ĐẶNG NGỌC HOÀNG THÀNH

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

ĐẶNG NGỌC HOÀNG THÀNH

C++

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Tài liệu học tập

PHỤ LỤC

GIỚI THIỆU

1. Cấu trúc của giáo trình

Giáo trình được chia ra làm 17 chương và mỗi chương được chia làm các mục khác nhau. Các chương được sắp xếp theo trình tự từ lập trình hướng thủ tục trên C++ đến lập trình hướng đối tượng và các lớp thư viện cơ bản. Độc giả có thể truy cập vào mục bất kì từ phần phụ lục nằm đầu sách. Nhiều mục bao gồm các ví dụ để mô tả cách sử dụng. Tôi khuyên các bạn nên đọc các ví dụ này và có thể hiểu mỗi đoạn mã chương trình trước khi đọc chương tiếp theo.

Một cách thức tốt để tăng lượng kiến thức nhận được đó là hãy chỉnh sửa, bổ sung mã lệnh mới dựa trên ví dụ mẫu, theo hướng tư duy của của bản thân, để từ đó có thể hiểu một cách đầy đủ về nội dung mà ta tiếp thu được.

Sau khi đọc xong giáo trình, tôi còn cung cấp một số bài tập thực hành đề nghị để độc giả nên thử nghiệm. Hãy giải những bài tập này, chúng sẽ rất hữu ích và giúp các bạn cũng cố lại kiến thức môn học cũng như hiểu sâu sắc hơn phần lý thuyết.

Một điều nữa mà độc giả cần lưu ý: hãy đọc trang cuối cùng của cuốn sách, để nắm được một số thuật ngữ anh-việt tương ứng được sử dụng trong giáo trình này. Tôi cũng có gắng sử dụng tên gọi phù hợp nhất với đại đa số các giáo trình hiện hành. Tuy nhiên, độc giả cũng nên nắm các thuật ngữ tiếng anh tương ứng, để có thể tham khảo thêm các tài liệu chuyên môn tiếng anh.

Khi viết giáo trình này, tôi không thể tránh khỏi sai sót. Rất mong sự đóng góp ý kiến quý báu của các bạn độc giả cũng như các bạn đồng nghiệp. Mọi sự đóng góp xin liên hệ theo địa chỉ email: *dnhthanh@hueic.edu.vn*. Hi vọng với các ý kiến đóng góp của các bạn, giáo trình này sẽ ngày càng hoàn thiện hơn.

2. Một vài chú ý về sự tương thích của C và C++

Chuẩn ANSI-C++ được một tổ chức tiêu chuẩn quốc tế thống nhất đưa ra. Nó được chính thức ra mắt vào tháng 11 năm 1997 và duyệt lại vào

năm 2003. Tuy nhiên, ngôn ngữ C++ đã tồn tại trước đó một thời gian khá dài (vào năm 1980). Trước đó, có rất nhiều trình dịch không hỗ trợ các tính năng mới bao gồm cả chuẩn ANSI-C++. Giáo trình này được xây dựng trên các chương trình dịch hiện đại hỗ trợ đầy đủ chuẩn ANSI-C++. Tôi đảm bảo rằng các ví dụ sẽ hoạt động tốt nếu độc giả sử dụng một trình dịch hỗ trợ ANSI-C++. Có nhiều sự chọn lựa, có thể là miễn phí hoặc các phần mềm thương mại. Trong giáo trình này, tôi giới thiệu đến các các bạn hai công cụ biên dịch C++ là GCC MinGW – miễn phí và Visual C++ - thương mại.

3. Trình biên dịch

Các ví dụ trong cuốn giáo trình này được xây dựng chủ yếu trên chế độ console (màn hình DOS). Điều đó có nghĩa là nó sử dụng chế độ văn bản để hiển thị các kết quả. Mọi trình dịch C++ đều hỗ trợ chế độ dịch console. Với một môi trường phát triển tích hợp IDE cho C++ miễn phí, chúng ta có thể sử dụng chương trình Codeblocks hoặc Eclipse. Chúng là các môi trường phát triển tích hợp hỗ trợ soạn thảo và biên dịch C++. Chúng hỗ trợ môi trường GCC để biên dịch cả C và C++. Với CodeBlocks, chúng ta có thể tải phần mềm tại địa chỉ http://www.codeblocks.org/downloads. Đối với Eclipse, nó là một trình soạn thảo và biên dịch ngôn ngữ lập trình chuyên nghiệp nhưng hoàn toàn miễn phí (vì ta có thể cấu hình kết hợp với các công cụ biên dịch khác nhau để tạo ra môi trường phát triển tích hợp cho các ngôn ngữ lập trình khác nhau). Chúng ta có thể dùng nó để soạn thảo và biên dịch Java, PHP, JSP, Python… và hiển nhiên là cả C/C++. Đây là một dự án mã nguồn mở, tiêu tốn hàng triệu đôla của IBM. Để tải về bản mới nhất cho đến thời điểm này (năm 2010) là Eclipse Helios, ta có thể truy cập đến địa chỉ bên dưới[1]. Đối với Eclipse, chúng ta nên sử dụng kết hợp với trình biên dịch C++ là MinGW, nó cũng là một dự án mở. Chúng ta có thể tải về tại địa chỉ bên dưới[2]. Với Eclipse, thì công việc cấu hình ban đầu tương đối phức tạp. Nhưng nó là một trình soạn thảo tuyệt vời. Ta có thể sử dụng nó để soạn thảo nhiều ngôn ngữ lập trình bằng cách cài đặt thêm plugin hỗ trợ. Nhiều nhà phát triển đã sử dụng Eclipse làm nền tải

[1] http://ftp.jaist.ac.jp/pub/eclipse/technology/epp/downloads/release/helios/R/eclipse-cpp-helios-win32.zip
[2] http://nchc.dl.sourceforge.net/project/mingw/Automated%20MinGW%20Installer/mingw-get-inst/mingw-get-inst-20100831/mingw-get-inst-20100831.exe

cho việc phát triển các ứng dụng của mình: Embarcadero sử dụng nó để phát triển JBuider, Adobe sử dụng nó để phát triển Flash Buider và rất nhiều các hãng phần mềm nổi tiếng khác.

Nếu là một lập trình viên Java, Eclipse là một sự lựa chọn không thể bỏ qua. Nếu phát triển Flash theo dự án mã nguồn mở từ Adobe, Eclipse cũng là sự lựa chọn hoàn hảo. Nếu phát triển C/C++, với các trình soạn thảo thì Eclipse cũng là sự lựa chọn không tồi. Việc sử dụng thành thạo Eclipse sẽ là một lợi thế cho chúng ta khi tiến hành nghiên cứu Java, lập trình Web, Flex, Python… sau này.

Bên cạnh đó, chúng tôi cũng giới thiệu môi trường phát triển tích hợp IDE Microsoft Visual Studio. Đây là trình biên dịch thương mại và là trình biên dịch chuyên nghiệp và nổi tiếng nhất trên hệ điều hành Windows. Ta có thể sử dụng để phát triển các ứng dụng trên nền NET hoặc các ứng dụng Win32. Nếu muốn phát triển các ứng dụng theo hướng của Microsoft, ta nên sử dụng Visual Studio. Phiên bản mới nhất đến thời điểm này là VS 2010. Nhưng cần lưu ý rằng, khi nghiên cứu Visual C++, hãy chọn lựa phiên bản dành cho Win32 mà không phải là ứng dụng CLI (common language infrastructure) bởi nó được phát triển trên nền NET. Và Visual C++ for NET có một số khác biệt so với Visual C++ for Win32.

MÔI TRƯỜNG PHÁT TRIỂN TÍCH HỢP IDE

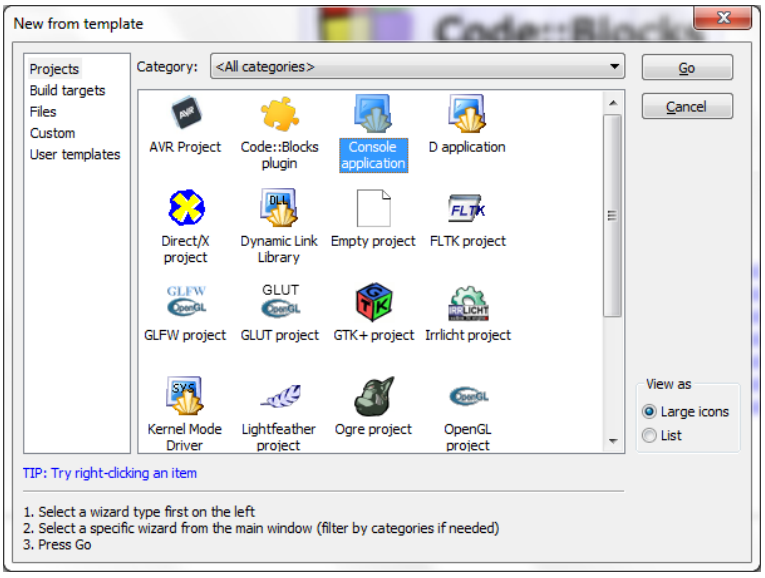
1. CodeBlocks

Trước tiên, chúng ta sẽ tìm hiểu cách tạo dự án, biên dịch một tập tin C++ trên CodeBlocks. Độc giả cũng cần lưu ý rằng, CodeBlocks tổ chức công việc theo các dự án. Chúng ta có thể biên dịch từng tập tin cpp một cách đơn lẻ. Tuy nhiên, làm việc theo dự án sẽ giúp ích cho chúng ta rất nhiều khi làm việc với những tác vụ lớn.

Đầu tiên chúng ta khởi động codeblocks, sau đó vào File > New > Project.

Trong hộp thoại hiện ra, chúng ta chọn **console application** (Hình 1).

Và nhấp Go, sau đó nhấp Next. Trong hộp thoại tiếp theo, ta chọn C++ và nhấp Next.



Hình 1 – Tạo mới dự án trong CodeBlocks

Hộp thoại yêu cầu điền thông tin về dự án sẽ xuất hiện. Hãy điền tên dự án, vị trí lưu trữ dự án. Sau đó nhấp Next. Cuối cùng nhấp Finish.

Trong cửa sổ quản lý dự án, ta nhấp đôi chuột vào tệp main.cpp. Nội dung soạn thảo sẽ được nhập vào trong tập tin này.

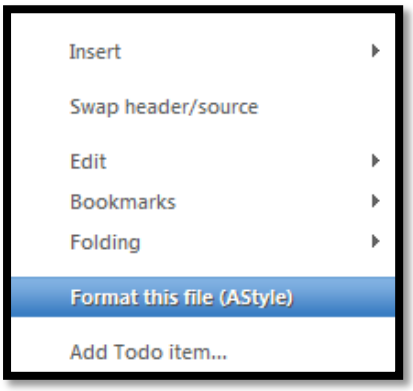
Nếu ta muốn bổ sung các tập tin khác hoặc các lớp đối tượng, ta có thể bổ sung chúng từ menu File > New.

Biên dịch chương trình:

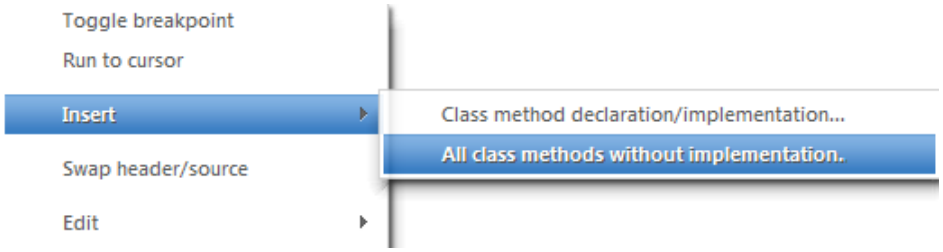
+ Nhấp vào **Build** > **Build and Run**.

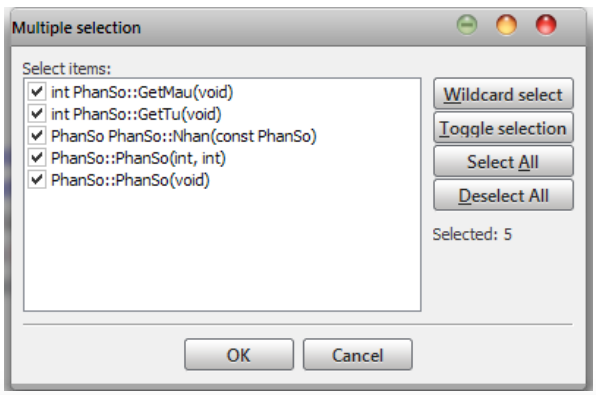
+ Hoặc nhấp phím **F9**.

Tự động định dạng mã. Khi viết một chương trình C++ hay bất kì một chương trình trên một ngôn ngữ lập trình nào khác, ta cần tuân thủ quy phạm định dạng mã nguồn. Có nhiều chuẩn mực cho các định dạng mã nguồn: chuẩn Hungary, chuẩn lạc đà... Dù rằng, chúng không ảnh hưởng đến việc biên dịch chương trình, nhưng chúng giúp người đọc có thể dễ hiểu chương trình của chúng ta hơn. Nếu ta không nắm vững các quy phạm này thì có thể sử dụng chức năng định dạng mã nguồn tự động của CodeBlocks. Hãy kích chuột phải vào vùng soạn thảo, sau đó chọn **Format this file (Astyle)**.



Tự động khởi tạo phần thân các phương thức của lớp. Để hỗ trợ cho việc soạn thảo, CodeBlocks cũng hỗ trợ chức năng khởi tạo nhanh mã nguồn. Để khởi tạo nhanh phần khai báo thân phương thức của lớp từ khai báo prototype của nó, chúng ta đặt trỏ chuột vào sau khai báo lớp (tức vị trí sẽ chèn khai báo thân phương thức), sau đó, kích chuột phải, chọn **Insert** > **All class methods without implementation.**





Trong hộp thoại hiện ra, hãy chọn những phương thức muốn khởi tạo phần thân tương ứng, sau đó, nhấp Ok.

Hình 2 – Khởi tạo thân phương thức

2. Eclipse Helios

Sau khi tải xong Eclipe Helios về máy, hãy tiến hành giải nén tập tin. Chương trình Eclipse không yêu cầu chúng ta phải cài đặt, nhưng nó có thể làm việc nếu trên máy tính đã cài một máy ảo Java. Để tải về máy ảo Java, chúng ta có thể truy cập vào trang chủ của Sun (nay là Oracle) tại địa chỉ sau đây[3].

Để xây dựng một chương trình C/C++ trên Eclipse, chúng ta cần:

- **Eclipse Helios for C/C++** (nếu phiên bản tải về là dành cho Java, ta cần phải cài đặt thêm plugin hỗ trợ); hoặc có thể sử dụng một ấn bản cũ hơn của Eclipse như Galileo, Europa….

- Công cụ biên dịch GCC – **MingW**.

- Máy ảo Java **JVM**.

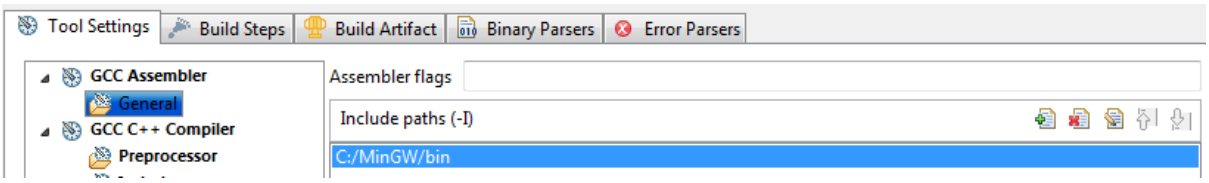
Các bước cấu hình trên Eclipse Helios

Bước 1. Cài đặt máy ảo Java.

Bước 2. Cài MinGW.

Bước 3. Giải nén Eclipse Helios, sau đó khởi động nó (nhấp vào tập tin eclipse.exe). Thông thường, Eclipse sẽ tự động cấu hình MinGW giúp ta. Nếu không, hãy thực hiện bước 4.

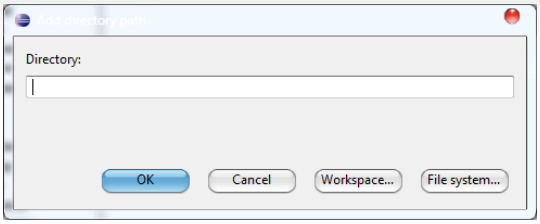
Bước 4. Vào menu Project > Properties. Trong hộp thoại xuất hiện, hãy chọn C/C++ Build > Settings.



Hình 3 – Cấu hình MinGW trong Eclipse Helios

Trong thẻ Tool Settings, ta chọn GCC Assembler > General. Sau đó, nhấp vào biểu tượng có dấu cộng mầu xanh. Hộp thoại sau sẽ hiện ra:

[3] http://javadl.sun.com/webapps/download/AutoDL?BundleId=41723



Hình 4 – Chọn đường dẫn đến thư mục bin của MinGW

Ta tiến hành hãy nhập tên đường dẫn đến thư mục bin của MinGW (hoặc nhấp vào nút File system để duyệt đến thư mục này). Mặc định khi cài đặt, thư mục này sẽ là C:\MinGW\bin. Sau đó nhấp Ok. Vậy là công việc cấu hình đã hoàn tất.

Xây dựng dự án đầu tiên trên Eclipse

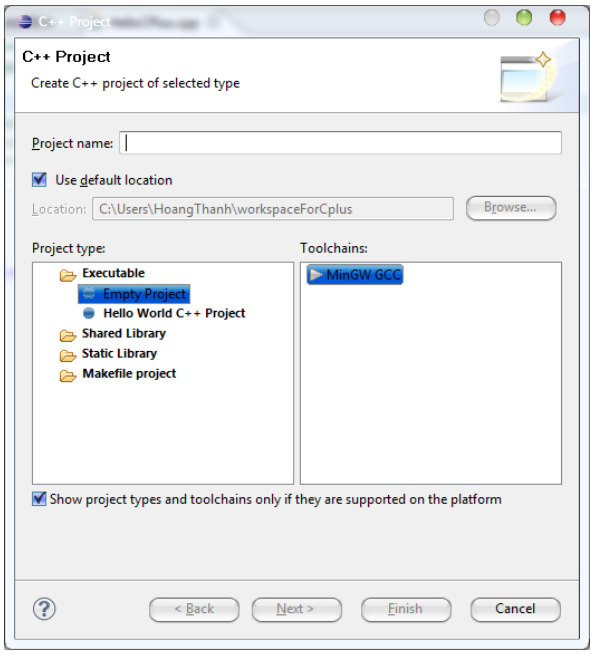
Cũng giống như CodeBlocks, Eclipse cũng tổ chức chương trình theo dự án. Để tạo mới một dự án trong Eclipse, chúng ta có ba cách:

- Vào File > New > C++ Project.

- Vào biểu tượng tạo mới dự án trên thanh công cụ, chọn C++ Project.

- Kích chuột phải vào cửa sổ Project Explorer > chọn New > C++ Project.

Tiếp đến, hộp thoại sau đây sẽ xuất hiện.



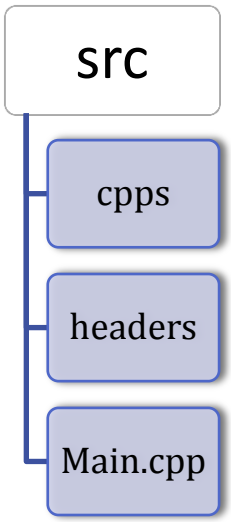
Hình 5 - Tạo mới dự án

Trong hộp thoại này, nếu chọn một dự án khả thi (executable), hãy chọn executable. Ta cũng có thể chọn thư viện dll (static library)... Tương ứng với dự án khả thi, chúng ta có thể chọn Empty Project hoặc Hello World C++ Project. Đối với Empty Project, nó sẽ tạo một dự án trống. Ngược lại với Hello World C++ Project, ta sẽ nhận được một file cpp chứa nội dung mà chúng ta sẽ thảo luận trong chương tiếp theo.

Sau đó, hãy nhập vào tên dự án và nhấp Next (nếu chưa cấu hình MinGW), hoặc nhấp Finish (nếu đã hoàn tất việc cấu hình).

Tạo mới một file nội dung trong Eclipse. Một chương trình trong C++ thường chia làm hai loại tệp: .cpp và .h. Tệp .cpp thường chứa nội dung chương trình, tệp .h thường chứa các khai báo.

Lời khuyên trước khi tạo mới các file: hãy tạo một thư mục chung để chứa toàn bộ nội dung sau này, tôi tạm gọi thư mục này là thư mục src. Trong thư mục src, hãy tạo hai thư mục, một thư mục cpps và một thư mục headers. Thư mục cpps sẽ chứa toàn bộ tệp .cpp, thư mục headers sẽ chứa toàn bộ tệp .h. Tệp Main.cpp chứa hàm main sẽ được đặt trong thư mục src. Nghĩa là ta sẽ có cấu trúc tương tự như sau:

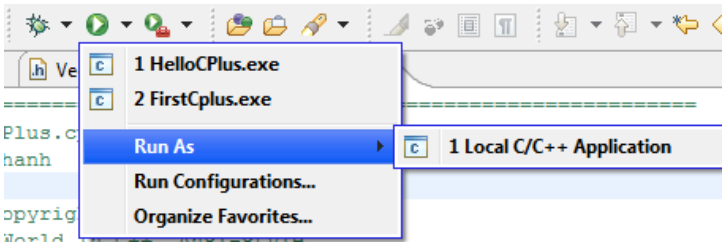


Hiển nhiên ta hoàn toàn không nhất thiết phải thực hiện theo như cấu trúc thư mục này. Tuy nhiên điều này sẽ làm cho dự án của ta trở nên sáng sủa hơn rất nhiều. Chúng ta có thể bổ sung thêm các thư mục phụ khác, nhưng nên tuân thủ cấu trúc cây này (ví dụ khi cần phân biệt các tập tin cpp thành nhiều loại khác nhau, thì trong thư mục cpps, hãy tạo thêm các thư mục con khác...)

Hình 6 - Cấu trúc thư mục của một dự án

Biên dịch một dự án

Để biên dịch một dự án, hãy nhấp vào biểu tượng sau đây trên thay công cụ của Eclipse.



Hình 7 - Biên dịch một dự án

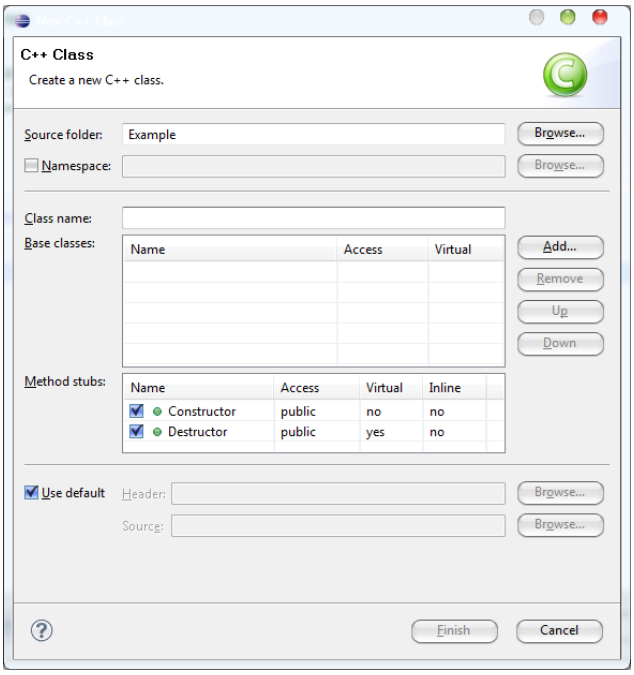
Chọn Run As > Local C/C++ Application.

Một số thủ thuật giúp soạn thảo nhanh

Eclipse chứa đựng một tập các tiện ích giúp chúng ta soạn thảo nhanh hơn, ít phát sinh lỗi hơn. Sau đây, tôi xin giới thiệu một vài tính năng giúp các chúng ta soạn thảo nhanh hơn.

Tạo mới một lớp

Vào New > Class. Hộp thoại sau đây sẽ hiện ra



Hình 8 - Hộp thoại tạo mới class

Trong hộp thoại này, cần lưu ý: source folder – thư mục chứa tập tin sẽ tạo mới (thường sẽ được phân tách thành tệp .h và .cpp), namespace – phạm vi tác dụng của nó trong namespace được chỉ định, class name – tên của lớp sẽ tạo mới, base class – tên của lớp cha mà nó sẽ thừa kế (bấm vào nút add để chọn các lớp tồn tại), constructor và destructor – cho phép khởi tạo hàm tạo và hàm hủy. Chúng ta sẽ tìm hiểu những khái niệm này khi làm quen với lập trình hướng đối tượng.

Tạo nhanh các phương thức Getter và Setter

Nếu khi khai báo một lớp, cùng với các thuộc tính của nó, thay vì sử dụng hàm tạo để thiết lập giá trị ban đầu, ta có thể dùng hàm setter; hoặc để tiếp nhận giá trị từ các thuộc tính, ta có thể dùng các hàm getter. Tôi sẽ giới thiệu chi tiết hơn về các phương thức này trong phần lập trình hướng đối tượng. Trong phần này, tôi sẽ hướng dẫn cách tạo chúng bằng thao tác nhấp chuột. Vào menu Source, chọn Generate Getters and Setter. Trong hộp thoại

hiện ra, hãy chọn các thuộc tính cần tạo phương thức getter và setter, sau đó nhấp Ok.

Một số phím tắt khác

Phím tắt	Công dụng
Ctrl+Space	Bật chế độ gợi nhắc lệnh.
main – Ctrl+Space	Khởi tạo nhanh hàm main.
Ctrl+Shift+F	Định dạng nhanh mã nguồn.
Ctrl+/	Comment vùng mã đã được bôi đen, nếu vùng bôi đen đã ở chế độ comment, thì dấu comment sẽ bị hủy bỏ.
Tab	Dịch toàn bộ nội dung bị bôi đen sang phải một tab.
Shift+Tab	Dịch toàn bộ nội dung bị bôi đen sang trái một tab.
Ctrl+1	Chỉnh sửa nhanh toàn bộ các từ giống với từ đang được bôi đen. Sau khi chỉnh sửa xong, nhấp Enter để kết thúc.
Ctrl+Shift+/	Tạo một khối comment cho vùng văn bản đã bị bôi đen.
Ctrl+Shift+\	Hủy bỏ vùng văn bản bị comment bởi khối comment.

Trên đây, tôi đã giới thiệu sơ qua hai chương trình soạn thảo miễn phí để lập trình C/C++: CodeBlocks và Eclipse. Với CodeBlocks, chỉ cần tải và cài đặt. Môi trường hỗ trợ biên dịch GCC đã được tích hợp sẵn. Với Eclipse, ta phải thực hiện cấu hình để kết hợp với trình biên dịch GCC. Nếu là người có nhiều trải nghiệm về máy tính, thì nên chọn Eclipse bởi nó là chương trình soạn thảo rất chuyên nghiệp. Nếu là người mới tiếp xúc máy tính, hãy chọn CodeBlock vì cài đặt đơn giản.

3. Visual Studio 2010 dành cho Visual C++

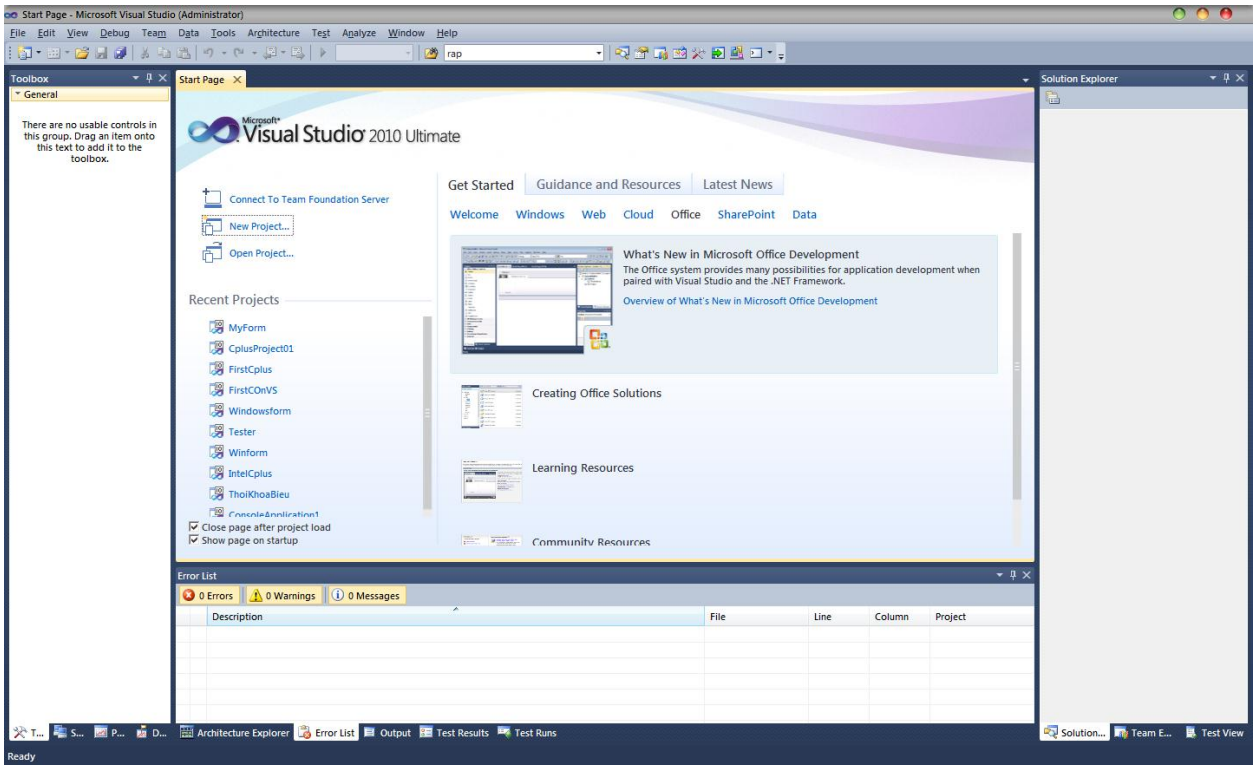
Visual Studio 2010 là một môi trường biên dịch tích hợp của Microsoft. Nó là trình biên dịch tốt nhất, hiện đại nhất trên hệ điều hành Windows. Chúng ta có thể sử dụng nó để biên dịch C++, C#, Visual Basic, J#... Ta sẽ tìm hiểu Visual Studio theo hướng tiếp cận với C++. Một điều cần lưu ý, với phiên bản 2010 này, Visual Studio có hai phiên bản dành cho C++: C++ for Net và C++ for Win32. Chúng ta chỉ tìm hiểu về tính năng C++ for Win32. Trong nội dung của giáo trình này, ta sẽ xây dựng các ứng dụng Console trên nền

Win32 mà không thảo luận thêm về Visual C++ for Net bởi vì nó thuộc một phạm trù tương đối khác so với Visual C++ for Win32.

Khởi động Visual Studio 2010.

Để khởi động VS 2010, ta có thể thực hiện một trong hai cách sau:

- Nhấp đối chuột vào biểu tượng VS 2010 trên nền Desktop.
- Vào Start > All Programs > Microsoft Visual Studio 2010, chọn biểu tượng VS 2010.

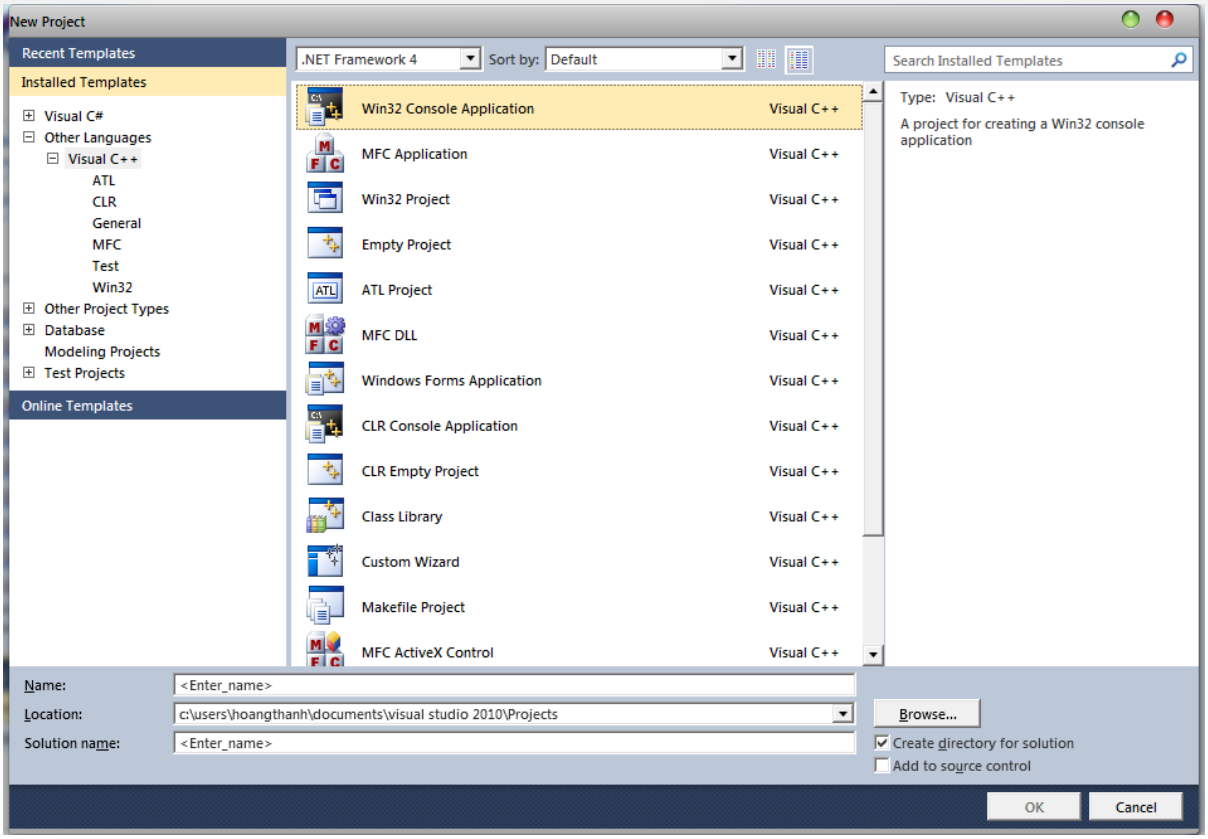


Hình 9 - Giao diện tổng thể của Visual Studio 2010

Tạo mới dự án trong VS 2010.

Cũng như Eclipse, VS cũng quản lý theo các workspace và các dự án. Trong VS, workspace được gọi là Solution. Trong mỗi workspace có thể chứa nhiều dự án. Nếu chưa tạo một dự án nào, thì khi tạo mới một dự án, workspace sẽ tự động được tạo. Để tạo mới một dự án, ta vào File > New Project (hoặc tổ hợp phím tắt Ctrl+Shift+N).

Trong hộp thoại xuất hiện, chúng ta chọn Win32 Console Application.

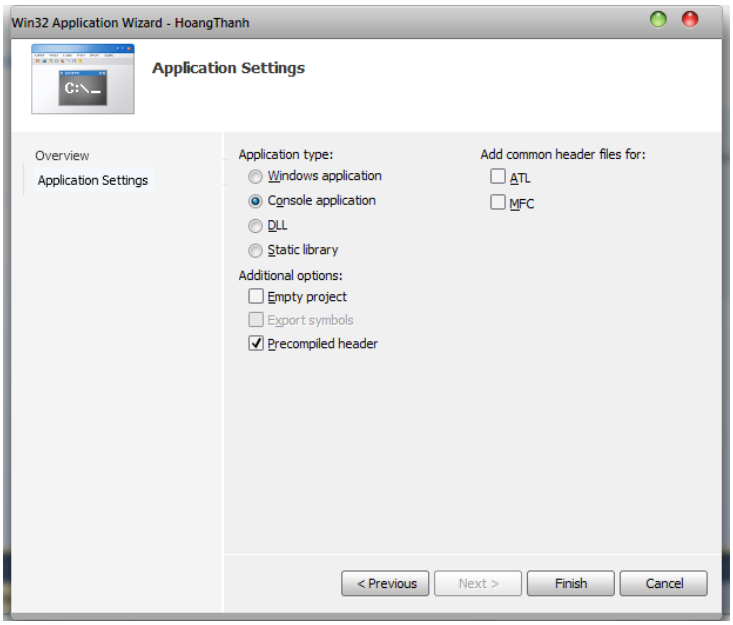


Hình 10 - Tạo dự án Win32 Console

Mục name: hãy nhập tên dự án mà cần tạo.

Mục Location: nhấp vào nút Browse để chọn vị trí lưu trữ. Mặc định, Visual Studio sẽ lưu trữ dự án ở thư mục Documents.

Mục Solution name: tạo một thư mục con trong thư mục dự án, hay tạo trực tiếp trong thư mục dự án.



Hình 11 - Win32 Application Wizard

Hộp thoại Hình 12 sẽ hiện ra.

Nhóm Application Type

+ Windows application: tạo ứng dụng winform.

+ Console application: tạo ứng dụng chạy trên DOS.

+ Dll: tạo thư viện dll.

+ Static library: tạo thư viện tĩnh.

Nhóm Add common header file

+ Alt: tạo header từ lớp thư viện Alt.

+ MFC: tạo header từ lớp thư viện MFC.

Nhóm Additional options

+ Empty project: tạo dự án rỗng không có tập tin.

+ Export symbols: xuất bản các biểu tượng.

+ Precompiled header: tạo tập tin tiêu đề tiền biên dịch.

Hãy chọn Console Application và chọn *Empty Project*. Sau đó, nhấp Finish.

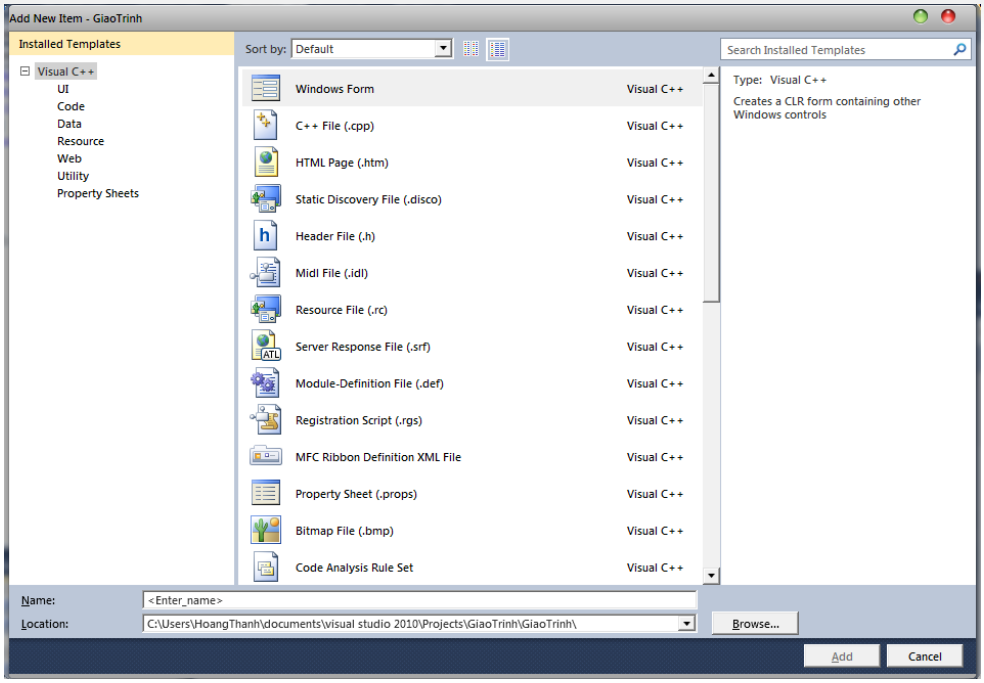
Tạo các tập tin trong dự án.

Trong cửa sổ Solution Explorer, hãy kích chuột phải và chọn Add:

- Nếu tập tin đã tồn tại, hãy chọn Add Existing Items. Sau đó, chúng ta duyệt đến vị trí tồn tại tập tin.

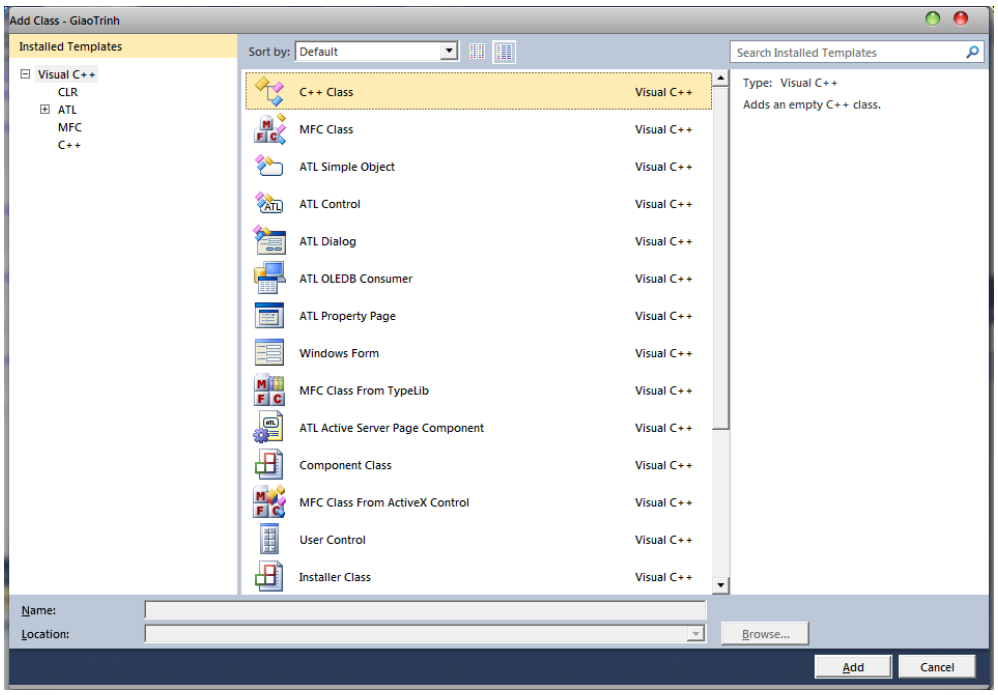
- Nếu tập tin chưa tồn tại, hãy chọn Add New Items. Trong cửa sổ xuất hiện, tùy thuộc vào tập tin mà chúng ta cần, hãy chọn loại tương ứng. Thông thường, trong dự án của C++, chúng ta sử dụng hai tập tin là tiêu đề .h và thân chương trình .cpp. Sau đó, hãy nhập tên của tập tin và nhấp Ok. Tệp tin tiêu đề .h thường chứa các khai báo prototype của hàm hoặc lớp. Ngoài ra, nó có thể chứa các hàm macro, các khai báo hằng và biến toàn cục được sử dụng trong toàn bộ chương trình. Tập tin .cpp thường chứa phần thân của các hàm hoặc lớp. Khi làm việc với các dự án trong C++, chúng ta nên tách chương trình thành nhiều phần và nên sử dụng các tệp tiêu đề để làm cho chương trình gọn gàng và dễ hiểu hơn.

Sau khi chọn được tập tin cần tạo, hãy nhập tên của tập tin, sau đó nhấp nút Add. Tập tin mới sẽ được bổ sung vào dự án.



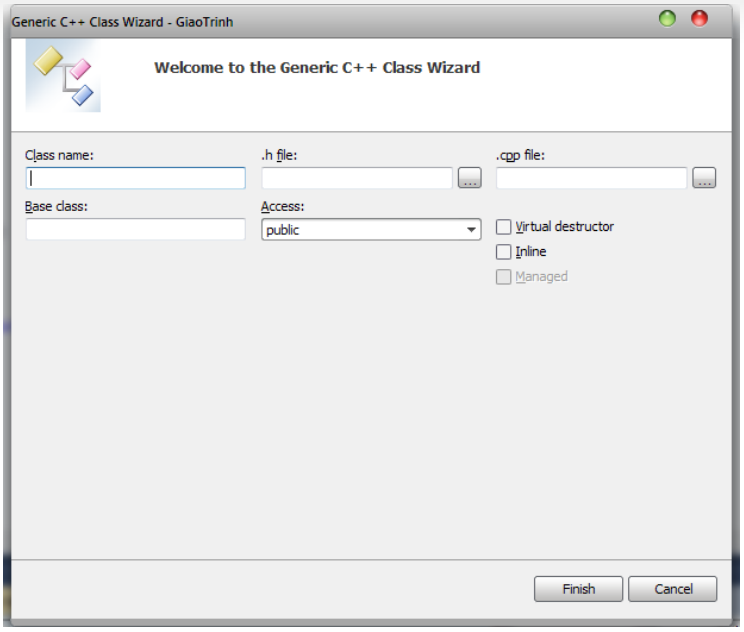
Hình 12 - Bổ sung thêm một tập tin

- Add Class: bổ sung các lớp đối tượng cho dự án. Ở đây, chúng ta chọn C++ class.



Hình 13 - Bổ sung thêm lớp đối tượng

Nhập Add. Cửa sổ sau đây sẽ xuất hiện



Hình 14 - Tạo lớp bằng Class Wizard

- Class name: tên của lớp.

- .h file: tên của tiêu đề lớp cũng là tên của tập tin tiêu đề.

- .cpp file: tên của tập tin .cpp tương ứng với lớp.

- Base class: nếu lớp mới tạo ra thừa kế từ một lớp khác, hãy nhập tên của lớp cơ sở vào đây.

- Access: mức thừa kế của lớp đang tạo từ lớp cơ sở.

- Virtual destructor: tạo một phương thức hủy ảo.

- Inline: tạo một phương thức inline. Tuy chúng ta có thể sử dụng từ khóa này, nhưng cơ chế làm việc của Visual C++ là tự động bổ sung inline khi biên dịch nếu phương thức được cho là phù hợp để sử dụng inline. Điều đó có nghĩa là chúng ta không cần dùng đến từ khóa này.

Biên dịch dự án.

- Để biên dịch và thực thi một dự án, chúng ta nhấp vào Debug > Start Debugging (hoặc Start without Debugging).

- Để biên dịch toàn bộ dự án mà không thực thi dự án, chúng ta vào Build, chọn Build Solution.

Một số phím tắt trong Visual Studio 2010.

- Tạo vùng comment (chú thích): bôi đen vùng mã cần tạo chú thích, nhấn tổ hợp Ctrl+K, Ctrl+C.

- Hủy bỏ vùng comment: bôi đen vùng mã đã comment, nhấn tổ hợp Ctrl+K, Ctrl+U.

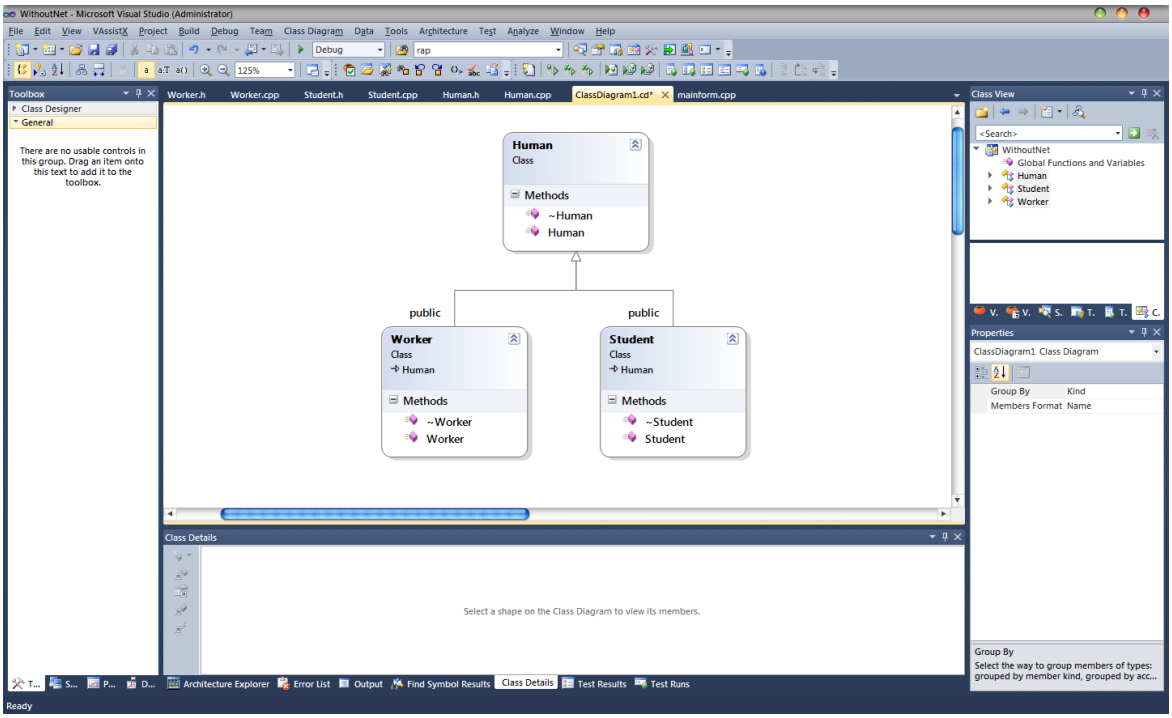
- Định dạng mã nguồn: bôi đen vùng mã cần định dạng, nhấn tổ hợp Ctrl+K, Ctrl+F.

- Tự động hoàn tất mã và gợi nhắc lệnh: tổ hợp Ctrl+Space.

Visual Studio 2010 không hỗ trợ các tính năng mạnh mẽ cho việc khởi tạo mã nguồn. Tuy nhiên, nếu chúng ta mong muốn làm việc đơn giản và hiệu quả hơn thì ta có thể sử dụng tiện ích bổ trợ. Một trong những tiện ích làm việc khá hiệu quả là Visual Assist. Phiên bản cho đến thời điểm này (năm 2010) là 10.6.

Xem biểu đồ lớp.

Để quan sát biểu đồ lớp trong VS 2010, ta nhấp chuột phải vào tên dự án (trong cửa sổ Solution Explorer), chọn **Show class diagram**. Sau đó, chúng ta kéo thả các lớp đối tượng vào vùng biểu đồ.



Hình 15 - Xem biểu đồ lớp

CHƯƠNG 1. CƠ BẢN VỀ C++

Cấu trúc của một chương trình C++

Một cách thức tốt nhất để học lập trình đó là hãy thử viết một chương trình đầu tiên. Nếu chúng ta đã từng làm quen với một ngôn ngữ lập trình nào đó, thì chắc hẳn ai cũng biết đến ví dụ kinh điển của một ngôn ngữ lập trình đó là chương trình "Hello, world !".

Mã chương trình

```
[1.]  //my first program
[2.]  #include <iostream>
[3.]  using namespace std;
[4.]  int main()
[5.]  {
[6.]    cout<<"Hello, world !";
[7.]    return 0;
[8.]  }
```

Kết quả

```
Hello, world !
```

Giải thích về chương trình:

[1.] Các kí tự nằm sau dấu // sẽ không được biên dịch mà nó được hiểu là dấu comment (dòng chú thích). Trong C++ cũng như C, việc chú thích trên một dòng sẽ được đặt sau dấu //. Nếu muốn tạo một chú thích nhiều dòng, chúng ta có thể sử dụng dấu /* *Tạo chú thích ở đây* */

[2.] Dòng này bắt đầu bằng kí tự #include. Tiếp đến là tên tập tin tiêu đề (chứa các thư viện). Thư viện iostream được đặt trong dấu <>. Nó chứa các hàm xuất nhập cơ bản. Hàm này là một phần của namespace std.

[3.] Trong C++, các thành phần của thư viện chuẩn được khai báo trong namespace. Ở đây là namespace std. Để có thể truy xuất đến các thành phần của nó, chúng ta mô tả nó bằng từ khóa using. Trong thư viện chuẩn của C++, đối tượng cout được tổ chức trong namespace std.

[4.] Bất kì một chương trình C++ nào cũng phải có một hàm main để thực thi chương trình. Một hàm sẽ được khai báo theo cấu trúc trên.
Từ khóa int mô tả kiểu dữ liệu mà hàm trả về là integer. Chúng ta cần lưu ý rằng, trong chương trình C thì ta có thể tùy ý khai báo là void hoặc int, nhưng trong C++ thì bắt buộc phải khai báo là int. Vậy int hay void

trong trường hợp này có thực sự quan trọng ? Chúng ta nên luôn khai báo hàm main có kiểu dữ liệu trả về là kiểu int. Sở dĩ như vậy là vì khi hàm main trả về kiểu int thì theo quy ước, nếu chương trình có lỗi, nó sẽ trả về một mã int khác 0 và ngược lại, nếu chương trình không có lỗi, nó sẽ trả về mã int 0. Lỗi ở đây là lỗi chương trình liên quan đến quá trình biên dịch, chứ không phải là lỗi liên quan đến cú pháp. Chúng ta sẽ nhận thấy mã mà nó trả về trong dòng thông báo cuối cùng khi biên dịch: *process returned 0 (0x0)*.

Tên hàm là main. Tiếp theo là cặp dấu ngoặc đơn dùng để chứa tham số đính kèm. Thông thường một chương trình ứng dụng sẽ chứa hai tham số trong hàm main là *int argc* và *char* args[]*. Các tham số này gọi là *tham số dòng lệnh*. Tiếp theo là dấu {}. Bên trong cặp dấu này là chương trình chính.

[5.] Dấu mở khối.

[6.] Đối tượng cout (đọc là C-out) là chuẩn dùng để xuất dữ liệu ra màn hình. Chúng ta cần lưu ý hàm printf vẫn hoạt động tốt trong trường hợp này. Nếu dùng hàm printf thì ta không cần khai báo thư viện iostream và namespace std ở trên. Khi sử dụng đối tượng cout, chúng ta cũng có thể bỏ qua dòng lệnh [3.] và thay vào đó ta sẽ viết std::cout. Khi sử dụng đối tượng cout, chúng ta có thêm một cách thức để xuống dòng thay vì dùng \n, đó là endl. Đối tượng cout thường đi với toán tử xuất <<. Chúng ta có thể sử dùng nhiều toán tử này khi muốn xuất nhiều phần tử riêng biệt: cout<<string1<<string2<<….<<endl.

[7.] Câu lệnh return dùng để trả về giá trị của hàm main. Nếu hàm có trả về giá trị, thì cần return một giá trị nào đó cùng kiểu dữ liệu trả về với hàm. Nếu hàm là void, thì không cần return.

[8.] Dấu đóng khối tương ứng với mở khối [5].

Chú ý:

- Cũng như C, C++ là ngôn ngữ phân biệt chữ hoa và chữ thường.
- Kết thúc một dòng lệnh trong C++ bao giờ cũng phải có dấu ;
- Một dấu ngoặc đơn (), dấu ngoặc nhọn {} bao giờ cũng song hành. Điều đó có nghĩa nếu dùng dấu mở thì phải có dấu đóng tương ứng. Dấu ngoặc đơn thường dùng sau tên hàm, và bên trong nó là tham số hình thức hoặc trong các lệnh có cấu trúc. Dấu ngoặc nhọn thường dùng để quy định phạm vi của một khối lệnh (scope). Một cách thức giúp chúng ta chuyên nghiệp hơn khi lập trình, là sau dấu mở, ta nên sử dụng tiếp

dấu đóng (thông thường các trình soạn thảo sẽ hỗ trợ một cách tự động). Sau đó hãy nhập nội dung cần thiết vào bên trong cặp dấu này. Điều đó sẽ tránh khỏi sự nhầm lẫn khi chương trình có quá nhiều dấu đóng mở.

- Để nhận biết được phạm vi ảnh hưởng của các khối lệnh – hãy sử dụng phím tab để tạo ra sự lồi lõm khi viết mã chương trình. Như trong ví dụ trên, đối tượng cout và hàm return sẽ nhảy vào một tab so với dấu khối lệnh tương ứng. Đừng bao giờ tiết kiệm sử dụng phím tab và phím enter. Nếu sử dụng hợp lí, chương trình sẽ rất sáng sủa và dễ đọc.

Bài tập 1.

1. Hãy viết chương trình in ra dòng chữ "Chao ban, ban co khoe khong".
2. Hãy viết chương trình in ra hai dòng chữ trên hai dòng phân biệt "Vietnam" và "Hoa ky".
3. Hãy viết chương trình in ra tam giác đều với các đỉnh là các dấu *.

```
   *
  * *
```

CHƯƠNG 2. BIẾN VÀ CÁC KIỂU DỮ LIỆU

Tương ứng với chương trình "Hello world", chúng ta cần thảo luận một vài chi tiết. Chúng ta có một vài dòng lệnh, biên dịch chúng và sau đó chạy chương trình để thu kết quả. Dĩ nhiên ta có thể làm nhanh hơn, tuy nhiên việc lập trình không chỉ đơn thuần là in ra các dòng thông báo đơn giản lên màn hình. Để đi xa hơn, chúng ta sẽ viết một chương trình thực thi một tác vụ hữu ích là giúp chúng ta tìm hiểu về khái niệm biến.

Giả sử có hai giá trị 5 và 2. Ta cần lưu hai giá trị này vào bộ nhớ. Bây giờ, nếu tôi muốn cộng thêm 1 vào số thứ nhất và lưu lại giá trị này cho nó, tiếp theo tôi muốn lấy hiệu của số thứ nhất sau khi thay đổi với số thứ hai. Tiến trình xử lý công việc trên có thể được viết trên C++ như sau:

Chương trình

```
int a = 5;

int b = 2;

a = a + 1;   // a=6

int result = a – b;   //result = 4
```

Biến được dùng để lưu giá trị và nó có thể thay đổi được. Một biến sẽ được quy định bởi một kiểu dữ liệu nào đó. Trong trường hợp ví dụ của chúng ta, biến có kiểu dữ liệu là int. Kiểu dữ liệu thường có hai loại: kiểu dữ liệu nguyên thủy (primitive data type) và kiểu dữ liệu tham chiếu (reference data type). Chúng ta sẽ thảo luận chi tiết về chúng trong phần tiếp theo. Nhưng ta có thể hiểu rằng, một kiểu dữ liệu đơn giản và có cấu trúc trong C là kiểu dữ liệu nguyên thủy. Đối với kiểu dữ liệu tham chiếu, tôi sẽ giới thiệu trong phần lập trình hướng đối tượng trong C++.

Từ khóa

Từ khóa trong C++ có thể có một hoặc nhiều từ. Nếu từ khóa có nhiều từ, thì giữa các từ có dấu gạch chân (_). Kí tự trắng và các kí tự đặc biệt không được phép sử dụng trong từ khóa, tên hàm, tên biến. Tên của chúng không được bắt đầu bằng kí tự số.

Bảng từ khóa chuẩn trong C++

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

Bảng từ khóa bổ sung trong C++

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

Kiểu dữ liệu nguyên thủy

Khi lập trình, chúng ta lưu các biến trong bộ nhớ máy tính, nhưng máy tính cần phải biết loại dữ liệu mà chúng ta muốn lưu. Điều này giúp bảo đảm đủ số lượng ô nhớ cần thiết để lưu dữ liệu.

Trong máy tính, bộ nhớ được tổ chức theo các byte. Một byte là một đơn vị đo lường tối thiểu mà chúng ta có thể quản lý trong C++. Một byte có thể lưu một biến char. Thêm vào đó, máy tính cũng quản lý những kiểu dữ liệu phức tạp hơn. Bảng sau đây liệt kê các kiểu dữ liệu và kích thước tương ứng.

Tên	Mô tả	Kích thước	Vùng giá trị
char	Kí tự hoặc số nguyên bé	1 byte	signed: -128 ->127 unsigned: 0 -> 255
short	Số nguyên ngắn	2 byte	signed: -2^{15} -> $2^{15}-1$ unsigned: 0 -> $2^{16}-1$
int	Số nguyên	4 byte	signed: -2^{31} -> $2^{31}-1$ unsigned: 0 -> $2^{32}-1$
long	Số nguyên dài	4 byte	signed: -2^{31} -> $2^{31}-1$ unsigned: 0 -> $2^{32}-1$
long long	Số nguyên cực dài	8 byte	signed: -2^{63} -> $2^{63}-1$ unsigned: 0 -> $2^{64}-1$
bool	Giá trị logic – true/false	1 byte	true và false
float	Số thập phân	4 byte	7 số thập phân
double	Số thập phân chấm động	8 byte	15 số thập phân
long	Số thập phân chấm động	8 byte	15 số thập phân

double	dài		
wchar_t	Kí tự dài	2/4 byte	

Kích thước trong bộ nhớ và miền giá trị của các kiểu dữ liệu còn phụ thuộc vào hệ thống và chương trình dịch tương ứng. Giá trị được đưa ra ở đây là trên hệ thống Windows 32 bit và trình dịch GCC MinGW. Nhưng đối với hệ thống khác, các giá trị này có thể thay đổi (ví dụ kiểu int và long trên Windows 32 bit và 64 bit là 4 byte, nhưng trên Linux 32 bit là 4 byte và trên Linux 64 bit là 8 byte).

Khai báo biến

Như ví dụ trên, ta thấy rằng, muốn sử dụng một biến trong C++, ta cần khai báo biến với kiểu dữ liệu mà ta mong muốn. Cấu trúc khai báo

<Tên kiểu dữ liệu> <Tên biến>;

Ví dụ

int a; //Khai báo biến a kiểu nguyên

float mynumber; //Khai báo biến mynumber kiểu float

bool istrue; //Khai báo biến istrue kiểu bool

long num1, num2, num3; //Khai báo ba biến num1, num2, num3 cùng kiểu long

Chú ý:

- Nếu khi khai báo biến thuộc các kiểu nguyên mà ta không sử dụng khai báo có dấu (signed) hoặc không dấu (unsigned), thì chương trình dịch mặc định sẽ quy định là kiểu nguyên có dấu.

int mynum;

//tương đương signed int mynum;

- Đối với kiểu char thì có ngoại lệ. Chúng ta nên khai báo tường minh là signed char hoặc unsigned char.
- Đối với signed int và unsigned int có thể viết đơn giản là signed hoặc unsigned.

- Nếu muốn chắc chắn về kích thước của kiểu dữ liệu mà ta cần sử dụng, hãy sử dụng hàm sizeof để xác định kích thước bộ nhớ của kiểu dữ liệu. Hàm sizeof(tên biến) hoặc sizeof(kiểu dữ liệu) – trả về kiểu dữ liệu nguyên.

Chương trình

```
#include <iostream>

using namespace std;

int main()

{

  int a;

  cout<<sizeof(a);

  //Hoặc có thể viết

  cout<<sizeof(int);

}
```

Kết quả

4 (trên windows 32 bit)

Phạm vi tác dụng của biến

Tất cả các biến mà tôi giới thiệu ở đây được sử dụng trong chương trình cần phải được khai báo với kiểu dữ liệu được chỉ định. Một biến được khai báo trong một khối lệnh nào, thì nó chỉ có tác dụng trong khối lệnh đó. Biến được khai báo theo kiểu này gọi là biến cục bộ (hoặc biến địa phương). Nếu một biến được khai báo ngoài tất cả các khối lệnh (kể cả hàm main) thì biến đó có tác dụng trong toàn bộ chương trình và gọi là biến toàn cục.

Chương trình

```
[1.]  #include <iostream>
[2.]  using namespace std;

[3.]  int a;
[4.]  char c;
[5.]  unsigned int d;

[6.]  int main()
[7.]  {//Khối lệnh 1
```

```
[8.]        signed long m;
[9.]        float n;
[10.]       {//Khối lệnh 2
[11.]         double x;
[12.]         x = 1;
[13.]         cout<<x;
[14.]       }
[15.]     }
```

Giải thích:

Các biến khai báo ở các dòng [3.], [4.] và [5.] được khai báo ngoài mọi khối lệnh, nó có tác dụng trong toàn bộ chương trình và nó được gọi là biến toàn cục (global variable). Các biến được khai báo trong khối lệnh 1 (tương ứng [8.] và [9.]) và khối lệnh 2 (tương ứng [11.] và [12.]) gọi là biến cục bộ (local variable), nó có tác dụng trong khối lệnh trực tiếp chứa nó. Có nghĩa là biến x chỉ có tác dụng trong khối lệnh 2; biến m, n có tác dụng trong khối lệnh 1.

Các biến toàn cục có thể được sử dụng trong toàn bộ chương trình, nó có thể được gọi trong các hàm, trong hàm chính main. Còn biến cục bộ được khai báo trong khối lệnh nào, thì nó chỉ có thể được sử dụng trong khối lệnh đó.

Trong một số tình huống, biến có thể được khai báo trong dấu ngoặc đơn (tình huống này thường gặp khi nghiên cứu các lệnh có cấu trúc), thì biến này cũng gọi là biến cục bộ

```
for (int a = 0; i<10; i++){
        ...nhập nội dung....
}
```

Lúc này, biến sẽ có tác dụng trong khối lệnh tương ứng (khối lệnh nằm trong vòng lặp for).

Khởi tạo giá trị cho biến

Khi một biến cục bộ được khai báo, giá trị mặc định của nó sẽ chưa được tạo ra. Vì vậy, muốn sử dụng được biến, ta cần phải khởi tạo giá trị cho biến. Có hai cách để khởi tạo giá trị của biến trong C++.

Cú pháp	**Ví dụ**
type tên_biến = giá_trị_khởi_tạo;	int a = 0;
type tên_biến (giá_trị_khởi_tạo);	int a (0);

Bây giờ, ta có thể viết lại đoạn chương trình tính toán giá trị của các biến ở trên, bằng cách sử dụng giá trị khởi tạo mặc định này.

Khởi tạo theo cách 1

```
#include <iostream>
using namespace std;
int main()
{
  int a = 5;
  int b = 2;
  a = a + 1;  // a=6
  int result  = a – b;  //result = 4
  cout<<result<<endl;
  result 0;
}
```

Khởi tạo theo cách 2

```
#include <iostream>
using namespace std;
int main()
{
  int a (5);
  int b (2);
  a = a + 1;  // a=6
  int result  (a – b);  //result = 4
  cout<<result<<endl;
  result 0;
}
```

Bài tập 2.

1. Hãy viết một chương trình tương đương, sử dụng cả hai kiểu khởi tạo trên.
2. Hãy chọn một cách khởi tạo tùy ý, để viết chương trình tính giá trị của biểu thức delta = b*b-4*a*c, với a, b, c lần lượt nhận các giá trị 1, 5, 3.

Khởi tạo giá trị cho biến tĩnh static

Một biến được khai báo bằng từ khóa static thì nó chỉ khởi tạo giá trị đúng một lần khi biến được tạo ra. Thông thường những biến này được đặt vào trong một tệp tiêu đề .h để sử dụng cho toàn bộ chương trình. Ví dụ sau đây

minh họa cho giá trị của biến static sẽ không khởi tạo lần thứ hai trong vòng lặp.

Chương trình	Kết quả

#include <iostream>
 	
using namespace std;
 	
int main()
{
  for (int i=0; i<5; i++)
  {
  static int x = 2;
  x++;
  cout<<x<<endl;
  }
}
	3
	4
	5
	6
	7

Giải thích: biến x được khởi tạo trong vòng lặp for, nếu không có từ khóa static, thì trong mỗi lần lặp, biến này sẽ được khởi tạo lại và giá trị in ra sẽ luôn là 3. Tuy nhiên, trong trường hợp này, ta sử dụng từ khóa static, do đó, giá trị của biến x chỉ được khởi tạo một lần duy nhất. Trong những lần lặp tiếp theo, giá trị của x vẫn được lưu lại. Kết quả ta nhận được như trên.

Giới thiệu về xâu kí tự

Một biến có thể dùng để lưu một loại dữ liệu không phải số, nhưng nó lại chứa nhiều kí tự (không như kiểu char) mà chúng ta sẽ gọi nó là kiểu xâu kí tự.

Trong thư viện ngôn ngữ lập trình C++, nó cung cấp cho chúng ta kiểu xâu nằm trong lớp string. Cần lưu ý rằng, để biểu diễn một biến thuộc kiểu xâu, chúng ta có thể sử dụng khai báo mảng kí tự, hoặc con trỏ kí tự như trong ngôn ngữ C, hoặc sử dụng khai báo kiểu string. Khi sử dụng kiểu khai báo tham chiếu của lớp string, ta cần có khai báo tệp tiêu đề là string.

Khai báo nguyên thủy	**Khai báo tham chiếu**
#include <iostream>	*#include <iostream>*
using namespace std;	*#include <string>*

```
int main()
{
  char a[] = “abc”;
  char* b = “abc”;
  return 0;
}
```

```
using namespace std;
int main()
{
  string a = “abc”;
  return 0;
}
```

Chúng ta cần lưu ý rằng, dù là một biến thuộc kiểu dữ liệu tham chiếu string, thì ta vẫn có thể sử dụng hai kiểu khởi tạo như trên. Điều này chỉ có thể áp dụng cho kiểu string mà thôi. ***Các kiểu dữ liệu tham chiếu khác không thể sử dụng hai cách khởi tạo này***.

Để biết thêm thông tin về kiểu string, các bạn nên tham khảo thêm thông tin về lớp string được cung cấp trong mục 4 của chương 17 trong cùng giáo trình này.

CHƯƠNG 3. HẰNG

Hằng: là một phần tử có giá trị cố định. Giá trị của nó được khởi tạo ngay khi hằng được tạo ra. Thông thường, người ta cũng sử dụng các chữ cái để đặt tên cho hằng. Tên hằng không chứa các kí tự đặt biệt, kí tự trắng hay bắt đầu bằng số, không được trùng với từ khóa. Trong C++, tên hằng thường được viết hoa toàn bộ. Hằng thường được chia ra làm: hằng số nguyên, hằng số thực, hằng kí tự, hằng xâu và hằng logic.

Hằng số nguyên

Hằng số nguyên là các hằng có giá trị là số nguyên. Hằng số nguyên có thể được biểu diễn dưới dạng thập phân, bát phân, hoặc thập lục phân.

Nếu hằng số nguyên dưới dạng thập phân thì có giá trị như số thập phân bình thường. Nếu là hằng số nguyên bát phân, thì nó bắt đầu bằng số 0 (ví dụ 011). Nếu là hằng số nguyên thập lục phân, thì nó bắt đầu bằng 0x (ví dụ 0x1b). Các quy tắc chuyển đổi số qua lại giữa các hệ đã được nghiên cứu trong học phần "Nhập môn tin học".

Nếu hằng số là số nguyên có dấu hoặc không dấu, có thể có một vài cách khai báo tương ứng.

Hằng nguyên có dấu và không dấu

```
75   //int
75u   //unsigned int
75l   //long
75ul   //unsigned long
```

Các tiền tố và hậu tố trong hai cách sử dụng ở trên có thể viết thường hoặc viết hoa (0x12 hay 0X12 là như nhau; hoặc 75ul hay 75UL là như nhau).

Hằng số thực có dấu chấm động

Chúng ta khảo sát số thực dưới dạng số thập phân dấu chấm động hoặc dưới dạng khoa học (hay còn gọi là dạng lũy thừa dạng E – tương ứng với lũy thừa 10). Ví dụ 314.159e-2 tương ứng với 3.14159 hay 6.02e23 tương ứng với $6.02*10^{23}$

Một hằng số thực mặc định là double. Nếu muốn chỉ định kiểu dữ liệu cho nó, ta có thể sử dụng cú pháp tương tự như đối với hằng số nguyên (3.1415L tương ứng long double, 3.1415F tương ứng với float).

Các kí tự e, f, l có thể biểu diễn dưới dạng chữ hoa hoặc chữ thường.

Hằng kí tự và hằng xâu kí tự

Hằng kí tự được sử dụng trong dấu nháy đơn, còn hằng xâu kí tự được sử dụng trong dấu nháy kép.

x	Tên biến
'x'	Kí tự x
"x"	Xâu kí tự x

Trong hằng xâu kí tự, có thể chứa các kí tự đặc biệt như kí tự xuống dòng, đặt tab… Sau đây là một vài kí tự đặc biệt đó và ý nghĩa của chúng.

Kí hiệu	Ý nghĩa
\n	Xuống dòng
\r	Di chuyển toàn bộ kí tự sau dấu \r đè lên các kí tự trước đó. Nếu số kí tự sau nhiều hơn số kí tự trước dấu \r, thì kết quả in ra sẽ là toàn bộ kí tự nằm sau. Ví dụ "abc\r1234" -> sẽ in ra **1234**, nếu "abc\r12" -> sẽ in ra **12c**.
\t	Đặt tab
\v	Đặt tab dọc
\b	Đặt backspace
\f	Đặt dấu form feed
\a	Tạo âm thanh beep
**\', \", \?, **	Tạo các kí tự ', ", ?, \

Một hằng xâu kí tự có thể chứa nội dung trên nhiều dòng. Khi đó, để viết nội dung ở dòng mới, thì cuối dòng trước đó, ta bổ sung thêm kí tự \. Các xâu kí tự có thể được ghép với nhau nhờ vào kí tự trắng.

Ví dụ

"Hom nay toi di hoc\ Ngay mai toi o nha"	Xâu kí tự viết trên nhiều dòng
"Toi " "yeu " "lap trinh"	Xâu kí tự ghép

Khi sử dụng hằng xâu kí tự với kiểu dữ liệu là wchar_t, ta cần thêm tiền tố L bên trước xâu kí tự đó. Ví dụ L"Xau ki tu wchar_t".

Các quy tắc ở trên có thể áp dụng cho bất kì hằng xâu kí tự thuộc kiểu dữ liệu nào (char*, wchar_t*, string hoặc mảng kí tự tương ứng).

Hằng logic

Hằng logic có hai giá trị là true (đúng) và false (sai). Một biểu thức logic sẽ có kiểu dữ liệu là bool. Nó chỉ có thể nhận một trong hai giá trị true và false. Trong C, ta chỉ có thể sử dụng kiểu số nguyên (short, int, long…) để quy định giá trị của biểu thức logic: nếu giá trị nhận được là 0 – tương ứng với giá trị sai; ngược lại, nếu giá trị nhận được là khác 0 – tương ứng với giá trị đúng. Cách quy định này vẫn còn hoạt động tốt trên C++. Tuy nhiên, trong C++, người ta đã định nghĩa hai hằng số true và false và kiểu dữ liệu bool. Hằng số true tương ứng với giá trị 1 và hằng số false tương ứng với giá trị 0. Ta hoàn toàn có thể sử dụng giá trị true (hoặc 1) và false (hoặc 0).

Định nghĩa một hằng #define

Khi định nghĩa một tên gọi cho hằng, ta có thể sử dụng nó thường xuyên mà không cần phải sắp xếp lại các biến chi phối bộ nhớ. Để định nghĩa một hằng, ta cần sử dụng từ khóa define.

Cú pháp	**Ví dụ**
#define tên_hằng giá_trị	#define PI 3.14 #define NewLine '\n'

Trong ví dụ trên, tôi đã định nghĩa hai hằng PI và Newline. Trong chương trình, tôi chỉ cần gọi nó để sử dụng, mà không cần triệu gọi đến giá trị cụ thể của nó.

Chương trình tính diện tích hình tròn	**Kết quả**
#include <iostream> using namespace std; #define PI 3.14 int main()	3.14

```
{
  double r = 1;
  double s;
  s = PI*r*r;
  cout<<s;
  return 0;
}
```

Thực chất, #define không phải là một câu lệnh trong C++, nó chỉ là một định hướng tiền xử lý (directive for the preprocessor). Cho nên, ta cần lưu ý rằng, kết thúc phần khai báo này, không có dấu chấm phẩy (;).

Khai báo hằng const

Để khai báo một hằng, ta sử dụng từ khóa const. Cấu trúc khai báo như sau:

Cú pháp	**Ví dụ**
const kiểu_dữ_liệu tên_hằng = giá_trị;	const int a = 10; const char x = '\t';

Trong ví dụ trên, ta có thể thấy cách khai báo hằng tương tự như khai báo biến, chỉ có duy nhất một điểm khác biệt là ta phải bổ sung từ khóa const vào trước khai báo này. Hằng và biến cũng tương tự nhau. Chúng chỉ khác nhau một điểm duy nhất là giá trị của hằng không thể thay đổi, còn biến thì có thể thay đổi.

CHƯƠNG 4. TOÁN TỬ

Chúng ta đã làm quen với biến và hằng, bây giờ chúng ta có thể tính toán giá trị của chúng. Các phép toán thực thi trên các biến hoặc hằng gọi là các toán tử. Và khi đó, các biến hoặc hằng đó gọi là các toán hạng.

Toán tử gán

Toán tử gán dùng để gán giá trị cho một biến. Ví dụ a = 5;

Câu lệnh gán sẽ thực hiện gán giá trị ở bên phải cho biến ở bên trái. Ta cũng có thể gán giá trị của hai biến cho nhau. Ví dụ a = b;

Hãy quan sát và suy ngẫm đoạn chương trình sau:

Chương trình

```
[1.]    #include <iostream>
[2.]    using namespace std;
[3.]    int main()
[4.]    {
[5.]      int a, b;
[6.]      a = 10;
[7.]      b = 4;
[8.]      a = b;
[9.]      b = 7;
[10.]     cout<<"a="<<a<<", b="<<b<<endl;
[11.]     return 0;
[12.]   }
```

Giải thích:

Dòng lệnh [5.] khai báo hai biến nguyên a, b. Khi đó giá trị của chúng chưa được khởi tạo. Dòng lệnh [6.] khởi tạo giá trị cho biến a là 10, biến b chưa được khở tạo. Dòng lệnh [7.] khởi tạo giá trị cho biến b là 4, biến a vẫn không thay đổi (10). Dòng lệnh [8.] thực hiện việc gán giá trị của biến b cho biến a, khi đó b vẫn không thay đổi; a nhận giá trị của b, tức là 4. Dòng lệnh

[9.] gán giá trị của biến b là 7, biến a không thay đổi. Do đó, giá trị cuối cùng của a là 4, b là 7. Output của chương trình sẽ là a=4, b=7.

Chúng ta cần chú ý rằng, toán tử gán thực hiện theo nguyên tắc phải-sang-trái. Nghĩa là luôn lấy giá trị ở vế phải để gán cho vế trái. Khi đó, giá trị của biến ở vế trái thay đổi, còn ở vế phải không thay đổi. Toán tử gán có thể thực hiện trong các biểu thức phức tạp hơn.

a = b + 2;	Giá trị của a bằng giá trị của b cộng thêm 2
a = a + 1;	Tăng giá trị của a lên 1

a = b = c = 5;	Gán đồng thời nhiều giá trị. Nó tương ứng với tập các lệnh sau:
	c = 5;
	b = c;
	a = b;

Toán tử thực hiện phép toán số học

Ngôn ngữ lập trình C++ hỗ trợ các toán tử số học sau:

Toán tử	Ý nghĩa
+	Phép cộng
-	Phép trừ
*****	Phép nhân
/	Phép chia (chia lấy phần nguyên đối với hai số nguyên)
%	Chia lấy dư (chỉ với hai số nguyên)

Chú ý rằng, phép chia có thể thực hiện trên số nguyên hoặc số thực. Nếu thực hiện phép chia trên hai số nguyên thì đây chính là kết quả của phép chia lấy phần nguyên. Còn nếu nó thực hiện trên hai số thực (hoặc một số thực và một số nguyên), thì kết quả được thực hiện trên phép chia bình thường. Như vậy, theo mặc định, hai số nguyên (hoặc thực) thực hiện phép toán tương ứng thì nó sẽ trả về kết quả nguyên (hoặc thực). Nếu phép toán thực hiện trên một số nguyên và một số thực, nó sẽ tự động chuyển đổi về kiểu cao hơn (thành số thực). Vậy làm thế nào để thực hiện phép chia 3 cho 2, nếu ta muốn nhận được kết quả là 1.5. Ta biết rằng 3 và 2 là hai số nguyên, nếu ta thực hiện phép chia 3/2 thì ta thu được số nguyên – là kết quả của phép chia lấy phần nguyên 3/2, tức là 1. Muốn thu được kết quả 1.5, ta cần chuyển đổi 3 và 2 về dạng số thực bằng một trong các cách sau:

- Khai báo 3 và 2 là các số thực (bằng cách quy định kiểu dữ liệu như float a = 3, float b = 2 hoặc 3.0, 2.0).
- Chuyển đổi kiểu dữ liệu (Xem thêm phần **toán tử chuyển đổi kiểu dữ liệu**).

Toán tử gán hợp nhất

Khi muốn thay đổi giá trị của một biến, chúng ta có thể sử dụng cách viết thông thường, nhưng trong C++ nó hỗ trợ các toán tử viết tắt.

Toán tử	Ví dụ	Ý nghĩa	Phạm vi
+=	a+=b	a=a+b	Phép toán số học
-=	a-=b	a=a-b	Phép toán số học
***=**	a*=b	a=a*b	Phép toán số học
/=	a/=b	a=a/b	Phép toán số học
%=	a%=b	a=a%b	Phép toán số học
&=	a&=b	a=a&b	Phép toán bit
|=	a|=b	a=a|b	Phép toán bit
^=	a^=b	a=a^b	Phép toán bit
>>=	a>>=b	a=a>>b	Phép toán bit
<<=	a<<=b	a=a<<b	Phép toán bit

Toán tử tăng và giảm

Một cách viết thu gọn hơn nữa, đó là sử dụng toán tử tăng và giảm. Nếu trong biểu thức a+=b, với b = 1 thì ta có thể viết thành a++. Tương tự, nếu a-=b, b = 1 thì ta có thể viết a--.

Chúng ta cũng lưu ý rằng, toán tử này có chút khác biệt. Nó có thể nằm trước hoặc nằm sau toán hạng. Có nghĩa là có thể có a++ hoặc ++a (tương ứng a-- hoặc --a).

Phép toán	Ý nghĩa
a++;	Thực hiện phép toán trước, sau đó mới thực hiện toán tử.
++a;	Thực hiện toán tử trước, sau đó mới thực hiện phép toán.
a--;	Tương tự a++;
--a;	Tương tự ++a;

Ví dụ	**Cách thực thi**
int a = 1;	a = 1, b chưa khởi tạo

int b = 1;	a = 1, b = 1
a+=b++;	Thực hiện phép toán a+=b trước, sau đó mới thực hiện phép toán b++. Tức là a=2, b=2.
a+=++b;	Thực hiện phép toán ++b trước, sau đó mới thực hiện phép toán a+=b. Tức là b=2, a=3.

Toán tử so sánh

Để thực hiện việc so sánh giá trị của hai biến hoặc hai biểu thức; ta có thể sử dụng toán tử so sánh. Giá trị của phép toán so sánh trả về kiểu bool.

Toán tử	Tên gọi	Giá trị biểu thức "*a Toán tử b*"	
		Đúng	***Sai***
==	Bằng	Nếu a bằng b	Nếu a khác b
!=	Khác	Nếu a khác b	Nếu a bằng b
>	Lớn hơn	Nếu a lớn hơn b	Nếu a nhỏ hơn hoặc bằng b
<	Bé hơn	Nếu a nhỏ hơn b	Nếu a lớn hơn hoặc bằng b
>=	Lớn hơn hoặc bằng	Nếu a lớn hơn hoặc bằng b	Nếu a nhỏ hơn b
<=	Bé hơn hoặc bằng	Nếu a nhỏ hơn hoặc bằng b	Nếu a lớn hơn b

Ví dụ	**Kết quả**

#include <iostream>
using namespace std;
int main()
{
  int a = 1;
	Kết quả 1: 1
	Kết quả 2: 0
	Kết quả 3: 1

```
  int b =2;
  cout<<"Kết quả 1:"<<(a==a);
  cout<<"Kết quả 2:"<< (a>=b);
  cout<<"Kết quả 3:"<< (a<=b);
}
```

Ta cần chú ý trong ví dụ này, cũng giống C, C++ chấp nhận giá trị 0 và 1 để quy định cho kiểu logic. Theo quy ước: true tương ứng với giá trị 1 (hoặc khác 0), false tương ứng với giá trị 0. Mặc dù C++ hỗ trợ kiểu dữ liệu bool, nhưng nó vẫn dùng số nguyên 0 và 1 để biểu diễn tính đúng sai. Ta có thể tạm hiểu true và false là hai hằng số đã được định nghĩa sẵn, tương ứng với 1 và 0 (nhờ vào #define). Nếu mong muốn in ra giá trị là true/false, ta cần sử dụng định dạng dữ liệu cho đối tượng cout. Chi tiết, hãy tham khảo mục 8, chương 17 trong giáo trình này.

Chú ý:

- Hãy sử dụng kiểu dữ liệu bool thay vì dùng kiểu dữ liệu int để biểu diễn tính đúng sai. Khi sử dụng kiểu bool, hãy nhớ rằng giá trị đúng tương ứng với true; giá trị sai tương ứng với false (chú ý chúng được viết thường).
- Hãy cẩn thận khi sử dụng toán tử so sánh bằng. Hãy chú ý rằng toán tử so sánh bằng là ==, khác với toán tử gán =.

Toán tử logic

	Phép toán	a	b	Kết quả
Toán tử phủ định !	*Phép toán một ngôi* ***!a***	true	-	false
		false	-	true
Toán tử hội &&	*Phép toán hai ngôi* ***a&&b***	true	true	true
		true	false	false
		false	true	false
		false	false	false
Toán tử tuyển ||	*Phép toán hai ngôi* ***a||b***	true	true	true
		true	false	true
		false	true	true
		false	false	false

Ví dụ	**Kết quả**

#include <iostream>
using namespace std;
int main()
{
  int a = true;
  int b =false;
  cout<<"Kết quả 1:"<<(a&&a);
  cout<<"Kết quả 2:"<< (!a&&b);
  cout<<"Kết quả 3:"<< !(a||b);
}
	Kết quả 1: 1
	Kết quả 2: 0
	Kết quả 3: 0

***Giải thích*:**

Kết quả 1 – tương ứng với biểu thức a&&a=a, nghĩa là true – 1.

Kết quả 2 – tương ứng với !a&&b. !a=false, false&&false=false – 0.

Kết quả 3 – tương ứng với !(a||b), a||b=true||false=true, !(a||b)=!true=false – 0.

Bài tập 3.
Hãy lập trình kiểm tra tính đúng đắn của định luật De Morgan:

a. !(a||b)=!a&&!b
b. !(a&&b)=!a||!b

Toán tử điều kiện

Toán tử điều kiện có dạng cú pháp như sau:

(bt_điều_kiện)?(kết_quả_1):(kết_quả_2);

***Giải thích*:** trả về giá trị **kết_quả_1** nếu **bt_điều_kiện** là đúng, ngược lại, nếu **bt_điều_kiện** là sai, thì trả về giá trị **kết_quả_2**.

Chương trình	**Kết quả**

#include <iostream>
using namespace std;
int main()
{
  int a = 1;
  int b = 2;
  int max = (a>b)?a:b;
  cout<<"Max là: "<<max;
  return 0;
}
	Max là: 2

Giải thích: chương trình trên tính giá trị lớn nhất giữa hai số a và b. Toán tử điều kiện kiểm tra điều kiện của biểu thức a>b, vì a=1, b=2, nên giá trị của nó là false. Chính vì vậy, biểu thức điều kiện sẽ nhận kết quả tương ứng với kết quả 2, tức là b.

Toán tử điều kiện luôn trả về một giá trị cụ thể. Như trong ví dụ trên, ta thấy nếu biểu thức a>b đúng, thì giá trị max nhận được là số a; ngược lại là số b. Tuy nhiên, không nhất thiết cần phải có một giá trị xác định cho toán tử điều kiện. Ví dụ sau đây sẽ minh họa điều này

Chương trình	**Kết quả**

#include <iostream>
using namespace std;
int main()
{
  int a = 1;
  int b = 2;
  (a>b)?(cout<<a<<" lon hon"):(cout<<b<<" lon hon");
  return 0;
}
	2 lon hon

Giải thích: trong ví dụ minh họa này, toán tử điều kiện không trả về một giá trị cụ thể nào. Nó chỉ đơn thuần kiểm tra điều kiện, nếu a>b đúng, thì in ra câu **a lớn hơn**, ngược lại sẽ in ra câu **b lớn hơn**. Ta cần lưu ý rằng, khi các câu lệnh nằm trong cặp dấu ngoặc của toán tử điều kiện, thì kết thúc câu lệnh không bao giờ có dấu chấm phẩy (;).

Nếu muốn sử dụng một tập các câu lệnh trong cặp dấu ngoặc này, ta có thể sử dụng toán tử phân tách được đề cập trong mục tiếp theo. Ví dụ sau đây sẽ cho thấy việc sử dụng tập các câu lệnh bên trong cặp dấu ngoặc của toán tử điều kiện.

Chương trình	Kết quả

#include <iostream>
using namespace std;
int main()
{
  int a = 1;
  int b = 2;
  int c;
  (a>b)?(c = a-b,cout<<"|a-b|="<<c):(c = b-a,cout<<"|a-b|="<<c);
  return 0;
}
	|a-b|=1

Giải thích: Trong ví dụ này, chương trình sẽ in ra giá trị tuyệt đối của a-b. Nếu a>b, thì giá trị tuyệt đối |a-b| = a-b; ngược lại nếu a<b, thì giá trị tuyệt đối |a-b| = b-a. Trong cặp dấu ngoặc đơn của toán tử điều kiện, câu lệnh gán c=a-b (hoặc c=b-a) và cout được phân tách bằng dấu phẩy (,). Một điều cần lưu ý, chúng ta **không được phép** khai báo biến trong cặp dấu ngoặc đơn này. Việc khai báo biến trong dấu ngoặc đơn, chỉ áp dụng duy nhất cho câu lệnh lặp for.

Toán tử phân tách

Toán tử này kí hiệu là **dấu phẩy**. Nó dùng để phân tách hai hay nhiều biểu thức chứa trong một biểu thức phức hợp tương ứng.

Ví dụ	Kết quả
… int a; int b; int c; c = (a=1, b=2, a+b); cout<<c; …	3

***Giải thích*:** trong biểu thức phức hợp, bên trong dấu ngoặc đơn là các biểu thức đơn được phân tách bằng toán tử phân tách. Trong một dãy các toán tử phân tách, nó sẽ ưu tiên thực hiện từ trái sang phải (xem thêm phần độ ưu tiên của toán tử được trình bày trong mục sau của chương này), nghĩa là a=1, sau đó b=2 và cuối cùng là c=a+b=1+2=3.

Toán tử dịch bit

Các toán tử này được sử dụng để tính toán trên các số nguyên bằng cách tính toán trên các bit.

<table>
<tr><th>Toán tử</th><th>Kết quả</th></tr>
<tr><td>~</td><td>Toán tử phủ định bit. Các bit 1 sẽ chuyển thành 0 và ngược lại.
Ví dụ ~101=010.</td></tr>
<tr><td>&</td><td>Toán tử hội bit. Hội của hai bit 1 bằng 1. Trong mọi trường hợp còn lại, ta nhận được 0.
Ví dụ.
<table>
<tr><td></td><td>1</td><td>0</td><td>1</td><td>1</td><td>(tương ứng 11 trong hệ thập phân)</td></tr>
<tr><td>&</td><td>0</td><td>1</td><td>0</td><td>1</td><td>(tương ứng với 5 trong hệ thập phân)</td></tr>
<tr><td></td><td>0</td><td>0</td><td>0</td><td>1</td><td>(tương ứng với 1 trong hệ thập phân)</td></tr>
</table>
Nghĩa là 11&5=1.</td></tr>
<tr><td>|</td><td>Toán tử tuyển bit. Tuyển của hai bit 0 bằng 0. Trong mọi trường hợp còn lại, ta nhận được 1.
Ví dụ.
<table>
<tr><td></td><td>1</td><td>0</td><td>1</td><td>1</td><td>(tương ứng 11 trong hệ thập phân)</td></tr>
<tr><td>|</td><td>0</td><td>0</td><td>0</td><td>1</td><td>(tương ứng với 1 trong hệ thập phân)</td></tr>
<tr><td></td><td>1</td><td>0</td><td>1</td><td>1</td><td>(tương ứng với 11 trong hệ thập phân)</td></tr>
</table>
Nghĩa là 11|1=11.</td></tr>
<tr><td>^</td><td>Toán tử tuyển loại bit. Tuyển loại của hai bit khác nhau bằng 1. Trong mọi trường hợp còn lại, ta nhận được 0.
Ví dụ.
<table>
<tr><td></td><td>1</td><td>0</td><td>1</td><td>1</td><td>(tương ứng 11 trong hệ thập phân)</td></tr>
<tr><td>^</td><td>0</td><td>0</td><td>0</td><td>1</td><td>(tương ứng với 1 trong hệ thập phân)</td></tr>
<tr><td></td><td>1</td><td>0</td><td>1</td><td>0</td><td>(tương ứng với 10 trong hệ thập phân)</td></tr>
</table>
Nghĩa là 11^1=10.</td></tr>
<tr><td>>></td><td>Toán tử dịch bit sang phải. Dịch chuyển toàn bộ dãy bit sang phải theo số bit được chỉ định. Nếu là số dương, ta bổ sung các bit 0 vào đầu. Nếu là số âm, ta bổ sung các số 1 vào đầu.</td></tr>
</table>

<table>
<tr>
<td></td>
<td>
Ví dụ.
<u>Đối với số dương</u>
<table>
<tr><td>...</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>(tương ứng 11 trong hệ thập phân)</td></tr>
<tr><td>>></td><td></td><td></td><td></td><td></td><td>1</td><td>dịch sang phải 1 bit</td></tr>
<tr><td>...</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>(tương ứng với 5 trong hệ thập phân)</td></tr>
</table>
Nghĩa là 11>>1=5.
<u>Đối với số âm</u>
<table>
<tr><td>...</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>(tương ứng -11 trong hệ thập phân)</td></tr>
<tr><td>>></td><td></td><td></td><td></td><td></td><td></td><td>2</td><td>dịch sang phải 2 bit</td></tr>
<tr><td>...</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>(tương ứng -3 trong hệ thập p ân)</td></tr>
</table>
Nghĩa là -11>>2=-3.
Các bạn cũng cần lưu ý rằng, trong các biểu diễn ở trên, nếu hệ thống được chọn là 32 bit, thì chúng ta cần lấp đầy số bit này:
- Nếu số dương thì các bit còn lại sẽ được bổ sung 0 vào phía trước.
- Nếu số âm thì các bit còn lại sẽ được bổ sung 1 vào phía trước.
Trong các ví dụ trên, phần dãy bit để trắng tương ứng với bit dấu – 1 tương ứng với – và 0 tương ứng với +.
</td>
</tr>
<tr>
<td><<</td>
<td>
Toán tử dịch bit sang trái. Dịch chuyển toàn bộ dãy bit sang trái theo số bit được chỉ định.
Ví dụ.
<u>Đối với số dương</u>
<table>
<tr><td>+</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>(tương ứng 11 trong hệ thập phân)</td></tr>
<tr><td><<</td><td></td><td></td><td></td><td></td><td></td><td>2</td><td>dịch sang trái 2 bit</td></tr>
<tr><td></td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>(tương ứng 44 trong hệ thập phân)</td></tr>
</table>
Nghĩa là 11<<2=44.
</td>
</tr>
</table>

&= |= ^= >>= <<=	Các phép toán gán hợp nhất trên bit.

Các toán tử hội bit, tuyển bit, tuyển loại bit và phủ định bit được tính như sau: chúng ta chuyển đổi các số thập phân sang nhị phân tương ứng, sau đó sử dụng các phép toán tương ứng cho từng bit theo vị trí của nó. Ví dụ như ở trên $2_{10}=10_2$, $3_{10}=11_2$ và ta sẽ thực hiện các phép toán tương ứng với từng bit. Bit thứ nhất (từ phải sang trái) là 0&1=1, bit thứ hai 1&1=1, như vậy kết quả của phép toán 2&3 là 11_2 hay 3_{10}. Tương tự cho các phép toán còn lại. Nếu hai số có độ dài bit khác nhau, thì ta chỉ việc bổ sung thêm 0 ở số có độ dài bit ngắn hơn, sau đó thực hiện các phép toán như đã nêu. Trong trường hợp này, ta cần lưu ý rằng phép toán tuyển loại sẽ có chân trị là 1 nếu hai bit tương ứng là khác nhau, nếu giống nhau thì tương ứng là 0(1^1=0^0=0, 1^0=0^1=1). Các phép toán hội, tuyển và phủ định vẫn còn đúng như phép toán hội, tuyển và phủ định trên kiểu dữ liệu logic.

Các toán tử dịch trái bit << và dịch phải bit >> nếu thực hiện trực tiếp trên số nguyên hệ thập phân, sẽ được tính như sau: $a<<b=a*2^b$ và $a>>b=a/2^b$.

Toán tử chuyển đổi kiểu dữ liệu

Toán tử này dùng để chuyển đổi một biến hay hằng thuộc kiểu dữ liệu này sang kiểu dữ liệu khác. Giả sử ta có biến int a = 3, int b = 2. Khi thực hiện phép chia để nhận được kết quả thực, ta chỉ cần viết như sau: (float)3/2. Hãy lưu ý rằng số 3 ở đây đã bị chuyển thành kiểu thực, và việc thực hiện phép chia một số thực cho số nguyên sẽ trả về kiểu thực 1.5. Nếu ta viết 3/(float)2, kết quả cũng tương tự. Ngược lại, nếu viết (float)(3/2) thì kết quả lại khác. Sở dĩ như vậy là vì, nó sẽ thực hiện phép chia nguyên 3/2 (kết quả là 1), sau đó nó sẽ chuyển giá trị 1 nguyên này sang 1 thực. Do đó, giá trị thu được vẫn là 1, nhưng thuộc kiểu số thực (tức là 1.0f).

Cách biểu diễn sự chuyển đổi một biến thuộc kiểu dữ liệu này, sang kiểu khác chỉ có thể thực hiện nếu kiểu của chúng tương đương. Ta có thể chuyển số thành số (sau này khi học về hướng đối tượng, ta có thể chuyển

giữa các đối tượng trong cùng một cây phả hệ). Ta không thể chuyển đổi từ số thành xâu, hay ngược lại (bằng cách thực hiện phép toán chuyển đổi kiểu). Ta có thể chuyển đổi một xâu số thành số và một số thành xâu số bằng nhiều cách khác nhau, nhưng việc sử dụng toán tử chuyển đổi kiểu là không được phép. Khi chuyển đổi, ta sử dụng một trong các cú pháp sau: ***(kiểu_dữ_liệu)biến hoặc (kiểu_dữ_liệu)(biến) hoặc kiểu_dữ_liệu(biến)***. Chúng ta nên sử dụng kiểu thứ 2 hoặc 3 để tránh các nhầm lẫn đáng tiếc khi biểu thức phức tạp.

Các toán tử khác

Trong phần lập trình hướng đối tượng, chúng ta sẽ làm quen thêm nhiều toán tử khác. Theo trình tự trình bày trong cuốn giáo trình này, chúng ta sẽ chưa thảo luận thêm về chúng. Ta sẽ tìm hiểu chi tiết trong phần hướng đối tượng của cuốn giáo trình này.

Thứ tự ưu tiên của các toán tử

Trong toán học, chúng ta biết rằng khi tính giá trị của một biểu thức, thì luôn có sự ưu tiên của các toán tử như: phép nhân thực hiện trước phép cộng, phép chia và nhân thực hiện đồng thời, ưu tiên từ trái sang phải… Trong các ngôn ngữ lập trình nói chung cũng như C++ nói riêng, các toán tử cũng có những độ ưu tiên nhất định. Trong một biểu thức phức tạp, ta cần chú ý đến độ ưu tiên của các toán tử, điều này rất dễ gây ra sai sót. Trong bảng sau đây, tôi xin đưa ra thứ tự ưu tiên của các toán tử trong lập trình C++.

Mức ưu tiên	Toán tử	Độ ưu tiên cùng loại
1	::	Trái-sang-phải
2	() [] . -> ++ -- (**hậu tố**) dynamic_cast static_cast reinterpret_cast const_cast typeid	Trái-sang-phải
3	++ -- (**tiền tố**) ~ ! sizeof new delete	Phải-sang-trái
	* &	
	+ - (**dấu dương âm**)	
4	(type) (**chuyển đổi kiểu**)	Phải-sang-trái
5	.* ->*	Trái-sang-phải
6	* / %	Trái-sang-phải
7	+ - (**phép toán công, trừ**)	Trái-sang-phải
8	<< >>	Trái-sang-phải
9	< > <= >=	Trái-sang-phải

10	== !=	Trái-sang-phải
11	&	Trái-sang-phải
12	^	Trái-sang-phải
13	|	Trái-sang-phải
14	&&	Trái-sang-phải
15	||	Trái-sang-phải
16	?:	Phải-sang-trái
17	= *= /= %= += -= >>= <<= &= ^= |=	Phải-sang-trái
18	,	Trái-sang-phải

Các toán tử được thực hiện theo **mức ưu tiên** từ trên xuống. Nếu các toán tử cùng mức, nó sẽ được thực hiện theo **độ ưu tiên cùng loại**.

Ví dụ:

- **a = (b=0, c=0, b+c)**. Toán tử gán = có độ ưu tiên 17, các toán tử cộng + có độ ưu tiên 7, toán tử () có độ ưu tiên 2 và toán tử , có độ ưu tiên 18. Do đó, toán tử () sẽ được thực hiện trước. Bây giờ ta xét các toán tử trong dấu (), chú ý rằng các biểu thức b=0, c=0, b+c là các biểu thức riêng biệt, chúng được phân tách bởi toán tử phân tách (,). Theo thứ tự ưu tiên của toán tử phẩy, nó sẽ thực hiện từ trái-sang-phải. Nghĩa là b=0, c=0 sau đó là b+c. Cuối cùng nó sẽ thực hiện toán tử gán giá trị của biểu thức phức hợp bên phải cho bên trái. Kết quả là 0.
- **a = (1+2)*3/2++**. Toán tử gán (độ ưu tiên 17), toán tử + (độ ưu tiên 7), toán tử * (độ ưu tiên 6), toán tử / (độ ưu tiên 6), toán tử ++ hậu tố (độ ưu tiên 2) và toán tử () (độ ưu tiên 2). Toán tử hậu tố ++ và toán tử () sẽ thực hiện trước. Theo độ ưu tiên cùng loại, nó sẽ thực thi từ trái-sang-phải. Như vậy, toán tử () sẽ được thực hiện đầu tiên. Nghĩa là ta nhận được biểu thức a = 3*3/2++. Tiếp theo, nó thực hiện toán tử hậu tố ++, tuy nhiên toán tử này chỉ tăng giá trị của 2 lên 1 sau khi thực hiện xong các phép toán trong biểu thức. Đến thời điểm này, ta nhận được biểu thức a=3*3/2. Toán tử * và / có cùng độ ưu tiên, nó sẽ được thực hiện theo thứ trự từ trái sang phải, nghĩa là a=9/2=4. Kết quả 4.

Lưu ý. Trong ví dụ thứ hai, việc sử dụng phép toán 2++ là không hợp lệ. Ở đây, chỉ có tác dụng minh họa trực quan. Còn 2 là một hằng số, ta không thể thực hiện phép toán 2++ để làm thay đổi giá trị của hằng. Trong C++, chúng ta cần thực hiện phép gán b = 2; sau đó là b++. Nghĩa là ta cần biểu diễn biểu thức như sau để thu được một kết quả chính xác có thể bảo đảm thực thi được trong C++.

a = (b=2, (1+2)*3/b++)

Bài tập 4.
Tính toán các biểu thức sau, dựa vào độ ưu tiên của toán tử, sau đó, viết chương trình trên C++ để kiểm tra kết quả.

a. 2+2*4%3+ ++2;
b. 2++ + ++2*(3-- - --2)
c. 5++ - 3== --2-(4+2%3)
d. 5>>2^3*(1+2)

CHƯƠNG 5. XUẤT NHẬP CƠ BẢN

Đến thời điểm này, chúng ta đã biết hai cách thức để xuất dữ liệu ra màn hình nhờ vào việc sử dụng đối tượng cout và hàm printf. Trong chương này, chúng ta sẽ tìm hiểu cụ thể hơn về cách xuất-nhập dữ liệu nhờ vào thiết bị nhập dữ liệu là bàn phím, và thiết bị hiển thị dữ liệu xuất ra là màn hình máy tính. Trong thư viện chuẩn của C++, các hàm xuất nhập cơ bản nằm trong tệp header là iostream.

Đối với thư viện này ta cần lưu ý một số điểm. Có hai lớp thư viện có chức năng hỗ trợ các hàm xuất nhập cơ bản đó là iostream và iostream.h. Về bản chất, cách thức sử dụng chúng không có nhiều sự khác biệt. Tuy nhiên việc sử dụng thư viện iostream có nhiều ưu điểm hơn hẳn so với thư viện iostream.h. Thư viện iostream.h đã ra đời cách đây quá lâu (trên 15 năm) trong khi thư viện iostream mới hơn rất nhiều. Việc sử dụng một thư viện mới, chuẩn mực hơn bao giờ cũng tốt hơn. Thư viện iostream hỗ trợ cả kiểu char lẫn kiểu w_char. Thư viện iostream được đặc tả trong namespace std, trong khi thư viện iostream.h được khai báo toàn cục. Việc khai báo toàn cục bao giờ cũng chiếm dụng không gian bộ nhớ lớn hơn. Vì vậy, nếu muốn thực hiện việc nhập xuất dữ liệu cơ bản trong C++, ta nên sử dụng thư viện iostream thay vì sử dụng iostream.h.

Tổng quát hóa, nếu các lớp thư viện có hai dạng tồn tại song song là .h và không có .h (string và string.h, new và new.h…), thì chúng ta nên sử dụng thư viện không có .h. Trong trường hợp không có dạng tương ứng, ta bắt buộc phải sử dụng thư viện .h (ví dụ math.h).

Xuất dữ liệu chuẩn cout

Đầu tiên, ta khảo sát việc xuất dữ liệu ra màn hình nhờ đối tượng cout. Nó được sử dụng kết hợp với **toán tử chèn dữ liệu >>** (kí hiệu giống toán tử dịch bit phải).

Cú pháp:

```
cout<<biến_1<<...<<biến_n;
```

Trong đó, biến_1,…,biến_n: là các biến số. Chúng đã được khởi tạo giá trị. Nếu biến chưa khởi tạo giá trị, ta sẽ nhận được một lỗi khi thực thi chương trình. Chương trình dịch sẽ thông báo về việc sử dụng biến mà

không khởi tạo giá trị cho nó. Các biến này có thể là biến thuộc kiểu dữ liệu nguyên thủy hoặc tham chiếu. Đối với các biến là các đối tượng thể hiện của các lớp, ta sẽ thảo luận ở mục "chồng chất toán tử nhập xuất" trong chương lập trình hướng đối tượng. Sau đây là một vài ví dụ về việc sử dụng đối tượng cout:

```
cout<<"Hello, world !";//In câu Hello, world ! ra màn hình
cout<<120;//In số 120 ra màn hình
cout<<x;//In giá trị của biến x ra màn hình
```

Đối tượng cout kết hợp với toán tử << có thể được ghép nhiều lần.

```
cout<<"Chao ban"<<" ban may tuoi";
cout<<"Chuc mung"<<endl;
cout<<x<<"+"<<y<<"="<<(x+y);
```

Như trong ví dụ trên, muốn xuống dòng, ta sử dụng kí hiệu endl, hoặc ta có thể sử dụng kí hiệu \n mà ta đã làm quen trong chương trước. Về mặt ngữ nghĩa, thì không có một sự khác nhau nào trong hai cách viết này. Khi làm việc với đối tượng cout, ta có một số cách thức định dạng dữ liệu được cung cấp ở mục 8 của chương 17 ở cuối giáo trình này.

Nhập dữ liệu chuẩn cin

Để nhập dữ liệu ta có thể sử dụng hàm scanf như đối với C. Nhưng theo xu hướng lập trình C++ hiện đại, hãy sử dụng đối tượng cin. Nó được sử dụng kết hợp với **toán tử trích tách dữ liệu <<** (giống toán tử dịch bit phải). Sau toán tử này, bắt buộc là một biến để lưu dữ liệu được tách ra.

Cú pháp:

```
cin>>biến_1>>...>>biến_n;
```

Các biến số biến_1,..., biến_n cần được khai báo. Thông thường, những biến này chưa được khởi tạo giá trị. Sau đây là một vài ví dụ về việc sử dụng đối tượng cout:

```
int age;
cin>>age;
float f;
cin>>f;
string s;
cin>>s;
```

Chú ý rằng kiểu dữ liệu của biến được sử dụng trong đối tượng cin này. Nếu có một sự vi phạm nào về kiểu dữ liệu (ví dụ biến là int, nhưng khi

nhập ta lại nhập vào một kí tự không phải là số) thì chương trình dịch sẽ bỏ qua việc khởi tạo giá trị cho biến đó. Chương trình hoàn toàn không phát sinh lỗi (process returned 0).

Khi sử dụng đối tượng cout và cin, ta cần khai báo không gian sử dụng namespace là std. Hoặc, có thể viết ngắn gọn hơn std::

Chương trình 1	**Chương trình 2**

#include <iostream>
using namespace std;
int main(){
  cout<<"Hello";
}
	#include <iostream>
	int main(){
	  std::cout<<"Hello";
	}

Đối tượng cin và xâu kí tự: trong ví dụ trên, tôi đã sử dụng đối tượng cin để tách một xâu kí tự và gán cho biến xâu kí tự s. Khi sử dụng đối tượng cin với xâu kí tự, cần lưu ý một điểm: đối tượng cin sẽ dừng việc trích tách nếu nó đọc thấy một kí tự trắng trong xâu kí tự đó (có nghĩa, nếu xâu nhập vào là "Toi di hoc" – thì nó chỉ tách được xâu "Toi" và gán cho biến s). Để khắc phục nhược điểm này của đối tượng cin, C++ cung cấp cho chúng ta một hàm khác là hàm getline, có chức năng tương tự.

Cú pháp:

getline(chuẩn_nhập_dữ_liệu, tên_biến_xâu)

Khi nhập xuất dữ liệu từ bàn phím và màn hình, tham số *chuẩn_nhập_xuất_dữ_liệu* luôn sử dụng là **cin**. Nếu làm việc với tập tin file, thì tham số này sẽ tương ứng với tên của file. Chúng ta sẽ tìm hiểu trường hợp này trong chương 16 của giáo trình.

Chương trình

```
#include <iostream>

#include <string>

using namespace std;

int main()

{
```

```
  string s;
  cout<<"Nhap ten: ";
  getline(cin, s);
  cout<<"Chao ban "<<s;
  return 0;
}
```

Nhập dữ liệu nhờ lớp stringstream

Để sử dụng lớp stringstream, chúng ta cần khai báo tệp header là <sstream>. Lớp này cho phép một đối tượng dựa trên cơ sở của xâu có thể được chuyển đổi như một luồng stream. Trong thư viện sstream, ta có ba lớp đối tượng luồng xâu cơ bản: stringstream, istringstream và ostringstream. Các luồng này có thể được sử dụng để tách hoặc chèn xâu, nó đặc biệt hữu dụng khi chuyển một xâu thành số và ngược lại. Ví dụ, nếu muốn tách một số integer từ một xâu "1201", ta có thể viết như sau:

```
string mystr = "1201";
int mynum;
stringstream(mystr)>>mynum;
```

Đây không phải là cách thức duy nhất giúp ta chuyển đổi một xâu thành số. Trong thư viện string, cung cấp cho chúng ta các hàm để thực thi công việc đó như hàm atof (xâu số thực thành số thực), atoll (xâu thành số nguyên dài thành số nguyên dài),... Tuy nhiên, các hàm này chủ yếu làm việc với xâu kí tự của C, tức là char*.

Ví dụ sau đây sẽ cho thấy cách sử dụng lớp stringstream để nhập dữ liệu từ bàn phím.

Chương trình

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

```
{
  string mystr;
  int mynum;
  getline(cin,mystr);
  stringstream(mystr)>>mynum;
  cout<<mynum;
  return 0;
}
```

Thay vì trực tiếp trích lọc số nguyên nhập vào, ta sử dụng hàm getline để trích lọc dữ liệu nhập vào dưới dạng xâu kí tự. Sau đó, chúng ta sử dụng lớp stringstream để chuyển đổi xâu đó thành số. Ví dụ sau đây cho phép chúng ta nhập vào một dãy giá trị, sau đó in ra tổng của các số vừa nhập. Số nhập vào được phân tách với nhau bằng dấu hai chấm :.

Chương trình	**Kết quả**

#include <iostream>
#include <sstream>
 	
using namespace std;
 	
int main()
{
string temp, s;
getline(cin, s);
istringstream ss(s);
double sum = 0;
while (getline(ss, temp, ':'))
{
float a;
stringstream(temp)>>a;
sum+=a;
}
cout<<"Sum = "<<sum;
return 0;
}
	1:2:3:4:5
	Sum = 15

Giải thích: trong ví dụ trên, hàm getline có thêm một biến thể (sau này, chúng ta sẽ gọi chúng là các chồng chất hàm). Nó có tất cả 4 dạng biến thể. Phương pháp này, thường được sử dụng khi dữ liệu nhập vào quá nhiều. Chúng ta có thể cho người dùng nhập vào thành một xâu, khi đó ta sẽ tiến hành xử lý nhờ lớp istringstream này.

Để có cái nhìn cụ thể hơn về các lớp stringstream, istringstream, ostringstream; hãy tham khảo thêm thông tin trong phần trợ giúp của MSDN[4] (Microsoft Developer Network).

[4] http://www.msdn.microsoft.com/en-us/visualc/default.aspx

CHƯƠNG 6. CÁC CẤU TRÚC LỆNH ĐIỀU KHIỂN

Một chương trình khi thực thi, nó không chỉ đơn thuần là một dãy các câu lệnh tuần tự. Trong quá trình xử lý, nó có thể kiểm tra điều kiện rồi thực thi đoạn mã, lặp đi lặp lại một đoạn mã nào đó… Với mục đích đó, C++ cung cấp cho chúng ta các cấu trúc điều khiển.

Với các cấu trúc điều khiển mà chúng ta sẽ tìm hiểu trong chương này, ta sẽ đề cập đến hai khái niệm: câu lệnh (mệnh đề - statement) và khối lệnh (block).

Câu lệnh: là một lệnh được kết thúc bằng dấu chấm phẩy (;). Nó có thể là lệnh đơn giản hoặc lệnh có cấu trúc. Các lệnh đơn giản như nhập xuất dữ liệu; các khai báo biến, hằng; lệnh gán… Các lệnh gán, xuất nhập…là các lệnh đơn. Các lệnh điều kiện, lựa chọn, lặp mà chúng ta tìm hiểu trong chương này là các lệnh có cấu trúc.

Khối lệnh: là một dãy các câu lệnh. Trong C++, khối lệnh được đặt trong cặp dấu {}.

Cấu trúc lệnh có điều kiện: if và else

Từ khóa if thường được sử dụng khi muốn thực thi một đoạn chương trình với một điều kiện nào đó. Cấu trúc của câu lệnh if trong trường hợp này

Cú pháp:

```
if (biểu_thức_điều_kiện_đúng)
{
  Các_lệnh;
}
```

Giải thích: kiểm tra giá trị của biểu thức điều kiện, nếu đúng thì các lệnh bên trong sẽ được thực hiện; ngược lại, lệnh sẽ không được thực hiện.

Ví dụ

```
if (x>0)
  cout<<x<<" la so duong";
if((x>0)&&(y>0))
{
   cout<<x<<" la so duong"<<endl;
   cout<<y<<" la so duong";
}
```

Nếu có nhiều câu lệnh chịu sự chi phối của câu lệnh if, thì chúng sẽ được đặt trong khối lệnh.

Trong trường hợp nếu biểu_thức_điều_kiện sai, ta cần thực thi một mệnh đề khác. Khi đó, ta sẽ sử dụng thêm từ khóa else.

Cú pháp:

```
if(biểu_thức_điều_kiện_đúng)
{
   ...
}else
{
   ...
}
```

Ví dụ

```
if (x%2==0)
   cout<<x<<" la so chan";
else
   cout<<x<<" la so le";
```

Cấu trúc if (lẫn else) có thể lồng vào nhau. Khi đó, chúng ta sẽ có một cấu trúc lệnh phức hợp.

Ví dụ

```
if(x>0)
```

```
  cout<<x<<" la so duong";
else if(x<0)
  cout<<x<<" la so am";
else
  cout<<x<<" la so 0";
```

Bài tập 5.

1. Viết chương trình giải phương trình bậc 2, với các hệ số nhập vào từ bàn phím.
2. Viết chương trình tính giá trị của hàm số sau:

$$f(x) = \begin{cases} \dfrac{1}{x}, x \neq 0 \\ \ln(|\sin(x)|), x > 0 \\ \dfrac{(x^2 + 2x)}{cos(x)}, x < 0 \end{cases}$$

3. Viết chương trình giải hệ phương trình ba ẩn đầy đủ bằng công thức định thức Cramer.

Cấu trúc lặp

Trong thực tế, chúng ta gặp nhiều tình huống lặp đi lặp lại. Với mục đích này, ngôn ngữ C++ cung cấp cho chúng ta các cấu trúc lặp tương ứng.

1. Vòng lặp while

Cú pháp:

```
while (biểu_thức_điều_kiện_đúng)
{
  ....
}
```

Giải thích: Nếu *biểu_thức_điều_kiện* đúng, các lệnh bên trong vòng lặp sẽ được thực hiện cho đến khi nó nhận giá trị sai.

Ví dụ	**Kết quả**
#include<iostream> using namespace std;	Nhap n:5 5

```
int main()
{
  int n;
  cout<<"Nhap n:";
  cin>>n;
  while (n>0){
      cout<<n<<endl;
      n--;
  }
  return 0;
}
```

```
4
3
2
1
```

Giải thích: đầu tiên nhập vào giá trị cho biến n. N nhập vào ở đây là 5. Vòng lặp while kiểm tra điều kiện n>0. Điều kiện này đúng, nên các lệnh trong vòng lặp sẽ được thực hiện. Nó sẽ in ra giá trị của n là 5. Sau đó, giá trị của n giảm đi 1, tức là n = 4. Vòng lặp lại tiếp tục thực hiện, vì n>0 còn đúng. Quá trình này cứ tiếp tục, cho đến khi n=0. Khi đó, điều kiện n>0 là sai. Do đó, vòng lặp sẽ dừng lại. Giá trị in ra màn hình là các số từ 5 giảm đến 1.

Lưu ý: khi sử dụng vòng lặp while cần lưu ý các điểm sau đây:

- Vòng lặp phải có tính dừng. Nghĩa là *biểu_thức_điều_kiện* phải có trường hợp sai. Trong một số tình huống, người ta vẫn sử dụng vòng lặp vô hạn, nhưng cần có cơ chế để thoát khỏi vòng lặp khi cần thiết.
- Nếu có nhiều lệnh chịu sự chi phối của while, thì chúng cần được đặt trong dấu khối lệnh.

2. Vòng lặp do...while

Cú pháp:

```
do
{
  ....
}while  (biểu_thức_điều_kiện_đúng);
```

Giải thích: Thực hiện các lệnh trong vòng lặp, sau đó kiểm tra biểu_thức_điều_kiện. Nếu biểu_thức_điều_kiện còn đúng, thì tiếp tục lặp.

Ví dụ	**Kết quả**

#include<iostream>
using namespace std;
int main(){
  int n;
  cout<<"Nhap n:";
  cin>>n;
  do {
  cout<<n<<endl;
  n--;
  }while(n>0);
  return 0;
}
	Nhap n:5
	5
	4
	3
	2
	1
	0

Giải thích: đầu tiên nhập vào giá trị cho biến n. Giá trị n nhập vào ở đây là 5. Vòng lặp do…while sẽ thực thi các lệnh bên trong nó. Nó sẽ in ra giá trị của n là 5. Sau đó, giá trị của n giảm đi 1, tức là n = 4. Vòng lặp kiểm tra giá trị của biểu thức n>0. Vì biểu thức này đúng, nên nó tiếp tục lặp. Quá trình này cứ tiếp tục cho đến khi n=0. Giá trị n=0 vẫn được in ra, sau khi kiểm tra n>0 không còn đúng nữa, vòng lặp kết thúc. Khác với vòng lặp while ở trên, nó sẽ in ra giá trị từ 5 giảm đến 0.

Lưu ý: khi sử dụng vòng lặp do…while cần lưu ý các điểm sau đây:

- Vòng lặp phải có tính dừng. Nghĩa là *biểu_thức_điều_kiện* phải có trường hợp sai. Trong một số tình huống, người ta vẫn sử dụng vòng lặp vô hạn, nhưng cần có cơ chế để thoát khỏi vòng lặp khi cần thiết.
- Nếu có nhiều lệnh chịu sự chi phối của do…while, thì chúng cần được đặt trong dấu khối lệnh.

- Vòng lặp do…while luôn thực hiện các lệnh bên trong nó, ít nhất 1 lần.

3. Vòng lặp for

Cú pháp:

```
for (biểu_thức_khởi_tạo; biểu_thức_giới_hạn; biểu_thức_tăng_giảm)
{
....
}
```

Giải thích: Thực hiện vòng lặp, với số vòng lặp từ *biểu_thức_khởi_tạo* cho đến *biểu_thức_giới_hạn* theo mức tăng là *biểu_thức_tăng_giảm*.

Ví dụ	**Kết quả**

#include<iostream>
using namespace std;
int main()
{
int n, i;
cout<<"Nhap n:";
cin>>n;
for (i=0; i<=n;i++) {
cout<<i<<endl;
};
return 0;
}
	Nhap n:5
	0
	1
	2
	3
	4
	5

Giải thích: đầu tiên nhập vào giá trị cho biến n. Giá trị n nhập vào ở đây là 5. Vòng lặp for sẽ thực thi các lệnh bên trong với số vòng lặp là từ 0 đến 5, theo bước nhảy là 1 – tương ứng với i++.

Lưu ý: khi sử dụng vòng lặp for cần lưu ý các điểm sau đây:

- Các tham số trong vòng lặp for có thể khuyết một hoặc vài (thậm chí là tất cả) tham số. Tuy nhiên, dấu chấm phẩy là luôn bắt buộc. Số bước lặp của vòng lặp for sẽ được tính như sau:

$$\frac{biểu\_thức\_giới\_hạn \ - biểu\_thức\_khởi\_tạo}{biểu\_thức\_tăng\_giảm}$$

- Nếu có nhiều lệnh chịu sự chi phối của for, thì chúng cần được đặt trong dấu khối lệnh.
- Ta có thể thực hiện việc khai báo biến trực tiếp bên trong dấu ngoặc đơn của vòng lặp for.

Ví dụ	**Kết quả**

#include<iostream>
using namespace std;
int main()
{
  int n;
  cout<<"Nhap n:";
  cin>>n;
  for (**int i**=0; i<=n;i++) {
  cout<<i<<endl;
  };
  return 0;
}
	Nhap n:5
	0
	1
	2
	3
	4
	5

4. Các câu lệnh nhảy

a. Câu lệnh break

Lệnh break dùng để thoát ra khỏi vòng lặp. Thông thường, khi ta sử dụng các vòng lặp không xác định được số lần lặp, để tránh lặp vô hạn, người ta thường sử dụng câu lệnh break để thoát khỏi vòng lặp.

Ví dụ

```
#include<iostream>
using namespace std;
int main()
{
  int n = 2;
  for(;;){
    cout<<n<<endl;
    n--;
    if (n<0) break;
  }
  return 0;
}
```

Kết quả

```
2
1
0
```

Giải thích: Giá trị khởi tạo của n = 2. Vòng lặp for sẽ tiến hành in giá trị của n, sau đó, giảm giá trị của n đi 1. Câu lệnh điều kiện bên trong for sẽ kiểm tra điều kiện của n. Nếu n<0, thì lệnh break sẽ được thực hiện và vòng lặp bị hủy.

b. Câu lệnh continue

Câu lệnh continue thường được dùng trong các vòng lặp. Khi lệnh continue được gọi, *bước lặp hiện tại* sẽ được bỏ qua, và tiến hành *bước lặp tiếp theo*.

Ví dụ

```
#include<iostream>
using namespace std;
int main()
{
  for(int i=0;i<10;i++){
    if (i%2!=0)
      continue;
    cout<<i<<endl;
  }
  return 0;
}
```

Kết quả

```
0
2
4
6
8
```

Giải thích: Vòng lặp for sẽ thực thi các lệnh bên trong nó. Biến i chạy từ 0 đến 9, kiểm tra điều kiện i có phải là số chẵn hay không. Nếu i là số lẻ, thì câu lệnh continue sẽ được thực hiện và bước lặp hiện tại sẽ bị bỏ qua. Nếu i là số chẵn, thì lệnh continue sẽ không được gọi và bước lặp hiện tại vẫn được thực hiện. Do đó, lệnh cout chỉ được thực hiện trong trường hợp biến i là chẵn. Như vậy, mỗi khi giá trị i là chẵn, nó sẽ in kết quả. Nếu giá trị của i là lẻ, thì kết quả sẽ không được in ra.

c. Lệnh goto

Lệnh goto cho phép tạo ra một bước nhảy đến một nhãn được ấn định sẵn. Tên nhãn sẽ được đặt như sau **tên_nhãn:** và lệnh goto sẽ nhảy đến tên nhãn. Một lời khuyên cho chúng ta là nên hạn chế tối đa việc sử dụng lệnh goto. Bởi vì lệnh goto thường làm phá vỡ cấu trúc của lập trình hiện đại. Nhiều ngôn ngữ lập trình họ nhà C ra đời sau C++ như Java đã tuyệt giao hoàn toàn với câu lệnh goto này.

Chương trình

```
#include <iostream>
using namespace std;
int main()
{
  int n = 5;
  loop://Tên nhãn
  cout<<n<<endl;
  n--;
  if(n>0) goto loop;
  return 0;
}
```

Kết quả

```
5
4
3
2
1
```

Giải thích: Giá trị khởi tạo của biến n là 5. Nhãn được đặt tên là loop. Nhãn có thể hiểu như một vị trí được đánh dấu (bookmark). Chương trình tiến hành in giá trị của biến n. Sau đó, giảm giá trị của n đi. Câu lệnh điều kiện, kiểm tra giá trị của biểu thức n>0, nếu đúng thi lệnh goto được gọi và nó sẽ được chuyển đến vị trí đã được đánh dấu là **loop**. Chương trình lại thực thi thêm lần nữa kể từ vị trí **loop** đó. Nếu n<=0, lệnh goto không được gọi. Chương trình kết thúc. Việc sử dụng các câu lệnh lặp, hoàn toàn có thể thay

thế cho lệnh goto. Hãy luôn ghi nhớ: CHỈ NÊN sử dụng goto khi thực sự cần thiết.

d. Lệnh exit

Lệnh exit dùng để thoát khỏi chương trình và trả về một mã được chỉ định. Mã chỉ định này tùy thuộc vào hệ điều hành, nó có thể sử dụng trong chương trình theo quy ước như sau: nếu chương trình kết thúc bình thường, thì mã chương trình là 0; nếu có một sự cố không mong muốn xảy ra, thì mã chương trình là một giá trị khác 0.

```
void exit(int mã_chỉ_định);
```

Nếu tham số *mã_chỉ_định* không được cấp vào, tức là exit (không có dấu ngoặc đơn), thì nó sẽ tiến hành theo mặc định – tức giá trị 0. Hàm exit nằm trong thư viện stdlib.h. Đây là một hàm tương đối cũ nằm trong thư viện .h. Ta chỉ có thể sử dụng lệnh exit nếu khai báo thư viện stdlib.h mà không có thư viện tương ứng là stdlib.

Cấu trúc lựa chọn: switch

Cú pháp:

```
switch(biểu_thức){
  case hằng_1:
    nhóm_các_lệnh;
    break;
  case hằng_2:
    nhóm_các_lệnh;
    break;
  ...
  default:
    nhóm_các_lệnh;
}
```

Giải thích: kiểm tra giá trị của biểu thức, nếu giá trị của biểu thức rơi vào danh sách hằng, thì nó sẽ thực hiện các lệnh trong từng trường hợp case tương ứng (nếu là *hằng_1* – các lệnh trong trường hợp *case hằng_1*, ….). Nếu biểu thức không thuộc vào danh sách hằng, thì nó sẽ thực hiện lệnh trong trường hợp default.

Chương trình

```
#include <iostream>
using namespace std;
int main()
{
  char n;
  cout<<"Ban la nam hay nu b/g:"<<endl;
  cin>>n;
  switch(n)
  {
    case 'b':
      cout<<"Nam";
      break;
    case 'g':
      cout<<"Nu";
      break;
    default:
      cout<<"Khong xac dinh";
  }
  return 0;
}
```

Kết quả

```
Ban la nam hay nu b/g:b
Nam
```

Giải thích: chương trình buộc người dùng nhập vào một kí tự (b – boy) hay (g – girl). Nếu người dùng nhập vào một kí tự khác, chương trình vẫn tính đến trường hợp này. Kí tự mà người dùng nhập vào được lưu trong biến n. Câu lệnh switch sẽ kiểm tra biến nhập vào đó có nằm trong danh sách hằng hay không (danh sách hằng ở đây là 'b' và 'g'). Nếu có, thì tương ứng với 'b' nó sẽ thực hiện trường hợp case 'b', nếu là 'g' nó sẽ thực hiện trường hợp case 'g'. Lệnh break trong mỗi trường hợp có tác dụng là thoát ra khỏi câu lệnh lựa chọn (cũng mang tính lặp), nếu không có lệnh break, tất cả các trường hợp đều được xét duyệt và nếu rơi vào trường hợp nào thì các lệnh tương ứng sẽ được thực thi, đồng thời lệnh ở trường hợp bên dưới nó cũng

được thực thi. Trong trường hợp, kí tự nhập vào không tương ứng trong danh sách hằng, nó sẽ thực thi trường hợp default. Vì default là trường hợp cuối cùng, nên nó không cần lệnh break.

Chú ý:

Lệnh switch chỉ được lựa chọn để sử dụng khi cần kiểm tra giá trị của một biểu thức có tương ứng với một tập các hằng số nào đó hay không (sự tương ứng ở đây có thể là thuộc hoặc không thuộc tương ứng với khái niệm trong tập hợp). Các hằng_1, hằng_2,... có thể là một vùng liên tục, hoặc gián đoạn (như các số từ 0..1, 'a'..'d',...). Nhưng nhất thiết các giá trị tương ứng với các trường hợp case phải là hằng số (hoặc khoảng hằng).

int a = 1;
switch(a>0)
{
case true:
cout<<"Duong";
break;
case false:
cout<<"Am";
break;
default:
cout<<"Khong";
}
	Liên tục

int a = 1;
switch(a)
{
case 1: case 2: case 3:
cout<<"Xuan";
break;
case 4: case 5: case 6:
cout<<"Ha";
break;
case 7: case 8: case 9:
cout<<"Thu";
break;
case 10: case 11: case 12:
cout<<"Dong";
break;
default:
cout<<"Khong phai thang cua nam";
	Rời rạc

}	

Biểu thức trong lệnh switch nhất thiết không phải là một kiểu có cấu trúc (mảng, xâu,…). Ví dụ sau đây sẽ phát sinh lỗi khi biên dịch, do biểu thức tương ứng với một xâu.

string s = “abc”;
switch(s)
{
  case “a”:
  case “ab”:
  case “abc”:
  default:
}
	Error

Trong hầu hết các ngôn ngữ lập trình, đại đa số đều không cho phép tham số trong switch là một xâu (cũng như lệnh case of trong họ Pascal – Delphi). Tuy nhiên, ngôn ngữ C# vẫn hỗ trợ xâu kí tự trong tham số của switch dù nó là một dẫn xuất của C++.

Một điều cần lưu ý, về bản chất thì câu lệnh switch sẽ tương ứng với một dãy các câu lệnh if. Chúng hoàn toàn có thể thay thế cho nhau. Cũng tương tự, các câu lệnh lặp while, do..while và for cũng có thể thay thế cho nhau một cách hoàn toàn. Có nghĩa là chúng ta chỉ cần nắm được cú pháp của một trong ba câu lệnh lặp này là có thể vận dụng trong mọi trường hợp. Tuy nhiên, chúng vẫn được sử dụng trong các trường hợp mang tính đặc trưng. Điều này rất hữu ích cho những người đã từng làm quen với ngôn ngữ lập trình họ Pascal – Delphi. Bảng sau đây tổng hợp các cách sử dụng của các lệnh có cấu trúc thường dùng.

Tên lệnh	**Cách dùng**
if…else	Khi cần kiểm tra một hoặc một vài điều kiện mang tính chất logic.
switch	Khi cần kiểm tra điều kiện hoặc tính thuộc vào của một biến số/biểu thức trong một danh sách hằng tương ứng.
for	Lặp có số vòng lặp xác định
while	Cần kiểm tra điều kiện lặp trước khi thực hiện lệnh, lặp không xác định số vòng lặp.
do…while	Kiểm tra điều kiện lặp sau khi thực hiện lệnh, lặp không xác định số vòng lặp.
break	Cần thoát khỏi vòng lặp.

continue	Bỏ qua vòng lặp hiện tại, thực thi bước lặp tiếp theo.
goto	Nhảy đến một nhãn được chỉ đinh. Nên tránh sử dụng, chỉ sử dụng trong những trường hợp thực sự cần thiết.

Bài tập 6.

1. Sử dụng các cấu trúc lặp for, while, do...while, goto để xây dựng các chương trình tính tích phân sau (với mỗi cấu trúc xây dựng mỗi chương trình) bằng phương pháp hình chữ nhật (trái, phải hoặc trung tọa).

$$\int_0^1 \sin(x)\,dx$$

2. Lựa chọn cấu trúc lệnh phù hợp, để tính giá trị của chuỗi hữu hạn sau đây

$$\sum_{n=0}^{100} \frac{n}{n^2+1}$$

3. Tính các tổng sau đây

$$S = \frac{1}{2} + \frac{2}{3} + \cdots + \frac{n}{n+1}$$

$$S = \frac{1!}{2} + \frac{2!}{3} + \cdots + \frac{n!}{n}$$

$$S = -\frac{1}{x+1} + \frac{2}{x^2+2!} + \cdots + \frac{(-1)^n}{x^n+n!}$$

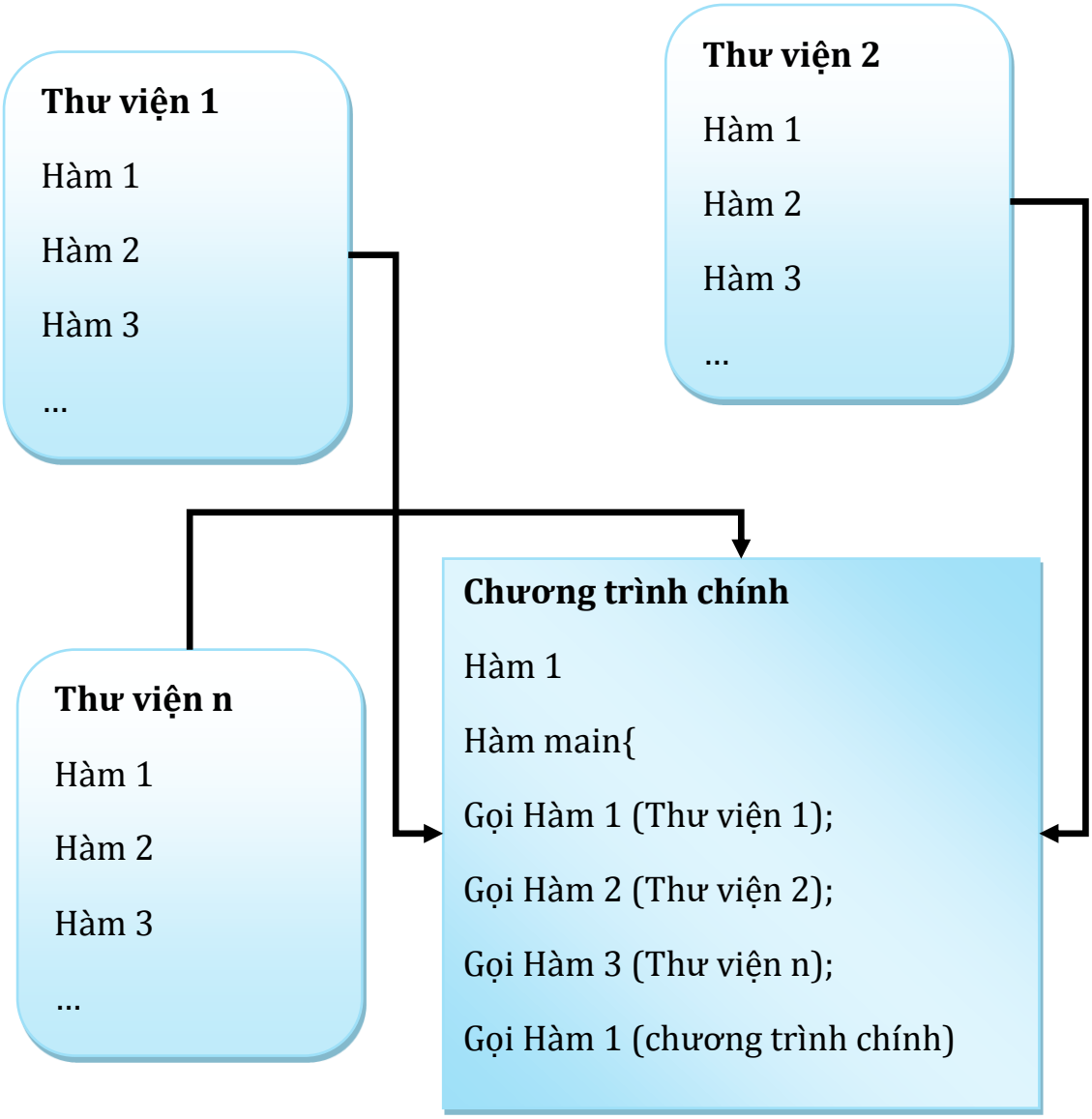
$$S = \frac{1!}{x+1} + \frac{3!}{x+3} + \cdots + \frac{(2n-1)!}{x+2n-1}$$

$$S = 2^{2!} + 4^{4!} + \cdots + (2n)^{2n!}$$

với n, x nhập vào từ bàn phím. Yêu cầu xây dựng hàm.

CHƯƠNG 7. HÀM

Hàm là một tập hợp các câu lệnh, nó được thực thi khi được gọi từ một vị trí khác trong chương trình. Hãy quan sát lược đồ bên dưới đây:



Hình 16 – Sơ đồ minh họa việc sử dụng hàm

Trong chương trình chính, chúng ta có thể gọi các hàm trong các thư viện, lẫn các hàm được khai báo trong chương trình chính.

Nhờ vào việc sử dụng hàm, chúng ta có thể phân chia chương trình thành các modul nhỏ. Đôi lúc, người ta gọi cách phân chia chương trình

thành các hàm như thế này là cách giải quyết bài toán theo phương pháp "chia để trị". Cách phân chia này có rất nhiều ưu điểm:

- *Làm cho chương trình trở nên gọn gàng dễ đọc hơn.*
- *Dễ cải biên chương trình.*
- *Dễ kiểm tra theo các modul.*

Đó là ý tưởng để xây dựng hàm. Vậy hàm được khai báo và sử dụng như thế nào. Chúng ta sẽ tìm hiểu trong chương này.

Khai báo và sử dụng hàm

Cú pháp:

```
kiểu_dữ_liệu tên_hàm(danh_sách_tham_số)
{
  Thân hàm;
}
```

Trong đó,

- **kiểu_dữ_liệu**: là kiểu dữ liệu mà hàm trả về.
- **tên_hàm**: là tên của hàm, do người lập trình đặt. Tên hàm không được chứa kí tự đặc biệt, không được bắt đầu bằng số, không chứa kí tự trắng, không trùng với từ khóa.
- **danh_sách_tham_số**: là danh sách các tham số dùng như các biến cục bộ. Nếu có nhiều tham số, thì chúng sẽ được phân tách theo các dấu phẩy.
- **Thân hàm**: là nội dung mà người lập trình xây dựng nên. Nếu hàm trả về kiểu dữ liệu khác void, ta cần sử dụng lệnh return để trả về biến chứa kết quả và có cùng kiểu dữ liệu với kiểu dữ liệu của hàm. Nếu hàm trả về kiểu dữ liệu void thì điều này là không cần thiết.

Ví dụ	**Kết quả**
#include <iostream> using namespace std; int add(int a, int b) {	3

```
  return a+b;
}
int main()
{
  cout<<add(1, 2);
  return 0;
}
```

Giải thích: Mọi chương trình trong C++ luôn bắt đầu bởi hàm main. Điều đó có nghĩa là các lệnh trong hàm main sẽ được thực thi một cách tuần tự. Đối tượng cout sẽ in giá trị của hàm add(1,2). Khi gọi đến hàm add, nó sẽ ánh xạ đến hàm add đã được khai báo ở trên, nó chỉ tạo ra một lời gọi hàm đến hàm đã được xây dựng có tên tương ứng (chúng ta cần lưu ý đến điều này để phân biệt với khái niệm hàm nội tuyến sẽ được tìm hiểu trong các mục tiếp theo). Với a, b là các tham số hình thức, chúng sẽ được thay thế bằng các giá trị cụ thể là 1 và 2. Hàm add này sẽ trả về giá trị là tổng của a và b nhờ từ khóa return.

Cách khai báo hàm trong ví dụ trên được gọi là khai báo trực tiếp. Chúng ta cũng còn một cách khai báo hàm gián tiếp nữa, mà ta thường gọi là khai báo hàm prototype như sau:

Ví dụ

```
#include <iostream>
using namespace std;
int add(int a, int b);
int main()
{
  cout<<add(1, 2);
  return 0;
```

Kết quả

```
3
```

```
}

int add(int a, int b)

{

  return a+b;

}
```

Trong khai báo hàm dạng này, cấu trúc khai báo hàm khuyết phần thân hàm. Chỉ có khai báo phần tên của hàm theo cú pháp chuẩn. Ta có thể đặt hàm xây dựng hoàn chỉnh ở bất kì vị trí nào. Cách khai báo hàm prototype có nhiều ưu điểm:

- Không cần quan tâm đến thứ tự khai báo hàm. Nếu không sử dụng khai báo prototype, thì hàm khai báo sau mới được phép gọi hàm khai báo trước nó. Điều ngược lại là không được phép. Nhưng đối với khai báo prototype thì ta hoàn toàn không cần quan tâm đến điều này.

- Ta có thể tách phần khai báo prototype và đặt nó vào trong một tập tin mới, thường là tập tin tiêu đề .h (với tên gọi tùy vào người lập trình quy định), phần thân hàm lại chứa trong một tệp khác, thường là .cpp hoặc trong chính tệp chứa chương trình chính. Cách làm này giúp chương trình sáng sủa hơn rất nhiều. Trong các dự án lập trình lớn, người ta thường phân tách theo dạng này. Chúng ta sẽ xét ví dụ minh họa sau.

Trong ví dụ minh họa này, dự án của tôi gồm có hai tệp: tieude.h để chứa khai báo prototype và main.cpp để chứa thân hàm và hàm main.

Đối với Codeblocks, hãy thực hiện theo các bước sau:

- Tạo mới một dự án C++ và lưu lại. Trong dự án này, mặc định Codeblocks sẽ tạo một tệp main.cpp.

- Vào New > File > chọn C/C++ header. Sau đó, hãy chọn vị trí để lưu trữ tệp tiêu đề (thông thường, ta nên tạo các thư mục khác nhau để lưu tệp .h cũng như tệp .cpp như tôi đã trình bày ở trên).

Đối với Eclipse, thực hiện như sau:

- Kích chuột phải vào thư mục cần đặt tệp .h, chọn New > Header File.

- Kích chuột phải vào thư mục cần đặt tệp .cpp, chọn New > Source File.

Đối với Visual Studio 2010, kích chuột phải vào tên dự án, chọn Add New Item. Sau đó chọn header file .h.

Tệp tieude.h	**Tệp main.cpp**

#ifndef TIEUDE_H_INCLUDED
#define TIEUDE_H_INCLUDED
 	
int sum(int, int);
void showmsg(void);
 	
#endif // TIEUDE_H_INCLUDED
	#include <iostream>
	#include "tieude.h"
	using namespace std;
	int main()
	{
	  showmsg();
	  return 0;
	}
	void showmsg(){
	  cout<<sum(1, 3);
	}
	int sum(int a, int b){
	  return a+b;
	}

Trong tệp tieude.h, ta chỉ việc nhập các khai báo prototype vào giữa #define và #endif. Trong tệp main.cpp, ta cần bổ sung khai báo thư viện #include "tieude.h". Chú ý rằng, tên tệp tiêu đề nằm trong dấu nháy kép "", mà không phải là dấu <>. Các hàm trong chương trình chính có thể sử dụng mà không cần quan tâm đến thứ tự khai báo. Như chúng ta thấy, các hàm khai báo sau hàm main (điều này chỉ có thể được phép đối với khai báo prototype). Hàm showmsg khai báo trước hàm sum nhưng có thể gọi được hàm sum. Thứ tự tiêu đề của hàm trong tệp tiêu đề hoàn toàn không quan trọng và nó không ảnh hưởng việc sử dụng các hàm theo thứ tự trước sau.

Lưu ý:

- Khi sử dụng khai báo prototype trên các tập tin .h, ta cần lưu ý, nếu dự án có sử dụng namespace, ví dụ std, ta chỉ có thể sử dụng cú pháp truy cập std:: mà không được sử dụng using namespace std trong tệp .h này.
- Nếu tệp .cpp và tệp .h nằm trong cùng thư mục, thì phần #include trong tệp cpp có thể viết tên tệp tiêu đề trong dấu "". Nếu chúng không nằm trong cùng thư mục, ta cần chỉ đường dẫn tương đối cho nó. Ví dụ tệp headers.h nằm trong thư mục headers và tệp main.cpp nằm trong thư mục cpps. Nếu tệp headers.h là tệp tiêu đề của tệp main.cpp, ta cần

include nó trong main.cpp. Giả sử headers và cpps nằm trong cùng thư mục src. Khi đó, trong tệp main.cpp, hãy khai báo như sau: #include"../headers/headers.h". Trong đó, dấu ../ để dịch lùi một mức trong cấu trúc cây thư mục (dịch lùi từ thư mục headers một mức chính là thư mục src), sau đó là headers/headers.h.

Phạm vi tác dụng của biến

Như tôi đã giới thiệu ở trên, biến toàn cục là những biến được khai báo ngoài tất cả các hàm (hay không bao trong bất kì dấu {} nào). Các biến này có tác dụng trong toàn bộ chương trình. Ta có thể gọi nó trong hàm main, hay trong các hàm khác. Ngược lại, những biến còn lại gọi là các biến cục bộ. Những biến cục bộ được khai báo trong phạm vi nào (được xác định nhờ dấu {}) thì chỉ có tác dụng trong phạm vi đó mà thôi.

Ví dụ

```
#include <iostream>

using namespace std;

int global;

int add(int a, int b)

{

  local result = a + b;

  return result;

}

int main()

{

  int local1;

  if(local1>0)

  {

    int local2;
```

Giải thích

- Biến global là biến toàn cục, nó có tác dụng trong toàn bộ chương trình. Ta có thể sử dụng nó trong hàm main, hàm add…

- Biến local, local1, local2 là các biến cục bộ. Biến local được khai báo trong hàm add, nó có phạm vi tác dụng trong phạm vi của hàm này. Biến cục bộ local1 được khai báo trong hàm main. Nó cũng chỉ có tác dụng trong hàm main. Biến local2 được khai báo trong phạm vi tác dụng của câu lệnh if, nó chỉ có tác dụng trong khối lệnh này. Nếu ta gọi biến này ngoài khối lệnh của if, chương trình dịch sẽ báo lỗi.

```
  }
  return 0;
}
```

Hàm không trả về giá trị - Hàm void.

Như chúng ta đã thấy trong các ví dụ trên, các hàm mà chúng ta sử dụng là các hàm có giá trị trả về. Đối với các loại hàm này, để hàm trả về một giá trị nào đó, ta sử dụng từ khóa return. Giá trị hàm trả về phải có kiểu dữ liệu cùng loại với kiểu dữ liệu mà ta quy định khi khai báo hàm.

Chúng ta cũng bắt gặp tình huống: nhiều lúc hàm mà ta xây dựng không trả về một giá trị nào, hoặc trả về nhiều hơn một giá trị. Khi đó, chúng ta sử dụng khai báo hàm void. Đối với hàm không trả về giá trị, ta có thể tham khảo ví dụ sau. Còn đối với hàm trả về nhiều hơn một giá trị, chúng ta sẽ thảo luận kĩ hơn trong phần tham biến.

Ví dụ

```
#include<iostream>
using namespace std;
void showMsg()
{
  cout<<"Hello, world !";
}
int main()
{
  showMsg();
  return 0;
}
```

Kết quả

```
Hello, world !
```

Chú ý:

Vì hàm có kiểu dữ liệu trả về luôn trả về một giá trị cụ thể, nên chúng ta có thể sử dụng trực tiếp loại hàm này trong các biểu thức tính toán (ví dụ a=b+sin(x)). Điều này là không thể đối với hàm void.

Khi sử dụng các hàm không có tham số hình thức, nếu ta gọi hàm theo cách sau: **tên_hàm();** là cách gọi hàm đúng. Nếu gọi hàm theo cách: **tên_hàm;**, thì dù chương trình dịch không báo lỗi, nhưng kết quả nhiều khi không chính xác. Vì vậy, chúng ta nên gọi hàm theo cách đầu tiên.

Tham biến và tham trị

Cho đến thời điểm này, các hàm mà chúng ta đã nghiên cứu đều truyền tham số theo **tham trị**. Điều này có nghĩa là khi gọi hàm, các giá trị từ các đối số truyền vào trong hàm sẽ được sao chép sang các tham số này, các tham số đó chỉ đóng vai trò là các tham số hình thức, chúng không lưu lại giá trị cho các đối số truyền vào, mà các giá trị đó đã bị các lệnh trong hàm làm thay đổi.

Ví dụ

```
#include <iostream>

using namespace std;

void setNum(int a)

{

  a = 0;

}

int main()

{

  int b = 1;

  setNum(b);

  cout<<b;

  return 0;
```

Giải thích

Nếu tham số a trong hàm setNum được sử dụng như trên (đơn thuần là int a) thì nó được quy định là truyền theo tham trị.

Khi truyền theo tham trị, giá trị của biến xuất hiện trong lời gọi hàm này, sẽ không thay đổi sau khi thoát ra khỏi hàm. Điều này có nghĩa là giá trị của biến b trước khi gọi hàm là 1, sau khi gọi hàm, nó vẫn nhận giá trị là 1.

```
}
```

Nếu muốn thay đổi giá trị của biến khi truyền tham số trong hàm, ta sử dụng khai báo **tham biến**. Với việc quy định các tham số truyền theo tham biến, thì khi khai báo ta chỉ bổ sung vào dấu **&** ở trước tên tham số đó. Bằng cách này, các biến là đối số trong lời gọi hàm sẽ bị làm thay đổi giá trị sau khi kết thúc lời gọi hàm.

Ví dụ

```
#include <iostream>

using namespace std;

void setNum(int &a)

{

  a = 0;

}

int main()

{

  int b = 1;

  setNum(b);

  cout<<b;

  return 0;

}
```

Giải thích

Nếu tham số a trong hàm setNum được sử dụng như trên (int &a) thì nó được quy định là truyền theo tham biến.

Khi truyền theo tham biến, giá trị của biến xuất hiện trong lời gọi hàm này, sẽ thay đổi sau khi thoát ra khỏi hàm. Điều này có nghĩa là giá trị của biến b trước khi gọi hàm là 1, sau khi gọi hàm, nó vẫn nhận giá trị là 0.

Việc sử dụng khai báo hàm theo tham biến có ưu điểm: ta biết rằng, một hàm chỉ có thể trả về một giá trị duy nhất. Nếu trường hợp, ta mong muốn hàm trả về nhiều giá trị, ta có thể sử dụng hàm void kết hợp với việc truyền tham số cho hàm theo tham biến. Ví dụ sau đây sẽ cho ta thấy rõ điều này.

Ví dụ **Kết quả**

```
#include <iostream>
using namespace std;
void swap(int &a, int &b)
{
  int c = a;
  a = b;
  b = c;
}
int main()
{
  int m = 1;
  int n = 2;
  swap(m, n);
  cout<<"m="<<m<<", n="<<n;
  return 0;
}
```

m= 2, n=1

Giải thích: trong khai báo hàm swap, tham số a và b được quy định truyền theo tham biến. Hàm này sẽ thực hiện việc hoán đổi giá trị của hai tham số hình thức a và b. Trong hàm main, hai biến m, n có giá trị tương ứng là 1 và 2. Khi gọi hàm swap, thì hai biến này sẽ bị hoán đổi giá trị cho nhau. Do đó, kết quả in ra là m=2, n=1 (vì tham số được truyền theo tham biến). Như vậy, hàm trong trường hợp này có thể xem như trả về hơn một giá trị mong muốn.

Lưu ý: Cách truyền tham biến như trên chỉ áp dụng cho C++, trong C ta chỉ có thể truyền tham biến nhờ con trỏ. Cách này, vẫn còn hoạt động tốt trên C++.

Ví dụ	**Kết quả**

#include <iostream>
using namespace std;
void swap(int *a, int *b)
{
  int *c;//hoặc đơn thuần chỉ là c
  *c = *a;
  *a = *b;
  *b = *c;
}
int main()
{
  int m = 1;
  int n = 2;
  swap(&m, &n);
  cout<<"m="<<m<<", n="<<n;
  return 0;
}
	m= 2, n=1

Giá trị mặc định của tham số hình thức

Khi khai báo các tham số hình thức bên trong hàm. Nếu các tham số đó được gán giá trị mặc định, thì khi gọi hàm, chúng ta sẽ có một vài cách gọi tương ứng với số lượng khác nhau của các tham số.

Chương trình	**Kết quả**
#include<iostream>	1

using namespace std;
int add(int a, int b=0, int c=0)
{
  return a+b+c;
}
int main()
{
  cout<<add(1)<<endl;
  cout<<add(1,2)<<endl;
  cout<<add(1,2,3)<<endl;
  return 0;
}
	3
	6

Giải thích: Hàm add được khai báo với ba tham số hình thức. Tham số thứ nhất là không thể thiếu, vì nó không quy định giá trị mặc định. Với hai tham số b, c còn lại, ta có thể để khuyết. Trong trường hợp để khuyết, nó sẽ nhận giá trị mặc định mà ta đã gán cho nó (cụ thể ở đây là 0). Do đó, khi gọi hàm add(1), nó sẽ tương ứng với lời gọi hàm add(1,0,0), tức giá trị là tổng của 1+0+0 bằng 1. Tương tự, khi gọi hàm add(1,2) thì sẽ tương ứng với add(1,2,0) và cho kết quả là 3. Khi gọi hàm đầy đủ ba tham số add(1,2,3) sẽ cho kết quả là 6.

Chồng chất hàm

Trong C++, hai hàm khác nhau có thể có cùng tên, nhưng **danh sách tham số** của chúng phải khác nhau. Chúng được biết đến với tên gọi là chồng chất hàm. Khái niệm chồng chất hàm khác hoàn toàn với khái niệm quá tải hàm mà chúng ta sẽ tìm hiểu trong phần lập trình hướng đối tượng.

Ví dụ	**Kết quả**
#include<iostream>	3

```
#include<string>
using namespace std;
int add(int a, int b)
{
  return a+b;
}
string add(string a, string b)
{
  return a+b;
}
int main()
{
  cout<<add(1,2)<<endl;
  cout<<add("ab","cd")<<endl;
  return 0;
}
```

```
abcd
```

Giải thích: Hai hàm mà ta xây dựng có cùng tên là add. Hàm add đầu tiên có chức năng cộng hai số. Hàm add thứ hai có chức năng cộng hai xâu kí tự. Kiểu dữ liệu trả về và tham số hình thức của chúng cũng khác nhau.

Hàm nội tuyến

Chỉ định inline trước khai báo một hàm sẽ giúp trình biên dịch nhận biết hàm này được khai báo là nội tuyến. Không có một sự thay đổi nào đáng kể giữa hàm bình thường với hàm nội tuyến. Chỉ có một điểm khác biệt duy nhất đó là: với hàm nội tuyến, trình biên dịch sẽ khởi tạo một thân hàm và chèn nó vào vị trí được gọi tại mỗi thời điểm mà hàm đó được gọi, thay vì nó chỉ chèn lời gọi hàm. Việc làm này sẽ cải thiện đáng kể tốc độ biên dịch chương trình.

```
inline type tên_hàm(danh_sách_tham_số)
{
  Thân hàm;
}
```

Trong hầu hết các trình biên dịch hiện đại, việc quy định hàm là inline là không cần thiết. Nếu một hàm có thể sử dụng khai báo inline, thì chương trình dịch sẽ tự động tối ưu mã nguồn để có thể sử dụng nó. Ngược lại, nếu một hàm không thể sử dụng khai báo inline, thì chương trình dịch sẽ bỏ qua khai báo này. Chỉ định inline chỉ có tác dụng định hướng cho chương trình dịch.

Hàm đệ quy

Hàm đệ quy là những hàm gọi lại chính nó. Nó hữu dụng trong các tác vụ như sắp xếp (Quicksort) hoặc tính toán các biểu thức truy hồi, giải các bài toán bằng giải thuật cùng tên … Hàm đệ quy tương ứng với khái niệm quy nạp trong toán học. Ta có thể sử dụng định nghĩa giai thừa theo quy nạp toán học như sau

$$\boldsymbol{n}! = \begin{cases} \mathbf{1}, & \boldsymbol{n} = \mathbf{0} \\ \boldsymbol{n}.(\boldsymbol{n}-\mathbf{1})!, & \boldsymbol{n} > 0 \end{cases}$$

Và chúng ta có thể xây dựng hàm đệ quy tương ứng với phép tính giai thừa này như sau

Ví dụ	**Kết quả**

#include <iostream>
using namespace std;
long Fac(long a)
{
  if(a>0)
  return a*Fac(a-1);
  else
  return 1;
}
	6

```
int main()
{
  long num = 3;
  cout<<Fac(num);
  return 0;
}
```

Giải thích: hàm Fac sẽ tính giá trị giai thừa của a theo công thức tính ở trên. Nếu a>0, thì hàm sẽ trả về giá trị là a*Fac(a-1), hàm Fac(a-1) lại gọi đến chính nó, và quá trình này cứ tiếp tục cho đến khi giá trị a = 0. Việc sử dụng lời gọi hàm Fac(0) sẽ cho kết quả tương ứng là 1.

Lưu ý: hàm đệ quy cũng như vòng lặp, nó có thể lặp vô hạn, nếu điều kiện dừng không được đảm bảo.

Ta xét thêm một ví dụ sử dụng đệ quy. Tính tổng:

$$S_n = 1 + 2 + \cdots + n$$

Với công thức này, ta có thể phân tích như sau

$$S_n = \underbrace{1 + 2 + \cdots + (n-1)}_{S_{n-1}} + n = S_{n-1} + n$$

Từ phân tích này, ta dễ dàng thấy công thức dưới dạng truy hồi:

$$S_n = \begin{cases} 1, nếu\ n = 1 \\ S_{n-1} + n, nếu\ n > 1 \end{cases}$$

Ví dụ

```
#include <iostream>
using namespace std;
long S(int n)
{
  if(n == 1)
```

Kết quả

```
6
```

```
    return 1;
  else
    return n+S(n-1);
}
int main()
{
  int  n = 3;
  cout<<S(n);
  return 0;
}
```

Bài tập 7.

1. Xây dựng hàm giải phương trình
$$\mathbf{x^2} + \boldsymbol{sin^2(x)} = \sqrt{\boldsymbol{x+1}}$$
2. Xây dựng hàm tính tổng của n số nguyên tố đầu tiên, với n là số nhập vào từ bàn phím.
3. Xây dựng hàm kiểm tra một số có phải là số chính phương hay không. Số chính phương là số nguyên có thể biểu diễn dưới dạng lũy thừa của một số nguyên.
4. Xây dựng hàm đệ quy để tính tổng của n số nguyên dương từ 0 đến n.
5. Xây dựng hàm đệ quy để tính xác định số hạng thứ n của dãy sau đây
$$\begin{cases} \boldsymbol{x_n = 4x_{n-1} - 3x_{n-2}} \\ \boldsymbol{x_0 = x_1 = 2} \end{cases}$$
Với, *n* nhập vào từ bàn phím.
6. Cho một dãy 5 số nguyên. Hãy xây dựng các hàm sau:
   a. Hàm nhập vào 5 số nguyên đó.
   b. Hàm sắp xếp 5 số nguyên đó theo thứ tự tăng dần.
   c. Hàm tính tổng của 5 số nguyên đó.
   d. Xuất các kết quả: các số nguyên sau khi sắp xếp và tổng của chúng.

CHƯƠNG 8. CÁC KIỂU DỮ LIỆU CÓ CẤU TRÚC

Mảng

Một mảng là một dãy các phần tử có cùng loại dữ liệu được sắp xếp liên tục trong bộ nhớ máy tính. Chúng có thể được truy cập theo chỉ số của nó.

Điều này có ưu điểm là chúng ta có thể khai báo 5 biến giá trị kiểu nguyên nhờ vào khai báo mảng, mà không cần phải khai báo riêng biệt.

	0	1	2	3	4
Mảng	←int→				

Khai báo mảng

kiểu_dữ_liệu tên_mảng[số_phần_tử];

Ví dụ

string humans[5]; //Khai báo mảng humans có 5 phần tử xâu kí tự
int numbers[10]; //Khai báo mảng numbers có 5 phần tử số nguyên

Các chỉ số của mảng được đánh thứ tự từ vị trí 0. Để truy cập đến phần tử mảng, chúng ta truy cập theo chỉ số như sau humans[0], humans[1],… tương ứng với các phần tử thứ nhất, thứa hai… trong mảng humans.

Khởi tạo mảng

Việc khởi tạo giá trị cho mảng có thể sử dụng cặp dấu {}.

```
string humans[5] = {“Lan”, “Nam”, “Binh”, “Hoa”, “Hieu”};
```

Khi truy cập vào mảng theo chỉ số, thì humans[0]=”Lan”, humans[1]=”Nam”,…

Một cách thức nữa để khởi tạo giá trị cho mảng, là ta sử dụng toán tử gán cho từng phần tử. Để vét toàn bộ mảng (để nhập dữ liệu cho mảng hoặc xuất dữ liệu từ mảng hoặc làm việc với các phần tử của mảng), ta có thể sử dụng vòng lặp for.

```
int numbers[10];
for (int i=0; i<10; i++)
  numbers[i]=i;
```

Mảng nhiều chiều

Mảng nhiều chiều có thể được xem như là mảng của mảng. Mảng hai chiều là mảng của mảng một chiều, mảng ba chiều là mảng của mảng hai chiều,…

```
//Khai báo ma trận
int matrix[4][4];//Ma trận có 16 phần tử (4 dòng, 4 cột)
```

Chúng ta có thể minh họa khai báo ma trận trực quan như sau:

	0	1	2	3	4
0					
1				matrix[1][3]	
2					
3					
4					

Tương tự, ta có thể tạo ra các khai báo mảng nhiều chiều khác. Việc truy cập vào mảng để xét duyệt các phần tử của nó, có thể được thực hiện với số câu lệnh lặp lồng nhau chính là số chiều của mảng: mảng hai chiều – hai vòng lặp lồng nhau, mảng ba chiều – ba vòng lặp lồng nhau,…

```
//In giá trị của toàn mảng
for(int i=0; i<4; i++){
  for(int j=0;j<4;j++)
    cout<<matrix[i][j]<<" ";
  cout<<endl;
```

```
}
```

Mảng giả nhiều chiều

Nếu ta khai báo mảng như sau

```
//Khai báo ma trận
int matrix[4*4];//Ma trận có 16 phần tử
```

thì mảng này gọi là mảng giả nhiều chiều (giả hai chiều). Số phần tử của mảng giả hai chiều này bằng số phần tử của mảng hai chiều. Thực chất của mảng giả nhiều chiều là mảng một chiều. Các thao tác xử lý với mảng giả nhiều chiều được thực thi như mảng một chiều.

Ta cần lưu ý rằng, việc chuyển đổi chỉ số qua lại giữa mảng nhiều chiều và mảng giả nhiều chiều là hoàn toàn có thể thực hiện được. Ví dụ sau đây sẽ in ra giá trị của các phần tử theo dạng ma trận bằng cách sử dụng khai báo mảng hai chiều và mảng giả hai chiều.

Mảng hai chiều	**Mảng giả hai chiều**

int matrix[4][4];
//Nhập mảng
for(int i=0; i<4; i++)
  for(int j=0;j<4;j++)
  matrix[i][j]=i+j;
//In giá trị mảng
for(int i=0; i<4; i++){
  for(int j=0;j<4;j++)
  cout<<matrix[i][j]<<" ";
  cout<<endl;
}
	int matrix[4*4];
	//Nhập mảng
	for(int i=0; i<4; i++)
	  for(int j=0;j<4;j++)
	  matrix[i*4+j]=i+j;
	//In giá trị mảng
	for(int i=0;i<4;i++){
	  for(int j=0; j<4;j++)
	  cout<<matrix[i*4+j]<<" ";
	  cout<<endl;
	}

Mảng là tham số hình thức: khi sử dụng mảng làm tham số hình thức, cần lưu ý các điểm sau đây:

- Trong trường hợp mảng một chiều, ta có thể không cần khai báo kích thước của mảng (**ví dụ type tên_hàm(int args[])**).
- Trong trường hợp mảng nhiều chiều, thì chỉ có số phần tử trong chiều thứ nhất là có thể không cần khai báo, còn các chiều còn lại, nhất thiết phải khai báo (ví dụ type tên_hàm(**int args[][10][10]**)).
- Mảng được truyền theo tham biến.

- Cách tốt nhất khi sử dụng mảng làm tham số hình thức là hãy sử dụng nó dưới dạng con trỏ. Chi tiết về con trỏ sẽ được trình bày trong chương sau.

Bài tập 8.

1. Xây dựng các hàm sau đây
   a. Nhập vào một mảng hai chiều các số nguyên.
   b. Thực hiện các phép cộng, nhân hai mảng này.
   c. Xác định phần tử lớn nhất, nhỏ nhất trong mỗi dòng.
   d. Xây dựng mới một ma trận từ ma trận cũ, trong đó các phần tử của ma trận cũ này có giá trị chẵn sẽ bằng chính nó cộng với phần tử lớn nhất của dòng đó. Các phần tử lẻ sẽ bằng chính nó trừ cho phần tử nhỏ nhất của dòng đó. Tương ứng với chỉ số đó, ta sẽ xây dựng nên ma trận mới từ ma trận cũ đã cho.
   Ví dụ:

1	2
3	2

   Phần tử lớn nhất của dòng 1: 2
   Phần tử lớn nhất của dòng 2: 3
   Phần tử nhỏ nhất của dòng 1: 1
   Phần tử nhỏ nhất của dòng 2: 2
   Phần tử 1x1 là lẻ, nên nó bằng 1-1=0, và đây chính là phần tử 1x1 trong ma trận mới. Tương tự cho các phần tử còn lại. Ma trận mới thu được là

0	4
1	5

2. Cho mảng một chiều, viết chương trình hoán đổi vòng các giá trị của mảng đó (phần tử đầu trở thành phần tử cuối, …). Ví dụ {1, 2, 3}, sau khi hoán đổi, ta thu được {2, 3, 1}
3. Cho ma trận, hãy sử dụng phương pháp khử Gauss để đưa ma trận này về dạng tam giác trên.

Xâu kí tự

Như tôi đã giới thiệu, thư viện chuẩn của C++ chứa một lớp string rất mạnh mẽ, mà nó có thể hữu dụng trong việc thực thi các tác vụ xử lý xâu. Tuy nhiên, bởi vì xâu là một mảng các kí tự, do đó, chúng ta có thể xử lý xâu như xử lý trên mảng.

Ví dụ, ta có một khai báo xâu như sau

```
char strings [20];
```

Xâu strings này chứa 20 kí tự.

Việc khởi tạo giá trị cho một xâu hoàn toàn tương tự như khởi tạo giá trị cho mảng. Tuy nhiên, chúng ta có thêm một cách khởi tạo thuận lợi hơn như sau

```
strings = “Chao ban”;
```

Khi phân bố vào trong bộ nhớ, xâu này sẽ được biểu diễn như mảng. Tuy nhiên, phần tử cuối cùng trong mảng kí tự này là phần tử kết thúc xâu, được kí hiệu là **\0**.

C	h	a	o		b	a	n	\0											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Việc khai báo xâu theo kiểu mảng ký tự hay theo kiểu string là hoàn toàn tương đương nhau. Vì vậy, chúng ta có thể tùy ý lựa chọn cách xử lý chúng. Ngoài ra, mảng cũng có thể được khai báo như con trỏ. Vì vậy, với xâu kí tự, chúng ta có ba cách khai báo: sử dụng mảng kí tự, sử dụng con trỏ và khai báo xâu string. Chi tiết về con trỏ, chúng ta sẽ học trong chương sau.

CHƯƠNG 9. CON TRỎ

Khi một biến được lưu vào trong các ô nhớ, thông thường ta không quan tâm đến cách bố trí theo vị trí vật lý của nó, chúng ta đơn thuần chỉ truy cập đến các biến theo định danh của nó. Bộ nhớ của máy tính được tổ chức theo các ô nhớ, mỗi một ô nhớ là kích thước nhỏ nhất mà máy tính có thể quản lý – còn gọi là bytes. Mỗi ô nhớ được đánh dấu theo một cách liên tục. Các ô nhớ trong cùng một khối ô nhớ được đánh dấu theo cùng một chỉ số như khối ô nhớ trước đó cộng 1. Theo cách thức tổ chức này, mỗi ô nhớ có một địa chỉ định danh duy nhất và tất cả các ô nhớ thuộc vào một mẫu liên tiếp. Ví dụ, nếu chúng ta tìm kiếm ô nhớ 1776, chúng ta biết rằng nó sẽ là ô nhớ nằm ngay vị trí giữa ô nhớ 1775 và 1777, hay chính xác hơn là ô nhớ sau 776 ô nhớ so với ô nhớ 1000 (hay trước ô nhớ 2776 là 1000 ô nhớ).

Toán tử tham chiếu &

Khi mô tả một biến, hệ điều hành sẽ cung cấp một số lượng ô nhớ cần thiết để lưu trữ giá trị của biến. Chúng ta không quyết định một cách trực tiếp vị trí chính xác để lưu trữ biến bên trong mảng các ô nhớ đó. May mắn thay, tác vụ này hoàn toàn tự động trong suốt quá trình Runtime của hệ điều hành. Tuy nhiên, trong một vài trường hợp, chúng ta có thể quan tâm đến địa chỉ mà các biến được lưu trữ để điều khiển chúng.

Địa chỉ mà các biến lưu bên trong bộ nhớ được gọi là sự tham chiếu đến biến đó. Sự tham chiếu đến biến có thể nhận được bằng cách bổ sung dấu **&** trước định danh của biến – nó được gọi là địa chỉ của biến đó.

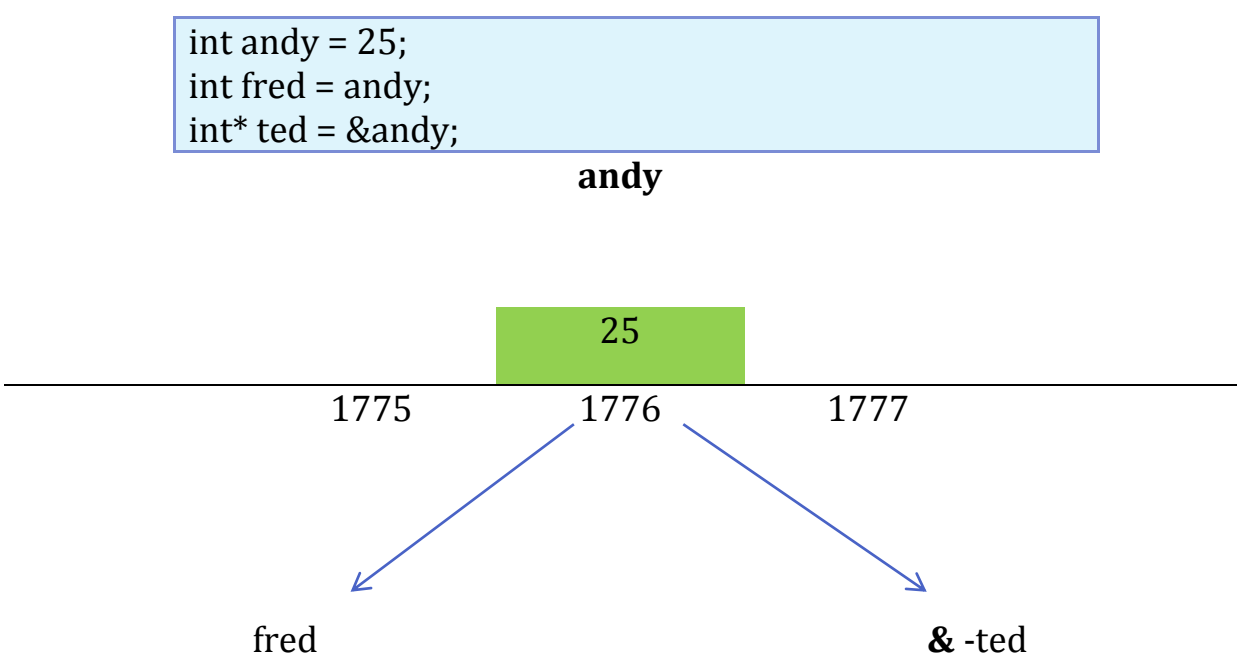
Ví dụ:

```
int a = 10;
int *adr = &a;
```

Khi khởi gán biến *adr cho địa chỉ của biến a, thì từ thời điểm này, việc truy cập tên biến a với tham chiếu & hoàn toàn không liên quan đến giá trị của nó, nhưng ta vẫn nhận được giá trị của biến a nhờ vào biến *adr.

Giả sử biến andy được lưu vào trong bộ nhớ tại ô nhớ 1776. Chúng ta có thể minh họa đoạn chương trình sau bằng lược đồ bên dưới

```
int andy = 25;
int fred = andy;
int* ted = &andy;
```



Hình 17 – Tham chiếu trong con trỏ

Đầu tiên giá trị 25 sẽ được gán cho biến andy, biến fred được khởi tạo từ biến andy (sao chép giá trị). Biến ted sẽ tham chiếu đến địa chỉ của biến andy, mà không sao chép giá trị của biến này vào ô nhớ của nó.

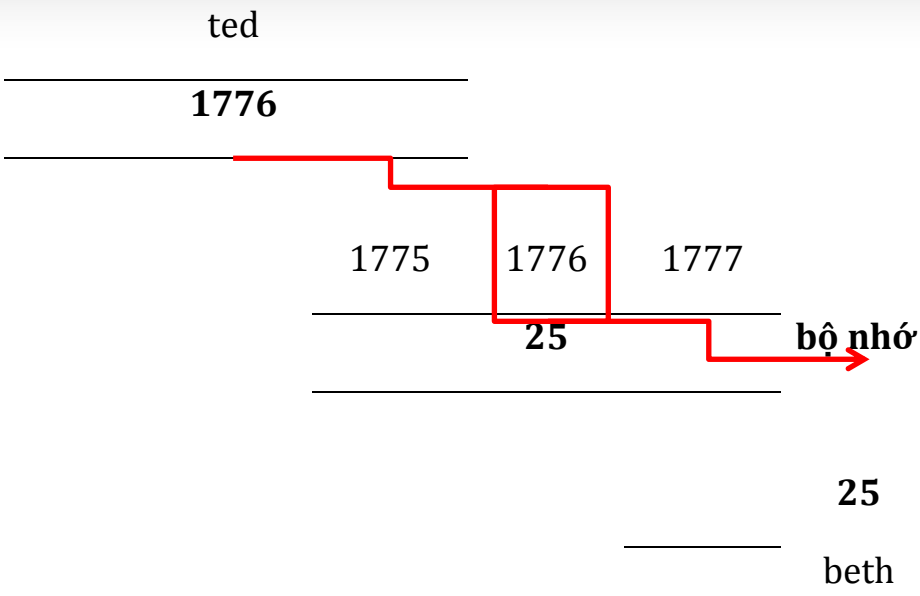
Toán tử tham chiếu ngược *

Một biến tham chiếu đến biến khác gọi là con trỏ. Con trỏ sẽ trỏ đến biến tham chiếu. Bằng việc sử dụng con trỏ, chúng ta có thể truy cập trực tiếp đến giá trị của biến được tham chiếu đến. Để thực thi được điều này, chúng ta đặt trước định danh của biến trỏ dấu *, khi đó, nó đóng vai trò là toán tử tham chiếu ngược và nó có thể gọi là "giá trị trỏ bởi".

Bởi vậy, chúng ta có thể viết như sau

```
beth = *ted;
```

Chúng ta có thể gọi: beth tương ứng với giá trị trỏ bởi ted. Để minh họa điều này, chúng ta có thể tham khảo lược đồ sau:



Hình 18 – Tham chiếu ngược trong con trỏ

Lược đồ này tương ứng với đoạn chương trình sau

```
beth = ted;//beth tương ứng với ted
beth = *ted;//beth tương ứng với giá trị trỏ bởi ted
```

Cần phân biệt chính xác giữa biến ted trỏ đến giá trị 1776, trong khi *ted trỏ đến giá trị lưu tại ô 1776, tức là 25. Như vậy, chúng ta cần phải phân biệt một cách chính xác hai toán tử: toán tử tham chiếu & và toán tử tham chiếu ngược*.

Toán tử tham chiếu &: đọc là **địa chỉ của**.

Toán tử tham chiếu ngược *: **đọc là giá trị trỏ bởi**.

Như vậy, một biến có thể tham chiếu nhờ toán tử & và có thể tham chiếu ngược bởi toán tử *.

Giả sử chúng ta có

```
int andy = 25;
int *ted = &andy;
```

Khi đó, các biểu thức sau đây sẽ cho giá trị đúng (giả sử địa chỉ của biến andy được lưu tại ô nhớ 1776)

```
andy == 25;
&andy == ted;//=1776
*ted == 25;
```

Ta có thể phát biểu tổng quát biểu thức *ted = &andy như sau: con trỏ *ted trỏ vào địa chỉ của andy, tương ứng với địa chỉ này của ô nhớ, ta có thể nhận được giá trị tương ứng là giá trị lưu tại ô nhớ này.

Khai báo biến con trỏ

Ta có thể lấy giá trị mà con trỏ trỏ đến một cách trực tiếp, nó cần thiết khi chúng ta muốn khai báo kiểu dữ liệu tương ứng với nó. Cú pháp khai báo con trỏ như sau:

```
type *tên_con_trỏ;
```

Ví dụ

```
int *pint;
char *pchar;
float *pfloat;
```

Trong ví dụ trên, chúng ta khai báo ba con trỏ có kiểu dữ liệu khác nhau, nhưng về bản chất, chúng – pint, pchar, pfloat là những con trỏ và chúng có cùng số ô nhớ trong không gian bộ nhớ (trên hệ thống windows 32bit, chúng chiếm 4byte – ta có thể sử dụng hàm **sizeof** để kiểm tra kích thước thực trên hệ thống máy tính đang sử dụng). Tuy nhiên, dữ liệu mà các con trỏ trỏ đến lại có kích thước khác nhau tương ứng với int, char và float mà chúng ta đã tìm hiểu (tương ứng trên hệ windows 32 bit lần lượt là 4, 1, 4).

Ví dụ	**Kết quả**

#include<iostream>
using namespace std;
int main()
{
  int *pint;
  long long *pll;
  cout<<sizeof(pint)<<endl;
  cout<<sizeof(pll)<<endl;
  cout<<sizeof(*pint)<<endl;
  cout<<sizeof(*pll)<<endl;
  return 0;
}
	4
	4
	4
	8

Giải thích: trong ví dụ này, biến pint và pll dùng để lưu địa chỉ của con trỏ, chúng luôn có kích thước mặc định là 4 bytes. Các biến *pint và *pll là các

biến trỏ vào các kiểu dữ liệu int và long long tương ứng. Biến int có kích thước 4 bytes và biến long long có kích thước 8 bytes.

Lưu ý, trong khai báo này, dấu * không phải là toán tử tham chiếu ngược, nó đơn thuần là con trỏ. Chúng có cùng kí hiệu, nhưng là hai thứ hoàn toàn khác nhau.

Ví dụ	**Kết quả**

#include<iostream>
using namespace std;
int main()
{
  int fval, sval;
  int *p;
  p = &fval;
  *p = 10;
  p = &sval;
  *p=20;
  cout<<fval<<endl;
  cout<<sval<<endl;
  return 0;
}
	10
	20

Giải thích: Bằng cách sử dụng biến con trỏ *p, chúng ta đã làm thay đổi giá trị của biến fval và sval. Biến trỏ này trong lần đầu tiên, nó trỏ đến địa chỉ của biến fval, từ ô nhớ của địa chỉ này, nó ánh xạ đến giá trị mà ta khởi gán là 10. Do đó, giá trị của biến fval cũng ánh xạ tương ứng đến 10. Tương tự cho biến sval.

Để minh họa con trỏ có thể tạo ra sự sai khác giá trị trong cùng một chương trình, chúng ta tham khảo ví dụ sau

Ví dụ	**Kết quả**

#include<iostream>
using namespace std;
int main()
{
  int fval=5, sval=15;
  int *p1, *p2;
  p1 = &fval;
  p2 = &sval;
  *p1 = 10;
	10
	20

```
  *p2 = *p1;
  p1 = p2;
  *p1 = 20;
  cout<<fval<<endl;
  cout<<sval<<endl;
  return 0;
}
```

Giải thích: Các biến *p1 và *p2 trỏ đến địa chỉ của fval và sval. Như vậy, *p1=10, sẽ tạo cho vùng địa chỉ mà nó trỏ đến, ánh xạ đến giá trị 10 (có nghĩa là tại thời điểm này fval = 10, sval = 15). Dòng lệnh *p2=*p1 sẽ làm cho biến trỏ *p2 trỏ đến giá trị mà *p1 trỏ đến (tức *p2 = 10). Và tại thời điểm này, biến sval có giá trị tương ứng là 10 (do ánh xạ theo vùng địa chỉ mà *p2 trỏ đến). Dòng lệnh p1=p2 sẽ gán địa chỉ mà p2 trỏ đến (địa chỉ của biến sval) cho địa chỉ mà p1 trỏ đến (địa chỉ của biến fval), như vậy, tại thời điểm này sval=fval=10. Dòng lệnh *p1=20 sẽ tạo cho vùng địa chỉ mà *p1 trỏ đến (cũng là địa chỉ của biến *p2 và fval) ánh xạ đến giá trị 20, nghĩa là fval = 20. Vì vậy, khi kết thúc chương trình, fval = 10, sval = 20. Cả hai biến *p1 và *p2 đều trỏ đến địa chỉ của biến fval.

Một con trỏ được khai báo dữ liệu của nó là một loại dữ liệu nào đó. Do đó, nếu có một biến không phải là con trỏ có cùng kiểu dữ liệu, chúng ta có thể sử dụng khai báo thu gọn

```
int *p1, *p2, num;
```

Con trỏ, mảng và xâu kí tự

Như tôi đã nói ở trên, mảng có thể được khai báo theo kiểu truyền thống hoặc theo cấu trúc con trỏ. Ví dụ, tôi có một mảng số nguyên, chúng có 20 phần tử

```
int numbers[20]; //Khai báo truyền thống
int *p = new int[20]; //Khai báo con trỏ
```

Hoặc ta có khai báo một biến xâu kí tự theo một trong 3 cách sau

```
char xau[20]; //Khai báo truyền thống
char *xau; //Khai báo con trỏ
string xau;//Khai báo xâu
```

Khởi tạo giá trị cho con trỏ khi nó đóng vai trò là mảng

Thứ nhất: ta sử dụng phép gán để sao chép toàn bộ mảng lên con trỏ

```
p = numbers;//p là con trỏ, numbers là mảng
```

Thứ hai: khởi tạo trực tiếp cho từng phần tử

```
int *p;
for (int i=0; i<20; i++){
  *(p+i)=numbers[i];
}
```

Chương trình nhập – xuất dữ liệu bằng việc sử dụng con trỏ.

Chương trình	Kết quả

#include<iostream>
using namespace std;
void Input(int *p, int length)
{
  for (int i=0; i<length; i++){
  cin>>*(p+i);
  }
}
void Output(int *p, int length)
{
  for (int i=0; i<length; i++){
  cout<<*(p+i)<<" ";
  }
}
int main()
{
  int *a;
  int lengtha;
  cout<<"Nhap do dai: ";
  cin>>lengtha;
  Input(a, lengtha);
  cout<<"========"<<endl;
  Output(a, lengtha);
  return 0;
}
	Nhap do dai: 4
	6
	8
	9
	0
	========
	6 8 9 0

Giải thích: biến con trỏ luôn được truyền theo kiểu tham biến, do đó, chúng ta không cần bổ sung dấu **&** vào bên trước tên biến. Hàm Input dùng để nhập mảng số nguyên có độ dài length, hàm Output dùng để xuất dữ liệu từ mảng nguyên đó. Khi truy xuất từ biến con trỏ, ngoài cách xuất như trên, ta

còn có thể viết p[i] tương ứng như mảng – p[i] và *(p+i) đều là phần tử thứ i. Nghĩa là ta hoàn toàn có thể viết

```
cin>>p[i];          cout<<p[i];
cin>>*(p+i);        cout<<*(p+i);
int c = p[i]+q[j];  int c = *(p+i)+*(q+j);
```

Việc sử dụng con trỏ để thay cho xâu cũng hoàn toàn tương tự. Nhưng hãy lưu ý, phần tử cuối cùng của nó là phần tử **\0**.

Các phép toán số học trên con trỏ

Các phép toán số học trên con trỏ tương đối khác với phép toán số học trên số nguyên. Chúng ta cũng tham khảo các phép toán tương ứng với nó.

Phép cộng và phép trừ

Giả sử, chúng ta có ba con trỏ khai báo như sau

```
char *pchar;
short *pshort;
int *pint;
```

và các biến trỏ này lần lượt trỏ vào các ô nhớ 1000, 2000 và 3000. Khi ta viết

```
pchar++;
pshort++;
pint++;
```

nó sẽ lần lượt nhảy sang địa chỉ của ô nhớ tiếp theo (ô nhớ dịch sang phải một đơn vị, tức là tương ứng với 1001, 2002 và 3004). Điều này cũng rất đơn giản. Kiểu dữ liệu char chiếm một byte, nên ô tiếp theo sẽ là địa chỉ của ô hiện tại +1. Kiểu dữ liệu short chiếm 2 byte, nên địa chỉ ô tiếp theo +2. Kiểu dữ liệu long chiếm 4 byte, nên địa chỉ của ô tiếp theo là +4.

Chương trình

```
int *p;

for(int i=0; i<4; i++){

  *(p+i)=i;

  cout<<*(p+i)<<"  "<<(p+i)<<endl;
```

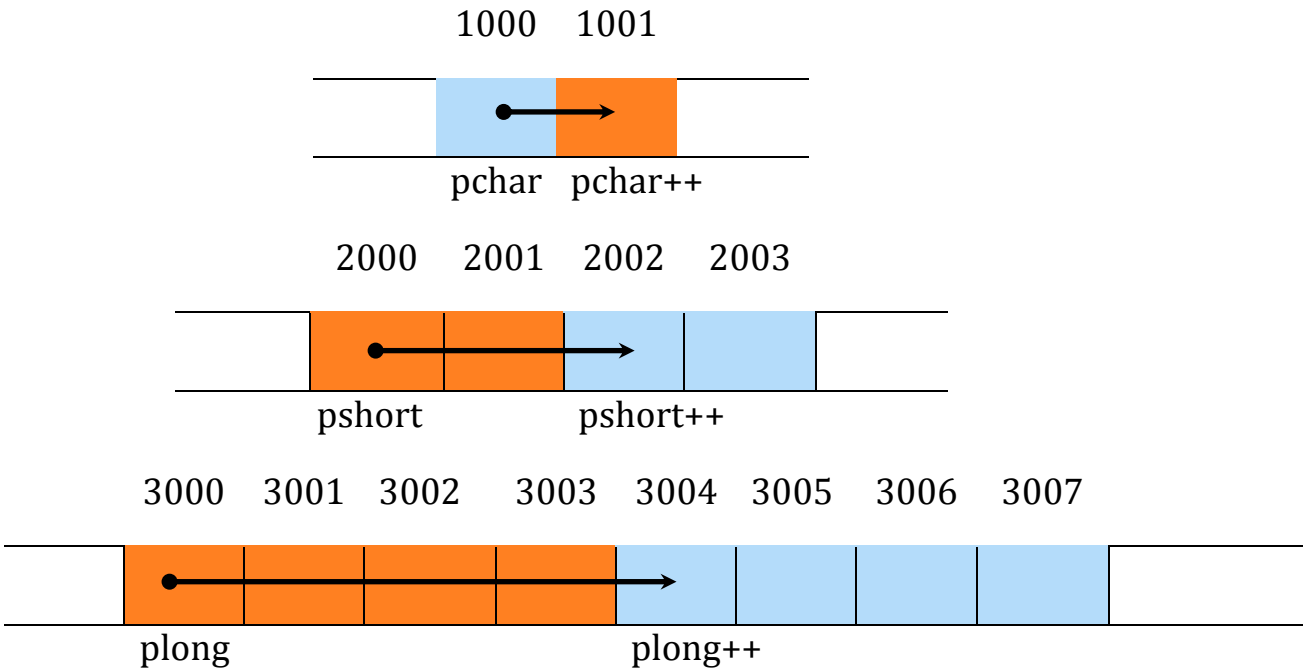
Kết quả

```
0   0x7ffd5000
1   0x7ffd5004
2   0x7ffd5008
3   0x7ffd500c
```

}

Ta có thể thấy giá trị và vùng địa chỉ của biến trỏ nguyên như trong kết quả trên. Dễ nhận thấy rằng địa chỉ tăng theo 4 byte (0x7ffd5000 tiếp theo 0x7ffd5004,…)

Để hiểu rõ hơn, ta có thể quan sát lược đồ sau đây



Hình 19 – Tăng/Giảm địa chỉ của con trỏ

Kết quả này tương đương với pchar+1, pshort+1 và plong+1.

Hoàn toàn tương tự với toán tử ++p, p-- và --p.

Toán tử tham chiếu ngược và toán tử tăng-giảm

Cả hai toán tử tăng và giảm có độ ưu tiên cao hơn toán tử tham chiếu ngược. Nếu ta đặt

```
*p++;
```

Do toán tử ++ có độ ưu tiên cao hơn toán tử *, nên biểu thức này tương ứng với *(p++). Vì vậy, biểu thức này có nghĩa là lấy giá trị ánh xạ bởi ô nhớ tiếp theo.

Ta cần chú ý rằng, biểu thức này hoàn toàn khác với

```
(*p)++;
```

Trong biểu thức này, dấu () có độ ưu tiên cao nhất, nên biểu thức *p sẽ thực hiện trước, do đó, giá trị của biểu thức này là giá trị ánh xạ bởi ô nhớ hiện tại cộng thêm 1.

Biểu thức

```
*p++ = *q++;
```

Bởi vì toán tử ++ có độ ưu tiên cao hơn toán tử *, cả hai biến p và q đều tăng, nhưng bởi vì chúng đều là toán tử hậu tố, nên toán tử * sẽ được thực hiện trước, sau đó mới tăng địa chỉ của biến tương ứng. Như vậy, biểu thức trên tương ứng với

```
*p = *q;
++p;   // hay p=p+1;
++q;   // hay q=q+1
```

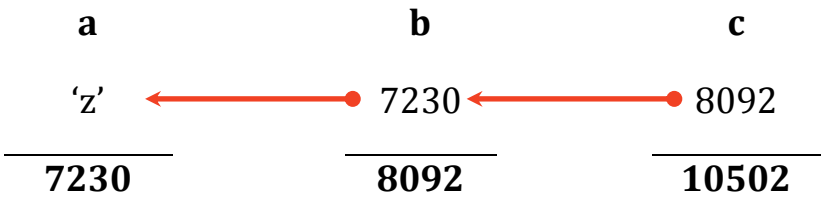
Trong trường hợp này, chúng ta nên sử dụng dấu () để có một cách nhìn nhận rõ ràng hơn.

Con trỏ trỏ vào con trỏ

C++ cho phép sử dụng con trỏ đa tầng, nghĩa là con trỏ trỏ vào con trỏ. Để khai báo con trỏ loại này, chúng ta chỉ cần bổ sung thêm vào biến trỏ một toán tử *.

```
char a;
char *b;
char **c;
a = 'z';
b = &a;
c = &b;
```

Để minh họa điều này, chúng ta có thể tạo một địa chỉ ngẫu nhiên trong bộ nhớ cho mỗi biến, ví dụ là 7230, 8092 và 10502.



Giá trị của mỗi biến được viết trong mỗi ô. Dưới mỗi ô là địa chỉ tương ứng của ô đó trong bộ nhớ. Địa chỉ của biến a là 7230, giá trị của ô nhớ tương ứng là 'z'. Biến *b trỏ vào địa chỉ của biến a và giá trị tương ứng

của biến b ánh xạ đến giá trị của ô nhớ biến a – tức giá trị là 'z'. Biến c trỏ vào địa chỉ của b, nó sẽ ánh xạ gián tiếp đến giá trị của a – tức giá trị 'z'.

Sự tương ứng giữa con trỏ trỏ vào con trỏ khác và mảng hai chiều

Việc khai báo biến **c như trên, có thể xem như là mảng hai chiều. Chúng ta lại xét bài toán ma trận.

Chương trình	Kết quả

#include <iostream>
using namespace std;
int main()
{
//Khai báo ma trận
int **matrix;
//Khởi tạo ma trận
matrix = new int*[3];//dòng
for(int i=0; i<3; i++)//phần tử
matrix[i] = new int[3];
//Nhập ma trận
for (int i=0; i<3; i++)
for (int j=0; j<3; j++)
{
((matrix+i)+j)=i+j;
}
//Xuất ma trận
for (int i=0; i<3; i++){
for (int j=0; j<3; j++)
cout<<*(*(matrix+i)+j)<<" ";
cout<<endl;
}
//Xóa ma trận
delete[] matrix;
return 0;
}
	0 1 2
	1 2 3
	2 3 4

Từ nay trở đi, khi xử lý bài toán trên mảng, ta hoàn toàn có thể sử dụng con trỏ để xử lý. Chúng hoàn toàn tương đương nhau. Chỉ có duy nhất một sự khác biệt trong khai báo: nếu khai báo theo kiểu truyền thống, ta cần chỉ ra kích thước ngay khi khai báo, còn khai báo theo kiểu con trỏ, ta có thể chỉ định kích thước sau nhờ vào toán tử new. Chúng ta sẽ tìm hiểu chi tiết về toán tử new trong chương tiếp theo.

Con trỏ void

Con trỏ void là loại con trỏ đặc biệt. Trong C++, void dùng để quy định sự không tồn tại của một kiểu dữ liệu (hay kiểu dữ liệu rỗng). Vì vậy, con trỏ void là con trỏ trỏ vào giá trị có kiểu dữ liệu void (cũng vì lẽ đó, mà nó không xác định độ dài và thuộc tính tham chiếu ngược).

Con trỏ void cho phép trỏ sang một kiểu dữ liệu bất kì. Nhưng khi chuyển đổi, chúng có một giới hạn rất lớn: dữ liệu trỏ bởi chúng không thể trực tiếp tham chiếu ngược, và vì nguyên nhân này, chúng ta cần ép kiểu địa chỉ của con trỏ void sang một con trỏ khác mà nó có một kiểu dữ liệu cụ thể trước khi tham chiếu ngược trở lại.

Chương trình	**Kết quả**

#include<iostream>
using namespace std;
void increase(void* data, int psize)
{
  if(psize==sizeof(char))
  {
  char* pchar;
  pchar=(char*)data;
  ++(*pchar);
  }else if(psize==sizeof(int))
  {
  int* pint;
  pint = (int*)data;
  ++(*pint);
  }
}
int main()
{
  char a = 'x';
  int b = 1602;
  increase(&a, sizeof(a));
  increase(&b, sizeof(b));
  cout<<a<<", "<<b<<endl;
  return 0;
}
	y, 1603

Giải thích: hàm increase có hai tham số: tham số data là một con trỏ void, tham số psize là kích thước của con trỏ data. Câu lệnh if sẽ kiểm tra điều kiện xem biến trỏ data thuộc kiểu dữ liệu nào – nếu psize==sizeof(char) thì

nó là con trỏ kiểu char, tương tự nếu psize==sizeof(int) thì nó là con trỏ kiểu int. Vì ở đây, ta chưa xác định được nó là con trỏ kiểu gì, nên ta sẽ sử dụng tham số là con trỏ void. Nếu là con trỏ char, ta sẽ sử dụng biến trỏ pchar để khởi tạo giá trị cho nó bằng cách ép kiểu từ con trỏ void. Hoàn toàn tương tự cho biến trỏ pint. Mục đích của hàm increase là tìm giá trị tiếp theo của tham số data. Trong hàm main, ta sử dụng hai biến char và int. Giá trị tiếp theo của kí tự 'x' là kí tự 'y', của số 1602 là 1603.

Con trỏ null

Một con trỏ null là con trỏ dùng để khởi gán cho một biến trỏ có kiểu dữ liệu bất kì. Trong C++, nó được quy định là 0.

```
int *p;
p = 0;
char *c;
c = 0;
```

Ta không nên nhầm lẫn giữa con trỏ null và con trỏ void. Một con trỏ null là một giá trị cụ thể mà mọi con trỏ đều có thể trỏ đến, và nó có thể trỏ đến "mọi vị trí". Trong khi đó, con trỏ void là con trỏ đặc biệt, nó có thể trỏ đến "vài vị trí".

Con trỏ hàm

C++ cho phép thực thi tính toán với con trỏ trỏ vào hàm. Khi thực thi điều này, nó sẽ xem hàm như là một tham số trỏ vào một hàm khác, nhưng chúng không tồn tại tham chiếu ngược. Để khai báo một con trỏ hàm, chúng ta cần khai báo nó như khai báo prototype cho một hàm, nhưng tên của hàm sẽ được bao trong dấu () đồng thời bổ sung * phía trước tên của nó.

Chương trình	**Kết quả**

#include<iostream>
using namespace std;
int addition(int a, int b)
{
  return a+b;
}
int substraction(int a, int b)
{
  return a-b;
	12
	8

```
}
int operation (int x, int y, int (*functocall) (int, int))
{
  int g;
  g=(*functocall)(x,y);
  return (g);
}
int main()
{
  int m, n;
  int (*minus)(int, int) = substraction;
  m = operation(7, 5, addition);
  cout<<m<<endl;
  n = operation(20, m, minus);
  cout<<n;
  return 0;
}
```

Giải thích: Hai hàm addition và substraction thực hiện các chức năng cộng và trừ hai số nguyên. Hàm operation sẽ gọi hàm tương ứng với con trỏ hàm được chỉ định. Trong hàm main, con trỏ hàm minus sẽ tương ứng với hàm substraction. Khi gọi hàm operation, tùy theo con trỏ hàm quy định ở tham số thứ ba, nó sẽ trỏ đến hàm tương ứng được chỉ định: operation(7, 5, addition) – trỏ đến hàm addition, tức cộng hai số 7 và 5. Tương tự, operation(20, m, minus) sẽ tương ứng với operation(20, m, substraction) và cho kết quả là phép trừ 20 cho m.

Bài tập 9.

1. Hãy xây dựng ba hàm cộng hai số nguyên **addi**, cộng hai số thực **addf** và cộng hai xâu **adds** bằng cách sử dụng con trỏ hàm. Hàm tổng quát sẽ được đặt tên là **add**.
2. Sử dụng con trỏ hàm để xây dựng hàm giải phương trình bậc nhất và phương trình bậc 2.
3. Sử dụng con trỏ hàm để tính các tích phân sau bằng phương pháp hình thang hoặc hình chữ nhật (phải, trái hoặc trung vị).

$$\int_0^1 \sin(x)\,dx$$

$$\int_0^1 \cos(x)\,dx$$

CHƯƠNG 10. BỘ NHỚ ĐỘNG

Mục đích tạo ra biến con trỏ là không chỉ xử lý các tác vụ tính toán, mà còn có thể quản lý bộ nhớ máy tính. Để thực hiện điều này, chúng ta cần khai báo số ô nhớ cung cấp cho mỗi biến con trỏ, khi không còn cần dùng đến chúng nữa, chúng ta có thể giải phóng các ô nhớ này đi. Để thực hiện được những tác vụ này, trong C++ cung cấp cho ta hai toán tử là new và delete.

Toán tử new và new[]

Để yêu cầu bộ nhớ động, chúng ta sử dụng toán tử new, theo sau nó là kiểu dữ liệu. Nếu nó là mảng của nhiều phần tử, thì số phần tử sẽ được ấn định bên trong dấu [] ngay sau kiểu dữ liệu. Khi đó, nó sẽ trả về con trỏ trỏ vào khối ô nhớ đầu tiên.

```
int*num = new int;//Khai báo biến trỏ int 4 byte
int*nums = new int[10];//Khai báo biến trỏ nums với 4*10 bytes
```

Tôi xin nhắc lại lần nữa, một kiểu dữ liệu tương ứng với một số lượng các ô nhớ được quy định sẵn. Như trong trường hợp này, trên hệ điều hành windows 32bit, biến num sẽ chiếm 4 bytes bộ nhớ bằng $2^{4.8}$ giá trị. Còn biến nums thì chiếm 40 byte bộ nhớ, tương ứng với $2^{40.8}$ giá trị. Nếu sử dụng hàm sizeof để kiểm tra kích thước của biến *nums trong trường hợp này, ta sẽ nhận được 4. Sở dĩ như vậy là vì, sau khi khởi tạo 10 phần tử kiểu int, con trỏ sẽ đặt vào phần tử đầu tiên, cho nên, khi sử dụng sizeof trong trường hợp này là sizeof của phần tử đầu tiên (tức kiểu int). Do đó, kết quả thu được là 4 bytes.

Khi khởi tạo cho biến trỏ trỏ vào biến trỏ khác (tạm gọi là biến trỏ nhiều tầng). Ta cần khởi tạo cho biến trỏ chung. Tương ứng với mỗi biến trỏ chung, sẽ có một mảng các biến trỏ khác tương ứng. Do đó, ta cần khởi tạo cho dãy biến trỏ này, nhờ vào vòng lặp for. Tương tự như vậy, chúng ta có thể khởi tạo cho biến trỏ đa tầng (có thể là hai, ba, bốn…).

```
int***num;
int length = 10;
//khởi tạo biến trỏ chung
num = new int**[length];
```

```
//khởi tạo biến trỏ tầng 1
for (int i=0; i<length; i++)
  num[i]=new int*[length];
//khởi tạo biến trỏ tầng 2
for (int i=0; i<length; i++)
  for(int j=0; j<length; j++)
    num[i][j]=new int[length];
```

Để truy cập đến từng ô nhớ, ta có thể truy cập theo num[i][j][k] hoặc *(*(*(num+i)+j)+k). Như vậy, ta có thể thấy rằng, thực chất con trỏ cũng là một mảng. Chỉ có một điểm khác biệt cơ bản là mảng có kích thước cố định ngay khi khai báo, con con trỏ có kích thước được chỉ định khi ta cần sử dụng bởi toán tử new.

Bộ nhớ động được yêu cầu bởi chương trình, và hệ điều hành sẽ cung cấp cho nó từ bộ nhớ heap. Tuy nhiên, bộ nhớ máy tính cũng hữu hạn, và nó có thể bị cạn kiệt. Chính vì lẽ đó, chúng ta cần đến một kĩ thuật để kiểm tra tình trạng này của bộ nhớ. C++ (và thậm chí C) cung cấp cho ta hai phương thức chuẩn để kiểm tra: vượt qua ngoại lệ (**throw** bad_alloc) và không vượt qua ngoại lệ (**nothrow** bad_alloc).

Bằng việc sử dụng một ngoại lệ là **bad_alloc**, hệ thống sẽ bỏ qua khi quá trình cung ứng ô nhớ bị thất bại (theo mặc định). Ngoại lệ là một tính năng mạnh mẽ của C++, chúng ta sẽ tìm hiểu chi tiết ở những chương sau. Nếu gặp một ngoại lệ, chúng ta cần cho biết có bỏ qua để tiếp tục chạy chương trình hay không. Toán tử new cung cấp theo phương thức mặc định là vượt qua (throw bad_alloc).

```
bobby = new int[5];
//Nếu thất bại, ngoại lệ sẽ bị vượt qua – throw bad_alloc
```

Nếu không vượt qua ngoại lệ, mà cố gắng tiếp tục chạy chương trình, thì

```
bobby = new (nothrow) int[5];
//Không vượt qua ngoại lệ, chương trình cố gắng tiếp tục
```

Nếu sử dụng vượt qua ngoại lệ throw bad_alloc, trong trường hợp cung ứng ô nhớ bị thất bại, con trỏ sẽ trả về con trỏ null, và chương trình vẫn tiếp tục. Chúng ta chỉ cần kiểm tra biến trỏ bobby. Nếu nó là null thì quá trình cung cấp bộ nhớ động thất bại và ngược lại thì thành công. Đây là cách làm thủ công, chúng ta có thể sử dụng cấu trúc try… catch để xử lý tình

huống này. Chi tiết, chúng ta sẽ thảo luận kĩ hơn trong phần sau. Lưu ý rằng, hai ngoại lệ này nằm trong ***thư viện <new>***.

Toán tử delete và delete[]

Nếu khi biến trỏ không cần dùng đến nữa, chúng ta có thể xóa bỏ các ô nhớ của nó để giải phóng bộ nhớ động. Quá trình này được thực thi bởi toán tử delete.

```
delete num;
delete[] nums;
```

Cách viết thứ nhất dùng để giải phóng con trỏ đơn tầng. Cách thứ hai dùng cho con trỏ đa tầng (hai tầng trở lên).

Toán tử delete chỉ có tác dụng với con trỏ được khởi tạo bởi toán tử new hoặc con trỏ null.

Chương trình	**Kết quả**

#include <iostream>
#include <new>
using namespace std;
int main()
{
  double *ptr[5];
  for (int i = 0; i < 5; i++)
  {
  ptr[i] = new (nothrow) double[50000000];
  if (ptr[i]!=0)
  cout << "Allocation is ok";
  else
  cout<<”Allocation fail”;
  }
  delete[] ptr;
  return 0;
}
	Allocation is ok
	Allocation is ok
	Allocation is ok
	Allocation is ok
	Allocation fail

Trong ANSI-C, hàm malloc, calloc, realloc và free được dùng thay cho new và delete và hiển nhiên chúng cũng hoạt động trên C++. Chúng ta không thảo luận chi tiết về hàm này, vì theo xu hướng của lập trình C++ hiện đại, người ta sử dụng hai toán tử new và delete mà ta thảo luận ở trên.

CHƯƠNG 11. KIỂU DỮ LIỆU STRUCT VÀ CON TRỎ STRUCT

Kiểu dữ liệu mảng mà chúng ta đã thảo luận ở trên chỉ giúp chúng ta lưu một tập dữ liệu cùng loại. Nếu chúng ta không đơn thuần lưu trữ cùng loại dữ liệu, mà có thể là nhiều kiểu dữ liệu khác nhau, thì mảng không thể giải quyết được vấn đề. Trong C++ (và cả C) cung cấp cho chúng ta một kiểu dữ liệu giúp ta giải quyết vấn đề này, đó là **struct**.

Struct

Một dữ liệu struct là một nhóm các phần tử có thể có kiểu dữ liệu khác nhau được đặt cùng một tên. Các phần tử dữ liệu này được gọi là các thành viên của struct. Cấu trúc khai báo struct trong C++ như sau

```
struct tên_struct{
  type   thành_viên_1;
  type   thành_viên_2;
  ...
} [tên_đối_tượng_struct];
```

Trong đó:

struct là một từ khóa.

tên_struct là tên kiểu dữ liệu struct mà ta định nghĩa.

thành_viên_1, … là các phần tử thành viên của struct.

tên_đối_tượng_struct: là tên biến thuộc kiểu tên_struct.

Hai cách khai báo sau đây là tương đương.

Ví dụ 1	**Ví dụ 2**

struct product{
  int weight;
  float price;
}
product apple;
product banana, melon;
	struct product{
	  int weight;
	  float price;
	}apple, banana, melon;

Giải thích: trong hai ví dụ trên product là kiểu dữ liệu struct mà ta tạo ra. Nó gồm có hai thành viên là weight và price. Tương ứng với kiểu dữ liệu

này, ta có các biến apple, banana, melon. Để khai báo nó, ta có thể sử dụng một trong hai cách theo như hai ví dụ này. Cần lưu ý rằng, các biến apple, banana, melon là các biến thuộc kiểu dữ liệu product.

Để truy cập đến các thành viên của biến struct, ta sử dụng toán tử chấm (.).

```
apple.weight = 200;
apple.price = 2;
banana.weight = 150;
banana.price = 1;
...
```

Chúng ta có thể hiểu apple.weight là khối lượng của táo, apple.price là giá tiền của táo... Có thể xem mỗi thành viên của một biến struct như là một biến độc lập mà ta có thể truy cập bằng cách **tên_biến.phần_tử_thành_viên**. Ta hoàn toàn có thể thực hiện các phép toán tương ứng với mỗi dữ liệu thành viên này (nghĩa là các phép toán đó tương ứng với các phép toán trên kiểu dữ liệu của các phần tử thành viên đó: nếu phần tử thành viên là kiểu nguyên thì ta có thể thực hiện các phép toán số nguyên với thành viên này, nếu phần tử thành viên là kiểu xâu thì có thể thực thi các phép toán với xâu cho biến thành viên này,...).

Ví dụ, tôi mua 400gam táo, và 300g chuối. Giá của mỗi 100g táo là 1$, và 100g chuối là 0.7$. Cần tính toán số tiền mà tôi cần trả.

Ví dụ

```
apple.weight = 400;
apple.price = (float)apple.weight*1/100;
banana.weight = 300;
banana.price = banana.weight*0.7/100;
float money = apple.price+banana.price;
```

Như trong ví dụ trên, tôi có thể thực hiện các phép toán số học với các phần tử thành viên là kiểu số như các biến số bình thường.

Struct là một kiểu dữ liệu do người dùng định nghĩa (nhờ vào từ khóa struct – từ khóa struct viết thường). Ta cũng có thể khai báo một mảng các phần tử thuộc kiểu struct. Ví dụ sau đây minh họa cho việc mua bán khi đi siêu thị.

Bài toán: *Trong siêu thị, giá táo và chuối được ấn định trên từng sản phẩm tùy thuộc chất lượng của sản phẩm, không phụ thuộc vào khối lượng của nó.*

a. *Tính khối lượng hàng hóa (kg) mà người tiêu dùng mua.*
b. *Nếu khách hàng mua hàng trị giá trên 10$, thì người tiêu dùng sẽ được khuyến mãi. Hãy kiểm tra xem người tiêu dùng có được khuyến mãi hay không.*

Biết rằng số lượng táo và chuối do người tiêu dùng lựa chọn.

Chương trình	**Kết quả**

#include<iostream>
using namespace std;
#define MAX 10
struct product{
  int weight;
  float price;
}apples[MAX], bananas[MAX];
/*
Khai báo hàm
*/
void buyProducts(product pd, string name, int &weight, float &price)
{
  int i = 0;
  cout<<"Buy "<<name<<endl;
  while (true){
  cout<<"weight and price: <<endl;
  cin>>pd[i].weight;
  cin>>pd[i].price;
  weight += pd[i].weight;
  price += pd[i].price;
  cout<<"Continue to buy "<<name<<" (y/n) ?";
  char yn;
  cin>>yn;
  if((yn=='n')||(yn=='N')) break;
  else i++;
  }
}
int main()
{
	Buy apples
	weight and price
	100
	2
	Continue to buy apples (y/n) ?y
	weight and price
	200
	3
	Continue to buy apples (y/n) ?n
	Buy bananas
	weight and price
	150
	1
	Continue to buy bananas (y/n) ?y
	weight and price
	250
	3
	Continue to buy bananas (y/n) ?n
	Total weight: 0.7
	No Promotion

```
  int weight = 0;
  float price = 0;
  buyProducts(apples, “apples”, weight,
price);
  buyProducts(bananas, “bananas”,
weight, price);
  cout<<”Total weight:
“<<(float)weight/1000<<endl;
  if(price>10)
    cout<<”Promotion”;
  else
    cout<<”No Promotion”;
  return 0;
}
```

Giải thích:

Mảng apples và bananas thuộc kiểu dữ liệu product. Hàm buyProducts dùng để tính toán khối lượng và tổng giá hàng hóa mỗi loại mua được. Nếu tham số products trong hàm là apples, thì nó sẽ tính tương ứng với khối lượng tổng cộng và tổng đơn giá của apples. Tương tự cho bananas. Ta sử dụng vòng lặp while vì ta không biết chính xác số lượng hàng hóa mà người dùng lựa chọn. Tuy nhiên, con số tổng không thể vượt hằng số MAX mà ta đã định nghĩa. Biến weight và price được truyền theo tham biến, do đó, trước phép cộng dồn (toán tử cộng đồng nhất +=), ta không cần khởi tạo giá trị 0 cho chúng, ta có thể khởi tạo cho chúng ngay đầu hàm main. Nếu khởi tạo đầu hàm buyProducts, thì ta chỉ có thể tính được khối lượng và giá của từng loại một mà thôi (khi đó, nếu muốn tính toán cho cả hai sản phẩm, ta bổ sung thêm biến để lưu lại các giá trị này). Sau khi gọi hàm này hai lần, tương ứng với apples và bananas, thì biến weight lưu lại tổng khối lượng của cả hai sản phẩm, biến price lưu lại giá của cả hai sản phẩm. Khối lượng được quy đổi sang kg, nên ta sẽ chia cho 1000. Nếu quy định đơn vị là kg, thì điều này là không cần thiết.

Trong phần thực thi chương trình, người tiêu dùng đã mua hai quả táo và hai quả chuối. Quả táo thứ nhất có khối lượng 100g và giá 2$, quả táo thứ hai – 200g và 3$. Quả chuối thứ nhất có khối lượng 150g và giá 1$, quả chuối thứ hai – 250g và 3$.

Con trỏ struct

Tương ứng với mảng struct, ta cũng có con trỏ trỏ vào struct. Với con trỏ trỏ vào struct, ta có thể tạo ra một mảng struct động.

Khai báo con trỏ struct: có hai cách khai báo

```
struct product{
  int weight;
  float price;
}*apples;
```

```
struct product{
  int weight;
  float price;
};
product* apples;
```

Tham chiếu: hoàn toàn tương tự như con trỏ trỏ vào kiểu dữ liệu nguyên thủy.

```
struct product{
  int weight;
  float price;
}apples1;
…
products* apples2;
apples2 = &apples1;
```

Truy cập giá trị của phần tử thành viên: có hai cách truy cập

apples2->weight	*(apples2.weight)
apples2->price	*(apples2.price)

Ta cần lưu ý sự khác nhau giữa apples2->weight và *apples2.weight. Trường hợp đầu nó sẽ tương đương với *(apples2.weight) và theo độ ưu tiên của toán tử, dấu () sẽ thực hiện trước tiên, nghĩa là thành viên weight của apples2. Tiếp đó là phép toán *. Điều này có thể hiểu là con trỏ apples2 trỏ vào địa chỉ của biến thành viên weight của apples 2. Trường hợp thứ hai, là giá trị trỏ bởi biến thành viên weight của apples2 (toán tử . có độ ưu tiên cao hơn toán tử *).

```
struct product{
  int weight;
  float price;
}*apples2;
….
apples2->weight
```

```
struct product{
  int *weight;
  float price;
}apples2;
…
*apples2.weight
```

Struct lồng nhau

Struct có thể được khai báo lồng nhau. Chúng ta có thể khai báo lồng bên trong, hoặc khai báo tuần tự

Khai báo lồng nhau	Khai báo tuần tự

struct family{
  string father;
  string mother;
  struct children{
  string son;
  string daughter;
  }first, second;
}myfamily;
	struct children{
	  string son;
	  string daughter;
	};
	struct family{
	  string father;
	  string mother;
	  children first, second;
	}myfamily;

Khi đó, việc truy cập cũng tương tự như trên

```
myfamily.first.son
....
```

Nếu là con trỏ, ta cũng có cách truy cập tương tự.

struct family{
  string father;
  string mother;
  struct children{
  string son;
  string daughter;
  }*first, second;
}*myfamily;
	myfamily->first->son
	myfamily->second.son

Kích thước bộ nhớ của struct

Kích thước của struct theo lý thuyết nó sẽ là kích thước tổng cộng của các dữ liệu thành viên. Tuy nhiên, theo cách thức tổ chức bộ nhớ, các dữ liệu thành viên của một struct sẽ được sắp xếp liền kề nhau. Việc tổ chức bộ nhớ của hệ thống máy tính 32bit sẽ theo xu hướng là nhóm 4 bytes. Điều này có nghĩa là, nếu dữ liệu thành viên thứ nhất đã lấp vào một số byte bộ nhớ, và nếu còn thừa, thì hệ điều hành sẽ xem xét để đưa dữ liệu thành viên tiếp theo vào. Nếu dữ liệu thành viên tiếp theo chiếm một số lượng các byte bộ nhớ còn thừa, thì nó sẽ được lấp vào phần bộ nhớ còn thừa đó. Nhưng nếu vượt quá, thì hệ thống sẽ được định hướng bổ sung thêm một nhóm 4 byte bộ nhớ mới để chứa dữ liệu thành viên tiếp theo (nếu cần nhiều hơn

thì cung cấp tiếp nhóm 4 byte nữa). Còn đối với số byte còn thừa, nó vẫn được để trống.

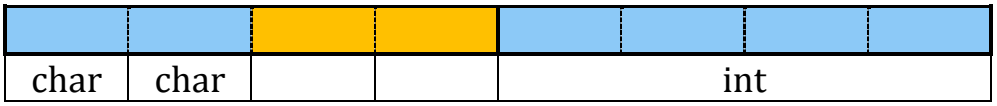
Ví dụ 1	**Ví dụ 2**

struct number{
char a;
int b;
char c;
}mynum;
…
cout<<sizeof(mynum);
//12
	struct number{
	char a;
	char c;
	int b;
	}mynum;
	…
	cout<<sizeof(mynum);
	//8

Giải thích:

Trong ví dụ 1: char – 1byte int – 4byte Đầu tiên, dữ liệu thành viên char sẽ lấp đầy nhóm 4byte, char chiếm 1byte nên còn thừa 3 byte. Dữ liệu thành viên tiếp theo là int. Vì int chiếm 4byte, nhưng chỉ thừa có 3 byte, nó không đủ chỗ cho int. Do đó, hệ thống sẽ tự động cung cấp thêm 3 byte để lấp đầy phần còn thừa. Tiếp theo, sẽ cung cấp nhóm 4 byte cho int. Không có byte thừa trong trường hợp này. Kiểu dữ liệu thành viên char còn lại chiếm 1byte, nhưng hệ thống cung cấp cho nó 4 byte, còn thừa 3 byte. Số byte thừa này sẽ được lấp đầy. Như vậy, số byte trong trường hợp này là (1+3)+4+(1+3)=12 byte.	Trong ví dụ 2: Cũng tương tự ví dụ 1. Dữ liệu thành viên char thứ nhất chiếm 1 byte, hệ thống cung cấp 4 byte, như vậy thừa 3 byte. Kiểu dữ liệu thành viên tiếp theo cũng kiểu char, chiếm 1byte, nó có thể lấp vào số byte còn thừa. Tại thời điểm này, còn thừa 2 byte. Tiếp theo, ta xét đến dữ liệu thành viên int. Dữ liệu này chiếm 4 byte, nhưng chỉ còn thừa 2 byte, do đó hệ thống sẽ lấp đầy 2 byte này và cung cấp thêm 4 byte mới cho int. Như vậy, số byte trong trường hợp này là (1+1+2)+4=8 byte.

Như vậy, tùy vào kiểu dữ liệu thành viên, ta cần chọn một cách sắp xếp hợp lý để có một kết quả tối ưu về bộ nhớ. Hãy quan sát lược đồ sắp xếp bộ nhớ hướng tối ưu như bên dưới (trường hợp 2).



CHƯƠNG 12. CÁC KIỂU DỮ LIỆU KHÁC

Kiểu dữ liệu tự định nghĩa

Ta có thể định nghĩa một kiểu dữ liệu mới thông qua một kiểu dữ liệu đã tồn tại

```
typedef   kiểu_dữ_liệu_tồn_tại   tên_kiểu_dữ_liệu_mới;
```

Sau khi khai báo kiểu dữ liệu mới, ta có thể sử dụng nó như kiểu dữ liệu thông thường

```
typedef char C;
typedef char* pchar;
typedef char string50[50];
…
C mychar;
pchar p;
string50 str;
```

Kiểu dữ liệu union thường

Với từ khóa union, ta có thể khai báo một kiểu dữ liệu mới từ các kiểu dữ liệu đã tồn tại.

```
union Tên_kiểu_union{
  kiểu_dữ_liệu_thành_viên_1    tên_thành_viên_1;
  kiểu_dữ_liệu_thành_viên_2    tên_thành_viên_2;
  …
}biến_kiểu_dữ_liệu_union;
```

Khi khái báo kiểu union, các dữ liệu thành viên sẽ được khai báo trên cùng một không gian bộ nhớ vật lý. Kích thước của nó là kích thước của kiểu dữ liệu thành viên lớn nhất.

Ví dụ

```
union myType{
  char C;
  double Db;
} newType;
//8 byte, vì double = 8byte, char = 1byte.
```

```
…
//Sử dụng
biến_kiểu_dữ_liệu_union.tên_thành_viên…
```

Kiểu dữ liệu union ẩn danh

Hoàn toàn tương tự với kiểu dữ liệu union thường, nhưng ta không cần chỉ định tên biến union mới tạo ra

```
union Tên_kiểu_union{
  kiểu_dữ_liệu_thành_viên_1    tên_thành_viên_1;
  kiểu_dữ_liệu_thành_viên_2    tên_thành_viên_2;
  …
};
```

Khi khai báo biến thuộc kiểu union ẩn danh, ta có thể khai báo như sau

Ví dụ

…

union Tên_kiểu_union tên_biến;

tên_biến.tên_thành_viên

…

Kiểu dữ liệu enum

Kiểu dữ liệu enum được tạo mới có thể chứa nhiều dữ liệu khác nhau mà không có sự giới hạn về giá trị.

```
enum Tên_kiểu_enum{
  giá_trị_1;
  giá_trị_2;
  …
}Tên_thể_hiện_enum;
```

Chúng ta có thể tham khảo thêm các ví dụ sau đây

```
enum colors{black=0, blue, green, cyan, red} mycolor;
enum colors{black=0, blue, green, cyan, red};
colors mycolor;
```

Enum là một loại dữ liệu chứa các biến được đánh chỉ số. Do đó, các giá trị hằng của nó luôn là số nguyên (để làm chỉ số).

Ví dụ

```
#include<iostream>

using namespace std;
enum colors{red, green, blue};

int main()
{
  for(int i=0; i<3; i++)
   if (colors(i)==green) cout<<"green"<<endl;
  return 0;
}
```

CHƯƠNG 13. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Lịch sử hình thành

Trước khi kĩ thuật lập trình hướng đối tượng ra đời, con người đã trải qua các thời kì lập trình tuyến tính, lập trình hướng thủ tục.

1. Lập trình tuyến tính

Máy tính đầu tiên được lập trình bằng mã nhị phân, sử dụng các công tắc cơ khí để nạp chương trình. Cùng với sự xuất hiện của các thiết bị lưu trữ lớn và bộ nhớ máy tính có dung lượng lớn, nên các ngôn ngữ lập trình cấp cao bắt đầu xuất hiện. Các ngôn ngữ lập trình này được thiết kế làm cho công việc lập trình trở nên đơn giản hơn. Các chương trình ban đầu chủ yếu liên quan đến tính toán, chúng tương đối ngắn. Chúng chủ yếu chạy theo các dòng lệnh một cách tuần tự, dòng trước chạy trước, dòng sau chạy sau.

Nhược điểm:

- Nếu ta cần sử dụng một đoạn lệnh nào đó nhiều lần, thì ta phải sao chép nó nhiều lần.
- Không có khả năng kiểm soát phạm vi nhìn thấy của biến.
- Chương trình dài dòng, khó hiểu, khó nâng cấp.

2. Lập trình hướng thủ tục

Với những nhược điểm trên, đòi hỏi có một ngôn ngữ lập trình mới thay thế. Đó chính là nguyên nhân ra đời của ngôn ngữ lập trình hướng thủ tục. Về bản chất, chương trình được chia nhỏ thành các modul (đơn vị chương trình). Mỗi đơn vị chương trình chứa các hàm hay thủ tục (nên gọi là hướng thủ tục). Tuy tách rời thành các modul riêng biệt, nhưng ngôn ngữ lập trình hướng thủ tục vẫn đảm bảo thông tin thông suốt giữa các modul nhờ vào cơ chế hoạt động của hàm, cơ chế truyền theo tham biến và tham trị. Với lập trình hướng thủ tục, một chương trình lớn có thể được chia nhỏ

thành các modul, để mỗi lập trình viên có thể đảm nhận. Tiêu biểu trong số này là C, Pascal.

Nhược điểm:

- Các hàm và thủ tục thường gắn kết với nhau, nếu muốn nâng cấp chương trình, thường phải chỉnh sửa tất cả các hàm và thủ tục liên quan.
- Không phù hợp với xu thế hiện đại vì không mô tả được thực thể trong cuộc sống thực.

3. Lập trình hướng đối tượng

a. Giới thiệu

Với xu thế hiện đại, ngôn ngữ lập trình hướng đối tượng đã ra đời. Cơ sở của lập trình hướng đối tượng là đối tượng. Đối tượng là sự thể hiện của một thực thể trong thế giới thực. Một thực thể trong thế giới thực thường có: các đặc trưng và các hành động. Ví dụ: con người trong thế giới thực có các đặc trưng như - tên gọi, tuổi, màu tóc, màu mắt, màu da… và các hành động như – ăn, nói, chạy, nhảy… Cách thức lập trình này mô tả một cách chính xác các sự vật, con người trong thế giới thực.

Bây giờ, ta sẽ xét một vài ví dụ để cho thấy sự cần thiết của lập trình hướng đối tượng.

Ví dụ 1. Chúng ta muốn xây dựng một chương trình quản lý sinh viên. Khi đó, ta cần lưu trữ các thông tin liên quan đến đối tượng sinh viên này: họ tên sinh viên, mã số sinh viên, ngày tháng năm sinh, quê quán, điểm các môn, điểm tổng kết,…. và rất nhiều thông tin khác liên quan. Sau khi kết thúc năm học, sinh viên sẽ nhận được đánh giá kết quả học tập của mình. Chúng ta cần có phương thức tiếp nhận kết quả để sinh viên đó có thể phản ứng lại với những gì mà mình nhận được, họ phải thực hiện các hành động học tập, tham gia vào các hoạt động của trường, của khoa… đó là những hành động mà mỗi sinh viên cần thực hiện.

Ví dụ 2. Chúng ta sẽ điểm qua một số tính năng trong chương trình soạn thảo văn bản Word của Microsoft. Chúng ta sẽ thảo luận về các đối tượng Drawing trong Word. Mỗi đối tượng đều có các thuộc tính: màu viền, dạng đường viền, kích thước viền, màu sắc viền, màu nền, có văn bản hay không trong đối tượng drawing…Khi chúng ta biến đổi hình dạng của mỗi đối

tượng: kéo giãn, làm lệch xiêng, quay vòng… chúng ta cần đưa ra một thông điệp để các đối tượng hình thể này thay đổi theo. Các hành động này thuộc quyền sở hữu của đối tượng.

Trong hai ví dụ minh họa trên, chúng ta thấy rằng hướng tiếp cận theo lập trình hướng đối tượng là rất gần gũi với cuộc sống thực. Chúng ta không quan tâm đến những khía cạnh không cần thiết của đối tượng, chúng ta chỉ tập trung vào các đặc trưng và các hành động của đối tượng. Kể từ thời điểm này trở đi, chúng ta sẽ gọi các đặc trưng của đối tượng là các thuộc tính thành viên của đối tượng đó (hoặc dữ liệu thành viên, biến thành viên của đối tượng) và các hành động của đối tượng là các phương thức thành viên (hay hàm thành viên) của đối tượng. Các cách gọi dữ liệu thành viên, thuộc tính thành viên, biến thành viên hay thuộc tính (tương ứng phương thức thành viên, hàm thành viên, phương thức) là không có sự phân biệt. Tôi chỉ đưa ra nhiều cách gọi khác nhau để chúng ta có thể quen khi tham khảo các giáo trình khác nhau. Bởi lẽ, nhiều giáo trình chọn lựa các cách gọi khác nhau. Các cách gọi này cũng tùy thuộc vào ngôn ngữ lập trình (trong C++ thông thường người ta sử dụng khái niệm dữ liệu thành viên – member data hoặc biến thành viên – member variable và hàm thành viên – member function, trong khi đó, các ngôn ngữ như Java, Delphi hay C# lại sử dụng khái niệm phương thức – method và thuộc tính – property). Khái niệm thành viên sẽ áp dụng cho cả dữ liệu thành viên lẫn hàm thành viên.

Phương châm của lập trình hướng thủ tục theo giáo sư Niklaus Wirth

Chương trình = Cấu trúc dữ liệu + Giải thuật

Còn phương châm của lập trình hướng đối tượng là

Chương trình = Đối tượng + Dữ liệu

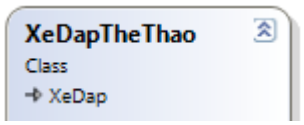
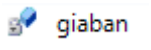
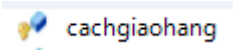
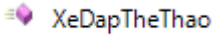
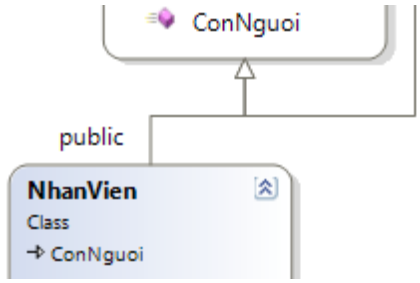
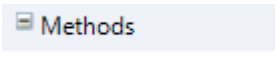
Tiêu biểu trong số này là C++, Java, C#, Delphi, Python…

b. Phương pháp phân tích và thiết kế hướng đối tượng

Trước khi bắt đầu viết một chương trình theo hướng đối tượng, thì ta cần phân tích và thiết kế các đối tượng. Từ sơ đồ cấu trúc nhận được, chúng ta có thể xây dựng nên chương trình. Chi tiết về cách thức phân tích và thiết kế đối tượng, chúng ta sẽ được tìm hiểu kĩ hơn trong học phần phân tích thiết kế hệ thống thông tin. Trong nội dung của giáo trình này, chúng ta chỉ thảo luận một phần nhỏ, để giúp các bạn có thể xây dựng nên một cấu trúc chương trình theo hướng đối tượng. Sơ đồ cấu trúc trong lập trình hướng đối tượng được sử dụng phổ biến là sơ đồ mô tả trên ngôn ngữ UML (Unified Modeling Languages). UML là ngôn ngữ chuyên dùng để mô hình

hóa các đối tượng. Nó không chỉ được áp dụng trong lập trình, mà còn được sử dụng rộng rãi trong các lĩnh vực khác của cuộc sống. Trong UML có nhiều dạng sơ đồ được hoạch định. Nhưng trong phạm trù của lập trình hướng đối tượng, sơ đồ lớp là sự mô tả gần gũi nhất. Do đó, tôi sẽ trình bày cách xây dựng một chương trình được mô tả bằng sơ đồ lớp.

Một số kí hiệu cần lưu ý trong UML. Trước khi tìm hiểu cách mô hình hóa một bài toán trong UML, chúng ta cần tìm hiểu một số kí hiệu được sử dụng trong UML. Các kí hiệu này có thể khác nhau trong nhiều chương trình mô phỏng. Những kí hiệu mà tôi sử dụng ở đây là kí hiệu dùng trên Visual Studio 2010.

Kí hiệu	Ý nghĩa
	Lớp
	Thuộc tính hoặc phương thức private
	Thuộc tính hoặc phương thức protected
	Thuộc tính hoặc phương thức public
	Biểu diễn tính kế thừa. Mũi tên luôn hướng về lớp cơ sở. Chiều còn lại luôn chỉ vào lớp con.
	Thuộc tính
	Phương thức

Phân tích và thiết kế mô hình. Việc phân tích và thiết kế một mô hình là công việc đòi hỏi các nhà thiết kế phải có một tư duy tốt. Đối với một bài toán phân tích và thiết kế, không phải chỉ có duy nhất một mô hình kết quả, mà có thể có một vài, thậm chí là nhiều mô hình khác nhau. Tuy nhiên, công việc chọn lựa một mô hình tối ưu là điều cần thiết. Trong nội dung giáo trình này, tôi chỉ giới thiệu sơ qua về cách hoạch định một mô hình. Chúng ta sẽ không đi sâu nghiên cứu vấn đề này. Trong học phần phân tích và thiết kế hệ thống thông tin sẽ trình bày chi tiết và cụ thể hơn.

Các bước phân tích và thiết kế. Để phân tích và thiết kế một mô hình hướng đối tượng, cần thực hiện các giai đoạn sau đây:

- **Bước 1**. Mô tả bài toán. Một bài toán sẽ được miêu tả dưới dạng ngôn ngữ tự nhiên. Nó chủ yếu dựa vào yêu cầu của khác hàng và sự trợ giúp khách hàng.
- **Bước 2**. Đặc tả các yêu cầu. Sau khi phân tích các nhân tố tham gia vào trong mô hình, ta cần tiến hành xem xét các tác nhân tác động vào từng nhân tố. Mối quan hệ giữa các nhân tố…
- **Bước 3**. Trích chọn đối tượng. Sau khi tổng hợp các tác nhân và nhân tố trong mô hình. Chúng ta cần tiến hành lựa chọn chúng. Việc loại bỏ các nhân tố và tác nhân không cần thiết là rất quan trọng. Nó sẽ giúp cho mô hình tập trung vào các nhân tố quan trọng và cần thiết, tránh phân tích và thiết kế tràn lan.
- **Bước 3**. Mô hình hóa các lớp đối tượng. Sau khi chọn lựa các đối tượng cần thiết. Chúng ta phân tích từng đối tượng. Khi phân tích đối tượng, ta cần lưu ý tập trung vào những thứ cốt lõi của mỗi đối tượng, tránh đưa vào những phương thức và thuộc tính không cần thiết, không quan trọng của đối tượng – đó chính là tính trừu tượng hóa của dữ liệu. Khi phân tích, chúng ta cũng cần lưu ý đến các tính chất chung của từng đối tượng. Nếu các đối tượng có nhiều tính chất chung, chúng ta nên xây dựng một đối tượng mới, chứa các tính chất chung đó, mỗi đối tượng còn lại sẽ thừa kế từ đối tượng này, để nhận được các tính chất chung.
- **Bước 4**. Thiết kế từng đối tượng. Chúng ta cần đảm bảo rằng, mỗi đối tượng phải có các phương thức và thuộc tính riêng lẫn các phương thức và thuộc tính chia sẻ. Các phương thức riêng chỉ có bản thân đối tượng mới có quyền thay đổi. Các phương thức chia sẻ có thể được truy cập bởi đối tượng khác theo các mức khác nhau.
- **Bước 5**. Xây dựng và kiểm thử mô hình. Bắt tay vào xây dựng mô hình. Ở đây, chúng ta sử dụng ngôn ngữ UML để mô tả. Sau khi xây dựng xong mô hình, cần tiến hành kiểm thử mô hình. Kiểm thử các mô hình trong các tình huống thực tế là cần thiết để đảm bảo rằng mô hình nhận được là phù hợp, trước khi bắt tay vào viết chương trình.

Trên đây, chỉ là những bước đề nghị để chúng ta có một cái nhìn tổng quát khi phân tích và thiết kế. Có thể có nhiều cách để phần tích và thiết kế một mô hình. Nhưng hãy luôn đảm bảo rằng, mô hình thu được không những đạt hiệu quả cao, mà còn đảm bảo rằng nó phải dễ dàng bảo trì và nâng cấp. Mỗi khi có một lỗi xuất hiện, chúng ta cũng cần biết khoanh vùng để thu nhỏ phạm vi phát hiện lỗi.

Chúng ta sẽ lấy một ví dụ nhỏ. Phân tích hướng đối tượng mô hình quản lý cửa hàng bán xe đạp. Trong mô hình này, ta cần quản lý các nhóm đối tượng sau: đối tượng xe đạp, đối tượng chi nhánh bán hàng, đối tượng khách hàng và đối tượng nhân viên bán hàng.

- Đối tượng xe đạp: chúng ta cần quản lý mã số xe (mã số gồm hai phần: phần id chi nhánh bán hàng + mã số vạch), loại xe, màu sắc, giá bán, nước

sản xuất (các thuộc tính chung). Đối tượng xe đạp địa hình: số bánh răng, cách lên răng (bằng tay/tự động), chống sooc. Đối tượng xe đạp du lịch: xe đơn/đôi, tự động (hỗ trợ tự chạy bằng điện hay không), chiếu sáng (có/không). Xe đua thể thao: điều chỉnh độ cao (có/không), các chế độ đạp (đạp thư giản, đạp tăng tốc, đạp chậm…).

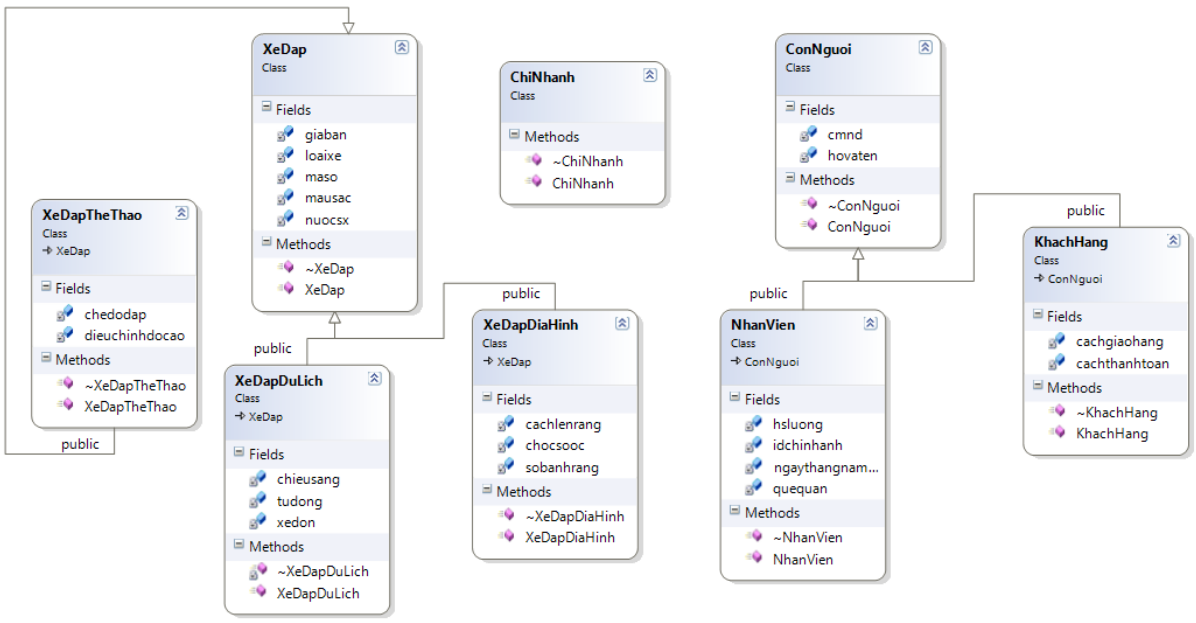
- Đối tượng khách hàng và nhân viên bán hàng: họ và tên, số CMND. Đối tượng khách hàng: cách thức thanh toán (tiền mặt/chuyển khoản), cách thức giao hàng (nhận tại chỗ/ đưa đến tận nhà). Đối tượng nhân viên bán hàng: id chi nhánh bán hàng, ngày tháng năm sinh, quê quán, địa chỉ, mã số thuế, lương…
- Đối tượng chi nhánh bán hàng: id chi nhánh bán hàng, địa chỉ.

Nếu yêu cầu quản lý nhiều hơn các thông tin của từng đối tượng, khi đó ta cần bổ sung thêm các thuộc tính tương ứng này.

Đối với các phương thức thực thi hành động, tương ứng với mỗi thuộc tính, ta có hai phương thức để chỉ định và tiếp nhận. Ví dụ, đối tượng nhân viên, có họ và tên. Tương ứng với thuộc tính này, ta có phương thức chỉ định để đặt tên cho nhân viên (đặt tên là thiết lập tên gọi trong phần mềm quản lý) và tiếp nhận tên khi có yêu cầu.

Đối tượng khách hàng có phương thức quyết định (quyết định thực hiện giao dịch). Đối tượng nhân viên bán hàng có phương thức tiếp nhận (nhận giao dịch). Đối tượng địa điểm bán hàng có phương thức nhận hàng (nếu hàng còn đầy thì không tiếp nhận thêm).

Theo như phân tích ở trên, đối tượng xe đạp là đối tượng chung. Các đối tượng xe đạp thể thao, xe đạp du lịch, xe đạp địa hình kế thừa từ lớp xe đạp. Đối tượng con người để quản lý thông tin chung. Và các đối tượng nhân viên và khách hàng thừa kế từ lớp con người. Cuối cùng là đối tượng chi nhánh bán hàng. Theo như cách phân tích này, ta có sơ đồ lớp như sau:



Hình 20 – Minh họa sơ đồ lớp

Trong sơ đồ này, các phương thức và thuộc tính của mỗi lớp đối tượng như đã phân tích ở trên. Để tránh rườm rà, các phương thức chỉ biểu diễn sơ lược.

Lớp và đối tượng

Lớp là sự biểu diễn của đối tượng trong lập trình và ngược lại đối tượng là sự thể hiện của lớp. Một đối tượng gồm có: thuộc tính và phương thức. Chúng ta có thể xem một lớp như là một kiểu dữ liệu, còn đối tượng là biến. Lớp được khai báo nhờ từ khóa **class**

```
class tên_lớp{
  các_thuộc_tính
  các_phương_thức
} [tên_đối_tượng];
```

tên_lớp: là tên của lớp.

tên_đối_tượng: là tên của đối tượng.

Khai báo lớp tương đối giống khai báo struct.

Các thuộc tính được khai báo như khai báo biến. Các phương thức được khai báo như khai báo hàm. Chúng có thể được chỉ định bằng một trong ba từ khóa: private, protected và public.

- **private**: các thành viên (biến thành viên hoặc hàm thành viên) của cùng một lớp **hoặc** từ một hàm bạn của nó có thể truy cập.
- **protected**: các thành viên của cùng một lớp hoặc từ một hàm bạn của nó hoặc từ một lớp dẫn xuất của nó hoặc bạn của một lớp dẫn xuất của nó đều có thể truy cập. Mức truy cập lớn nhất trong trường hợp này là bạn của lớp dẫn xuất. Chúng ta sẽ thảo luận thêm trong những phần sau.
- **public**: các thành viên có thể truy cập lẫn nhau từ mọi lớp.

Theo mặc định, nếu không chỉ định từ khóa, thì private được ấn định.

```
class Humans{
  string name;
  int age;
public:
  void setName(string);
  void setAge(string);
  string getName(void);
  int getAge(void);
```

```
}man;
```

Humans là tên lớp, chứa các thuộc tính là name và age, chúng không được chỉ định từ khóa, nên private sẽ được sử dụng. Các phương thức setName, setAge, getName và getAge được chỉ định là public. Trong trường hợp này, **man** là một đối tượng thể hiện của lớp Humans.

Sau khi khai báo lớp, ta cần bổ sung phần thân lớp – tương ứng với các hàm thành viên. Hoặc ta có thể bổ sung trực tiếp vào trong lớp – tương tự khai báo hàm trực tiếp, hoặc sử dụng khai báo prototype. Đối với khai báo prototype, để xác định một phương thức là của một lớp nào đó, ta sử dụng toán tử phạm vi **::** theo sau tên lớp.

Chương trình	**Kết quả**

#include<iostream>
using namespace std;
class Humans{
  string name;
  int age;
public:
  void setName(string);
  void setAge(int);
  string getName(void);
  int getAge(void);
};
void Humans::setName(string s){
  name = s;
}
void Humans::setAge (int a){
  age = a;
}
string Humans::getName(void){
  return name;
}
int Humans::getAge(void){
  return age;
}
int main(){
  Humans man;
  man.setName(“Jack”);
  man.setAge(21);
  cout<<”The man: “<<man.getName()<<”, age
	The man: Jack, age 21

"<<man.getAge(); return 0; }	

Giải thích: Hàm setName sẽ gán biến s cho thành viên name, tương tự cho hàm setAge sẽ gán biến a cho thành viên age. Hàm getName trả về dữ liệu nhận được từ thành viên name và hàm getAge – nhận được dữ liệu từ thành viên age. Hai phương thức setName và setAge gọi là phương thức **setter**. Phương thức getName và getAge gọi là phương thức **getter**. Các phương thức setter dùng để nhập dữ liệu cho các thuộc tính thành viên, các phương thức getter dùng để nhận giá trị từ các thuộc tính thành viên đã được nhập bởi setter (hoặc phương thức khởi tạo). Chúng ta cần lưu ý rằng, hàm main không thuộc lớp Humans, do đó, nó không thể truy xuất đến các thuộc tính thành viên trong trường hợp này, vì chúng được khai báo mặc định là private.

Toán tử phạm vi :: sẽ giới hạn sự truy cập bất hợp lệ của những hàm không thuộc lớp Humans hay là bạn của Humans. Đây là một cách thức để quy định một phương thức có phải là thành viên của một lớp hay không. Cách thức thứ hai, có thể khai báo hàm trực tiếp ngay bên trong lớp. Về bản chất, hai cách này không có sự khác biệt nào.

Lớp cũng có tác dụng như là một kiểu dữ liệu, do đó, ta cũng có thể khai báo nhiều đối tượng của cùng một lớp hay mảng các đối tượng.

Chương trình	**Kết quả**

#include<iostream>
using namespace std;
 	
class Humans{
  string name;
  int age;
public:
  void setName(string);
  void setAge(int);
  string getName(void);
  int getAge(void);
};
 	
void Humans::setName(string s){
  name = s;
	Name and age of Student 1
	Nam
	21
	Name and age of Student 2
	Binh
	22
	Name and age of Student 3
	Xuan
	22
	Name and age of Student 4
	Tuan

```
}
void Humans::setAge (int a){
  age = a;
}
string Humans::getName(void){
  return name;
}
int Humans::getAge(void){
  return age;
}
#define MAX 5

int main(){
  Humans man[MAX];
  for(int i=0; i<MAX; i++){
    string s;
    int a;
    cout<<"Name and age of Student "
<<(i+1)<<endl;
    cin>>s;
    cin>>a;
    man[i].setName(s);
    man[i].setAge(a);
  }
  cout<<"=====Students=====<<endl;
  for(int i=0; i<MAX; i++)
    cout<<"The man: "<<man[i].getName()<<",
age "<<man[i].getAge()<<endl;
  return 0;
}
```

```
21
Name and age of Student
5
Lan
22
=====Students=====
The man: Nam, age 21
The man: Binh, age 22
The man: Xuan, age 22
The man: Tuan, age 21
The man: Lan, age 22
```

Trong trường hợp này, biến man là một mảng các đối tượng Humans. Chương trình minh họa cho việc nhập tên sinh viên, tuổi của họ và lưu vào một mảng. Sau đó, xuất kết quả ra màn hình. Dù là cùng là sự thể hiện của lớp Humans, nhưng các đối tượng man[1], man[2],... có các thuộc tính hoàn toàn khác nhau.

Cơ sở của lập trình hướng đối tượng là dữ liệu thành viên và các hàm thành viên của một đối tượng. Chúng ta hoàn toàn không sử dụng một tập các biến toàn cục để truyền qua một hàm (hay tập các biến cục bộ truyền theo tham biến), mà thay vào đó, chúng ta sử dụng các đối tượng cùng với

dữ liệu thành viên và hàm thành viên của nó. Các hàm thành viên tác động trực tiếp lên các dữ liệu thành viên.

Hàm tạo và hàm hủy

Trước khi sử dụng một đối tượng, chúng ta cần khởi tạo giá trị cho nó để tránh gặp phải những giá trị không mong muốn khi thực thi chương trình. Một cách thức mà chúng ta đã sử dụng ở trên là sử dụng phương thức setter. Một phương thức đơn giản hơn, chúng ta có thể sử dụng hàm khởi tạo (hay gọi tắt là hàm tạo). Việc khai báo hàm tạo cũng tương tự như khai báo hàm thành viên khác, tuy nhiên nhất thiết tên hàm tạo phải trùng với tên lớp.

Chương trình	**Ví dụ**

#include<iostream>
using namespace std;
class Humans{
  string name;
  int age;
public:
  Humans(string, int);
  string getName(void);
  int getAge(void);
};
Humans::Humans (string s, int a){
  name = s;
  age = a;
}
string Humans::getName(void){
  return name;
}
int Humans::getAge(void){
  return age;
}
int main(){
  Humans man(“Jack”, 21);
  cout<<”The man: “<<man.getName()<<”, age “<<man.getAge();
  return 0;
}
	The man: Jack, age 21

Hàm tạo không có kiểu dữ liệu trả về - tương ứng với kiểu void. Tuy nhiên chúng ta sẽ **không sử dụng từ khóa void** trước khai báo hàm tạo.

Nếu một đối tượng đã được tạo ra, nhưng ta không muốn sử dụng đến nó nữa, để thu hồi bộ nhớ, ta cần sử dụng một phương thức để hủy bỏ các dữ liệu thành viên của nó – đó là hàm hủy. Hàm hủy cũng là một hàm thành viên của lớp. Nó không có kiểu dữ liệu trả về, nhưng ta cũng **không sử dụng từ khóa void** trước khai báo hàm hủy. Hàm hủy có tên trùng với tên lớp và phía trước tên hàm hủy là dấu ~. Hàm hủy sẽ tự động được gọi khi phạm vi hoạt động của đối tượng kết thúc. Phạm vi hoạt động của một đối tượng cũng giống như phạm vi hoạt động của một biến cục bộ - khai báo trong phạm vi nào, thì chỉ có tác dụng trong phạm vi đó.

Chương trình	**Ví dụ**

#include<iostream>
using namespace std;
class Humans{
  string name;
  int age;
public:
  Humans(string, int);
  ~Human();
  string getName(void);
  int getAge(void);
};
Humans::Humans (string s, int a){
  name = s;
  age = a;
}
Humans::~Humans()
{
  //do something
  //delete pointer;
}
string Humans::getName(void){
  return name;
}
int Humans::getAge(void){
  return age;
}
int main(){
  Humans man(“Jack”, 21);
  cout<<”The man: “<<man.getName()<<”, age “<<man.getAge();
	The man: Jack, age 21

return 0; }	

Chồng chất hàm tạo

Cũng như các hàm khác trong C++ cho phép chồng chất hàm, hàm tạo cũng có thể bị chồng chất. Khi chồng chất hàm tạo thì danh sách tham số phải khác nhau (kiểu dữ liệu hoặc số lượng tham số). Chúng ta cần nhớ rằng, khi chồng chất hàm thì trình biên dịch sẽ gọi một hàm tương ứng với danh sách tham số của nó. Trong trường hợp chồng chất hàm tạo, thì quá trình gọi là tự động khi đối tượng được tạo ra, do đó, một hàm tạo sẽ được gọi tự động tương ứng với danh sách tham số của nó.

Chương trình	**Ví dụ**

#include<iostream>
using namespace std;
class Humans{
  string name;
  int age;
public:
  Humans(void);
  Humans(string, int);
  string getName(void);
  int getAge(void);
};
Humans::Humans (void){
  name = "Default";
  age = 21;
}
Humans::Humans (string s, int a){
  name = s;
  age = a;
}
string Humans::getName(void){
  return name;
}
int Humans::getAge(void){
  return age;
}
int main(){
  Humans man("Jack", 21);
  Humans default;
	The man: Jack, age 21
	The default: Default, age 21

```
  cout<<"The man: "<<man.getName()<<", age "<<man.getAge()<<endl;
  cout<<"The default: "<<default.getName()<<", age "<<default.getAge();
  return 0;
}
```

Giải thích: trong trường hợp ví dụ trên, để tạo đối tượng thuộc lớp Humans, ta có thể sử dụng một trong hai hàm tạo tương ứng: Humans(void) hoặc Humans(string, int). Nếu gọi theo phương thức hàm tạo không đối số, thì tên gọi và tuổi sẽ được tạo mặc định. Còn nếu gọi theo phương thức có đối số, thì tên gọi và tuổi sẽ được tạo theo tham số truyền vào. Ta cũng cần lưu ý trong cách gọi hàm tạo, đối với hàm tạo có đối số, thì sau tên đối tượng, chúng ta cần cung cấp tham số tương ứng với tham số hàm tạo bên trong dấu (). Còn đối với hàm tạo không đối số, thì hãy khai báo nó như khai báo biến mà không hề có bất kì dấu () nào.

```
Humans man();   //Sai
Humans man;     //Đúng
Humans man = Humans();      //Đúng
```

Chú ý: Khi ta không khai báo một hàm tạo mặc định không tham số và chỉ khai báo hàm tạo mặc định có tham số, thì việc khai báo một đối tượng theo cách Humans man; là không được phép. Nếu ta không tạo ra một hàm tạo nào, thì điều này là hợp lệ.

Sao chép hàm tạo

Một đối tượng có thể được tạo ra từ hàm tạo theo cách khởi gán các dữ liệu thành viên của nó cho một giá trị nào đó. Chúng ta hoàn toàn có thể khởi tạo một đối tượng từ một đối tượng khác bằng cách sử dụng toán tử gán. Tuy nhiên, trong thực tế, nếu dữ liệu của đối tượng lớn, phức tạp, thì việc sử dụng toán tử gán sẽ thực thi rất chậm và có thể gây ra một số lỗi liên quan đến hủy đối tượng trong vùng bộ nhớ. Để khắc phục nhược điểm này, ta có thể sử dụng hàm tạo sao chép. Khi sử dụng hàm tạo sao chép, trình biên dịch sẽ sao chép toàn bộ dữ liệu thành viên của đối tượng đó sang đối tượng khởi tạo.

```
Humans man("Jack", 21);
//Sao chép trực tiếp
Humans man2 = man;
//Sao chép hàm tạo
```

```
Humans::Humans(const Humans& m){
  name = m.name;
  age = m.age;
}
Humans man2(man);
```

Chúng ta lưu ý rằng, việc sao chép hàm tạo sẽ được quy định theo tham chiếu hằng **const Humans&**. Nếu ta không viết hàm sao chép hàm tạo, thì trình biến dịch sẽ tự động làm giúp(nghĩa là ta luôn có thể sử dụng cách khởi tạo đối tượng theo kiểu Object newObj(oldObj); với oldObj là đối tượng thuộc lớp Object đã được tạo, dù ta có tạo ra phương thức sao chép hàm tạo hay không). Hay nói cách khác, **hàm tạo sao chép là mặc định đối với đại đa số trình biên dịch ANSI C++ hiện đại (GCC, Visual C++, Borland C++, Intel C++).**

Tham chiếu hằng. Phương thức sao chép hàm tạo (hoặc tổng quát là các phương thức có sử dụng tham chiếu hằng đến lớp đối tượng) có thể thực hiện theo tham chiếu hoặc tham chiếu hằng (tương ứng với không hoặc có từ khóa const), nhưng hãy luôn quy định là tham chiếu (có toán tử &). Khi quy định tham chiếu, đối tượng tham chiếu sẽ tham chiếu đến địa chỉ của đối tượng gốc. Dữ liệu của đối tượng tham chiếu sẽ được ánh xạ theo địa chỉ của đối tượng được tham chiếu (không thực hiện việc sao chép trực tiếp mà là gián tiếp thông qua địa chỉ của biến tham chiếu). Tuy nhiên, cũng vì lí do này mà đối tượng được tham chiếu có thể bị thay đổi giá trị (tương tự như truyền theo tham biến). Điều này làm vi phạm tính đóng gói trong lập trình hướng đối tượng. Cũng vì lí do này, C++ cung cấp cho ta từ khóa const để quy định một đối tượng khi được tham chiếu sẽ không bị làm thay đổi dữ liệu thành viên và nó được gọi là tham chiếu hằng. Như vậy, ta cần phân biệt ba cách truyền tham số đối tượng trong một phương thức: truyền theo tham trị – dữ liệu của đối tượng có thể được thay đổi bên trong phương thức nhưng sự thay đổi này không lưu lại, việc sao chép dữ liệu trong trường hợp này là thực thi trực tiếp nên thường chỉ áp dụng cho các kiểu dữ liệu nguyên thủy đơn giản; truyền theo tham chiếu – dữ liệu của đối tượng có thể bị thay đổi trong phương thức và nó được lưu lại, nó thực hiện việc sao chép dữ liệu một cách gián tiếp nên dữ liệu có cấu trúc phức tạp (như lớp đối tượng, con trỏ) có thể được sao chép nhanh hơn rất nhiều so với truyền theo tham trị; tham chiếu hằng – tương tự như tham chiếu nhưng không cho phép thay đổi dữ liệu của đối tượng ngay cả trong phương thức.

Hợp lệ	Không hợp lệ

…
class Humans{
  string name;
  int age;
public:
  Humans(string, int);
  Humans(const Humans&);
  string getName(void);
  int getAge(void);
};
Humans::Humans(const Humans& m){
  name = m.name;
  age = m.age;
}
…
	…
	class Humans{
	  string name;
	  int age;
	public:
	  Humans(string, int);
	  Humans(**const** Humans&);
	  ~Human();
	  string getName(void);
	  int getAge(void);
	};
	Humans::Humans(**const** Humans& m){
	  name = m.name;
	  age = m.age = 22;//Error
	}
	…

Ta có thể thấy trong trường hợp **không hợp lệ**, chúng ta quy định đối tượng được tham chiếu m sẽ cho phép đối tượng khác tham chiếu đến nó theo tham chiếu hằng. Nhưng đối tượng tham chiếu đến nó, lại cố gắng thay đổi thuộc tính age của nó. Trong trường hợp này, chương trình sẽ phát sinh lỗi. Nếu quy định là tham chiếu bình thường (bỏ đi từ khóa const) thì khai báo được xem là hợp lệ (tuy nhiên vi phạm tính đóng gói).

Khi một phương thức của lớp đối tượng sử dụng tham số chính là đối tượng của lớp đó, chúng ta có thể truy cập trực tiếp đến thuộc tính của đối tượng tham chiếu kể cả nó được quy định là private (cả tham chiếu lẫn không tham chiếu). Bên cạnh đó, nếu ta quy định là tham chiếu bình thường, thì ta có thể sử dụng các phương thức như getter và setter để truy cập đến các thuộc tính của nó. Nhưng nếu ta sử dụng tham chiếu hằng, thì không được phép truy cập đến các phương thức của đối tượng tham chiếu hằng. Chúng ta chỉ có thể truy cập đến các **phương thức hằng** của đối tượng tham chiếu hằng này. Phương thức hằng là những phương thức được bổ sung vào từ khóa **const** vào cuối khai báo phương thức trong tiêu đề hàm prototype và tiêu đề trong khai báo hàm đầy đủ. Hãy quan sát các ví dụ sau đây.

Ví dụ Tham chiếu Hằng	Ví dụ Tham chiếu
… class PhanSo	… class PhanSo

```
{
  private:
    int Tu;
    int Mau;
  public:
    //Khai báo các hàm tạo
    PhanSo Nhan(const PhanSo&);
    int GetTu(void);
    int GetMau(void);
};

PhanSo PhanSo::Nhan(const
PhanSo& p)
{
  return PhanSo(Tu*p.Tu,
Mau*p.Mau);
//không được phép viết
PhanSo(Tu*p.GetTu(),
Mau*p.GetMau());
}

int PhanSo::GetTu(void)
{
  return Tu;
}

int PhanSo::GetMau(void)
{
  return Mau;
}
...
```

```
{
  private:
    int Tu;
    int Mau;
  public:
    //Khai báo các hàm tạo
    PhanSo Nhan(PhanSo&);
    int GetTu(void);
    int GetMau(void);
};

PhanSo PhanSo::Nhan(PhanSo& p)
{
  return PhanSo(Tu*p.GetTu(),
Mau*p.GetMau());
//hoặc PhanSo(Tu*p.Tu,
Mau*p.Mau);
}

int PhanSo::GetTu(void)
{
  return Tu;
}

int PhanSo::GetMau(void)
{
  return Mau;
}
...
```

Trong trường hợp ta muốn sử dụng phương thức cho đối tượng tham chiếu hằng, thì cần khai báo phương thức GetTu và GetMau là phương thức hằng. Khi đó, chúng ta sử dụng cú pháp sau đây:

int GetTu(void) const;

int GetMau(void) const;

```
...
class PhanSo
{
  private:
    int Tu;
    int Mau;
  public:
    //Khai báo các hàm tạo
    PhanSo Nhan(const PhanSo&);
    int GetTu(void) const;
    int GetMau(void) const;
};

PhanSo PhanSo::Nhan(const PhanSo& p)
{
  return PhanSo(Tu*p.Tu, Mau*p.Mau);
  //hoặc PhanSo(Tu*p.GetTu(), Mau*p.GetMau());
}

int PhanSo::GetTu(void) const
{
  return Tu;
}

int PhanSo::GetMau(void) const
{
  return Mau;
}
...
```

Việc bổ sung từ khóa const vào sau khai báo phương thức sẽ giúp cho đối tượng tham chiếu hằng có thể gọi phương thức hằng này.

Thêm một khái niệm nữa trong C++ mà chúng ta cần biết là **phương thức tham chiếu**. Một phương thức tham chiếu cho phép ta sử dụng nó như một biến – ta có thể gán trực tiếp một giá trị biến cho phương thức đó mà không gặp phải một trở ngại nào.

Chương trình

```
#include <iostream>

using namespace std;
```

```
class complex{
private:
	float img;
	float real;
public:
	complex();
	complex(float, float);
	float &getimg();
	float &getreal();
};

complex::complex( float img, float real )
{
	this->img = img;
	this->real = real;
}

complex::complex()
{
	real = 0;
	img = 0;
}

float & complex::getreal()
{
	return real;
}

float & complex::getimg()
{
	return img;
}

int main () {
	complex c;
	c.getreal() = 2;
	c.getimg() = 1;
	cout<<c.getreal()<<" + I*"<<c.getimg();
}
```

Trong ví dụ này, chúng ta thấy các phương thức getter được khai báo là phương thức tham chiếu. Ta có thể gán trực tiếp giá trị cho các phương thức này. Trong trường hợp này, phương thức getter có cả hai tính năng:

vừa là getter vừa là setter. Nhưng với tính năng của một phương thức setter đơn (tức chỉ thiết lập một giá trị duy nhất).

Tính đóng gói – Encapsulation

Ví dụ trên đưa ra cho ta hai phương án: **nên** hay **không nên** sử dụng từ khóa const. Câu trả lời là hãy nên quy định việc sao chép hàm tạo là truyền theo tham chiếu hằng, bởi lẽ các đối tượng khác nhau, không có quyền chỉnh sửa dữ liệu thành viên của nhau, nó chỉ có thể truyền thông điệp cho nhau mà thôi, việc chỉnh sửa dữ liệu thành viên là do bản thân của đối tượng đó. Điều này là sự thể hiện tính đóng gói trong lập trình hướng đối tượng. Tính đóng gói của lập trình hướng đối tượng còn thể hiện ở các mức độ cho phép truy cập đối với dữ liệu và hàm thành viên – tương ứng với từ khoá private, protected và public mà ta đã thảo luận ở trên.

Khái niệm: tính đóng gói là tính chất không cho phép người dùng hay đối tượng khác thay đổi dữ liệu thành viên của đối tượng nội tại. Chỉ có các hàm thành viên của đối tượng đó mới có quyền thay đổi trạng thái nội tại của nó mà thôi. Các đối tượng khác muốn thay đổi thuộc tính thành viên của đối tượng nội tại, thì chúng cần truyền thông điệp cho đối tượng, và việc quyết định thay đổi hay không vẫn do đối tượng nội tại quyết định.

Chúng ta có thể khảo sát ví dụ sau: nếu một bệnh nhân cần phải thay nội tạng để có thể sống, thì việc thay thế nội tạng đó cần phải có sự đồng ý của bệnh nhân. Không ai có thể tự động thực hiện điều này (chỉ khi bệnh nhân đã rơi vào tình trạng hôn mê, thì người nhà bệnh nhân mới quyết định thay họ). Nội tạng là các thuộc tính cố hữu của bệnh nhân. Các phương thức thay thế nội tạng của đối tượng bác sĩ không phải là phương thức thành viên của đối tượng bệnh nhân (bệnh nhân không thể tự thay thế nội tạng cho mình và bác sĩ không có quyền thay thế nội tạng cho bệnh nhân nếu không có sự đồng ý của họ). Do đó, họ muốn thực hiện thì cần có phương thức đồng ý của bệnh nhân (phương thức thành viên của đối tượng bệnh nhân). Phương thức đồng ý của bệnh nhân này cũng không thể nào áp dụng cho bệnh nhân kia (bệnh nhân A không thể quyết định thay thế nội tạng cho bệnh nhân B). Như vậy, dữ liệu thành viên của đối tượng nào, thì chỉ có đối tượng đó mới có quyền thay đổi.

Trong một vài giáo trình, tính chất này còn được gọi là tính đóng gói và ẩn dấu thông tin (encapsulation and information hiding).

Con trỏ đối tượng

Chúng ta đã làm quen với mảng đối tượng và chúng ta cũng đã biết rằng có sự tương ứng 1-1 giữa mảng và con trỏ. Trong phần này, chúng ta sẽ thảo luận về con trỏ đối tượng. Chúng ta vẫn sử dụng lớp Humans ở trên cho các ví dụ minh họa trong phần này. Việc khai báo con trỏ đối tượng hoàn toàn tương tự như khai báo con trỏ dữ liệu.

```
Humans *man;
```

Để truy cập đến các phương thức thành viên bên ngoài lớp (hàm thành viên), ta sử dụng dấu **->** (vì chỉ có các phương thức thành viên được chỉ định là public). Khi gọi phương thức khởi tạo, ta có thể gọi theo cách mà ta đã sử dụng cho con trỏ dữ liệu. Hoặc có thể sử dụng toán tử new.

Chương trình | **Kết quả**

```
...

int main()

{
  Humans man("Andy", 22);
  Humans *man0 = &man;
  //Hoặc
  Humans *man1 = new Humans("Jack", 21);
  cout<<man0->getName()<<", "<<endl<<man0->getAge()<<endl;
  cout<<man1->getName()<<endl<<man1->getAge();
  return 0;
}
```

```
Andy, 22
Jack, 21
```

Ngay sau toán tử new, chúng ta gọi phương thức khởi tạo của nó. Trong ví dụ trên, ta đang khởi tạo một đối tượng duy nhất. Nếu muốn tạo một danh sách các đối tượng theo dạng con trỏ, ta có thể sử dụng toán tử new[] mà ta đã thảo luận ở trên.

Khi liên đới đến con trỏ, có nhiều vấn đề liên quan đến cách đọc. Chúng ta có thể tổng kết theo bảng bên dưới đây

Biểu thức	Cách đọc
***x**	trỏ bởi x
&x	địa chỉ của x
x.y	thành viên y của đối tượng x
x->y	thành viên y của đối tượng trỏ bởi x
(*x).y	thành viên y của đối tượng trỏ bởi x
x[i]	đối tượng thứ i trỏ bởi x

Lớp được khai báo nhờ từ khóa struct và union

Trong C++, một lớp có thể được khai báo nhờ vào từ khóa struct hoặc từ khóa union. Chúng ta đã biết từ khóa struct dùng để khai báo kiểu dữ liệu struct và nó chứa các dữ liệu thành viên. Từ khóa union dùng để khai báo kiểu dữ liệu union và cũng chứa các dữ liệu thành viên. Tuy nhiên, chúng vẫn có thể chứa các hàm thành viên. Khi khai báo lớp bằng từ khóa struct, không có một sự khác biệt nào so với từ khóa class. Chỉ có duy nhất một sự khác biệt, đó là theo mặc định, những phương thức thành viên và dữ liệu thành viên nào không được chỉ định từ khóa quy định mức truy cập (private, protected, public) thì trong lớp được khai báo bằng từ khóa class sẽ là private còn trong lớp được khai báo bằng struct sẽ là public. Còn đối với từ khóa union có vài sự khác biệt, tuy không thể dùng để khai báo một lớp hoàn hảo như từ khóa struct hay class, nhưng nó vẫn có thể chứa các phương thức bên trong nó. Nếu không chỉ định từ khóa quy định mức truy cập, thì nó sẽ mặc định là public.

Nếu viết một lớp với đầy đủ hàm tạo, hàm hủy và các phương thức khác bằng từ khóa class, thì khi thay thế bằng từ khóa struct, sẽ không có nhiều sự thay đổi. Nếu thay thế bằng từ khóa union, thì trình dịch sẽ thông báo lỗi. Sở dĩ như thế là bởi vì dù union cho phép chứa phương thức thành viên, nhưng nó không hỗ trợ khai báo prototype, không hỗ trợ dữ liệu kiểu string.

Chú ý: Hãy luôn sử dụng từ khóa class để khai báo lớp.

Con trỏ this

Con trỏ this trỏ vào dữ liệu thành viên của chính nó. Điều này có nghĩa là con trỏ this chỉ có phạm vi tác dụng trong một lớp. Một điều cực kì

quan trọng, là con trỏ this chỉ hoạt động với các dữ liệu thành viên và các hàm thành viên được khai báo là không tĩnh (non-static). Các dữ liệu thành viên và hàm thành viên tĩnh (static) không hỗ trợ con trỏ this.

Ví dụ trong phương thức hàm tạo của lớp Complex trên, chúng ta có thể sử dụng this->real để truy cập thuộc tính real, this->img – để truy cập thuộc tính img. Ta cũng có thể so sánh một đối tượng khác với đối tượng nội tại nhờ vào con trỏ this này.

Ví dụ	**Kết quả**

#include<iostream>
using namespace std;
class Complex{
  float real;
  float img;
public:
  Complex(float, float);
  bool isMe(const Complex&);
};
Complex::Complex(float real, float img){
  this->real = real;
  this->img = img;
}
bool Complex::isMe(const Complex& c){
  if(&c==this)
  return true;
  else
  return false;
}
int main(){
  Complex a(3, 2);
  Complex b(2, 2);
  Complex *c = &a;
  cout<<a.isMe(b)<<endl;
  cout<<a.isMe(*c);
  return 0;
}
	0
	1

Giải thích: với việc sử dụng con trỏ this trong hàm tạo, ta có thể đặt tên các tham số trong hàm tạo trùng với tên các dữ liệu của lớp. Để truy cập đến các thuộc tính của lớp, ta sử dụng con trỏ this. Hàm thành viên isMe sẽ kiểm tra một đối tượng có phải là chính nó hay không (có cùng địa chỉ trên bộ

nhớ). Dù là một bản sao của nó (có dữ liệu thành viên giống nhau) thì kết quả nhận được cũng là sai (0). Trong hàm main, ta khởi tạo hai đối tượng a và b. Đối tượng con trỏ c sẽ trỏ vào địa chỉ của đối tượng a. Điều này có nghĩa là c sẽ có cùng vùng địa chỉ với a, còn b thì không. Khi gọi hàm a.isMe(b) sẽ cho kết quả là sai (0) và a.isMe(*c) sẽ cho kết quả là đúng (1).

Thành viên tĩnh – Từ khóa static

Một lớp có thể chứa các thành viên tĩnh hoặc không tĩnh. Nếu không chỉ định từ khóa là static cho các thành viên, thì theo mặc định, nó sẽ là non-static. Nếu muốn quy định cho một thành viên nào là tĩnh, thì ta bổ sung từ khóa static vào trước nó. Nếu là thành viên không tĩnh, ta không cần khai báo bất kì từ khóa nào.

Một dữ liệu thành viên tĩnh của lớp như là một biến toàn cục của lớp đó. Bởi mọi sự thay đổi dữ liệu thành viên tĩnh của đối tượng này đều có tác dụng lên toàn bộ các dữ liệu thành viên tĩnh của các đối tượng khác.

Một phương thức không tĩnh có quyền truy cập đến các dữ liệu thành viên không tĩnh. Một phương thức tĩnh có thể truy cập đến dữ liệu thành viên không tĩnh. Trong trường hợp này, ta cần tạo ra một sự thể hiện của đối tượng và truy cập đến các thuộc tính không tĩnh từ đối tượng này.Để truy cập đến thành viên không tĩnh, ta sử dụng một sự thể hiện của đối tượng, sau đó là dấu chấm (hoặc ->), tiếp đến là thành viên không tĩnh.Để truy cập đến đối tượng tĩnh, ta sử dụng toán tử phạm vi ngay sau tên lớp, tiếp đến là thành viên tĩnh. Các phương thức tĩnh và không tĩnh có thể truy cập lẫn nhau.

Ví dụ	**Kết quả**

#include <iostream>
using namespace std;
class Car
{
public:
  string name;
  int serial;
public:
  static int count;
  static void Show()
  {
  Car vehicle;
	---Call by NonStatic method---
	Name: Ford
	Serial: 123
	---Call by Static method---
	Name: Ford
	Serial: 123
	
	Count: 2

```
      vehicle.name  = "Ford";
      vehicle.serial = 123;
      cout << "\nName: " << vehicle.name;
      cout << "\nSerial: " << vehicle.serial;
   }
   static void CallShowStatic(){
      Show();
   }
   void CallShowNonStatic(){
      Show();
   }
};
int Car::count = 2;
int main()
{
   Car a;
   cout<<"---Call by NonStatic method---";
   a.CallShowNonStatic();
   cout<<"\n---Call by Static method---";
   Car::CallShowStatic();
   cout << "\n\nCount: " <<Car::count;
   return 0;
}
```

Giải thích: Hàm thành viên Show là static, nên muốn truy cập đến các dữ liệu thành viên không tĩnh thì nó cần tạo một sự thể hiện của lớp đó là đối tượng vehicle. Hàm CallShowStatic là static, hàm CallShowNonStatic là non-static đều có thể truy cập đến hàm Show là static một cách trực tiếp. Trong hàm main, các hàm non-static được gọi thông qua một sự thể hiện lớp, còn hàm static được gọi thông qua toán tử phạm vi. Dữ liệu static là count cũng được truy cập thông qua toán tử phạm vi.

Mặc dù thành viên static có thể được truy cập trực tiếp thông qua toán tử phạm vi, nhưng nó cũng chịu sự chi phối của các mức truy cập (private, protected, public).

Chỉ có các thành viên không tĩnh mới có thể sử dụng con trỏ this.

Hàm bạn và lớp bạn

Hàm bạn: nếu một thành viên của lớp được quy định là private hoặc protected thì chỉ có các hàm thành viên của lớp mới có quyền truy cập đến nó. Nếu một phương thức không phải là thành viên của lớp muốn truy cập

đến, thì nó phải là hàm bạn của lớp đó. Phương thức bạn có thể được khai báo nhờ từ khóa **friend**.

Ví dụ	**Kết quả**

#include <iostream>
using namespace std;
class Rectangle
{
private:
  int w;
  int h;
public:
  Rectangle(int, int);
  friend int Area(Rectangle);
};
Rectangle::Rectangle(int w, int h){
  this->w = w;
  this->h = h;
}
int Area(Rectangle rec){
  return (rec.w*rec.h);
}
int main()
{
  Rectangle rec(2, 5);
  cout<<Area(rec);
  return 0;
}
	10

Giải thích: hàm Area là một hàm toàn cục, nó không phải là thành viên của lớp (vì không sử dụng toán tử phạm vi khi khai báo). Nếu ta cố tình truy cập đến các dữ liệu w và h thì chương trình dịch sẽ báo lỗi, bởi chúng được quy định là private. Khi ta khai báo hàm Area là hàm bạn, nó sẽ giải quyết vấn đề này.

Lớp bạn: nếu ta có hai lớp A và B, và khai báo rằng B là bạn của A, thì khi đó, các phương thức của lớp A có thể truy cập đến các thuộc tính private và protected của lớp B.

Ví dụ	**Kết quả**
#include <iostream> using namespace std;	Square: 5x5

```
class Rectangle
{
private:
   int w;
   int h;
public:
   Rectangle(int, int);
   friend class MakeSquare;
};
class MakeSquare{
private:
      int w;
      int h;
public:
   MakeSquare(Rectangle);
   void ShowSquare(void);
};
MakeSquare::MakeSquare(Rectangle rec){
   this->w = max(rec.w, rec.h);
   this->h = max(rec.w, rec.h);
}
void MakeSquare::ShowSquare(void){
   cout<<"Square: "<<w<<"x"<<h;
}
int main()
{
   Rectangle rec(2, 5);
   MakeSquare mk(rec);
   mk.ShowSquare();
   return 0;
}
```

Giải thích: Lớp Rectangle được quy định là lớp bạn của lớp Square, do đó, lớp Square có quyền truy cập đến các thuộc tính private và protected của lớp Rectangle. Hàm tạo của lớp Square truy cập đến các dữ liệu thành viên của lớp Rectangle để lấy chiều dài và chiều rộng của đối tượng rec (dù chúng là private), để tạo nên đối tượng mk. Đối tượng Square được tạo mới với cạnh của nó là số đo lớn nhất các cạnh của đối tượng Rectangle.

Ta cũng lưu ý rằng A là bạn của B, thì không có nghĩa là B cũng là bạn của A. Như vậy, tình bạn có thể là một chiều hoặc hai chiều tùy thuộc vào sự quy định của người lập trình.

Chồng chất toán tử

Trong ngôn ngữ lập trình hướng đối tượng, có nhiều ngôn ngữ hỗ trợ chồng chất toán tử (các ngôn ngữ hỗ trợ bao gồm C++, Delphi 2009, C#,VB.net, … nhưng mức độ hỗ trợ khác nhau; các ngôn ngữ không hỗ trợ bao gồm Java, Python,…). Chồng chất toán tử (operator overloading) là cách thức xây dựng các hàm thành viên mà tên gọi của chúng là các toán tử đã được định nghĩa trước đó (+, -, *, v.v.). C++ là ngôn ngữ hỗ trợ chồng chất toán tử hoàn hảo. Các toán tử sau đây có thể được chồng chất trong C++

Các toán tử được phép chồng chất
+ - * / = < > += -= *= /= << >> <<= >>= ==
!= <= >= ++ -- % & ^ ! | ~ &= ^= |= || &&
%= [] () , ->* -> new delete new[] delete[]

Cấu trúc khai báo chồng chất toán tử

type operator toán_tử(tham số){…thân hàm…}

Ví dụ sau đây sẽ minh họa cho việc chồng chất toán tử. Chúng ta sẽ xây dựng một lớp số phức, xây dựng các phép toán cộng hai số phức (phép toán hai ngôi) và phép tăng số phức lên 1 đơn vị thực và 1 đơn vị ảo(phép toán một ngôi) bằng cách sử dụng chồng chất toán tử.

Ví dụ	**Kết quả**

#include<iostream>
using namespace std;
class Complex{
  float real;
  float img;
public:
  Complex(float, float);
  Complex operator +(const Complex&);
  Complex operator ++(void);
  void toString(void);
};
Complex::Complex(float a, float b){
  real = a;
  img = b;
}
Complex Complex::operator +(const Complex& b){
  Complex c(0, 0);
  c.real = real + b.real;
	2 + 3*I
	4 + 3*I

```
    c.img = img + b.img;
    return c;
}
Complex Complex::operator ++(void){
    Complex c(0, 0);
    c.real += ++real;
    c.img += ++img;
    return c;
}
void Complex::toString(void){
    cout<<real<<" + "<<img<<"*I"<<endl;
}
int main(){
    Complex a(3, 2);
    Complex b(-1, 1);
    (a+b).toString();
    (++a).toString();
    return 0;
}
```

Ta lưu ý rằng, trong phương thức toán tử, số tham số hình thức luôn bằng hạng của toán tử trừ 1. Điều này có nghĩa là với phép toán một ngôi sẽ không có tham số hình thức, với toán tử hai ngôi sẽ có một tham số hình thức. Điều này là dễ hiểu, bởi đã có một tham số mặc định – đó chính là bản thân đối tượng nội tại (đối tượng tương ứng với con trỏ this). Phép toán cộng, sẽ cộng đối tượng nội tại với một đối tượng khác. Phép toán tăng một đơn vị thực, một đơn vị ảo sẽ làm thay đổi giá trị của đơn vị thực và đơn vị ảo của đối tượng nội tại lên 1. Vì các toán tử này trả về kiểu số phức, nên ta hoàn toàn có thể thực hiện phép toán phức hợp với chúng (tức là một biểu thức có nhiều toán tử loại này thực hiện trên các hạng tử là các số phức).

```
(a+ ++b+(a+b)).toString();
```

Bằng việc sử dụng chồng chất toán tử, biểu thức tính toán sẽ trở nên đơn giản hơn. Ta cũng có thể sử dụng cách gọi **a.operator+(b)**. Hai cách này cho kết quả như nhau. Đối với hàm toán tử + và ++ ở trên, ta có thể viết ngắn gọn hơn mà không cần khai báo thêm một biến tạm:

```
Complex Complex::operator +(const Complex& b){
    return Complex(real + b.real, img + b.img);
}
```


```
Complex Complex::operator ++(void){
   return Complex(++real, ++img);
}
```

Việc thực hiện các toán tử trên các đối tượng cần yêu cầu đối tượng trước đó phải được khởi tạo giá trị. Nghĩa là phải có một hàm tạo cho đối tượng đó. Mặc dù C++ hỗ trợ chồng chất nhiều toán tử, nhưng ta không nên lạm dụng nó. Chúng ta nên sử dụng chồng chất toán tử với mục đích đúng đắn (cộng hai số phức thì sử dụng toán tử + mà không phải là toán tử khác, ...).

Việc sử dụng chồng chất toán tử như là hàm thành viên áp dụng cho tất cả các toán tử mà C++ hỗ trợ. Trừ các toán tử gán, hợp nhất, () và ->. Các toán tử còn lại cũng áp dụng cho các hàm toàn cục. Hàm toàn cục cũng như hàm thành viên, nhưng nó không thuộc một lớp nào. Việc khai báo hàm toàn cục sẽ được thực hiện như sau

type operator@(A)

Trong đó, @ là kí hiệu toán tử, A là tên lớp.

Ví dụ	**Kết quả**

#include<iostream>
using namespace std;
class Complex{
public:
  float real;
  float img;
 	
  Complex(float, float);
  void toString(void);
};
Complex::Complex(float a, float b){
  real = a;
  img = b;
}
void Complex::toString(void){
  cout<<real<<" + "<<img<<"*I"<<endl;
}
Complex operator -(const Complex &a, const Complex &b){
  return Complex(a.real - b.real, a.img - b.img);
}
int main(){
	4 + 1*I

```
    Complex a(3, 2);
    Complex b(-1, 1);
    (a-b).toString();
    return 0;
}
```

Giải thích: trong ví dụ này, hàm toán tử - không phải là hàm thành viên của lớp Complex. Do đó, muốn truy cập đến các thuộc tính của nó, ta phải quy định các thuộc tính này là public hoặc phải tạo thêm các phương thức getter để thu thập dữ liệu hoặc quy định nó là hàm bạn. Cũng vì nó không phải là hàm thành viên của lớp Complex, nên số tham số trong phép toán một ngôi là 1, trong phép toán hai ngôi là 2.

Đối với chồng chất toán tử nhập xuất - IO overloading, chúng ta có một số chú ý:

+ Nếu khai báo hàm toán tử là một thành viên của lớp.

Ví dụ	Kết quả

```
#include <iostream>

using namespace std;

class Vector2D{
   private:
      int x, y;
   public:
      Vector2D(){
         x = 0;
         y = 0;
      }
      Vector2D(int x1, int y1){
         x = x1;
         y = y1;
      }
      ostream& operator<<(ostream& os){
         os<<"("<<x<<", "<< y<<")";
         return os;
      }
};

int main()
{
```

(4, 5)

```
   Vector2D ab(4, 5);
   //ab.operator<<(cout);
   ab<<(cout);
   return 0;
}
```

Giải thích: hàm toán tử << sẽ thực thi việc in giá trị của đối tượng nội tại (vì nó là thành viên của lớp Vector2D). Nó là toán tử đơn hạng, do đó, trong hàm toán tử không có mặt tham số Vector2D. Đối với cách sử dụng này, ta chỉ có thể gọi nó trong hàm main bằng một trong hai cách sau: ab.operator<<(cout) hoặc ab<<(cout). Cả hai cách gọi này đều không trùng khớp với toán tử xuất (hay toán tử chèn dữ liệu). Thông thường, ta sẽ xuất dữ liệu theo chuẩn cout<<dữ_liệu. Để thực hiện điều này, ta cần sư dụng cách hai.

+ Nếu khai báo hàm toán tử không phải là thành viên của lớp.

Ví dụ	Kết quả

#include <iostream>
 	
using namespace std;
 	
class Vector2D{
  private:
  int x, y;
  public:
  Vector2D(){
  x = 0;
  y = 0;
  }
  Vector2D(int x1, int y1){
  x = x1;
  y = y1;
  }
  friend ostream& operator<<(ostream&, const Vector2D&);
};
 	
ostream& operator<<(ostream& os, const Vector2D& v){
  os<<"("<<v.x<<", "<<v.y<<")";
  return os;
}
	(4, 5)

```
int main()
{
   Vector2D ab(4, 5);
   cout<<ab;
   return 0;
}
```

Giải thích: trong ví dụ này, hàm toán tử là một hàm bạn của lớp Vector2D. Nó chứa tham số Vector2D bởi nó không phải là thành viên của lớp nội tại. Khi in giá trị, nó sẽ in giá trị của Vector2D này.

Khi sử dụng toán tử nhập dữ liệu >> (hay toán tử trích tách dữ liệu), ta khai báo hoàn toàn tương tự. Kiểu dữ liệu trả về lúc này là istream& thay vì sử dụng ostream& như trên. Chúng ta tiến hành nhập dữ liệu cho nên tham số Vector2D trong hàm cũng cần thay đổi – chúng ta cần bỏ đi từ khóa const bởi lẽ ta đang tiến hành nhập dữ liệu cho nó nên không thể quy định truyền giá trị theo tham chiếu hằng (tức không cho phép thay đổi giá trị).

Ví dụ	Kết quả

#include <iostream>
 	
using namespace std;
 	
class Vector2D{
  private:
  int x, y;
  public:
  Vector2D(){
  this->x = 0;
  this->y = 0;
  }
  Vector2D(int x, int y){
  this->x = x;
  this->y = y;
  }
  friend istream& operator>>(istream&, Vector2D&);
};
 	
istream& operator>>(istream& is, Vector2D& v){
  is>>v.x>>v.y;
  return is;
}
	(4, 5)

```
int main()
{
   Vector2D ab;
   cin>>ab;
   return 0;
}
```

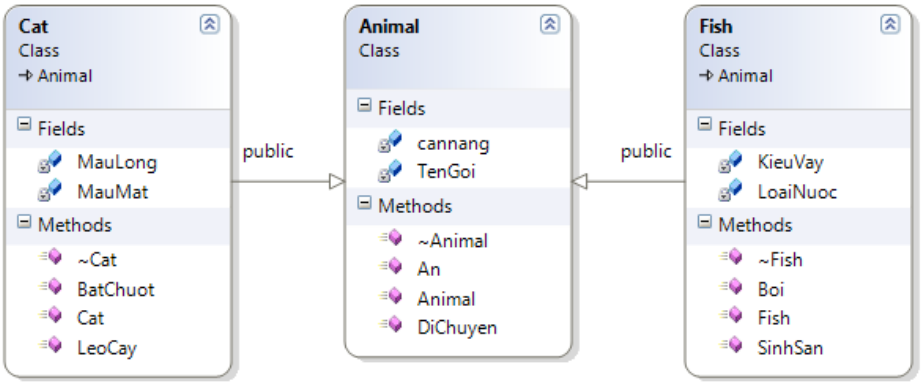
Các kiểu dữ liệu ostream& và istream& nằm trong thư viện iostream của namespace std (dấu & để quy định là truyền theo tham chiếu hoặc phương thức tham chiếu).

Tính kế thừa - Inheritance

Một tính năng theng chốt của lập trình hướng đối tượng đó là tính kế thừa. Nhờ vào tính kế thừa, nó cho phép một lớp có thể dẫn xuất từ một lớp khác, chính vì thế chúng sẽ tự động tiếp nhận các thành viên của bố mẹ và bổ sung thêm các thành viên của riêng chúng. Tính kế thừa cho phép lớp mới có thể nhận được mọi dữ liệu thành viên (private, protected, public) và các hàm thành viên (**trừ** hàm tạo, hàm hủy, hàm bạn và hàm toán tử gán =).

Ta có thể xét ví dụ về lớp động vật Animal và minh họa tính kế thừa bằng lược đồ bên dưới (Hình 13).

Lớp động vật Animal có các thuộc tính thành viên: tên gọi, cân nặng. Các hàm thành viên: di chuyển, ăn. Ta xét hai lớp dẫn xuất của nó là lớp mèo Cat và lớp cá Fish. Lớp Cat có các thuộc tính thành viên riêng: màu lông, màu mắt. Các hàm thành viên riêng: Bắt chuột, Leo cây. Lớp Fish có các thuộc tính thành viên riêng: kiểu vẩy, loại nước (nước ngọt, nước mặn, nước lợ). Các hàm thành viên: bơi, sinh sản (cách thức sinh con như thế nào).



Hình 21 – Tính kế thừa

Theo như tính thừa kế, lớp Cat và Fish không những có những thuộc tính thành viên và hàm thành viên riêng của từng lớp, mà nó còn có những thuộc tính thành viên và hàm thành viên của lớp Animal.

Từ nay, ta sẽ gọi lớp dẫn xuất Cat và Fish là các lớp con và lớp được dẫn xuất Animal là lớp cơ sở (hay lớp cha). Ta cần lưu ý rằng, tên gọi cũng mang tính tương đối, vì một lớp có thể là con của lớp này, nhưng lại là lớp cơ sở của lớp khác. Do đó, để tránh nhầm lẫn, trong những trường hợp cần phân biệt, ta sẽ gọi cụ thể là lớp con của lớp nào, hay lớp cơ sở của lớp nào.

Để quy định một lớp là dẫn xuất từ lớp khác, ta sử dụng toán tử : theo cấu trúc sau

```
class Animal{
...
};
class Cat:  Từ_khóa_mức_truy _cập   Animal{
...
};
class Fish:  Từ_khóa_mức_truy _cập   Animal{
...
};
```

Theo cấu trúc khai báo này, thì Cat và Fish là lớp con của lớp cơ sở Animal.

Ví dụ

```
#include <iostream>
using namespace std;
class Animal{
protected:
   string name;
   int weight;
public:
   Animal(void);
   Animal(string, int);
   void move(void);
   void eat(void);
};
class Cat:public Animal{
private:
```

Kết quả

```
-----Animal Object-----
I can eat
I can move
-----Cat Object-----
I can eat
I can move
I can catch mouse
I can climb tree
```

```
    string colorf;
    string colore;
public:
    Cat(string, int, string, string);
    void catchmouse(void);
    void climb(void);
};
void Animal::move(void){
    cout<<"I can move"<<endl;
}
void Animal::eat(void){
    cout<<"I can eat"<<endl;
}
Animal::Animal(){
    this->name = "";
    this->weight = 0;
}
Animal::Animal(string name, int weight){
    this->name = name;
    this->weight = weight;
}
void Cat::catchmouse(void){
    cout<<"I can catch mouse"<<endl;
}
void Cat::climb(void){
    cout<<"I can climb tree"<<endl;
}
Cat::Cat(string name, int weight, string colorfc,
string colorec){
    this->name = name;
    this->weight = weight;
    this->colorf = colorf;
    this->colore = colore;
}
int main()
{
```

```
    Animal an("Gau", 100);
    Cat ca("Cat1", 3, "black", "blue");
    cout<<"-----Animal Object-----"<<endl;
    an.eat();
    an.move();
    cout<<"-----Cat Object-----"<<endl;
    ca.eat();
    ca.move();
    ca.catchmouse();
    ca.climb();
    return 0;
}
```

Giải thích: trong chương trình này lớp Cat thừa kế từ lớp Animal. Nó sẽ kế thừa mọi dữ liệu thành viên và hàm thành viên của lớp Animal. Để hàm tạo của lớp Cat có thể truy cập đến các dữ liệu thành viên của lớp Animal, thì các dữ liệu thành viên này phải được khai báo mức truy cập là protected hoặc public. Đối tượng ca của lớp Cat chỉ có thể truy cập đến các phương thức thành viên của lớp cơ sở là Animal khi lớp Animal này được public (Cat:public Animal). Một điều cần lưu ý nữa đó là hàm tạo. Khi thừa kế, thì lớp con sẽ không thừa kế hàm tạo từ lớp cơ sở, nhưng lớp cơ sở cần có một hàm tạo mặc định không đối số (hàm tạo này luôn tồn tại; nếu ta khai báo thêm một vài hàm tạo, thì cần khai báo một hàm tạo không có đối số).

Các mức truy cập

Mức độ cho phép truy cập đến các dữ liệu thành viên từ một lớp được cho trong bảng sau

Phạm vi	public	protected	private
Thành viên của cùng một lớp	được phép	được phép	được phép
Thành viên của lớp dẫn xuất	được phép	được phép	không được phép
Còn lại	được phép	không được phép	không được phép

Chúng ta cần lưu ý rằng trong cách viết về tính kế thừa *Cat:public Animal* có một số quy tắc chuyển đổi. Nếu các thành viên của lớp cơ sở có mức truy cập là A, khi thừa kế ta quy định mức truy cập của lớp con đối với lớp cơ sở là B (A và B có thể là private < protected < public) và giả sử rằng A<B, thì

các thành viên này của lớp cha sẽ trở thành các thành viên của lớp con có mức truy cập là mức truy cập bé nhất A.

Như tôi đã giới thiệu ở trên, một biến thành viên được chỉ định từ khóa chỉ mức truy cập là private thì chỉ có các phương thức trong cùng một lớp hoặc các phương thức bạn mới có quyền truy cập (bao gồm hàm bạn và lớp bạn). Nếu mức truy cập là public, thì mọi phương thức đều có quyền truy cập đến. Chúng ta sẽ tìm hiểu kĩ hơn về từ khóa protected. Tôi đã trình bày về các khả năng mà một phương thức có thể truy cập đến một biến thành viên được khai báo là protected:

- Tương tự như các mức truy cập của private (chính nó và bạn của nó).

- Từ các phương thức của một lớp dẫn xuất.

- Từ các phương thức bạn của lớp dẫn xuất (bao gồm hàm bạn và lớp bạn).

Đối với trường hợp đầu tiên, chúng ta đã tìm hiểu nó trong phần hàm bạn và lớp bạn. Chúng ta sẽ khảo sát hai khả năng sau cùng. Đối với khả năng thứ hai, hãy quan sát ví dụ sau đây:

Ví dụ	Kết quả

#include <iostream>
using namespace std;
class Polygon{
  protected:
  int w, h;
  public:
  void SetValue(int w, int h){
  this->w = w;
  this->h = h;
  }
};
class Rectangle:public Polygon{
  public:
  int GetArea(){
  return w*h;
  }
};
 	
int main()
{
	20

```
    Rectangle rec;
    rec.SetValue(4, 5);
    cout<<rec.GetArea();
    return 0;
}
```

Giải thích: bạn lưu ý trong phương thức GetArea của đối tượng Rectangle. Nó sử dụng các biến thành viên được thừa kế từ lớp Polygon. Những biến thành viên này được khai báo là protected, do đó, nó có quyền truy cập đến.

Trong trường hợp, bạn của một lớp dẫn xuất, ta có thể quan sát ví dụ minh họa sau đây:

Ví dụ	Kết quả

#include <iostream>
using namespace std;
class Polygon{
  protected:
  int w, h;
  public:
  void SetValue(int w, int h){
  this->w = w;
  this->h = h;
  }
};
class Rectangle:public Polygon{
  public:
  int GetArea(){
  return w*h;
  }
  friend void ShowWH(Rectangle);
};
void ShowWH(Rectangle p){
  cout<<p.w<<"x"<<p.h;
}
 	
int main()
{
  Rectangle rec;
  rec.SetValue(4, 5);
  ShowWH(rec);
  return 0;
}
	4x5

Giải thích: trong trường hợp này, phương thức ShowWH là bạn của lớp dẫn xuất Rectangle, nó có quyền truy cập đến các biến thành viên được chỉ định protected.

Tính đa kế thừa – Multiple Inheritance

Trong ngôn ngữ lập trình hướng đối tượng, tính kế thừa chia làm hai loại: ngôn ngữ đơn thừa kế và ngôn ngữ đa thừa kế.

- ***Tính đơn thừa kế***: là tính chất cho phép một lớp chỉ có thể kế thừa từ một lớp cơ sở duy nhất. Nếu muốn sử dụng tính năng đa thừa kế trong ngôn ngữ lập trình loại này, ta có thể cần phải sử dụng đến khái niệm giao diện interface. Ngôn ngữ đơn thừa kế tiêu biểu gồm: Java, C#, Delphi.
- ***Tính đa thừa kế***: là tính chất cho phép một lớp có thể kế thừa từ nhiều lớp cơ sở. Ngôn ngữ đa thừa kế tiêu biểu gồm: C++.

Khai báo tính đa kế thừa trong C++ tuân theo cú pháp sau

class A: TKMTC1 B, TKMTC2 C, TKMTC3 D,...;

Trong đó,

+ TKMTC1, TKMTC2, TKMTC3 là các từ khóa chỉ mức truy cập. Chúng có thể là public, protected hoặc private.

+ Lớp A gọi là lớp con; lớp B, C, D gọi là các lớp cơ sở.

Ví dụ

```
#include <iostream>
using namespace std;
class A{
   int a;
public:
   void showA(void);
};
class B{
   int b;
public:
   void showB(void);
};
```

Kết quả

```
I'm A
I'm B
I'm C
```

```
class C: public A, public B{
   int c;
public:
   void showC(void);
};
void A::showA(void){
   cout<<"I'm A"<<endl;
}
void B::showB(void){
   cout<<"I'm B"<<endl;
}
void C::showC(void){
   cout<<"I'm C"<<endl;
}
int main()
{
   C c;
   c.showA();
   c.showB();
   c.showC();
   return 0;
}
```

Giải thích: trong ví dụ này, lớp C kế thừa từ lớp A và lớp B. Khi ta khai báo c là đối tượng của lớp C, do tính kế thừa nên đối tượng c chứa không chỉ thành viên của lớp c, mà còn có các thành viên của lớp A và B.

Tính đa hình – Polymorphism

Con trỏ trỏ vào lớp cơ sở

Một trong những tính năng theng chốt của lớp dẫn xuất là con trỏ trỏ vào lớp dẫn xuất sẽ tương thích kiểu với một con trỏ của lớp cơ sở. Đó chính là sự thể hiện của tính đa hình (cùng một lớp cơ sở, nhưng mỗi con trỏ của lớp dẫn xuất có các hình thái thể hiện khác nhau). Tính đa hình này mang lại cho kĩ thuật lập trình hướng đối tượng thêm những ưu điểm trong việc tạo dựng những tính năng đơn giản nhưng hữu dụng và linh hoạt.

Chúng ta sẽ bắt đầu viết chương trình về hình chữ nhật và tam giác. Lớp chữ nhật và tam giác kế thừa từ lớp đa giác và chúng có những phương

thức thành viên riêng. Phương thức thành viên này cùng nội dung, nhưng lại có cách thể hiện khác nhau (cùng tính diện tích, nhưng diện tích hình chữ nhật và hình tam giác có công thức tính khác nhau).

Ví dụ

```
#include <iostream>
using namespace std;
class Polygon{
   protected:
      int w, h;
   public:
      void setValue(int w, int h){
         this->w = w;
         this->h = h;
      }
};
class Rectangle:public Polygon{
   public:
      int area(){
         return w*h;
      }
};
class Triangle:public Polygon{
   public:
      int area(){
         return w*h/2;
      }
};
int main()
{
   Rectangle rec;
   Triangle tri;
   Polygon *pol1 = &rec;
   Polygon *pol2 = &tri;
   pol1->setValue(4, 5);
   pol2->setValue(4, 5);
```

Kết quả

```
Area of Rectangle: 20
Area of Triangle: 10
```

```
   cout<<"Area of Rectangle: "<<rec.area()<<endl;
   cout<<"Area of Triangle: "<<tri.area()<<endl;
   return 0;
}
```

Trong hàm main, chúng ta tạo ra hai con trỏ đối tượng của lớp Polygon là pol1 và pol2. Chúng ta truy cập tham chiếu cho rec và tri đến hai con trỏ này, vì chúng là hai đối tượng của lớp dẫn xuất từ Polygon, nên phép gán trong trường hợp này là hợp lệ.

Chỉ có một giới hạn là việc sử dụng *pol1 và *pol2 thay thế cho rec và tri là không thể (ta không thể sử dụng phương thức pol1->area() hay pol2->area()). Bởi vì phương thức area() không phải là một thành viên của Polygon. Để giải quyết vấn đề này, chúng ta sẽ sử dụng phương thức thành viên ảo.

Như chúng ta thấy trong ví dụ này, hai đối tượng tri và rec là hai đối tượng khác nhau thừa kế từ polygon. Nhưng chúng có cùng phương thức area để tính diện tích. Tuy nhiên, cách tính diện tích của hình chữ nhật và hình tam giác là hoàn toàn khác nhau.

Tính đa hình. Là tính chất thể hiện nhiều hình thái của đối tượng. Các đối tượng khác nhau có thể có cùng phương thức thực thi một hành động. Nhưng mỗi đối tượng lại thực thi hành động theo cách riêng của mình, mà không giống nhau cho tất cả các đối tượng.

Thành viên ảo

Để quy định một phương thức là ảo, chúng ta sử dụng từ khóa **virtual**. Nhờ vào phương thức ảo, ta có thể định nghĩa lại một phương thức thành viên của lớp cơ sở bên trong lớp dẫn xuất.

Ví dụ

```
#include <iostream>
using namespace std;
class Polygon{
   protected:
      int w, h;
   public:
      void setValue(int w, int h){
```

Kết quả

```
Area of Rectangle: 20
Area of Triangle: 10
Area of Polygon: 0
```

```
        this->w = w;
        this->h = h;
    }
    virtual int area(){
        return (0);
    };
};
class Rectangle:public Polygon{
  public:
    int area(){
        return w*h;
    }
};
class Triangle:public Polygon{
  public:
    int area(){
        return w*h/2;
    }
};
int main()
{
  Rectangle rec;
  Triangle tri;
  Polygon pol;
  Polygon *pol1 = &rec;
  Polygon *pol2 = &tri;
  Polygon *pol3 = &pol;
  pol1->setValue(4, 5);
  pol2->setValue(4, 5);
  pol3->setValue(4, 5);
  cout<<"Area of Rectangle: "<<pol1-
>area()<<endl;
  cout<<"Area of Triangle: "<<pol2-
>area()<<endl;
  cout<<"Area of Polygon: "<<pol3-
>area()<<endl;
```

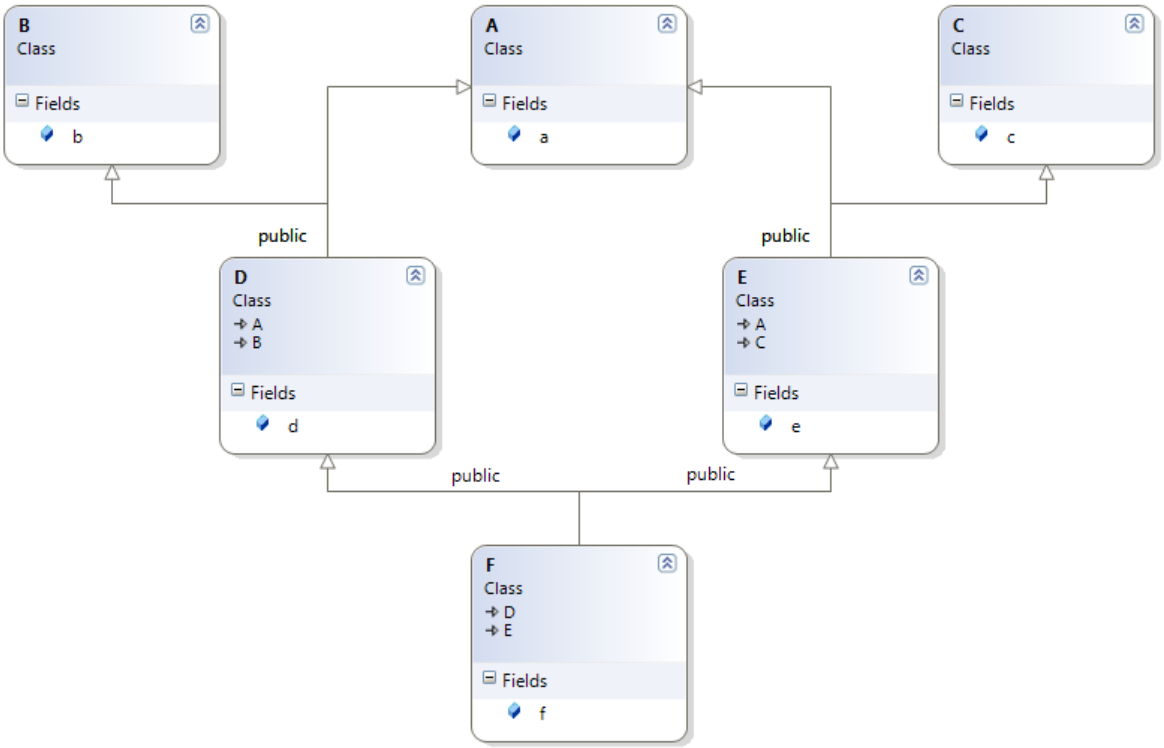

```
    return 0;
}
```

Nếu gỡ bỏ từ khóa virtual thì kết quả sẽ là 0, 0 và 0. Sở dĩ là vì trong trường hợp này area() là phương thức thành viên của lớp Polygon. Dù phương thức này đã bị quá tải trong các lớp thành viên, nhưng các đối tượng *pol1, *pol2, *pol3 là các đối tượng của lớp Polygon, nên khi gọi phương thức area() là gọi đến phương thức area() của Polygon.

Như vậy, với từ khóa virtual, các phương thức của các lớp dẫn xuất có thể thực hiện các phương thức thành viên riêng mà phương thức thành viên đó có thể trùng tên với phương thức thành viên của lớp cơ sở.

Ta cần lưu ý rằng, một số ngôn ngữ quy định virtual là mặc định cho các phương thức thành viên (nếu không khai báo từ khóa này) như Java, C#. Nhưng ngôn ngữ lập trình C++ thì không như vậy. ***Trong C++, từ khóa virtual là không mặc định cho các phương thức thành viên***.

Từ khóa virtual còn có thêm một ứng dụng trong định nghĩa ***lớp cơ sở ảo***. Chúng ta xét ví dụ sau



Hình 22 – Lớp cơ sở ảo

Trong cây thừa kế này, ta thấy rằng lớp F thừa kế từ lớp D và E. Lớp D thừa kế từ lớp A và B; lớp E thừa kế từ lớp A và C. Điều đó có nghĩa là lớp F thừa kế lớp A 2 lần. Khi ta muốn truy cập đến thuộc tính a (thừa kế từ lớp D và E) thì trình biên dịch không nhận biết được ta muốn truy cập đến giá trị a của lớp nào.

Ví dụ

```
#include<iostream>

using namespace std;

class A
{
public:
   int a;
};

class B
{
public:
   int b;
};

class C
{
public:
   int c;
};

class D:public A, public B
{
public:
   int d;
};

class E:public A, public C
```

Kết quả

error C2385: ambiguous access of 'a'

```
{
public:
   int e;
};

class F:public D, public E
{
public:
   int f;
};

int main()
{
   F f;
   f.a = 1;//Lỗi
   return 0;
}
```

Trình biên dịch sẽ thông báo lỗi "*error C2385: ambiguous access of 'a'*". Để khắc phục nhược điểm này, ta sử dụng lớp cơ sở ảo. Không có nhiều sự khác biệt trong trường hợp này. Đơn thuần chỉ bổ sung từ khóa virtual vào trước các mức thừa kế của lớp D và E (bởi hai lớp này đều thừa kế từ lớp A và được thừa kế bởi lớp F, đó là nguyên nhân gây nên lỗi thừa kế lớp A nhiều lần).

Ví dụ | **Kết quả**

```
#include<iostream>

using namespace std;

class A
{
public:
   int a;
};

class B
```

Không lỗi

```
{
public:
   int b;
};

class C
{
public:
   int c;
};

class D:virtual public A, public B
{
public:
   int d;
};

class E:virtual public A, public C
{
public:
   int e;
};

class F:public D, public E
{
public:
   int f;
};

int main()
{
   F f;
   f.a = 1;
   return 0;
}
```


Thêm một khái niệm mà ta cần quan tâm – **quá tải hàm**. Chúng ta cần lưu ý, khái niệm chồng chất hàm thành viên khác với khái niệm quá tải hàm thành viên (đôi lúc gọi là ghi đè hàm thành viên). Chồng chất hàm thành viên (**overload**) là việc quy định nhiều hàm cùng tên nhưng khác tham số, các hàm này là thành viên của cùng một lớp nội tại hoặc các hàm toàn cục. Trong khi đó, quá tải hàm thành viên (**override**) là các hàm có cấu trúc giống nhau nhưng thuộc hai lớp khác nhau và một trong số chúng thừa kế từ lớp còn lại, khi đó, ta nói rằng phương thức của lớp cha đã bị phương thức của lớp con quá tải.

Lớp cơ sở trừu tượng

Lớp cơ sở trừu tượng (abstract base class) có nhiều nét tương đồng với lớp Polygon của ví dụ trên. Chỉ có sự khác biệt là phương thức area() của lớp Polygon không thực hiện hoàn toàn chức năng của một hàm mà đơn thuần ta sẽ khai báo **virtual int area() = 0;**. Nghĩa là ta chỉ bổ sung vào giá trị 0 sau toán tử gán. Với sự khai báo dạng này, người ta gọi là phương thức ảo thuần túy (pure virtual function). Một lớp được gọi là lớp cơ sở trừu tượng, khi nó chứa ít nhất một phương thức thành viên ảo thuần túy.

Sự khác biệt căn bản giữa lớp cơ sở trừu tượng và lớp đa hình là lớp cơ sở trừu tượng không thể tạo ra một sự thể hiện cho nó. Nghĩa là ta không thể viết **Polygon pol;** như trong ví dụ trên. Ta chỉ có thể sử dụng một con trỏ, để trỏ đến nó và sử dụng các tính năng đa hình của nó mà thôi.

Ví dụ

```
#include <iostream>
using namespace std;
class Polygon{
   protected:
      int w, h;
   public:
      void setValue(int w, int h){
         this->w = w;
         this->h = h;
      }
      virtual int area() = 0;
};
```

Kết quả

```
Area of Rectangle: 20
Area of Triangle: 10
```

```
class Rectangle:public Polygon{
   public:
      int area(){
         return w*h;
      }
};
class Triangle:public Polygon{
   public:
      int area(){
         return w*h/2;
      }
};
int main()
{
   Rectangle rec;
   Triangle tri;
/* Polygon pol; → Phát sinh lỗi vì Polygon là một lớp trừu tượng, ta không thể tạo một thể hiện cho lớp trừu tượng */
   Polygon *pol1 = &rec;
   Polygon *pol2 = &tri;
   pol1->setValue(4, 5);
   pol2->setValue(4, 5);
   cout<<"Area of Rectangle: "<<pol1->area()<<endl;
   cout<<"Area of Triangle: "<<pol2->area()<<endl;
   return 0;
}
```

Trong ví dụ trên, ta đã sử dụng một phương thức ảo thuần túy đơn giản. Khi khai báo một phương thức ảo thuần túy, ta cần chú ý:

- Tham số của nó nếu rỗng thì ta quy định là void. Nếu nó là một kiểu tham chiếu (như các đối tượng của lớp) thì ta nên quy định là con trỏ void* và sau này, khi cần sử dụng đến nó, ta có thể chuyển đổi con trỏ void sang con trỏ đối tượng.
- Giá trị trả về trong phương thức ảo thuần túy NÊN quy định là 0.

Ví dụ sau đây sẽ cho ta thấy cách sử dụng lớp trừu tượng trong trường hợp tham số bên trong phương thức ảo thuần túy là một đối tượng. Ta có một lớp trừu tượng Point (lớp điểm) và hai lớp Point2D và Point3D thừa kế thừa Point. Lớp Point là một lớp trừu tượng chứa phương thức ảo thuần túy là KhoangCach. Các lớp Point2D và Point3D sẽ quá tải phương thức ảo thuần túy này.

Ví dụ

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

using namespace std;

class Point
{
protected:
	float x, y;
public:
	float& getX(void);
	float& getY(void);
	virtual float KhoangCach(void*)=0;
};

float& Point::getX(void)
{
	return x;
}

float& Point::getY(void)
{
	return y;
}

class Point2D:public Point
{
public:
	float KhoangCach(void*);
};

class Point3D:public Point
{
```

Kết quả

```
.::-----Nhap toa do cho hai Point2D u va v-----::.
+ Toa do u:
1
1
+ Toa do v:
1
2
D(u,v) = 1
.::-----Nhap toa do cho hai Point3D u va v-----::.
+ Toa do u:
1
1
1
+ Toa do v:
1
1
2
D(u3,v3) = 1
```

```
	float z;
public:
	float& getZ(void);
	float KhoangCach(void*);
};

float Point2D::KhoangCach(void* p)
{
	Point2D* q = (Point2D*)(p);
	return sqrt(pow(q->x-x,2)+pow(q->y-y,2));
}

float& Point3D::getZ(void)
{
	return z;
}

float Point3D::KhoangCach(void* p)
{
	Point3D* q = (Point3D*)(p);
	return sqrt(pow(q->x-x,2)+pow(q->y-
y,2)+pow(q->z-z,2));
}

int main()
{
	Point2D v2;
	Point2D u2;
	cout<<".::-----Nhap toa do cho hai Point2D u
va v-----::."<<endl;
	cout<<"+ Toa do u:"<<endl;
	cin>>v2.getX();
	cin>>v2.getY();
	cout<<"+ Toa do v:"<<endl;
	cin>>u2.getX();
	cin>>u2.getY();
	Point* u = &v2;
	Point* v = &u2;
	cout<<"D(u,v) = "<<u->KhoangCach(v);
	Point3D v3;
	Point3D u3;
	cout<<endl<<".::-----Nhap toa do cho hai
```


```
Point3D u va v-----::."<<endl;
        cout<<"+ Toa do u:"<<endl;
        cin>>v3.getX();
        cin>>v3.getY();
        cin>>v3.getZ();
        cout<<"+ Toa do v:"<<endl;
        cin>>u3.getX();
        cin>>u3.getY();
        cin>>u3.getZ();
        Point* p = &v3;
        Point* q = &u3;
        cout<<"D(u3,v3) = "<<p->KhoangCach(q);
        cout<<endl;
        return 0;
}
```

Giải thích: chúng ta cần chú ý rằng, nếu khai báo một lớp trừu tượng (chứa phương thức ảo thuần túy) thì lớp thừa kế từ nó nên có một phương thức có khai báo hoàn toàn trùng khớp với phương thức ảo thuần túy này (về tên gọi, danh sách tham số; chỉ trừ phép gán = 0 và từ khóa virtual sẽ không xuất hiện trong phương thức quá tải của lớp con). Nếu ta khai báo không trùng khớp, thì lớp con sẽ có phương thức ảo thuần túy này, và hiển nhiên nó cũng sẽ là lớp trìu tượng.

Trong ví dụ này, khoảng cách có thể là giữa hai điểm Point2D hoặc hai điểm Point3D. Do đó, trong lớp trừu tượng, ta sẽ sử dụng void*. Trong lớp con, ta sẽ chuyển đổi từ void* sang Point2D* hoặc Point3D* tùy thuộc vào lớp Point2D hay Point3D.

Tính trừu tượng hóa - Abstraction

Tính trừu tượng hóa là tính chất chỉ tập trung vào những phần cốt lõi của đối tượng, bỏ qua những tiểu tiết không cần thiết. Nó còn thể hiện ở lớp trừu tượng cơ sở: lớp trừu tượng cơ sở chứa các đặc tính chung, tổng quát cho một nhóm đối tượng. Khi đó, nhóm đối tượng sẽ thừa kế từ lớp trừu tượng này để nhận các thuộc tính chung, đồng thời bổ sung thêm tính năng mới.

Khi phân tích một nhóm đối tượng, ta thường tìm ra các điểm chung và đặc trưng cho các đối tượng, rồi từ đó xây dựng nên một lớp trừu tượng cơ sở để chứa các phương thức tác động đến các các đặc trưng chung đó. Mỗi một đối tượng trong nhóm đối tượng trên khi thừa kế từ lớp trừu tượng cơ sở

sẽ có phương thức đặc trưng cho nhóm đối tượng này. Tính trừu tượng cũng là một đặc trưng của ngôn ngữ lập trình hướng đối tượng.

Hàm mẫu – Template Function

Trong nhiều trường hợp chúng ta cần xây dựng hàm mà kiểu dữ liệu của các tham số và kiểu dữ liệu của hàm trả về là không tường minh (có nghĩa là chúng có dạng tổng quát, có thể là số nguyên, số nguyên dài, xâu…). Khi đó, để giải quyết vấn đề này, chúng ta có thể sử dụng quá tải hàm như trên, con trỏ hàm và thêm một cách thức nữa, đó là hàm mẫu – template function. Để khai báo kiểu dữ liệu không tường minh này, chúng ta có thể khai báo như sau:

```
template <class T>
```

Ví dụ sau đây minh họa việc sử dụng hàm mẫu template function.

Ví dụ	**Kết quả**

#include <iostream>
using namespace std;
template <class T>
T add(T a, T b)
{
  return a+b;
}
int main()
{
  cout<<add<int>(1, 2)<<endl;
  cout<<add<string>(“ab”, “cd”)<<endl;
}
	3
	abcd

Giải thích: khai báo hàm mẫu template function được dùng để tạo một kiểu dữ liệu không tường minh. Việc xây dựng hàm add với kiểu dữ liệu trả về là kiểu T và các tham số của nó cũng là kiểu T. Do đó, hàm add này có thể sử dụng dưới dạng tổng quát, dùng để cộng số, cộng xâu… Để quy định là cộng số, ta có thể quy định ngay sau tên hàm: ***tên_hàm<kiểu_dữ_liệu>***. Như vậy, trong ví dụ trên, đối tượng cout thứ nhất in ra giá trị tổng của hai số nguyên 1 và 2. Đối tượng cout thứ hai in ra giá trị ghép nối của hai xâu “ab” và “cd”.

Lớp mẫu – Template class

Tương tự hàm template, ta cũng có khai báo lớp template.

```
template <class T>
```

```
class mypair{
  T values[2];
  public:
    mypair(T first, T second)
    {
      values[0]=first;
      values[1]=second;
    }
};
int main()
{
    mypair<int>x(2, 3);
    mypair<float>f(2.3, 3.3);
    return 0;
}
```

Ta có thể sử hàm tạo với tham số first và second thuộc loại T bằng các tham số cụ thể được ấn định. Ví dụ mypair<int> x(1, 2) sẽ sử dụng hàm tạo của lớp mypair để tạo đối tượng chứa hai thuộc tính là 1 và 2 thuộc số nguyên. Tương tự cho trường hợp mypair<float>f(2.3, 3.3).

Khi cần sử dụng nhiều kiểu dữ liệu không tường minh, ta có thể bổ sung thêm trong khai báo template:

template<class T, class U, class V,...>

Và khi triệu gọi phương thức, ta chỉ đơn thuần chỉ đích danh kiểu dữ liệu tương ứng:

myclass<kiểu_dữ_liệu_1, kiểu_dữ_liệu_2, kiểu_dữ_liệu_3,..> đối_tượng;

Lớp template được sử dụng rộng rãi trong lập trình C++ bởi có nhiều lớp có chung các phương thức và dữ liệu nhưng chúng chỉ khác nhau kiểu dữ liệu cho các biến thành viên. Ví dụ như trong lớp số phức: các dữ liệu thành viên là phần thực và phần ảo có thể là số nguyên, số thực. Ví dụ sau đây sẽ xây dựng lớp số phức theo hướng tiếp cận template class.

Ví dụ	Kết quả
#include <iostream> using namespace std; template <class T>	real=1 image=2 real=3 image=4 4+I*6

```
class complex{
private:
	T real;
	T imag;
public:
	complex(void);
	complex(T, T);
	complex(const complex<T>&);
	complex operator+(const complex<T>&);
	template <class T, class charT, class traits> friend
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>&, const
complex<T>&);
	template <class T, class charT, class traits> friend
basic_istream<charT, traits>& operator>>(basic_istream<charT,
traits>&, complex<T>&);
};

template <class T>
complex<T>::complex(void)
{
	real = 0;
	imag = 0;
}

template <class T>
complex<T>::complex(T real, T imag)
{
	this->real = real;
	this->imag = imag;
}

template <class T>
complex<T>::complex(const complex<T>& c)
{
	this->real = c.real;
	this->imag = c.imag;
}

template <class T>
complex<T> complex<T>::operator+(const complex<T>& c)
{
```


```
        return complex(real+c.real, imag+c.imag);
}

template <class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const
complex<T>& c)
{
        os<<c.real<<"+I*"<<c.imag;
        return os;
}

template <class T, class charT, class traits>
basic_istream<charT, traits>& operator>>(basic_istream<charT,
traits>& is, complex<T>& c)
{
        cout<<"real=";
        is>>c.real;
        cout<<"image=";
        is>>c.imag;
        return is;
}

int main () {
        complex<int> c;
        complex<int> d;
        cin>>c>>d;
        complex<int> x = c + d;
        cout<<x;
        return 0;
}
```

Giải thích: trong ví dụ này, không có nhiều sự khác biệt so với lớp số phức thông thường. Chúng ta chỉ đơn thuần bổ sung khai báo template<class T> vào bên trước khai báo lớp và khai báo phương thức có sử dụng kiểu dữ liệu dùng chung T. Riêng đối với phương thức nhập xuất dữ liệu, cần lưu ý: ta không thể sử dụng kiểu dữ liệu thông thường là ostream& và istream&, mà thay vào đó là basic_ostream<?, ?> và basic_istream<?,?>. Đây cũng là hai lớp template với hai đối số. Điều này giải thích tại sao trước các toán tử nhập/xuất lại bổ sung thêm khai báo template<class T, class charT, class traits>. Chúng ta cũng cần chú ý thêm một điểm, tất cả các từ complex trong ví dụ trên được phân thành hai nhóm: nhóm kiểu dữ liệu complex và nhóm

tên hàm complex. Tên hàm tạo complex – ta không bổ sung kiểu dữ liệu chỉ định T – chỉ đơn thuần là complex. Đối với các từ complex còn lại, nhất thiết phải chỉ định kiểu dữ liệu T – complex<T>. Định nghĩa class template chỉ có phạm vi tác dụng trong lớp hoặc hàm được chỉ định ngay sau nó. Nó không phải là một khai báo có thể dùng chung. Điều này giải thích vì sao ta lại sử dụng quá nhiều khai báo class template như trên.

Tổng kết về lập trình hướng đối tượng

Cơ sở của lập trình hướng đối tượng đó là các **đối tượng** và **lớp**. Đối tượng là một thực thể trong thế giới thực. Lớp mô tả đối tượng bằng ngôn ngữ lập trình. Ngược lại, một đối tượng là một sự thể hiện (instance) của lớp.

Ngôn ngữ lập trình hướng đối tượng có 4 đặc trưng cơ bản:

- **Tính trừu tượng hóa dữ liệu** (abstraction): là khía cạnh mà các ngôn ngữ không quan tâm đến những tiểu tiết của đối tượng. Nó chỉ tập trung vào những thứ cốt lõi của đối tượng. Tính trừu tượng còn thể hiện ở khái niệm lớp trừu tượng cơ sở.

- **Tính đóng gói** (encapsulation) hay **tính đóng gói và che dấu thông tin** (encapsulation and information hiding): là tính chất chỉ mức độ chia sẻ thông tin. Một đối tượng khác không có quyền truy cập và làm thay đổi thông tin của đối tượng nội tại, chỉ có đối tượng nội tại mới quyết định có nên thay đổi thông tin của mình hay không.

- **Tính đa hình** (polymorphism): là nhiều hình thái. Các đối tượng khác nhau có thể có cùng phương thức thực thi hành động, nhưng cách thức thực thi hành động đó có thể khác nhau.

- **Tính thừa kế** (hay kế thừa - inheritance): là tính chất cho phép các đối tượng tiếp nhận dữ liệu từ một đối tượng khác mà nó thừa kế. Nhờ vào tính thừa kế, các đối tượng có thể sử dụng các tính năng của đối tượng khác. Tính thừa kế chia làm hai loại: đơn thừa kế và đa thừa kế. Đơn thừa kế là tính chất chỉ cho phép một lớp thừa kế từ một lớp cơ sở; còn tính đa thừa kế cho phép một lớp kế thừa từ nhiều lớp cơ sở. C++ là ngôn ngữ lập trình hỗ trợ đa thừa kế.

CHƯƠNG 14. NAMESPACE

Từ khóa namespace

Nhờ vào namespace, ta có thể nhóm các thực thể như lớp, đối tượng và các hàm dưới một tên gọi tương ứng với từ khóa namespace. Theo cách này, các phạm vi toàn cục lại được chia nhỏ ra thành các phạm vi toàn cục con, mà mỗi một phạm vi toàn cục con này có một tên gọi riêng.

Để khai báo namespace, ta sử dụng từ khóa namespace theo cú pháp sau

```
namespace tên_của_namespace
{
  các_thực_thể
}
```

Để truy cập đến các thực thể của namespace, ta sử dụng toán tử phạm vi ::

Ví dụ	Kết quả

#include <iostream>
using namespace std;
namespace first
{
  int var = 5;
}
namespace second
{
  double var = 3.14;
}
int main()
{
  cout<<first::var<<endl;
  cout<<second::var<<endl;
  return 0;
}
	5
	3.14

Như ta thấy, trong ví dụ này có hai biến toàn cục là var (lưu ý, hai biến var này đều là biến toàn cục vì phạm vi hoạt động của nó là toàn bộ chương trình). Dù là trùng tên, nhưng do chúng thuộc vào các namespace khác nhau, nên khai báo này là hợp lệ.

Từ khóa using

Từ khóa using được sử dụng để đưa một tên gọi từ namespace sang vùng khai báo hiện tại. Khi sử dụng ***using namespace tên_namespace***, chúng ta không cần sử dụng tên_namespace khi gọi đến thực thể của nó.

Ví dụ	Kết quả

#include <iostream>
using namespace std;
namespace first
{
  int x = 5;
  int y = 5;
}
namespace second
{
  double x = 1.60;
  double y = 3.14;
}
int main()
{
  using namespace first;
  cout<<x <<endl;
  cout<<y<<endl;
  return 0;
}
	5
	5

Trong trường hợp này, khi sử dụng namespace first, chúng ta có thể sử dụng các biến trong namespace của nó mà không cần gọi đến tên của namespace. Nhưng cũng lưu ý rằng, nếu using cả hai namespace thì trong trường hợp này, ta không thể truy cập đến các biến x và y theo cách này (vì chúng trùng tên), mà chỉ có thể truy cập theo cách sử dụng toán tử phạm vi ::

Ta cũng có thể sử dụng từ khóa using như sau

Ví dụ	Kết quả

#include <iostream>
using namespace std;
namespace first
{
  int x = 5;
  int y = 5;
	5
	3.14

}
namespace second
{
  double x = 1.60;
  double y = 3.14;
}
int main()
{
  using first::x;
  using second::y;
  cout<<x <<endl;
  cout<<y<<endl;
  return 0;
}	

Phạm vi của namespace

Một namespace được khai báo sử dụng bằng từ khóa using chỉ có tác dụng trong phạm vi mà nó được khai báo. Điều đó có nghĩa là nếu ta **using namespace tên_namespace**, thì nó chỉ có tác dụng trong khối lệnh mà ta khai báo.

Ví dụ	**Kết quả**

#include <iostream>
using namespace std;
namespace first
{
  int x = 5;
  int y = 5;
}
namespace second
{
  double x = 1.60;
  double y = 3.14;
}
int main()
{
  {
  using namespace first;
  cout<<x<<endl;
  cout<<y<<endl;
  }
  {
	5
	5
	1.6
	3.14

```
    using namespace second;
    cout<<x<<endl;
    cout<<y<<endl;
  }
  return 0;
}
```

Tái định danh cho namespace

Nếu muốn khai báo một tên mới cho namespace đã tồn tại, hãy sử dụng cú pháp

```
namespace tên_mới_của_namespace = tên_đã_khai_báo_của_name_space;
```

Namespace std

Tất cả các tệp trong thư viện chuẩn của C++ đều được khai báo như là thực thể của namespace std. Điều đó giải thích tại sao khi viết một chương trình trong C++ có sử dụng các hàm nhập xuất cơ bản, các kiểu dữ liệu như string… thì ta phải sử dụng khai báo **using namespace std**.

CHƯƠNG 15. NGOẠI LỆ

Các ngoại lệ là cách thức giúp chúng ta tác động ngược trở lại với các tình huống sinh ra ngoại lệ đó.

Để nắm bắt được ngoại lệ, chúng ta sử dụng cú pháp try…catch hoặc throw.

Mệnh đề try…catch

Nếu viết một chương trình có khả năng nảy sinh ngoại lệ, chúng ta cần đặt nó vào trong khối lệnh của từ khóa try, nếu ngoại lệ phát sinh, hành động xử lý sẽ được đặt trong khối lệnh của từ khóa catch.

Ví dụ	**Kết quả**

#include <iostream>
using namespace std;
int main()
{
  int a;
  try{
  throw 20;
  }catch(int e){
  cout<<e;
  }
  return 0;
}
	20

Giải thích: Trong chương trình này, lệnh throw đang cố vượt qua một ngoại lệ tương ứng với mã 20, nhưng ngoại lệ này bị nắm bắt bởi câu lệnh try…catch. Do ngoại lệ phát sinh, nên lệnh trong mệnh đề catch được thực thi.

Nếu có nhiều ngoại lệ phát sinh, ta có thể sử dụng cấu trúc đa tầng của mệnh đề catch: **try{…}catch(…){…}catch(…){…}…**

Mệnh đề throw

Khi khai báo một hàm, nếu trong hàm đó có khả năng phát sinh ngoại lệ, chúng ta có thể chỉ định từ khóa throw cho nó

type tên_hàm(danh_sách_tham_số) throw (int)

Nếu chỉ có throw() – nghĩa là không chỉ định loại dữ liệu trong throw – thì hàm sẽ cho phép vượt qua mọi ngoại lệ. Nếu hàm không có mệnh đề throw thì sẽ không được phép vượt qua ngoại lệ.

Thư viện chuẩn exception

Thư viện chuẩn của C++ cung cấp cho chúng ta một thư viện để quản lý các ngoại lệ đó là exception. Nó nằm trong namespace std. Lớp này có một hàm tạo mặc định, một hàm tạo sao chép, các toán tử, hàm hủy và một hàm thành viên ảo what(). Hàm này trả về con trỏ kí tự phát sinh ngoại lệ. Nó có thể được quá tải trong lớp dẫn xuất.

Ví dụ	Kết quả

#include <iostream>
#include <exception>
 	
using namespace std;
 	
class myexception:public exception{
virtual const char* what() const throw(){
return “Co ngoai le !”;
}
};
int main(){
try{
myexception myex;
throw myex;
}catch(exception& e){
cout<<e.what();
}
return 0;
}
	Co ngoai le !

Mỗi khi có ngoại lệ xảy ra, mệnh đề catch sẽ được thực hiện, và ngoại lệ sẽ được nắm bắt, kết quả in ra luôn là câu thông báo “Co ngoai le”.

Khi ta xây dựng một ứng dụng có khả năng phát sinh ngoại lệ, ta cần quy định lớp đối tượng được xây dựng phải thừa kế từ lớp exception này. Điều này giúp chúng ta có thể nắm bắt ngoại lệ và bỏ qua chúng bởi trong một số trường hợp các ngoại lệ này có thể gây ra các lỗi không mong muốn cho ứng dụng. Ví dụ sau đây giúp ứng dụng của chúng ta vượt qua phép chia cho 0 và tiếp tục thực thi chương trình.

Ví dụ	Kết quả

#include<iostream>
 	
using namespace std;
 	
class MyNumber:public exception
{
  private:
  float x, y;
  public:
  virtual const char* what() const throw()
  {
 	
  return "Chia cho 0";
  }
  MyNumber(float, float);
  float DivMe(void);
};
 	
MyNumber::MyNumber(float x, float y)
{
  this->x = x;
  this->y = y;
}
 	
float MyNumber::DivMe(void)
{
  x/y;
}
 	
int main()
{
  MyNumber m(1, 0);
  try
  {
  m.DivMe();
  throw m;
  }
  catch(exception& e)
  {
  cout<<e.what();
  }
	Chia cho 0

```
        return 0;
}
```

Giải thích: trong ví dụ này, lớp khai báo có khả năng phát sinh ngoại lệ vì nó thực hiện phép chia (ngoại lệ tương ứng là chia cho 0). Điều này giải thích vì sao ta cần cho nó thừa kế từ lớp exception. Mỗi khi giá trị nhập vào cho biến thành viên y là 0, thì ngoại lệ "Chia cho 0" sẽ phát sinh. Khi ngoại lệ phát sinh, ta sử dụng cú pháp try để bao bọc quanh vùng lệnh phát sinh ra ngoại lệ (cụ thể là *m.DivMe()*). Khi ngoại lệ phát sinh, mệnh đề catch sẽ thực thi và in ra lỗi tương ứng với hàm thành viên what – tức là "Chia cho 0".

CHƯƠNG 16. LÀM VIỆC VỚI FILE

C++ cung cấp cho ta các lớp sau đây để làm việc với file

- ofstream: lớp ghi dữ liệu ra file.
- ifstream: lớp đọc dữ liệu từ file.
- fstream: lớp để đọc/ghi dữ liệu từ/lên file.

Các lớp này là dẫn xuất trực tiếp hoặc gián tiến từ lớp istream và ostream. Chúng ta sử dụng đối tượng cin là một thể hiện của lớp istream và cout là một thể hiện của lớp ostream. Chúng ta cũng có thể sử dụng các đối tượng của lớp ofstream, ifstream hoặc fstream để làm việc trực tiếp với file. Ví dụ sau đây sẽ cho thấy điều này

Ví dụ

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
	ofstream myfile;
	myfile.open("example.txt");
	myfile.<<"Ghi du lieu ra file";
	myfile.close();
	return 0;
}
```

Ví dụ này chỉ đơn thuần ghi câu "Ghi du lieu ra file" lên tệp example.txt.

Chúng ta sẽ nghiên cứu từng bước khi làm việc với các đối tượng của ba lớp mà chúng ta nêu ở trên..

Mở file

Để mở file trong chương trình bằng một đối tượng stream, chúng ta sử dụng hàm thành viên **open(tên_file, chế_độ_mở)**.

Trong đó,

- tên_file: là tên của file mà chúng ta cần mở. Ta cần đảm bảo cung cấp đường dẫn chính xác đến tập tin này. Ta cũng cần lưu ý đường dẫn đến tập tin. Đường dẫn có thể là đường dẫn tuyệt đối hoặc tương đối. Nếu cung cấp đường dẫn tương đối, ta cần tuân thủ nguyên tắc như khi làm việc với tệp .cpp và .h như tôi đã trình bày ở trên.

- chế_độ_mở: là tham số tùy chọn, thường trong C++ nó có thể là các cờ hiệu sau đây:

Cờ hiệu	Giải thích
ios::in	Mở file để đọc.
ios::out	Mở file để ghi.
ios::binary	Mở file ở chế độ nhị phân (thường áp dụng cho các file mã hóa).
ios::ate	Thiết lập vị trí khởi tạo tại vị trí cuối cùng của file. Nếu cờ hiệu này không thiết lập bất kì giá trị nào, vị trí khởi tạo sẽ đặt ở đầu file.
ios::app	Mọi dữ liệu được ghi ra file sẽ tiến hành bổ sung vào cuối file (không ghi đè lên file). Cờ hiệu này chỉ có thể sử dụng trong tác vụ mở file để ghi.
ios::trunc	Nếu một file được mở để ghi đã tồn tại, nó sẽ ghi đè lên nội dung cũ.

Các cờ hiệu này có thể được kết hợp bằng cách sử dụng toán tử dịch bit OR (|). Ví dụ, tôi muốn mở một file nhị phân example.bin để ghi dữ liệu và bổ sung dữ liệu ở cuối file này, tôi có thể viết như sau

```
ofstream myfile;
myfile.open("example.bin", ios::out|ios::app|ios::binary);
```

Thành viên open của các lớp ofstream, ifstream và fstream có tham số chế_độ_mở mặc định (trong trường hợp tham số này không được chỉ định) được đưa ra trong bảng sau:

Lớp	chế_độ_mở mặc định
ofstream	ios::out
ifstream	ios::in
fstream	ios::in|ios::out

Nếu tham số được ấn định một giá trị cụ thể, thì tham số được sử dụng sẽ ghi đè lên tham số mặc định mà không phải là kết hợp với tham số mặc định. Ví dụ, nếu sử dụng ofstream để mở file với tham số chế_độ_mở được

quy định là ios::binary, thì tham số mở sẽ là ios::binary mà không phải là ios::out|ios::binary.

Nếu sử dụng hàm khởi tạo cho các lớp này, thì phương thức thành viên open sẽ tự động được triệu gọi. Nghĩa là ta có thể viết

```
ofstream myfile(“example.bin”, ios::out|ios::app, ios::binary);
```

thay cho cách viết ở trên.

Để kiểm tra một file đã mở thành công hay chưa, chúng ta có thể sử dụng phương thức is_open. Nếu đã mở thành công, nó sẽ trả về giá trị true và ngược lại, nếu mở không thành công, nó sẽ trả về giá trị false.

Đóng file

Khi chúng ta hoàn tất công việc với một file, chúng ta cần thực hiện thao tác đóng file lại. Tác vụ này là bắt buộc nếu ta đã hoàn tất các tác vụ trên file. Khi đó, ta chỉ đơn thuần triệu gọi phương thức thành viên close

```
myfile.close();
```

Nếu phương thức hủy của đối tượng được triệu gọi, phương thức close sẽ tự động được gọi theo.

File văn bản

Đối với một file văn bản, thì cờ hiệu ios::binary sẽ không bao giờ được sử dụng. Những file văn bản chỉ đơn thuần chứa văn bản. Để đọc ghi dữ liệu trên file này ta sử dụng toán tử xuất – nhập dữ liệu (<< và >>).

Ghi dữ liệu lên file văn bản

```
#include <iostream>
#include<fstream>

using namespace std;

int main(){
        ofstream myfile (“example.txt”);
        if (myfile.is_open(){
                myfile<<”Dong 1 da ghi\n”;
                myfile<<”Dong 2 da ghi\n”;
                myfile.close();
        }
        else cout<<”Khong the ghi du lieu len file”;
```

```
        return 0;
}
```

Ví dụ trên cho thấy việc ghi dữ liệu lên file văn bản nhờ vào toán tử <<. Ví dụ tiếp theo sau đây sẽ minh họa cho việc đọc dữ liệu từ file văn bản bằng toán tử >>.

Đọc dữ liệu từ file văn bản

```
#include <iostream>
#include<fstream>
#include<string>

using namespace std;

int main(){
        ifstream myfile ("example.txt");
        if (myfile.is_open(){
                        while(!myfile.eof()){
                                getline(myfile, line);
                                cout<<line<<endl;
                        }
                myfile.close();
        }
        else cout<<"Khong the ghi du lieu len file";
        return 0;
}
```

Trong ví dụ này, chúng ta có sử dụng hàm thành viên eof của đối tượng ifstream. Hàm thành viên này có chức năng kiểm tra vị trí đọc đã là vị trí cuối cùng của file hay chưa, nếu chưa, dữ liệu từ file sẽ tiếp tục được đọc. Ngược lại, nó sẽ dừng việc đọc dữ liệu.

Kiểm tra trạng thái của các cờ hiệu

Ví dụ trên cho ta một các thức để kiểm tra trạng thái của các cờ hiệu. Bảng sau đây sẽ liệt kê các trạng thái cờ hiệu có thể được sử dụng trong C++.

Trạng thái	Giải thích
bad()	Nếu tác vụ đọc/ghi file bị lỗi, nó sẽ trả về giá trị true; ngược lại, nó sẽ trả về giá trị false.
fail()	Trả về giá trị true trong trường hợp như bad(), nhưng nếu gặp lỗi về định dạng, nó cũng trả về giá trị true (ví dụ đọc số từ một file văn bản).
eof()	Trả về giá trị true nếu file đã được đọc đến vị trí cuối

	cùng của file, ngược lại, trả về giá trị false.
good()	Nó sẽ trả về giá trị true nếu bad(), fail() và eof() không phát sinh lỗi.

Để thiết lập lại các mức kiểm tra trạng thái cờ hiệu, ta sử dụng phương thức thành viên clear().

Con trỏ get và put

Mọi đối tượng luồng xuất nhập đều có ít nhất một con trỏ luồng:

- Luồng ifstream có con trỏ istream mà ta gọi là con trỏ get để trỏ vào phần tử có thể đọc dữ liệu.

- Luồng ofstream có con trỏ ostream mà ta gọi là con trỏ put để trỏ vào phần tử có thể ghi dữ liệu.

- Luồng fstream có cả hai con trỏ get và put để đọc và ghi dữ liệu.

Những con trỏ luồng nội tại này trỏ vào vị trí đọc và ghi với luồng có thể sử dụng các hàm thành viên sau đây:

Hàm thành viên tellg() và tellp()

Hai hàm thành viên này không có tham số và trả về giá trị của một kiểu dữ liệu dạng pos_type. Kiểu dữ liệu này bản chất là một số nguyên integer. Nó mô tả vị trí hiện tại của của con trỏ luồng get và con trỏ luồng put.

Hàm thành viên seekg() và seekp()

Những hàm thành viên này cho phép chúng ta thay đổi vị trí hiện tại của con trỏ luồng get và put. Cả hai hàm này được chồng chất với hai prototype khác nhau. Prototype thứ nhất:

seekg(vị_trí);

seekp(vị_trí);

Việc sử dụng các prototype này giúp làm thay đổi vị trí tuyệt đối (vị trí này tính từ đầu file). Kiểu dữ liệu của tham số này trùng với kiểu dữ liệu của hai hàm tellg() và tellp() ở trên.

Prototype thứ hai:

seekg(vị_trí, kiểu);

seekp(vị_trí, kiểu);

Việc sử dụng prototype này sẽ làm thay đổi vị trí hiện tại của con trỏ get và con trỏ put được xác định theo vị trí tương đối theo tham số *vị_trí* và tham số *kiểu*. Tham số *vị_trí* của một thành viên thuộc kiểu dữ liệu *off_type*, nó cũng là một kiểu số nguyên, nó tương ứng với vị trí của con trỏ get/put được đặt vào. Tham số *kiểu* là một kiểu dữ liệu *seekdir*, nó là một kiểu enum để xác định vị_trí của con trỏ get/put kể từ tham số *kiểu*, nó có thể nhận một trong các giá trị sau đây.

ios::beg	vị_trí được đếm từ vị trí bắt đầu của luồng
ios::cur	vị_trí được đếm từ vị trí hiện tại của luồng
ios::end	vị_trí được đếm từ vị trí cuối của luồng

Điều này có nghĩa là hàm chồng chất hai tham số này cũng tương tự hàm một tham số, nhưng vị trí bắt đầu tính trong hàm một tham số luôn là từ vị trí đầu tiên, còn hàm hai tham số có ba vị trí có thể bắt đầu đếm – bắt đầu (ios::beg), hiện tại (ios::cur) hay cuối file (ios::end).

Ta hãy quan sát ví dụ sau đây

Ví dụ	Kết quả
1. #include <iostream> 2. #include <fstream> 3. using namespace std; 4. int main(){ 5. long begin, end; 6. ifstream myfile(“example.txt”); 7. begin = myfile.tellg(); 8. myfile.seekg(0, ios::end); 9. end = myfile.tellg(); 10. myfile.close(); 11. cout<<”Size=”<<(end-begin)<<” bytes”; 12. return 0; 13. }	Size=10 bytes

Giải thích: trong chương trình trên chúng ta đang mở một file example.txt. Chúng ta đếm kích thước của file này. Khi mở file, con trỏ get sẽ đặt vào vị trí đầu file. Khi đó, dòng lệnh 7 sẽ gán giá trị khởi đầu cho biến begin (trong trường hợp này sẽ là 0). Dòng lệnh 8 sẽ đặt con trỏ get vào vị trí cuối cùng của file (vị trí 0 kể từ cuối file tính lên). Dòng lệnh 9 sẽ gán vị trí hiện tại – vị trí cuối file cho biến end. Điều đó có nghĩa là giá trị end-begin chính là kích

thước của file. Ta cũng cần lưu ý rằng, trong file văn bản, một kí tự tương ứng với 1 byte – đó cũng chính là quy định trong C++ (một kiểu char chiếm 1 byte). Hay nói chính xác, chương trình này đếm số kí tự trong file văn bản.

File nhị phân

Đối với file nhị phân, việc đọc ghi dữ liệu bằng toán tử tích trách >> và toán tử chèn << cũng như hàm getline là không có hiệu lực, bởi chúng không được định dạng theo kiểu văn bản như đối với file văn bản ở trên (không dùng phím space để tạo khoảng cách, không có kí tự xuống dòng…).

Các luồng của file gồm hai hàm thành viên để đọc và ghi dữ liệu là read và write. Hàm thành viên write là hàm thành viên của lớp ostream thừa kế cho ofstream. Và hàm read là thành viên của lớp istream thừa kế cho ifstream. Các đối tượng của lớp fstream có cả hai hàm thành viên này. Chúng có prototype như sau:

write(khối_bộ_nhớ, kích_thước);

read(khối_bộ_nhớ, kích_thước);

Ở đó, *khối_bộ_nhớ* là một con trỏ kiểu char (char*) và nó biểu diễn địa chỉ của một mảng các byte mà nó đọc hoặc ghi được. Biến *kích_thước* là một kiểu số nguyên integer, nó chỉ định số các kí tự có thể đọc/ghi lên khối bộ nhớ. Chúng ta hãy quan sát ví dụ sau đây

Ví dụ

```
#include<iostream>
#include<fstream>
using namespace std;
ifstream::pos_type size;
char* memblock;
int main(){
	ifstream file("example.bin", ios::in|ios::binary|ios::ate);
	if(file.is_open()){
		size = file.tellg();
		memblock = new char[size];
		file.seekg(0, ios::beg);
		file.read(memblock, size);
		file.close();
		cout<<"Hoan tat !";
		//Làm việc với dữ liệu trong con trỏ memblock
```

```
            delete[] memblock;
        }
        else cout<<"Khong mo duoc file.";
        return 0;
}
```

Giải thích: trong chương trình, ta mở file example.bin. Chế độ mở file để đọc (ios::in), theo kiểu file nhị phần (ios::binary), đặt con trỏ get vào cuối file (ios::ate). Sau khi mở file, hàm file.tellg() sẽ cho biết kích thước thực của file. Sau đó hàm file.seekg sẽ đặt vị trí con trỏ get vào đầu file (vị trí 0 kể từ vị trí đầu tiên) và tiến hành đọc theo khối bộ nhờ nhờ vào file.read. Sau khi hoàn tất, phương thức close được triệu gọi để kết thúc việc đọc file. Khi đó, dữ liệu từ file đã đọc vào mảng memblock. Chúng ta có thể bổ sung tác vụ thao tác với dữ liệu nếu muốn. Cuối cùng, con trỏ memblock sẽ bị xóa để giải phóng bộ nhớ.

Bộ đệm và Đồng bộ hóa

Khi thực thi các tác vụ đọc/ghi dữ liệu với file, chúng ta thực thi như trên nhưng thông qua một bộ đệm có kiểu dữ liệu streambuf. Bộ đệm này là một khối bộ nhớ đóng vai trò trung gian giữa các luồng và file vật lý. Ví dụ, với ofstream, mỗi thời điểm hàm put được gọi, kí tự không ghi trực tiếp lên file mà nó sẽ được ghi lên bộ đệm. Khi bộ đệm đầy, mọi dữ liệu chứa trong đó sẽ được ghi lên file (nếu đó là luồng ghi dữ liệu) hay xóa bỏ để làm rãnh bộ nhớ (nếu đó là luồng đọc dữ liệu). Tiến trình này được gọi là đồng bộ hóa và có các tình huống sau đây:

- Khi file đã đóng: trước khi đóng một file, tất cả dữ liệu trong bộ nhớ nếu chưa đầy vẫn được đồng bộ và chuẩn bị để đọc/ghi lên file.
- Khi bộ nhớ đầy: bộ đệm có kích thước giới hạn. Khi nó đầy, nó sẽ tự động đồng bộ hóa.
- Bộ điều phối: khi các bộ điều phối được sử dụng trên luồng, một tiến trình đồng bộ dứt điểm sẽ được diễn ra. Những bộ điều phối này bao gồm: flush và endl.
- Hàm thành viên sync(): nếu hàm thành viên sync() được triệu gọi, tiến trình đồng bộ hóa sẽ diễn ra. Hàm này trả về một kiểu integer (int) tương ứng với -1, nếu luồng không có bộ đệm liên kết hoặc trong trường hợp đọc/ghi thất bại. Ngược lại, nó sẽ trả về giá trị 0.

CHƯƠNG 17. CÁC LỚP THƯ VIỆN

1. Lớp số phức complex

Đây là lớp template. Khi khởi tạo một lớp đối tượng số phức, ta cần chỉ định kiểu dữ liệu cho nó. Nó nằm trong thư viện complex. Ví dụ về việc khởi tạo một số phức:

complex<float> c(10.2, 3);

sẽ khởi tạo một đối tượng số phức mà phần thực và phần ảo của nó là các số thực. Thông thường, kiểu dữ liệu được chỉ định cho lớp số phức là kiểu dữ liệu thực: float, double hoặc long double. Nhưng chúng ta hoàn toàn có thể sử dụng kiểu số nguyên.

Các hàm thành viên của lớp số phức

Tên thành viên	Mức truy cập	Chức năng
real imag	private	Phần thực và phần ảo.

Phương thức khởi tạo:
complex<?> c;
complex<?> c(real, imag);
complex <?> c(d);
	public	Các phương thức khởi tạo: không tham số, có hai tham số và hàm tạo sao chép.

imag()	public	Phương thức getImage – trả về giá trị phần ảo.
real()	public	Phương thức getReal – trả về giá trị của phần thực.
operator= operator+= operator-= operator*= operator/=	public	Các hàm toán tử thành viên: toán tử gán, toán tử cộng hợp nhất, toán tử trừ hợp nhất, toán tử nhân hợp nhất, toán tử chia hợp nhất.
_Add(complex) _Sub(complex) _Mul(complex) _Div(complex)	protected	Các phương thức cộng, trừ, nhân và chia hai số phức.

Các hàm toàn cục (hoặc hàm bạn)

Tên phương thức	Chức năng
operator + operator – operator * operator / operator = operator == operator != operator >> operator <<	Toán tử cộng hai số phức. Toán tử trừ hai số phức. Toán tử nhân hai số phức. Toán tử chia hai số phức. Toán tử gán số phức. Toán tử so sánh bằng. Toán tử so sánh khác. Toán tử nhập số phức. Toán tử xuất số phức.
real(complex)	Trả về phần thực của số phức.
imag(complex)	Trả về phần ảo của số phức.
abs(complex)	Trả về giá trị modul của số phức. Modul của số phức được tính theo công thức $\sqrt{real^2 + imag^2}$.
norm(complex)	Trả về giá trị là chuẩn của số phức. Chuẩn của số phức là bình phương giá trị của modul, tức là $real^2 + imag^2$.
conj(complex)	Trả về số phức liên hợp. Số phức liên hợp của số phức a nhận được bằng cách thay phần ảo của a bằng –a.
polar(float, float)	Trả về một số phức trong hệ tọa độ Decac. Hai tham số truyền vào tương ứng với modul và argument của số phức. $\begin{cases} real = modul * \cos(\arg\ \ ) \\ imag = modul * \sin(\arg\ \ ) \end{cases}$
cos(complex)	Trả về giá trị cosin của số phức.
sin(complex)	Trả về giá trị sin của số phức.
tan(complex)	Trả về giá trị tan của số phức.
cosh(complex)	Trả về giá trị cosin hyperbol của số phức.
sinh(complex)	Trả về giá trị sinh hyperbol của số phức.
tanh(complex)	Trả về giá trị tan hyperbol của số phức.
exp(complex)	Trả về giá trị e lũy thừa của số phức.
sqrt(complex)	Trả về giá trị căn bậc hai của số phức.
log(complex)	Trả về giá trị logarith cơ số tự nhiên (logarith nepe) của số phức.
log10(complex)	Trả về giá trị logarith cơ số 10 của số phức.
pow(complex, <T>)	Trả về lũy thừa của số phức. Tham số thứ hai có thể là số phức, số thực, số nguyên.

Đây là lớp số học khá hữu dụng trong tính toán khoa học. Chúng ta có thể sử dụng nó mà không cần xây dựng lại lớp này. Tuy nhiên, khi bắt đầu tiếp xúc với lập trình hướng đối tượng, cần thiết phải xây dựng nó.

2. Lớp ngăn xếp stack

Lớp stack cũng là một lớp template. Nó làm việc theo nguyên tắc hằng đợi – vào trước ra sau. Lớp stack cung cấp các phương thức để làm việc theo nguyên tắc Lifo như trong học phần cấu trúc dữ liệu và giải thuật.

Khai báo một đối tượng thuộc lớp stack:

stack<int> s;

Cần lưu ý rằng, lớp stack nằm trong thư viện stack. Sau đây là một số hàm thành viên của lớp stack.

Tên phương thức	**Mức truy cập**	**Chức năng**
stack<?>c; stack<?,deque, allocator>	public	Hàm tạo. Đối với hàm tạo thứ hai, ta cần sử dụng các đối tượng deque và allocator. Đây cũng là các lớp template.
empty()	public	Phương thức hằng. Trả về kiểu bool. Nhận giá trị true nếu stack rỗng và ngược lại stack không rỗng thì nhận giá trị false.
size()	public	Phương thức hằng. Trả về kiểu số nguyên là kích thước của stack (tức số phần tử của stack).
top()	public	Có hai phương thức top được quá tải: hằng và không hằng. Nó trả về phần tử nằm ở đỉnh của stack (tức phần tử được đưa vào sau cùng).
push()	public	Bổ sung một phần tử mới vào trong stack.
pop()	public	Lấy ra một phần tử trong stack. Phương thức này không trả về giá trị của phần tử vừa được lấy ra.

Stack trong trường hợp này không bị giới hạn kích thước (bởi nó được khai báo động).

3. Lớp hàng đợi queue

Queue là một lớp template. Để sử dụng queue, ta cần khai báo thư viện queue tương ứng. Queue làm việc theo nguyên tắc Fifo – tức vào trước thì ra trước. Sau đây là ví dụ về việc tạo một đối tượng queue:

queue<int> q;

Các phương thức thành viên của lớp queue:

Tên phương thức	Mức truy cập	Chức năng
queue<?> queue<?, deque, allocator>	public	Hàm tạo. Đối với hàm tạo thứ hai, ta cần sử dụng các đối tượng deque và allocator. Đây cũng là các lớp template.
empty()	public	Phương thức hằng. Trả về kiểu bool. Nhận giá trị true nếu queue rỗng và ngược lại queue không rỗng thì nhận giá trị false.
size()	public	Phương thức hằng. Trả về kiểu số nguyên là kích thước của queue (tức số phần tử của queue).
front()	public	Có hai phương thức front được quá tải: hằng và không hằng. Nó trả về phần tử nằm ở đầu ra (tức phần tử được đưa vào đầu tiên).
back()	public	Có hai phương thức front được quá tải: hằng và không hằng. Nó trả về phần tử nằm ở đầu vào (tức phần tử được đưa vào sau cùng).
push()	public	Bổ sung một phần tử mới vào trong queue. Phần tử được đưa vào theo hướng back.
pop()	public	Lấy ra một phần tử trong queue. Phương thức này không trả về giá trị của phần

		tử vừa được lấy ra. Phần tử được lấy ra theo hướng front.

Cũng tương tự như stack, queue trong trường hợp này cũng không bị giới hạn kích thước (vì được kha báo động).

3. Lớp vector

Cần lưu ý rằng, lớp vector có cấu trúc tương đối giống với mảng. Vector không phải là một lớp đối tượng như trong hình học. Các phần tử của vector được sắp xếp liên tục trong bộ nhớ. Chúng ta không thể rời rạc hóa các phần tử (khi bổ sung các phần tử, cần bổ sung một cách liên tục; nếu không sẽ bị lỗi cấp phát bộ nhớ). Chúng ta có thể truy cập đến các phần tử của vector thông qua chỉ số hoặc iterator.

Chương trình	**Kết quả**

#include <iostream>
#include <vector>
 	
using namespace std;
 	
int main()
{
vector<int> v;
v.push_back(10);
v.push_back(20);
v.push_back(22);
//=====Cách 1=====
cout<<"Theo chi so:"<<endl;
for (unsigned i=0; i<v.size(); i++)
cout<<v[i]<<endl;
//=====Cách 2=====
cout<<"Theo con tro iterator:"<<endl;
vector<int>::iterator it;
for (it=v.begin(); it<v.end();++it)
cout<<*it<<endl;
return 0;
}
	Theo chi so:
	10
	20
	22
	Theo con tro iterator:
	10
	20
	22

Theo cách 1, chúng ta truy cập đến các phần tử của vector theo cách truy cập như đối với mảng; cách thứ hai cho phép ta truy cập thông qua con trỏ iterator. Sau đây, chúng ta sẽ tham khảo các phương thức thành viên của lớp vector.

Tên phương thức	Mức truy cập	Chức năng
vector<?> vector<?, allocator>	public	Hàm tạo.
~vector()	public	Hàm hủy.
operator =	public	Toán tử gán.
begin()	public	Trả về iterator tương ứng với iterator đầu tiên của vector.
end()	public	Trả về iterator tương ứng với iterator cuối của vector.
rbegin()	public	Trả về iterator nghịch đảo của iterator đầu tiên của vector.
rend()	public	Trả về iterator nghịch đảo của iterator cuối của vector.
size()	public	Trả về kích thước của vector.
max_size()	public	Trả về kích thước cực đại của vector.
resize(int) resize(int, ?)	public	Thay đổi kích thước của vector. Nó có hai chồng chất hàm, tham số int thứ nhất tương ứng với kích thước mới của vector; tham số template thứ hai tương ứng với giá trị được bổ sung mặc định.
capacity()	public	Trả về kích thước bộ nhớ đã cấp phát cho các phần tử của vector. Cần lưu ý, cơ chế cấp phát này trong C++ là tự động và luôn đảm bảo size()<=capacity().
empty()	public	Trả về giá trị true nếu vector rỗng và ngược lại.
reserve(int)	public	Thay đổi kích thước cho vùng bộ nhớ lưu trữ các phần tử đã khởi tạo của vector. Tham số của hàm tương ứng với giá trị trả về của phương thức capacity.
operator[int]	public	Trả về phần tử tương ứng với vị trí được chỉ định (như đối với mảng).
at(int)	public	Tham chiếu đến phần tử tương ứng với chỉ số được chỉ định. Phương thức này tương tự như

		toán tử [] ở trên.
front()	public	Trả về phần tử đầu tiên (không phải là iterator như begin).
back()	public	Trả về phần tử cuối cùng (không phải là iterator như end).
assign(iterator, iterator) assign(int, ?)	public	Khởi gán giá trị cho vector.
push_back(?)	public	Bổ sung một phần tử vào cuối vector.
pop_back()	public	Loại bỏ phần tử phía cuối của vector.
iterator(iterator, ?) insert(iterator, ?) insert(iterator, int , ?) insert(iterator, iterator, iterator)	public	Phương thức thứ nhất trả về iterator, ba phương thức còn lại trả về void. Vector sẽ được nới rộng bằng cách bổ sung thêm các phần tử mới. Phương thức đầu tiên trả về iterator cuối cùng của vector sau khi nới rộng kích thước của nó.
erase(iterator) erase(iterator, iterator)	public	Xóa bỏ các phần tử của vector. Phương thức đầu xóa một phần tử, phương thức hai – xóa các phần tử trong vùng giữa hai tham số được chỉ định. Sau khi xóa bỏ, nó trả về iterator của phần tử cuối.
swap(vector&)	public	Hoán đổi giá trị của hai vector.
clear()	public	Xóa bỏ hoàn toàn các phần tử của vector. Sau khi xóa bỏ, kích thước của nó là 0.
get_allocator()	public	Trả về số lượng các đối tượng được cấp phát bộ nhớ sử dụng khi khở tạo vector.

4. Lớp string

Thư viện chuẩn string của C++ cung cấp lớp template tương ứng với string và wstring. Các phương thức thành viên của lớp string bao gồm:

Tên phương thức	**Mức truy cập**	**Chức năng**
string() string(const string&)	public	Các phương thức khởi tạo: - Khởi tạo không tham số.

string(const string&, int, int) string(const char*, int) string(const char*) string(int, char)		- Sao chép hàm tạo. - Sao chép xâu con từ vị trí int thứ nhất với độ dài int thứ hai. - Sao chép các kí tự của mảng xâu kí tự với độ dài là tham số int (kể từ vị trí đầu tiên). - Tạo một xâu kí tự có nội dung là kí tự char và độ dài là tham số int.
Các iterator: begin, end, rbegin, rend	public	Xem ở phần lớp vector.
size, max_size, resize, capacity, reserve, clear, empty	public	Xem ở phần lớp vector.
operator[int] at	public	Xem ở phần lớp vector.
length()	public	Trả về độ dài của xâu.
operator+=	public	Cộng dồn xâu.
append(const string&) append(const string&, int, int) append(const char*, int) append(const char*) append(int, char)	public	Bổ sung xâu hoặc một phần của xâu vào xâu cũ. Các tham số này tương tự như trong phương thức khởi tạo.
push_back(char)	public	Bổ sung một kí tự vào xâu.
assign, erase, swap	public	Xem ở lớp vector.
insert(int, const string&) insert(int, const string&, int, int) insert(int, const char*) insert(int, int, char) insert(iterator, char) insert(iterator, int, char) insert(iterator, iterator, iterator)	public	Chèn xâu con vào xâu. Các phương thức sử dụng tham số iterator hoàn toàn tương tự như trường hợp lớp vector. Phương thức 1 sẽ chèn xâu con vào xâu ban đầu tại vị trí int. Phương thức hai tương tự, nhưng xâu con chỉ lấy từ tham số int thứ 2 với độ dài là tham số int thứ 3. Phương thức 3 tương tự phương thức 1. Phương thức 4 sẽ chèn kí tự char vào vị trí tham số int thứ nhất với số lần là tham số

		int thứ 2.
replace	public	Thay thế một phần của xâu theo các tham số tương tự như phương thức khởi tạo.
c_str	public	Chuyển sang xâu dạng C tương ứng.
data	public	Trả về mảng các kí tự.
get_allocator	public	Xem ở lớp vector.
copy(char*, int, int=0)	public	Sao chép xâu.
find(const string&, int=0) find(const char*, int, int) find(const char*, int=0) find(char, int=0)	public	Tìm kiếm xâu con trong một xâu.
find_fist_of find_last_of find_fist_not_of find_last_not_of	public	Tìm kiếm xâu đầu tiên, cuối cùng, không phải là đầu tiên, không phải là cuối cùng.
substr(int, int)	public	Trả về xâu con từ vị trí int thứ nhất với độ dài int thứ hai.
compare	public	So sánh hai xâu. Nhận giá trị 0, nếu hai xâu bằng nhau và ngược lại.

Các hàm toàn cục.

Tên phương thức	**Chức năng**
operator+	Cộng hai xâu.
swap(string&, string&)	Hoán đổi nội dung của hai xâu.
operator== operator!= operator< operator> operator>= operator<=	Các toán tử so sánh hai xâu. Xâu s>ss nếu các kí tự trong s đứng trước ss trong bảng chữ cái.
getline	Nhập xâu.
operator<<	Chèn xâu và chuyển hóa thành luồng stream.
operator>>	Trích tách xâu từ luồng stream.

5. Lớp list

Tên phương thức	Mức truy cập	Chức năng
list<?>(allocator) list<?>(int, ?, allocator) list<?>(list)	public	Hàm tạo.
~list()	public	Hàm hủy.
begin, end, rbegin, rend	public	Xem ở lớp vector.
empty, size, max_size, resize	public	Xem ở lớp vector.
front, back, push_front, push_back, pop_front, pop_back, insert, erase, swap, clear.	public	Xem ở lớp vector.
splice(iterator, list)	public	Chuyển các phần tử từ danh sách list sang danh sách chứa từ vị trí được chỉ định.
remove(const T&)	public	Xóa các phần tử có giá trị được chỉ định.
remove_if	public	Xóa phần tử và dồn danh sách lại.
unique	public	Xóa các phần tử trùng lặp.
merge	public	Nhập hai danh sách lại.
sort	public	Sắp xếp danh sách.
reverse	public	Đảo ngược thứ tự các phần tử trong danh sách.
get_allocator	public	Xem ở lớp vector.

6. Lớp map

Map là một cấu trúc dữ liệu gồm có hai phần là khóa và giá trị. Mỗi khóa là duy nhất và nó tương ứng với một giá trị cụ thể. Lớp map là một lớp template.

Tên phương thức	Mức truy cập	Chức năng
map<?,?>	public	Hàm tạo.
~map()	public	Hàm hủy.
begin, end, rbegin, rend	public	Xem ở lớp vector.
empty, size, max_size, resize	public	Xem ở lớp vector.
insert, erase, swap, clear.	public	Xem ở lớp vector.
operator[]	public	Xem ở lớp vector.

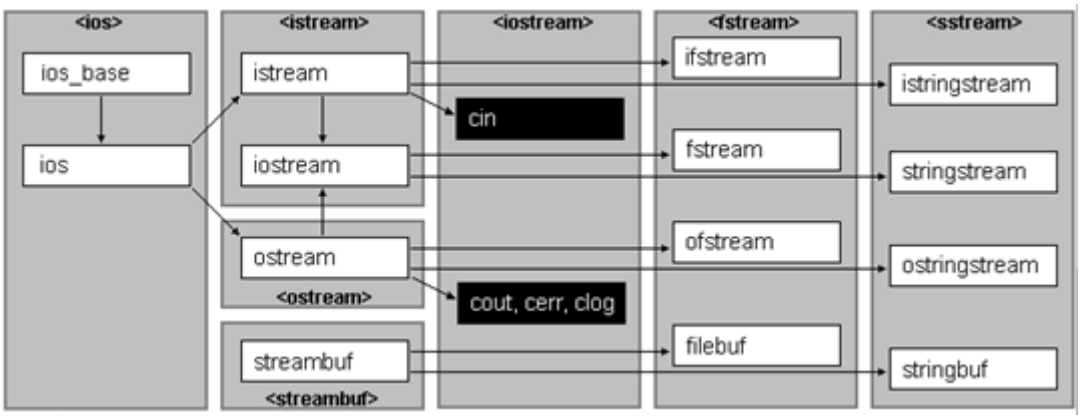
key_comp	public	Trả về key của đối tượng.
value_comp	public	Trả về giá trị của đối tượng.
find	public	Trả về chỉ số dạng iterator của khóa tìm kiếm.
count		Đếm số đối tượng có khóa chỉ định.
lower_bound		Trả về iterator của biên dưới.
upper_bound		Trả về iterator của biên trên.
equal_range		Trả về cặp cận trên và cận dưới.
get_allocator	public	Xem ở lớp vector.

7. Lớp set

Set là một dạng cấu trúc dữ liệu mà mỗi phần tử của nó là một khóa (nghĩa là không cho phép chứa các phần tử trùng lặp). Các phương thức của set hoàn toàn tương tự như của map. Set cũng là một lớp template.

8. Các lớp thư viện nhập xuất

Thư viện các luồng nhập xuất



Thư viện iostream là thư viện hướng đối tượng cung cấp các chức năng nhập xuất cơ bản sử dụng luồng stream. Một luồng là một đối tượng trừu tượng có thể làm việc với các thiết bị xuất nhập. Một luồng về cơ bản có thể được biểu diễn như một dãy các kí tự có độ dài vô hạn.

Các luồng được quản lý và lưu trữ tài nguyên vật lý dưới dạng kí tự như tập tin, bàn phím, console, các kí tự này có thể được đọc từ hoặc ghi lên luồng.

Ví dụ các tập tin trong C++ được tổ chức và tương tác với tập tin. Một luồng tập tin được sử dụng để mở tập tin, đọc, ghi trên tập tin.

Để làm việc với luồng, C++ cung cấp thư viện iostream chuẩn chứa các đối tượng sau:

Các lớp template cơ bản

Cơ sở của thư viện iostream là cấu trúc cây phả hệ của các lớp template. Các lớp template cung cấp hầu hết các tính năng trong thư viện của C++ để có thể làm việc với các kiểu dữ liệu hiện đại được bổ sung sau này. Nó là một tập hợp các lớp template, mỗi lớp có hai tham số template: kiểu char (charT) để xác định kiểu của các phần tử và tham số traits để cung cấp các đặc tính bổ sung cho mỗi loại phần tử. Lớp template trong cây phả hệ lớp có cùng tên với đối tượng thể hiện của lớp char và có tiền tố là basic_. Ví dụ lớp template của istream là basic_istream, của fstream là basic_fstream,… Chỉ có duy nhất một ngoại lệ đó là ios_base là một lớp độc lập và không tồn tại lớp tương ứng là base.

Thực thể của lớp template

Thư viện được chia thành hai tập hợp thực thể của lớp iostream: một là theo hướng hẹp để quản lý các phần tử kiểu char và một phần tử thuộc kiểu khác; theo hướng rộng để tổ chức các phần tử kiểu wchar_t.

Thực thể theo hướng hẹp char có thể được biết đến nhiều hơn như là một thực thể của thư viện iostream. Các lớp như ios, iostream và ofstream là các thực thể theo hướng rộng. Biểu đồ ở trên là biểu đồ biểu diễn tên và mối quan hệ giữa các lớp theo hướng hẹp.

Các thực thể theo hướng rộng wchar_t có cùng tên với thực thể theo hướng hẹp nhưng có tiền tố bổ sung là w. Ví dụ wios và ios, wistream và wofstream.

Các đối tượng chuẩn

Là một phần của thư viện iostream, được đặc tả trong tệp tiêu đề <iostream> thường được sử dụng để thực hiện các chức năng nhập xuất chuẩn. Chúng được chia thành hai nhóm: nhóm đối tượng định hướng hẹp bao gồm ba đối tượng phổ biến nhất: cin, cout, cer và clog và nhóm định hướng rộng bao gồm wcin, wcout, werr và wlog.

Các kiểu dữ liệu

Các lớp iostream sử dụng các kiểu cơ bản. Chúng sử dụng kiểu dữ liệu cơ bản phụ thuộc vào sự thể hiện của chúng. Ví dụ theo mặc định char và wchar_t, kiểu streampos, streamoff và streamsize dùng để biểu diễn vị trí, offset và kích thước tương ứng.

Các thao tác thực hiện

Các thao tác thực thi trên luồng là các hàm toàn cục được thiết kế để sử dụng cùng với toán tử chèn << và toán tử trích tách >>. Ngoài ra còn có các chức năng định dạng như endl, hex, scientific.

Các tập tin

- <ios>, <istream>, <ostream>, <streambuf> và <iosfwd> không thường được sử dụng một cách trực tiếp trong các chương trình C++. Chúng mô tả lớp cơ sở của cây phả hệ và được đưa vào trong chương trình thông qua một tập tin tiêu đề khác của thư viện chứa các lớp dẫn xuất.
- <iostream> đặc tả các đối tượng sử dụng để kết nối thông qua các chuẩn nhập xuất (bao gồm cả cin, cout).
- <fstream> xác định các lớp luồng (như các lớp template basic_ifstream hay lớp ofstream) cũng như các đối tượng bộ đệm (basic_filebuf). Những lớp này được sử dụng để làm việc với tập tin bằng cách sử dụng luồng.
- <sstream> các lớp xác định trong tập tin này thường sử dụng đối tượng string nếu chúng là luồng.
- <iomanip> đặc tả một vài chức năng chuẩn với các tham số được sử dụng kết hợp với các toán tử tách và chèn để chỉnh sửa các cờ hiệu và các thông số định dạng.

Các thành phần của thư viện iostream (với thực thể char)

1) Các lớp trong iostream (có thể tham khảo thêm trong thư viện này).

- *Lớp ios_base*
- *Lớp ios*
- *Lớp istream*
- *Lớp ostream*
- *Lớp ifstream*

- *Lớp ofstream*
- *Lớp fstream*
- *Lớp istringstream*
- *Lớp ostringstream*
- *Lớp streambuf*
- *Lớp filebuf*
- *Lớp stringbuf*

2) Các đối tượng

- *Đối tượng cin*

Xem chương "Nhập xuất cơ bản". Các phương thức và thuộc tính khác xem thêm ở lớp istream.

- *Đối tượng cout*

Xem chương "Nhập xuất cơ bản". Các phương thức và thuộc tính của lớp khác xem thêm ở lớp ostream.

- *Đối tượng cerr*

Thuộc lớp ostream được sử dụng để in luồng lỗi cơ bản. Mặc định, hầu hết hệ thống đều có một chuẩn lỗi cơ bản để in ra màn hình. Bởi vì cerr là một đối tượng của lớp ostream, chúng ta có thể viết các kí tự theo dữ liệu được định dạng nhờ vào toán tử chèn << hoặc không định dạng dữ liệu nếu sử dụng hàm thành viên write.

```
#include <iostream>

int main()
{
        int num;
        std::cin>>num;
        if (std::cin.good())
        {
            std::cout<<"Good !";
        }else
        {
            std::cerr<<"Error";
            //Hoặc std::cerr.write("Error",
sizeof(std::streamsize));
```

```
        }
        return 0;
}
```

Nếu số nguyên nhập vào đúng định dạng thì sẽ thực hiện chức năng in ra "Good", ngược lại sẽ in ra "Error".

- *Đối tượng clog*

Tương tự như cerr, clog dùng để biểu diễn luồng đăng nhập chuẩn.

3) Các kiểu dữ liệu

- *Kiểu fpos*: xác định vị trí trong luồng. Về bản chất nó là một lớp template. Nó có hai thành viên getter và setter tương ứng là: state() và state(stateT).
- *Kiểu streamoff*: xác định vị trí của offset trong luồng. Ta hoàn toàn có thể chuyển đổi nó thành streamsize, fpos hoặc streampos.
- *Kiểu streampos*: tương tự với fpos.
- *Kiểu streamsize*: biểu diễn kích thước luồng.

4) Các thao tác thực hiện

- *boolalpha*: thường sử dụng kết hợp với toán tử nhập >> hoặc xuất <<. Nếu tham số là boolalpha thì tham số bool là 1 và 0 tương ứng sẽ được hiểu là true và false.

```
#include <iostream>

int main()
{
        bool num;
        std::cin>>std::boolalpha>>num;
        std::cout<<num;
        return 0;
}
```

```
true
true
```

- *dec:* đọc ghi dữ liệu số theo dạng thập phân (mặc định).

```
#include <iostream>

int main()
```

```
true
true
```

```
{
        int num = 10;
        std::cout<<std::dec<<num;
        return 0;
}
```

- *endl*: chèn dấu xuống dòng. Tương tự như "\n".
- *ends*: chèn kí tự trống "\0".
- *fixed*: áp dụng cho định dạng số thực có dấu chấm. Khi cout.precision(int) được triệu gọi, thì số chữ số thập phân (tham số int) sẽ được áp dụng nếu chuẩn xuất dữ liệu có sử dụng fixed (khác với none - sẽ áp dụng theo mặc định và không phụ thuộc vào việc triệu gọi phương thức trên).

```
#include <iostream>

int main()
{
        float a = 3.141516f, b = 1.00f;
        std::cout.precision(3);
        std::cout<<a<<"\t"<<b<<std::endl;
        std::cout<<std::fixed<<a<<"\t"<<b;
        return 0;
}
```

```
3.14    1
//Không bật fixed

3.142   1.000
//Bật fixed
```

- *flush*: đồng bộ hóa bộ đệm với con trỏ luồng. Điều này có nghĩa là tất cả các kí tự chưa được ghi trong bộ đệm sẽ được ghi ra liên tục.

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

int main () {

  ofstream outfile
("test.txt");

  for (int n=0; n<100; n++)
    outfile << n << flush;

  outfile.close();

  return 0;
}
```

```
//Dữ liệu sẽ được ghi liên tục lên tập tin test.txt. Dữ liệu trong file này sẽ được cập nhập liên tục 100 lần.
```

- *hex*: thiết lập cờ hiệu chuỗi thập lục phân cho số được in ra.

cout<<hex<<15; //Kết quả - F

- *internal*: kéo giãn vùng dữ liệu về hai phía (tương tự justify trong Microsoft Word) theo độ rộng được thiết lập bởi cout.width(int).
- *left*: tương tự internal, nhưng dữ liệu được kéo về bên trái.
- *noboolalpha*: ngược lại với boolalpha, chỉ in ra giá trị là 1 và 0, dù cho giá trị đúng có được thiết lập là true, và giá trị sai được thiết lập là false.
- *noshowbase*: không hiển thị dạng thức của cơ số. Ví dụ cơ số 8 sẽ có dạng 0x… Nếu cờ hiệu này được triệu gọi, dòng dữ liệu in ra sẽ không chứa 0x.
- *noshowpoint*: không hiển thị dấu chấm động.
- *noshowpos*: không hiển thị dấu của số dương.
- *noskipws*: không bỏ qua các kí tự trắng khi tách dữ liệu (khi kết hợp sử dụng lớp istringstream).
- *nounitbuf*: bộ đệm sẽ không bị làm cạn sau mỗi lần thực hiện thao tác ghi dữ liệu.
- *nouppercase*: không hiển thị phần chữ trong các giá trị của hệ cơ số 16 dưới dạng chữ hoa (nghĩa là luôn viết thường a, b, c, d, e, f).
- *oct*: thiết lập cờ hiệu cho chuỗi bát phân in ra (chuyển một số thành chuỗi bát phân).
- *resetiosflags* : hủy bỏ cờ hiệu tương ứng với hệ cơ số được chọn.
- *right*: tương tự internal, nhưng dữ liệu được kéo về bên phải.
- *scientific*: tương tự fixed, nhưng nó hiển thị số thập phân dấu chấm động dưới dạng E.
- *setbase*: thiết lập hệ cơ số sẽ chuyển đổi. Nó tương đương với hex – nếu cơ số thiết lập là 16; oct – 8; dec – 10.
- *setfill*: bổ sung các kí tự được chỉ định vào phần còn trống trong vùng dữ liệu được chỉ định.
- *setiosflags*: thiết lập cờ hiệu với hệ cơ số được chọn.
- *setprecision*: thiết lập số ô trống dành cho cả phần nguyên lẫn phần thập phân (không tính dấu chấm). Cũng hoạt động tương ứng với fixed.
- *setw*: thiết lập độ rộng cho vùng dữ liệu hiển thị.
- *showbase*: hiển thị dạng thức cơ số. Cờ hiệu này ngược với noshowbase.

- *showpoint*: hiển thị dấu chấm động.
- *showpos*: hiển thị dấu + trước số dương.
- *skipws*: bỏ qua kí tự trắng khi tách dữ liệu.
- *unitbuf*: bộ đệm sẽ bị làm cạn sau mỗi lần ghi dữ liệu.
- *uppercase*: hiển thị phần chữ trong hệ cơ số 16 dưới dạng chữ hoa (A, B, C, D, E, F).
- *ws*: bỏ qua các kí tự trắng (hoặc các kí tự tab, enter) khi tách dữ liệu.

HƯỚNG DẪN THỰC HÀNH

BÀI THỰC HÀNH SỐ 1
Nhập xuất cơ bản và Hàm trong C++.

Hãy thực hiện các công việc sau đây:

a. Xây dựng một chương trình gồm có hai phần: phần file header đặt tên là tieude.h, phần chương trình chính là main.cpp.

b. Tất cả các hàm trong chương trình cần phải khai báo prototype. Các tiêu đề file được đặt trong tệp tiêu đề. Chương trình chính chứa các hàm thực hiện các công việc sau và hàm main.

- Sử dụng hàm nhập xuất cơ bản để xây dựng hàm nhập vào một xâu kí tự dưới dạng kiểu dữ liệu string. Sau đó:

+ In ra xâu đảo ngược. Ví dụ "Toi di hoc", thì in ra "coh id ioT". Hàm này gọi là hàm **DaoXau**.

+ In ra các kí tự đảo ngược. Ví dụ "Toi di hoc", thì in ra "hoc di Toi". Hàm này gọi là hàm **DaoTu**.

- Nhập vào một mảng số nguyên, in ra tổng các phần tử của chúng. Hàm này gọi là hàm **TinhTong**.

- Sử dụng khai báo chồng chất hàm, để xây dựng hai hàm giải phương trình bậc nhất và phương trình bậc hai. Tên gọi của hai hàm này là **GiaiPhuongTrinh**.

- Sử dụng khai báo hàm với tham số mặc định để giải phương trình bậc nhất và phương trình bậc hai, hàm này có tên gọi là **GiaiPhuongTrinhTSMD**.

- Trong hàm main, hãy thực hiện các công việc sau: in ra thông báo nhập vào một xâu kí tự và gọi hàm DaoXau và DaoTu, nhập vào một mảng số nguyên và gọi hàm TinhTong. Sử dụng hàm xuất nhập cơ bản để đưa ra thông báo nhập các hệ số cho phương trình. Nếu hệ số c nhập vào bằng 0, thì thực hiện giải phương trình bậc nhất, còn c khác không thì thực hiện giải

phương trình bậc hai (giải các phương trình này bằng cách gọi các hàm xây dựng ở trên).

BÀI THỰC HÀNH SỐ 2
Xây dựng Lớp và Làm việc với Đối Tượng

1. Xây dựng lớp HangHoa gồm có các phương thức và thuộc tính sau:

Tên gọi	Mức truy cập	Loại	Giải thích
tenHang	private	Thuộc tính	Tên mặt hàng
ngaySanXuat	private		Ngày sản xuất
donGia	private		Đơn giá
soLuong	private		Số lượng
SetTenHang	public	Phương thức	Thiết lập tên hàng
GetTenHang	public		Tiếp nhận tên hàng
SetNgaySanXuat	public		Thiết lập ngày sản xuất
GetNgaySanXuat	public		Tiếp nhận ngày sản xuất
SetDonGia	public		Thiết lập đơn giá
GetDonGia	public		Tiếp nhận đơn giá
SetSoLuong	public		Thiết lập số lượng
GetSoLuong	public		Tiếp nhận số lượng
TinhTien	public		Tính tiền

Các phương thức trên bao gồm setter và getter. Phương thức **TinhTien** là phương thức dùng để tính số tiền mà khách hàng mua. Tính tiền sẽ bằng **donGia*soLuong**.

2. Xây dựng lớp KhachHang gồm các phương thức và thuộc tính sau:

Tên gọi	Mức truy cập	Loại	Giải thích
username	private	Thuộc tính	Tên tài khoản người dùng
password	private		Mật khẩu người dùng
SetUsername	public	Phương thức	Lập tài khoản người dùng
GetUsername	public		Tiếp nhận tài khoản người dùng
SetPassword	public		Lập mật khẩu người dùng
GetPassword	public		Tiếp nhận mật khẩu người dùng
MuaHang	public		Mua hàng

Các phương thức setter và getter hoàn toàn tương tự như trên. Phương thức **MuaHang** sẽ triệu gọi các phương thức thiết lập tên hàng, ngày sản xuất, và tính tiền của đối tượng HangHoa ở trên. Phương thức mua hàng

này, chỉ được chấp nhận khi username và password của khách hàng nhập vào trùng với username và password trong hệ thống.

Trong chương trình chính, hãy tạo hai đối tượng của hai lớp HangHoa và KhachHang. Username và Password của khách hàng nhập vào từ bàn phím. Kiểm tra nếu username và password trùng với username và password được thiết lập sẵn thì khi đó mới tiến hành khởi tạo đối tượng KhachHang, nếu ngược lại, hủy bỏ giao dịch và in thông báo: "*Xin loi, tai khoan cua quy khach khong ton tai trong he thong. Xin lien he dang ki voi chi nhanh khach hang gan nhat*". Trong trường hợp đăng nhập thành công, hãy thực hiện hành động mua hàng của đối tượng khách hàng này (thao tác mua được nhập vào từ bàn phím).

BÀI THỰC HÀNH SỐ 3
Hàm tạo, sao chép hàm tạo, hàm bạn, con trỏ this

Lưu ý: *trong bài thực hành ở chương này, chỉ cho phép sử dụng con trỏ đối tượng để thực hiện. Mọi phương án sử dụng khai báo đối tượng thông thường sẽ không được chấp nhận !*

1. Quay trở lại với bài thực hành số 2. Hãy tạo chương trình bằng cách thay thế các phương thức setter bằng các hàm tạo tương ứng. Hãy sử dụng con trỏ this trong trường hợp này. Bổ sung phương thức **ResetHangHoa** cho lớp HangHoa để đưa về các tham số mặc định cho các thuộc tính (xâu kí tự thì thiết lập về "", số nguyên/thực thì thiết lập về 0). Bổ sung phương thức **HuyBo** cho đối tượng KhachHang để hủy bỏ việc mua hàng. Trong chương trình chính, hãy tạo con trỏ đối tượng để thực hiện hành động mua hàng và hành động hủy bỏ.

2. Xây dựng lớp điểm Point và lớp hình tròn Round. Sử dụng hàm random để tạo tọa độ ngẫu nhiên cho một đối tượng điểm trong hàm tạo của đối tượng điểm. Cho trước tọa độ tâm của đường tròn, bán kính của nó. Hãy đưa ra các kết luận về một điểm được khởi tạo nằm trong hay ngoài đường tròn. Đây là phương thức thành viên của lớp hình tròn.

Khuyến khích: Nên sử dụng hàm bạn và lớp bạn.

BÀI THỰC HÀNH SỐ 4
Chồng chất Toán tử trong C++

Xây dựng 2 lớp số phức và lớp phân số. Sử dụng chồng chất toán tử để thực hiện các thao tác tính toán sau:

- **Lớp số phức**: phương thức khởi tạo (hoặc phương thức setter), phương thức sao chép hàm tạo, phương thức + (cộng hai số phức), - (trừ hai số phức), * (nhân hai số phức), / (chia hai số phức) và phương thức tính modul và argument của số phức. Xây dựng hàm chồng chất toán tử nhập/xuất dữ liệu với lớp số phức này.

- **Lớp phân số**: phương thức khởi tạo, phương thức sao chép hàm tạo, phương thức + (cộng hai phân số), phương thức – (trừ hai phân số), phương thức * (nhân hai phân số), phương thức / (chia hai phân số). Xây dựng hàm chồng chất toán tử nhập/xuất dữ liệu với lớp phân số này.

BÀI THỰC HÀNH SỐ 5
Kĩ thuật thừa kế trong C++

Xây dựng lớp HinhKhoi, chứa thuộc tính chiều cao (chieucao). Xây dựng hàm tạo tương ứng và phương thức sao chép hàm tạo.

Xây dựng lớp HinhKhoi1 thừa kế từ lớp HinhKhoi. Bổ sung thêm thuộc tính chiều dài (chieudai). Bổ sung hàm tạo và phương thức sao chép hàm tạo. Xây dựng tiếp hàm tính thể tích cho HinhKhoi1 (bằng chieucao*chieudai^2).

Xây dựng lớp lớp HinhKhoi2 thừa kế từ lớp HinhKhoi1. Bổ sung thêm thuộc tính chiều rộng (chieurong). Bổ sung hàm tạo và phương thức sao chép hàm tạo. Quá tải hàm tính thể tích cho HinhKhoi2 (bằng chieucao*chieudai*chieurong).

Xây dựng lớp HinhKhoi3 thừa kế từ lớp HinhKhoi. Bổ sung thêm thuộc tính bán kính (bankinh). Bổ sung hàm tạo và phương thức sao chép hàm tạo. Xây dựng hàm tính thể tích cho HinhKhoi3 (bằng chieucao*Pi*bankinh^2).

BÀI THỰC HÀNH SỐ 6
Lớp cơ sở trừu tượng trong C++

Xây dựng một lớp cơ sở trừu tượng Vector chứa ba phương thức ảo thuần túy: TinhDoDai (tính độ dài), SinGoc (tính sin của góc giữa hai Vector), TrucGiao (tìm vector trực giao – tức vector vuông góc với vector trên), hai thuộc tính thành viên là tọa độ x và y, hai phương thức setter (không sử dụng hàm tạo trong trường hợp này, vì lớp trừu tượng không có khả năng tạo ra một sự thể hiện, và hàm tạo cũng không được thừa kế); hoặc sử dụng phương thức tham chiếu getter.

Xây dựng lớp Vector2D thừa kế từ lớp cơ sở trừu tượng Vector để thực thi các phương thức ảo thuần túy nêu trên.

Xây dựng lớp Vector3D thừa kế từ lớp cơ sở trừu tượng Vector (bổ sung thêm tọa độ z và các phương thức getter, setter tương ứng) để thực thi các phương thức ảo thuần túy nêu trên.

Trong đó, Vector2D là vector 2 chiều (chỉ có hai tọa độ x và y); Vector3D là vector 3 chiều (có ba tọa độ x, y và z).

THANG ĐIỂM ĐÁNH GIÁ KĨ NĂNG						
Bài thực hành số	1	2	3	4	5	6
Điểm	20	15	20	15	20	10
Ngưỡng đạt	60-70					
Cộng điểm	70-80		80-90		90-100	
Quy đổi	+1		+1.5		+2	

- Mỗi bài thực hành có thể thực hiện ở nhà hoặc trên lớp, nhưng phải nộp bài đúng thời hạn.

- Hạn nộp mỗi bài thực hành tương ứng với buổi thực hành. Ví dụ: Bài thực hành số 1 phải nộp đúng hạn vào buổi thứ nhất (mới đạt điểm tối đa). Nếu muộn một buổi, trừ đi 2 điểm. Nếu nộp tất cả các bài vào buổi cuối cùng, thì tối đa chỉ đạt **Ngưỡng Đạt**.

- **Điểm tổng kết** môn học gồm có 4 cột điểm quá trình và 1 điểm thi. Điểm **Quá trình 1** = Điểm chuyên cần; **Điểm Quá trình 2**, **Điểm Quá trình 3** sẽ tương ứng với 2 bài trắc nghiệm; **Điểm Quá trình 4** = Điểm Thực hành/10. Điểm thi cuối cùng sẽ bằng kết quả thi đạt được cộng thêm điểm Quy đổi. Điểm tổng kết sẽ được tính theo công thức:

$$\frac{(QT_1 + QT_2 + QT_3 + QT_4)}{8} + \frac{THI}{2}$$

BÀI TẬP NÂNG CAO

Bài tập 1.

Xây dựng các hàm để tính các tổng sau đây:

$$S = \frac{1}{2} + \frac{2}{3} + \cdots + \frac{n-1}{n}$$

$$S = 1! + 2! + \cdots + n!$$

$$S = 1 - 2 + 3 - \cdots + (-1)^{n+1}.n$$

$$S = \frac{1!}{x+k} + \frac{2!}{x^2-k^2} + \cdots + \frac{n!}{x^n + (-1)^{n+1}.k^n}$$

$$S = 1! + 3! + \cdots + (2n-1)!$$

Trong đó: n , x và k là các tham số nhập vào từ bàn phím.

Bài tập 2.

Khai báo chuỗi kí tự bằng cách sử dụng con trỏ. Sau đó xây dựng các hàm để thực hiện các câu sau:

- Đếm số kí tự có giá trị là **a**.

- Đảo xâu kí tự.

- Đảo từ.

- Đếm số từ.

- Nhóm các kí tự cùng loại. Ví dụ:

Xâu ban đầu: aaabbbaacbd

Xâu in ra: 5a4b1c1d

- Thuật toán nén dữ liệu RLE (Run length Encoding) là thuật toán nén không mất dữ liệu lossless. Nó được sử dụng để nén ảnh đối với định dạng bmp. Thuật toán RLE sẽ thực hiện đếm số kí tự giống nhau và liên tiếp, sau

đó, sẽ thay thế toàn bộ dãy kí tự giống nhau này bằng **số kí tự đếm được** sau đó là **kí tự tương ứng**. Ví dụ:

Xâu ban đầu: aaaabbbbcdddAA

Xâu sau khi nén: 4a4b1c3d2A

+ Hãy xây dựng hàm RLE để nén dữ liệu. Với dữ liệu nhập vào từ bàn phím.

+ Hãy xây dựng hàm IRLE để giải nén dữ liệu. Với dữ liệu nhập vào từ bàn phím.

Bài tập 3.

Xây dựng các hàm thực hiện các chức năng sau đây trên mảng một chiều bằng hai cách: khai báo theo kiểu thông thường và khai báo bằng con trỏ.

- Hàm nhập dữ liệu cho mảng 1 chiều.

- Hàm xuất dữ liệu cho mảng 1 chiều.

- Hàm tính tổng các phần tử của mảng.

- Hàm tính tổng các phần tử của mảng là số nguyên tố.

- Hàm tính tổng các phần tử của mảng là số chính phương.

- Hàm tính tổng các số nguyên tố của mảng lớn hơn 10 và nhỏ hơn 100.

- Hàm tính tổng các phần tử của mảng là số chính phương chẵn.

- Hàm đếm số phần tử của mảng là số nguyên tố.

- Hàm đếm số phần tử của mảng là số chính phương.

- Hàm đếm số phần tử của mảng là số nguyên tố lớn hơn 10 và nhỏ hơn 50.

- Hàm đếm số phần tử của mảng là số chính phương chẵn.

- Hàm tìm kiếm chỉ số phần tử có giá trị x của mảng.

- Hàm tính giá trị trung bình của mảng.

- Hàm tìm chỉ số của phần tử có giá trị nhỏ nhất.

- Hàm tìm chỉ số của phần tử có giá trị lớn nhất.

- Hàm sắp xếp mảng theo thứ tự tăng dần (có thể sử dụng thuật toán sắp xếp bất kì).

- Hàm tính giá trị trung bình của các phần tử của mảng có giá trị chẵn.

- Hàm dồn tất cả các phần tử chẵn về một phía, các phần tử lẻ về một phía. Ví dụ, mảng ban đầu là: 1 4 5 6 2 3 thì kết quả sẽ là 1 5 3 4 6 2.

- Hàm xác định phần tử có giá trị gần với giá trị trung bình của mảng nhất.

- Hàm đẩy các phần tử của mảng lên n vị trí. Ví dụ mảng ban đầu là **1 3 2 5 4** 7 9. Nếu n = 2, thì mảng thu được sẽ là 7 9 **1 3 2 5 4**.

Bổ sung hàm main và các thư viện để nhận được một chương trình hoàn chỉnh. Các hàm cần được khai báo theo prototype.

Bài tập 4.

Xây dựng các hàm để thực hiện các chức năng sau đây trên mảng hai chiều bằng hai cách khai báo: theo kiểu thông thường và khai báo bằng con trỏ.

- Hàm nhập giá trị cho mảng hai chiều.

- Hàm xuất giá trị của mảng hai chiều theo dạng ma trận.

- Hàm cộng hai ma trận.

- Hàm nhân hai ma trận.

- Hàm thay thế tất cả các phần tử có giá trị lẻ của ma trận thành 0.

- Hàm thay thế tất cả các phần tử âm của ma trận bằng phần tử dương tương ứng.

- Hàm thay thế các phần tử có giá trị nhỏ hơn giá trị trung bình của ma trận bằng phần tử 0.

- Hàm tính lũy thừa của ma trận vuông.

Các hàm yêu cầu được xây dựng theo prototype. Các ma trận (mảng hai chiều) trong các bài tập trên là ma trận vuông. Bổ sung hàm main và các thư viện cần thiết để được một chương trình hoàn chỉnh.

Bài tập 5.

Hãy chọn lựa các phương pháp phù hợp trong lập trình hướng đối tượng để lập trình giải các bài toán sau đây.

a) Xây dựng lớp TamGiac (tam giác) gồm có ba cạnh với các phương thức sau:

- Các phương thức khởi tạo cho tam giác: không tham số, có tham số và sao chép hàm tạo.

- Các phương thức Getter.

- Phương thức KiemTra để cho biết nó có phải là một tam giác thực sự không.

- Phương thức tính diện tích tam giác nếu nó là một tam giác thực sự.

- Các phương thức nhập/xuất cho tam giác. Xây dựng theo toán tử.

b) Xây dựng lớp Diem (điểm) gồm có 3 tọa độ x, y, z và các phương thức sau:

- Các phương thức khởi tạo.

- Các phương thức Getter.

- Phương thức tính khoảng cách giữa hai điểm.

Xây dựng lớp Vector gồm có hai thuộc tính tương ứng với hai đối tượng Diem (điểm đầu và điểm mút). Hãy bổ sung các phương thức sau cho lớp Vector:

- Các phương thức khởi tạo.

- Các phương thức Getter.

- Phương thức tính độ dài của vector.

- Phương thức cộng hai vector. Xây dựng theo toán tử.

- Phương thức tính tích vô hướng của hai vector.

- Phương thức tính cosin góc giữa hai vector.

- Phương thức tính tích hữu hướng của hai vector.

Xây dựng lớp HinhCau (hình cầu) gồm hai thuộc tính là đối tượng Diem tương ứng với tâm hình cầu và bán kính R. Hãy bổ sung các phương thức sau cho lớp HinhCau:

- Các phương thức khởi tạo.

- Các phương thức Getter.

- Phương thức tính diện tích hình cầu.

- Phương thức xác định vị trí tương đối giữa hình cầu và một điểm.

- Phương thức xác định vị trí tương đối giữa hai hình cầu.

Yêu cầu chung: hãy bổ sung các hàm toán tử nhập xuất cho mỗi lớp đối tượng trên.

Bài 6.

Xây dựng lớp ConNguoi gồm có hai thuộc tính thành viên là: tên và tuổi. Các phương thức khởi tạo và các phương thức Getter tương ứng.

Lớp NhanVien thừa kế từ lớp ConNguoi, bằng cách bổ sung thêm hai thuộc tính là: mã nhân viên, lương và mức đóng góp (tính theo lương). Bổ sung phương thức khởi tạo và các phương thức Getter.

Các công ty quản lý nhân viên của mình. Đối tượng CongTy có các thuộc tính: tên công ty, mức đóng góp chuẩn, ngân sách hiện có, nguồn thu theo tháng, nguồn chi theo tháng. Giả sử nguồn chi này không bao gồm chi phí trả lương cho nhân viên. Đối tượng CongTy có phương thức tuyển dụng để tuyển thêm nhân viên, và phương thức sa thải để sa thải nhân viên.

Một nhân viên sẽ bị sa thải, nếu mức đóng góp của họ nhỏ hơn mức đóng góp chuẩn của công ty.

Một công ty sẽ tuyên bố phá sản nếu vốn điều lệ của họ bị âm. Vốn điều lệ là tổng ngân sách hiện có cộng với mức đóng góp của mỗi nhân viên trừ cho nguồn chi theo tháng và trừ tiếp cho tổng lương chi trả cho toàn bộ nhân viên. Hãy xây dựng chương trình để thực thi mô tả này.

Bài 7.

Hãy xây dựng chương trình ứng dụng theo mô tả sau.

- Lớp đối tượng người dùng gồm các thuộc tính: username, password, câu hỏi bảo mật và câu trả lời. Bổ sung các phương thức tương ứng cho phù hợp. Khi chạy chương trình, người dùng nhập vào username và password. Nếu trùng khớp với username và password đã tạo trong hệ thống thì thông báo đăng nhập thành công.

- Sau khi đăng nhập thành công, người dùng có quyền triệu gọi các phương thức tính toán của lớp số phức và phân số (cần xây dựng thêm hai lớp này).

- Nếu người dùng đăng nhập không thành công, yêu cầu họ xác minh rằng có phải họ đã quên mật khẩu hay không bằng một câu hỏi bảo mật. Nếu trả lời đúng, thì cho phép họ thay đổi mật khẩu.

- Nếu đăng nhập không thành công và trả lời sai câu hỏi bảo mật, hãy in ra thông báo "Bạn chưa phải là thành viên", hãy chọn "y" để đăng kí và chọn "n" để thoát.

Bài 8.

Mỗi đối tượng Shape trong Microsoft Word đều có có các thuộc tính: màu viền, màu nền, nội dung văn bản bên trong, thứ bậc, tình trạng đang được chọn hay không và các phương thức khởi tạo, thay đổi giá trị cho mỗi thuộc tính (phương thức setter).

Hãy tạo ra một mảng 10 phần tử Shape. Các giá trị thứ bậc không được trùng nhau (và phân bố từ 0-9). Trong 10 đối tượng này, tại mỗi thời điểm, chỉ có đúng một đối tượng đang ở tình trạng chọn. Nếu đối tượng ở tình trạng chọn, thì ta mới có quyền thay đổi giá trị cho nó.

Hãy bổ sung thêm các phương thức cần thiết để thực hiện các yêu cầu trên.

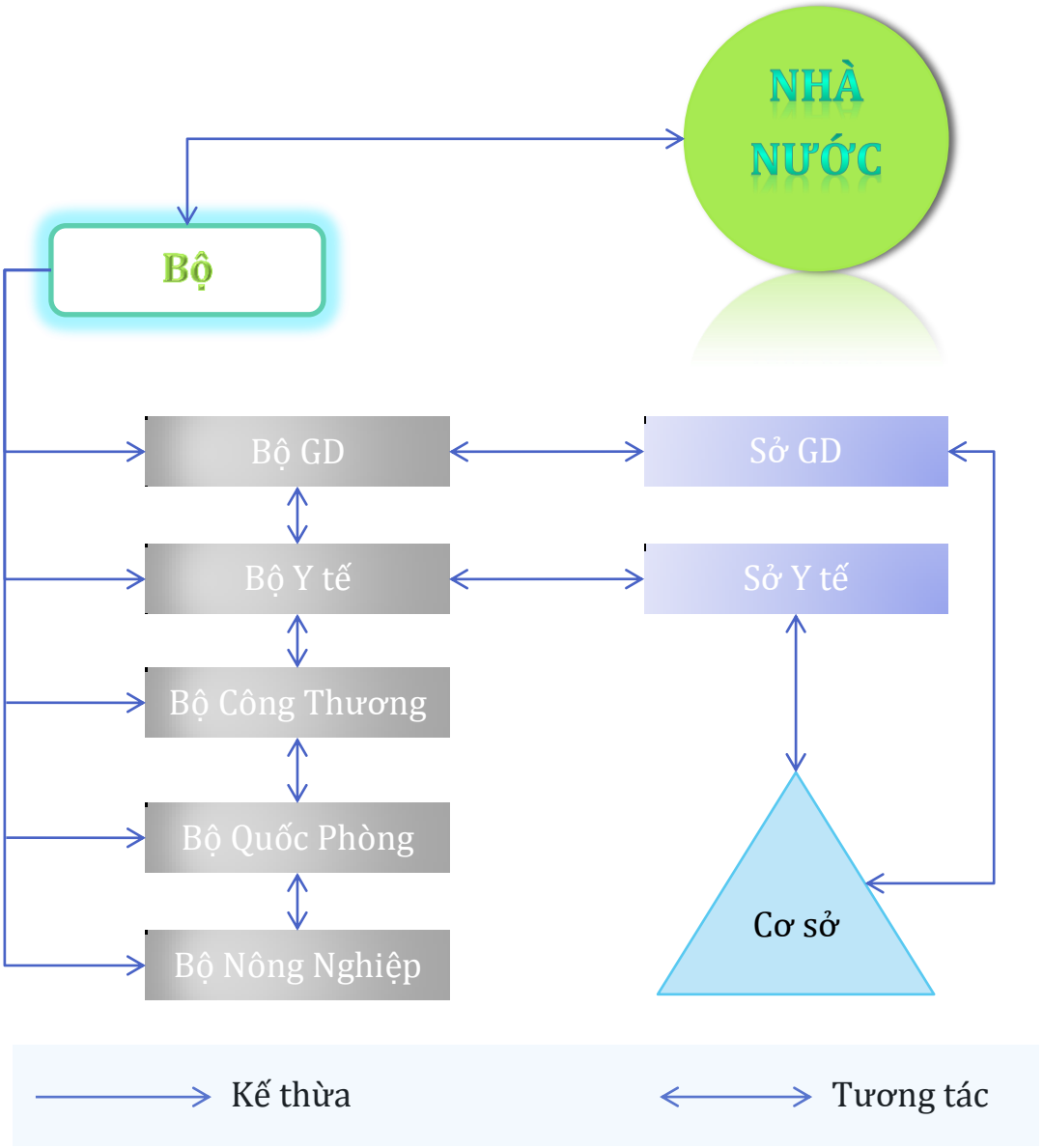
Bài 9.

Hãy phân tích mô hình quản lý sinh viên trong trường đại học. Từ mô hình phân tích được, hãy xây dựng chương trình quản lý sinh viên. Trong mô hình này, yêu cầu quản lý không dưới 10 lớp đối tượng.

Gợi ý: Các lớp đối tượng trong mô hình này bao gồm – Sinh viên, Giáo viên chủ nhiệm, Phòng Công tác học sinh sinh viên, Phòng Đào tạo, Phòng Tài chính, Khoa chuyên môn, Đoàn TN, Lớp, Phòng học, Môn học…

Bài 10.

Hãy phân tích theo hướng đối tượng mô hình quản lý Nhà nước thu nhỏ được cho bên dưới đây. Từ mô hình phân tích đó, hãy xây dựng chương trình để quản lý Nhà nước thu nhỏ. Biết rằng, tương tác giữa các đối tượng có tính bắc cầu.



Hình 23 – Mô hình tương tác quản lý Nhà nước.

BÀI TẬP LỚN

Dự án 1. Phân tích, thiết kế và xây dựng chương trình quản lý thư viện. Trong chương trình này, cần đảm bảo các chức năng sau đây:

- Trong mô hình quản lý này cần đảm bảo các thông tin: mã sách, tên sách, tên tác giả, năm xuất bản, nhà xuất bản, số trang, giá và số lượng.

- Cho phép bổ sung thêm một hoặc nhiều quyển sách.

- Cập nhập lại số lượng sách khi có người mượn/trả sách.

- Tìm kiếm sách theo tên tác giả hoặc tên sách.

- Kiểm tra tình trạng sách còn hay không.

- Chương trình thực thi trên màn hình Console. Có menu tùy chọn. Dữ liệu được lưu trữ vào tập tin data.dat dưới dạng mã hóa nhị phân.

Dự án 2. Phân tích, thiết kế và xây dựng chương trình quản lý website bán máy tính. Trong chương trình trình này, cần đảm bảo các chức năng sau đây:

- Trong mô hình quản lý này, cần đảm bảo các thông tin: mã hàng, màu sắc, nước sản xuất, hãng, giá tiền, số lượng, thời gian bảo hành, có cài đặt hệ điều hành hay không, địa chỉ khác hàng, số điện thoại khách hàng.

- Tìm kiếm mặt hàng theo tên hãng, giá tiền và nước sản xuất.

- Kiểm tra tình trạng còn hàng hay không.

- Bổ sung (nhập thêm hàng) hoặc xóa bỏ (bán hàng).

- Chương trình thực thi trên màn hình Console. Có menu tùy chọn. Dữ liệu được lưu trữ vào tập tin data.dat dưới dạng mã hóa nhị phân.

Dự án 3. Phân tích, thiết kế và xây dựng chương trình quản lý nhân viên trong công ty. Trong chương trình trình này, cần đảm bảo các chức năng sau đây:

- Trong mô hình quản lý này, cần đảm bảo các thông tin: mã nhân viên, họ tên nhân viên, ngày tháng năm sinh, hệ số lương, năm bắt đầu công tác, tình trạng hôn nhân, bộ phận làm việc.

- Tìm kiếm nhân viên theo họ tên.

- Thống kê số lượng nhân viên theo bộ phận làm việc

- Bổ sung (tuyển dụng) hoặc xóa bỏ (kết thúc hợp đồng).

- Chương trình thực thi trên màn hình Console. Có menu tùy chọn. Dữ liệu được lưu trữ vào tập tin data.dat dưới dạng mã hóa nhị phân.

Dự án 4. Phân tích, thiết kế và xây dựng game FarmVille (một game nổi tiếng trên Facebook). Trong game này, cần đảm bảo các chức năng sau đây:

- Trong mô hình quản lý này, cần đảm bảo các thông tin: người chơi – email, tên người chơi, tổng số tiền. Các đối tượng trong game: tên đối tượng, trị giá, thời gian khởi tạo, thời gian thu hoạch.

- Tìm kiếm người chơi theo họ tên.

- Thống kê số tiền thu được của một người chơi.

- Bổ sung người chơi hoặc đối tượng trong game.

- Khi thu hoạch một đối tượng, thì trị giá của đối tượng sẽ được cập nhập vào cho tổng tiền của người chơi, đồng thời đối tượng cũng sẽ bị hủy. Đối tượng chỉ có thể được thu hoạch nếu: thời gian hiện tại – thời gian khởi tạo >= thời gian thu hoạch.

- Chương trình có menu điều khiển, không yêu cầu tạo giao diện đồ họa.

Ghi chú: Các sinh viên nộp đủ bài thực hành vào trước buổi thứ 5 sẽ có cơ hội nhận được bài tập lớn. Khi nhận được bài tập lớn, ngoài yêu cầu bổ sung của giảng viên, sinh viên cần thực thi thêm các yêu cầu sau:

- Phân tích mô hình lên giấy (nộp bản in).

- Nộp chương trình hoàn chỉnh. Trong đó, chương trình hoàn chỉnh và tập tin word (*.doc; *.docx) phải được ghi lên đĩa CD, bên ngoài đĩa có ghi: tên sinh viên, lớp và "Bài tập lớn: Lập trình hướng đối tượng C++. Giảng viên hướng dẫn: ….".

DANH SÁCH HÌNH

TRA CỨU TỪ KHÓA

MỘT SỐ THUẬT NGỮ ANH-VIỆT ĐƯỢC SỬ DỤNG TRONG GIÁO TRÌNH

Nguyên bản tiếng Anh	Dịch sang tiếng Việt
Abstract base class	Lớp cơ sở trừu tượng
Abstraction	Tính trừu tượng
Arithmetic operators	Toán tử số học
Assignment operators	Toán tử gán
Base class	Lớp cơ sở/ Lớp cha
Bitwise operators	Toán tử dịch bit
Child class/SubClass	Lớp con
Class	Lớp
Comma operators	Toán tử phân tách
Compound assignment operator	Toán tử gán hợp nhất
Conditional operators	Toán tử điều kiện
Encapsulation	Tính đóng gói
Exception	Ngoại lệ
Explicit type casting operators	Toán tử chuyển đổi kiểu dữ liệu
Increase and decrease operators	Toán tử tăng giảm
Information hiding	Che dấu/ẩn dấu thông tin
Inheritance	Tính thừa kế/ Tính kế thừa
Instance	Sự thể hiện
Logical operators	Toán tử logic
Multiple inheritance	Tính đa thừa kết/Tính đa kế thừa
Object	Đối tượng
Operator	Toán tử
Operator overloading	Chồng chất toán tử
Overload	Chồng chất
Override	Quá tải
Polymorphism	Tính đa hình
Prototype	Nguyên mẫu
Pure virtual function	Hàm ảo thuần túy
Reference	Tham chiếu
Relational and equality operators	Toán tử quan hệ và so sánh

TÀI LIỆU THAM KHẢO

[1]. http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp // Mục: XL C/C++ V8.0 for AIX // Language Reference
[2]. http://msdn.microsoft.com/en-us/library/3bstk3k5%28v=VS.80%29.aspx
[3]. Ivor Horton// Beginning Visual C++ 2010// Wrox pub.
[4]. C++ for Mathematicians// An introduction for Student and Professional//Edward Scheinerman//Chapman & Hall/CRC.