạn có thể cản trở hàm nén ra bất kỳ kiểu dữ liệu nào. Để thực hiện các hạn chế, bạn phải thêm **throw** vào định nghĩa hàm. Dạng tổng quát như sau:

```
ret-type func-name(arg-list) throw(type-list)
{
      //…
}
```

Here only those data types contained in the comma-separated type-list may be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don't want a function to be able to throw any exceptions, use an empty list.

Ở đây, chỉ có các ngoại lệ có kiểu trong danh sách *type_list* (phân cách bằng dấu phẩy) là được ném ra từ hàm. Ngược lại, việc ném ra các ngoại lệ thuộc kiểu khác là chương trình kết thúc bất thường.

*If your compiler complies with Standard C++, when a function attempts to throw a disallowed exception the standard library function **unexpected()** is call. By default, this causes the **terminate()** function to be called, which causes abnormal program*

termination. However, you can specify your own termination handler, if you like. You will need to refer to your compiler's documentation for directions on how this can be accomplished.

Nếu trình biên dịch của bạn tuân theo chuẩn trù định của tiêu chuẩn C++, thì việc thử ném ra một ngoại lệ mà không có mệnh đề **catch** nào bắt được sẽ làm gọi hàm **unexpected().** Theo mặc định, hàm **unexpected()** sẽ gọi tiếp hàm **abort()** để gây kết thúc chương trình một cách bất thường. Tuy nhiên, bạn hoàn toàn có thể định lại để hàm **unexpected()** gọi đến một hàm xử lý kết thúc khác nếu bạn thích. Muốn biết chi tiết, bạn hãy kiểm tra lại tài liệu hướng dẫn sử dụng thư viện của trình biên dịch.

*If you wish to rethrow an expression from within an exception handler, you can do so by simply calling **throw**, by itself, with no exception. This causes the current exception to be passed on to an outer **try/catch** sequence.*

Nếu muốn ném một lần nữa một ngoại lệ từ bên trong bộ điều khiển ngoại lệ, bạn gọi một lần nữa từ khóa **throw** không cần kèm theo giá trị của ngoại lệ. Cách này cho phép ngoại lệ hiện hành được truyền đến một đoạn **try/catch** khác.

EXAMPLES - VÍ DỤ

*The following program illustrates **catch(…)**:*

Chương trình sau đây minh họa cách dùng **catch(…)**:

```
// This example catches all exceptions.
#include <iostream>
using namespace std;


void Xhandler(int test)
{
     try
     {
          if(test==0)
               throw test; // throw int
```

```
            if(test==1)
                  throw 'a'; // throw char
            if(test==2)
                  throw 123.23; // throw double
      }
      catch(...)
      {
            // catch all exceptions
            cout << "Caught One!\n";
      }
}

int main()
{
      cout << "start\n";

      Xhandler (0);
      Xhandler (1);
      Xhandler (2);

      cout << "end";

      return 0;
}
```

This program displays the following output:

Chương trình xuất ra kết quả như sau:

```
start
Caught One!
Caught One!
Caught One!
End
```

*As you can see, all three **throws** were caught using the one **catch** statement.*

Trong chương trình trên, bạn có thể thấy là cả ba lần ném ngoại lệ đều được bắt lại được bởi cùng chung một bộ điều khiển ngoại lệ **catch**.

*One very good use for **catch(…)** is as the last **catch** of a cluster of catches. In this capacity it provides a useful default or "catch all" statement. For example, this slightly different version of the relies upon **catch(…)** to catch all others:*

Một cách sử dụng **catch(…)** rất hay là đưa nó vào cuối các bộ điều khiển ngoại lệ **catch** khác trong chương trình. Bộ điều khiển ngoại lệ **catch(…)** có khả năng bắt được ngoại lệ thuộc bất kỳ kiểu nào. Trong ví dụ sau, **catch(...)** được dùng để bắt tất cả các ngoại lệ có kiểu khác kiểu nguyên, chương trình ví dụ này là một bản hiệu chỉnh lại từ chương trình ví dụ ở trên:

```
// This example uses catch(...) as a default.
#include <iostream>
using namespace std;
```

```
void Xhandler (int test)
{
     try
     {
          if(test==0)
               throw test; // throw int
          if(test==1)
               throw 'a'; // throw char
          if(test==2)
               throw 123.23; // throw double
     }
     catch (int i)
     {
          // catch an int exception
          cout << "Caught" << i << '\n';
     }
     catch (...)
     {
          // catch all other exceptions
          cout << "Caught One!\n";
     }
}

int main()
```

```
{
     cout << "start\n";
     Xhandler (0);
     Xhandler (1);
     Xhandler (2);

     cout << "end";

     return 0;
}
```

The output produced by this program is shown here:

Chương trình xuất ra kết quả như sau:

```
start
Caught 0
Caught One!
Caught One!
end
```

*As this example suggests, using **catch(…)** as a default is a good way to catch all exceptions that you don't want to handle explicitly. Also, by catching all exceptions, you prevent an unhandled exception from causing an abnormal program termination.*

Ví dụ trên cho thấy **catch(…)** được dung để tạo ra một bộ xử lý lỗi mặc định có thể bắt tất cả các ngoại lệ ngoài các ngoại lệ mà ta đã quan tâm và xử lý một cách tường minh. Hơn nữa, cách sử dụng này còn giúp chúng ta ngăn ngừa được việc ngoại lệ ném ra mà không có bộ xử lý nào bắt được, làm dẫn đến kết thúc chương trình một cách bất thường.

The following program shows how to restrict the types of exceptions that can be thrown from a function:

Chương trình tiếp theo minh họa cách hạn chế các kiểu của ngoại lệ được phép ném ra từ trong một hàm:

```
// Restricting function throw types.
#include <iostream>
using namespace std;

// This function can only throw ints, chars, and
doubles.
void Xhandler (int test) throw (int, char, double)
{
     if(test==0)
          throw test; // throw int
     if(test==1)
          throw 'a'; // throw char
     if(test==2)
          throw 123.23; // throw double
}
```

```
int main()
{
     cout << "start\n";

     try
     {
        Xhandler (0); // also, try passing 1 and 2
        to Xhandler()
     }
     catch(int i)
     {
          cout << "Caught int\n";
     }
     catch(char c)
     {
          cout << "Caught char\n";
     }
     catch(double d)
     {
          cout << "Caught double\n";
     }

     cout << "end";

     return 0;
}
```

*In this program, the function **Xhandler()** can throw only integer, character, and **double** exceptions. If it attempts to throw any other type off exception, an abnormal program termination will occur. (That is, **unexpected()** will be called.) To see an example of this, remove **int** from the list and retry the program.*

Trong chương trình trên, hàm **Xhandler()** chỉ có thể ném ra các ngoại lệ thuộc kiểu nguyên, ký tự và **double**. Việc ném thử các ngoại lệ thuộc kiểu khác với ba kiểu nói trên sẽ làm chương trình kết thúc bất thường. (Tức là gọi hàm **unexpected()**). Để thử điều đó, hãy xóa từ **int** ra khỏi danh sách và thực hiện lại chương trình một lần nữa.

*It is important to understand that a function can only be restricted in what types of exceptions it throws back to the **try** block that called it. That is, a **try** block within a function can throw any type of exception so long as it is caught within that function. The restriction applies only when throwing an exception out of the function.*

Điều quan trọng mà bạn cần biết là một hàm chỉ bị hạn chế kiểu ngoại lệ được ném ra từ khối **try** mà hàm này được gọi. Nói cách khác là, một khối **try** đặt bên trong một hàm chỉ có thể ném bất kỳ ngoại lệ nào mà chúng bị bắt bởi các bộ xử lý **catch** cũng nằm bên trong hàm đó. Việc hạn chế kiểu chỉ có tác dụng khi ngoại lệ được ném ra khỏi hàm.

*The following change to **Xhandler()** prevents if from throwing any exceptions:*

Nội dung mới của hàm **Xhandler()** giúp nó ngăn ngừa việc ném bất kỳ ngoại lệ nào

```
// This function can throw NO exceptions!
```

```
void Xhandler (int test) throw ()
{
   /* The following statements no longer work.
   Instead,     they will cause an abnormal program
   termination.*/
     if(test==0)
          throw test;
     if(test==1)
          throw 'a';
     if(test==2)
          throw 123.23;
}
```

*As you have learned, you can rethrow an exception. The most likely reason for doing so is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a **catch** block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same **catch** statement. It will propagate to an outer **catch** statement. The following program illustrates rethrowing an exception. It rethrows a **char *** exception.*

Như bạn đã học, ta có thể ném lần nữa một ngoại lệ. Nguyên nhân thường gặp của việc ném lại này là cho phép bộ xử lý truy cập nhiều lần một ngoại lệ. Lấy ví dụ, có thể là một bộ xử lý thực hiện một vấn đề nào đó của ngoại lệ, trong khi bộ xử lý thứ hai đối phó với một vấn đề khác của ngoại lệ đó. Ngoại lệ chỉ có thể bị ném ra lần nữa từ bên trong bộ điều khiển ngoại lệ (hay gián tiếp từ một hàm gọi từ bộ điều khiển đó). Khi ngoại lệ bị ném lần nữa, nó không bị bắt lại bởi cùng một bộ điều khiển ngoại lệ **catch**, thay vào đó là một bộ điều khiển ngoại lệ khác ở vòng ngoài. Chương trình sau đây sẽ minh họa cho vấn đề trên, nó ném lại một ngoại lệ kiểu **char***:

```
// Example of rethrowing- an exception.
```

```
#include <iostream>
using namespace std;

void Xhandler()
{
     try
     {
          throw "hello"; // throw a char *
     }
     catch (char *)
     {
          // catch a char *
          cout << "Caught char * inside Xhandler\n";
          throw; // rethrow char * out of function
     }
}

int main()
{
     cout << "start\n";

     try
     {
          Xhandler();
     }
```

```
      catch (char *)
      {
            cout << "Caught char * inside main\n";
      }

      cout << "end";

      return 0;
}
```

This program displays the following output:

Chương trình xuất ra kết quả như sau:

```
start
Caught char * inside Xhandler
Caught char * inside main
End
```

EXERCISES

BÀI TẬP

Before continuing, compile and run all of the examples in this section. Be sure you understand why each program produces the output that it does.

Trước tiên, bạn hãy biên dịch và cho thực thi tất cả các ví dụ trong đề mục này. Hãy đảm bảo là bạn hiểu được hoạt động của chương trình, và vì sao chúng in

ra các kết xuất như vậy.

What is wrong with this fragment?

Có gì không đúng trong đoạn chương trình sau?

```
try
{
      //...
      throw 10;
}
catch (int *p)
{
      //...
}
```

Show one way to fix the preceding fragment.

Hãy sửa lại đoạn chương trình trên cho đúng.

*What **catch** expression catches all types of exceptions?*

Biểu thức **catch** nào bắt được tất cả các kiểu của ngoại lệ?

*Here is a skeleton for a function called **divide()**.*

Dưới đây là phần khung của một hàm có tên **divide()**.

```
double divide (double a, double b)
{
     // add error handling
     return a/b;
}
```

*This function returns the result of dividing **a** by **b**. Add error checking to this function using C++ exception handling. Specifically, prevent a divide-by-zero error. Demonstrate your solution in a program.*

Hàm này trả về giá trị của phép toán chia **a** cho **b**. Bạn hãy thêm vào hàm **divide()** chức năng kiểm tra lỗi. Đặc biệt là lỗi sai vì chia cho giá trị 0. Trình diễn giải pháp của bạn trong một chương trình.

11.5. HANDLING EXCEPTIONS THROWN - SỬ DỤNG NHỮNG NGOẠI LỆ ĐƯỢC NÉM

*In Chapter 4 you learned that the modern specification for the **new** operator states that it will throw an exception if an allocation request fails. Since in Chapter 4 exceptions had not yet been discussed, a description of how to handle that exception was deferred until later. Now the time has come to examine precisely what occurs when **new** fails.*

Trong Chương 4 bạn học đăc tả hiện đại cho toán tử **new** những trạng thái thao tác mà nó sẽ ném một ngoại lệ nếu một yêu cầu định vị thất bại. Từ đó trong Chương 4 những ngoại lệ chưa được được bàn luận, sự mô tả xử lý ngoại lệ đó như thế nào được trì hoãn cho đến sau này. Bây giờ thời gian để nghiên cứu chính xác cái mà xảy ra khi **new** thất bại.

*Before we begin, it is necessary to state that the material in this section describes the behavior of **new** as specified by Standard C++. As you should recall from Chapter 4, the precise action that **new** takes on failure has been changed several times since C++ was invented. Specifically, when C++ was*

*first invented, **new** returned null on failure. Later this was changed such that **new** caused an exception on failure. Also, the name of this exception has changed over time. Finally, it was decided that a **new** failure will generate an exception by default, but that a null pointer could be returned instead, as an option. Thus, **new** has been implemented differently, at different times, by compiler manufactures. Although all compilers will eventually implement **new** in compliance with Standard C++, not all currently do. If the code examples shown here do not work with your compiler, check your compiler's documentation for details on how it implements **new**.*

Trước khi chúng ta bắt đầu, nó thật sự cần thiết cho trạng thái của vật liệu trong khu vực này mô tả hành vi của **new** được chỉ rõ bởi chuẩn C++. Hãy nhớ lại chương 4, hoạt động chính xác của **new** gặp những thất bại đã được thay đổi vài lần từ khi C++ được phát minh. Đặc biệt, khi C++ được phát minh lần đầu tiên, **new** trả lại giá trị null trên sự thất bại. Sau đây sự thay đổi sao cho **new** gây ra một ngoại lệ trên sự thất bại. Đồng thời, tên của ngoại lệ này cũng thay đổi theo thời gian. Cuối cùng, nó được quyết định rằng **new** thất bại sẽ phát sinh một ngoại lệ mặc định, trừ khi một con trỏ null có thể trở lại thay vào đó, như một tùy chọn. Như thế, **new** đã được thi hành khác nhau, vào những thời điểm khác nhau, bởi sự sản xuất của trình biên dịch. Mặc dù tất cả các trình biên dịch dần dần sẽ thực thi **new** đúng theo chuẩn C++, không phải như mọi thứ hiện thời làm. Nếu những đoạn mã ví dụ ở đây không làm việc với trình biên dịch của bạn, hãy kiểm tra lại tài liệu thư viện của trình biên dịch của bạn những chi tiết làm thế nào để thi hành **new**.

*In Standard C++, when an allocation request cannot be honored, **new** throws a **bad_alloc** exception. If you don't catch this exception, your program will be terminated. Although this behavior is fine for short sample programs, in real applications you must catch this exception and process it in some rational manner. To have access to this exception, you must include the header <**new**> in your program.*

Trong chuẩn C++, khi một yêu cầu định vị không thể được khen ngợi, **new** ném một ngoại lệ **bad_alloc**. Nếu bạn không bắt lấy ngoại lệ này, chương trình của bạn sẽ chấm dứt. Mặc dù hành vi này tốt cho những chương trình gắn, trong những ứng dụng thật sự bạn phải bắt ngoại lệ này và xứ lý nó vào trong dạng hợp lý. Để truy cập ngoại lệ này, bạn phải bao gồm đầu mục <**new**> trong chương trình của bạn.

Note *Originally this exception was called **xalloc**, and at the time of this writing many compilers still use the older name. However,*

***bad_alloc** is the name specified by Standard C++, and it is the name that will be used in the future.*

***Chú ý** Trước đây ngoại lệ n‡y được gọi l‡ **xalloc**, v‡ lúc đó cách viết n‡y được nhiều trình biên dịch sử dụng với tên cũ hơn. Ng‡y nay, **bad_alloc** l‡ tên được chỉ rõ bởi chuẩn C++, v‡ nó l‡ tên sẽ được sử dụng trong tương lai.*

*In Standard C++ it is also possible to have **new** return null instead of throwing an exception when an allocation failure occurs. This form of **new** is most useful when you are compiling older code with a modern C++ compiler. It is also valuable when you are replacing calls to **malloc()** with **new**. This form of **new** is shown here.*

Trong chuẩn C++ nó có thể xảy ra để **new** trả lại con trỏ null thay vì việc ném một ngoại lệ khi một định vị thất bại xuất hiện. Mẫu này của **new** là hữu ích nhất khi bạn là một trình biên dịch mã cũ hơn một trình biên dịch C++ hiện đại. Nó có thêm giá trị khi bạn thay thế việc gọi **malloc()** bằng **new**. Dạng này của **new** được cho thấy ở đây.

```
p_var = new(nothrow) type;
```

*Here p_var is a pointer variable of type. The **nothrow** form of **new** works like the original version of **new** from years ago. Since it returns null on failure, it can be "dropped into" older code and you won't have to add exception handling. However, for new code, exceptions provide a better alternative.*

Ở đây *p_var* là một biến con trỏ của *type*. **Nothrow** hình thành từ **new** những công việc như phiên bản nguyên bản của **new** cách đây vài năm. Từ khi nó trả lại con trỏ null trên sự thất bại, nó có thể "rơi vào trong" mã cũ và bạn sẽ không phải thêm xử lý ngoại lệ. Tuy nhiên, cho mã mới, những ngoại lệ cung cấp những giải pháp tốt hơn.

EXAMPLES - VÍ DỤ

*Here is an example of **new** that uses a **try/catch** block to monitor for an allocation failure.*

Ở đây là một ví dụ của **new** sử dụng khối **try/catch** để theo dõi một sự định vị thất bại.

```
#include <iostream>
#include <new>
using namespace std;

int main()
{
     int *p;

     try
     {
          p = new int; // allocate memory for int
     }
     catch (bad_alloc xa)
     {
          cout << "Allocation failure.\n";
          return 1;
     }

     for (*p = 0; *p < 10; (*p)++)
          cout << *p << "";

     delete p; // free the memory

     return 0;
```

```
}
```

*Here if an allocation failure occurs, it is caught by the **catch** statement.*

Ở đây nếu một sự định vị thất bại xuất hiện, nó sẽ bị bắt bởi sự phát biểu **catch**.

*Since the previous program is unlikely to fail under any normal circumstance, the following program demonstrates **new's** exception-throwing capability by forcing an allocation failure. It does this by allocating memory until it is exhausted.*

Một khi chương trình trước đây không chắc để thất bại dưới bất kỳ hoàn cảnh bình thường nào, chương trình sau đây trình diễn sự quăng ngoại lệ của **new** bởi việc bắt buộc một sự định vị thất bại. Nó làm điều này bởi sự cấp phát bộ nhớ cho đến khi nó kiệt quệ.

```
// Force an allocation failure.
#include <iostream>
#include <new>
using namespace std;

int main()
{
     double *p;

     // this will eventually run out of memory
     do
     {
```

```
            try
            {
                  p = new double [100000];
            }
            catch (bad_alloc xa)
            {
                  cout << "Allocation failure.\n";
                  return 1;
            }
            cout << "Allocation OK.\n";
      }
      while(p);

      return 0;
}
```

*The following program shows how to use the **new(nothrow)** alternative. It reworks the preceding program and forces an allocation failure.*

Chương trình sau đây chỉ ra như thế nào để sử dụng giải pháp **new(nothrow)**. Nó reworks có trước lập trình và bắt buộc một sự định vị thất bại.

```
// Demonstrate the new(nothrow) alternative.
#include <iostream>
#include <new>
using namespace std;
```

```
int main()
{
      double *p;

      // this will eventually run out of memory
      do
      {
            p = new (nothrow) double [100000];
            if(p)
                  cout << "Allocation OK.\n";
            else
                  cout << "Allocation Error.\n";
      }
      while(p);

      return 0;
}
```

*As this program demonstrates, when you use the **nothrow** approach, you must check the pointer returned by **new** after each allocation request.*

Như chương trình này trình diễn, khi bạn sử dụng cách tiếp cận **nothrow**, bạn phải kiểm tra con trỏ được trả lại bởi **new** sau mỗi yêu cầu định vị

EXERCISES - BÀI TẬP

*Explain the difference between the behavior of **new** and **new(nothrow)** when an allocation failure occurs.*

Giải thích sự khác nhau giữa hành vi của **new** và **new(nothrow)** khi một sự định vị thất bại xuất hiện.

Given the following fragment, show two ways to convert it into modern C++-style code.

Cho đoạn sau đây, hãy cho biết hai cách để cho nó chuyển đổi vào trong mã C++ kiểu hiện đại.

```
p = malloc(sizeof (int));


if(!p)
{
      cout << "Allocation error.\n";
      exit(1);
}
```

SKILLS CHECK

KIỂM TRA KỸ NĂNG

Mastery Skills Check

Kiểm tra kỹ năng lĩnh hội

At this point you should be able to perform the following exercises and answer the questions.

Đến đây bạn có thể làm các bài tập và trả lời các câu hỏi sau:

Create a generic function that returns the mode of an array of values. (The mode of a set is the value that occurs most often.)

Hãy tạo ra một hàm chung trả về giá trị mode của một mảng. (Giá trị mode là giá trị bắt gặp nhiều nhất trong dãy)

Create a generic function that returns the summation of an array of values.

Hãy tạo hàm tính và trả về giá trị tổng của các phần tử trong một mảng.

Create a generic bubble sort (or use any other sorting algorithm you like).

Hãy tạo hàm chung thực hiện giải thuật sắp xếp nổi bọt (hay bất kỳ giải thuật sắp xếp nào khác mà bạn thích).

*Rework the **stack** class so that it can store pairs of different-type objects on the stack. Demonstrate your solution.*

Định lại lớp **stack** để nó có lưu trữ một cặp gồm hai đối tượng có kiểu dữ liệu khác nhau. Viết chương trình biểu diễn lớp **stack** mới này.

*Show the general forms of **try**, **catch**, and **throw**. In your own words, describe their operation.*

Trình bày dạng sử dụng của **try**, **catch**, và **throw**. Hãy mô tả cách làm việc của chúng.

*Again, rework the **stack** class so that stack over-and underflows are handled as exceptions.*

Một lần nữa, bạn hãy định lại lớp **stack** để nó có khả năng xử lý các lỗi: chồng bị đầy, và chồng bị rỗng.

*Check your compiler's documentation. See whether it supports the **terminate()** and **unexpected()** functions. Generally, these functions can be configured to call any function you choose. If this is the case with your compiler, try creating your own set of customized termination functions that handle otherwise unhandled exceptions.*

Hãy kiểm tra các tài liệu hướng dẫn liên quan đến thư viện của trình biên dịch của bạn. Để tìm xem nó có hỗ trợ hai hàm **terminate()** và **unexpected()** hay không? Nói chung, chúng ta có thể định lại cho hai hàm này gọi đến bất kỳ hàm nào bạn muốn. Nếu được như vậy, bạn hãy tạo ra một bộ các hàm kết thúc chương trình của riêng mình để xử lý các lỗi bất ngờ.

*Thought question: Give a reason why having **new** generate an exception is a better approach than having **new** return null on failure.*

Câu hỏi tư duy: Hãy giải thích tại sao **new** sinh ra một ngoại lệ thì tốt hơn cách tiếp cận **new** trả về một con trỏ null trên sự thất bại.

Cumulative Skills Check

Kiểm tra kỹ năng tổng hợp

This section checks how well you have integrated material in this chapter with that from preceding chapters.

Phần này kiểm tra xem khả năng của bạn kết hợp chương này với chương trước như thế nào.

In Chapter 6, Section 6.7, Example 3.a safe array class was shown. On your own, convert it into a generic safe array.

Ở ví dụ 3, mục 6.7, trong chương 6 có minh họa lớp của dãy có số lượng hữu hạn các phần tử. Hãy chuyển lớp này thành một lớp chung.

*In Chapter 1, overloaded versions of the **abs()** function were create. As a better solution, create a generic **abs()** function on your own that will return the absolute value of any numeric object.*

Trong chương 1, có một chương trình quá tải hàm **abs()**. Để thực hiện một cách giải tốt hơn, bạn hãy tạo ra một hàm chung **abs()**, nó tính và trả về giá trị tuyệt đối của một giá trị số thuộc bất kỳ kiểu nào.

CHAPTER 12

RUN-TIME TYPE IDENTIFICATION AND THE CASTING OPERATORS – KIỂU THỜI GIAN THỰC VÀ TOÁN TỬ ÉP KHUÔN

Chapter Objectives

Cách Dùng **dynamic_cast**

12.3 USING **CONST_CAST, REINTERPRET_CAST,** AND **STATIC_CAST.**

Cách Dùng **const_cast, reinterpret_cast,** và **static_cast.**

This chapter discusses two features that were recently added to the C++ language: run-time type identification (RTTI for short) and the new casting operators. RTTI allows you to identify the type of an object during the execution of your program. The casting operators give you safer, more controlled ways to cast. As you will see, one of the casting operators ***dynamic_cast****, relatives directly to RTTI, so it makes sense to discuss these two subjects in the same chapter.*

Trong chương này chúng ta sẽ thảo luận 2 vấn đề được thêm vào trong ngôn ngữ C++: sự nhận ra kiểu thời gian thực (RTTI đơn giản) và toán tử khuôn mẫu. RTTI cho phép bạn xác định kiểu của đối tượng đang thi hành trong chương trình của bạn. Toán tử khuôn mẫu đưa cho bạn một cách an toàn hơn để kiểm soát các phương thức dạng khuôn mẫu. Như bạn biết, một trong những toán tử khuôn mẫu là **dynaminc_cast**, có liên quan trực tiếp với RTTI, vì vậy nó là hai chủ đề được thảo luận trong chương này.

Review Skills Check

Kiểm tra Kĩ năng lĩnh hội

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Trước khi tiếp tục, bạn nên trả lời chính xác những câu hỏi và làm những bài tập sau:

1. What is a generic function and what is its general form?

Hàm chung là gì? Dạng tổng quát của nó?

2. What is a generic class and what is its general form?

Lớp chung là gì? Dạng tổng quát của lớp chung?

*3. Write a generic function called **gexp()** that returns the value of one of its arguments raised to the power of the other.*

Viết một hàm chung tên **gexp()** trả về giá trị là kết quả của phép tính lũy thừa, trong đó đối số này là lũy thừa của đối số kia.

*4. In Chapter 9, Section 9.7, Example 1, a **coord** class that holds integer coordinates was created and demonstrated in a program Create a generic version of the **coord** class that can hold coordinates of any type. Demonstrate your solution in a program.*

Trong ví dụ 1 ở đề mục 9.7 trong chương 9 lớp **coord** lưu số các tọa độ là số nguyên được phát sinh và biểu diễn trong một chương trình. Bạn hãy tạo ra lớp chung **coord** có thể lưu tọa độ của một kiểu dữ liệu bất kì. Viết chương trình minh họa cho lớp trên.

*5. Briefly explain how **try, catch,** and **throw** work together to provide C++ exception handling.*

Hãy giải thích ngắn gọn về cách mà **try, catch,** và **throw** được sử dụng xử lý ngoại lệ trong C++

6. Can throw be used if execution has not passed through a try block?

Ta có thể dùng **throw** được hay không nếu như nó không được gọi từ bên trong một khối **try**?

*7. What purpose do **terminate()** and **unexpected()** serve?*

Công dụng của hàm **terminate()** và **unexpected()** là gì?

*8. What form of **catch** will handle all types of exceptions?*

Dạng sử dụng như thế nào của **catch** sẽ bắt được tất cả các kiểu của ngoại lệ?

13.1. UDERSTANDING RUN-TIME TYPE IDENTIFICATION (RTTI) - TÌM HIỂU VỀ SỰ NHẬN DẠNG THỜI GIAN THỰC

Run-time type information might be new to you because it is not found in non-polymorphic languages such as C. In non-polymorhic languages, there is no need for run-time type information because the type of each objects is known at compile time (i.e. When the program is written). However, in polymorphic languages such as C++, there can be situations in which the type of an object is unknown at compile time because the precise nature of that object is not determined until the program is executed. As you know, C++ implements polymorphism through the use of class hierarchies, virtual functions, and base class pointers. In this approach, a base class pointer can be used to point to objects of the base class or to any object derived from that base. Thus, it is not always possible to know in advance what type of object will be pointed to by a base pointer at any given moment in time. This determination must be made at run time, using run-time type identification.

Thông tin thời gian thực là điều mới mẻ với bạn vì nó không có trong ngôn ngữ đơn giản như C. Trong ngôn ngữ không hướng đối tượng, thông tin thời gian thực là không cần thiết vì những đối tượng chạy theo thời gian của trình biên dịch (ví dụ khi chương trình được tạo ra). Tuy nhiên trong ngôn ngữ đa hình như C++, có trường hợp kiểu của đối tượng không được trình biên dịch nhận ra vì sự chính xác tự nhiên của nó chỉ được biết đến khi chương trình được thực thi. Như bạn biết, C++ thể hiện tính đa hình thông qua sự phân cấp lớp, những chức năng thực tế, và những lớp con trỏ cơ sở. Trong cách tiếp cận này một lớp con trỏ cơ sở có thể trỏ tới đối tượng của lớp cơ sở hay bất kì lớp nào kế thừa từ lớp cơ sở. Như vậy luôn có trường hợp có thể không biết được kiểu đối tượng sẽ được trỏ tới bởi lớp con trỏ cơ sở vào khoảnh khắc đưa ra, điều này có thể làm được trong khi chạy bằng cách dùng định nghĩa loại thời gian thực.

*To obtain an object's type, use **typeid**. You must include the header <**typeinfo**> in order to use **typeid**. The most common form of **typeid** is shown here:*

Để định nghĩa một đối tượng sử dụng **typeid**. Bạn phải dùng đầu mục

<**typeinfo**> để sử dụng **typeid**. Sự định nghĩa của hình thức **typeid** được dùng như sau:

```
typeid(object)
```

Here object is the object whose type you will be obtaining. ***typeid*** *returns a reference to an object of type* ***type_info*** *that describes the type of object defined by object. The* ***type_info*** *class defines the following public members:*

Ở đây *object* là một đối tượng bạn tạo ra. **typeid** trả về dạng của đối tượng mà **type_info** đã mô tả khi định nghĩa loại của đối tượng *object*. Lớp **type_info** định nghĩa hàm thành viên

```
bool operator==(const type_info &ob);
bool operator!=(const type_info &ob);
bool before(const type_info &ob);
const char*name();
```

The overloaded == and != provide for the comparison of types. The ***before()*** *function returns true if the invoking object is before the object used as a parameter in collation order. This function is mostly for internal use only. Its return value has nothing to do with inheritance or class hierarchies). The* ***name()*** *function returns a pointer to the name of the type.*

Toán tử chồng == và != cung cấp sự so sánh các kiểu. Hàm **before()** trả về giá trị đúng nếu đối tượng khai báo trước đó được dùng như một đối số để tạo đối tượng. Hàm này được sử dụng rộng rãi, nó trả về giá trị của lớp kế thừa hay lớp phân cấp. Hàm **name()** trả về con trỏ giữ tên của kiểu

While ***typeid*** *will obtain the type of any object, its most important use is its application through a pointer of a polymorphic base class. In this case, it will automatically return the type of the actual object being pointed to, which can be a base class object or an object derived from that base. (Remember, a base class pointer can point to an object of the base class or of any class derived from that base). Thus, using* ***typeid*** *you can determine at run time the type of the object that is*

*being pointed to by a base class pointer. The same applies to references. When **typeid** is applied to a reference to an object of a polymorphic class, it will return the type of the object actually being referred to, which can be of a derived type. When **typeid** is applied to a non-polymorphic class, the base type of the pointer or reference is obtained.*

Trong khi **typeid** định nghĩa kiểu của đối tượng, cách dùng quan trọng nhất của nó được ứng dụng thông qua con trỏ đa hình của lớp cơ sở. Trong trường hợp này, nó tự động trả về giá trị thực của đối tượng mà con trỏ trỏ tới. (Nhớ rằng một lớp con trỏ cơ sở có thể trỏ vào một đối tượng của lớp cơ sở hay bất kì lớp nào được tạo ra từ lớp cơ sở). Điều này có nghĩa sử dụng **typeid** bạn có thể xác định thời gian thi hành của đối tượng đang được trỏ tới bởi lớp con trỏ cơ sở. Áp dụng tương tự cho lớp dẫn xuất. Khi **typeid** được áp dụng cho một đối tượng của lớp đa hình nó sẽ trả về kiểu của đối tượng thực sự được dẫn xuất, điều này dùng để khai báo kiểu. Khi **typeid** được dùng trong 1 lớp không có tính đa hình lớp con trỏ cơ sở hoặc dẫn xuất vẫn được sử dụng.

*There is a second form of **typeid**, one that takes a type name as its argument. This form is shown here:*

Có hai hình thức của **typeid**, một cách dùng tên kiểu như một quy tắc. Hình thức này như sau:

```
typeid(type-name)
```

*The main use of this form of **typeid** is to obtain a **type_info** object that describes the specified type so that it can be used in a type comparison statement.*

Hình thức chính của **typeid** được xuất phát từ một đối tượng **type_info** sao cho mô tả này có thể dùng như một sự phát biểu khi so sánh kiểu.

*Because **typeid** is commonly applied to a dereferenced pointer (i.e., one to which the *operator has been applied), a special exception has been created to handle the situation in which the pointer being dereferenced is null. In this case, **typeid** throws a **bad_ typeid** exception.*

Thông thường **typeid** được ứng dụng vào con trỏ toán tử (ví dụ nó được áp

dụng trong toán tử *operator), một ngoại lệ đặc biệt được tạo ra và nắm giữ trong hoàn cảnh tương tự thì một con trỏ chỉ tới null. Trong trường hợp này thì **typeid** chỉ tới một trường hợp xấu **bad_typeid.**

Run-time type identification is not something that every program will use. However, when you are working with polymorphic types, it allows you to know what type of object is being operated upon in any given situation.

Kiểu nhận dạng thời gian thực không phải được sử dụng trong mọi chương trình. Tuy nhiên khi chúng ta làm việc với các kiểu đa hình, nó cho phép chúng ta biết được kiểu của đối tượng được xử lý trong những hòan cảnh tương tự.

EXAMPLES - VÍ DỤ

1. *The following program demonstrates **typeid**. It obtains type information about one of C++'s built-in types, **int**. It then displays the types of objects pointed to by **p**, which is a pointer of type **BaseClass**.*

Chương trình sau biểu diễn kiểu **typeid**. Nó dùng kiểu thông tin về một trong những kiểu xây dựng sẵn của C++, **int.** Nó hiển thị kỉeu của đối tượng được trỏ tới bởi **p**, con trỏ của lớp cơ sở **BaseClass**

```
// An example that uses typeid
#include <iostream.h>
#include <typeinfo>
using namespace std;

class BaseClass
{
     virtual void f()
     {
```

```
    };   // make BaseClass polymorphic
    // ...
};

class Derived1:public BaseClass
{
        //....
};

class Derived2:public BaseClass
{
         //....
};

int main()
{
    int i;
    BaseClass *p, baseob;
    Derived1 ob1;
    Derived2 ob2;

    // First display type name of a built-in type.
    cout<<"Typeid of i is: ";
    cout<<typeid(i).name()<<endl;
```

```
    // Demonstrate typeid with polymorphic types
    p = &baseob;
    cout<<"p is pointing to an object of type ";
    cout<<typeid(p).name()<<endl;

    p = &ob1;
    cout<<"p is poiting to an object of type: ";
    cout<<typeid(p).name()<<endl;

    p = &ob2;
    cout<<"p is pointing to an object of type: ";
    cout<<typeid(p).name()<<endl;

    return 0;
}
```

The output produced by this program is shown here.

Kết quả xử lý của chương trình như sau:

```
Typeid of i is int
p is pointing to an object of type class Baseclass
p is pointing to an object of type class Derived1
p is pointing to an object of type class Derived2
```

*As explained, when **typeid** is applied to a base class pointer of a polymorphic type, the type of object pointed to will be determined at run time, as the output produced by the program shows. As an experiment, comment out the virtual function **f()** in **BaseClass** and observe the results.*

Giải thích, khi **typeid** được áp dụng cho lớp con trỏ cơ sở của kiểu đa hình, kiểu của đối tượng được trỏ tới sẽ được định rõ trong khi chạy, là kết quả xuất ra của chương trình. Như trong ví dụ, ghi chú về hàm ảo **f()** trong **BaseClass** quan sát từ kết qủa.

*As explained, when **typeid** is applied to a reference to a polymorphic base class, the type returned is that of the actual object being referred to. The circumstance in which you will most often make use of this feature is when objects are passed to functions by reference. For example, in the following program, the function **WhatType()** declares a reference parameter to objects of type **BaseClass**. This means that **WhatType()** can be passed references to objects of type **BaseClass** or any class derived from **BaseClass**. When the **typeid** operator is applied to this parameter, it returns the actual type of the object being passed.*

Như đã biết, khi **typeid** tham chiếu tới một lớp có tính đa hình, kiểu trả về thật sự của đối tượng sẽ được quy định sẵn. Trong trường hợp này bạn sẽ sử dụng đặc tính này thường xuyên khi một đối tượng tham chiếu tới một hàm. Ví dụ, trong chương trình sau, hàm **WhatType()** sẽ tham chiếu tới đối số của đối tượng thuộc lớp **BaseClass**. Điều này có nghĩa **WhatType()** có thể truy xuất tới đối số của đối tượng thuộc lớp **BaseClass** hoặc bất kì lớp dẫn xuất nào từ **BaseClass**. Khi toán tử **typeid** áp dụng cho hàm này, nó trả về kiểu thực sự của đối tượng được gọi.

```
// Use a reference with typeid

#include <iostream.h>

#include <typeinfo>

using namespace std;


class BaseClass

{

     virtual void f()

     {};  // make BaseClass polymorphic

     // ...

};
```

```
class Derived1:public BaseClass
{
     //....
};

class Derived2:public BaseClass
{
     //....
};

// Demonstrate typeid with a reference parameter
void WhatType(BaseClass &ob)
{
     cout<<"ob is referencing an object of type: ";
     cout<<typeid(ob).name()<<endl;
}
int main()
{
     int i;
     BaseClass baseob;
     Derived1 ob1;
     Derived2 ob2;

     WhatType(baseob);
```

```
    WhatType(ob1);
    WhatType(ob2);

    return 0;
}
```

The output produced by this program is shown here.

```
ob is referencing an object of type class BaseClass
ob is referencing an object of type class Derived1
ob is referencing an object of type class Derived2
```

*Although obtaining the type name of an object is useful in some circumstances, often all you need to know is whether the type of one object matches that of another. Since the **typeid_info** object returned by **typeid** overloads the == and != operators, this too is easy to accomplish. The following program demonstrates the use of these opeator*

Mặc dù sử dụng kiểu tên đối tượng hữu dụng trong nhiều trường hợp, thông thường bạn cần hiểu cách mà kiểu của một đối tượng kết nối với những đối tượng khác. Từ khi đối tượng **typeid_info** trả về **typeid** bằng toán tử chồng == và !=, điều này thật dễ làm. Chương trình sau mô tả cách dùng của những toán tử đó.

```
// Use a reference with typeid
#include <iostream>
#include <typeinfo>
using namespace std;

class x
{
    virtual void f()
```

```
    {
    }
};

class y
{
    virtual void f()
    {
    }
};

int main()
{
    x x1,x2;
    y y1;

    if(typeid(x1) == typeid(x2))
        cout<<" x1 and x2 are same types\n ";
    else
        cout<<" x1 and x2 are different types\n ";

    if(typeid(x1) != typeid(y1))
        cout<<" x1 and y1 are different types\n ";
    else
        cout<<" x1 and y1 are same types\n ";
```

```
}
```

The program displays the following output:

Kết quả của chương trình hiển thị:

```
x1 and x3 are same types

x1 and y1 are different types
```

*Althouth the preceding examples demonstrate the mechanics of using **typeid**, they don't show its full potential because the types in the preceding programs are knowable at compile time. In the following program, this is not the case. The program defines a simple class hierarchy that draws shapes on the screen. At the top of the hierarchy is the abstract class **Shape**. Four concreate sunclasses are created: **Line, Square, Rectangle** and **NullShape**. The function **generator()** generates an object and returns a pointer to it. The actual object created is determined randomly based upon the outcome of the random number generator **rand()**. (A function that produces objects is sometimes called an object factory**)**. Inside **main()**, the shape of each object is displayed for all objects but **NullShape** objects, which have no shape. Since objects are generated randomly, there is no way to know in advance what type of object will be created next. Thus, the use of RTTI is required.*

Mặc dù ví dụ trước đã giải thích về cơ bản cách dùng của **typeid**, nhưng chúng không chỉ ra khả năng đầy đủ bởi kiểu trả về trong chương trình có thể được biết trước khi biên dịch. Chương trình sau thì không như thế. Chương trình sẽ định nghĩa sự phân cấp lớp đơn giản và cố gắng thể hiện trên màn hình. Ở đầu của sự phân cấp là lớp cơ bản **Shape.** Bốn lớp được tạo ra: **Line, Square, Rectangle** và **NullShape.** Hàm **generator()** được tự động tạo ra một đối tượng và trả về con trỏ trỏ tới nó. Giá trị thực của đối tượng được tạo ngẫu nhiên bằng hàm **rand()**. (một hàm khởi tạo đối tượng đôi khi được gọi là xưởng tạo đối tượng). Trong **main(),** hình dạng của tất cả các đối tượng được hiển thị nhưng **NullShape** thì không. Khi một đối tượng được tao ngẫu nhiên, không thể xác định được kiểu của đối tượng tiếp theo được tạo ra là gì. Vì thế, hàm RTTI là cần thiết.

```
#include <iostream>
#include <cstdlib>
#include <typeinfo>
using namespace std;

class Shape
{
     public:
          virtual void example() = 0;
};

class Rectangle: public Shape
{
     public:
          void example()
          {
               cout<<"*****\n*     *\n* *\n*****\n";
          }
};

class Triangle: public Shape
{
     public:
          void example()
          {
```

```
                cout<<"*\n*          *\n* *\n*****\n";
            }
};

class Line: public Shape
{
     public:
          void example()
          {
               cout<<"*****\n";
          }
};

class NullShape: public Shape
{
     public:
          void example()
          {
          }

};

// A factory for objects derived from Shape.
Shape *generator()
{
     switch (rand() %4)
```

```
        {
            case 0:
                return new Line;
            case 1:
                return new Rectangle;
            case 2:
                return new Triangle;
            case 3:
                return new NullShape;
        }
        return NULL;
}

int main()
{
        int i;
        Shape *p;

        for(i=0; i<10; i++)
        {
            p = generator(); // get next object

            cout<<typeid(*p).name()<<endl;

            // draw object only if it is not a
        NullShape
```

```
            if(typeid(*p) != typeid(NullShape))
                  p->example();
      }

      return 0;
}
```

Sample output from the program is shown here.

Kết quả của chương trình:

```
Class Rectangle
*****
*      *
*      *
*****
Class NullShape
Class Triangle
*
* *
*     *
******
Class Line
*****
Class Rectangle
*****
*      *
```

```
*       *

*****

Class Line

*****

Class Triangle

*

* *

*     *

******
```

*The **typeid** operator can be applied to template classes. For example, consider the following program. It creates a hierarchy of template classes that store a value. The virtual function **get_val()** returns a value that is defined by each class. For class **Num**, this is the value of the number itself. For **Square**, it is the square of the number, and for **Sqr_root**, it is the square root of the number. Objects derived from **Num** are generated by the **generator()** function. The **typeid** operator is used to determine what type of object has been generated.*

Toán tử **typeid** có thể dùng cho lớp khuôn mẫu. Ví dụ, trong chương trình này, nó tạo lớp phân tầng để lưu dữ liệu. Hàm ảo **get_val()** trả về giá trị được xác định bởi từng lớp. Với lớp **Num**, hàm trả về giá trị của chính nó. Với **Square**, nó trả về số đó, và **Sqr_root** nó trả về căn bậc hai. Đối tượng dẫn xuất từ **Num** được tạo bởi hàm **generator()**. Toán tử **typeid** dùng để trả về kiểu mà đối tượng được tạo ra.

```
// typeid can be used with templates

#include <iostream>

#include <typeinfo>

#include <cmath>

#include <cstdlib>
```

```
using namespace std;

template<class T> class Num
{
     public:
          T x;
          Num(T i)
          {
               x = i;
          }
          virtual T get_val()
          {
               return x;
          };
};

template <class T>
class Square:public Num<T>
{
     public:
          Square (T i) : Num<T>(i) {}
          T get_val()
          {
               return x*x;
          }
```

```
};

template <class T>
class Sqr_root : public Num<T>
{
     public:
          Sqr_root (T i) : Num<T>(i) {}
          T get_val()
          {
               return sqrt((double)x);
          }
};

// A factory for objcets derived from Num
Num<double> *generator()
{
     switch (rand()%2)
     {
          case 0:
               return new Square<double> (rand()
     %100);
          case 1:
               return new Sqr_root<double>
     (rand()%100);
     }
     return NULL;
}
```

```
int main()
{
     Num<double> ob1(10), *p1;
     Square<double> ob2(100.0);
     Sqr_root<double> ob3(999.2);
     int i;

     cout<< typeid(ob1).name()<<endl;
     cout<< typeid(ob2).name()<<endl;
     cout<< typeid(ob3).name()<<endl;

     if(typeid(ob2) == typeid(Square<double>))
          cout<<"is Square<double>\n";

     p1 = &ob2;

     if(typeid(*p1) != typeid(ob1))
          cout<<"Value is: "<<p1->get_val();
     cout<<"\n\n";

     cout<<"Now, generator some objects.\n";
     for(i = 0;i < 10; i++)
     {
          p1 = generator(); // get next object
```

```
        if(typeid(*p1) == typeid(Square<double>))
            cout<<"Square object: ";
        if(typeid(*p1) ==
    typeid(Sqr_root<double>))
            cout<<"Sqr_root object: ";

        cout<<"Value is: "<<p1->get_val();
        cout<<endl;
    }

    return 0;
}
```

The output from the program is shown here.

Kết quả của chương trình được chỉ ra sau đây:

```
class Num<double>
class Squre<double>
class Sqr_root<double>
is Square<double>
value is: 10000

Now, generate some objects
Sqr_root object: Value is: 8.18535
Square object: Value is: 0
```

```
Sqr_root object: Value is: 4.89898

Square object: Value is: 3364

Square object: Value is: 4096

Sqr_root object: Value is: 6.7082

Sqr_root object: Value is: 5.19613

Sqr_root object: Value is: 9.53939

Sqr_root object: Value is: 6.48074

Sqr_root object: Value is: 6
```

EXERCISES - BÀI TẬP

1. Why is RTTI a necessary feature of C++?

Tại sao RTTI cần thiết trong C++?

2. Try the experiment described in Example 1. What output do you see?

Hãy làm bài trong ví dụ 1. Kết qủa bạn thấy là gì?

3. Is the following fragment correct?

Câu lệnh sau đúng hay sai?

```
cout<<typeid(float).name();
```

*4. Given this fragment, show how to determine whether **p** is pointing to a **D2** object.*

Trong đoạn sau, chỉ ra làm thế nào **p** trỏ tới đối tượng **D2**

```
class B
{
     virtual void f() {}
```

```
};

class D1: public B
{
     void f() {}
};

class D2: public B
{
     void f() {}
};

int main()
{
     B *p;
}
```

*5. Assuming the **Num** class from Example 5, is the following expression true or false?*

Với lớp **Num** trong ví dụ 5, biểu thức sau đúng hay sai?

```
typeid(Num<int>) == typeid(Num<double>)
```

6. On your own, exeriment with RTTI. Although its use might seem a bit esoteric in the context of simple, sample programs, it is a powerful construct that allows you to manage objects at run time.

Với sự hiểu biết của bạn, với RTTI. Mặc dù dường như nó là một nội dung

đơn giản, chương trình đơn giản, nó thật mạnh mẽ khi cho phép bạn quản lý đối tượng khi chạy.

1.2. USING DYNAMIC_CAST – SỬ DỤNG DYNAMIC_CAST

*Although C++ still fully supports the traditional casting operator defined by C, C++ adds four new ones. They are **dynamic_cast,const_cast, reiterpret_cast,** and **static_cast**. We will examine **dynamic_cast** first because it relates to RTTI. The other casting operators are examined in the following section.*

Mặc dù C++ vẫn hỗ trợ đầy đủ toán tử khuôn trong C, nó còn thêm 4 cái mới. Đó là **dynamic_cast,const_cast, reiterpret_cast,** và **static_cast**. Chúng ta sẽ tìm hiểu về **dynamic_cast** trước vì nó liên quan tới RTTI. Trong những chương sau ta sẽ học về các toán tử khuôn còn lại.

*The **dynamic_cast** operator performs a run-time cast that verifies the validety of a cast. If, at the time **dynamic_cast** is executed, the cast is invalid, the cast fails. The general form of **dynamic_cast** is shown here:*

Toán tử **dynamic_cast** biểu diễn khuôn mẫu thời gian thực được xác định bởi tính khuôn mẫu. Nếu, trong khi **dynamic_cast** được gọi, nếu không có khuôn mẫu thì lời gọi nó thất bại. Dạng chung của **dynamic_cast** được chỉ ra dưới đây:

```
dynamic_cast<target-type>(expr)
```

*Here target-type specifies the target type of the cast and expr is the expression being cast into the new type. The target type must be a pointer or reference. Thus, **dynamic_cast** can be used to cast one type of pointer into another or one type of reference into another.*

Ở đây *target_type* chỉ ra kiểu đích của khuôn mẫu và *expr* là biểu thức mẫu cho kiểu mới. Kiểu đích phải được trỏ tới hay được tham chiếu. Vì thế, **dynamic_cast** có thể

dùng để làm kiểu khuôn của con trỏ để trỏ tới đối tượng khác hay một kiểu tham chiếu tới đối tượng khác.

*The purpose of **dynamic_cast is** is to perform casts on polymorphic types. For example, given the two polymorphic classes **B** and **D**, with **D** derived from **B**, a **dynamic_cast** can always cast a **D*** pointer into a **B*** pointer. This is because a base pointer can always point to a derived object. But a **dynamic_cast** can cast a **B*** pointer into an **D*** pointer only if the object being pointed to actually is a **D** object. In general, **dynamic_cast** will succeed if the pointter (or reference) being cast is a pointer (or reference) to either an object of the target type or an object derived from the target type. Otherwise, the cast will fail. If the cast fails, **dynamic_cast** evaluates to null if the cast involves pointers. If a **dynamic_cast** on reference types fails, a **bad_castN** exception is thrown.*

Mục đích của **dynamic_cast** là làm khuôn mẫu cho kiểu đa hình. Ví dụ, cho hai lớp đa hình **B** và **D**, D kế thừa từ B, **dynamic_cast** luôn làm khuôn cho con trỏ **D*** sang con trỏ **B*.** Lý do vì con trỏ lớp cơ sở có thể trỏ tới con trỏ lớp dẫn xuất. Nhưng **dynamic_cast** có thể chuyển khuôn từ **B*** sang **D*** chỉ khi nào đối tượng đang được trỏ đến thực sự là đối tượng **D**. Thông thường, **dynamic_cast** sẽ thành công nếu con trỏ (hay tham chiếu) đang làm khuôn cho con trỏ (hay tham chiếu) tới một đối tượng là kiểu đích or đối tượng kế thừa từ đối tượng có kiểu đích. Ngòai ra ép khuôn thất bại. Nếu thất bại, **dynamic_cast** trả về null nếu khuôn đang là con trỏ. Nếu **dynamic_cast** tham chiếu thất bại, một hàm **bad_cast** sẽ được gọi.

*Here is a simple example, assume that B**ase** is a polymorphic class and that **Derived** is derived from **Base**.*

Trong ví dụ sau, **Base** là lớp đa hình và **Derived** là lớp kế thừa của nó.

```
Base *bp, b_ob;

Derived *dp, d_ob;


bp = & d_ob; // base pointer points to Derived object

dp = dynamic_cast<Derived*> (dp);

if(dp) cout<<"Cast Ok";
```

*Here the cast from the base pointer **bp** to the derived pointer **dp** works because **bp** is actually pointing to a **Derived** object. Thus, this fragment displays **Cast OK.** But in the next fragment, the cast fails because **bp** is pointing to a **Base** object and it is illegal to cast a base object into a derived object.*

Ở trên ép khuôn từ con trỏ cơ sở **bp** sang con trỏ kế thừa **dp** làm được vì **bp** thực ra đang trỏ tới lớp kế thừa **Derived.** Vì thế kết quả hiển thị là **Cast Ok.** Nhưng trường hợp kế tiếp, ép khuôn thất bại vì **bp** đang trỏ đến đối tượng lớp cơ sở **Base** và nó chỉ được chấp nhận ép khuôn từ đối tượng cơ sở sang đối tượng kế thừa.

```
bp = &b_ob;// base pointer points to Base object

dp = dynamic_cast<Derived*> (bp);

if(!dp) cout<< "Cast Fails";
```

*Because the cast fails, this fragment displays **Cast Fails**. The **dynamic_cast** operator can sometimes be used instead of **typeid** in certain cases. For example, again assume that **Base** is a polymorphic base class for **Derived**. The following fragment will assign **dp** the address of the object pointed to bye **bp** if and only if the object is really a **Derived** object.*

Bởi vì ép khuôn thất bại, kết quả hiện **Cast Fails.** Toán tử **dynamic_cast** thỉnh thoảng có thể dùng để thay cho **typeid** trong vài trường hợp. Ví dụ, thừa nhận **Base** là lớp đa hình cơ sở của lớp **Derived.** Đoạn chương trình sau sẽ gán cho **dp** địa chỉ của đối tượng được trỏ đến bởi **bp** nếu và chỉ nếu đối tượng thực sự là đối tượng của lớp **Derived**

```
Base *bp;

Derived *dp;

//...

if(typeid(*bp) == typeid(Derived)) dp = (Derived*) bp;
```

*In this case a C-style cast is used to actually perform the cast. This is safe because the **if** statement checks the legality of the cast using **typeid** before the cast actually occurs. However, a better way to accomplish this is to replace the **typeid** operators and **if** statement with this **dynamic_cast:***

Trong trường hợp này một cách ép khuôn phong cách C được dùng để ép khuôn. Điều này an toàn vì câu lệnh **if** sẽ kiểm tra tính hợp pháp của việc ép khuôn dùng **typeid** trước khi việc ép khuôn được thực thi. Tuy nhiên, cách tốt hơn để hoàn thành điều

này là thay thế toán tử **typeid** và câu lệnh **if** bằng **dynamic_cast.**

```
dp = dynamic_cast<Derived *> (bp);
```

*Because **dynamic_cast** succeeds only if the object being cast is either already an object of the target type or an object derived from the target type, after this statement executes **dp** will contain either a null or a pointer to an object of type **Derived**. Since **dynamic_cast** succeeds only if the cast is legal, it can simplify the logic in certain situations.*

Bởi vì **dynamic_cast** chỉ đúng nếu đối tượng ép khuôn là đối tượng kiểu đích hoặc đối tượng kế thừa từ kiểu đích, sau khi thực thi câu lệnh **dp** sẽ chứa một trong hai giá trị null hoặc trỏ tới đối tượng kiểu **Derived**. Khi **dynamic_cast** đúng chỉ khi ép khuôn hợp quy tắc, nó rất đơn giản về logic

EXAMPLES - VÍ DỤ

*The following program demonstrates **dynamic_cast:***

Chương trình này giải thích về **dynamic_cast:**

```
// Demonstrate dynamic_cast

#include <iostream>

using namespace std;


class Base

{

     public:

          virtual void f()

          {

               cout<<"Inside Base \n";

          }

          //....

};
```

```
class Derived: public Base
{
     public:
          void f()
          {
               cout<<"Inside Derived\n";
          }
};

int main()
{
    Base *bp, b_ob;
    Derived *dp, d_ob;

    dp = dynamic_cast<Derived*> (&d_ob);
    if(dp)
    {
         cout<<"Cast from Derived * to Derived * Ok
         \n";
         dp->f();
    }
    else
         cout<<" Error\n";

    cout<<endl;
```

```
bp = dynamic_cast<Base*> (&d_ob);
if(bp)
{
     cout<<"Cast from Derived * to Derived * Ok
\n";
     bp->f();
}

cout<<endl;

bp = dynamic_cast<Base*> (&b_ob);
if(bp)
{
     cout<<"Cast from Derived* to Derived*
Ok\n”;         bp->f();
}
else
     cout<<"Error\n";

cout<<endl;

dp = dynamic_cast<Derived*> (&b_ob);
if(dp)
     cout<<"Error\n";
else
```

```
     cout<<"Cast from Base * to Derived * not
     Ok\n";

cout<<endl;

bp = &d_ob;    // bp points to Derived object
dp = dynamic_cast<Derived*> (bp);
if(dp)
{
     cout<<"Casting bp to a Derived *Ok\n:";
     cout<<"because bp is really pointing\n";
     cout<<"to a Derived object\n";
     dp->f();
}
else
     cout<<"Error\n";

cout<<endl;

bp = &b_ob;//bp points to Base object
dp = dynamic_cast<Derived*>(bp);
if(dp)
     cout<<"Error\n";
else
{
```

```
        cout<<"Now casting bp to a Derived *\n";
        cout<<"is not Ok because bp is really\n";
        cout<<"pointing to a Base object\n";

    }
    cout<<endl;
    dp = &d_ob; // dp points to Derived object
    bp = dynamic_cast<Base*> (dp);
    if(dp)
    {
        cout<<"Casting dp to a Base * is Ok\n";
        bp->f();
    }
    else
        cout<<"Error\n";

    return 0;
}
```

The program produces the following output.

Kết quả chạy chương trình:

```
Cast from Derived *to Derived * Ok
Inside Derived

Cast from Base * to Base * Ok
Inside Base
```

```
Cast from Base * to Derived * not Ok


Casting bp toa Derived *Ok

because bp is really pointing

to a Derived object

Inside Derived


Now casting bp to a Derived *

is not Ok because bp is really pointing to a Base
object;


Casting dp to a Base * is Ok

Inside Derived.
```

*The following example illustrates how a **dynamic_cast** can be used to replace **typeid***

Ví dụ sau chỉ ra cách mà **dynamic_cast** có thể bị thay thế bởi **typeid**

```
// Use dynamic_cast to replace typeid

#include <iostream>

#include <typeinfo>

using namespace std;


class Base

{
```

```
    public:
         virtual void f() {}
};

class Derived :public Base
{
    public:
         void derivedOnly()
         {
              cout<<"Is a Derived Object \n";
         }
};

int main()
{
    Base *bp, b_ob;
    Derived *dp, d_ob;

    //**********************
    // Use typeid
    //**********************
    bp = &b_ob;
    if(typeid(*bp) == typeid(Derived))
    {
         dp = (Derived *) bp;
```

```
     dp->derivedOnly();
}
else
     cout<<"Cast from Base to Derived failed
\n";
bp = &d_ob;
if(typeid(*bp) == typeid(Derived))
{
     dp = (Derived *) bp;
     dp->derivedOnly();
}
else
     cout<<"Error, cast should work! \n";
//***********************
// Use dynamic_cast
//***********************
bp = &b_ob;
dp = dynamic_cast<Derived*> (bp);
if(dp)
     dp->derivedOnly();
else
     cout<<"Cast from Base to Derived failed\n";

bp = &d_ob;
dp = dynamic_cast<Derived*> (bp);
if(dp)
```

```
            dp->derivedOnly();
        else
            cout<<"Error, cast should work!\n";


        return 0;
    }
```

*As you can see, the use of **dynamic_cast** simplifies the logic required to cast a base pointer into a derived pointer. The output from the program is shown here.*

Như bạn thấy, dùng **dynamic_cast** thì đơn giản về mặt logic hơn để chuyển khuôn mẫu từ con trỏ cơ sở sang con trỏ dẫn xuất. Kết quả chạy chương trình được chỉ ra dưới đây:

```
Cast from Base to Derived failed.
Is a Derived Object.
Cast from Base to Derived failed.
Is a Derived Object.
```

*The **dynamic_cast** operator can also be used with template classes. For example, the following program reworks the template class from Example 5 in the preceding section so that it uses **dynamic_cast** to determine the type of object returned by the **generator()** function.*

Toán tử **dynamic_cast** có thể dùng với lớp chung. Ví dụ, chương trình sau làm việc với lớp chung từ ví dụ 5 trong phần xử lý mà dùng **dynamic_cast** để chuyển kiểu của đối tượng trả về bởi hàm **generator()**.

EXERCISES - BÀI TẬP

*1. In your own words, explain the purpose of **dynamic_cast.***

Với hiểu biết của mình, hãy chỉ ra mục đích dùng **dynamic_cast.**

*2. Given the following fragment and using **dynamic_cast**, show how you can assign **p** a pointer to some object **ob** if and only if **ob** is a **D2** object.*

Có đoạn chương trình sau dùng **dynamic_cast**, chỉ cho bạn cách chuyển con trỏ **p** thành đối tượng **ob** nếu và chỉ nếu **ob** là đơi tương **D2**

```
class B
{
     Virtual void f() {}
};
class D1:public B
{
     void f() {}
};

class D2:public B
{
     void f() {}
};
B *p;
```

*3. Convert the **main()** function in Section 12.1, Exercise 4, so that it uses **dynamic_cast** rather than **typeid** to prevent a **NullShape** object from being displayed.*

Chuyển hàm **main()** trong phần 12.1 bài tập 4, sử dụng **dynamic_cast** thay thế cho **typeid** để không cho đối tượng **NullShape** hiển thị

*Using the **Num** class hierarchy from Example 3 in this section , will the following work?*

Dùng lớp **Num** trong bài tập 3 của phần này, chương trình sau có hoạt động không?

```
Num<int> *Bp;

Square<double> *Dp;

//... Dp = dynamic_cast <Num<int>> (Bp);
```

1.3. USING CONST_CAST, REINTERPRET_CAST, AND STATIC_CAST - CÁCH DÙNG CONST_CAST, REINTEPRET_CAST VÀ STATIC_CAST

Although dynamic_cast is the most important of the new casting operators, the other three also valuable to the programer. Their general forms are shown here:

Mặc dù toán tử **dynamic_cast** là quan trọng nhất của toán tử ép khuôn, những toán tử khác cũng rất hữu ích cho lập trình viên. Dạng chung của chúng được trình bày như sau:

```
const_cast<target-type>(expr)

reinterpret_cast<target-type>(expr)

static_cast<target-type>(expr)
```

Here target-type specifies the target type of the cast and expr is the expression being cast into the new type. In general, these casting operators provide a safer, more explicit means of performing certain type conversions than that provided by the C-style cast.

Ở đây *target-type* chỉ định kiểu đích của ép khuôn và *expr* biểu thị khuôn mẫu sang kiểu mới. Thông thường, ba toán tử còn lại đưa ra cách an toàn, rõ ràng hơn về cách chuyển đổi khuông mẫu được cung cấp trong phong cách lập trình C

*The **const_cast** operator is used to explicitly override **const** and or **volatile** in a cast. The target type must be the same as the source type except for the alteration of its **const** or **volatile** attributes. The most common use of **const_cast** is to remove **const**-ness.*

Toán tử **const_cast** dùng ghi chồng **const** hoặc **volatile** trong khuôn mẫu. Kiểu đích phải giống như kiểu nguồn cho sự thay đổi thuộc tính từ **const** hoặc **volatile.** Cách dùng của **const_cast** là di chuyển **const**-ness

*The **static_cast** operator performs a non-polymorphic cast. For example, it can be used to cast a base class pointer into a derived class pointer. It can also be used for any standard conversion. No run-time checks are performed.*

Toán tử **static_cast** dùng cho khuôn mẫu không đa hình. Ví dụ, nó dùng ép khuôn từ lớp con trỏ cơ sở sang con trỏ lớp kế thừa. Nó cũng dùng làm dạng chuyển đổi chuẩn. Không có sự kiểm tra thời gian nào được thực hiện.

*The **reinterpret_cast** operator changes one pointer type into another, fundamentally different, pointer type. It can also change a pointer into an integer and an integer into a pointer. A **reinterpret_cast** should be used for casting inherently incompatible pointer types.*

Toán tử **reinterpret_cast** chuyển một kiểu con trỏ sang kiểu khác, kiểu khác cơ bản, kiểu con trỏ. Nó cũng chuyển con trỏ sang số nguyên và ngược lại. Có thể dùng ép khuôn kế thừa kiểu con trỏ.

*Only **const_cast** can cast away **const-**ness. That is, neither **dynamic_cast, static_cast,** nor **reinterpret_cast** can alter the **const**-ness of an object.*

Chỉ có **const_cast** có thể ép khuôn **const**-ness. Điều này có nghĩa không cái nào trong **dynamic_cast, static_cast,** hoặc **reinterpret_cast** có thể làm được.

EXAMPLES - VÍ DỤ

*The following program demonstrates the use of **reinterpret_cast.***

Chương trình mô tả cách dùng **reinterpret_cast.**

```
// An example that uses reinterpret_cast
#include <iostream>
using namespace std;

int main()
{
     int i;
     char *p = "This is a string";

     i = reinterpret_cast<int> (p); // cast pointer to
     integer

     cout<<i;

     return 0;
}
```

*Here **reinterpret_cast** converts the pointer **p** into an integer. This conversion represents a fundamental type change and is a good use of **reinterpret_cast.***

Ở đây **reinterpret_cast** chuyển từ con trỏ **p** thành số nguyên. Sự chuyển đổi này thay đổi kiểu và là cách dùng hay của **reinterpret_cast**

*The following program demonstrates **const_cast**.*

Chương trình mô tả **const_cast.**

```
// Demonstrate const_cast
#include <iostream>
```

```
using namespace std;

void f(const int*p)
{
   int *v;

   // cast away const-ness
   v = const_cast<int*>(p);

   *v=100; // now, modify object through v
}

int main()
{
   int x = 99;

   cout<<"x before call: "<<x<<endl;
   f(&x);
   cout<<"x after call: "<<x<<endl;

   return 0;
}
```

The output produced by this program is shown here.

Kết quả xử lý chương trình:

```
x befoer call: 99
```

```
x after call: 100
```

*As you can see, **x** was modified by **f()** even though the parameter to **f()** was spwecified as a **const** pointer.*

Như bạn thấy, x bị thay đổi bởi **f()** dù cho đối số của **f()** là con trỏ hằng

*It must be be stressed that the use of **const_cast** to cast way **const-**nest is a pontentially dangerous feature. Use it with care.*

Có thể sai khi dùng **const_cast** để ép khuôn. Hãy cẩn thận khi dùng nó.

*The **static_cast** operator is essentially a substitute for the original cast operator. It simply performs a non-polymorphic cast. For example, the following casts a **float** into an **int.***

Toán tử **static_cast** về cơ bản là thay cho toán tử ép khuôn gốc. Nó dễ thực hiện ép khuôn không đa hình. Ví dụ sau chuyển từ **float** sang **int**

```
// Use static_cast

#include <iostream>

using namespace std;


int main()

{

     int i;

     float f;


     f = 199.10;


     i = static_cast<int> (f);


     cout<<i;
```

```
    return 0;
}
```

EXERCISES - BÀI TẬP

1. *Explain the rationale for* ***const_cast,reinterpret_cast,*** *and* ***static_cast.***

Chỉ ra nguyên nhân dùng **const_cast,reinterpret_cast,** và **static_cast.**

2. *The following program contains an error. Show how to fix it using a* ***const_cast.***

Chương trình sau có lỗi. Chỉ lỗi và sửa nó dùng **const_cast**

```
#include<iostream>
using namespace std;
void f (const double &i)
{
     i = 100; // Error -> fix using const_cast
}

int main()
{
     double x = 98.6;

     cout<<x<<endl;
     f(x);
     cout<<x<<endl;
```

```
    return 0;
}
```

*3. Explain why **const_cast** should normally be resserved for special cases.*

Giải thích về **const_cast** từ trường hợp đơn giản đến phức tạp

SKILLS CHECK (KIỂM TRA KĨ NĂNG)

Mastery skills check

At this point you should be able to perform the following exercises and answer the questions.

Đến lúc này bạn có thể thực hành và trả lời các câu hỏi sau:

*1. Describe the operation of **typeid.***

Mô tả toán tử **typeid**

*2. What header must you include in order to use **typeid?***

Điều chủ đạo cần nhớ khi dùng **typeid**

3. In addition to the standard cast, C++ defines four casting operators. What are they and what are they for?

Trong phép cộng thông thường, C++ định nghĩa bốn toán tử chuyển kiểu. Chúng là toán tử nào và dùng làm gì?

4. Complete the following partial program so that it reports which type of object has been selected by the user.

Hoàn chỉnh chương trình sau, cho biết kiểu đối tượng được chọn bởi người dùng

```
#include <iostream>

#include <typeinfo>
```

```
using namespace std;

class A
{
    virtual void f() {}
};

class B:public A
{
};

class C: public A
{
};

int main()
{
    A *p, a_ob;
    B b_ob;
    C c_ob;
    int i;

    cout<<"Enter 0 for A objects: ";
    cout<<"1 for B objects: ";
    cout<<"2 for C objects: ";
```

```
    cin>>i;


    if(i==1)
        p=&b_ob;
    else
        if(i==2)
            p = &c_ob;
        else
            p=&a_ob;
    // report type of object selected by user
    return 0;
}
```

*5. Explain how **dynamic_cast** can somtimes be an alternative to **typeid.***

Khi nào **dynamic_cast** có thể thay cho **typeid**

*What type of object is obtained by the **typeid** operator?*

Kiểu giá trị trả về của toán tử **typeid** có thể là gì?

Cumulative Skills check

This section checks how well you have integrated material in this chapter with that from the preceding chapters.

Phần này kiểm tra bạn hiểu đến đâu về những điều trình bày trong chương này

*1. Rework the program in Section 12.1, Example 4 so that it uses exception handling to watch for an allocation failure within the **generator()** function.*

Làm lại chương trình Chương 12.1 ví dụ 4 tìm cách giữ giá trị để quan sát địa chỉ mà không dùng hàm **generator()**

2. *Change the **generator()** function from Question 1 so that it uses the **nothrow** version of **new.** Be sure to check for errors*

Thay hàm **generator()** trong câu 1 bằng cách dùng **nothrow** của **new**. Chắc chắn là không gây lỗi.

3. *Special Challenge: on your own, create a class hierarchy that has at its top an abstract class called **DataStruct**. Create two concrete subclasses. Have one implement a stack, the other a queue. Create a function called **DataStructFactory()** that has this prototype:*

Bài đặc biệt: bằng hiểu biết của mình, tạo lớp đầu tiên bằng cách gọi hàm **DataStruct**. Tạo 2 lớp không giá trị trả về. Có thể dùng tới stack hoặc hàng đợi. Tạo hàm **DataStructFactory()** có chuẩn sau:

```
DataStruct *DataStructFactory(char what):
```

*Have **DataStructFactory()** create a stack if what is s and a queue if what is q. Return a pointer to the object created. Demonstrate that your factory function works.*

Với **DataStructFactory()** tạo một stack nếu *what* là s và hàng đợi nếu là q. Trả về con trỏ của đối tượng tạo ra. Hãy làm cho xưởng tạo của bạn làm việc.

CHAPTER 13

NAMESPACES, CONVERSION FUNCTIONS, AND MISCELLANEOUS TOPICS - NAMESPACES, CÁC HÀM CHUYỂN ĐỔI VÀ CÁC CHỦ ĐỀ KHÁC NHAU

CHAPTER OBJECTIVES

This chapter discusses namespace, conversion functions, static and const class members, and other specialized features of C++.

Chương này thảo luận về namespace, các hàm chuyển đổi và các hằng và tĩnh thành viên của một lớp, và các đặc điểm đặc biệt của C++.

REVIEW SKILLS CHECK

ÔN TẬP KỸ NĂNG:

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Trước khi bắt đầu, bạn có thể sửa lại câu trả lời các câu hỏi sau và làm các bài tập.

What are the casting operators and what do they do?

1. Thế nào là toán tử sắp xếp và chúng hoạt động như thế nào?

*What is **type_info**?*

2. Thế nào là **type_info**

What operator determines the type of an object?

3. Toán tử nào xác định loại của một đối tượng?

*Given this fragment, show how to determine whether **p** points to an object of **Base** or an object of **Derived**.*

4. Cho các đoạn chương trình sau, làm thế nào xác định con trỏ **p** cho một đối tương

Base hoặc một đối tượng của **Derived**.

```
class Base
{
     virtual void f() {}
};

class Derived : public Base
{};

int main()
{
     Bae *p, b_ob;
     Derived d_ob;
     //…
}
```

*A **dynamic_cast** succeeds only if the object being cast is a pointer to either an object of the targettype or an object _____from the target type. (Fill in the blank)*

5. Một **dynaic_cast** thành công chỉ một đối tượng được sắp xếp là một con trỏ đối tượng của loại mục tiêu hoặc một đối tương _____ từ loại mục tiêu. (Điền

vào chỗ trống.)

*Can a **dynamic_cast** cast away **const**-ness?*

6. **dynamic_cast** có thể quăng **const**-ness không?

13.1. NAMESPACES

*Namespaces were briefly introduced in Chapter 1. Now it is time to look at them in detail. Namespace are a relatively recent addition to C++. Their purpose is to localize the names of identifiers to avoid name collisions. In the C++ programming environment, there has been an explosion of variable, function, and class names. Prior to the invention of namespaces, all of these names competed for slots in the global namespace, and many conflicts arose. For example, if your program defined a function called **toupper( )**, it could (depending upon its parameter list) override the standard library function **toupper( )** because both names would be stored in the global namespace. Name collisions were compounded when two or more third-party libraries were used by the same program. In this case, it was possible - even likely – that a name defined by one library would conflict with the same name defined by another library.*

Namespace được giới thiệu ngắn gọn trong chương 1. Bây giờ là lúc xem xét chúng một cách chi tiết. Namespace là phần thêm vào của C++. Mục đích của nó là xác định tên để tránh sự trùng tên. Trong môi trường chương trình C++, có sự bùng nổ khác nhau, hàm và tên lớp. Ưu tiên cho sự phát minh của namespace, tất cả các tên cạnh tranh cho chỗ trống trong toàn bộ namespace, và rất nhiều sự xung đột xuất hiện. Ví dụ, nếu chương trình của bạn xác định một hàm được gọi **toupper( )**, nó có thể (phụ thuộc danh sách tham số của nó) vượt qua hàm **toupper( )** thư viện chuẩn, bởi vì cả hai tên sẽ được chứa trong namespace. Sự va chạm tên được kết hợp khi hai hay nhiều thư viện thứ ba được sử dụng bởi cùng chương trình. Trong trường hợp này, nó có thể thậm chí, một cái tên được xác định bởi một thư viện có thể xung đột với cùng tên được xác định bởi thư viện khác.

*The creation of the **namespace** keyword was a response to these problems. Because it localizes the visibility of names declared within it, a namespace allows the same name to be used in different contexts without giving rise to conflicts. Perhaps the most noticeable beneficiary of namespace is the C++ standard library. In early versions of C++, the entire C++ library was defined within the global namespace (which was, of course, the only namespace). Now, however, the C++ library is defined within its own namespace, **std**, which reduces the chance of name collisions. You can also create your own namespaces within your program to localize the visibility of any names that you think might cause conflicts. This is especially important if you are creating class or function libraries.*

Sự tạo của từ khóa **namespace** là một trách nhiệm cho các vấn đề này. Bởi vì nó định vị các tên được khai báo, một namespace cho phép cùng tên được sử dụng trong các ngữ cảnh khác nhau mà không dẫn đến xung đột. Có thể các thông báo thừa kế của namespace là thư viện chuẩn C++. Trong phiên bản trước của C++, toàn bộ thư viện C++ được xác định bên trong toàn namespace (dĩ nhiên chỉ namespace). Bây giờ, tuy nhiên, thư viện C++ được xác định trong namespace, **std**, giảm cơ hội của sự xung đột tên. Bạn cũng có thể tạo một namespace trong chương trình của bạn để xác định các tên đã đặt mà bạn nghĩ có thể xảy ra xung đột. Đây là điều quan trọng đặc biệt nếu bạn tao một lớp hoặc một thư viện hàm.

*The **namespace** keyword allows you to partition the global namespace by creating a declarative region. In essence, a namespace defines a scope. The general form of **namespace** is shown here:*

Từ khóa **namespace** cho phép bạn chia toàn bộ namespace bằng cách tạo một vùng khai báo. Một namespace cần thiết xác định phạm vi hoạt động. Định dạng của **namespace** là:

```
namespace name
{
     // declarations
}
```

*Anything defined within a **namespace** statement is within the scope of that namespace.*

Here is an example of a namespace:

Bất cứ điều gì được xác định bên trong **namespace** là bên trong phạm vi hoạt động của namespace. Đây là ví dụ của namespace

```
namespace MyNameSpace
{
     int ix k;
     void myfunc(int j)
     {
          cout << j;
     }
     class myclass
```

```
    {
        public:
            void seti(int x)
            {
                i = x;
            }
            int geti()
            {
                return i;
            }
    };
}
```

*Here **i,k, myfunc( ),** and the class **myclass** are part of the scopedefined by the **MyNameSpace** namespace.*

i, k, myfunc( ), và lớp **myclass** là một phần phạm vi hoạt động được xác định bởi namespace **MyNameSpace.**

*Identifiers declared within a namespace can be referred to directly within that namespace. For example, in **MyNameSpace** the **return i** statement uses **i** directly. However, since **namespace** defines a scope, you need to use the scope resolution operator to refer to objects declared within a namespace from out side that namespace. For example, to assign the value 10 to **i** from code ouside **MyNameSpace**, you must use this statement:*

Những phần xác định được khai báo bên trong namespace có thể truy cập trực tiếp bên trong namespace. Ví dụ, trong **MyNameSpace** trả về **i**, sử dụng **i** trực tiếp. Tuy nhiên, từ khi **namespace** được xác định một phạm vi hoạt động, bạn cần sử dụng toán tử tầm vực để truy cập đến đối tượng được khai báo bên trong một namespace từ bên ngoài namespace. Ví dụ, để gán giá tri 10 chi **i** từ code bên ngoài **MyNameSpace**, bạn phải làm như sau:

```
MyNameSpace::i = 10;
```

*Or, to declare an object of type **myclass** from outside **MyNameSpace**, you use a statement like this:*

Hoặc khai báo một kiểu đối tượng **myclass** từ bên ngoài **MyNameSpace**, bạn sử dụng câu lệnh như sau:

```
MyNameSpace::myclass ob;
```

In general, to access a member of a namespace from outside its namespace, precede the member's name with the name of the namespace followed by the scope resolution operator.

Tóm lại, để truy cập một thành viên của namespace từ bên ngoài, các tên trước của thành viên với tên của namespace sau bởi toán tử tầm vực

*As you can imagine, if your program includes frequent references to the members of a namespace, the need to specify the namespace and the scope resolution operator each time you need to refer to one quickly becomes a tedious chore. The **using** statement was invented to alleviate this problem. The **using** statement has these two general forms:*

Như bạn đã đoán, nếu chương trình của bạn bao gồm các lệnh truy xuất thường xuyên đến các thành viên của một namespace, cần thiết chỉ rõ namespace và toán tử tầm vực mỗi lần bạn cần truy cập đến chúng một cách nhanh chóng trở thành một công việc buồn tẻ, chán ngắt. Câu lệnh **using** được lập ra để làm giảm nhẹ bớt vấn đề này. Câu lệnh **using** có hai cách sử dụng:

```
using namespace name;

using name::member;
```

*In the first form , name specifies the name of the namespace you want to access. When you use this form, all of the members defined within the specified namespace are brought into the current namespace and can be used without qualification. If you use the second from, only a specific member of the namespace is made visible. For example, assuming **MyNameSpace** as shown above, the following busing **statements** and assignments are valid:*

Trong cách thứ nhất, *name* chỉ rõ tên của namespace mà bạn cần truy cập. Khi bạn sử dụng cách này, tất cả các thành viên được định nghĩa bên trong namespace được chỉ rõ mang đến namespace hiện hành và có thể được sử dụng mà không cần điều kiện. Nếu bạn sử dụng cách thứ hai, chỉ có thành viên được chỉ định của namespace là thấy được. Ví dụ, **MyNameSpace** được chỉ ra bên trên, câu lệnh theo sau và sự quy cho là có giá trị.

```
using MyNameSpace::k;  // only k is made visible

k = 10;  // OK because k is visible


using namespace MyNameSpace; // all members are visible

i = 10;  // OK because all members of MyNameSpace are
visible
```

There can be more than one namespace declaration of the same name. This allows a namespace to be split over several files or even separated within the same file. Consider the following example:

Có thể có nhiều hơn một namespace được khai báo cùng tên. Điều này cho phép một namespace được chia thành nhiều tập tin hoặc thậm chí riêng rẽ bên trong cùng tập tin. Xem ví dụ sau:

```
namespace NS

{

     int i;

}


     …


namespace NS

{

     int j;

}
```

*Here **NS** is split into two pieces. However, the contents of each piece are still within the same namespace. **NS**.*

NS được chia làm hai. Tuy nhiên, nội dung trong mỗi phần vẫn còn bên trong cùng namespace. **NS**.

A namespace must be declared outside of all other scopes, with one exception: a namespace can be nested within another. This means that you cannot declare namespaces that are localized to a function, for example.

Một namespace phải được khai báo bên ngoài của tầm vực khác, với một ngoại lệ: một namespace có thể được chứa bên trong một cái khác. Điều này có nghĩa bạn không thể khai báo các namespace được định vị cho một hàm, ví du:

There is a special type of namespace called an unnamed namespace. that allows you to create identifiers that are unique within a file. It has this general form:

Có một kiểu đặc biệt được gọi là *unnamed namespace*. Nó cho phép bạn tạo các định nghĩa duy nhất bên trong một file. Nó có định dạng như sau:

```
namespace

{

     //declartion

}
```

Unnamed namespaces allow you to establish unique indentifiers that are known only within the scope of a single file. That is, within the file that contains the unnamed namespace, the members of that namespace can be used directly, without qualification. But outside the file, the identifiers are unknown.

Unnamed namespace cho phép bạn tạo một định nghĩa thành lập duy nhất mà bạn biết chỉ bên trong tầm vực một file đơn. Đó là, bên trong một file chứa unnamed namespace, các thành viên của namespace có thể sử dụng trực tiếp, mà không cần điều kiện. Nhưng bên ngoài tập tin, định nghĩa không được sử dụng.

You will not usually need to create namespaces for most small - to medium – sized programs. However, if you will be creating libraries of reusable code or if you want to ensure the widest portability, consider wrapping your code within a namespace.

Bạn sẽ không cần thường xuyên tạo namespace cho các chương trình vừa và nhỏ. Tuy nhiên, nếu bạn muốn tạo một thư viên cho các đoạn mã sử dụng lại hoặc nếu bạn muốn bảo đảm độ rộng, xem bao trùm đoạn mã của bạn bên trong một namespace.

here is an example that illustrates the attributes of a namespace.

Đây là một ví dụ minh họa cho thuộc tính của một namespace

```
// Namespace Demo
#include <iostream>
using namespace std;

// define a namespace
namespace firstNS
{
     class demo
     {
          int i;
          public:
               demo(int x)
               {
                    i = x;
               }
               void seti(int x)
               {
                    i = x;
               }
               int geti()
               {
                    return i;
               }
     };
```

```
     char str[] = "Illustrating namespaces\n";
     int counter;
}

// define another namespace
namespace secondNS
{
     int x, y;
}

int main()
{
     // use scope resolution
     firstNS::demo ob(10);
     /* Once ob has been declared, its member
functions can be used without namespace qualification.
*/
     cout << "Value of ob is: " << ob.geti();
     cout << endl;
     ob.seti(99);
     cout << "Value of ob is now: " << ob.geti();
     cout << endl;

     // bring all of firstNS into current scope
     using namespace firstNS;
```

```
    for(counter = 10; counter; counter--)
        cout << counter << "\n";
    cout << endl;

    // use secondNS namespace
    secondNS::x = 10;
    secondNS::y = 20;

    cout << "x, y: " << secondNS::x;
    cout << " " << second::y << endl;

    // bring another namespace into view
    using namespace secondNS;
    demo xob(x), yob(y);

    cout << "xob, yob: " << xob.geti << ", ";
    cout << yob.geti() << endl;

    return 0;
}
```

The output produced by this program is shown here.

```
Value of ob is : 10
Value of ob is now : 99
Illustrating namespaces
```

```
10 9 8 7 6 5 4 3 2 1

x, y: 10, 20

xob, yob: 10, 20
```

The program illustrates one important point: using one namespace does not override another. When you bring a namespace into view, it simply adds its names to whatever other namespaces are currently in effect. Thus, by the end of this program the std, firstNS, and secondNS namespaces have been added to the global namespace.

Chương trình minh họa một điểm quan trọng: sử dụng một namespace không vượt qua cái khác. Khi bạn mang một namespace vào xem, nó thêm các tên vào bất cứ namespace nào khác đang hoạt động. Vì vậy, cuối chương trình **std, firstNS,** và **secondNS** namespaces được thêm vào toàn bộ namespace.

As mentioned, a namespace can be split between files or within a single file; its contents are additive. Consider this example:

Như đã đề cập, một namespace có thể được chia thành các file bên trong một file đơn, nó chứa các phần thêm vào. Xem ví dụ sau:

```
/ Namespace are additive

#include <iostream>

using namespace std;


namespace Demo

{

     int a; // In Demo namespace

}


int x; // this is in global namespace


namespace Demo

{
```

```
    int b; // this is in Demo namespace, too
}

int main()
{
    using namespace Demo;

    a = b      = x = 100;

    cout << a << " " << b << x;

    return 0;
}
```

*Here the **Demo** namespace contains both **a** and **b**, but not **x**.*

Namespace **Demo** chứa cả a và b, nhưng không chứa x.

*As explained, Standard C++ defines its entire library in its own namespace, **std**. This is the reason that most of the programs in this book have included the following statement:*

Như đạ giải thích, chuẩn C++ định nghĩa toàn bộ thư viện bên trong namespace của nó, std. Đây là lý do hầu hết chương trình trong cuốn sách này đều bao gồm câu lệnh sau:

```
using namespace std;
```

*This causes the **std** namespace to be brought into the current namespace, which gives you direct access to the names of the functions and classes defined within the library without having to qualify each one with **std::**.*

Đây là nguyên nhân namespace std được mang vào namespace hiện hành, nó cho bạn truy cập trực tiếp đến các tên hàm và lớp được định nghĩa bên trong thư viện mà không cần điều kiên cho mỗi cái với std::

*Of course, you can explicitly qualify each name with **std::** if you like. For example, the following program does not bring the library into the global namespace.*

Dĩ nhiên, bạn có thể đặt điều kiện rõ ràng mỗi tên trong std:: nếu bạn muốn. Ví dụ, chương trình sau không mang thư viện vào toàn bộ namespace.

```
// use explicit std:: qualification.

#include <iostream>


int main()

{

     double val;


     std::cout << "Enter a number: ";


     std::cin >> val;


     std::cout << "this is your number: ";

     std::cout << val;


     return 0;

}
```

*Here **cout** and **cin** are both explicitly qualified by their namespace. That is, to write to standard output you must specify **std::cout**, and to read from standard input you must use **std::cin**.*

Cout và cin đều được đặt điều kiện bởi namespace của nó. Đó là, để viết một xuất chuẩn bạn phải chỉ rõ std::cout, và đọc từ nhập chuẩn bản phải sử dụng std::cin.

You might not want to bring the Standard C++ library into the global namespace if your program will be making only limited use of it. However, if

*your program contains hundreds of references to library names, including **std** in the current namespace is far easier than qualifying each name individually.*

Bạn không muốn mang thư viện chuẩn C++ vào toàn bộ namespace nếu chương trình bạn chứa hàng trăm lệnh truy cập đến tên thư viện, bao gồm std trong namespace hiện hành là sớm hơn nhiều so với điều kiện mỗi tên được chia ra.

*If you are using only a few names from the standard library, it might make more sense to specify a **using** statement for each individually. The advantage to this approach is that you can still use those names without an **std::** qualification but you will not be bringing the entire standard library into the global namespace. Here's an example:*

Nếu bạn sử dụng nếu chỉ một số tên từ thư viện chuẩn, nó có thể tạo nhiều ý nghĩa để chỉ rõ một câu lệnh sử dụng cho mỗi phần. Điều thuận lợi cho việc tiếp cận là bạn có thể còn sử dụng những tên này mà không cần điều kiện std::qualification nhưng bạn sẽ không mang toàn bộ thư viện chuẩn vào toàn bộ namespace. Đây là một số ví dụ:

```
// Bring only a few names into the global namespace.

#include <iostream>


// gain access to cout and cin

using std::cout;

using std::cin;


int main()

{

     double val;


     cout << "Enter a number";


     cin >> val;
```

```
        cout << "This is your number: ";
        cout << val;

        return 0;
    }
```

*Here **cin** and **cout** can be used directly, but he rest of the **std** namespace has not been brought into view.*

Cin và Cout có thể được sử dụng trực tiếp, nhưng phần còn lại của namespace std không được mang vào xem.

*As explained, the original C++ library was defined in the global namespace. If you will be converting older C++ programs, you will need to either include a **using namespace std** statement or qualify each reference to a library member with **std::**. This is especially important if you are replacing old **.h** header files with the new-style headers. Remember, the old **.h** headers put their contents into the global namespace. The new-style headers put their contents into the **std** namespace.*

Như đã giải thích, thư viện gốc C++ được định nghĩa trong toàn bộ namespace. Nếu bạn chuyển đổi chương trình C++ cũ, bạn sẽ cần cả câu lệnh using namespace std hoặc điều kiện mỗi câu lệnh truy cập vào một thư viện thành viên với std::. Đây là điều quan trọng đặc biệt nếu bạn thay .h đầu file với tiêu đề theo cách mới. Nhớ rằng, tiêu đề .h cũ đặt nội dung của nó vào toàn bộ namespace. Cách tiêu đề mới đặt nội dung của nó vào namespace std.

*In C, if you want to restrict the scope of a global name to the file in which it is declared, you declare that name as **static**. For example, consider the following two files that are part of the same program.*

Trong C, nếu bạn muốn từ chối tầm vực của một tên chung của một file được khai báo, bạn khai báo các tên đó theo static. Ví dụ, xem hai file sau là một phần của cùng chương trình.

File One

```
static int conter;

void f1( )

{

     Conter = 99; // OK
```

File Two

```
extern int counter;

void f2( )

{

     counter = 10; // error
```

```
}
```

```
}
```

*Because **counter** is defined in File One, it can be used in File One. In File Two, even though **counter** is specified as **extern**, it is still unavailable, and any attempt to use it results in an error. By preceding **counter** with **static** in File One, the programmer has restricted its scope to that file.*

Bởi vì counter được định nghĩa trong file một, nó có thể được sử dụng trong fiel một. Trong file hai, thậm chí counter được chỉ rõ như extern, nó không có giá trị, và bất kỳ sự cố gắng nào sử dụng nó thì kết quả đều báo lỗi. Lập trình viên bị từ chối tầm vực của một file, bởi counter vời static trong file một.

*Although the use of **static** global declarations is still allowed in C++, a better way to accomplish the same end is to use an unnamed namespace, as shown in this example:*

Mặc dù cách sử dụng static toàn bộ khai báo là còn được phép trong C++, một cách tốt hơn để hoàn thành cùng cách kết thúc là sử dụng một unnamed namespace, như đã chỉ ra trong ví dụ này:

File One

```
namespace
{
     int counter;
}
void f1( )
{
     counter = 99; // OK
}
```

File Two

```
extern int counter;
void f2( )
{
     counter = 10; // error
}
```

*Here **counter** is also restricted to File One. The use of the unnamed namespace rather than **static** is the method recommended by Standard C++.*

Counter này cũng bị hạn chế trong file một. Cách sử dụng của unnamed namespace hơn static là phương pháp yêu cầu bởi chuẩn C++.

EXERCISES:

*Convert the following program from Chapter 9 so that it does not use the **using namespace std** statement:*

Chuyển đổi chương trình sau từ chương 8 không sử dụng using namespace std:

```
// Convert spaces to d/
#include <iostream>
#include <istream>
using namespace std;

int main (int arge, char *argv[])
{
     if(argv != 3)
     {
          cout << "Usage: CONVERT <input>  <output>\n";
          return 1;
     }

     ifstream.fin argv[1]; // open input file
     ofstream.fout argv[2]; // create putput file

     if(!foul)
     {
          cout << "Cannot open output file.\n";
          return 1;
     }
```

```
        if(!fin)
        {
              cout << "Cannot open input file.\n";
              return 1;
        }

        char ch;

        fin.unsetf(ios::skipws); // do not skip spaces
        while(!fin.eof())
        {
              fin >> ch;
              if(ch == ' ')
                    ch = '!';
              if(!fin.eof())
                    fout << ch;
        }

        fin.close();
        fout.close();

        return 0;
  }
```

Explain the operation of an unnamed namespace.

Giải thích toán tử unnamed namespace.

*Explain the difference between the two forms of **using**.*

Giải thích sự khác nhau của hai cách sử dung using

*Explain why most programs in this book contain a **using** statement. Describe one alternative.*

Giải thích tại sao hầu hết chương trình trong cuốn sách này sử dụng câu lệnh using. Miêu ta cách thay đổi.

Explain why you might want to put reusable code that you create into its own namespace.

Giải thích tại sao bạn muốn đặt đoạn code sử dụng lại mà bạn tạo trong namespace của nó.

13.2. CREATING A CONVERSION FUNCTION – TẠO MỘT HÀM CHUYỂN ĐỔI

Sometime it is useful to convert an object of one type into an object of another. Although it is possible to use an overloaded operator function to accomplish such a conversion, there is often an easier (and better) way: using a conversion function. A conversion function converts an object into a value compatible with another type, which is often one of the built-in C++ types. In essence, a conversion function automatically converts an object into a value that is compatible with the type of the expression in which the object is used.

Đôi khi nó hữu dụng để chuyển đổi một đối tượng của một loại vào một loại đối tượng khác. Mặc dù nó có thể sử dụng một hàm toán tử nạp chồng để hoàn thành một chuyển đổi, có một cách dễ hơn (và tốt hơn) : sử dụng hàm chuyển đổi. Một hàm chuyển đổi thay đổi một đối tượng thành một giá trị thích hợp với một loại khác, điều thường là một đối tượng được xây dưng trong C++. Điều cần thiết, một hàm chuyển đổi tự động thay đổi một đối tượng thành một giá trị thích hợp với loại được xác định mà đối tượng sử dụng.

The general form of a conversion function is shown here:

Kiểu mẫu của hàm chuyển đổi là:

```
operator type()
```

```
{
     return value;
}
```

Here type is the target type you will be converting to and value is the value of the object after the conversion has been performed.

Type là loại mục tiêu bạn sẽ chuyển đổi thành và *value* là giá trị của đối tượng sau khi chuyển đổi được thực hiện.

Conversion functions return a value of type type. No parameters can be specified, and a conversion function must be a member of the class for which it performs the conversion.

Các hàm chuyển đổi trả về một giá trị của loại *type*. Không tham số nào có thể được chỉ rõ, và một hàm chuyển đổi phải là một thành viên của lớp mà nó thực hiện cuộc chuyển đổi.

As the examples will illustrate, a conversion function generally provides a cleaner approach to converting an object's value into another type than any other method available in C++ because it allows an object to be included directly in an expression involving the target type.

Như các ví dụ trên đã chứng minh, một hàm chuyển đổi có thể cung cấp một cách tiếp cận chung để chuyển đổi giá trị của một đối tượng thành loại khác hơn bất kỳ phương pháp có giá trị nào trong C++ bởi vì nó cho phép một đối tượng được bao gồm trực tiếp trong một cách giải quyết ấn tượng loại mục tiêu.

*In the following program, the **coord** class contains a conversion function that converts an object to an integer. In this case, the function returns the product of the two coordinates; however, any conversion appropriate to your specific application is allowed.*

Trong chương trình sau, lớp **coord** chứa một hàm chuyển đổi thay đổi một đối tượng thành số nguyên. Trong trường hợp này, hàm trả về sản phẩm của hai hàm đồng nhau. Tuy nhiên, bất kỳ sử chuyển đổi nào thích hợp sử áp dụng được chỉ rõ.

```
// A simple conversion function example.

#include <iostream>
```

```
using namespace std;

class coord
{
     int x, y;
     public:
          coord(int i, int j)
          {
               x = i;
               y = j;
          }
          operator int()
          {
               return x*y; // conversion function
          }
};

int main()
{
     coord o1(2,3), o2(4,3);
     int i;

     i = o1; // automatically convert to integer
     cout << i << '\n';
```

```
    i = 100 - o2; // convert o2 to integer
    cout << i << '\n';

    return 0;
}
```

This program displays 6 and 112.

*In this example, notice that the conversion function is called when **o1** is assigned to an integer and when **o2** is used as part of a larger integer expression. As stated, by suing a conversion function, you allow classes that you create to be integrated into "normal" C++ expressions without having to create a series of complex overloaded operator functions.*

Trong ví dụ này, chú ý hàm chuyển đổi được gọi khi o1 được thừa hưởng từ một số nguyên và khi o2 được sử dụng như một phần của số nguyên dài. Như đã nói, bằng cách sử dụng một hàm chuyển đổi, bạn cho phép các lớp mà bạn đã tạo hợp thể thành C++ bình thường mà không cần tạo một số hàm toán tử nạp chồng phức tạp.

*Following is another example of conversion function. This one converts a string of type **strtype** into a character pointer to **str**.*
Sau đây là một ví dụ khác của hàm chuyển đổi. Đây là một sự chuyển đổi một chuỗi loại strtype thành một con trỏ chuỗi str.

```
#include <iostream>
#include <cstring>
using namespace std;

class strtype
{
    char str[80];
```

```
        int len;
        public:
                strtype(char *s)
                {
                        strcpy(str, s);
                        len = strlen(s);
                }
                operator char *()
                {
                        retrun str; // convert to char*
                }
};

int main()
{
        strtype s("this is a test/n");
        char *p, s2[80];

        p = s; // convert to char*
        cout << "Here is string: " << p<< '\n';

        //convert to char* in function call
        strcpy(s2,s);
        cout << "Here is copy of string: " << s2 << '\n';
```

```
    return 0;
}
```

This program displays the following:

```
Here is string: This is a test
Here is copy of string: This is a test
```

*As you can see, not only is the conversion function invoked when object **s** is assigned to **p** (which is of type **char***), but it is also called when **s** is used as a parameter to **strcpy( )**. Remember **strcpy( )** has the following prototype:*

Như bạn đã thấy, không chỉ hàm chuyển đổi cần thiết khi đối tượng s thừa hưởng p (loại char*) , nhưng nó cũng được gọi khi s được sử dụng như một tham số của strcpy( ). Nhớ rằng strcpy( ) có định dạng như sau:

```
char*strcpy(char*s1, const char *s2);
```

*Because the prototype specifies that s2 is of type **char*** , the conversion function to **char*** is automatically called. This illustrates how a conversion function can also help you seamlessly integrate your classes into C++'s standard library functions.*

Bởi vì định dạng chỉ rõ rằng s2 là một loại char*, hàm chuyển đổi thành char* được tự động gọi. Điều này chứng minh làm thế nào một hàm chuyển đổi có thể giúp bạn gọi lớp của bạn thành các hàm thư viện chuyển của C++.

EXERCISES:

*Using the **strtype** class from Example 2, create a conversion that converts to type **int**. In this conversion, return the length of the string held in **str**. Illustrate that your conversion function works.*

Sử dụng lớp strtype từ bài tập 2, tạo một hàm chuyển đổi thay đổi thành số nguyên. Trong chuyển đổi này, trả về chiều dài của chuỗi giữ trong str. Chứng minh rằng hàm chuyển đổi của bạn hoạt động.

Given this class:

Cho lớp sau:

```
class pwr
{
     int bse;
     int exp;
     public:
          pwr(int b, int e)
        {
          base = b;
          exp = e;
          // create conversion to integer here
        }
};
```

*create a conversion function that converts an object of type **pwr** to type integer. Have the function return the result of **base**^{exp}.*

tạo một hàm chuyển đổi thay đổi một đối tượng loại pwr thành loại số nguyên. Có hàm trả về kết quả của base^{exp}.

13.3. STATIC CLASS AND MEMBERS – LỚP TĨNH VÀ CÁC THÀNH PHẦN

*It is possible for a class member variable to be declared as **static**. By using **static** member variables, you can bypass a number of rather tricky problems. When you declare a member variable as **static**, you cause only one copy of that variable to exist-*

*no matter how many objects of that class are created. Each object simply shares that one variable. Remember, for a normal member variable, each time an object is created, a new copy of that variable is created, and that copy is accessible only by that object. (That is, when normal variables are used, each object possesses its own copies.) However, there is only one copy of a **static** member variable, and all objects of its classes derived from the class that contains the **static** member.*

Có khả năng cho một thành viên lớp lưu động được khai báo dạng tĩnh. Bằng cách sử dụng thành viên tĩnh lưu động, bạn có thể đi vòng một số vấn đề khá phức tạp. Khi bạn khai báo một thành viên động như tĩnh, bạn gây ra chỉ một sao chép của thành viên động để không xuất hiện vấn đề, làm thế nào nhiều đối tượng của một lớp được khởi tạo. Mỗi đối tượng đơn giản chia sẻ thành nhiều động. Nhớ rằng, cho một thành viên động bình thường, mỗi lần một đối tượng được tạo, một bản copy mới của thành viên động được khởi tạo, và một copy được truy cập chỉ có đối tượng đó. (Đó là, khi một động bình thường được sử dụng, mỗi đối tượng chiếm hữu copy của nó). Tuy nhiên, chỉ có một copy của thành viên tĩnh thay đổi, và tất cả đối tượng của các lớp dẫn xuất từ lớp chứa thành viên tĩnh.

*Although it might seem odd when you first think about it, a **static** member variable exists before any object of its class is created. In essence, a **static** class member is a global variable that simply has its scope restricted to the class in which it is declared. In fact, as you will see in one of the following examples, it is actually possible to access a **static** member variable independent of any object.*

Mặc dù nó có vẻ dư thừa khi lần đầu bạn nghĩ về nó, một thành viên tĩnh có thể thay đổi xuất hiện trước khi bất kỳ đối tượng của lớp của nó được khởi tạo. Về bản chất, một thành viên lớp tĩnh là hoàn toàn thay đổi hạn chế tầm hoạt động của nó để lớp trong được khai báo. Sự thật, bạn sẽ thấy trong một ví dụ sau, nó thật sự có thể truy cập một thành tố tĩnh có thể thay đổi của bất kỳ đối tượng độc lập nào.

*When you declare a **static** data member within a class, you are not defining it. Instead, you must provide a definition for it elsewhere, outside the class. To do this, you redeclare the **static** variable, using the scope resolution operator to identify which class it belongs to.*

Khi bạn khai báo dữ liệu thành tố tĩnh bên trong một lớp, bạn không định nghĩa nó. Thay về vậy, bạn phải cung cấp một định nghĩa cho nó, bên ngoài lớp. Để làm điều này, bạn phải khai báo lại biến tĩnh thay đổi, sử dụng toán tử tầm vực để xác định lớp nó thuộc về.

*All **static** member variables are initialized to 0 by default. However, you can give a **static** class variable an initial value of your choosing if you like.*

Tất cả thành tố tĩnh có thể thay đổi được bắt đầu từ 0 như mặc định. Tuy nhiên, bạn có thể cho một lớp tĩnh có thể thay đổi một giá trị khởi đầu theo lựa chọn của bạn.

Keep in mind that the principal reason **static** member variables are supported by C++ is to avoid the need for global variables. As you can surmise, classes that rely upon global variables almost always violate the encapsulation principle that is so fundamental to OOP and C++.

Nhớ rằng lý do chính thành viên tĩnh có thể thay đổi được cung cấp bởi C++ là để tránh cần thiết cho toàn bộ thay đổi. Như bạn dự đoán, các lớp dựa vào toàn bộ thay đổi hầu hết luôn luôn xâm phạm gói gọn chính mà nó cơ bản cho OOP và C++.

*It is also possible for a member function declared as **static** can this usage is not common. A member function to be declared as **static** can access only other **static** members of its class. (Of course, a **static** member function can access non-**static** global data and functions.) A **static** member function does not have a **this** pointer. Virtual **static** member functions are not allowed. Also, **static** member functions cannot be declared as **const** or **volatile**. A **static** member function can be invoked by an object of its class, or it can be called independent of any object, via the class name and the scope resolution operator.*

Nó cũng có thể cho một hàm thành viên được khai báo như tĩnh có thể cách sử dụng này không phổ biến. Một hàm thành viên được khai báo tĩnh có thể truy cập chỉ thành tố tĩnh của lớp. (Dĩ nhiên, một hàm thành viên tĩnh có thể truy cập toàn bộ dữ liệu và hàm không tĩnh.) Một hàm thành viên tĩnh không có con trỏ this. Các hàm thành viên tĩnh ảo không được phép. Các hàm thành viên tĩnh không được khai báo như const hay volative. Một hàm thành viên tĩnh có thể viện dẫn bởi một đối tượng của lớp, hoặc nó có thể được gọi bất kỳ đối tượng độc lập, theo đường tên lớp và toán tử tầm vực.

Here is a simple example that uses a **static** member variable:
Đây là một ví dụ đơn giản sử dụng thành tố tĩnh thay đổi

```
// A static member variable example.

#include <iostream>

using namespace std;


class myclass
```

```
{
     static int i;
     public:
          void seti(int n)
          {
               i = n;
          }
          int geti()
          {
               return i;
          }
};

// Definition of myclass::i. i is still private to
myclass
int myclass::i;

int main()
{
     myclass o1, o2;

     o1.seti(10);

     cout << "o1.i: " << o1.get << '\n'; // displays 10
     cout << "o2.i: " << o2.get << '\n'; // also
displays 10
```

```
    return 0;
}
```

This program displays the following:

```
o1.i: 10
o2.i: 10
```

*Looking at this program, you can see that only object **o1** actually sets the value of **static** member **i**. However, since **i** is shared by both **o1** and **o2** (and, indeed, by any object of type **mycalss**), both calls to **geti( )** display the same result.*

Xem chương trình này, bạn có thể thấy rằng chỉ đối tượng ó thật sự có giá trị của thành tố tĩnh i. Tuy nhiên, từ khi i được chia sẻ bởi cả o1 và o2 (và, thật sự, bởi bất kỳ đối tượng của loại myclass), cả hai cuộc gọi cho geti( ) đều cho một kết quả.

*Notice how **i** is declared within **myclass** but defined outside of the class. This second step ensures that storage for **i** is defined. Technically, a class declaration is just that, only a declaration. No memory is actually set aside because of a declaration. Because a **static** data member implies that memory is allocated for that member, a separate definition is required that causes storage to be allocated.*

Chú ý làm thế náo i được khai báo bên trong myclass nhưng được định nghĩa bên ngoài lớp. Bước thứ hai này bảo đảm rằng chỗ chứa cho i được định nghĩa. Kỹ thuật, một khai báo lớp chỉ là một khai báo. Không có bộ nhớ thật sự tao riêng ra bởi vì một khai báo. Bởi vì thành viên dữ liệu tĩnh bao hàm ý bộ nhớ được chỉ định cho thành tố, một định nghĩa rời rạc được yêu cầu gây ra chỗ chứa được chỉ định.

*Because a **static** member variable exists before any object of that class is created, it can be accessed within a program independent of any object. For example, the following variation of the preceding program sets the value of **i** to 100 without any reference to a specific object. Notice the use of the scope resolution operator and class name to access **i**.*

Bởi vì thành tố tĩnh thay đổi xuất hiện trước khi bất kỳ đối tượng nào của lớp

được tạo, nó có thể truy cập bên trong một chương trình của đối tượng độc lập. Ví dụ, sự thay đổi của chương trình trước đặt giá trị của i là 100 mà không truy cập đến giá trị thích hợp. Chú ý rằng cách sử dụng của toán tử tầm vực và tên lớp để truy cập i.

```
// Use a static member variable independent of any
object.

#include <iostream>

using namespace std;


class myclass

{

     public:

          static int i;

          void seti(int n)

          {

               i = n;

          }

          int geti()

          {

               return i;

          }

};


int myclass::i;


int main()

{
```

```
    myclass o1,o2;


    // set i directly
    myclass::i = 100; // no object is referenced


    cout << "o1.i: " << o1.geti() << '\n'; //
displays 100
    cout << "o2.i: " << o2.geti() << '\n'; // also
displays 100
    return 0;
}
```

*Because **i** is set to 100, the following output is displayed:*

```
o1.i: 100
o2.i: 100
```

*One very common use of a **static** class variable is to coordinate access to a shared resource, such as a disk file, printer, or network server. As you probably know from your previous programming experience, coordinating access to a shared resource requires some means of sequencing events. To get an idea of how **static** member variables can be used to control access to a shared resource, examine the following program. It creates a class called **output**, which maintains a common output buffer called **outbuf** that is, itself, a **static** character array. This buffer is used to receive output sent by the **outbuf( )** member function. This function sends the contents of **str** one character at a time. It does so by first acquiring access to the buffer and then sending all the characters in **str**. It locks out access to the buffer by other objects until it is done outputting. You should be able to follow its operation by studying the code and reading the comments.*

Một cách sử dụng chung của lớp tĩnh thay đổi là truy cập ngang hàng đến nguồn được chia sẽ, như là file đĩa, máy in, hoặc mạng. Như bạn biết từ kinh nghiệm chương trình trước, truy cập ngang hàng đến nguồn mở yêu cầu vài giá trị trung bình của dãy số. Để lấy một ý kiến của làm thế nào thành tố tĩnh thay đổi có thể được sử dụng để điều khiển truy cập nguồn mở, ví dụ chương trình sau. Nó tạo

một lớp gọi là output, duy trì một xuất chung vật đệm được gọi là outbuf, dó là chuỗi ký tự tĩnh. Vật đệm này được sử dụng để nhận phần xuất được gửi bởi hàm thành viên outbuf( ). Hàm này gửi nội dung của str một ký tự mỗi lần. Nó là yêu cầu đầu tiên truy cập đến vật đệm và sau đó gửi tất cả ký tự vào str. Nó khóa truy cập đến vật đệm bởi các đối tượng khác đến khi nó thực hiện phần xuất. Nó có thể theo sau toán tử bởi nghiên cứu code và đọc lệnh.

```
// A shared resource example.

#include <iostream>

#include <cstring>

using namespace std;

class output

{

     static char outbuf[255]; // this is the shared
resource

     static int inuse; // buffer available if 0;

     static into index; // index of outbuf

     char str[80];

     int i; // index of next char in str

     int who; // identifies the object, must be > 0

     public:

          output(int w, char *s)

          {

               strcpy(str, s);

               i = 0;

               who = w;

          }


          /* This function returns -1 if waiting for
```

```
buffer
        it returns 0 if it is done outputting, and
        it returns who if it is still using the
        buffer.
     */
     int putbuf()
     {
          if(!str[i]) // done putputting
          {
               inuse = 0; // release buffer
               return 0; // signal termination
          }
          if(!inuse)
               inuse = who; // get buffer
          if(inuse != who)
               return -1; // inuse by someone else
          if(str[i]) // still chars to output
          {
               outbuf[oindex[ = str[i];
               i++;
               oindex++;
               outbuf[oindex] = '\0'; // always
          keep null-terminated
               return 1;
          }
          return 0;
```

```
        }
        void show()
        {
            cout << outbuf << '\n';
        }
};

char output::outbuf[255]; // this is the shared
resource
int output::inuse = 0; // buffer available if 0
int output::oindex = 0; // index of outbuf

int main()
{
    output o1(1, "This is a test"), o2(2, "of
statics");

    while(o1.putbuf()|| o2.putbuf()); // output chars

    o1.show();

    return 0;
}
```

static *member functions have limited applications, but one good use for them is to "preinitialize" private* ***static*** *data before any object is actually created. For example, this is a perfectly valid C++ program:*

Hàm thành viên tĩnh giới hạn sự áp dụng, nhưng cách sử dụng hay là khởi động trước dữ liệu tĩnh riêng trước khi bất kỳ đối tượng nào thật sử được tao. Ví dụ, đây là một chương trình C++:

```
#include <iostream>
using namespace std;

class static_func_demo
{
     static int i;
     public:
          static void init(int x)
          {
               i = x;
          }
          void show()
          {
               cout << i;
          }
};

int static_func_demo::i; //define i

int main()
{
     // init static data before object creation
```

```
        static_func_demo::init(100);
        static_func_demo x;
        x.show(); //displays 100

        return 0;
    }
```

EXERCISES:

Rework Example 3 so that it displays which object is currently outputting characters and which one or ones are blocked from outputting a character because the buffer is already in use by another.

Làm lại ví dụ 3 mà nó thực hiện với đối tượng hiện hành xuất ra các ký tự và một hoặc nhiều bị chặn từ phần xuất một ký tự bởi vì vùng đệm đã được sử dụng.

*One interesting use of a **static** member variable is to keep track of the number of objects of class that are in existence at any given point in time. The way to do this is to increment a **static** member variable each time the class ' constructor is called and decrement it each time the class' destructor is called.*
Implement such a scheme and demonstrate that it works.

Một cách sử dụng của thành tố tĩnh thay đổi là giữ dấu của một số đối tượng của lớp đã xuất hiện tại bất kỳ điểm nào cho trước mỗi lần. Đây là cách làm lớn lên một thành tố tĩnh thay đổi mỗi lần phương thức khởi tạo của lớp được gọi và làm giảm một sự sắp xếp mỗi lần phương thức phá hủy được gọi.

Thực hiện như một cách sắp xếp và giải thích rằng cách hoạt động của nó.

13.4. CONST MEMBER FUNCTIONS AND MUTABLE - HÀM THÀNH PHẦN KHÔNG ĐỔI VÀ CÓ THỂ THAY ĐỔI

Class member function can be declared as const. When this is done, that function cannot modify the obj that invokes it. Also, a const obj cannot invoke a non-const member function. However, a const member function can be called by either const or

non-const obj.

To specify a member function as const, use the form shown in the following example:

Hàm lớp thành phần có thể được khai báo như là const. Khi nó được thực hiện, hàm đó không thể thay đổi đối tượng gọi nó. Cũng vậy,một đối tượng const có thể không gọi một hàm thành phần non-const. Tuy nhiên, một hàm thành phần const có thể được gọi bởi cả đối tượng const hay non-const.

Chỉ rõ một hàm thành phần như là const, dùng một dạng cho thấy trong ví dụ dưới đây:

```
class X

{

     int some_var;

     public:

          int f1() const; // const member function

};
```

As you can see, the const follows the function's parameter declaration.

Sometimes there will be one or more members of a class that you want a const function to be able to modify even though you don't want the function to be able to modify any of its other members. You can accomplish this through the use of mutable, which overrides const-ness. That is, a mutable member can be modified by a const member function.

Như bạn thấy, const cho theo sau hàm khai báo tham số.

Đôi khi sẽ có một hay nhiều thành phần của một lớp mà bạn muốn một hàm const có thể thay đổi thậm chí khi bạn không muốn hàm có thể thay đổi bất kì thành phần nào khác. Bạn có thể hoàn thành việc này thông qua cách sử dung mutable, có thể ghi đè const-ness. Khi đó, một hàm mutable có thể thay đổi bởi một hàm thành phần.

EXAMPLES (VÍ DỤ)

1. The purpose of declaring a member function as const is to prevent it from modifying the obj that invokes it. For example, consider the following program.

Mục đích của việc khai báo một hàm thành phần như const để bảo vệ nó khỏi sự thay đổi

đối tượng mà gọi nó. Ví dụ, xem xét chương trình dưới đây.

```
/* Demonstrate const member function.This program won't
compile. */

#include <iostream>

using namespace std;


class Demo

{

     int i;

     public:

          int geti() const

               return i; // ok

          void seti(int x) const

               i=x; // error!

};


int main()

{

     Demo ob;


     ob.seti(1900);

     cout<<ob.geti();

     return 0;

}
```

This program will not compile because seti() is declared as const. This means that it is not allowed to modify the invoking obj. Since it attempts to change i, the program is in

error. In contrast, since geti() does not attempt to modify i, it is perfectly acceptable.

Chương trình này sẽ không biên dịch bởi vì seti() được khai báo như là const. Điều này có nghĩa là nó sẽ không được cho phép thay đổi đối tượng. Khi nó cố thay đổi i, chương trình sẽ bị lỗi. Ngược lại, khi geti() không cố thay đổi i, nó có thể được chấp nhận hoàn hảo.

2. To allow selected members to be modified by a const member function, specify them as mutable. Here's an example:

Cho phép những thành phần được chọn được thay đổi bởi một hàm thành phần const, chỉ rõ nó như là mutable. Như ví dụ dưới đây:

```
// Demonstrate mutable.

#include <iostream>

using namespace std;

class Demo

{

     mutable int i;

     int j;

     public:

          int geti() const

               return i; // ok

          void seti(int x) const

               i=x; // now, ok.


/* The following function won't compile.

void setj(int x) const

     j=x; // still wrong!

*/

};
```

```
int main()

{

      Demo ob;


      ob.seti(1900);

      cout<<ob.geti();


      return 0;

}
```

Here i is specified as mutable, so it can be changed by the seti() function. However, j is not mutable, so setj() is unable to modify its value.

Ở đây i được chỉ rõ như mutable, vì nó có thể được thay đổi bởi hàm seti(). Tuy nhiên, j không là mutable, vì vậy setj() không thể thay đổi được giá trị của nó.

EXERCISES (BÀI TẬP)

1. The following program attempts to create a simple countdown timer that rings a bell when the time period is over. You can specify the time period and increment when a CountDown obj is created. Unfortunately, the program will not compile as a shown here. Fix it.

Chương trình dưới đây cố tạo đồng hồ đếm ngược thời gian đơn giản mà chuông sẽ reng khi hết thời gian. Bạn có thể chỉ rõ khoảng thời gian và số lượng khi một đối tượng CountDown được tạo. Đáng tiếc, chương trình sẽ không biên dịch như một trình bày này. Hãy sắp xếp nó.

```
// This program contains an error.

#include <iostream>
```

```
using namespace std;

class CountDown
{
     int incr;
     int target;
     int current;
     public:
          CountDown(int delay, int i=1)
          {
               target=delay;
               incr=i;
               current=0;
          }
     bool counting() const
     {
          current += incr;
          if(current >= target)
          {
               cout<<”\a”;
               return false;
          }
          cout<<current<<” “;
          return true;
     }
```

```
};


int main()
{
     CountDown ob(100, 2);


     while(ob.counting());
     return 0;
}
```

2. Can a const member function call a non-const function? Why not?

Một hàm thành phần const có thể gọi như một hàm non-const? Tại sao không?

13.5. A FINAL LOOK AT CONSTRUCTORS - CÁI NHÌN CUỐI CÙNG VỀ HÀM

Although constructors were described early on in this book, there are still a few points that need to be made. Consider the following program:

Mặc dù hàm dựng được mô tả sớm trong cuốn sách này, vẫn còn một vài điểm cần được làm. Xem xét chương trình dưới đây:

```
#include <iostream>
using namespace std;


class myclass
{
     int a;
     public:
```

```
        myclass(int x)
             a=x;
        int geta()
             return a;
};

int main()
{
     myclass ob(4);

     cout<<ob.geta();

     return 0;
}
```

Here the constructor for myclass takes one parameter. Pay special attention to how ob is declared in main(). The value 4, specified in the parentheses following ob, is the argument that is passed to myclass() 's parameter x, which is used to initialize a. This is the form of initialization that we have been using since the start of this book. However, there is an alternative. For example, the folllowing statement also inittializes a to 4:

Ở đây hàm dựng cho myclass nhận một tham số. Đưa ra một chú ý đặc biệt là ob được khai báo trong hàm main() như thế nào. Giá trị 4, đã chỉ rõ trong ngoặc đơn của ob dưới đây, là đối số mà vượt qua tham số x của myclass(), được dùng để khởi tạo a. Đây là dạng của khởi tạo mà chúng ta đã sử dụng từ khi bắt đầu cuốn sách này. Tuy nhiên, đây là một sự thay thế. Ví dụ, đoạn chương trình dưới đây cũng khởi tạo a là 4.

```
myclass ob=4; // automatically converts into myclass(4)
```

As the comment suggests, this form of initialization is automatically converted into a call to the myclass constructor with 4 as the argument. That is, the preceding statement

is handled by the compiler as if it were written like this:

Như lời chú giải đề nghị, dạng này của sự khởi tạo được tự động chuyển đổi thành một lời gọi hàm dựng myclass với 4 như là một đối số. Điều đó là, đoạn chương trình trước được vận dụng bởi trình biên dịch như là nó được viết như:

```
myclass ob(4);
```

In general, any time that you have a constructor that requires only one argument, you can use either ob(x) or ob=x to initialize an obj. The reason for this is that whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class.

Nói chung, bất kỳ thời điểm nào bạn cũng có một hàm dựng chỉ cần một đối số, bạn có thể dùng cả ob(x) hay ob=x để khởi tạo một đối tượng. Lý do là bất cứ khi nào bạn cũng có thể tạo một hàm dựng mà lấy một đối số, bạn cũng hoàn toàn tạo sự chuyển đổi từ dạng đối số này sang dạng đối số khác của lớp.

If you do not want implicit conversions to be made, you can prevent them by using explicit. The explicit specifier applies only to constructors. A constructor specified as explicit will be used only when an initialization uses the normal constructor syntax. It will not perform any automatic conversion. For example, if the myclass constructor is declared as explicit, the automatic conversion will not be supplied. Here is myclass() declared as explicit.

Nếu bạn không muốn sự chuyển đổi ngầm được thực hiện, bạn có thể ngăn chúng bằng cách sử dụng explicit. Explicit không chỉ áp dụng cho hàm dựng. Một hàm dựng chỉ rõ như là explicit sẽ được dùng chỉ khi một sự khởi tạo dùng cú pháp dựng bình thường. Nó sẽ không trình bày bất kỳ đối số tự động nào. Ví dụ, nếu hàm dựng myclass được dẫn xuất như là explicit, đối số tự động sẽ không được cung cấp. Ở đây myclass() được dẫn xuất như là explicit.

```
#include <iostream>

using namespace std;


class myclass

{

     int a;

     public:
```

```
        explicit myclass(int x)
            a=x;
        int geta()
            return a;
};
```

Now only constructors of the form

```
myclass ob(110);
```

will be allowed.

EXAMPLES (VÍ DỤ)

1. There can be more than one converting constructor in a class. For example, consider this variation on myclass:

Có nhiều hơn một sự chuyển đổi của hàm dựng trong một lớp. Ví dụ, xem xét sự khác biệt trong lớp myclass.

```
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass
{
    int a;
    public:
        myclass(int x)
            a=x;
        myclass(char *str)
            a=atoi(str);
```

```
        int geta()
             return a;
};

int main()
{
     myclass ob1=4; // converts to myclass(4)
     myclass ob2="123"; // converts to myclass("123");

     cout<<"ob1:"<<ob1.geta()<<endl;
     cout<<"ob2:"<<ob2.geta()<<endl;

     return 0;
}
```

Since both constructors use different type arguments(as, of course, they must), each initialization statement is automatically converted into its equivalent constructor call.

Khi cả hai hàm dựng dùng loại đối số khác nhau(như là, dĩ nhiên, nó phải), mỗi sự khởi tạo đoạn chương trình được tự động chuyển đổi thành lời gọi hàm dựng tương đương của nó.

2. The automatic conversion from the type of a constructor's first argument into a call to the constructor itself has interesting implications. For example, assuming myclass from Example 1, the following main() function makes use of the conversions from int and char to assign ob1 and ob2 new values.*

Sự chuyển đổi tự động từ loại của một đối số đầu tiên của hàm dựng trong một lời gọi chính hàm dựng có một sự liên quan thú vị. Ví dụ, thừa nhận rằng lớp myclass trong ví dụ 1, hàm main() dưới đây sử dụng cách chuyển đổi từ int và char* để gán cho ob1 và ob2 giá trị mới.

```
int main()
```

```
{
     myclass ob1=4; // converts to myclass(4)
     myclass ob2="123"; // converts to myclass("123");

     cout<<"ob1:"<<ob1.geta()<<endl;
     cout<<"ob2:"<<ob2.geta()<<endl;

     // use automatic conversion to assign new values
     ob1="1776"; // converts into ob1=myclass("1776");
     ob1=2001; //converts into ob2=myclass(2001);

     cout<<"ob1:"<<ob1.geta()<<endl;
     cout<<"ob2:"<<ob2.geta()<<endl;

     return 0;
}
```

3. To prevent the conversions just shown, you could specify the constructors as explicit, as shown here:

Để ngăn sự sự chuyển đổi chỉ được trình bày, bạn có thể chỉ rõ hàm dựng như là explicit, như được trình bày dưới đây:

```
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass
{
```

```
    int a;
    public:
        explicit myclass(int x)
            a=x;
        explicit myclass(chat *str)
            a=atoi(str);
        int geta()
            return a;
};
```

EXERCISES (BÀI TẬP)

*In Example 3, if only myclass(int) is made explicit, will myclass(char *) still allow implicit conversions? (Hint: try it.)*

Trong ví dụ 3, chỉ duy nhất myclass(int) được làm explicit, myclass(char*) sẽ vẫn cho phép sự chuyển đổi ngầm? (Gợi ý:thử làm nó đi)

Will the following fragment work?

Đoạn dưới đây sẽ làm việc như thế nào?

```
class Demo
{
    double x;
    public:
        Demo(double i)
            x=i;
    // ...
};

// ...
```

```
Demo counter=10;
```

Justify the inclusion of the explicit keyword. (In other words, explain why implicit constructor conversions might not be a desirable feature of C++ in some cases.)

Giải thích sự thêm vào của từ khóa explicit. (Mặt khác, giải thích tại sao sự chuyển đổi hàm dựng có thể không là một tính năng đáng ao ước của C++ trong một số trường hợp.)

13.6. USING LINKAGE SPECIFIERS AND THE ASM KEYWORD

***C++** providers two important mechanisms that make it easier to link C++ to other languages. One is the linkage specifier. Which tells the compiler that one or more functions in your C++ program wil be linked with another language that might have a different approach to naming, parameter passing, stack restoration, and the like. The second is the **asm** keyword, which allows you to embed assembly language instructions in your C++ source code. Both are examined here.*

C++ cung cấp cho chúng ta 2 kỹ thuật quan trọng để có thể kết nối dễ dàng các ngôn ngữ với nhau. Một trong những cách đó là kỹ thuật "chỉ rõ kết nối". Kỹ thuật này cho phép trình biên dịch có thể nhận diện ra được các hàm trong chương trình được liên kết với các ngôn ngữ khác mà có thể có sự khác nhau về cách đặt tên, tham biến, stack, và những cái đại loại như thế. Kỹ thuật thứ 2 là dùng từ khóa asm, kỹ thuật này cho phép nhúng cấu trúc lệnh của hợp ngữ vào mã nguồn C++. Tất cả đều được nghiên cứu dưới đây.

By default, all functions in C++ program are compiled and linked as C++ functions. However, you can tell the c++ compiler to link a function so that it is compatible with another type of language. All C++ compilers allow functions to be linked as either C or C++ functions. Some also allow you to link functions with languages such as PasCal, Ada or FORTRAN. To cause a function to be linked for a different language, use this general form of the linkage specification:

Ở chế độ mặc định, tất cả các hàm trong chương trình C++ được biên dịch và liên kết như là các hàm C++. Tuy nhiên, bạn có thể chỉ cho trình biên dịch của C++ biên dịch một hàm để cho hàm đó tương thích với các ngôn ngữ lập trình khác. Tất cả các trình biên dịch C++ đều cho phép hàm được liên kết không là hàm C hoặc C++. Một vài cái khác

thì cho phép bạn liên kết các hàm với các ngôn ngữ lập trình khác, chẳng hạn như Pascal, Ada hay FORTRAN. Bởi vì lý do một hàm có thể liên kết với các ngôn ngữ khác, nên một hình thức phổ biến là chỉ rõ "ngôn ngữ liên kết "

Here language is the name of the language with which you want the specified function to link. If you want to specify linkage for more than one function, use this form of the linkage specification:

Phía dưới là tên của ngôn ngữ mà bạn muốn chỉ rõ hàm liên kết. Nếu bạn muốn chỉ rõ liên kết cho nhiều hàm, có thể sử dụng cấu trúc như sau:

```
Exten " language" function-prototype:

extern" language"

{

function prototype;

}
```

All linkage specifications must be global: they cannot be used inside a function.

Mọi sự " chỉ rõ liên kết " phải là toàn cục, chúng không thể sử dụng bên trong thân hàm.

The most common use of linkage specifications occurs when linking C++ programs to C code. By specifying "C" linkage you prevent the compiler from mangling(also known as decorating) the names of functions with embedded type information. Because of C++'s ability to overload functions and create member functions, the link-name of a function usually has type information added to it. Since C does not support overloading or member functions. It cannot recognize a mangled name. Using "C" linkage avoids this problem.

Việc sử dụng kỹ thuật " chỉ rõ sự liên kết" thường xảy ra phổ biến khi liên kết chương trình C++ với mã lệnh C. Bằng cách chỉ rõ liên kết "C" bạn có thể ngăn cản trình biên dịch khỏi sự sai lạc (trang hoàng) tên của hàm với các loại thông tin được nhúng vào. Bởi vì C++ cho phép nạp chồng các hàm và tạo các hàm thành viên, tên liên kết của hàm thường là loại thông tin được thêm vào nó. Trong khi đó, C không hỗ trợ nạp chồng hay là hàm thành viên. Nó không thể nhận diện một tên đọc sai. Sử dụng liên kết "C" sẽ tránh được điều này.

Although it is generally possible to link assembly language routines with a C++ program, there is often an easier way to use assembly language. C++ supports the

special keyword asm, which allows you to embed assembly language instructions within c C++ function. These assembler is that your entire program is completely defined as a C++ program and there is no need to link separate assembly language files. The general form of the asm keyword is shown here:

Mặc dù thường có thể liên kết "thường trình" hợp ngữ với chương trình C++, Nhưng cũng có một cách dễ dàng hơn để sử dụng hợp ngữ. C++ cung cấp từ khóa đặc biệt **asm,** cho phép bạn nhúng cấu trúc lệnh hợp ngữ vào trong hàm C++. Những chương trình dịch hợp ngữ này nằm trong chương trình của bạn và được định nghĩa hoàn toàn như là một chương trình C++ và cũng không cần có liên kết chia sẻ với các tập tin hợp ngữ. Hình thức phổ biến của từ khóa **asm** được thể hiện dưới đây:

Asm("op_code");

Where op-code is the assembly language instruction that will be embedded in your program.

It's important to note that several compilers accept these three slightly different forms of the asm statement.

Nơi nào mà mã thao tác là cấu trúc lệnh hợp ngữ thì sẽ được nhúng vào trong chương trình của bạn.

Một chú ý quan trọng là một vài trình biên dịch chấp nhận 3 hình thức khác nhau (không đáng kể) của cấu trúc lệnh **asm.**

```
Asm op-code;

Asm op-code newline

Asm

{

     Instruction sequence;

}
```

Here op-code is not enclosed in double quotes. Because embedded assembly language instructions tends to be implementation dependent. You will want to read your compiler's user manual on this issue.

***NOTE**: Microsoft Visual C++ uses_ _ asm for embedding assembly code. It is otherwise similar to asm.*

Đây là mã lệnh thao tác không được kèm theo trong nháy kép. Bởi vì lệnh hợp ngữ được nhúng vào thi hành một cách độc lập. Bạn sẽ muốn đọc trình biên dịch của bạn biên dịch như thế nào?

CHÚ Ý: Mircosoft Visual C++ sử dụng __asm để nhúng mã hợp ngữ. Nó thì khác với **asm.**

EXAMPLE:

1. This program links func() as a C, rather than a C++, function:

Chương trình này liên kết hàm **func()** như là một hàm của C, hơn là của C++, Hàm như sau:

```
// Illustrate linkage specifier.

#include <iostream>

using namespace std;


extern "C" int func(int x); // link as C function


// This function now links as a C function.

int func(int x)

{

  return x/3;

}
```

This function can now be linked with code compiled by a C compiler.

Hàm này có thể liên kết với mã lệnh biên dịch bởi trình biên dịch C.

2. The following fragment tells the compiler that f1(), f2(), and f3() should be linked as C functions:

Sự sắp xếp bên dưới chỉ cho trình biên dịch biết rằng f1(), f2(), và hàm f3() có thể liên kết như là hàm C:

```
extern "C" {
```

```
  void f1();
  int f2(int x);
  double f3(double x, int *p);
}
```

3. This fragment embeds several assembly language instructions into func():

Những đoạn này được nhúng một vài lệnh hợp ngữ vào hàm **func():**

```
void func()
{
  asm("mov bp, sp");
  asm("push ax");
  asm("mov cl, 4");
  // ...
}
```

Remember:
You must be an accomplished assembly language programmer in order to successfully use in-line assembly language. Also, be sure to check your compiler's user manual for details regarding assembly language usage.

Nhớ rằng:

Bạn phải là một người lập trình hợp ngữ tốt để sử dụng thành công hợp ngữ nội hàm. Bạn cũng phải chắc chắn việc kiểm tra người sử dụng trình biên dịch cảu bạn nắm vững về cách sử dụng hợp ngữ.

EXERCISE:

On your own, study the sections in your compiler's user manual that refer to linkage specifications and assembly language interfacing.

1. Theo bạn, học phần nào trong sách hướng dẫn sử dụng trình biên dịch của màn mà giới thiệu đến kỹ thuật " chỉ rõ liên kết " và giao diện hợp ngữ.

13.7. ARRAY-BASE I/O – MẢNG – CƠ SỞ NHẬP/XUẤT

In addition to console and file I/O, C++ supports a full set of functions that use character arrays as the input or output devide. Although C++'s array-base I/O parallels, in concept, the array-base I/O found in C++ (specifically, C's sscanf() and sprintf() functions), C++'s array-base I/O is more flexible and useful because it allows user defined types to be integrated into it. Although it is not possible to cover every aspect of array-base I/O here, the most important and commonly used features are examined.

Thêm vào cách giải quyết và tập tin I/O , C++ cung cấp đầy đủ các hàm để sử dụng cho mảng ký tự để nhập xuất từ thiết bị. Mặc dù C++ có các mảng cơ sở song song, tương tự, mảng cơ sở I/O cũng được tìm thấy trong C++(đặc biệt là các hàm C : sscanf(), sprintf). Mảng cơ sở trong C++ uyển chuyển hơn và tiện lợi hơn bởi vì nó cho phép người dùng định nghĩa loại được tích hợp vào nó. Mặc dù không thể bao gồm hết các khía cạnh của mảng cơ sở I/O, Những đặc điểm quan trong nhất và phổ biến nhất sẽ được nghiên cứu dưới đây.

It is important to understand from the outset that array-based I/O still operates through streams. Everything you learned about C++ I/O in Chapters 8 and 9 is applicable to array-based I/O. In fact, you need to learn to use just a few new functions to take full advantage of array-based I/O . These functions link a stream to a region of memory. Once this has been accomplished, all I/O takes place through the I/O functions you know already.

Việc hiểu mảng cơ sở I/O vẫn thao tác trên dòng thì rất quan trọng. Mọi thứ bạn học được ở chương 8 và 9 đều hữu dụng cho mảng cơ sởI/O. Sự thật, bạn can phải học thêm cách sử dụng một vài hàm mới để sử dụng hết những điểm thuận lợi của mảng cơ sở I/O. Những hàm này liên kết " dòng thực thi" với một vùng nhớ. Một trong những hàm đó đã được học, và mọi I/O xảy ra thông qua các hàm I/O mà bạn đã biết.

Before you can use array-based I/O , you must be sure to include the header <strstream> in your file. IN this header are defined the classes istrstream, ostrstream, and strstream. These classes create array-based input, output, and input/output streams, respectively. These classes have as a base ios, so all the functions and manipulators istrstream, opstrstream, and strstream.

To use a character array for output, use this general form of the ostrstream

constructor:

Trước khi bạn có thể sử dụng mảng cơ sở I/O, bạn phải chắc chắn là đã kèm file <strstream> vào trong tập tin của bạn. Trong phần đầu trang này được định nghĩa các lớp istrstream, ostrstream, và strstream. Những lớp này tạo ra mảng cơ sở để nhập xuất, và tách biệt nhập và xuất trên " dòng thực thi". Những lớp này là một cơ sở ios, mà mọi hàm và thao tác trên isstrstream, opstrstream, và strstream.

Để sử dụng mảng ký tự để nhập, dùng hình thức chung của phương thức khởi tạo ostrstream như sau:

```
ostrstream ostr( char * buf, streamsize size, openmode mode = ios::out);
```

Here ostr will be the stream associated with the array buf. The size of the array is specified by size. Generally, mode is simply defaulted to output, but you can see any output mode flag defined by ios if you like. (Refer to Chapter 9 for details.)

*H*àm **ostr** sẽ là một dòng thực thi kết hợp với mảng đệm. Kích thước của mảng được chỉ rõ bởi **size.** Nói chung, trạng thái mặc định đơn giản là xuất, nhưng bạn có thể thấy bất kỳ cờ trạng thái xuất được định nghĩa bởi ios nếu bạn muốn. (xem chi tiết chương 9).

One an array has been opened for output, characters will be put into the array until it is full. The array will not be overrun. Any array attempt to overfill the array will result in an I/O error. To find out how many characters have been written to the array, use the pcount() member function, shown here:

Một mảng đang được mở để nhập, một ký tự sẽ được đạt vào mảng cho đến khi mảng đầy. Mảng sẽ không tràn. Bất kỳ mảng nào cố gắng điền vào mảng sẽ có kết quả lỗi I/O. Để tìm xem có bao nhiêu ký tự được ghi vào mảng, dùng hàm thành viên **pcount()**:

streamsize pcount();

You must call the function in conjunction with a stream, and it will return the number of characters written to the array, including any null terminator.

To open an array for input, use this form of the isstrstream constructor:

Bạn phải gọi hàm chung trong dòng thực thi, và nó sẽ trả về số lượng ký tự được nhập vào mảng, bao gồm cả ký tự ngắt NULL.

Để mở một mảng để nhập, sử dụng hình thức thiết lập isstrstream như sau:

Isstrstream istr(const char buf);*

Here buf is a pointer to the array that will be used for input. The input stream will be

called istr. When input is being read from an array, eof() will return true when the end of the array has been reached.

Phần buf là một con trỏ vào mảng dùng để nhập. Dòng thực thi nhập sẽ được gọi istr. Khi mà sự kiện nhập diễn ra từ mảng, hàm eof() sẽ trả về true khi đến cuối mảng

To open an array for input/output operations, use this form of the strstream constructor:

> *Here is iostr will be an input/output stream that uses the array pointed to by buf, which is size characters long.*
>
> *The character-based streams are also referred to as char* streams in some C++ literature.*

It is important to remember that all I/O functions described earlier operate with array-based I/O, including the binary I/O functions and the random-access functions.

Để mở một mảng để cho thao tác nhập xuất, sử dụng hình thức thiết lập của strstream như sau:

> Iostr sẽ là dòng nhập xuất dùng mảng được trỏ đến bởi buf, có độ dài là size.
>
> Dòng thực thi ký tự cở sở cũng được quy vào như là char* trong một vài tài liệu C++.

Nhớ rằng tất cả các hàm I/O đều mô tả thao tác dễ dành hơn với mảng cơ sở I/O, bao gồm cả I/O nhị phân và hàm truy xuất ngẫu nhiên.

*strtream iostr(char * buf, streamzie size, openmode mode ios::in||ios::out);*

***Note :** the character-based stream classes have been deprecated by Standard C++. This means that they are still valid, but future versions of the C++ language might not support them. They are included in the book because they are still widely used. However, for new code you will probably want to use one of the containers described in Chapter 14.*

Chú ý: lớp dòng thực thi ký tự cơ sở bị phản đối bởi C++ chuẩn. Điều đó có nghĩa là chúng vẫn có giá trị, tuy nhiên trong nhưng phiên bản mới hơn của ngôn ngữ C++ sẽ không hỗ trợ chúng nữa. Chúng có trong cuốn sách này bởi vì chúng vẫn được dùng phổ biến. Tuy nhiên, Một mã lệnh mớ mà bạn có lẽ muốn dùng được mô tả trong chương 14.

EXAMPLES:

Hers is a short example that shows how to open an array for output and write data to it:

Đây là một ví dụ ngắn để chỉ ra làm cách nào để mở một mảng để xuất và ghi dữ liệu vào nó.

```
#include <iostream>

#include <strstream>

using namespace std;


int main()

{

     char buf[255]; // output buffer


     ostrstream ostr(buf, sizeof buf); // open output
array


     ostr << "Array-based I/O uses streams just like ";

     ostr << "'normal' I/O\n" << 100;

     ostr << ' ' << 123.23 << '\n';


     // you can use manipulators, too

     ostr << hex << 100 << ' ';

     // or format flags

     ostr.setf(ios::scientific);

     ostr << dec << 123.23;

     ostr << endl << ends; // ensure that buffer is
     null-terminated
```

```
        // show resultant string
        cout << buf;

        return 0;
}
```

Here is an example of array-based input:
This program reads and then redisplays the values contained in the input array ***buf.***

Đây là một ví dụ của mảng cơ sở nhập:

Chương trinh đọc và sau đó thể hiện giá trị chứ trong mảng nhập buf

```
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
        char buf[] = "Hello 100 123.125 a";

        istrstream istr(buf); // open input array

        int i;
        char str[80];
        float f;
        char c;
        istr >> str >> i >> f >> c;
        cout << str << ' ' << i << ' ' << f;
```

```
        cout << ' ' << c << '\n';
        return 0;
}
```

Keep in mind that an input array, once linked to a stream, will appear the same as a file. For example, this program uses get() and the eof() function to read the contents of buf.

Nhớ rằng một mảng nhập, một liên kết với dòng thực thi, sẽ xuất hiện trong cùng một file. Ví dụ như, chương trinh này sử dụng hàm get() và eof() để đọc nội dung trong buf.

```
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
        char buf[] = "Hello 100 123.125 a";
        istrstream istr(buf);
        char c;
        while(!istr.eof())
        {
                istr.get(c);
                if(!istr.eof()) cout << c;
        }

  return 0;
}
```

Chương trình này biểu diễn nhập xuất trên một mảng:

Chương trình đầu tiên sẽ ghi dữ liệu xuất vào iobuf, Sau đó đọc nó trở lại. Nó đọc đầu tiên toàn bộ dòng " this is a test " sử dụng hàm getline(). Sau đó đôc số thập phân có giá trị là 100 và số thập lục phân có giá trị 0x64.

```
#include <iostream>

#include <strstream>

using namespace std;


int main()

{

	char iobuf[255];

	strstream iostr(iobuf, sizeof iobuf);

	iostr << "This is a test\n";

	iostr << 100 << hex << ' ' << 100 << ends;

	char str[80];

	int i;

	str.getline(str, 79); // read string up to \n

	iostr >> dec >> i; // read 100

	cout << str << ' ' << i << ' ';

	iostr >> hex >> i;

	cout <<  hex << i;

	return 0;

}
```

EXERCISES:

Sửa đổi vì dụ 1 để nó thể hiện số ký tự được ghi vào buf cho đến khi gặp ngắt.

Write an application that uses array-based I/O to copy the contents of one array to another. (This is, of course, not the most efficient way to accomplish this task.)

Viết một ứng dụng mà sử dụng mảng cơ sở I/O để sao chép nội dung của một mảng khác. (Dĩ nhiên, đây không phải là cách tốt nhất cho bài tập này)

Using array-based I/O , write a program that converts a string that contains a floating-point value into its internal representation.

Sử dụng mảng cơ sở I/O, viết chương trình chuyển đổi chuỗi số thực thành giá trị

Skills check

Mastery Skills Check

At this pont you should be able to perform the following exercises and answer the questions.

Tại thời điểm này bạn có thể làm các bài tập dưới đây và trả lời các câu hỏi.

What makes a static member variable different from other member variables?

Thành phần tĩnh có gì khác so với các thành phần khác?

What header must be included in your program when you use array-based I/O ?

Khi sử dụng mảng cơ sở I/O, header trong chương trình của bạn phải bao gồm cái gì ?

Aside from the fact that array-based I/O uses memory as an input and/or output device, is there any difference between it and " normal" I/O in C++?

Bên cạnh sự thật là mảng cơ sở i/O sử dụng bộ nhớ để nhập và/ hay xuất, có những điểm nào khác nhau giữa nó và "bình thường " I/O trong C++ ?

Given a function called counter(), show the statement that causes the compiler to compile this function for C language linkage.

Cho một chương trình gọi hàm **counter(),** chỉ ra cấu lệnh nào là nguyên nhân mà trình biên dịch dịch hàm này ra ngôn ngữ C.

What does a conversion function do?

Hàm chuyển đổi làm cái gì?

Explain the purpose of explicit.

Giải thích mục đích của thành phần explicit.

What is the principal restriction placed on a const member function?

Hạn chế chủ yếu của hàm thành viên hằng là gì?

Explain namespace

Giải thích namespace.

What does mutable do?

Mutbale là cái gì ?

Cumulative Skills check

This section checks how well you have integrated material in this chapter with that from the preceding chapters.

Trong phần này kiểm tra xem bạn đã kết hợp kiến thức của chương này với các chương trước đó như thế nào.

Since a constructor that requires only one argument, provides an automatic conversion of the type of that argument to its class type, is there a situation in which this feature eliminates the need to create an overloaded assignment operators?

Từ khi phương thức thiết lập chỉ cần một tham số, cung cấp một hàm chuyển đổi tự đông của loại của lệnh thuộc loại lớp của nó. Có cần một vị trí….

Can a const_cast be used within a const member function to allow it to modify its invoking object ?

Const_cat có thể được sử dụng trong hàm thành viên hằng để cho phép nó thay đổi yêu cầu của đối tượng không ?

Thought question : since the original C++ library was contained in the global namespace and all old C++ programs have already dealt with this fact, What is the advantage to moving the library into the std namespace " after the fact," so to speak ?

Từ câu hỏi : khi mà thư viện gốc C++ được chứ trong biến namespave và tất cả các chương trình C++ cũ đã thỏa thuận với điều này. Điều thuận lợi của việc di chuyển thư viện vào std namespace sau sự thật đó"giải thích?

Look back at the examples in the first twelve chapters. Think about what member functions can be made const or static. Are there examples in defining a namespace would be appropriate ?

Nhìn lại những ví dụ trong phần đầu chương 12. Hàm thành viên có thể là hằng hoặc là tĩnh. Có nhưng ví dụ nào trong phần nhận dạng namespace cũng thể hiện điều này ?

CHAPTER 14

INTRODUCING THE STANDARD TEMPLATE LIBRARY – GIỚI THIỆU VỀ THƯ VIỆN GIAO DIỆN CHUẨN

CHAPTER OBJECTIVES

14.1 AN OVERVIEW OF THE STANDARD TEMPLATE LIBRARY

Tổng quan về thư viện giao diện chuẩn

14.2 THE CONTAINER CLASSES - Các lớp chứa đựng

14.3 VECTORS

14.4 LISTS

14.5 MAPS

14.6 ALGORITHMS - Các thuật toán

14.7 THE STRING CLASS - Lớp chuỗi

CONGRATULATIONS if you have worked your way through the preceding chapters of this book, you can definitely call yourself a C++ programmer. In this, the final chapter of the book, we will explore of C++'s most exciting- and most advanced- features : the Standard Template Library.

Chúc mừng bạn nếu bạn làm việc theo cách của bạn trong những chương trước của cuốn sách, bạn có thể tự gọi mình là một lập trình viên C++. Trong chương cuối của cuốn sách, chúng ta sẽ khám phá những điểm thú vị nhất và tiện ích nhật : Thư viên Template chuẩn.

The instruction of the Standard Template Library, or STL, was one of the major efforts that took place during the standardization of C++. The STL was not a part of the original specification for C++ but was added during the standardization proceedings.

The STL provides general purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures. For example, it includes support for vectors, lists,queues, and stacks. It also defines various routines that access them. Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.

Cấu trúc của thư viên Template chuẩn, hay STL, là một trong những nỗ lực lớn nhất trong việc chuẩn hóa C++. STL không là một phần của sự phân loại gốc cho C++ tuy nhiên nó được thêm vào trong quá trình chuẩn hóa. STL cung cấp những mục đích chung, các lớp mẫu và hàm thực thi rất phổ biến và thường dùng trong tính toán và cấu trúc dữ liệu. Ví dụ như, STL hỗ trợ lớp vectors, lists, queues, và stack. Nó cũng định nghĩa những đĩnh trình khác nhau để thực thi chúng. Bởi vì STL được xây dựng từ lớp mẫu, nên các thuật toán và cấu trúc dữ liệu có thể được cung cấp gần giống với các loại dữ liệu khác.

It must be stated at the outset that the STL is a complex piece of software engineering that uses some of C++'s most sophisticated features. To understand and use the STL you must be comfortable with all of the material in the preceding chapters. Specifically, you must feel at home with templates. The template syntax that describes the STL can seem quite intimidating- although it looks more complicated than it actually is. While there is nothing in this chapter that is any more difficult than the material in the rest of this book, don't be surprised or dismayed if you find the STL confusing at first. Just be patient, study the examples, and don't let the unfamiliar syntax distract you from the STL's basic simplicity.

Nó phải ở trạng thái bắt đầu mà STL thì là một phần phức tạp của kỹ sư phần mềm, họ sử dụng một vài đặc điểm phức tạp của C++. Để hiểu và sử dụng STL bạn phải nắm rõ tất cả các kiến thức trong các chương trước. Đặc biệt bạn phải có hứng thú với templates. Cấu trúc của template được mô tả STL khá phức tạp- mặc dù nó phức tạp hơn là thực sự như thế. Trong khi không có gì trong chương này mà có thể khó hơn những kiến thức trong chương cuối này, đừng ngạc nhiên và mất tinh thần nếu như bạn bị STL từ chối ngày từ đầu. Hãy kiên nhẫn, nghiên cứu ví dụ, và đừng vì những cấu trúc không bình thương làm bạn xao lãng tính cơ bản của STL.

The STL is a large library, and not all of its features can be described in this chapter. In fact, a full description of the STL and all of its features, nuances, and programming techniques to familiarize you with its basic operation, design philosophy, and programming fundamentals. After working through this chapter, you will be able to easily explore the remainder of the STL on your own.

This chapter also describes one of C++'s most important new classes: ***string****. The* ***string*** *class defines a string data type that allows you to work with character strings much as you do with other data types, using operators.*

STL là một thư viện lớn, và không thể mô tả hết tất cả các đặc điểm của nó trong chương này. Sự thật. một mô tả đầy đủ của STL và tất cả các đặc điểm của nó, sắc thài và kỹ thuật lập trình giúp cho bạn hiểu rõ với các thao tác cơ bản, lý luận thiết kế, và các chương trình cơ bản. Sau khi học xong chương này, bạn sẽ có thể dễ dàng khám phá những phần còn lại của STL bằng cách của bạn.

Chương này cũng mô tả một trong những lớp mới quan trọng nhất của C++: string. Lớp chuỗi định nghĩa một kiểu dữ liệu chuỗi cho phép bạn làm việc với các chuỗi ký tự như làm việc với các kiểu dữ liệu khác, sử dụng các toán tử.

Before proceeding, you should be able to correctly answer the following questions and do the exercises.

Trước khi thực thi, bạn có thể trả lời đúng những câu hỏi dưới đây và làm những bài tập sau.

*Explain why **namespace** was added to C++.*

Giải thích tại sao **namespace** được thêm vào C++.

*How do you specify a **const** member function ?*

Làm thế nào mà bạn chỉ rõ các hàm thành viên hằng ?

*The **mutable** modifier allows a library function to be changed by the user of your program. True or false?*

Sửa đổi **mutable** cho phép hàm thư viện thay đổi bởi người sử dụng chương trình của bạn. Đúng hay sai ?

Given this class:

```
class X

{
          int a,b;

public:

          X(int I, int j)( a= I; b=j;}

          // Create conversion to int here

};
```

*Create an integer conversion function that returns the sum of **a and b.***

Cho lớp sau:

```
class X
```

```
{
            int a,b;
public:
            X(int I, int j)( a= I; b=j;}
            // Create conversion to int here
};
```

Tạo một hàm chuyển đổi mà trả về tổng của a và b.

*A **static** member variable can be used before an object of its class exists. True or false?*

Một biến thành phần tĩnh có thể dùng trước khi đối tượng của lớp của nó tồn tại. Đúng hay sai ?

Given this class

Cho lớp:

```
class Demo
{
            Int a;
      public:
            Explicit demo(int i) { a=I;}
            Int geta() { return a;}
};
```

Is the following declaration legel?

Mô tả sau đây đúng quy cách hay không?

Demo o =10;

14.1. AN OVERVIEW OF THE STANDARD TEMPLATE LIBRARY – TỔNG QUAN VỀ THƯ VIỆN GIAO DIỆN CHUẨN

Although the Standard Templates Library is large and its syntax is, at times, rather intimidating, it is actually quite easy to use once you understand how it is constructed and what elements it employs. Therefore, before looking at any code examples, an overview of the STL is warranted

Mặc dù thư viện chuẩn Templates và cấu trúc của nó thì lớn, tại thời điểm này thì khá là kinh sợ, nó thực sự là khá dễ dàng để sử dụng một lần bạn sẽ hiểu được nó được cấu tạo như thế nào và các thành phần của nó làm việc thế nào. Do vậy, trước khi xem xét bất kì một đoạn mã nào, cần phải có 1 cái nhìn tổng quát về STL.

At the core of the Standard Templates Library are three foundational items: containers, algorithms, and iterators . These items work in conjunction with one another to provide off the shelf solutions to a variety of programming problems.

Cốt lõi của thư viện chuẩn Templates là 3 thành phần chính : phần chứa, thuật toán, và biến lặp. Những thành phần này làm việc chung với các thành phần khác để cung cấp các giải pháp cho các vấn đề khác nhau của chương trình.

*Containers are objects that hold other objects. There are several dynamic array, **queue** creates a queue, and **list** also defines associative containers, which allow efficient retrieval of values base on keys. For example, the **map** class defines a map that provides access to values with unique keys. Thus, a map stores a key/value pair and allows a value to be retrieved when its key is given.*

Phần chứa là một đối tượng chứa các đối tượng khác. Có một vài mảng động, **queue** tạo một hàng đợi, **và list** cũng nhận dạng phần chứa mẫu, cho phép có khả năng phục hồi giá trị trên cơ sở từ khóa. Ví dụ, lớp **map** định nghĩa một ánh xạ mà cung cấp cách truy cập giá trị với những từ khóa duy nhất. Vì vậy, một ánh xạ chứ một cặp từ khóa/ giá trị và co phép giá trị phục hồi khi mà nó nhận được từ khóa.

Each container class defines a set of functions that can be applied to the container. For example, a list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

Mỗi lớp phần chứa định nghĩa một hàm thiết lập mà có thể cung cấp cho phần chứa. Ví dụ, một phần chứa danh sách bao gồm những hàm co thể chèn, xòa và trộn các thành phần. Một stack bao gồm hàm push và pop các giá trị.

Algorithms act on containers. Some of the services algorithms perform are initializing, sorting, searching, and transforming the contents of containers. Many algorithms operate on a sequence, which is a linear list of elements within a container.

Thực hiện thuật toán trên phần chứa. Một vài thao tác thuật toán phục vụ được khởi tạo, sắp xếp, tìm kiếm và thay đổi nội dung của phần chứa. Có nhiều thuật toán thao tác tuần tự, là một danh sách tuyến tính các thành phần trong một phần chứa.

Iterators are object that are, more or less, pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array. The five types iterators are described in the following table.

Biến lặp là một đối tượng nhiều hơn hay ít hơn con trỏ. Nó cung cấp cho chúng ta khả năng xử lý theo chu kì thông qua nội dung của phần chứa theo một cách giống như là bạn sử dụng con trỏ để tạo xử lý chu kỳ trong mảng. Có 5 loại biến lặp được mô tả trong bảng dưới.

Iterator	***Access Allowed***
Random access	*Stores and retrieves values. Elements can be accessed randomly* Chứa và nhận giá trị. Các thành phần có thể truy xuất ngẫu nhiên
Bidirectional	*Stores and retrieves values. Forward and backward moving* Chứa và nhận giá trị. Di chuyển tới trước và sau
Forward	*Stores and retrieves values. Forward moving only.* Chứ và nhận giá trị. Chỉ cho phép di chuyển tới.
Input	*Retrieves but does not store values. Forward moving only* Phục hồi nhưng không lưu trữ giá trị. Chí cho phép di chuyển tới.
Output	*Stores but does not retrieve values. Forward moving only.* Chứa nhưng không phục hồi giá trị. Chỉ cho phép di chuyểntới.

In general, an iterator that has greater access capabilities can be used in place of one that has lesser capabilities. For example, a forward iterator can be used in place of an input iterator.

Nói chung, một biến lặp có khả năng truy cập lớn có thể dùng trong trường hợp nhỏ hơn khả năng của nó. Ví dụ, một biến lặp tới có thể dùng để đặt một biến lặp nhập.

Iterators are handled just like pointers. You can increment and decrement using the ***iterator*** *type defined by the various containers.*

Những biến lặp được truy cập chỉ như là con trỏ. Bạn có thể tăng hoặc giảm sử dụng loại biến lặp được đinh nghĩa bằng các phần chứa khác nhau.

The STL also supports reverse iterators. Reverse iterators are either bidirectional or random-access iterators that move through a sequence in reverse direction. Thus, if a reverse iterator points to the end of a sequence, incrementing that iterator will cause it to point to one element before the end.

STL cũng cung cấp biến lặp đảo ngược. Biến lặp đảo ngược không là biến lặp hai chiều cũng như là biến lặp truy xuất ngẫu nhiên, nó được duy chuyển tuần tự theo trình tự đảo ngược. Vì vậy, nếu biến lặp đảo ngược trỏ đến cuối của thao tác, tăng giá trị của biến lặp đó sẽ là nguyên nhân làm nó chỉ đến một thành phần trước kết thúc.

When referring to the various iterator types in template descriptions, this book will be use the terms listed in the following table.

Khi thống kê các loại biến lặp khác nhau trong phần mô tả template, cuốn sách này dùng bang điền kiện dưới đây:

Term	***Integrator type***
Bilter	*Bidirectional iterator* *(* Biến lặp hai chiều
Forlter	*Forward iterator* Biến lặp tới
Inlter	*Input iterator* Biến lặp nhập
Outlter	*Output iterator* Biến lặp xuất
Randlter	*Random-access iterator* Biến lặp truy xuất ngẫu nhiên

In addition to containers, algorithms, and iterators, the STL relies upon several other standard components for support. Chief among these are allocators, predicates, and comparison functions.

Thêm vào phần chứa, thuật toán , và biến lặp, STL cung cấp thêm một vài thành phần chuẩn nữa. Chủ yếu là cấp phát, thuộc tính và hàm so sánh.

Each allocation has an allocator defined for it. Allocators manage memory allocation for containers. The default allocators is an object of class ***allocator, but you can define*** *your own allocations if you need them for specialize applications. For most uses, the default allocator is sufficient.*

Mỗi một vùng cấp phát có một định nghĩa cấp phát cho nó. Hàm cấp phát quản lý vùng nhớ cho phần chứ. Mặc định hàm cấp phát là một đối tượng của lớp **allocator,** nhưng bạn co thể định nghĩa nó bằng cách của bạn nếu như bạn cần trong những trường hợp đặc biệt.

Several of the algorithms and containers use a special type of function called a predicate. There are two variations of predicates: unary and binary. A unary predicate takes one argument, and a binary predicate has two arguments. These functions return true or false; the precise conditions that make them return true or false are defined by the programmer. In this chapter, when a unary predicate function is used. It will be notated with the type ***unpred.*** *In a binary predicate, the arguments are always in the order of first, second. For both binary and binary predicates, the arguments will contain values of the same type as the objects being stored by the container*

Một vài thuật toán và phần chứa dùng một loại hàm đăc biệt để gọi thuộc tính. Có 2 thuộc tính khác nhau : unary và binary. Thuộc tính unary chứ một đối số và thuộc tính binary chứa 2 đối số. Những hàm này trả về đúng hay sai; những điều kiện cho trước làm nó trả về giá trị đúng hay sai được định nghĩa bởi người lập trình. Trong chương này, khi mà hàm thuộc tính unary được sử dụng. Nó sẽ được ký hiệu với loại **unpred.** Trong thuộc tính binary, những đối được yêu cầu trước nhất và thứ hai. Cả 2 thuộc tính unary và binary, các đối số của nó chứa giá trị của cùng loại của đối tượng được chứa trong phần chứa.

Some algorithms and classes use a special type of binary predicate that compares two elements. Called a comparison function, this type of predicate returns true if its first argument is less than its second. Comparison functions will be notated by the type ***Comp.***

Một vài thuật toán và lớp dùng một loại thuộc tính binary đặc biệt đó là so sánh 2 thành

phần. Gọi hàm so sánh, một loại thuộc tính sẽ trả về giá trị dung nếu đối số 1 nhỏ hơn đối số thứ 2. Hàm so sánh được ký hiệu bởi **Comp.**

*In addition to the headers required by the various STL classes, the C++ standard library includes the< **utility> and <functional>** headers, which provide support for the STL. For example, **<utility>** contains the definition of the template class **pair,** which can hold a pair of values. We will make use of **pair** later in this chapter.*

Thêm vào phần đầu là yêu cầu của các lớp STL khác nhau, thư viện C++ chuẩn bao gồm <**utility> và <functional> headers,**hỗ trợ cho STL. Ví dụ **<utility>** chứ định nghĩa của lớp template **pair,** có thể quản lý được cặp giá trị, Chúng ta sẽ dùng **pair** sau trong chương này.

*The templates in **<functional>** help you construct objects that define **operator().** These are called function objects, and they can be used in place of function pointers in many places. There are several predefined function objects declared within < **functional>** . Some are shown in the following table.*

Templates trong <**functional>** giúp bạn xây dựng đối tượng mà định nghĩa thao tác. Chúng được gọi hàm đối tượng, và chúng có thể dùng trong hàm cont trỏ ở nhiều nơi. Một vài hàm đối tượng định nghĩa trước được mô tả trong <**functional>.** Một vài hàm được trình bày ở bảng dưới.

Plus	*Minus*	*Multiplies*	*Divides*	*Modulus*
Negate	*Equal_to*	*Not_qual_to*	*Greater*	*Greater_equal*
Less	*Less_equal*	*Logical_and*	*Logical_or*	*Logical_not*

*Perhaps the most widely used function object is **less,** which determines whether the value of one object is less than the value of another. Function objects can be used in place of actual function pointers in the STL algorithms described later. Using function objects rather than function pointers allows the STL to generate more efficient code. However, for the purpose of this chapter, function objects are not need and we won't be using them directly. Although function objects are not difficult, a detailed discussion of function objects is quite lengthy and is beyond the scope of this book. They are something that you will need, however, to get maximum efficiency from the STL.*

Có lẽ việc sử dụng hàm đối tượng phổ biến thì khá hiếm, quyết định giá trị của đối tượng của một đối tượng thì ít hơn là giá trị của đối tượng khác. Hàm đối tượng có thể dùng trong hàm con trỏ thực sự trong STL thuật toán sẽ được mô tả sau. Sử dụng hàm đối tượng thì tốt hơn hàm con trỏ cho phép STL tạo ra nhiều đoạn mã hiệu quả. Tuy nhiên, vì

mục đích của chương này, hàm đối tượng không cần và chúng ta sẽ không sử dụng chúng trực tiếp. Mặc dù hàm đối tượng không khó, nhưng mô tả chi tiết của hàm thì khá là dài và vượt quá phạm vi của cuốn sách.Tuy nhien chúng thì có một vài thứ cần thiết cho bạn để đạt được hiệu quả tối đa từ STL.

EXERCISES

As they relate to the STL, what are containers, algorithms, and iterators?

Liên quan đến STL, phần chứa, thuật toán, và biến lặp là gì?

What are the two type of predicate?

Cái gì thì có 2 loại thuộc tính ?

What are the five types of iterators?

Kể tên 5 loại biến lặp ?

THE CONTAINER CLASSES – LỚP CONTAINER

As explained, containers are the STL objects that actually store data. The containers defined by the STL are shown in Table 14-1. also shown are the headers you must include to use each container. The ***string*** *class, which manages character strings, is also a container, but it is discussed later in this chapter.*

Theo như nghĩa, những container là những đối tượng STL lưu trữ dữ liệu. Những container được định nghĩa bởi STL theo như trong bảng 14-1. Ban đầu bạn phải include để sử dụng từng container. Lớp **string**, dung để quản lí những kí tự trong chuỗi, cũng là một container, nhưng nó sẽ được đề cập đến sau trong chương này.

Container	***Description***	***Required header***
Bitset	*a set of bits*	*<bitset>*
deque	*a double-ended queue*	*<queue>*
list	*a linear list*	*<list>*
map	*Stores key/value pairs in which each key is associated with only one value*	*<map>*
multimap	*Stores key/value pairs in which one key can be associated with two or more values*	*<map >*
multiset	*A set in which each element is not necessarily unique*	*<set>*

priority_queue	*A priority queue*	*<queue>*
queue	*A queue*	*<queue>*
set	*A set in which each element is unique*	*<set>*
stack	*A stack*	*<stack>*
vector	*A dynamic array*	*<vector>*

Container	**Description**	**Required header**
Bitset	Một lệnh của bit	<bitset>
deque	Hàng đợi 2 đầu	<queue>
list	Danh sách tuyến tính	<list>
map	Lưu trữ cặp khóa/giá trị mà trong đó mỗi khóa chỉ được kết hợp với 1 giá trị.	<map>
multimap	Lưu trữ cặp khóa/giá trị mà trong đó một khóa có thể kết hợp với 2 hay nhiều hơn 2 giá trị.	<map >
multiset	Một lệnh mà trong đó mỗi bộ phận không nhất thiết phải đơn trị.	<set>
priority_queue	Một hàng đợi ưu tiên.	<queue>
queue	Một hàng đợi	<queue>
set	Một lệnh mà trong đó các thành phần của lệnh phải đơn trị.	<set>
stack	Một ngăn xếp.	<stack>
vector	Một mảng động	<vector>

*Because the names of the placeholder types in a template class declaration are arbitrary, the container classes declare **typedef**ed versions of these types. This makes the type names concrete. Some of the most common **typedef** names are shown in the following table.*

Bởi vì tên của loại giữ chỗ nằm trong một lớp chuẩn khai báo tùy ý, do đó lớp container khai báo được **typedef** thành những kiểu tương tự. Điều này làm cho kiểu đặt tên được cụ thể. Một vài tên **typedef** phổ biến được đưa ra trong bảng sau:

Typedef Name	***description***
size_type	*An integral type equivalent to **size_t***
reference	*A reference to an element*
const_reference	*A **const** reference t an element*
iterator	*An iterator*
const_iterator	*A **const** iterator*
reverse_iterator	*A reverse iterator*
const_reverse_iterator	*A **const** reverse iterator*

value_type	*The type of a value stored in a container*
allocator_type	*The type of the allocator*
key_type	*The type of a key*
key_compare	*The type of a function that compares two key*
value_compare	*The type of a function that compares two values*

Typedef Name	**Mô tả**
size_type	Một số nguyên tương đương **size_t**
reference	Một tham chiếu đến 1 phân tử
const_reference	Một tham trị đến một phần tử.
iterator	Một biến lặp
const_iterator	Một tham trị lặp
reverse_iterator	Một biến lặp ngược
const_reverse_iterator	Một tham trị lặp ngược
value_type	Kiểu dữ liệu được lưu trữ trong container
allocator_type	Kiểu của allocator.
key_type	Kiểu của khóa.
key_compare	Loại hàm so sánh 2 khóa
value_compare	Loại hàm so sánh 2 giá trị.

Although it is not possible to examine each container in this chapter, the next sections explore three representatives: ***vector, list,*** *and* ***map.*** *Once you understand how these containers work, you will have no trouble using the others.*

Mặc dù nó không thể không thể xem xét mỗi loại container trong chương này, nhưng ở phần sau sẽ nghiên cứu kĩ 3 đại diện: **vector, list,** và **map.** Một khi bạn hiểu được cách mà những container này làm việc, thì bạn sẽ không gặp khó khăn gì trong việc sử dụng những loại khác.

14.3.VECTORS

Perhaps the most general-purpose of the containers is the vector. The ***vector*** *class supports a dynamic array. This is an array that can grow as needed. As you know, in C++ the size of an array is fixed at compile time. Although this is by far the most efficient way to implement arrays, it is also the most restrictive, because the size of the array cannot be adjusted at run time to accommodate changing program conditions. A vector solves this problem by allocating memory as needed. Although a vector is dynamic, you can still use the standard array subscript notation to access its elements.*

Có lẽ mục đích chung của container là vector. Lớp **vector** hỗ trợ cho mảng động. Đây là một mảng mà có thể phát triển khi cần. Như bạn biết, trong C++ kích thước của một mảng là thì được chỉnh sữa tại thời điểm biên dịch. Mặc dù, đây là một cách tác động có hiệu quả nhất đến mảng hiện thời, nhưng nó cũng là mặt hạn chế nhất, bởi vì kích thuowcsc của mảng không thể thay đổi được khi chạy để cung cấp các điều kiện chuyển đổi chương trình. Một vector sẽ giải quyết vấn đề này bằng cách chỉ định vùng nhớ khi cần thiết. Mặc dù vector là mảng động nhưng bạn vẫn có thể sử dụng một chỉ số của mảng chuẩn để truy xuất đến từng phần tử của chúng.

The template specification for ***vector*** *is shown here:*

Template <class T, class Allocator = allocator <T>> class vector
*Here **T** is the type of the data being stored and **Allocator** specifies the allocator, which defaults to the standard allocator. **vector** has the following constructors:*

Những đặc trưng chuẩn của vector được trình bày dưới đây:

```
Template<class T, class Allocator = allocator <T>>
class vector
```

Ở đây T là kiểu dữ liệu được lưu trữ và **allocator** chỉ định biến tạm, ngầm định là biến tạm chuẩn. **vector** có một số hàm thiết lập dưới đây:

```
explicit vector(const Allocator &a= Allocator());
explicit vector(size_type num, const T&val=T()
                      , const Allocator &a = Allocator());
vector (const vector <T Allocator> &obj);
template <class Inlter> vector (Inlter start, lnlter end
                      , const Allocator &a=Allocator ());
```

*The first form constructors an empty vector. The second form constructors a vector that has **num** elements with the value **val.** The value of **val** can be allowed to default. The third form constructs a vector that contains the same elements as **ob.** The fourth form constructs a vector that contains the elements in the range specified by the iterators **start** and **end.***

Dạng đầu tiên tạo một vector rỗng. Dạng thứ 2 tạo một vector có **num** phần tử với giá trị là **val.** Giá trị của **val** có thể được phép mặc định. Dạng thứ 3 tạo nên một vector có những phần tử giống như **ob.** Dạng thứ 4 tạo một vector bao gồm những phần tử trong một dãy được chỉ định bởi biến lặp từ **start** đến **end.**

Any object that will be stored in a vector must define a default constructor. It must also define the <and == operations. Some compilers might require that other comparison operators be defined. (Because implementations vary, consult your compiler's documentation for precise information.) All of the built-in-types automatically satisfy thee requirements.

Bất cứ một đối tượng nào được lưu trữ trong vector đều phải định nghĩa hàm khởi tạo mặc định. Nó cũng phải định nghĩa toán tử < và ==. Một số chương trình biên dịch còn yêu cầu định nghĩa them một số toán tử so sánh khác.(Bởi vì sự thi hành thì đa dạng, và phục vụ cho việc tra cứu thông tin chính xác của trình biên dịch.) Tất cả các kiểu khởi tạo tự động đòi hỏi 3 yêu cầu:

Although the template syntax looks rather complex, there is nothing difficult about declaring a vector. Here are some examples:

vector<int> iv; //creates a zero-length int vector
vector<char> cv(5); //creates a 5-element char vector
vector<char> cv(5, 'x') ; //initializes a 5-element char vector
vector<int> in2(iv) ; //creates an int vector from an int vector

The following comparison operators are defined for vector :

==, <, <=, !=, >, >=

Mặc dù, hình thức trông có vẻ là khá phức tạp nhưng thực chất là không có gì khó khăn trong việc khai báo một vector. Đây là một vài ví dụ:

vector<int> iv; //Tạo một vector số nguyên có chiều d‡i l‡ 0
vector<char> cv(5); //Tạo một vector 5 kí tự
vector<char> cv(5, 'x') ; //Khởi tạo một vector 5 kí tự
vector<int> in2(iv) ; //Tạo một vector số nguyên từ một vector số nguyên

Những toán tử so sánh được định nghĩa cho vector bên dưới:

==, <, <=, !=, >, >=

*The subscripting operator [] is also defined for **vector.** This allows you to access the elements of a vector using standard array subscripting notation.*

*The member functions defined by the **vector** are shown in Table 14-2 (Again, it is important not to be put off by the syntax.) Some of the most important member functions are **size(), begin(), end(), push_back(), insert(),** and **erase()**. The **size()** function returns the current size of the vector. This function is quite useful because it allows you to determine the size of a vector at run time. Remember, vectors will increase in size as needed, so the size of a vector must be determined during execution, not during compilation..*

Kí hiệu toán tử [] cũng được định nghĩa cho vector. Phương thức này cho phép bạn truy xuất đến các phần tử của vector bằng cách sử dụng kí hiệu chỉ số mảng chuẩn.

Hàm thành phần được định nghĩa bởi **vector** được trình bày ở bảng 14-2. Một số hàm thành phần quan trọng là **size(), begin(), end(), push_back(), insert(),** và **erase()**. Hàm **size()** trả về kích thước hiện thời của vector. Hàm này thì khá hữu dụng bởi vì nó cho phép bạn xác định kích thước của một vector khi chạy chương trình. Nhớ rằng, những vector sẽ tăng kích thước khi cần, vì vậy kích thước của một vector phải được xác định trong suốt quá trình thực thi, chứ không phải trong quá trình biên soạn.

*The **begin()** function returns an iterator to the start of the vector. The **end()** function returns an iterator to the end of the vector. As explained, iterators are similar to pointers, and it is through the use of the **begin()** and **end()** functions that you obtain an iterator to the beginning and end of a vector.*

Hàm **begin()** trả về một biến lặp để bắt đầu của vector. Hàm **end()** trả về một biến lặp để kết thúc vector. Theo đúng nghĩa của nó, thì biến lặp giống như là con trỏ, và thông qua hàm **begin() , end()** bạn có thể lấy được biến lặp để bắt đầu và kết thúc của một vector.

template<class lnlter> *void assign(lnlter start, lnlter end);*	*Assigns the vector the sequence defined by start and end*
Template<class Size, class T) *Void assign(Size num, const T &val = T());*	*Assigns the vector num elements of value val*
Reference at(size_type l); *Const reference at(size_type l) const;*	*Returns a reference to an elements of value val.*
Referenc back(size_type l); *Const reference at(size_type l) const;*	*Returns a reference to the last element in the vector.*
Iterator begin(); *Const iterator begin() const;*	*Returns an iterator to the first the element in the vector.*
Size_type capacity() const;	*Returns the current capacity of the vector. This is the number of elements it can hold before it will need to allocate more*

	memory.
Void clear();	*Removes all elements from the vector*
Bool empty() const;	*Returns true if the invoking vector is empty and false otherwise.*
Iterator end(); *Const_iterator end() const*	*Returns an iterator to the end of the vector.*
iterator erase(iterator i);	*Removes the element pointed to by i. returns an iterator to the element after the one removed.*
Iterator erase(iterator start, iterator end);	*Removes the elements in the range start to end. Returns an iterator to the element after the last vector.*
Reference front(); *Const_reference front() const;*	*Returns a reference to the first element in the vector.*
Allocator_type get_allocator() const;	*Returns the vector's allocator.*
Iterator insert(iterator I, const T&val=T());	*Inserts val immediately before the element specified be I. an iterator to the element is returned.*
Void insert(iterator I, size_type num, const T& val);	*Inserts num copies of val immediately before the element specified be i.*
Template<class lnlter> *Void insert(iterator I, lnlter start, lnltr end);*	*inserts the sequence defined by start and end immediately before the element specified by i.*
Size_type max_size() const;	*Returns the maximum number of element that the vector can hold.*
Reference operator[](size_type i) const; *Const_reference operator[](size_type i) const;*	*Returns a reference to the element specified by i.*
Void pop_back();	*Removes the last element in the vector.*
Void push_back(cons T&val);	*Adds an element with the value specified be val to the end of the vector.*
Reverse_iterator rbegin(); *Const_reverse_iterator rbegin() const;*	*Returns a reverse iterator to the end of the vector.*
Reverse_iterator rend(); *Const_reverse_iterator rend() const;*	*Returns a reverse iterator to the start of the vector.*
Void reverse (size_type num);	*Sets the capacity of the vector so that it is equal to at least num.*
Void resize (size_type num, T val =T());	*Changes the size of the vector to that specified by num. if the vector must be lengthened , elements with the value specified by val are added to the end.*
Size_type size() const;	*Returns the number of elements currently in the vector.*
Vois swap(vector<T, Allocator>&ob)	*Exchanges the elements stored in the invoking vector with those on ob.*

Hàm thành phần	**Mô tả**
template<class lnlter> void assign(lnlter start, lnlter end);	Gán giá trị cho vector theo trình tự từ start đến end.
Template<class Size, class T) Void assign(Size num, const T &val = T());	Gán giá trị của val cho num phần tử của vector.
Reference at(size_type l); Const_reference at(size_type l) const;	Trả về một tham chiếu đến một phần tử được chỉ định bởi i.
Reference back(size_type l); Const_reference at(size_type l) const;	Trả về một tham chiếu đến phần tử cuối cùng của vector.
Iterator begin(); Const _iterator begin() const;	Trả về một biến lặp chỉ định phần tử đầu tiên của vector.
Size_type capacity() const;	Trả về dung lượng hiện thời của vector. Đây là số lượng các phần tử mà nó có thể chứa trước khi nó cần cấp phát thêm vùng nhớ.
Void clear();	Xóa tất cả các phần tử trong vector.
Bool empty() const;	Trả về true nếu vector rỗng và trả về false nếu ngược lại.
Iterator end(); Const_iterator end() const	Trả về một biến lặp để kết thúc một vector.
iterator erase(iterator i);	Xóa một phần tử được chỉ bởi i. Trả về một biến lặp chỉ đến phần tử sau phần tử được xóa.
Iterator erase(iterator start, iterator end);	Xóa những phần tử trong dãy từ start đến end. Trả về một biến lặp chỉ đến phần tử sau cùng của vector.
Reference front(); Const_reference front() const;	Trả về một tham chiếu đến phần tử đầu tiên của vector.
Allocator_type get_allocator() const;	Trả về vùng nhớ được cấp phát cho vector.
Iterator insert(iterator I, const T&val=T());	Chèn **val** trực tiếp vào trước thành phần được chỉ định bởi i. biến lặp chỉ đến phần tử được trả về.
Void insert(iterator I, size_type num, const T& val);	Chèn **num** val một cách trực tiếp trước phần tử được chỉ định bởi i.
Template<class lnlter> Void insert(iterator I, lnlter start, lnltr end);	Chèn chuỗi xác định từ start đến end trực tiếp trước một phần tử được chỉ định bởi i.
Size_type max_size() const;	Trả về số lượng phần tử lớn nhất mà vector có thể chứa.
Reference operator[](size_type i) const; Const_reference operator[](size_type i) const;	Trả về một tham chiếu đến phần tử được chỉ định bởi i.

Void pop_back();	Xóa phần tử cuối cùng trong vector.
Void push_back(cons T&val);	Thêm vào một phần tử có giá trị **val** vào cuối của vector.
Reverse_iterator rbegin(); Const_reverse_iterator rbegin() const;	Trả về biến lặp nghịch chỉ điểm kết thúc của vector.
Reverse_iterator rend(); Const_reverse_iterator rend() const;	Trả về một biến lặp ngịch chỉ điểm bắt đầu của vector.
Void reverse (size_type num);	Thiết lập kích thước của vector ít nhất là bằng **num.**
Void resize (size_type num, T val =T());	Chuyển đổi kích thước của vector được xác định bởi **num.** Nếu như kích thước của vector tăng lên thì các phần tử có giá trị **val** sẽ được thêm vào cuối vector.
Size_type size() const;	Trả về số lượng các phần tử hiện thời của trong vector.
Vois swap(vector<T, Allocator>&ob)	Chuyển đổi những phần tử được lưu trong vector hiện thời với những phần tử trong **ob.**

*The **push_back()** function puts a value onto the end of the vector. If necessary, the vector is increased in length to accommodate the new element. You can also add elements to the middle using **insert()**. A vector can also be initialized. In any event, once a vector contains elements, you can use array subscripting to access or modify those elements. You can remove elements from a vector using **erase().***

Hàm **push_back()** đưa một giá trị vào cuối vector. Nếu cần thiết thì vector tăng độ dài lên để cung cấp cho phần tử mới. Bạn cũng có thể thêm những phần tử vào giữa sử dụng **insert().** Một vector cũng có thể được khởi tạo. Trong bất kì công việc nào, một khi một vector chứa phần tử, thì bạn đều có thể sử dụng chỉ số mảng để truy xuất hoặc sữa đổi những phần tử đó. Bạn có thể xóa những phần tử ở trong vector bằng cách sử dụng hàm **erase().**

EXAMPLES:

Here is a short example that illustrates the basic operation of a vector.

Đây là một chương trình thể hiện những thao tác căn bản của một vector.

```
#include "stdafx.h"


// Vector basics.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
```

```
    vector<int> v; // create zero-length vector
    int i;

    // display original size of v
    cout << "Size = " << v.size() << endl;

    /* put values onto end of vector --
       vector will grow as needed */
    for(i=0; i<10; i++) v.push_back(i);

    // display current size of v
    cout << "Current contents:\n";
    cout << "Size now = " << v.size() << endl;

    // display contents of vector
    for(i=0; i<v.size(); i++)
         cout << v[i] << " ";
    cout << endl;

    /* put more values onto end of vector --
       again, vector will grow as needed */
    for(i=0; i<10; i++)
         v.push_back(i+10);

    // display current size of v
    cout << "Size now = " << v.size() << endl;

    // display contents of vector
    cout << "Current contents:\n";
    for(i=0; i<v.size(); i++)
         cout << v[i] << " ";
    cout << endl;

    // change contents of vector
    for(i=0; i<v.size(); i++) v[i] = v[i] + v[i];

    // display contents of vector
    cout << "Contents doubled:\n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;


    return 0;
}
```

*Let's look at this program carefully. In **main()**, an integer vector called **v** is created. Since no initialized is used, it is an empty vector with an initial capacity of zero. That is, it is a zero-length vector. The program confirms this by calling the **size()** member*

*function. Next ten elements are added to the end of **v** with the member function **push_back().** This causes **v** to grow in order to accommodate the new elements. As the output shows, its size after these additions is 10. next, the contents of **v** are displayed. Notice that the standard array subscripting notation is employed. Next, ten more elements are added and **v** is automatically increased in size to handle them. Finally, the values of **v**'s elements are altered using standard subscripting notation.*

Hãy xem xét kĩ chươn trình này. Trong hàm **main()**, tạo một vector các số nguyên được gọi là **v.** Kể từ khi nó không được khởi tạo, thì nó là một vector rỗng với số phần tử bằng 0. Đó có nghĩa là độ dài của vector cũng bằng 0. Chương trình xác nhận điều này bằng cách gọi hàm thành phần **size()**. Tiếp đến, 10 phần tử được thêm vào cuối vector **v** bằng hàm thành phần **push_back().** Điều này làm cho kích thước của tăng lên để cung cấp cho các phần tử mới. Khi xuất, kích thước của nó sau khi thêm là 10. Kế tiếp, nọi dung của **v** được xuất ra ngoài màn hình. Chú ý rằng chỉ số kiểu mảng chuẩn thì được sử dụng. Tiếp theo, thêm 10 phần tử nữa và **v** tự động tăng kích thước lên để xử lí chúng. Cuối cùng, những giá trị của các phần tử của **v** được sửa đổi bằng cách sử dụng chí số của mảng chuẩn.

*There is one other point of interest in this program. Notice that the loops that displays the contents if **v** use as their target v.**size().** One of the advantages that vectors have over arrays is that it is impossible to find the current size of a vector. As you can imagine, this is quite useful in a variety of situations.*

Có một con trỏ khác trong chương trình này. Chú ý rằng những vòng lặp xuất nội dung ra màn hình nếu như **v** sử dụng như là định vị của chúng **v.size()**. Một trong số những thuận tiện của vector hơn mảng là: mảng không thể biết được kích thước hiện thời của một vector. Khi bạn hình dung rằng, điều này thì khá hữu dụng trong nhiều tình huống.

As you know, arrays and pointers are tightly linked in C++. An array can be accessed either through subscripting or through a pointer. The parallel to this in the STL is the link between vectors and iterators. You cac access the members of a vector by using an iterator. The following example shows both of these approaches.

Như bạn biết, mảng và con trỏ có mối quan hệ chặt chẽ với nhau trong C++. Một mảng có thể được truy xuất thông qua chỉ số, hoặc một con trỏ. Sự tương đương này trong STL là mối quan hệ giữa biến lặp (chỉ số) và vector. Ban có thể truy xuất đến các thành phần của một vector bằng sử dụng một biến lặp. ví dụ sau đây sẽ thể hiện cả 2 phương pháp.

```
// Access a vector using an iterator.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v; // create zero-length vector
  int i;
```

```
  // put values into a vector
  for(i=0; i<10; i++) v.push_back(i);

  // can access vector contents using subscripting
  for(i=0; i<10; i++) cout << v[i] << " ";
  cout << endl;

  // access via iterator
  vector<int>::iterator p = v.begin();
  while(p != v.end())
  {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

The output from this program is:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

*In this program, the vector is initially created with zero length. The **push_back()** member function puts values onto the end of the vector, expanding its size as needed.*

Trong chương trình này, vector ban đầu được tạo có chiều dài 0. Hàm **push_back()** đưa các giá trị vào cuối vector, mở rộng kích thước của nó theo yêu cầu.

*Notice how the iterator **p** is declared. The type **iterator** is defined by the the container classes. Thus, to obtain an iterator for a particular container, you will use a declaration similar to that shown in the example: simply quality **iterator** with the name of the container. In the program, **p** is initialized to point to the start of the vector by using the **begin()** member function. This function returns an iterator to the start of the vector. This iterator can then be used to access the vector an element at a time by incrementing it as needed. This process is directly parallel to the way a pointer can be used to access the elements of an array. To determine when the end of the vector has been reached, the **end()** member function is employed. This function returns an iterator to the location that I some last element in the vector. Thus, when **p** equals **v.erase()**, the end of the vector has been reached.*

Chú ý cách khai báo con trỏ **p**. Kiểu **iterator** được định nghĩa bởi lớp container. Thay vì, một container thông thường chứa một biến lặp, bạn sẽ sử dụng một khai báo tương tự, nó được đưa ra trong ví dụ: đặc trưng cơ bản của **iterator** liên quan đến tên của contianer. Trong chương trình, **p** ban đầu chỉ đến vị trí ban đầu của vector nhờ hàm thành phần là **begin().** Hàm này trả về biến lặp chỉ đến vị trí đầu tiên của vector. Biến lặp có thể được sử dụng để truy xuất lần lượt từng phần tử của vector bằng cách tăng dần nó lên 1. Quá trình này thì tương tự như cách sử dụng một con trỏ để truy xuất các phần tử của một mảng. Để chứng minh điều này, khi kết thúc vector, thì sử dụng hàm **end().** Hàm này trả về một biến lặp chỉ vị trí phần tử cuối cùng trong vector. Thay vì dùng **p** tương tự như

v.earase(), thì tiến đến điểm cuối của vector.

3. *In addition to putting new values on the end of vector, you can insert elements into the middle using the **insert()** function. you can also remove elements using **erase()**. The following program demonstrates **insert()** and **erase()**.*

3. Thêm vào giá trị vào cuối vector, bạn có thể chèn những phần tử vào giữa bằng cách sử dụng hàm **insert()**. Bạn cũng có thể xóa sử dụng **erase().** Chương trình sau đây thể hiện hàm **insert()** và **erase().**

```
// Demonstrate insert and erase.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v(5, 1); // create 5-element vector of 1s
  int i;

  // display original contents of vector
  cout << "Size = " << v.size() << endl;
  cout << "Original contents:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl << endl;

  vector<int>::iterator p = v.begin();
  p += 2; // point to 3rd element

  // insert 10 elements with value 9
  v.insert(p, 10, 9);

  // display contents after insertion
  cout << "Size after insert = " << v.size() << endl;
  cout << "Contents after insert:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl << endl;

  // remove those elements
  p = v.begin();
  p += 2; // point to 3rd element
  v.erase(p, p+10); // remove next 10 elements

  // display contents after deletion
  cout << "Size after erase = " << v.size() << endl;
  cout << "Contents after erase:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;
```

```
  return 0;
   }
```

4. *Here is an example that uses a vector to store objects of a programmer-defined class. Notice that the class defines the default constructor and that overloaded versions of < and == are provided. Remember, depending upon how your compiler implements the STL, other comparison operators might need to be defined*

Đây là một ví dụ sử dụng một vector để lưu trữ đối tượng của class do người lập trình định nghĩa. Chú ý rằng class định nghĩa hàm thiết lập mặc định và cung cấp toán tử < và == được nạp chồng. Nhớ rằng, dựa trên chương trình biên dịch của bạn thực thi STL, mà các toán tử so sánh khác có thể cần được định nghĩa.

```
// Store a class object in a vector.
#include <iostream>
#include <vector>
using namespace std;

class Demo
{
     double d;
     public:
          Demo() { d = 0.0; }
          Demo(double x) { d = x; }

          Demo &operator=(double x)
          {
               d = x; return *this;
          }

           double getd() { return d; }
};

bool operator<(Demo a, Demo b)
{
  return a.getd() < b.getd();
}

bool operator==(Demo a, Demo b)
{
  return a.getd() == b.getd();
}

int main()
{
  vector<Demo> v;
  int i;
```

```
  for(i=0; i<10; i++)
    v.push_back(Demo(i/3.0));

  for(i=0; i<v.size(); i++)
    cout << v[i].getd() << " ";

  cout << endl;

  for(i=0; i<v.size(); i++)
    v[i] = v[i].getd() * 2.1;

  for(i=0; i<v.size(); i++)
    cout << v[i].getd() << " ";

  return 0;
}
```

The output from this program is shown here:

Kết quả của chương trình này là:

```
0  0.333333  0.666667 1 1.33333 1.66667 2 2.33333 2.66667 3
0  0.7  1.4  2.1  2.8  3.5  4.2  4.9  5.6  6.3
```

EXERCISES:

Try examples just shown, marking small modifications and observing their effects.
*in example 4, both a default (i.e, parameterless) and a parameterized constructor were defined for **Demo.** Can you explain why this is important?*
*Here is a simple **Coord** class. Write a program that stores objects of type **Coord** in a vector. (Hint : Remember to define the < and == operators relative to **Coord.**)*

Kiểm tra lại các ví dụ vừa mới được đưa ra, làm những sự thay đổi nhỏ và xem xét những ảnh hưởng của chúng.

Trong ví dụ 4, cả hàm khởi tạo mặc định và khởi tạo có tham số đều được định nghĩa cho **demo**. Bạn có thể giải thích tại sao có điều này?

Ở đây là lớp **Coord.** Viết một chương trình lưu trữ các đối tượng dạng **Coord** trong một vector. (Hướng dẫn: Định nghĩa các toán tử < và == liên quan đến **Coord.)**

```
class Coord
{
     public:
          int x, y;
          Coord() { x = y = 0; }
          Coord(int a, int b) { x = a; y = b; }
};
```

LISTS - DANH SÁCH

*The **list** class supports a bidirectional, linear list. Unlike a vector, which supports random access, a list can be accessed sequentially only.because, lists are bidirectional, they can be accessed front to back or back to front.*

Danh sách lớp hỗ trợ danh sách 2 chiều, tuyến tính. Không giống như vector, hỗ trợ truy xuất một cách ngẫu nhiên, một danh sách chỉ có thể được truy xuất một cách tuần tự. Bởi vì, danh sách thì 2 chiều, chúng có thể được truy xuất từ đầu đến cuối, hoặc ngược lại.

*The **list** class has this this template specification:*

Template<class T, class Allocator = allocator<T>> class list

*Here **T** is the type the type of data stored in the list. The allocator is specified be **Allocator**, which defaults to the standard allocator. This class has the following constructors:*

Một danh sách lớp có khai báo như sau:

Template <class T, class Allocator = Allocator<T>> class list.

Ở đây T là kiểu dữ liệu được lưu trữ trong danh sách. Allocator được chỉ định bởi **Allocator,** nó mặc định allocator chuẩn. Lớp này có cấu trúc như sau:

```
Explicit list (const Allocator &a= Allocator());
Explicit list (size_type num, const T&val=T(), const
Allocator &a= Allocator());
List (const list <T, Allocation> &ob);
Template<class lnlter> list (lnlter start, lnlter end,
const Allocator &a=Allocator());
```

*The first form constructs an empty list. The second form constructs a list that has **num** elements with the value **val**, which can be allowed to default. The third form constructs a list that contains the same elements as **ob**. The fourth form constructs a list that contains the elements in the range specified by the iterators **start** and **end.***
*The following comparison operators are defined for **list:***

==, <, <=, !=, >, >=

Dạng đầu tiên xây dựng nên một danh sách rỗng. Dạng thứ 2 xây dựng một danh sách có **num** phần tử có giá trị là **val,** nó có thể được mặc định. Dạng thứ 3 xây dựng một danh sách bao gồm những thành phần giống như **ob.** Dạng thứ 4 tạo nên một danh sách bao gồm những thành phần trong dãy được định nghĩa bởi biến lặp từ **start** đến **end.**
Những toán tử so sánh được định nghĩa cho danh sách:

==, <, <=, !=, >, >=

*The member functions defined for **list** are shown in Table 14-3. Like a vector a list can have elements put into it with the **push_back()** function. you can put elements on the front o the list by using **push_front(),** and you can insert an element into the middle of a list by using **insert()**. You can use **splice()** to join two lists, and you can merge one list into another by using **merge().***

Những hàm thành phần được định nghĩa cho **list** được trình bày trong bảng 14-3. Giống như một vector, một danh sách có những thành được đưa vào với hàm **push_back().** Bạn có thể chèn một phần tử vào giữa của một danh sách bằng hàm **insert()**. Bạn có thể sử dụng **splice()** để nhập 2 danh sách lại với nhau, và bạn có thể trộn danh sách này với một danh sách khác bằng sử dụng hàm **merge().**

Any data type that will be held in a list must define a default constructor. It must also define the various comparison operators. At the time of this writing, the precise requirements for an object that will be stored in a list vary from compiler to compiler and are subject to change, so you will need to check your compiler's documentation.

Bất kì loại dữ liệu nào được lưu trong danh sách đều phải định nghĩa hàm khởi tạo mặc định. Bạn cũng phải định nghĩa thêm những toán tử so sánh khác. Tại thời điểm này của bài viết, điều kiện cho một đối tượng sẽ được lưu trong một danh sách thay đổi từ chương trình biên dịch này sang chương trình biên dịch khác, và là một chủ thể để chuyển đổi, vì vậy bạn sẽ cần kiểm tra chương trình biên dịch của mình.

Member function	***Description***
Template<class lnIter> *Void assign(lnIter start, lnIter end);*	*Assigns the list the sequence defined by start and end.*
Template<class Size, class T> *Void assign (Size num, const T&val=T());*	*Assigns the list num elements of value.*
Reference back();	*Returns a reference to the last element in the list.*
Const_reference back()const; *Iterator begin();*	*Return an iterator to the first element in the list,.*
Void clear();	*Removes all elements from the list.*
Bool empty()const;	*Returns true if the invoking list is empty and false otherwise.*
Iterator end(); *Const_iterator end() const;*	*Returns an iterator to the end of the list.*
Iterator erase(iterator i);	*Removes the element pointed to by i. returns an iterator to the element after the one removed.*
Iterator erase(iterator start, iterator end);	*Removes the element in the range start to end. Returns an iterator to the element after the last element removed.*
Reference front(); *Const_referencefront() const;*	*Returns a reference to the first element in the list.*
Allocator_type get_allocator() const;	*Returns the list's allocator.*
Iterator insert(iterator i, const T &val=T());	*Inserts val immediately before the element specified by i. An iterator to the element is returned.*
Void insert(iterator I, size_type num, const T&val=T());	*Inserts* ***num*** *copies of* ***val*****immediately** before the element specified by i.
Template<class lnIter>	*Inserts the sequence defined by start and*

Void insert(iteratir i, lnlter start, lnlter end);	*end immediately before the element specified by i.*
Size_type max_size max_size() const;	*Returns the maximum number of elements that the list can hold.*
Void merge(list<T. allocator>&ob); *Template<class Comp>* *Void merge(<list<T, Allocator>&ob, Comp cmpfn);*	*Merges the ordered list contained in ob with the invoking ordered list. The result is ordered. After the merge, the list contained in ob is empty. In the second form, a comparison function can be specified to determine whether the value of one element is less than that of another.*
Void pop_back();	*Removes the last element in the list.*
Void pop_front();	*Removes the first element in the list.*
Void push_back(const T&val);	*Adds an element with the value specified by val to the end of the list.*
Void push_front(const T&val);	*Adds an element with the value specified by val to the front of the list.*
Reverse_iterator rbegin(); *Const_reverse_iterator rbegin() const;*	*Returns a reverse iterator to the end of the list.*
Void remove(const T&val);	*Removes elements with the value val from the list.*
Template <class UnPred> *Void remove_if(UnPred pr);*	*Removes elements for which the unary precidate pr is true.*
Reverse_iterator rend(); *Const_reverse_iterator rend() const;*	*Returns a reverse iterator to the start of the list.*
Void resize(isze_type num, T val=T());	*Changes the size of the list to that specified by num. if the list must be lengthened, elements with the value specified by val are added to the end.*
Void reverse();	*Reverses the invoking list.*
Size_type size() const;	*Returns the number of elements currently in the list.*
Void sort (); *Template <class Comp>* *Void sort(conm cmpfn);*	*Sorts the list. The second form sorts the list using the comparison function cmpfn to determine whether the value of one element is less than that of another.*
Void spilice(iterator i, list <T, allocator> &ob);	*Inserts the contents of ob into the invoking list at the location pointed to by i. after the operation ob is empty.*
Void splice(iterator i, list <allocator> &ob, iterator el);	*Removes the element pointed to by **el** from the list **ob** and stores it in the invoking list at the location pointed to by i.*
Void spice(iterator I, list<T, Allocator>&ob, iterator start, iterator end).	*Removes the range defined by **start** and **end** from **ob** and stores it in the invoking list beginning at the location pointed to by i.*

Void swap(list<T, Allocator> &ob);	*Exchanges the elements stored in the invoking list with those in ob.*
Void unique(); *Template<class BinPred>* *Void unique(BinPred pr);*	*Removes duplicate elements from the invoking list. The second form uses pr to determine uniqueness.*

Hàm thành viên	**Mô tả**
Template<class lnlter> Void assign(lnlter start, lnlter end);	Gán giá trị cho danh sách theo trình tự từ bắt đầu đến kết thúc.
Template<class Size, class T> Void assign (Size num, const T&val=T());	Gán *num* phần tử của danh sách bằng giá trị *value*.
Reference back();	Trả về một tham chiếu đến phần tử cuối cùng trong danh sách. (Chỉ đến phần tử cuối cùng của danh sách.)
Const_reference back()const; Iterator begin();	Trả về một biến lặp chỉ đến phần tử đầu tiên của danh sách. (Chỉ đến phần tử đầu tiên danh sách.)
Void clear();	Xóa tất cả các phần tử trong danh sách.
Bool empty()const;	Trả về *true* nếu danh sách rỗng, và *false* nếu ngược lại.
Iterator end(); Const_iterator end() const;	Trỏ đến cuối danh sách.
Iterator erase(iterator i);	Xóa phần tử được chỉ định bởi *i.* Chỉ đến phần tử kế sau phần tử đã được xóa.
Iterator erase(iterator start, iterator end);	Xóa những phần tử từ vị trí *start* đến *end.* Chỉ đến phần tử kế sau phần tử cuối cùng được xóa.
Reference front(); Const_referencefront() const;	Trả về một tham chiếu chỉ đến phần tử đầu tiên của danh sách.
Allocator_type get_allocator() const;	Trả về vùng nhớ được cấp phát cho danh sách.
Iterator insert(iterator i, const T &val=T());	Chèn giá trị *val* một cách trực tiếp vào trước phần tử được chỉ định bởi *i.* Trả về một biến lặp chỉ đến phần tử này.
Void insert(iterator I, size_type num, const T&val=T());	Chèn ***num*** giá trị ***val*** trực tiếp trước phần tử được chỉ định bởi ***i***.
Template<class lnlter> Void insert(iteratir i, lnlter start, lnlter end);	Chèn theo trình tự được xác định từ ***start*** đến ***end*** trực tiếp trước phần tử được chỉ định bởi ***i.***
Size_type max_size max_size() const;	Trả về số lượng các phần tử mà danh sách có thể giữ.
Void merge(list<T. allocator>&ob); Template<class Comp> Void merge(<list<T, Allocator>&ob, Comp cmpfn);	Trộn danh sách có thứ tự được chứa trong **ob** với một danh sách có thứ tự đang gọi thực hiện phương thức. Kết quả là một danh sách có thứ tự. Sau khi trộn, danh

	sách chứa trong **ob** là rỗng. Ở dạng thứ 2, một hàm so sánh được định nghĩa để xác định giá trị của một phần tử thì nhỏ hơn giá trị một phần tử khác.
Void pop_back();	Xóa phần tử cuối cùng trong danh sách.
Void pop_front();	Xóa phần tử đầu tiên trong danh sách.
Void push_back(const T&val);	Thêm một phần tử có giá trị được xác định bởi ***val*** vào cuối danh sách.
Void push_front(const T&val);	Thêm một phần tử có giá trị được xác định bởi ***val*** vào đầu danh sách.
Reverse_iterator rbegin(); Const_reverse_iterator rbegin() const;	Trả về một biến lặp ngược chỉ đến cuối danh sách.
Void remove(const T&val);	Xóa những phần tử có giá trị ***val*** trong danh sách.
Template <class UnPred> Void remove_if(UnPred pr);	Xóa những mà phần tử nếu như ***pr*** là true.
Reverse_iterator rend(); Const_reverse_iterator rend() const;	Trả về một biến lặp ngược chỉ đến đầu danh sách.
Void resize(isze_type num, T val=T());	Thay đổi kích thước của danh sách được xác định bởi ***num.*** Nếu như danh sách dài ra thêm, thì những phần tử có giá trị ***val*** được thêm vào cuối.
Void reverse();	Đảo ngược danh sách đang gọi thực hiện phương thức.
Size_type size() const;	Trả về số lượng các phần tử hiện tại trong mảng.
Void sort (); Template <class Comp> Void sort(conm cmpfn);	Sắp xếp danh sách. Dạng thứ 2 sử dụng hàm so sánh ***cmpfn*** để xác định giá trị của phần tử này có nhỏ hơn giá trị của phần tử kia không.
Void spilice(iterator i, list <T, allocator> &ob);	Chèn nội dung của ***ob*** vào trong danh sách đang gọi thực hiện phương thức tại vị trí được chỉ định bởi ***i.*** Sau thao tác này ***ob*** là rỗng.
Void splice(iterator i, list <allocator> &ob, iterator el);	Xóa phần tử được chỉ định bởi ***el*** trong danh sách ***ob*** và lưu trữ nó vào trong danh sách đang gọi thực hiện phương thức tại vị trí được chỉ định bởi ***i.***
Void splice(iterator I, list<T, Allocator>&ob, iterator start, iterator end).	Xóa một mảng phần tử bắt đầu từ vị trí ***start*** đến vị trí ***end*** và lưu nó vào trong danh sách đang gọi thực hiện phương thức bắt đầu từ vị trí ***i.***
Void swap(list<T, Allocator> &ob);	Hoán vị nội dung những phần tử được lưu trữ trong danh sách đang gọi thực hiện phương thức với những phần tử trong ***ob.***
Void unique();	Xóa những phần tử trùng trong danh sách

Template<class BinPred> Void unique(BinPred pr);	đang gọi thực hiện phương thức. Dạng 2 sử dụng ***pr*** để định nghĩa sự duy nhất.

EXAMPLES:

Here is a simple example of a list:

1. Ở đây là một ví dụ đơn giản của một danh sách:

```
// List basics.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst; // create an empty list
  int i;

  for(i=0; i<10; i++) lst.push_back('A'+i);

  cout << "Size = " << lst.size() << endl;

  list<char>::iterator p;

  cout << "Contents: ";
  while(!lst.empty())
  {
    p = lst.begin();
    cout << *p;
    lst.pop_front();
  }

  return 0;

}
```

The output produced by this program is shown here.

Kết quả được sản xuất bởi chương trình này được cho thấy ở đây.

```
Size = 10
Contents: ABCDEFGHIJ
```

*This program creates a list of characters. First, an empty list object is created. Next, ten characters, the letters A througn J, are put into the list. This is acconmplished with the **push_back()** function, which puts each new value on the end of the existing list. Next, the size of the list is pisplayed. Then, the contents of the list are output by repeatedly obtaining, displaying, and then removing the first element in the list. This process stops when the list is empty.*

Chương trình này tạo ra một danh sách những đặc tính. Đầu tiên, một đối tượng **list** rỗng được tạo ra. Tiếp theo, mười đặc tính, những mẫu tự từ A qua J, được đặt vào trong danh

sách. Nó được hoàn thành với hàm **push_back()**, mang mỗi giá trị mới vào vị trí kết thúc của danh sách hiện hữu. Tiếp theo, kích thước của danh sách được trình bày. Rồi, nội dung của danh sách được gửi ra bởi nhiều lần sử dụng, trình bày, và sau đó loại bỏ phần tử đầu tiên trong danh sách. Quá trình này dừng lại khi danh sách rỗng.

In the previous example, the list was emptied as it aws traversed. This is, of course, not necessary. For example, the loop that displays the list could be recorded as shown here.

2. Trong ví dụ trước đây, danh sách trở nên rỗng như nó được xét toàn bộ. Điều này, tất nhiên, không cần thiết. Chẳng hạn, vòng mà trình bày danh sách có thể được lưu giữ như được cho thấy ở đây.

```
list<char>::iterator p = lst.begin();

while(p != lst.end())
{
  cout << *p;
  p++;
}
```

*Here the iterator **p** is initialized to point to the start of the list. Each time through the loop, **p** is incremented, causing it to point to the next lelement. The loop ends when **p** points to the end of the list.*

Ở đây iterator **p** được khởi tạo để trỏ vào vị trí bắt đầu của danh sách. Mỗi lần thông qua vòng,**p** tăng trị số,đó là nguyên nhân việc tạo ra nó để trỏ vào phần tử kế tiếp. Vòng kết thúc khi **p** trỏ vào vị trí kết thúc của danh sách.

3. *Because lists are bidirectional, elements can be put on a list either at the front or at the back. For example, the following program creates two lists, with the first being the reverse of the second.*

3. Vì những danh sách là hai chiều, những phần tử có thể được mang một danh sách tại mặt trước hay ở phía sau. Chẳng hạn, chương trình sau đây tạo ra hai danh sách liệt kê, với danh sách đầu tiên là sự nghịch đảo của danh sách thứ hai.

```
// Elements can be put on the front or end of a list.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst;
  list<char> revlst;
  int i;

  for(i=0; i<10; i++) lst.push_back('A'+i);

  cout << "Size of lst = " << lst.size() << endl;
  cout << "Original contents: ";
```

```
    list<char>::iterator p;

    /* Remove elements from lst and put them
       into revlst in reverse order. */
    while(!lst.empty())
    {
      p = lst.begin();
      cout << *p;
      lst.pop_front();
      revlst.push_front(*p);
     }
    cout << endl << endl;

    cout << "Size of revlst = ";
    cout << revlst.size() << endl;
    cout << "Reversed contents: ";
    p = revlst.begin();
    while(p != revlst.end())
    {
      cout << *p;
      p++;
    }

    return 0;
  }
```

This program produces the following output.

Chương trình này xuất kết quả sau đây.

```
Size of list = 10
Original contents: ABCDEFGHIJ
Size of revlst = 10
Reversed contents: JIHGFEDCBA
```

*The program, the list is reversed by removing elements from the start of **lst** and pushing them onto the front of **revlst.** This causes the elements to be stored in the reverse order in **revlst.***

Trong chương trình, danh sách được đảo ngược bởi việc loại bỏ những phần tử bắt đầu của **1st** và đẩy họ lên trên mặt trước của **revlst**. Việc này gây ra những phần tử sẽ được cất giữ với thứ tự ngược trong **revlst**.

*4. You can sort a list by calling the **sort()** member function. the following program creates a lost of random characters and then pits the list intosorted order.*

4. Bạn có thể phân loại một danh sách bằng cách gọi chức năng thành viên **sort()**. Chương trình sau đây tạo ra một danh sách những đặc tính ngẫu nhiên và sau đó đặt danh sách vào trong thứ tự được phân loại.

```
// Sort a list.
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
  list<char> lst;
  int i;

  // create a list of random characters
  for(i=0; i<10; i++)
    lst.push_back('A'+ (rand()%26));

  cout << "Original contents: ";
  list<char>::iterator p = lst.begin();
  while(p != lst.end())
  {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  // sort the list
  lst.sort();

  cout << "Sorted contents: ";
  p = lst.begin();
  while(p != lst.end())
  {
    cout << *p;
    p++;
  }

  return 0;
}
```

Here is sample out put produced by the program.

Đây là kết quả xuất ra bởi chương trình.

```
Original contents : PHQGHUMEAY
Sorted contents: AEGHMPQUY
```

5. One ordered list can be mergerd with another. The result is an ordered list that contains the contents of the two original lists. The new list I sleft in the incoking list and the second list is left empty. This example merges two lists. The first contains the letters ACEGI and the second BDFHJ. These lists are then merged to produce the sequence BCDEFGHIJ.

5. Một danh sách thứ tự có thể được hòa trộn với danh sách khác. Kết quả là một danh sách thứ tự mà chứa đựng nội dung của cả hai danh sách nguyên bản. Danh sách mới được để lại trong danh sách kéo theo và danh sách thứ hai trống rỗng. Ví dụ này hòa trộn hai danh sách. Danh sách đầu tiên chứa đựng những ký tự ACEGI và danh sách thứ hai là BDFHJ . Những danh sách này được hòa trộn rồi xuất ra chuỗi ABCDEFGHIJ.

```
// Merge two lists.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst1, lst2;
  int i;

  for(i=0; i<10; i+=2) lst1.push_back('A'+i);
  for(i=1; i<11; i+=2) lst2.push_back('A'+i);

  cout << "Contents of lst1: ";
  list<char>::iterator p = lst1.begin();
  while(p != lst1.end())
  {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  cout << "Contents of lst2: ";
  p = lst2.begin();
  while(p != lst2.end())
  {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  // now, merge the two lists
  lst1.merge(lst2);
  if(lst2.empty())
  cout << "lst2 is now empty\n";

  cout << "Contents of lst1 after merge:\n";
  p = lst1.begin();
  while(p != lst1.end())
  {
```

```
    cout << *p;
    p++;
  }

  return 0;
}
```

The output produced by this program is shown here:

Kết quả của chương trình này như sau:

```
Contents of lst1: ACEGI
Contents of lst2: BDFHJ
Lst2 is now empty
Content of lst1 after merge: ABCDEFGHIJ
```

6. *Here is an example that uses a list to store objects of type.* ***Project,*** *which is a class that helps manage software projects. Notice that the <, >, != and == operators are overloaded for objects of type* ***project.*** *These are the operators that were required by Microsoft's Visual C++ 5(the compiler used to test the STL examples in this chapter). Other compilers might require you to overload additional operators. The STL uses these functions to determine the ordering and equality of container. Even through a list is not an ordered container, it still needs a way to compare elements when searching, sorting, or merging.*

6. Ở đây là một ví dụ mà sử dụng một danh sách để cất giữ những đối tượng kiểu **Project**, mà một lớp trợ giúp quản lý những dự án phần mềm. Chú ý các toán tử <, >,!=, và == bị tràn cho những đối tượng của kiểu **Project**. Đây là những toán tử mà được đòi hỏi bởi Microsoft's Visual C++5 (trình biên dịch thường kiểm tra những ví dụ STL trong chương này). Những trình biên dịch khác có lẽ đã yêu cầu bạn bổ sung những toán tử quá tải. STL sử dụng những chức năng này để xác định trật tự điều chỉnh và quyền bình đẳng của những đối tượng trong một vùng chứa. Mặc dù một danh sách không là một vùng chứa thứ tự. Nó vẫn còn cần một cách so sánh những phần tử khi tìm kiếm, chọn hay hòa trộn.

```
#include "stdafx.h"

#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class Project
{
     public:
          char name[40];
         int days_to_completion;
          Project()
          {
```

```
          strcpy(name, "");
          days_to_completion = 0;
           }
          Project(char *n, int d)
          {
               strcpy(name, n);
               days_to_completion = d;
          }

          void add_days(int i)
          {
               days_to_completion += i;
          }

          void sub_days(int i)
          {
               days_to_completion -= i;
          }

          bool completed() { return !
days_to_completion; }

          void report()
          {
               cout << name << ": ";
               cout << days_to_completion;
               cout << " days left.\n";
          }
};

bool operator<(const Project &a, const Project &b)
{
     return a.days_to_completion < b.days_to_completion;
}

bool operator>(const Project &a, const Project &b)
{
      return a.days_to_completion > b.days_to_completion;
}

bool operator==(const Project &a, const Project &b)
{
     return a.days_to_completion == b.days_to_completion;
}

bool operator!=(const Project &a, const Project &b)
{
```

```
    return a.days_to_completion != b.days_to_completion;
}

int main()
{
    list<Project> proj;

    proj.push_back(Project("Compiler", 35));
    proj.push_back(Project("Spreadsheet", 190));
    proj.push_back(Project("STL Implementation", 1000));

    list<Project>::iterator p = proj.begin();

  /* display projects */
    while(p != proj.end())
    {
         p->report();
        p++;
    }

  // add 10 days to 1st project
    p = proj.begin();
    p->add_days(10);

  // move 1st project to completion
    do
    {
        p->sub_days(5);
        p->report();
    } while (!p->completed());

    return 0;
}
```

The output from this program is shown here.

Kết quả của chương trình như sau:

```
Compiler: 35 days left
Spreadsheer: 190 days left.
STl Implementation: 1000 days left
Compiler: 40 days left.
Compiler: 35 days left.
Compiler: 30 days left.
Compiler: 25 days left.
Compiler: 20 days left.
Compiler: 15 days left.
Compiler: 10 days left.
Compiler: 5 days left.
Compiler: 0 days left.
```

EXERCISE:

Experiment with the examples, trying minor varations.
In example 1, the list was emptied in the process of displaying it. In example 2, you saw another way to traverse a list that does not destroy it. Can you think of another way to traverse a list without emptying it? Show that your solution works by substituting it into the program in Example 1.
Expand Example 6 by creating another list of projects that consists of the following:

1. Thử nghiệm những ví dụ, thử những sự biến đổi phụ

Trong ví dụ 1, danh sách được trở nên rỗng trong quá trình trình bày nó. Trong Ví dụ 2, bạn nhìn thấy cách khác để duyệt qua một danh sách mà không phá hủy nó. Bạn có thể nghĩ cách khác để duyệt qua một danh sách mà không làm rỗng nó không? Chỉ ra giải pháp làm việc của bạn bởi việc thay thế nó vào trong chương trình trong ví dụ 1.

3. Mở rộng ví dụ 6 tạo ra một danh sách liệt kê những dự án mà gồm có sự tiếp sau về thời gian:

Project	***Days to Completion***
Database	*780*
Mail merge	*50*
COM objects	*300*

Dự án	**Ngày hoàn thành**
Database	780
Mail merge	50
COM objects	300

Next, sort both lists and then merge them together. Display the final result

Tiếp theo, phân loại cả hai danh sách và sau đó hòa trộn chúng với nhau. Trình bày kết quả cuối cùng.

14.4. MAPS - BẢN ĐỒ:

*The **map** class supports an associative container in which unique keys are mapped with values. In essence, a key is simply a name that you give to a value. Once a value has been stored,you can retrieve it by using its key. Thus , in its most general senes a map is a list of key/ value pairs. The power of a map is*

that you can look up a value given its key. For example, you could define a map that uses a person's name as its key and stores that person's telephone number as its value. Associative container are becoming more popular in programming.

Bản đồ lớp hỗ trợ một dữ liệu liên quan trong chìa khóa duy nhất nào được vẽ trong bản đồ với những giá trị. Thật chất, chìa khóa đơn giản là một tên mà bạn đưa vào một giá trị. Một lần một giá trị được cất giữ, bạn có thể khôi phục được nó bởi một chìa khóa. Như vậy, trong xu thế chung nhất bản đồ là một danh sách chìa khóa/ giá trị. Điểm mạnh của bản đồ là bạn có thể nhìn thấy giá trị được đưa cho chìa khóa đó. Ví dụ, bạn có thể định nghĩa một bản đồ sử dụng tên như chìa khóa và đưa những giá trị vào là những con số điện thoại. Liên kết chứa trở nên thông dụng hơn trong lập trình.

*As mentioned, a map can hold only unique keys. Duplicate keys are not allowed. To create a map that allows nonunique keys use **mulitimap**.*

Như đã đề cập một bản đồ giữ có thể giữ duy nhất một chìa khóa. Những chìa khóa bản sao không được cho phép. Để tạo tra bản đồ mà cho phép sử dụng các chìa không duy nhất **mulitimap.**

The map container has the following template specification:

Bản đồ chứa có khung mẫu như sau:

Template <class Key, class T, class Comp=less<Key>, class Allocator = allocator<T> class map

*Here **Key** is the data type of the keys, T is the data type of the values being stored (mapped), and **Comp** is a function that compares two keys. This defaults to the standard **less**() utility function object. **Allocator** is the allocator (which defaults to allocator).*

Ở đây **key** là dữ liệu những chìa khóa, T là dữ liệu của những giá trị được cất giữ (bản đồ), và **Comp** là hàm dùng để so sánh hai chìa khóa. Những xác lập mặc định chuẩn **less** () là tiện ích đối tượng hàm. **Allocator** là một bộ phận(mà chương trình phân bố).

*The **map** class has the following constructors:*

Bản đồ lớp được xây dựng như sau:

explicit map (const Comp & cmpfn = Comp(), const Allocator & a= Allocator());

map(const map<Key, T , Comp, Allocator> & ob);

template<class Inter> map(Inter start, Inter end, const Comp & cmpgfn = Comp(), const Allocator &a = Allocator));

The first form constructors an empty map. The second form constructors a map that contains the same elements as ob. The third form constructs a map that contains the elements in the range specified by the iterators the ordering of the map.

Đầu tiên là phải xây dựng một biểu đồ rỗng. Thứ hai xây dựng một bản đồ chứa đựng cùng phần tử ob. Thứ ba từ việc xây dựng một bản đồ có chứa những phần tử trong phạm vi chỉ trỏ bởi những iterator điều khiển bản đồ.

In general, any object used as a key must define a default constructor and overload any necessary comparison operators.

Nói chung, bất kì đối tượng nào được sử dụng như một chìa khóa phải định nghĩa phương thức mặc định và hàm chồng khi toán tử so sánh cần thiết.

*The following comparison operators are defined for **map**.*

Những toán tử so sánh sau được định nghĩa cho **bản đồ.**

==, <, <=, !=, >, >=

*The member functions defined by map are shown in Table 14-4. In the descriptions **key_type** is the of the type of the key and **value_type** represents **pair< Key, T***>

Những chức năng thành viên được định nghĩa bởi bản đồ được đưa vào Bảng 14-4. Trong phần mô tả **key_type** là kiểu của những chìa khóa và **value_type** trình bày**pair< Key, T>**

Member Function	**Description**
iterator begin();	Return an iterator to the first element in the map.
const_iterator begin() const	
void clear();	Removes all elements form the map.
size_type count(const *key_type* & k) const;	Return the number of times k occours in the map(1 or 0).
bool empty() const;	Return true if the invoking map is empty and false otherwise.
iterator end();	Return an iterator to the end of the map.
const_iterator end() const;	
pair<iterator, iterator> equa_range (const key_type & k);	Return a pair of iterator that point to the first and last elements in the map that contain the specified key.
pair<const_iterator, const_iterator> equa_range (const *key_type* & k) const;	
void erase(iterator i);	Removes the element pointed to by i.
void arese(iterator strart,iterator end);	Removes the elements in the range start to end.
size_type erase(const key_type & k);	Removes elements that have keys with the value k.
iterator find(const key_type &k);	Returns an iterator to the specified key. If the key is not found, an iterator to the end of the map is returned.
const_iterator find(const key_type & k) const;	Return the map's allocator.
allocator_type get_allocator() const;	Insert val at or after the element specified by i. An iterator to the element is returned.
iterator insert (iterator I, const value_type & val);	Inserts a range of elements
template<class Inter> void insert(Inter start, Inter end);	Iterator val into the invoking map. An iterator to the element is returned. The element is inserted only if it does not already exist, if the element was insert, **pair<iterator, true**> is returned.
pair<iterator, bool> void insert(const value_type & val);	

key_compate key_comp()const;	Otherwise, **pair<iterator, false**> is returned.

iterator lower_bound(const key_type & k);
const_iterator
lower_bound(const key_type &k) const;
	Returns the function object that compares keys.
	Return an iterator to the fisrt element in the map with the key equal to or greater than k.

size_type max_zise()const;	Return the maximun number of elements that the map can hold
reference operator[](const key_type & i)	Returns a reference to the element specified by i. If this element does not exist, it is inserted.

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
size_type size () const;
	Returns a reverse iterator to the end of the map.
	Returns a reverse iterator to the start of the map.
	Return the number of elements currently in the map.

void swap(map<Key, T, Comp, Allocator> &ob);	Exchanges the elements strored in the invoking map with those in ob.
iterator upper_bound(const key_type & k); const_iterator upper_bound(const key_type & k) const;	Returns an iterator to the first element in the mao with the key greater than k.
value_compare value_comp() const;	Returns the function object that compares values.

Member Function	**Description**
iterator begin(); const_iterator begin() const	Trả lại một iterator của phần tử đầu tiên trong bản đồ.
void clear();	Lạoi bỏ tất cả các phần tử tử bản đồ.
size_type count(const *key_type* & k) const;	Trả lại những con số xuất hiện k lần trong bản đồ(1 hoặc 0).
bool empty() const;	Trả lại là đung nếu viêc kéo theo trong bản đôg là rỗng và sai ngươc lại.
iterator end(); const_iterator end() const;	Trả về một iterator khi kết thúc bản đồ.
pair<iterator, iterator> equa_range (const key_type & k); pair<const_iterator, const_iterator> equa_range (const *key_type* & k) const;	trả lại một cặp iterator và đưa con trỏ về đầu tiên và những phần tử cuối cùng trong bản đồ chứa dựng chía khóa được chỉ rõ.
void erase(iterator i);	.
void arese(iterator strart,iterator end);	Loại bỏ phần tử con trỏ tại vị trí i. Loại bỏ những phần tử trong phạm vi từ bắt đầu đến kết thúc.
size_type erase(const key_type & k);	Loại bỏ những phần tử mà có những chìa khóa có giad trị là k.
iterator find(const key_type &k);	

const_iterator find(const key_type & k) const;	Nếu chìa khóa không tìm thấy, một iterator tới kết thúc bản đồ được trả lại.
allocator_type get_allocator() const;	Trả lại sự phân bố bản đồ.
iterator insert (iterator I, const value_type & val);	Chèn *val* tại hoặc sau phần tử chỉ rõ tại vị trí i.Một iterator tới phần tử được trả lại.
template<class Inter> void insert(Inter start, Inter end);	Chèn những phạm vi của phần tử.
pair<iterator, bool> void insert(const value_type & val);	Iterator val vào trong kéo bản đồ. Một iterator tới phần tử được trả lại. Phần tử chèn vào khi phần tử chưa tồn tại, nếu phần tử chèn, **pair<iterator, true>** được trả lại.Mặt khác, **pair<iterator, false>** được trả lại.
key_compate key_comp()const;	Những sự trở lại đối tượng mà những sự so sánh chức năng là chìa khoá.
iterator lower_bound(const key_type & k); const_iterator lower_bound(const key_type &k) const;	Trả lại một iterator của phần tử đầu tiên trong bản đồ với khóa bằng nhau hay lớn hơn so với k.
size_type max_zise()const;	Trả về số lượng phần tử mà bản đồ có thể giữ.
reference operator[](const key_type & i)	Trả lại tham chiếu của phần tử được chỉ rõ tại vị trí i. Nếu phần tử không tồn tại, nó được chèn vào.
reverse_iterator rbegin();	Trả lại một iterator đảo ngược cho tới kết thúc trong bản đồ.
const+reverse_iterator rbegin() const;	Trả lại một iterator đảo ngược đầu tiên trong bản đồ.
reverse_iterator rend();	Trả lại số lượng phần tử hiện thời trong bản đồ.
const_reverse_iterator rend() const;	Trao đổi những phần tử cất giữ trong việc kéo theo bản đồ với đối tượng ob.
size_type size () const;	Trả lại một iterator phần tử đầu tiên trong bản đồ với chìa khoá lớn hơn so với k.
void swap(map<Key, T, Comp, Allocator> &ob);	Những sự trở lại đối tượng mà những sự so sánh chức năng đánh giá.
iterator upper_bound(const key_type & k);	
const_iterator upper_bound(const key_type & k) const;	
value_compare value_comp() const;	

Key/ value pairs are stored in a map as object of type pair, which has this template specification:

Chía khoá/ giá trị là những cặp cất giữ trong bản đồ khi đối tượng được ghép đôi, mà được chỉ dẫn dưới:

```
template < class Ktype, class
Vtype> struct pair { typedef Ktype
```

```
    first_type; // type of key
    typedef Vtype second _type; //
    type of value
    Ktype first ; // contains the key
    Vtype second; // contains the value

    //constructors
    pair();
    pair( const Ktype &k, const Vtype
    &v);
    template<class A, class B>
    pair(const <A, B> ob);
}
```

*As the comments suggest, the value in first contains the key and the value in **second** contains the value associated with that key.*

Như những bình luận gợi ý, giá trị đầu tiên chứa đựng chìa khóa và giá trị thứ hai chứa đựng giá trị được liên quan đến mà chìa khóa.

*You can construct a pair using either one of **pair's** constructors or by using **make_pair()** , which constructs a **pair** object based upon the types of the data used as parameters **make_pair()** is a generic function that has this prototype:*

Bạn có thể xây dựng một cặp đang sử dụng cả hai một **cặp** những người xây dựng hay gần sử dụng **make_pair()** , mà xây dựng một **cặp** đối tượng dựa trên những kiểu dữ liệu được dùng như những tham số **make_pair()** là một chức năng chung mà có nguyên mẫu này:

Template<class *Ktype*, class *Vtype*> pair <*Ktype*,
Vtype> make_pair(const *Ktype* &k , const
Vtype & v);

As you can see, it return a pair object consisting of values of the types specified by Ktype and Vtype. The advantage of make_pair() is that it allows the types of the types of the objects being stored to be determined automatically by the compiler rather than being explicitly specified.

Như bạn có thể nhìn thấy, nó trả lại một đối tượng ở chỗ những giá trị của kiểu được chỉ rõ bởi Ktype và Vtype. Lợi thế của make_pair() là nó cho phép những kiểu của những kiểu những đối tượng đang được cất giữ sẽ được xác định tự động gần của người biên tập đúng hơn so với rõ ràng được chỉ rõ.

EXAMPLES - Ví Dụ

The following program illustrates the basics of using a map. It stored ten key/ value pairs. The key is a character and the value is an integer. The key/ value pairs stored are

Chương trình dưới đây minh họa cơ sở của việc sử dụng một bản đồ. Nó cất giữ 10 cặp chìa khóa/ giá trị. Chìa khóa là một đặc tính và giá trị là một số nguyên. Những

cặp chìa khóa/ giá trị được cất giữ:

```
A       0
B       1
C       2
```

and so on. Once the pair's have been stored, the user is prompted for a key(I.e , a letter form A through J), and the value associated with that key is displayed.

vân vân. Một lần nhưng cặp có được cất giữ, người sử dụng được nhắc cho một chìa khóa (I.e, một bức thư hình thành A xuyên qua J), và giá trị kết hợp với chìa khóa kia được trình bày.

```
// A simple map demonstration
#include <iostream>
#include <map>
using namespace std;

int main()
{
     map<char, int> m;
     int i;

     //put pair into map
     for(i = 0; i < 10 ;i++)
     {
          m.insert(pair<char, int>('A'+i, i));
     }

     char ch;
     cout<< "Enter key :";
     cin>> ch;
     map<char , int>::iterator p;

     // find value given key
     p=m.find(ch);
     if(p!=m.end ())
          cout<< p->second;
     else
          cout<< "Key not in map \n";

     return 0;
}
```

Notice the use of the pair template to construct the key/ value pairs. The data types specified by pair must match those of the map into which the pair are being inserted.

Chú ý những cặp sử dụng khung mẫu cặp để xây dựng chìa khóa/ giá trị. Những kiểu dữ liệu được chỉ rõ bởi cặp phải phù hợp của bản đồ vào trong đó cặp đang

được chèn vào.

*Once the map has been initialized with keys and value, you can search for a value given its key by using the **find()** function. **Find()** returns an iterator to the matching element or to the end of the map if the key is not found. When a match is found, the value associate with the key is contained in the **second** member of **pair**.*

Một lần bản đồ đã được khởi tạo với những chìa khóa và giá trị, bạn có thể tìm kiếm một giá trị cho những bởi chìa khóa gần sử dụng hàm **find().** Hàm **find()** trả lại một iterator cho phần tử thích ứng hay tới kết thúc bản đồ nếu chìa khóa không được tìm thấy. Khi một kết quả được tìm thấy, giá trị liên tưởng với chìa khóa được chứa đựng trong **Second** của thành viên **cặp.**

*In the preceding example, key/ value pairs were constructed explicitly, using pair< char, int > . Although there is nothing wrong with this approach, it is often easier to use **make_pair()** , which constructs a pair object upon the types of the data used as parameters. For example, assuming the previous program, this line of code will also insert key/ value pairs into **m:***

Trong ví dụ có trước, những cặp chìa khóa/ giá trị xây dựng rõ ràng, sử dụng pải < char, int>. Dù không có gì sai với cách tiếp cận này, nó thường dễ dàng hơn đối với sự sử dụng **make_pair()** , mà xây dựng một cặp đối tượng ở trên những kiểu dữ liệu sử dụng như tham số. Chẳng hạn, giả thiết chương trình trước đây, những cặp chìa khóa/ giá trị chèn đồng ý định mã cũng này vào trong **M:**

```
m.insert (make_pair((char) ('A' +i), i));
```

*Here the cast to char is needed to override the automatic conversion to **int** when I is added to 'A'. Otherwise, the type determination is automatic.*

Ở đây cần sắc thái để ký tự để sự chuyển đổi tự động tới **int** khi tôi được thêm vào 'At'. Cách khác, sự xác định kiểu tự động.

*Like all of the containers, maps can be used to store object of types that you create. For example, the program shown here create that a map of words with their opposite. To do this it creates two classes called **word** and **opposite**. Sine a map maintains a soteres list of keys, the program also defines the <operator for operator objects of type **word** . In general, you must defines the< operator for any classes called that you will use as keys.(Some compilers might require that additional comparison operators be defined.)*

Cũng như tất cả những chứa, những bản đồ có thể được dùng để cất giữ đối tượng những kiểu mà bạn tạo ra. Ví dụ, chương trình được cho thấy ở đây tạo ra mà một bản đồ những từ với đối ngược nhau. Để làm đều này tạo ra hai lớp được gọi là **call** và **opposite**. Từ khi một bản đồ bảo cất giữ liệt kê của những chìa khóa, chương trình cũng định nghĩa thao tác viên < cho những đối tượng thao tác với kiểu dử

liệu **word.** . Nói chung,bạn phải định nghĩa thao tác cho bất kỳ lớp nào được gọi là mà bạn sẽ sử dụng như những chìa khóa. .(Một số người biên tập có lẽ đã yêu cầu những toán tử so sánh bổ sung được định nghĩa.)

```
//A map oppsites
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class word
{
     char str[20];
     public:
          word()
          {
               strcpy(str, "");
          }
          word(char * s)
          {
               strcpy(str, s);
          }
          char *get()
          {
               return str;
          }
};

//must define less than reletive to word objects
bool operator<(word a, word b)
{
     return strcmp(a.get(),b.get())<0;
}

class opposite
{
     char str[20];
     public:
          opposite()
          {
               strcmp(str, " ");
          }
          opposite(char *s)
          {
               strcpy(str, s);
          }
          char *get()
```

```
            {
                return str;
            }
};

int main()
{
     map<word ,opposite> m;

     //put words and opposite into map
     m.insert(pair<word,opposite>(word("yes"),
               opposite("no")));
     m.insert(pair<word,opposite>(word("good"),
               opposite("bad")));
     m.insert(pair<word,opposite>(word("left"),
               opposite("right")));
     m.insert(pair<word,opposite>(word("up"),
               opposite("down")));

     //given a word, find opposite
     char str[80];
     cout<< "Enter word : ";
     cin>> str;
     map<word, opposite>::iterator p;

     p=m.find(word(str));
     if(p!=m.end())
           cout<< " Opposite : " << p-
               >second.get();
     else
          cout<< " Word not in map .\n";

          return 0;
}
```

EXERICES - Bài Tập

Experiment with the examples, trying small variations.

Thử nghiệm những ví dụ, thử những sự biến đổi nhỏ.

Create a map that contains names and telephone munbers. Allow names and numbers to be entered, and set up your program so that a number can be found when a name is given.

Tạo ra một bản đồ mà chứa đựng những tên và những số điện thoại. Cho phép những tên và những số sẽ được vào, và thiết lập chương trình các bạn sao cho một số có thể

được tìm thấy khi một tên được cho.

(Hint: Use Example 3 as a model.)

Do you need to define the <operator for object used as keys in map ?

Bạn cần định nghĩa thao tác viên< operator đối tượng được dùng như những chìa khóa trong bản đồ không?

14.5. ***ALGORITHMS - Những thuật giải***

*As explained, algorithms act on containers. Although each container provides support for its own basic operations, the standard algorithms provides more extended or extended or complex actions. They also allow you to word with two different types of containers at the same time. To have access to the STL algorithms, you must include <**algorithm**> in your program.*

Như được giải thích, những giải thuật hành động theo những chứa. Dù mỗi chứa cung cấp sự hỗ trợ những thao tác cơ bản của mình, những giải thuật tiêu chuẩn cung cấp những hoạt động mở rộng hơn hay mở rộng hay phức tạp. Chúng cũng cho phép bạn tới từ với hai kiểu khác nhau của những chứa cùng lúc. Để có sự truy nhập tới những giải thuật STL , bạn khai báo include <**algrithm>** trong chương trình của các bạn.

The STL defines a large number of algorithms, which are summarized in Table 14-5. All of the algorithms are template functions. This means that they can be applied to any type of container. The example that follow demonstrate a representative sample.

STL định nghĩa một số lớn giải thuật, mà được tổng kết trong Bảng 14-5. Tất cả những giải thuật là những chức năng khung mẫu. Những phương tiện này mà chúng có thể được ứng dụng vào bất kỳ kiểu nào của chứa. Ví dụ mà đi theo sau trình diễn một mẫu đại diện.

Algorithm	Purpose
adjacent_find	Searches for adjacent matching element within a sequence and return an iterator to the first match.
binary_search	Performs a binary search on an ordered sequence.
copy	Copies a sequence.
copy_backward	Same as **copy()** except that it moves the elements from the end of the sequence first.
count	Returns the number of elements in the sequence.
count_if	Returns the number of elements in the sequence that satisfy some predicate.
equal	Determines whether two range are the same.
equal_range	Returns a range in which an element can be inserted into a sequence without disrupting the ordering of the sequence.
fill	Fills range with the specified value.
fill_n	Searches a range for a value an returns an iterator to the

find	first occurrence of the element.
find_end	Searches a range for a subsequence.This function returns an iterator to the end of the subsequence within the range.
find_fisrt_of	Finds the first element within a sequence that matches an element within a range.
find_if	Searches a range for an element for which a user-defined unary predicate returns true.
for_each	Applies a function to a range of elements
generate	Assigns to elements in a range the values returned by a generator function.
generate_n	Determines whether one sequence includes all of the elements in another sequence.
includes	Merges a range with another range. Both ranges must be sorted I increasing order. The resulting sequence is sorted.
inplace_merge	Exchanges the values pointed to by two iterator arguments.
iter_swap	Alphabetically compares one sequence with another.
lexicographical_compare	Finds the first point in the sequence that is not less than a specified value.
lower_bound	Constructs a heap from a sequence
make_heap	Returns the maximun of two values
max	Returns an iterator to the maximum element within a range.
max_element	Merges two ordered sequences, placing the result into a third sequence.
merge	Returns the minimum of two values.
min	Returns an iterator to the minimun element within a range.
min_element	Finds the first mismatch between the element in two sequences. Iterators to the two elements are returned.
mismatch	Constructs the next permutation of a sequence.
next_permutation	Arranges a sequence such that all elements less than a specified element E come before that element and all elements greater than E come after it.
nth_element	Sorts a range.
partial_sort	Sorts a range and than copies as many elements as will fit into a reasult sequence.
partial_sort_copy	Arranfges a sequence such that all elements for which a predicate returns true come before those for which the predicate returns flase.
partition	Exchanges the first last-1 elements and then rebuilds the heap.
pop_heap	Constructs the previous permutation of a sequence.
prev_permutation	Pushes an element into the end of a heap.
	Randomizes a sequence.
	Revomes elements from a specified range.

push_heap	
random_shuffle	Replaces element within a range.
revome	
revome_if	
revome_copy	
revome_copy_if	Reverses the order of a range
replace	Left-rotate the elements in range.
replace_copy	
replace_if	Searches for a subsequence within a sequence.
replace_copy_if	Searches for a subsequence of a specified number of similar
reverse	element.
reverse_copy	Produces a sequence that contains the difference between
rotate	two ordered sets.
rotate_copy	Produces a sequence that contains the intersection of two
search	ordered sets.
search_n	Produces a sequence that contains the symmetric difference
	between two ordered sets.
set_difference	Produces a sequence that contains the union of two ordered
	sets.
set_intersection	Sort a range.
	Sorts a heap within a specified range.
set_symmetric_defference	Arranges a sequence such that all elements for which a
	predicate returns true come before those for which the
set_union	predicate return false. The partitioning is stable; the
	relative ordering of the sequence is preserved.
sort	Sorst a range. The sort is stable; equal elements are the not
sort_heap	rearranged.
stable_partion	Exchanges two values.
	Exchanges elements in a range.
	Applies a function to a range of elements and stored the
	uotcome in a new sequence.
stable_sort	Eliminates duplicate elements from a range.
swap	Finds the last point in a sequence that is not greater than
swap_range	some value.
transfrom	
unique	
unique_copy	
upper_bound	

Algorithm	Purpose
adjacent_find	Tìm kiếm phần tử thích ứng kề bên bên trong một chuỗi

	và sự trở lại một iterator tới đầu tiên phù hợp.
binary_search	Thực hiện một sự tìm nhị phân trên một chuỗi được ra lệnh.
copy	Sao chép một chuỗi.
copy_backward	Cùng như **copy ()** sự loại trừ mà nó di chuyển những phần tử từ kết thúc chuỗi đầu tiên.
count	Trả lại những số phần tử trong chuỗi.
count_if	Trở lại số phần tử trong chuỗi mà thỏa mãn vị từ nào đó.
equal	Xác định liệu có phải hai phạm vi giống như vậy
equal_range	Trả lại một phạm vi trong đó một phần tử có thể được chèn vào trong một chuỗi không có việc phá vỡ sự điều chỉnh chuỗi.
fill	Những sự điền vào cùng dãy với giá trị được chỉ rõ
fill_n	Tìm kiếm một phạm vi một giá trị một iterator những trở
find	lại tới biến cố đầu tiên phần tử.
find_end	Tìm kiếm một phạm vi một sự đến sau.Chức năng trả lại một iterator cho kết thúc sự đến sau bên trong phạm vi.
	Tìm thấy phần tử đầu tiên bên trong một chuỗi mà phù hợp với một phần tử bên trong một phạm vi.
find_fisrt_of	Tìm kiếm một phạm vi một phần tử mà đó một vị từ đơn nguyên do người dùng định ra trả lại đúng.
find_if	Áp dụng một chức năng tới một phạm vi những phần tử.
for_each	Gán tới những phần tử trong một phạm vi những giá trị được trả lại bởi một chức năng máy phát
generate	Xác định liệu có phải một chuỗi bao gồm mọi thứ những phần tử trong chuỗi khác.
generate_n	Hòa trộn một phạm vi với phạm vi khác. Cả hai phạm vi
includes	việc phải được phân loại Tôi tăng ra lệnh. Chuỗi kết quả được phân loại.
inplace_merge	Trao đổi những giá trị được trỏ vào bởi hai lý lẽ iterator.
	Theo thứ tự abc so sánh một chuỗi với khác.
	Những sự tìm kiếm đầu tiên chỉ trong chuỗi mà không ít
iter_swap	hơn so với một giá trị được chỉ rõ.
	Xây dựng một đống từ một chuỗi.
lexicographical_compare	Trả lại cực đại hai giá trị.
lower_bound	Một iterator những trở lại tới phần tử cực đại bên trong một phạm vi.
make_heap	Những sự hòa trộn hai sắp đặt những chuỗi, đặt kết quả
max	vào trong một chuỗi thứ ba.
max_element	Trả lại min hai giá trị.
	Trả lại một iterator phần tử nhỏ nhất bên trong một
merge	phạm vi.
	Tìm thấy sự không thích ứng đầu tiên giữa phần tử trong
min	hai chuỗi. Iterators tới những hai phần tử được trở lại.
min_element	Xây dựng sự hoán vị kế tiếp của một chuỗi.

mismatch	Thu xếp một chuỗi sao cho tất cả các phần tử ít hơn so với một phần tử được chỉ rõ E đến trước phần tử kia và tất cả phần tử lớn hơn so với đến sau nó.
next_permutation	Phân loại một phạm vi.
nth_element	Phân loại một phạm vi so với sao chép nhiều phần tử thích hợp vào trong một chuỗi reasult.
partial_sort	Sắp đặt một chuỗi sao cho tất cả các phần tử mà đó một vị từ trả lại đúng đến trước khi người vị từ nào trả lại flase
partial_sort_copy	Trao đổi những phần tử-1 cuối cùng đầu tiên và sau đó xây dựng lại đống.
partition	Xây dựng sự hoán vị trước đây một chuỗi.
pop_heap	Đẩy một phần tử vào trong kết thúc một đống.
prev_permutation	Làm ngẫu nhiên một chuỗi.
push_heap	Xóa những phần tử từ một phạm vi được chỉ rõ.
random_shuffle	
revome	Thay thế phần tử bên trong một phạm vi.
revome_if	
revome_copy	
revome_copy_if	
replace	Đảo ngược một phạm vi.
replace_copy	
replace_if	Left-rotate Những phần tử trong phạm vi.
replace_copy_if	
reverse	Tìm kiếm một sự đến sau bên trong một chuỗi.
reverse_copy	Những sự tìm kiếm một sự đến sau một số được chỉ rõ phần tử tương tự.
rotate	
rotate_copy	Kết quả một chuỗi mà chứa đựng sự khác nhau giữa hai ra lệnh những tập hợp.
search	
search_n	Kết quả một chuỗi mà chứa đựng sự khác nhau cân đối giữa hai sắp đặt những tập hợp.
set_difference	Kết quả một chuỗi mà chứa đựng liên hiệp hai sắp đặt những tập hợp.
set_intersection	Kết quả một chuỗi mà chứa đựng liên hiệp (của) hai sắp đặt những tập hợp.
set_symmetric_defference	Phân loại một phạm vi.
set_union	Phân loại một đống bên trong một phạm vi được chỉ rõ.
sort	Thu xếp một chuỗi sao cho tất cả các phần tử một thuộc tính những sự trở lại đúng đến trước khi người vị từ nào trở lại sai. Phần dành riêng ổn định; dạng tương quan ra lệnh chuỗi được giữ gìn.
sort_heap	Loại một phạm vi. Loại ổn định; những phần tử bằng nhau là không phải được nuôi.

stable_partion	Trao đổi hai giá trị.
	Trao đổi những phần tử trong một phạm vi.
	Áp dụng một chức năng vào một phạm vi (của) những phần tử và được cất giữ trong một chuỗi mới.
stable_sort	Loại trừ những phần tử bản sao từ một phạm vi.
swap	Những sự tìm kiếm cuối cùng chỉ trong một chuỗi mà
swap_range	không là thú vị so với là giá trị nào đó.
transfrom	
unique	
unique_copy	
upper_bound	

EXAMPLES

*Two of the simplest algorithms are **count()** and **count_if()**. Their general forms are shown here:*

Hai trong số những giải thuật đơn giản nhất **count()** và **count_if()**. Những có dạng chung được cho thấy ở đây:

```
template<class Inter, class T>
 size_t count(Inter start, Inter end, const T & val);
template<class Inter, class T>
 size_t count(Inter start, Inter end, Unpred T & pfn);
```

*The **count()** algorithm returns the number of elements in the sequence beginning at start and ending at end that match val.*

Giải thuật **count()** trả lại số lượng phần tử trong chuỗi đang bắt đầu tại bắt đầu và kết thúc tại kết thúc mà phù hợp *val.*

*The **count_if()** algorithm returns the number of elements in the sequence beginning at start and ending at end for which the unary predicate pfn returns true.*

Giải thuật **count_if()** trả lại số lượng phần tử trong chuỗi bắt đầu tại *start* và kết thúc tại *end* vị từ đơn nguyên nào *pfn* những sự trở lại đúng.

*The following program demonstrates **count()** and **count_if().***

Chương trình sau đây trình diễn **count()** và **count _if().**

```
//Demonstrate count and count_if()
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
/* This is a unary predicate that determies if value
is even */
bool even (int x)
{
      return !(x%2);
}
int main()
{
vector<int> v;
int i;

for(i=0; i<20; i++)
    {
                if(i%2)
                     v.push_back(1);
                else
                     v.push_back(2);
    }
cout << "Sequence :";
for(i=0 ; i< v.size();i++)
                cout<< v[i] <<" ";
cout<<endl;

int n;
n= count(v.begin(), v.end(),1);
cout<<n<< "elements are 1 \n";

n= count_if(v.begin(), v.end(),even);
cout<<n<< "elements are even \n";

return 0;
}
```

This program dislays the following output:

Kết quả của chương trình là :

```
Sequence : 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
10 element are 1
10 element are even.
```

The program begin by creating a 20 –element vector that containrns alternating 1s and 2s. Next, ***count()*** *is used to count the number of 1s. Then,* ***cuont_if()*** *counts the number of elements that are even.Notice how the unary predicate* ***even()*** *is coded. All unary predicates receive as a parameter an object that is of the same type as that stored in the contianer upon which the predicate is operating. The predicate must then return a true or false result based upon this object.*

Chương trình bắt đầu bởi việc tạo ra một 20-vectơ phần tử mà containrns đang xen kẽ 1s và 2s. Rồi ,**cuont_if()** đếm số phần tử mà chẵn. Chú ý như thế nào vị từ đơn

nguyên **even()** được viết mã. Tất cả các vị từ đơn nguyên nhận được khi một tham số là một đối tượng mà cùng kiểu với điều đó cất giữ ở trên vị từ nào đang vận hành. Vị từ phải rồi trả lại một kết quả đúng hay sai được dựa trên đối tượng này.

*Sometimes it is useful to generate a new sequence that consists of only certain items from an original sequence. One algorithm that does this is **revome_copy()**. Its general forms is shown here:*

Đôi khi phát sinh một chuỗi mới mà gồm có những tiết mục nhất định duy nhất từ một chuỗi nguyên bản. Một giải thuật mà làm điều này**revome_copy()**. Những dạng chung những được cho thấy ở đây:

Template<class Inlter, class Outlter, class T>
Outlter revome_copy(Inter start. Inlter end, Outlter result,const T &val)

*The **remove-copy()** algorithm copies elements from the specified range that are equal to val and puts the result into the sequence pointed to by result. It returns an iterator to the end of the result. The output containner must be large enough to hold the result.*
*The following program demonstrates **revome_copy()**. It creates a sequence of 1s and 2s. it then revomes all the 1s from the sequence.*

Giải thuật **remove_copy()** sao chép những phần tử từ phạm vi được chỉ rõ mà bằng nhau tới *val* và đặt kết quả vào trong chuỗi được trỏ tại vị trí *result.* . Nó trả lại một iterator tới kết thúc kết quả. Đầu vào chứa phải đủ lớn để giữ kết quả.
Chương trình sau đây trình diễn **revome_copy()**. Nó tạo ra một chuỗi 1svà 2s. Rồi nó xoa tất cả các tử trong chuỗi 1s.

```
//Demonstrate revome_copy
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
vector<int> v, v2(20);
int i;

for(i=0; i<20; i++)
   {
          if(i%2)
               v.push_back(1);
          else
               v.push_back(2);
   }

cout<< "sequence :";
for(i=0; i<v.size(); i++)
          cout<< v[i] << " ";
cout<< endl;
```

```
//Revome 1s
remove_copy(v.begin(),v.end(), v2.begin(),1);
cout<< "Result: ";
for(i=0; i< v2.size(); i++)
          cout<< v2[i]<< " ";
cout<<endl<<endl;
return 0;
}
```

The output procduced by this program is shown here:

Kết quả của chương trình là:

```
Sequence:2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
Result:2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0
```

*An often useful algorithm is **reverse()**, which reveses a sequence. Its general form is*

Một giải thuật thường hữu ích **reverse()**, mà xoay một chuỗi. Dạng chung của nó là:
template<class Bilter> void reverse(Bilter *stasrt,*Bilter *end);*

*The **reverse()** algorithm reverses the order of the range specified by start and end.*

Giải thuật **reverse()** đảo ngược mệnh lệnh của phạm vi được chỉ rõ by *start* và *end.*

*The following program demonstrates **reverse():***

Kết quả của chương trình **revese()** là:

```
//Demonstrate reverse
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
         vector<int> v;
int i;

for(i=0; i<10; i++)
         v.push_back(i);

cout<< "Initial :";

for(i=0; i<v.size(); i++)
         cout<< v[i] << " ";
cout<< endl;
```

```
reverse(v.begin(), v.end());

cout<< "Reverse : ";
for(i=0; i< v.size(); i++)
         cout<< v[i]<< " ";

return 0;
}
```

The output from this program is shown here:

```
Initial: 0 1 2 3 4 5 6 7 8 9
Reverse: 9 8 7 6 5 4 3 2 1 0
```

*Once of the more interesting algorithms is **tramsform(),** which modified each element in a range according to a function that you provide. The **tramsform()** algorithm has these two general forms:*

Một trong những giải thuật thú vị hơn là **tramsform(),** phần tử được sửa đổi từng cái trong một phạm vi theo một chức năng mà bạn cung cấp. Giải thuật **tramsform()**có hai dạng chung này:

temp<class Inlter1, class Outlter, class Func) Outlter stransform(Inlter *srart,* Inlter *end,* Outlter *result*, Func *unaryfunc);*
temp<class Inlter1, class Inlter2, class Outlter, class Func) Outlter stransform(Inlter1 *srart1,* Inlter1 *end1,* Inlter2 *end1,* Inlter2 *start,* Outlter *result*, Func *unaryfunc);*

*The **transform()** algorithm applies a function to a range of elements and stores the outcome in result. In the first form, the range is specified by start and end. This function receives the value of an element in its parameter, and it must return the element's transformation. In the second form, the transformation is applied using a binary operator function that receives the value of an element form the sequence to be transformed in its first parameter. Both versions return an iterator to the end of the resulting the original sequence.*

Giải thuật **transform()** áp dụng một chức năng vào một phạm vi những phần tử và cất giữ hậu qủa trong *result.* Trong mẫu đầu tiên, phạm vi được chỉ rõ boiwr *srart* và *end.* Chức năng này nhận được giá trị một phần tử trong tham số của nó, và nó phải trả lại biến đổi của phần tử. Trong mẫu thứ hai, sự biến đổi được áp dụng sử dụng một chức năng toán tử nhị phân mà nhận được giá trị của một phần tử hình thành chuỗi sẽ được thay đổi trong tham số đầu tiên của nó. Cả hai phiên bản trả lại một iterator tới kết thúc kết quả chuỗi nguyên bản.

*The following program uses a simple transformation function called **xform()** to square the contents of a list. Notice that the resulting sequence is sroted in the same list that provided the original sequence.*

Chương trình sau đây sử dụng một chức năng biến đổi đơn giản được gọi là **xform()**

để làm nội dung của một danh sách. Chú ý chuỗi kết quả là sroted trong danh sách giống như vậy mà cung cấp chuỗi nguyên bản.

```
 //An example of the transform algorithm
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
// A simple transformation function
int xform(int i)
{
     return i*i;//square orginal value
}
int main()
{
list<int> x1;
int i;

//put value into list
for(i=0; i<10; i++)
        x1.push_back(i);

cout<< "orginal contents of x1 :";
list<int>::iterator p=x1.begin();

while (p!=x1.end())
  {
        cout<< *p << " ";
        p++;
  }

cout<< endl;
//transform x1
p=transform(x1.begin(),   x1.end(),x1.begin(),xform);

cout<< "Transformed contents of x1: ";
p=x1.begin();
while(p!=x1.end())
  {
        cout<< *p << " ";
        p++;
  }
return 0;
}
```

The output produced by the program is shown here:

Kết quả của chương trình là :

```
Orginal contents of x1: 0 1 2 3 4 5 6 7 8 9
Transformed contents of x1: 0 1 4 9 16 25 36 49 64 81
```

*as you can see, each element in the **x1** list has been squared.*

trong khi bạn có thể nhìn thấy, mỗi phần tử trong **x1** danh sách đã được làm vuông.

EXERCISES

*The **sort()** algorithm has these forms:*

Giải thuật **sort()**có những mẫu này:

template<class Randlter> void sort(Randlter *start*, Randlter *end);*
template<class Randlter *start,* class Comp>
void sort(Randlter *start,* Randlter *end, Comp cmpfn);*

*It sorts the range specified by start and end. The second form allows you to specify a comparison function that determines whether one element is less than another. Write a program that demonstrates **sort().** (Use either form you like.)*

Những loại phạm vi chỉ rõ bởi bắt đầu và kết thúc. Hai hình thành cho phép bạn chỉ rõ một hàm so sánh mà xác định liệu một phần tử là ít hơn phần tử khác. Viết một chương trình mà biểu diễu **sort().** (Sử dụng mọi mẫu bạn thích.)

*The **merge()** algorithm merges two ordered sequences and places the result into a third. One of its general forms is shown here:*

Giải thuật **merge()** hòa trộn hai chuỗi được ra lệnh và đặt kết quả vào trong một cơ mác trước. Một trong số những dạng chung nó được cho thấy ở đây:
template<class Inlter1, class Inlter2, class Outlter>
Outlter merge(Inlter1 start1, Inlter1 end, Inlter2 start2, Inlter2 end2, Outlter result);

The sequence to be meged are defined by start 1, end1 and start2, end2. The result is put into sequence pointed to by result. An iterator to the end of the reslting sequence is returned. Demonstrate this algorithm.

Chuỗi để là bắt đầu được định nghĩa gần bởi srart1, end1 và start2, end2. Kết quả được đặt vào trong chuỗi được trỏ vào bởi kết quả. Trình diễn giải thuật này.

14.6. THE STRING CLASS - Lớp Chuỗi

*As you know, c++ does not support a built-in string type, per se. it does, however, provide two ways to handle strings. First, you can use the traditional, null-terminated charater array with which you are already familiar. This is sometimes referred to as a string. The second method is to use a class object of type **string**. This is the approach that is examined here.*

Như bạn đã biết , C++ không có sự hỗ trợ một chuỗi định sẵn, nó làm.Tuy nhiên,

cung cấp hai cách để xử lý những chuỗi. Đầu tiên, bạn có thể sử dụng cách truyền thống, dùng mảng rỗng để chưa các ký tự với bạn đã quen thuộc. Đây thì đôi khi được tham chiếu tới như một chuỗi. Phương pháp thứ hai sẽ sử dụng một đối tượng lớp kiểu **string**. Điều này là cách tiếp cận mà được khảo sát ở đây.

*Actually, the **stirng** class is a specialization of a more general template class called **basic_string.** In fact, there are two specializations of **basic_string : string**, which supports wide characters strings in normal programming, **string** is the version of **basic_string** examined here.*

Thật sự,**stirng** lớp là một sự chuyên môn hóa một lớp khung mẫu chung hơn được gọi là **basic_string.** Thật ra,ở đó thì hai sự chuyên môn hóa của **basic_string: string**, mà hỗ trợ chuỗi những đặc tính rộng trong lập trình bình thường, **string** là phiên bản của **basic_string** khảo sát ở đây.

*Before you look at the **string** class, it is important that you understand why it is part of the C++ library. Standard classes have not been casually added to C++ . In fact, a significant amount of thought arrays, it migth at first seem that the inclusion of the **string** class in an exception to this rule. However, this is actually far from the truth. Here id why: Null-terminated strings cannot be manipulated by any of the standard C++ operators, nor can they take part in normal C++ expressions. For example. Consider this fragment:*

Trước đây bạn quan sát **string** trong lớp , quan trọng rằng bạn hiểu tại sao đó là phần thư viện C++ . Những lớp tiêu chuẩn không có trong C++ được thêm vào. Thật ra, một số lượng quan trọng mảng sự suy nghĩ, trước hết có vẻ sự bao gồm **string** phân loại trong một ngoại lệ tới quy tắc này. Tuy nhiên, đây thật sự xa với sự thật. Ở đây tại sao : những chuỗi được kết thúc bởi null không thể là thao tác bởi bất kỳ cái nào của thao tác thư viên chuẩn của C++, mà cũng không chúng có thể tham gia vào những biểu thức C++ bình thường. Chẳng hạn. Xem xét đoạn này:

```
char s1[80], s2[80], s3[80];
s1="one" ; //can't do
s2="two" ; //can't do
s3=s1+s2 ; // error, not allowed
```

As the comments show, in C++ it is possible to use the assignment operator to give a character array a new value (exept during initialization), nor is it possible to use the + operator to concatenate two strings. These operations must be written using library functions, as shown here:

Trong khi những bình luận hiện ra, trong C++ nó có thể xảy ra sử dụng thao tác thư viện ấn định để cho một đặc tính bày biện một giá trị mới (Trừ trong thời gian sự khởi tạo), mà cũng không nó có thể xảy ra để sử dụng +operator để ghép nối hai chuỗi. Những thao tác này phải được viết sử dụng những hàm thư viện chuẩn sau, như được cho thấy ở đây:

```
strcpy(s1 , "one");
strcpy(s1 , "two");
strcpy(s3, s1);
```

```
strcpy(s3, s2);
```

*Since null-terminated arrays are not technically data types in their own right, the C++ operators cannot be applied to them. This makes even the most rudimentary string operations clumsy. More than anything else, it is the inability to operator on null-terminated strings using the standard C++ operators that has driven the development of a standard **string** class. Remember, when you define a class in C++, you are defining an new data type that can be fully integrated into the C++ environment. This, of course, means that the operators can be overloaded relative to the new class. Therefore, with the addition of a standard **string** class, it becomes possible to manage strings in the same way that any other type of data is managed: through the use of operators.*

Một khi những mảng được kết thúc bởi null về mặt kỹ thuật là những kiểu dữ liệu trong quyền của mình, những thao tác thư viện C++ không thể được ứng dụng vào họ. Cái này làm cho thậm chí chuỗi sơ khai nhất là những thao tác vụng về. Hơn bất kỳ vật gì khác,đó là sự không có khả năng tới thao tác thư viện trên những chuỗi được kết thúc bởi null sử dụng những thao tác thư viện chuẩn C++ mà điều khiển sự phát triển một tiêu chuẩn lớp **string**. Nhớ, khi bạn định nghĩa một lớp trong C++, bạn phải định nghĩa một kiểu dữ liệu mới mà có thể hoàn toàn tổng hợp vào trong môi trường C++. Điều này, tất nhiên, có nghĩa rằng những thao tác viên có thể bị chất quá nặng tương đối tới lớp mới. Bởi vậy, với sự thêm một tiêu chuẩn lớp **string**, nó được trở nên có thể xảy ra để quản lý những chuỗi trong cách giống như vậy mà mọi kiểu khác dữ liệu được quản lý thông qua sự sử dụng những thao tác viên.

*There is, however, one other reason for the standard **string** class: safety. An inexperienced or careless programmer can very easily overrun the end of an array that holds a null-terminated string. For example, consider the standard string copy function **strcpy().** This function contains no provision for checking the boundary of the target array. If the source array contains more characters than the target array can hold, a program error of system crash is possible (likely) . As you will see, the standard **string** class prevents such errors.*

Vấn dề đó, tuy nhiên, một lý do khác tiêu chuẩn lớp **string**: sự an toàn. Một lập trình viên thiếu kinh nghiệm hay cẩu thả có thể rất dễ dàng tràn qua kết thúc một mảng mà giữ chuỗi được bởi NULL kết thúc mảng. Chẳng hạn, cho rằng chuỗi tiêu chuẩn sao chép chức năng **strcpy().** Chức năng này không chứa đựng sự chuẩn bị để kiểm tra ranh giới của mảng đích. . Nếu mảng nguồn chứa những đặc tính so với mảng đích có thể giữ(chiếm), một lỗi chương trình của sự va chạm hệ thống là có thể xảy ra (thích hợp).

*In the final analysis, there are three reasons for the inclusion of the standard **string** class :consistency (a string now defines a data type), convenience (you can use the standard C++ operators) and safety (array boundaries will not be overrun) . Keep in mind that there is no reason that you should abandon normal, null-terminated strings altogether. They are still the most efficient way*

*in which to implement strings. However, when speed is not an overloading cpncern, the new **string** class gives you access to a safe and fully integrated way to manage strings.*

*Although not traditionally thought of as part of the STL, **sting** is another container class defined by C++. This means that it supports the algorithms described in the previous section. However, strings have additional capabilities. To have access to the **string** class you must include <**string**> in your program.*

Mặc dù chưa theo truyền thống được nghĩ như phần STL, **sting** lớp chứa các ý được định nghĩa bởi C++. Cái này có nghĩa rằng nó hỗ trợ những giải thuật được mô tả trong mục trước đây. . Tuy nhiên, những chuỗi có những khả năng bổ sung. Để có sự truy nhập Tới **string** lớp bạn phải khai báo include<**string**> trong chương trình của bạn.

*The **string** class is very large, with many constructors and member functions. Also, many member functions have multiple overloaded forms. For this reason, it is not possible to look at the entire contents of **string** in this chapter. Instead, we will examine several of its most commonly used features. Once you have a general understanding of how **string** words, you will be able to easily explore the rest of it on your own.*

Lớp **string** thì rất lớn, với nhiều người xây dựng và thành viên vận hành. Đồng thời, nhiều thành viên những chức năng có nhiều quá tải trong những mẫu. Lý do này, có thể xảy ra không quan sát toàn bộ nội dung **string** trong chương này. Thay vào đó, chúng tôi sẽ khảo sát riêng biệt những đặc tính thường sử dụng nhất của nó. Một lần bạn cho phép một tổng quan hiểu như thế nào những từ trong **string**, bạn sẽ có khả năng để dễ dàng tự ý khám phá phần còn lại của nó.

*The **string** class support several constructors. The prototypes for three of its most commonly used constructors are shown here.*

Lớp **string** hỗ trợ cho người xây dựng. Những nguyên mẫu ba trong số những người xây dựng thường sử dụng nhất được cho thấy ở đây.

```
string();
string(const char *str);
string(const string & str);
```

*the first form create an empty **string** object. The second creates a **string** object from the null-terminated string pointed to by str. This form provides a conversion from null-terminated strings to **string** object. The third form creates a **string** object from another **string** object.*

đầu tiên hình thành tạo ra một làm rỗng **string** đối tượng. Hai tạo ra một **string** đối tượng từ chuỗi được kết thúc bởi giá trị NULL được trỏ tại *str*. Mẫu này cung cấp một sự chuyển đổi từ những chuỗi được kết thúc bởi số không Tới **string** đối tượng. Mẫu thứ ba tạo ra một **string** đối tượng từ **string** đối tượng khác.

*A number of operators that apply to strings are defined for **string** object, including those listed in the following table:*

Một số thao tác viên mà ứng dụng vào những chuỗi được định nghĩa Cho **string** đối tượng, bao gồm được vào danh sách đó trong bảng sau đây:

Operator	**Meaning**
=	Assignment
+	Concatenation
+=	Concatenation assignment
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
[]	Subscripting
≪	Output
≫	Input

*These operator allow the use of **string** object in normal expressions and eliminate the need for calls to functions such as **strcpy()** or **strcat()**, for example. In general, you can mix **string** objects with normal, null-terminated strings in expressions. For example, a **string** object can be assigned a null-terminated string.*

Thao tác viên này cho phép sự sử dụng **string** phản đối trong những biểu thức bình thường và loại trừ nhu cầu gọi để vận hành như **strcpy()** hay **strcat()**, chẳng hạn. Nói chung, bạn có thể pha trộn **string** những đối tượng với những chuỗi bình thường, được kết thúc bởi số không trong những biểu thức. Chẳng hạn, một **string** đối tượng có thể được gán một giá trị NULL kết thúc chuỗi.

*The + operator can also be used to concatenate a **string** object with another **string** object or a **string** object with C-style string. That is, the following variations are supported:*

Toán tử + có thể cũng được quen đa hệ ghép một **string** đối tượng với **string** đối tượng khác hay một **string** đối tượng với kiểu chuỗi C. Nghĩa là, những sự biến đổi sau đây được hỗ trợ:

string + string
string + C-string
C-string + string

The + operator can also be used to concatenate a character onto the end of a string.

Toán tử + có thể cũng được dùng để ghép nối một đặc tính lên trên kết thúc của một chuỗi.

*The **string** class defines the constant **npos,** which is usually-1. This constant represents the length of the longest possible string.*

Lớp **string** định nghĩa hằng số **npos,** mà thông thường là-1. Hằng số này đại diện cho chiều dài của chuỗi có thể xảy ra dài nhất.

*Although most simple string operations can be accomplished with **string** clsas member functions. Although there are far too many to discuss in this chapter, we will examine several of the most common ones. To assign one string to another, use the **assign()** function. Two of its forms are shown here:*

Mặc dù đa số những thao tác chuỗi đơn giản có thể được hoàn thành Với **string** những chức năng thành viên của lớp. Mặc dù có cũng có nhiều người tranh luận trong chương này, chúng tôi sẽ khảo sát riêng biệt một cách chung nhất. Để gán một chuỗi tới chuỗi khác, sử dụng chức năng **assign()**. Hai trong số những mẫu nó là chỉ ra ở đây:

string&assign(const string &strob, size_type start, size_type num);
*string &assign(const char *str, size_type num);*

*In the first form, num characters from strob beginning at the index specified by start will be assigned to the invoking object. In the second form **insert()** inserts num characters from strob. Beginning at inStart into the invoking string at the index specified by Start .*

Trong mẫu đầu tiên,*num* những đặc tính đến từ *strob* bắt đầu tại chỉ số được chỉ rõ bởi star*t* sẽ là gán tới việc kéo theo đối tượng. Trong trường hợp thứ hai **insert()** những sự chèn *num* những đặc tính đến từ *strob.* Bắt đầu tại *inStart* vào trong kéo theo chuỗi tại chỉ số được chỉ rõ bởi *start*.

*Beginning at start, the first form of **replace( )** replaces num characters from the invoking string, with strob. The second form replaces orgNum characters, beginning at start, in the invoking string with replaceNum characters from the string specified by strob, beginning at replaceStart. In both cases, a reference to the invoking object is returned.*

Bắt đầu tại *tsart*, đầu tiên hình thành **replace()**thay thế *num* những đặc tính đến từ kéo theo chuỗi, với *strob*. Hai hình thành thay thế*orgNum*những đặc tính, sự bắt đầu tại S*bánh nhân mứt*, trong kéo theo chuỗi với *replaceNum* những đặc tính từ chuỗi được chỉ rõ gần *strob*, bắt đầu tại *replaceStart.* Trong cả hai trường hợp, một sự tham khảo tới việc kéo theo đối tượng được trở lại.

*You can append part of one string to another using the **append()** member function. Two of its forms are shown here:*

Bạn có thể nối vào bộ phận của một chuỗi tới khác sử dụng **append()** chức năng thành viên. Hai trong số những mẫu nó được cho thấy ở đây

string &erase(size_type *start* = 0, size_type *num* = npos);

It removes num characters from the invoking string, beginning at start. A reference to the invoking string is returned.

Nó loại bỏ *num* những đặc tính đến từ kéo theo chuỗi, bắt đầu tại *start*. Một sự

tham khảo tới việc kéo theo chuỗi được trở lại.

*The **string** class provides several member functions that search a string, including **find( )** and **rfind( )**. Here are the prototypes for the most common versions of these functions:*

Lớp **string** cung cấp vài chức năng thành viên mà tìm kiếm một chuỗi, bao gồm **find()** và **rfind()**. ở đây những nguyên mẫu chung nhất là những phiên bản đây những chức năng:

size_type find(const string &strob, size_type start=0) const;
size_type rfind(const string &strob, size_type start=npos) const;

*Beginning at start, **find( )** searches the invoking string for the first occurrence of the string contained in strob. If found, **find( )** returns the index at which the match occurs within the invoking string. If no match is found, then **npos** is returned. **rfind( )** is the opposite of **find( )**. Beginning at start, it searches the invoking string in the reverse direction for the first occurrence of the string contained in strob (i.e., it finds the last occurrence of strob within the invoking string). If found, **rfind( )** returns the index at which the match occurs within the invoking string. If no match is found, **npos** is returned.*

Bắt đầu tại *start*, **find()** những sự tìm kiếm kéo theo chuỗi biến cố đầu tiên của chuỗi được chứa đựng trong *strob*. Nếu tìm thấy,**find()** trả lại chỉ số mà trận đấu xuất hiện bên trong việc kéo theo chuỗi. Nếu không có trận đấu được tìm thấy, rồi **npos** được trở lại **rfind()** đối diện của **find()**. Bắt đầu tại *start*, nó tìm kiếm kéo theo chuỗi trong sự nghịch đảo cho biến cố đầu tiên của chuỗi được chứa đựng trong *strob* (thí dụ, nó tìm thấy cuối cùng biến cố *strob* bên trong việc kéo theo chuỗi). Nếu tìm thấy, **rfind()** trả lại chỉ số tại trận đấu nào xuất hiện bên trong việc kéo theo chuỗi. Nếu không có trận đấu được tìm thấy,**npos** trở lại.

*To compare the entire contents of one string object with another, you will normally use the overloaded relational operators, described earlier. However, if you want to compare a portion of one string with another, then you will need to use the **compare( )** member function, shown here:*

Tới sự so sánh nguyên bằng nhau của một đối tượng chuỗi với nhau, bạn có thể định bình thường sử dụng những toán tử quan hệ toán tử chồng, được mô tả trước đó. Tuy nhiên, nếu bạn muốn so sánh một phần của một chuỗi, rồi chuỗi với chuỗi khác sẽ cần với sự sử dụng **compare ()**thành viên vận hành, cho thấy ở đây:

int compare(size_type start, size_type num, const string &strob) const;

*Here, num characters in strob, beginning at start, will be compared against the invoking string. If the invoking string is less than strob, **compare( )** will return less than 0. If the invoking string is greater than strob, it will return greater than 0. If strob is equal to the invoking string, **compare( )** will return 0.*

Ở đây,*num* những đặc tính trong *strob*, bắt đầu tại *start*, sẽ là so sánh g lại việc kéo theo chuỗi. Nếu việc kéo theo chuỗi ít hơn so với *strob*, **compare()**sẽ trở lại ít so với 0. Nếu Việc kéo theo chuỗi (thì) lớn hơn so với *strob*, ý định trả lại

lớn hơn so với0. Nếu *strob* bằng nhau tới việc kéo theo chuỗi,**compare()**sẽ trở lại 0.

*Although **string** objects are useful in their own right, there will be times when you will need to obtain a null-terminated character-array version of the string. For example, you might use a **string** object to construct a file name. However, when opening a file, you will need to specify a pointer to a standard, null-terminated string. To solve this problem, the member function **c_str( )** is provided. Its prototype is shown here:*

Mặc dù **string** những đối tượng hữu ích trong quyền của mình, sẽ có những lần khi bạn sẽ cần thu được một giá trị NULL - đặc tính được kết thúc- phiên bản mảng chuỗi. Ví dụ, bạn có lẽ đã sử dụng một **string** đối tượng để xây dựng một tên tập tin. Tuy nhiên, khi mở một hồ sơ, bạn sẽ cần chỉ rõ một con trỏ tới một tiêu chuẩn, được kết thúc bởi số không chuỗi. Để giải quyết vấn đề này, chức năng thành viên **c_str()** được cung cấp. Nguyên mẫu của nó được cho thấy ở đây:

*const char *c_str( ) const;*

*This function returns a pointer to a null-terminated version of the string contained in the invoking **string** object. The null-terminated string must not be altered. It is also not guaranteed to be valid after any other operations have taken place on the **string** object.*

Chức năng này trả lại một con trỏ cho một phiên bản được kết thúc bởi số 0 của chuỗi được chứa đựng Trong kéo theo **string** đối tượng. Chuỗi được kết thúc bởi số 0 thì không phải biến đổi. Nó cũng không phải bảo đảm để hợp lệ sau khi mọi thao tác khác đã tiếp tục xảy ra **string** đối tượng.

*Because it is a container, **string** support the **begin()** and **end()** functions that return an iterator to the start and end of string, respectively. Also included is the **zise()** function, which returns the munber of charaters currently in a string.*

Bởi vì đó là một chứa, **string** sự hỗ trợ **begin()** và **end()** những chức năng mà trả lại một iterator cho bắt đầu và kết thúc chuỗi, tương ứng. Cũng được bao gồm**Z ()** chức năng, mà trả lại số tử hiện thời trong một chuỗi.

EXAMPLES

*Although the traditional, C-style strings have always been simple to use, the C++ string classes make string handling extraordinarily easy. For example, by using **string** objects, you can use the assignment operator to assign a quoted string to a **string**, the + operator to concatenate strings, and the comparison operators to compare strings. The following program illustrates these operations:*

Mặc dù t ruyền thống,C- Kiểu những chuỗi có luôn luôn đơn giản tới sự sử dụng, tông ta chuỗi C++ những lớp làm sự dùng chuỗi dễ dàng lạ thường. . Chẳng hạn,

gần sử dụng **string** những đối tượng, bạn có thể sử dụng thao tác viên ấn định để gán một chuỗi được báo giá tới một **string**,toán +operator thao tác viên tới những chuỗi dạng chuỗi, và những toán tử so sánh tới những chuỗi so sánh. Chương trình sau đây minh họa những thao tác:

```
// A short string demonstration.
#include <iostream>
#include <string>
using namespace std;
int main()
{
     string str1("The string class gives ");
     string str2("C++ high-powered string
     handling.");
     string str3;
     // assign a string
     str3 = str1;
     cout << str1 << "\n" << str3 << "\n";
     // concatenate two strings
     str3 = str1 + str2;
     cout << str3 << "\n";
     // compare strings
     if(str3 > str1) cout << "str3 > str1\n";
     if(str3 == str1+str2)
     cout << "str3 == str1+str2\n";
     /* A string object can also be
     assigned a normal string. */
     str1 = "This is a null-terminated string.\n";
     cout << str1;
     // create a string object using another
     string object
     string str4(str1);
     cout << str4;
     // input a string
     cout << "Enter a string: ";
     cin >> str4;
     cout << str4;
     return 0;
}
This program produces the following output:
The string class gives
The string class gives
The string class gives C++ high-powered string
handling.
str3 > str1
str3 == str1+str2
This is a null-terminated string.
```

```
This is a null-terminated string.
Enter a string: Hello
Hello
```

*As you see, object of type **string** can be manipulated with techniques similar to those used to word with C++'s built-class data types. In fact, this is the main advantage to the string class.*

Trong khi bạn nhìn thấy, đối tượng kiểu **string** có thể được thao tác với kỹ thuật tương tự như người (vật) được dùng tới từ với C++' xây dựng- những kiểu dữ liệu lớp. Trong thực tế điều này là lợi thế chính tới lớp chuỗi.

*Notice the ease with which the string handling is accomplished. For example, the + is used to concatenate strings, and the **>** is used to compare two strings. To accomplish these operations using C-style, null-terminated strings, less convenient calls to the **strcat( )** and **strcmp( )** functions would be required. Because C++ **string** objects can be freely mixed with C-style strings, there is no disadvantage to using them in your program - and there are considerable benefits to be gained.*

Chú ý việc sự dễ dàng mà với chuỗi xử lý được hoàn thành. Chẳng hạn, + thường ghép nối những chuỗi, và > được quen sự so sánh hai chuỗi. Hoàn thành những thao tác này đang sử dụng C- kiểu, những chuỗi được kết thúc bởi số không, những sự gọi ít tiện lợi hơn tới **strcat()** và **strcmp()** những chức năng được yêu cầu. Vì C++ **string** những đối tượng còn tùy thích pha trộn với những chuỗi kiểu C, không có sự bất lợi để sử dụng họ trong chương trình (của) các bạn - và có những lợi ích đáng kể sẽ được kiếm được.

*There is one other thing to notice in the preceding program: The size of the strings are not specified. **string** objects are automatically sized to hold the string that they are given. Thus, when assigning or concatenating strings, the target string will grow as needed to accommodate the size of the new string. It is not possible to overrun the end of the string. This dynamic aspect of **string** objects is one of the ways that they are better than null-terminated strings (which are subject to boundary overruns).*

Có một thứ khác để chú ý trong chương trình có trước: kích thước (của) những chuỗi không được chỉ rõ. **String** những đối tượng tự động được đo cỡ để giữ chuỗi điều đó họ được cho. Như vậy, khi sự chỉ định (nhượng lại) hay việc ghép nối những chuỗi, ý định chuỗi đích lớn lên chuỗi như được cần để điều tiết kích thước mới. Nó không có thể xảy ra đối với sự tràn qua kết thúc của chuỗi. Khía cạnh động này của **string** những đối tượng là một trong số những cách điều đó họ tốt hơn hơn những chuỗi được kết thúc bởi số không (màchinh phục tới ranh giới những sự tràn qua).

*The following program demonstrates the **insert( )**, **erase( )**, and **replace( )** functions:*

Chương trình sau đây trình diễn **insert()**, **erase()**, và **replace( )** những chức năng:

```
// Demonstrate insert(), erase(), and replace().
#include <iostream>
#include <string>
using namespace std;
int main()
{
     string str1("This is a test");
     string str2("ABCDEFG");
     cout << "Initial strings:\n";
     cout << "str1: " << str1 << endl;
     cout << "str2: " << str2 << "\n\n";
     // demonstrate insert()
     cout << "Insert str2 into str1:\n";
     str1.insert(5, str2);
     cout << str1 << "\n\n";
     // demonstrate erase()
     cout << "Remove 7 characters from str1:\n";
     str1.erase(5, 7);
     cout << str1 <<"\n\n";
     // demonstrate replace
     cout << "Replace 2 characters in str1 with
     str2:\n";
     str1.replace(5, 2, str2);
     cout << str1 << endl;
     return 0;
}
```

The output produced by this program is shown here:

Đầu rao được sản xuất bởi chương trình này được cho thấy ở đây

```
Initial strings:
str1: This is a test
str2: ABCDEFG
Insert str2 into str1:
This ABCDEFGis a test
Remove 7 characters from str1:
This is a test
Replace 2 characters in str1 with str2:
This ABCDEFG a test
```

*Since **string** defines a data type, it is possible to create containers that hold objects of type **string**. For example, here is a better way to write the dictionary program shown earlier:*

Từ đó định nghĩa **string** một dữ liệu đánh máy, có thể xảy ra tạo ra những côngtenơ mà giữ những đối tượng của kiểu **string**. Chẳng hạn,ở đây là một cách tốt hơn hơn để viết chương trình từ điển được cho thấy trước đó :

```
// Use a map of strings to create a dictionary.
```

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main()
{
     map<string, string> dictionary;
     dictionary.insert(pair<string,
     string>("house",
     "A place of dwelling."));
     dictionary.insert(pair<string,
     string>("keyboard",
     "An input device."));
     dictionary.insert(pair<string,
     string>("programming",
     "The act of writing a program."));
     dictionary.insert(pair<string, string>("STL",
     "Standard Template Library"));
     string s;
     cout << "Enter word: ";
     cin >> s;
     map<string, string>::iterator p;
     p = dictionary.find(s);
     if(p != dictionary.end())
     cout << "Definition: " << p->second;
     else
     cout << "Word not in dictionary.\n";
     return 0;
}
```

EXERCISES:

*Using object of type **string**, store the following string in **a list**.*

Sử dụng đối tượng của kiểu **string**, cất giữ chuỗi sau đây trong một **list**

one	two	three	four	five
six	seven	eight	nine	ten

Next, sort the list. Finally, display the sorted list.

Tiếp theo, phân loại danh sách. Cuối cùng, trình bày danh sách được phân loại.

*Since **string** is container, it can be used with the standard algorithm. Create a program that inputs a string from the user. Then, using **count()**, report how many e's are in the string.*

Từ đó **string** là chứa, nó có thể được sử dụng với giải thuật tiêu chuẩn. Tạo ra một

chương trình mà nhập vào một chuỗi từ người sử dụng. Rồi, sử dụng **count()**, báo cáo được nhiều người dùng trong chuỗi.

*Modify your answer to Exercise 2 so that it report the number of characters that are lowercase. (Hint :use **count-if()**).*

Sửa đổi những câu trả lời các bạn ở bài tập 2 sao cho nó báo cáo số đặc tính mà thường. (Gợi ý: sự sử dụng **count- nếu ()**).

*The **string** class is a specialization of what template class ?*

Lớp **string** là một chuyên môn hóa của lớp khung mẫu nào?

SKILLS CHECK
KIỂM TRA KỸ NĂNG

Mastery Skills Check
Kiểm trả kỹ năng tổng hợp

At this point you should be able to perform the following exercises and answer the questions.

Tại điểm này bạn cần phải có khả năng để thực hiện những bài tập dưới và trả lời những câu hỏi.

How does the STL make it easier for you to create more reliable programs?

Hãy làm với STL làm nó dễ dàng hơn cho bạn để tạo ra những chương trình đáng tin cậy hơn?

Define a container, an iterator, and a algorithm as they relate to the STL

Định nghĩa một cái chứa, một iterator, và một thuật toán liên quan đến STL.

Write a program that creates a ten-element vector that contains the numbers 1 through 10. Next, copy only the even elements from this vector into a list.

Viết một chương trình mà tạo ra một vectơ chứa 10 phần tử mà chứa đựng những số từ 1 đến 10. Tiếp theo, chỉ sao chép những phần tử chẵn từ vectơ này vào trong một danh sách.

*Give one advantage of using the **string** data type. Give one disadvantage.*

Cho một lợi thế sử dụng kiểu dữ liệu **string**. Đưa cho một sự bất lợi.

What is a predicate?

Một vị từ là gì?

*On your own, recode your answer to Exercise 2 in Section 14.5 so that is uses **string**.*

Tự ý các bạn, recode câu trả lời các bạn để Baif tập 2 trong Mục 14.5 vì vậy điều đó là những sự sử dụng **string**.

*Begin exploring the STL function objects. To get started, examine the two standard classes **unary_function** and **binary_function**, which aid in the construction of function objects.*

Bắt đầu việc khám phá những đối tượng chức năng STL. Để bắt đầu bắt đầu, khảo sát những hai lớp tiêu chuẩn**unary_function** và**binary_function**, sự giúp đỡ nào trong những đối tượng xây dựng một hàm.

Study the STL documentation that comes with your compiler. You will find several interesting features and techniques.

Học tài liệu STL mà đến với người biên tập các bạn. Bạn sẽ tìm thấy vài đặc tính thú vị và kỹ thuật.

Mastery Skills Check
Kiểm trả kỹ năng tổng hợp

This section checks how well you have integrated material in this chapter with that from the preceding chapters.

Mục này kiểm tra bạn hiểu được bảng chất tổng hợp trong chương này với điều đó từ những chương có trước tốt ra sao.

You have come a long way since Chapter 1. Take some time to skim through the book again. As you do so, think about ways you can improve the examples (especially those in the first six chapters) so that they take advantage of all the features of C++ you have learned.

Bạn có đến là một đường dài từ Chương 1. Đưa một lúc nào đó đi sự lướt qua thông qua sách lần nữa. Như bạn làm vì thế, nghĩ về những cách bạn có thể cải thiện những ví dụ (Một cách đặc biệt là người trong những sáu chương đầu tiên) Vì thế mà chúng tận dụng tất cả những đặc tính) C++ bạn được học.

Programming is learned best by doing. Write many C++ programs. Try to exercise those features of C++ that are unique to it.

Chương trình được học tốt nhất phải làm. Viết nhiều chương trình C++. Thử để luyện tập những đặc tính đó C++ mà duy nhất đối với nó.

Continue to explore the STL. In the future, many programming tasks will be framed in terms of the STL because often a program can be reduced to manipulations of containers by algorithms.

Tiếp tục tìm hiểu trong STL. Việc trong tương lai, nhiều người lập trình giao nhiệm vụ sẽ được kết cấu dưới dạng STL bởi vì thường một chương trình có thể được giảm bớt tới những sự thao tác chứa những giải thuật.

Finally, remember : C++ gives you unprecedented power. It is important that you learn to use this power wisely. Because of this power, C++ lets you push the limits of your programming ability. However, if this power is misused, you can also create programs that are hard to understand, nearly impossible to follow, and extremely difficult to maintain. C++ is a powerful tool. But, like any other tool, it is only as good as the person using it.

Cuối cùng, nhớ: C++ đưa cho bạn sức mạnh chưa hề thấy. Quan trọng rằng bạn học sức mạnh này một cách khôn ngoan. Bởi vì sức mạnh này, trong C++ để cho bạn đầy đủ những khả năng lập trình. Tuy nhiên , nếu sức mạnh này được sử dụng, bạn có thể cũng tạo ra những chương trình mà sẽ khó khăn hơn, gần không thể nào đi theo sau, và vô cùng khó tới phần chủ yếu. C++ là một công cụ mạnh. Nhưng, cũng như mọi công cụ khác, đó chỉ là lợi ích như người đang sử dụng nó.

A

A Few More Between C and C

For the most part, C++ is as superset of ANSI-standard C, and virtually all C program are also C++ program. However, a few differnces do exist, sevaral of which were mentioned in Chapter 1.Here are a few more that you should be aware of:

Đa số phần, C++ như là như tập hợp của C ANSI- chuẩn, và thực tế tất cả C lập trình là chương trình C++ nữa. Tuy nhiên, vài differnces tồn tại, sevaral (của) mà được đề cập trong Chương 1.ở đâyare vài nữa bạn cần phải (thì) ý thức (của) :

▼ *A small but potentially important difference between C and C++ is that in C, a character constant is automatically elevated to an integer, whereas in C++, it is not.*

Một Nhỏ mà tiềm tàng quan trọng khác biệt giữa C và C++ là trong C, một

hằng số đặc tính tự động được nâng lên tới một số nguyên, trong khi mà trong C++, nó không tự động tăng.

▼ *In C, it is not an error to declare a global varible sveral times, even though it is bad programming practice. In C++, this is an error.*

Trong C, đó không là một lỗi để khai báo những một lần varible sveral toàn cầu, mặc dù đó là thực hành lập trình xấu. Trong C++, đây là một lỗi.

▼ *In C, an identifier will have at least 31 significant characters. In C++, all characters are considered signnficant. However, from a practical point of view, extremely long identifiers are unwieldy and are seldom needed.*

Trong C, một định danh sẽ có ít nhất 31 những đặc tính quan trọng. Trong C ++, tất cả các đặc tính đều là đặt tính được xem xét. Tuy nhiên, từ một điểm thực hành của đặt tính, từ những định danh thì cồng kềnh và được cần ít khi.

▼ *In C, you can call **main()** from within a program, although this would be unusual. In C++, this is not allowed.*

Trong C, bạn có thể gọi **main()** từ bên trong một chương trình, mặc dù điều này thì khác thường. Trong C++, đây không được cho phép.

▼ *In C, you cannot take the address of a **register** variable. In C++, you can.*

Trong C, bạn không thể cầm lấy địa chỉ biến của một **thanh ghi**. Trong C++, bạn có thể.

▼ *In C, the type **wchar_t** is defined with a **typedef.** In C++, **wchar_t** is a keyword.*

Trong C, kiểu **wchar_t** được định nghĩa với một **typedef.** Trong C++, **wchar_t** là một từ khóa.

********** HẾT **********