TRƯỜNG ĐẠI HỌC KỸ THUẬT CÔNG NGHIỆP
KHOA ĐIỆN TỬ

BÀI GIẢNG MÔN: LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

CHƯƠNG 1

CÁC KHÁI NIỆM CƠ SỞ CỦA LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

1.1. Phương pháp tiếp cận của lập trình truyền thống

1.1.1 Lập trình tuyến tính

Lập trình tuyến tính còn gọi là lập trình phi cấu trúc là phương pháp lập trình theo lối tuần tự. Chương trình sẽ được thực hiện tuần tự từ đầu đến cuối, lệnh này kế tiếp lệnh kia cho đến khi kết thúc chương trình. Do đó lập trình tuyến tính giải quyết các bài toán tương nhỏ, đối đơn giản

- **Đặc điểm:**
  - Chỉ gồm một chương trình chính
  - Gồm một dãy tuần tự các câu lệnh
  - Chương trình ngắn, ít hơn 100 dòng
- **Ưu điểm**: Do tính đơn giản, lập trình tuyến tính có ưu điểm là chương trình đơn giản, dễ hiểu. Lập trình tuyến tính được ứng dụng cho các chương trình đơn giản.
- **Nhược điểm:**
  - Không sử dụng lại được các đoạn mã
  - Không có khả năng kiểm soát phạm vi truy xuất dữ liệu
  - Mọi dữ liệu trong chương trình là toàn cục
  - Dữ liệu có thể bị sửa đổi ở bất cứ vị trí nào trong chương trình
  - Không đáp ứng được việc triển khai phần mềm

Ngày nay, lập trình tuyến tính chỉ tồn tại trong phạm vi các modul nhỏ nhất của các phương pháp lập trình khác. Ví dụ trong một chương trình con của lập trình cấu trúc, các lệnh cũng được thực hiện theo tuần tự từ đầu đến cuối chương trình con.

1.1.2 Lập trình cấu trúc

Trong lập trình hướng cấu trúc, chương trình chính được chia nhỏ thành các chương trình con và mỗi chương trình con thực hiện một công việc xác định. Chương trình chính sẽ gọi đến chương trình con theo một giải thuật, hoặc một cấu trúc được xác định trong chương trình chính.

Các ngôn ngữ lập trình cấu trúc phổ biến là Pascal, C và C++. Riêng C++ ngoài việc có đặc trưng của lập trình cấu trúc do kế thừa từ C, còn có đặc trưng của lập trình hướng đối tượng. Cho nên C++ còn được gọi là ngôn ngữ lập trình nửa cấu trúc, nửa hướng đối tượng.

- ***Đặc trưng***

Đặc trưng cơ bản nhất của lập trình cấu trúc thể hiện ở mối quan hệ:

Chương trình = Cấu trúc dữ liệu + Giải thuật

Trong đó:

Cấu trúc dữ liệu là cách tổ chức dữ liệu, cách mô tả bài toán dưới dạng ngôn ngữ lập trình

Giải thuật là một quy trình để thực hiện một công việc xác định

Trong chương trình, giải thuật có quan hệ phụ thuộc vào cấu trúc dữ liệu:

- Một cấu trúc dữ liệu chỉ phù hợp với một số hạn chế các giải thuật.
- Nếu thay đổi cấu trúc dữ liệu thì phải thay đổi giải thuật cho phù hợp.
- Một giải thuật thường phải đi kèm với một cấu trúc dữ liệu nhất định.

- ***Tính chất***

- Mỗi chương trình con có thể được gọi thực hiện nhiều lần trong một chương trình chính.
- Các chương trình con có thể được gọi đến để thực hiện theo một thứ tự bất kì, tuỳ thuộc vào giải thuật trong chương trình chính mà không phụ thuộc vào thứ tự khai báo của các chương trình con.
- Các ngôn ngữ lập trình cấu trúc cung cấp một số cấu trúc lệnh điều khiển chương trình.

- ***Ưu điểm***

- Chương trình sáng sủa, dễ hiểu, dễ theo dõi.
- Tư duy giải thuật rõ ràng.

- ***Nhược điểm***

- Lập trình cấu trúc không hỗ trợ việc sử dụng lại mã nguồn: Giải thuật luôn phụ thuộc chặt chẽ vào cấu trúc dữ liệu, do đó, khi thay đổi cấu trúc dữ liệu, phải thay đổi giải thuật, nghĩa là phải viết lại chương trình.
- Không phù hợp với các phần mềm lớn: tư duy cấu trúc với các giải thuật chỉ phù hợp với các bài toán nhỏ, nằm trong phạm vi một modul của chương trình. Với dự án phần mềm lớn, lập trình cấu trúc tỏ ra không hiệu quả trong việc giải quyết mối quan hệ vĩ mô giữa các modul của phần mềm.

1.1.2. Nhược điểm lập trình truyền thống

Cách tiếp cận lập trình truyền thống là lập trình hướng thủ tục (LTHTT). Theo cách tiếp cận này thì một hệ thống phần mềm được xem như là dãy các công việc cần thực hiện như đọc dữ liệu, tính toán, xử lý, lập báo cáo và in ấn kết quả v.v... Mỗi công việc đó sẽ được thực hiện bởi một số hàm nhất định. Như vậy trọng tâm của cách tiếp cận này là các hàm chức năng. LTHTT sử dụng kỹ thuật phân rã hàm chức năng theo cách tiếp cận trên xuống (top-down) để tạo ra cấu trúc phân cấp. Các ngôn

ngữ lập trình bậc cao như COBOL, FORTRAN, PASCAL, C, v.v..., là những ngôn ngữ lập trình hướng thủ tục. Những nhược điểm chính của LTHTT là:

- Chương trình khó kiểm soát
- Khó khăn trong việc bổ sung, nâng cấp chương trình
- Khi thay đổi, bổ sung dữ liệu dùng chung thì phải thay đổi gần như tất cả thủ tục/hàm liên quan
- Khả năng sử dụng lại các đoạn mã chưa nhiều
- Không mô tả đầy đủ, trung thực hệ thống trong thực tế

1.1.3. Tiếp cận hướng đối tượng

Trong thế giới thực, chung quanh chúng ta là những đối tượng, đó là các thực thể có mối quan hệ với nhau. Ví dụ các *phòng* trong một công ty kinh doanh được xem như những đối tượng. Các *phòng* ở đây có thể là: phòng quản lý, phòng bán hàng, phòng kế toán, phòng tiếp thị,... Mỗi *phòng* ngoài những cán bộ đảm nhiệm những công việc cụ thể, còn có những dữ liệu riêng như thông tin về nhân viên, doanh số bán hàng, hoặc các dữ liệu khác có liên quan đến bộ phận đó. Việc phân chia các phòng chức năng trong công ty sẽ tạo điều kiện dễ dàng cho việc quản lý các hoạt động. Mỗi nhân viên trong phòng sẽ điều khiển và xử lý dữ liệu của phòng đó. Ví dụ phòng kế toán phụ trách về lương bổng nhân viên trong công ty. Nếu bạn đang ở bộ phận tiếp thị và cần tìm thông tin chi tiết về lương của đơn vị mình thì sẽ gởi yêu cầu về phòng kế toán. Với cách làm này bạn được đảm bảo là chỉ có nhân viên của bộ phận kế toán được quyền truy cập dữ liệu và cung cấp thông tin cho bạn. Điều này cũng cho thấy rằng, không có người nào thuộc bộ phận khác có thể truy cập và thay đổi dữ liệu của bộ phận kế toán. Khái niệm như thế về đối tượng hầu như có thể được mở rộng đối với mọi lĩnh vực trong đời sống xã hội và hơn nữa - đối với việc tổ chức chương trình. Mọi ứng dụng có thể được định nghĩa như một tập các thực thể - hoặc các đối tượng, sao cho quá trình tái tạo những suy nghĩa của chúng ta là gần sát nhất về thế giới thực.

1.1.4. Lập trình hướng đối tượng

Lập trình hướng đối tượng (Object Oriented Programming - OOP) là phương pháp lập trình lấy đối tượng làm nền tảng để xây dựng thuật giải, xây dựng chương trình. Đối tượng được xây dựng trên cơ sở gắn cấu trúc dữ liệu với các phương thức (các hàm/thủ tục) sẽ thể hiện được đúng cách mà chúng ta suy nghĩ, bao quát về thế giới thực. LTHĐT cho phép ta kết hợp những tri thức bao quát về các quá trình với những khái niệm trừu tượng được sử dụng trong máy tính.

Điểm căn bản của phương pháp LTHĐT là thiết kế chương trình xoay quanh dữ liệu của hệ thống. Nghĩa là các thao tác xử lý của hệ thống được gắn liền với dữ liệu

và như vậy khi có sự thay đổi của cấu trúc dữ liệu thì chỉ ảnh hưởng đến một số ít các phương thức xử lý liên quan.

LTHĐT không cho phép dữ liệu chuyển động tự do trong hệ thống. Dữ liệu được gắn chặt với từng phương thức thành các vùng riêng mà các phương thức đó tác động lên và nó được bảo vệ để cấm việc truy nhập tùy tiện từ bên ngoài. LTHĐT cho phép phân tích bài toán thành tập các thực thể được gọi là các đối tượng và sau đó xây dựng các dữ liệu cùng với các phương thức xung quanh các đối tượng đó.

Tóm lại LTHĐT có những đặc tính chủ yếu như sau:

1. Tập trung vào dữ liệu thay cho các phương thức.
2. Chương trình được chia thành các lớp đối tượng.
3. Các cấu trúc dữ liệu được thiết kế sao cho đặc tả được các đối tượng.
4. Các phương thức xác định trên các vùng dữ liệu của đối tượng được gắn với nhau trên cấu trúc dữ liệu đó.
5. Dữ liệu được bao bọc, che dấu và không cho phép các thành phần bên ngoài truy nhập tự do.
6. Các đối tượng trao đổi với nhau thông qua các phương thức.
7. Dữ liệu và các phương thức mới có thể dễ dàng bổ sung vào đối tượng nào đó khi cần thiết.
8. Chương trình được thiết kế theo cách tiếp cận bottom-up (dưới -lên).

1.2. Các khái niệm cơ bản của lập trình hướng đối tượng

1.2.1. Đối tượng

Trong thế giới thực, khái niệm đối tượng được hiểu như là một thực thể, nó có thể là người, vật hoặc một bảng dữ liệu cần xử lý trong chương trình,...

Trong lập trình hướng đối tượng, tất cả các thực thể trong hệ thống đều được coi là các đối tượng cụ thể. Đối tượng là một thực thể hoạt động khi chương trình đang chạy, đối tượng là biến thể hiện của *lớp*.

Ví dụ: Trong bài toán quản lí nhân viên của một văn phòng, mỗi nhân viên trong văn phòng được coi là một đối tượng. Chẳng hạn, nhân viên tên là "Minh", 25 tuổi làm ở phòng hành chính là một đối tượng.

Một đối tượng là một thực thể đang tồn tại trong hệ thống và được xác định bằng ba yếu tố:

- Định danh đối tượng: xác định duy nhất cho mỗi đối tượng trong hệ thống, nhằm phân biệt các đối tượng với nhau.
- Trạng thái của đối tượng: là sự tổ hợp của các giá trị của các thuộc tính mà đối tượng đang có.

- Hoạt động của đối tượng: là các hành động mà đối tượng có khả năng thực hiện được.

Để biểu diễn đối tượng trong lập trình hướng đối tượng, người ta trừu tượng hoá đối tượng để tạo nên khái niệm lớp đối tượng.

1.2.2. Lớp

Trong lập trình hướng đối tượng, đối tượng là một thực thể cụ thể, tồn tại trong hệ thống. Trong khi đó, lớp là một khái niệm trừu tượng, dùng để chỉ một tập hợp các đối tượng có mặt trong hệ thống.

Ví dụ: Trong bài toán quản lí nhân viên của một văn phòng, mỗi nhân viên trong văn phòng được coi là một đối tượng. Nhưng khái niệm "Nhân viên" là một lớp đối tượng dùng để chỉ chung chung các nhân viên của văn phòng.

Lưu ý:

- Lớp là một khái niệm, mang tính trừu tượng, dùng để biểu diễn một tập các đối tượng.
- Đối tượng là một thể hiện cụ thể của lớp, là một thực thể tồn tại trong hệ thống.

Lớp được dùng để biểu diễn đối tượng, cho nên lớp cũng có thuộc tính và phương thức:

- Thuộc tính của lớp tương ứng với thuộc tính của các đối tượng.
- Phương thức của lớp tương ứng với các hành động của đối tượng.

Ví dụ, lớp xe ô tô được mô tả bằng các thuộc tính và phương thức:

Lớp Xe ô tô

Thuộc tính:

Nhãn hiệu xe

Màu xe

Giá xe

Công suất xe (mã lực)

Phương thức:

Khởi động xe

Chạy xe

Dừng xe

Tắt máy

Lưu ý:

Một lớp có thể có một trong các khả năng sau:

- Hoặc chỉ có thuộc tính, không có phương thức.
- Hoặc chỉ có phương thức, không có thuộc tính.

- Hoặc có cả thuộc tính và phương thức, trường hợp này là phổ biến nhất.
- Đặc biệt, lớp không có thuộc tính và phương thức nào là các lớp trừu tượng. Các lớp này không có đối tượng tương ứng.

Lớp và đối tượng, mặc dù có mối liên hệ tương ứng lẫn nhau, nhưng bản chất lại khác nhau:

- Lớp là sự trừu tượng hoá của các đối tượng. Trong khi đó, đối tượng là một thể hiện của lớp.
- Đối tượng là một thực thể cụ thể, có thực, tồn tại trong hệ thống. Trong khi đó, lớp là một khái niệm trừu tượng, chỉ tồn tại ở dạng khái niệm để mô tả các đặc tính chung của một số đối tượng.
- Tất cả các đối tượng thuộc về cùng một lớp có cùng các thuộc tính và các phương thức.
- Một lớp là một nguyên mẫu của một đối tượng. Nó xác định các hành động khả thi và các thuộc tính cần thiết cho một nhóm các đối tượng cụ thể.

Nói chung, lớp là khái niệm tồn tại khi phát triển hệ thống, mang tính khái niệm, trừu tượng. Trong khi đó, đối tượng là một thực thể cụ thể tồn tại khi hệ thống đang hoạt động.

Chú ý: Trong LTHĐT thì lớp là khái niệm tĩnh, có thể nhận biết ngay từ văn bản chương trình, ngược lại đối tượng là khái niệm động, nó được xác định trong bộ nhớ của máy tính, nơi đối tượng chiếm một vùng bộ nhớ lúc thực hiện chương trình. Đối tượng được tạo ra để xử lý thông tin, thực hiện nhiệm vụ được thiết kế, sau đó bị hủy bỏ khi đối tượng đó hết vai trò.

1.2.3. Trừu tượng hóa dữ liệu và bao gói thông tin

2.1.3 Trừu tượng hoá đối tượng theo chức năng

Trừu tượng hoá đối tượng theo chức năng chính là quá trình mô hình hoá phương thức của lớp dựa trên các hành động của các đối tượng. Quá trình này được tiến hành như sau:

- Tập hợp tất cả các hành động có thể có của các đối tượng.
- Nhóm các đối tượng có các hoạt động tương tự nhau, loại bỏ bớt các hoạt động cá biệt, tạo thành một nhóm chung.
- Mỗi nhóm đối tượng đề xuất một lớp tương ứng.
- Các hành động chung của nhóm đối tượng sẽ cấu thành các phương thức của lớp tương ứng.

Ví dụ, trong bài toán quản lí cửa hàng bán ô tô. Mỗi ô tô có mặt trong của hàng là một đối tượng. Mặc dù mỗi chiếc xe có một số đặc điểm khác nhau về nhãn hiệu, giá xe, màu sắc… nhưng có chung các hành động của một chiếc xe ô tô là:

- Có thể khởi động máy.
- Có thể chạy.
- Có thể dừng lại.
- Có thể tắt máy.

Ngoài ra, một số ít xe có thể thực hiện một số hành động cá biệt như:

- Có thể giấu đèn pha
- Có thể tự bật đèn pha
- Có thể tự động phát tín hiệu báo động.

Tuy nhiên, không phải xe nào cũng thực hiện được các hành động này. Cho nên ta loại bỏ các hành động cá biệt của một số xe, chỉ giữ lại các hành động chung nhất, để mô hình thành các phương thức của đối tượng xe ô tô tương ứng với các hành động chung nhất của các xe ô tô.

Lớp Xe ô tô

Phương thức:

Khởi động xe

Chạy xe

Dừng xe

Tắt máy

2.1.4 Trừu tượng hoá đối tượng theo dữ liệu

Trừu tượng hoá đối tượng theo dữ liệu chính là quá trình mô hình hoá các thuộc tính của lớp dựa trên các thuộc tính của các đối tượng tương ứng. Quá trình này được tiến hành như sau:

- Tập hợp tất cả các thuộc tính có thể có của các đối tượng.
- Nhóm các đối tượng có các thuộc tính tương tự nhau, loại bỏ bớt các thuộc tính cá biệt, tạo thành một nhóm chung.
- Mỗi nhóm đối tượng đề xuất một lớp tương ứng.
- Các thuộc tính chung của nhóm đối tượng sẽ cấu thành các thuộc tính tương ứng của lớp được đề xuất.

Ví dụ, trong bài toán quản lí cửa hàng bán ô tô. Mỗi ô tô có mặt trong của hàng là một đối tượng. Mặc dù mỗi chiếc xe có một số đặc điểm khác nhau về nhãn hiệu, giá xe, màu sắc… nhưng có chung các thuộc tính của một chiếc xe ô tô là:

- Các xe đều có nhãn hiệu.
- Các xe đều có màu sắc
- Các xe đều có giá bán
- Các xe đều có công suất động cơ

Ngoài ra, một số ít xe có thể có thêm các thuộc tính:

- Có xe có thể có dàn nghe nhạc
- Có xe có thể có màn hình xem ti vi
- Có xe có lắp kính chống nắng, chống đạn…

Tuy nhiên, đây là các thuộc tính cá biệt của một số đối tượng xe, nên không được đề xuất thành thuộc tính của lớp ô tô. Do đó, ta mô hình lớp ô tô với các thuộc tính chung nhất của các ô tô.

Lớp Xe ô tô

Thuộc tính:

Nhãn hiệu xe

Màu xe

Giá xe

Công suất xe (mã lực)

Ưu điểm của việc trừu tượng hóa

Những ưu điểm của việc trừu tượng hóa là:

- Tập trung vào vấn đề cần quan tâm
- Xác định những đặc tính thiết yếu và những hành động cần thiết
- Giảm thiểu những chi tiết không cần thiết

Việc trừu tượng hóa dữ liệu là cần thiết, bởi vì không thể mô tả tất cả các hành động và các thuộc ính của một thực thể. Vấn đề mấu chốt là tập trung đến những hành vi cốt yếu và áp dụng chúng trong ứng dụng.

Trừu tượng hóa là cách biểu diễn những đặc tính chính và bỏ qua những chi tiết vụn vặt hoặc những giải thích. Khi xây dựng các lớp, ta phải sử dụng khái niệm trừu tượng hóa. Ví dụ ta có thể định nghĩa một lớp để mô tả các đối tượng trong không gian hình học bao gồm các thuộc tính trừu tượng như là kích thước, hình dáng, màu sắc và các phương thức xác định trên các thuộc tính này.

Việc đóng gói dữ liệu và các phương thức vào một đơn vị cấu trúc lớp được xem như một nguyên tắc *bao gói thông tin*. Dữ liệu được tổ chức sao cho thế giới bên ngoài (các đối tượng ở lớp khác) không truy nhập vào, mà chỉ cho phép các phương thức trong cùng lớp hoặc trong những lớp có quan hệ kế thừa với nhau mới được quyền truy nhập. Chính các phương thức của lớp sẽ đóng vai trò như là giao diện giữa dữ liệu của đối tượng và phần còn lại của chương trình. Nguyên tắc bao gói dữ liệu để ngăn cấm sự truy nhập trực tiếp trong lập trình được gọi là sự che giấu thông tin.

1.2.4. Kế thừa

Kế thừa là quá trình mà các đối tượng của lớp này được quyền sử dụng một số tính chất của các đối tượng của lớp khác. Sự kế thừa cho phép ta định nghĩa một lớp mới trên cơ sở các lớp đã tồn tại. Lớp mới này, ngoài những thành phần được kế thừa,

sẽ có thêm những thuộc tính và các hàm mới. Nguyên lý kế thừa hỗ trợ cho việc tạo ra cấu trúc phân cấp các lớp.

Xét trường hợp bài toán quản lí nhân sự và sinh viên của một trường đại học. Khi đó, ta có hai lớp đối tượng chính là lớp Nhân viên và lớp Sinh viên:

Lớp Nhân viên

Thuộc tính:

Tên

Ngày sinh

Giới tính

Lương

Phương thức:

Nhập/xem tên

Nhập/xem ngày sinh

Nhập/xem giới tính

Nhập/xem lương

Lớp Sinh viên

Thuộc tính:

Tên

Ngày sinh

Giới tính

Lớp

Phương thức:

Nhập/xem tên

Nhập/xem ngày sinh

Nhập/xem giới tính

Nhập/xem lớp

Ta nhận thấy rằng hai lớp này có một số thuộc tính và phương thức chung: tên, ngày sinh, giới tính. Tuy nhiên, không thể loại bỏ các thuộc tính cá biệt để gộp chúng thành một lớp duy nhất, vì các thuộc tính lương nhân viên và lớp của sinh viên là cần thiết cho việc quản lí. Vấn đề nảy sinh như sau:

- Ta phải viết mã trùng nhau đến hai lần cho các phương thức: nhập/xem tên, nhập/xem ngày sinh, nhập/xem giới tính. Rõ rang điều này rất tốn công sức.
- Nếu khi có sự thay đổi về kiểu dữ liệu, chẳng hạn kiểu ngày sinh được quản lí trong hệ thống, ta phải sửa lại chương trình hai lần.

Để tránh rắc rối do các vấn đề nảy sinh như vậy, lập trình hướng đối tượng sử dụng kỹ thuật kế thừa nhằm nhóm các phần giống nhau của các lớp thành một lớp

mới, sau đó cho các lớp ban đầu kế thừa lại lớp được tạo ra. Như vậy, mỗi lớp thừa kế (lớp dẫn xuất, lớp con) đều có các thuộc tính và phương thức của lớp bị thừa kế (lớp cơ sở, lớp cha).

1.2.5. Tương ứng bội

Tương ứng bội là khả năng của một khái niệm (chẳng hạn các phép toán) có thể sử dụng với nhiều chức năng khác nhau. Ví dụ, phép + có thể biểu diễn cho phép "cộng" các số nguyên (int), số thực (float), số phức (complex) hoặc xâu ký tự (string) v.v... Hành vi của phép toán tương ứng bội phụ thuộc vào kiểu dữ liệu mà nó sử dụng để xử lý.

Tương ứng bội đóng vai quan trọng trong việc tạo ra các đối tượng có cấu trúc bên trong khác nhau nhưng cùng dùng chung một giao diện bên ngoài (như tên gọi).

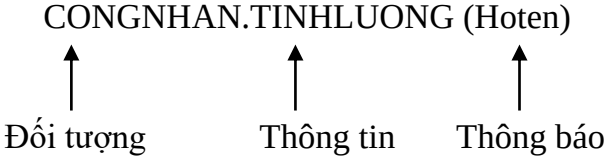
1.2.6. Liên kết động

Liên kết động là dạng liên kết các thủ tục và hàm khi chương trình thực hiện lời gọi tới các hàm, thủ tục đó. Như vậy trong liên kết động, nội dung của đoạn chương trình ứng với thủ tục, hàm sẽ không được biết cho đến khi thực hiện lời gọi tới thủ tục, hàm đó.

1.2.7. Truyền thông báo

Các đối tượng gửi và nhận thông tin với nhau giống như con người trao đổi với nhau. Chính nguyên lý trao đổi thông tin bằng cách truyền thông báo cho phép ta dễ dàng xây dựng được hệ thống mô phỏng gần hơn những hệ thống trong thế giới thực. *Truyền thông báo* cho một đối tượng là yêu cầu đối tượng thực hiện một việc gì đó. Cách ứng xử của đối tượng được mô tả bên trong lớp thông qua các phương thức.

Trong chương trình, thông báo gửi đến cho một đối tượng chính là yêu cầu thực hiện một công việc cụ thể, nghĩa là sử dụng những hàm tương ứng để xử lý dữ liệu đã được khai báo trong đối tượng đó. Vì vậy, trong thông báo phải chỉ ra được hàm cần thực hiện trong đối tượng nhận thông báo. Thông báo truyền đi cũng phải xác định tên đối tượng và thông tin truyền đi. Ví dụ, lớp CONGNHAN có thể hiện là đối tượng cụ thể được đại diện bởi Hoten nhận được thông báo cần tính lương thông qua hàm TINHLUONG đã được xác định trong lớp CONGNHAN. Thông báo đó sẽ được xử lý như sau:



Trong chương trình hướng đối tượng, mỗi đối tượng chỉ tồn tại trong thời gian nhất định. Đối tượng được tạo ra khi nó được khai báo và sẽ bị hủy bỏ khi chương trình ra khỏi miền xác định của đối tượng đó. Sự trao đổi thông tin chỉ có thể thực hiện trong thời gian đối tượng tồn tại.

1.3. Các bước cần thiết để thiết kế chương trình theo hướng đối tượng

Chương trình theo hướng đối tượng bao gồm một tập các đối tượng và mối quan hệ giữa các đối tượng với nhau. Vì vậy, lập trình trong ngôn ngữ hướng đối tượng bao gồm các bước sau:

1. Xác định các dạng đối tượng (lớp) của bài tóan.

2. Tìm kiếm các đặc tính chung (dữ liệu chung) trong các dạng đối tượng này, những gì chúng cùng nhau chia xẻ.

3. Xác định lớp cơ sở dựa trên cơ sở các đặc tính chung của các dạng đối tượng.

4. Từ lớp cơ sở, xây dựng các lớp dẫn xuất chứa các thành phần, những đặc tính không chung còn lại của các dạng đối tượng. Ngoài ra, ta còn đưa ra các lớp có quan hệ với các lớp cơ sở và lớp dẫn xuất.

1.4. Các ưu điểm của lập trình hướng đối tượng

Cách tiếp cận hướng đối tượng giải quyết được nhiều vấn đề tồn tại trong quá trình phát triển phần mềm và tạo ra được những sản phẩm phần mềm có chất lượng cao. Những ưu điểm chính của LTHĐT là:

1. Thông qua nguyên lý kế thừa, có thể loại bỏ được những đoạn chương trình lặp lại trong quá trình mô tả các lớp và mở rộng khả năng sử dụng các lớp đã được xây dựng.

2. Chương trình được xây dựng từ những đơn thể (đối tượng) trao đổi với nhau nên việc thiết kế và lập trình sẽ được thực hiện theo quy trình nhất định chứ không phải dựa vào kinh nghiệm và kỹ thuật như trước. Điều này đảm bảo rút ngắn được thời gian xây dựng hệ thống và tăng năng suất lao động.

3. Nguyên lý che giấu thông tin giúp người lập trình tạo ra được những chương trình an toàn không bị thay bởi những đoạn chương trình khác.

4. Có thể xây dựng được ánh xạ các đối tượng của bài toán vào đối tượng của chương trình.

5. Cách tiếp cận thiết kế đặt trọng tâm vào đối tượng, giúp chúng ta xây dựng được mô hình chi tiết và gần với dạng cài đặt hơn.

6. Những hệ thống hướng đối tượng dễ mở rộng, nâng cấp thành những hệ lớn hơn.

7. Kỹ thuật truyền thông báo trong việc trao đổi thông tin giữa các đối tượng giúp cho việc mô tả giao diện với các hệ thống bên ngoài trở nên đơn giản hơn.

8. Có thể quản lý được độ phức tạp của những sản phẩm phần mềm.

Không phải trong hệ thống hướng đối tượng nào cũng có tất cả các tính chất nêu trên. Khả năng có các tính chất đó còn phụ thuộc vào lĩnh vực ứng dụng của dự án tin học và vào phương pháp thực hiện của người phát triển phần mềm.

1.5. Các ngôn ngữ hướng đối tượng

Lập trình hướng đối tượng không là đặc quyền của một ngôn ngữ nào đặc biệt. Cũng giống như lập trình có cấu trúc, những khái niệm trong lập trình hướng đối tượng có thể cài đặt trong những ngôn ngữ lập trình như C hoặc Pascal,... Tuy nhiên, đối với những chương trình lớn thì vấn đề lập trình sẽ trở nên phức tạp. Những ngôn ngữ được thiết kế đặc biệt, hỗ trợ cho việc mô tả, cài đặt các khái niệm của phương pháp hướng đối tượng được gọi chung là ngôn ngữ đối tượng. Dựa vào khả năng đáp ứng các khái niệm về hướng đối tượng, ta có thể chia ra làm hai loại:

1. Ngôn ngữ lập trình dựa trên đối tượng
2. Ngôn ngữ lập trình hướng đối tượng

Lập trình dựa trên đối tượng là kiểu lập trình hỗ trợ chính cho việc bao gói, che giấu thông tin và định danh các đối tượng. Lập trình dựa trên đối tượng có những đặc tính sau:

- Bao gói dữ liệu
- Cơ chế che giấu và truy nhập dữ liệu
- Tự động tạo lập và xóa bỏ các đối tượng
- Phép toán tải bội

Ngôn ngữ hỗ trợ cho kiểu lập trình trên được gọi là ngôn ngữ lập trình dựa trên đối tượng. Ngôn ngữ trong lớp này không hỗ trợ cho việc thực hiện kế thừa và liên kết động, chẳng hạn Ada là ngôn ngữ lập trình dựa trên đối tượng.

Lập trình hướng đối tượng là kiểu lập trình dựa trên đối tượng và bổ sung thêm nhiều cấu trúc để cài đặt những quan hệ về kế thừa và liên kết động. Vì vậy đặc tính của LTHĐT có thể viết một cách ngắn gọn như sau:

Các đặc tính dựa trên đối tượng + kế thừa + liên kết động.

Ngôn ngữ hỗ trợ cho những đặc tính trên được gọi là ngôn ngữ LTHĐT, ví dụ như C++, Smalltalk, Object Pascal v.v...

Việc chọn một ngôn ngữ để cài đặt phần mềm phụ thuộc nhiều vào các đặc tính và yêu cầu của bài toán ứng dụng, vào khả năng sử dụng lại của những chương trình đã có và vào tổ chức của nhóm tham gia xây dựng phần mềm.

1.6. Một số ứng dụng của LTHĐT

LTHĐT đang được ứng dụng để phát triển phần mềm trong nhiều lĩnh vực khác nhau. Trong số đó, có ứng dụng quan trọng và nổi tiếng nhất hiện nay là hệ điều hành

Windows của hãng Microsoft đã được phát triển dựa trên kỹ thuật LTHĐT. Một số những lĩnh vực ứng dụng chính của kỹ thuật LTHĐT bao gồm:

+ Những hệ thống làm việc theo thời gian thực.
+ Trong lĩnh vực mô hình hóa hoặc mô phỏng các quá trình
+ Các cơ sở dữ liệu hướng đối tượng.
+ Những hệ siêu văn bản, multimedia
+ Lĩnh vực trí tuệ nhân tạo và các hệ chuyên gia.
+ Lập trình song song và mạng nơ-ron.
+ Những hệ tự động hóa văn phòng và trợ giúp quyết định.

...

CHƯƠNG 2

LỚP

Lớp là khái niệm trung tâm của lập trình hướng đối tượng, nó là sự mở rộng của các khái niệm cấu trúc (struct) của C. Ngoài các thành phần dữ liệu, lớp còn chứa các thành phần hàm, còn gọi là phương thức (method) hoặc hàm thành viên (member function). Lớp có thể xem như một kiểu dữ liệu các biến, mảng đối tượng. Từ một lớp đã định nghĩa, có thể tạo ra nhiều đối tượng khác nhau, mỗi đối tượng có vùng nhớ riêng.

Chương này sẽ trình bày cách định nghĩa lớp, cách xây dựng phương thức, giải thích về phạm vi truy nhập, sử dụng các thành phần của lớp, cách khai báo biến, mảng cấu trúc, lời gọi tới các phương thức .

2.1. Định nghĩa lớp

Cú pháp: Lớp được định nghĩa theo mẫu :

```
class tên_lớp
 {
 private:    [Khai báo các thuộc tính]
        [Định nghĩa các hàm thành phần (phương thức)]
 public :    [Khai báo các thuộc tính]
             [Định nghĩa các hàm thành phần (phương thức)]
  } ;
```

Thuộc tính của lớp được gọi là dữ liệu thành phần và hàm được gọi là phương thức hoặc hàm thành viên. Thuộc tính và hàm được gọi chung là các thành phần của lớp. Các thành phần của lớp được tổ chức thành hai vùng: vùng sở hữu riêng (private) và vùng dùng chung (public) để quy định phạm vi sử dụng của các thành phần. Nếu không quy định cụ thể (không dùng các từ khóa private và public) thì C++ hiểu đó là private. Các thành phần private chỉ được sử dụng bên trong lớp (trong thân của các hàm thành phần). Các thành phần public được phép sử dụng ở cả bên trong và bên ngoài lớp. Các hàm không phải là hàm thành phần của lớp thì không được phép sử dụng các thành phần này.

Khai báo các thuộc tính của lớp: được thực hiện y như việc khai báo biến. Thuộc tính của lớp không thể có kiểu chính của lớp đó, nhưng có thể là kiểu con trỏ của lớp này,

Ví dụ:

```
class A
```

```
{
    A    x;   //Không cho phép, vì x có kiểu lớp A
    A    *p ;        // Cho phép, vì p là con trỏ kiểu lớp A
} ;
```

Định nghĩa các hàm thành phần: Các hàm thành phần có thể được xây dựng bên ngoài hoặc bên trong định nghĩa lớp. Thông thường, các hàm thành phần đơn giản, có ít dòng lệnh sẽ được viết bên trong định nghĩa lớp, còn các hàm thành phần dài thì viết bên ngoài định nghĩa lớp. Các hàm thành phần viết bên trong định nghĩa lớp được viết như hàm thông thường. Khi định nghĩa hàm thành phần ở bên ngoài lớp, ta dùng cú pháp sau đây:

```
Kiểu_trả_về_của_hàm  Tên_lớp::Tên_hàm(khai báo các tham số)
                                        { [nội dung hàm]
                                          }
```

Toán tử :: được gọi là toán tử phân giải miền xác định, được dùng để chỉ ra lớp mà hàm đó thuộc vào.

Trong thân hàm thành phần, có thể sử dụng các thuộc tính của lớp, các hàm thành phần khác và các hàm tự do trong chương trình.

Chú ý :

- Các thành phần dữ liệu khai báo là private nhằm bảo đảm nguyên lý che dấu thông tin, bảo vệ an toàn dữ liệu của lớp, không cho phép các hàm bên ngoài xâm nhập vào dữ liệu của lớp .
- Các hàm thành phần khai báo là public có thể được gọi tới từ các hàm thành phần public khác trong chương trình .

2.2. Tạo lập đối tượng

Sau khi định nghĩa lớp, ta có thể khai báo các biến thuộc kiểu lớp. Các biến này được gọi là các đối tượng. Cú pháp khai báo biến đối tượng như sau:

```
        Tên_lớp    Danh_sách_biến ;
```

Đối tượng cũng có thể khai báo khi định nghĩa lớp theo cú pháp sau:

```
class tên_lớp
    {
      ...
    } <Danh_sách_biến>;
```

Mỗi đối tượng sau khi khai báo sẽ được cấp phát một vùng nhớ riêng để chứa các thuộc tính của chúng. Không có vùng nhớ riêng để chứa các hàm thành phần cho mỗi

đối tượng. Các hàm thành phần sẽ được sử dụng chung cho tất cả các đối tượng cùng lớp.

2.3. Truy nhập tới các thành phần của lớp

- Để truy nhập đến dữ liệu thành phần của lớp, ta dùng cú pháp:

Tên_đối_tượng. Tên_thuộc_tính

Cần chú ý rằng dữ liệu thành phần riêng chỉ có thể được truy nhập bởi những hàm thành phần của cùng một lớp, đối tượng của lớp cũng không thể truy nhập.

- Để sử dụng các hàm thành phần của lớp, ta dùng cú pháp:

Tên_đối_tượng. Tên_hàm (Các_khai_báo_tham_số_thực_sự)

Ví dụ 3.1

```
#include <conio.h>
#include <iostream.h>
class DIEM
    {
private :
     int x,y ;
public :
 void nhapsl( )
 {
  cout << "\n Nhap hoanh do va tung do cua diem:";
  cin >>x>>y ;
 }
  void hienthi( )
  { cout<<"\n x = " <<x<<" y = "<<y<<endl;}
     } ;
void main()
 { clrscr();
   DIEM d1;
   d1.nhapsl();
   d1.hienthi();
   getch();
 }
```

Ví dụ 3.2

```
#include <conio.h>
#include <iostream.h>
class A
```

```
  { int m,n;
    public :
     void nhap( )
{
            cout << "\n Nhap hai so nguyen : " ;
            cin>>m>>n ;
       }
     int max()
       {
       return m>n?m:n;
         }
     void hienthi( )
       {     cout<<"\n Thanh phan du lieu lon nhat x = "
<<max()<<endl;}
   };
void main ()
 { clrscr();
   A ob;
   ob.nhap();
   ob.hienthi();
   getch();
 }
```

Chú ý: Các hàm tự do có thể có các đối là đối tượng nhưng trong thân hàm không thể truy nhập đến các thuộc tính của lớp. Ví dụ giả sử đã định nghĩa lớp:

```
class DIEM
{
private :
      double x,y ; // toa do cua diem
public :
      void nhapsl()
            {
              cout << " Toa do x,y : " ;
              cin >> x>>y ;
            }
      void in()
            {
```

```
                cout << “x =””<<x<<”y=”<<y ;
            }
    } ;
```

Dùng lớp DIEM, ta xây dựng hàm tự do tính độ đài của đoạn thẳng đi qua hai điểm như sau :

```
    double  do_dai ( DIEM d1, DIEM d2 )
     {
    return sqrt(pow(d1.x-d2.x,2) + pow(d1.y-d2.y,2));
     }
```

Chương trình dịch sẽ báo báo lỗi đối với hàm này. Bởi vì trong thân hàm không cho phép sử dụng các thuộc tính d1.x,d2.x,d1.y của các đối tượng d1 và d2 thuộc lớp DIEM .

Ví dụ 2.3 Ví dụ sau minh họa việc sử dụng hàm thành phần với tham số mặc định:

```
    #include <iostream.h>
    #include <conio.h>
    class Box
{
  private:
    int dai;
    int rong;
    int cao;
public:
 int get_thetich(int lth,int wdth = 2,int ht = 3);
};
    int Box::get_thetich(int l, int w, int h)
{
    dai = l;
    rong = w;
    cao = h;
    cout<< dai<<'\t'<< rong<<'\t'<<cao<<'\t';
    return dai * rong * cao;
}
    void main()
{
    Box ob;
```

```
    int x = 10, y = 12, z = 15;
    cout <<"Dai Rong Cao Thetich\n";
    cout << ob.get_thetich(x, y, z) << "\n";
    cout << ob.get_thetich(x, y) << "\n";
    cout << ob.get_thetich(x) << "\n";
    cout << ob.get_thetich(x, 7) << "\n";
    cout << ob.get_thetich(5, 5, 5) << "\n";
      getch();
}
```

Kết quả chương trình như sau:

```
Dai Rong Cao Thetich
10 12 15 1800
10 12 3 360
10 2 3 60
10 7 3 210
5 5 5 125
```

Ví dụ 2.4 Ví dụ sau minh họa việc sử dụng hàm **inline** trong lớp:

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class phrase
{
private:
  char dongtu[10];
  char danhtu[10];
  char cumtu[25];
public:
  phrase();
  inline void set_danhtu(char* in_danhtu);
  inline void set_dongtu(char* in_dongtu);
  inline char* get_phrase(void);
};
    void phrase::phrase()
{
  strcpy(danhtu,"");
```

```
  strcpy(dongtu,"");
  strcpy(cumtu,"");
}
     inline void phrase::set_danhtu(char* in_danhtu)
{
  strcpy(danhtu, in_danhtu);
}
     inline void phrase::set_dongtu(char* in_dongtu)
{
  strcpy(dongtu, in_dongtu);
}
     inline char* phrase::get_phrase(void)
{
  strcpy(cumtu,dongtu);
  strcat(cumtu," the ");
  strcat(cumtu,danhtu);
  return cumtu;
}
     void main()
{
 phrase text;
 cout << "Cum tu la : -> " << text.get_phrase()
          << "\n";
 text.set_danhtu("file");
 cout << "Cum tu la : -> " <<
          text.get_phrase()<<"\n";
 text.set_dongtu("Save");
 cout << "Cum tu la : -> " <<
          text.get_phrase()<<"\n";
 text.set_danhtu("program");
 cout << "Cum tu la : -> " <<
          text.get_phrase()<<"\n";
}
```

Kết quả chương trình như sau:

Cum tu la : -> the

Cum tu la : -> the file

Cum tu la : -> Save the file

Cum tu la : -> Save the program

Ví dụ 3.5 Ví dụ sau minh họa việc sử dụng từ khóa const trong lớp:

```
#include <iostream.h>
#include <conio.h>
class constants
{
 private:
   int number;
 public:
   void print_it(const int data_value);
};
void constants::print_it(const int data_value)
{
   number = data_value;
   cout << number << "\n";
}
void main()
{
   constants num;
   const int START = 3;
   const int STOP = 6;
   int index;
   for (index=START; index<=STOP; index++)
  {
      cout<< "index = " ;
      num.print_it(index);
      cout<< "START = " ;
      num.print_it(START);
  }
      getch();
      }
```

Kết quả chương trình như sau:

```
index = 3
START = 3
```

index = 4
START = 3
index = 5
START = 3
index = 6
START = 3

2.4. Con trỏ đối tượng

Con trỏ đối tượng dùng để chứa địa chỉ của biến đối tượng, được khai báo như sau :

Tên_lớp * Tên_con_ trỏ ;

Ví dụ : Dùng lớp DIEM, ta có thể khai báo:

```
DIEM    *p1,  *p2,   *p3 ;  // Khai báo 3 con trỏ p1, p2, p3
DIEM    d1, d2 ;            //Khai báo hai đối tượng d1, d2
DIEM    d [20] ;            // Khai báo mảng đối tượng
```

Có thể thực hiện câu lệnh :

```
p1 = &d2 ;        //p1 chứa địa chỉ của d2, p1 trỏ tới d2
p2 =d ;           // p2 trỏ tới đầu mảng d
p3 =new DIEM //tạo một đối tượng và chứa địa chỉ của nó vào p3
```

Để truy xuất các thành phần của lớp từ con trỏ đối tượng, ta viết như sau :

Tên_con_trỏ -> Tên_thuộc_tính

Tên_con_trỏ -> Tên_hàm(các tham số thực sự)

Nếu con trỏ chứa đầu địa chỉ của mảng, có thể dùng con trỏ như tên mảng.

Ví dụ 2.6

```
#include <iostream.h>
#include <conio.h>
class mhang
 { int maso;
   float gia;
   public: void getdata(int a, float b)
         {maso= a; gia= b;}
    void show()
         { cout << "maso" << maso<< endl;
           cout << "gia" << gia<< endl;
         }
  };
const int k=5;
```

```
void main()
 { clrscr();
   mhang *p = new mhang[k];
   mhang *d = p;
   int x,i;
   float y;
   cout<<"\nNhap vao du lieu 5 mat hang :";
   for (i = 0; i <k; i++)
     { cout <<"\nNhap ma hang va don gia cho mat hang thu " <<i+1;
       cin>>x>>y;
       p -> getdata(x,y);
       p++;}
       for (i = 0; i <k; i++)
         { cout <<"\nMat hang  thu : " << i + 1 <<
               endl;
           d -> show();
           d++;
         }
   getch();
 }
```

2.5. Con trỏ this

Mỗi hàm thành phần của lớp có một tham số ẩn, đó là con trỏ **this**. Con trỏ this trỏ đến từng đối tượng cụ thể. Ta hãy xem lại hàm nhapsl() của lớp DIEM trong ví dụ ở trên:

```
void nhapsl( )
{
 cout << "\n Nhap hoanh do va tung do cua diem : ";
 cin >>x>>y;
}
```

Trong hàm này ta sử dụng tên các thuộc tính x,y một cách đơn độc. Điều này dường như mâu thuẩn với quy tắc sử dụng thuộc tính. Tuy nhiên nó được lý giải như sau: C++ sử dụng một con trỏ đặc biệt trong các hàm thành phần. Các thuộc tính viết trong hàm thành phần được hiểu là thuộc một đối tượng do con trỏ **this** trỏ tới. Như vậy hàm nhapsl() có thể viết một cách tường minh như sau:

```
void nhapsl( )
{
```

```
  cout << "\n Nhap hoanh do va tung do cua diem:" ;
  cin >>this->x>>this->y ;
 }
```

Con trỏ **this** là đối thứ nhất của hàm thành phần. Khi một lời gọi hàm thành phần được phát ra bởi một đối tượng thì tham số truyền cho con trỏ **this** chính là địa chỉ của đối tượng đó.

Ví dụ: Xét một lời gọi tới hàm nhapsl() :

```
DIEM d1 ;
d1.nhapsl();
```

Trong trường hợp này của d1 thì this =&d1. Do đó this -> x chính là d1.x và this-> y chính là d1.y

Chú ý: Ngoài tham số đặc biệt **this** không xuất hiện một cách tường minh, hàm thành phần lớp có thể có các tham số khác được khai báo như trong các hàm thông thường.

Ví dụ 3.7

```
#include <iostream.h>
#include <conio.h>
class time
{ int h,m;
public :
void nhap(int h1, int m1)
 { h= h1; m = m1;}
void hienthi(void)
{ cout <<h << " gio "<<m << " phut"<<endl;}
void tong(time, time);
};
void time::tong(time t1, time t2)
{ m= t1.m+ t2.m;       //this->m = t1.m+ t2.m;
  h= m/60;                    //this->h = this->m/60;
  m= m%60;                          //this->m = this->m%60;
  h = h+t1.h+t2.h;       //this->h = this->h + t1.h+t2.h;
}
void main()
{
 clrscr();
 time ob1, ob2,ob3;
```

```
ob1.nhap(2,45);
ob2.nhap(5,40);
ob3.tong(ob1,ob2);
cout <<"object 1 = "; ob1.hienthi();
cout <<"object 2 = "; ob2. hienthi();
cout <<"object 3 = "; ob3. hienthi();
getch();
}
```

Chương trình cho kết quả như sau :

```
object 1 = 2 gio 45 phut
object 2 = 5 gio 40 phut
object 3 = 8 gio 25 phut
```

2.6. Hàm bạn

Trong thực tế thường xãy ra trường hợp có một số lớp cần sử dụng chung một hàm. C++ giải quyết vấn đề này bằng cách dùng hàm bạn. Để một hàm trở thành bạn của một lớp, có 2 cách viết:

Cách 1: Dùng từ khóa *friend* để khai báo hàm trong lớp và xây dựng hàm bên ngoài như các hàm thông thường (không dùng từ khóa friend). Mẫu viết như sau :

```
class  A
        {
                private :
                //  Khai báo các thuộc tính
                public :
                ...
                // Khai báo các hàm bạn của lớp A
                friend void f1 (...) ;
                friend double f2 (...) ;
                ...
        }  ;
// Xây dựng các hàm f1,f2,f3
void f1 (...)
        {
                ...
        }
double f2 (...)
        {
```

```
        ...
    }
```

Cách 2: Dùng từ khóa friend để xây dựng hàm trong định nghĩa lớp . Mẫu viết như sau :

```
class  A
    {
        private :
        //  Khai báo các thuộc tính
        public :
         ...
        // Khai báo các hàm bạn của lớp A
        void f1 (...)
            {
                ...
            }
        double f2 (...)
            {
                ...
            }

    } ;
```

Hàm bạn có những tính chất sau:

- Hàm bạn không phải là hàm thành phần của lớp.
- Việc truy nhập tới hàm bạn được thực hiện như hàm thông thường.
- Trong thân hàm bạn của một lớp có thể truy nhập tới các thuộc tính của đối tượng thuộc lớp này. Đây là sự khác nhau duy nhất giữa hàm bạn và hàm thông thường.
- Một hàm có thể là bạn của nhiều lớp. Lúc đó nó có quyền truy nhập tới tất cả các thuộc tính của các đối tượng trong các lớp này. Để làm cho hàm f trở thành bạn của các lớp A, B và C ta sử dụng mẫu viết sau :

```
class A ; //Khai báo trước lớp A
class B ; // Khai báo trước lớp B
class C ; // Khai báo trước lớp C
// Định nghĩa  lớp A
class A
    {
```

```
                // Khai báo f là bạn của A
                friend void f(... )
        } ;
    // Định nghĩa lớp B
    class B
        {
                // Khai báo f là bạn của B
                friend void f(...)
        } ;
    // Định nghĩa lớp C
    class C
        {
                // Khai báo f là bạn của C
                friend void f(...)
        } ;
    // Xây dựng hàm f
    void f(...)
        {
                ...
        } ;
```

Ví dụ 2.8

```
#include <iostream.h>
#include <conio.h>
class sophuc
{float a,b;
 public :    sophuc() {}
             sophuc(float x, float y)
    {a=x; b=y;}
  friend sophuc tong(sophuc,sophuc);
  friend void  hienthi(sophuc);
};
sophuc  tong(sophuc c1,sophuc c2)
 {sophuc c3;
  c3.a=c1.a + c2.a ;
  c3.b=c1.b + c2.b ;
  return (c3);
```

```
  }
void hienthi(sophuc c)
 {cout<<c.a<<" + "<<c.b<<"i"<<endl; }
void main()
{ clrscr();
  sophuc d1 (2.1,3.4);
  sophuc d2 (1.2,2.3) ;
  sophuc d3 ;
  d3 = tong(d1,d2);
  cout<<"d1= ";hienthi(d1);
  cout<<"d2= ";hienthi(d2);
  cout<<"d3= ";hienthi(d3);
  getch();
  }
```

Chương trình cho kết quả như sau :

d1= 2.1 + 3.4i

d2= 1.2 + 2.3i

d3= 3.3 + 5.7i

Ví dụ 2.9

```
#include <iostream.h>
#include <conio.h>
class LOP1;
class LOP2
  {
    int v2;
    public:
      void nhap(int a)
    { v2=a;}
      void hienthi(void)
    { cout<<v2<<"\n";}
      friend void traodoi(LOP1 &, LOP2 &);
  };
class LOP1
  {
    int v1;
```

```
   public:
     void nhap(int a)
  { v1=a;}
     void hienthi(void)
  { cout<<v1<<"\n";}
     friend void traodoi(LOP1 &, LOP2 &);
  };
void traodoi(LOP1 &x, LOP2 &y)
  {
   int t = x.v1;
   x.v1 = y.v2;
   y.v2 = t;
  }
void main()
  {
   clrscr();
   LOP1 ob1;
   LOP2 ob2;
   ob1.nhap(150);
   ob2.nhap(200);
   cout << "Gia tri ban dau :" << "\n";
   ob1.hienthi();
   ob2.hienthi();
   traodoi(ob1, ob2); //Thuc hien hoan doi
   cout << "Gia tri sau khi thay doi:" << "\n";
   ob1.hienthi();
   ob2.hienthi();
   getch();
  }
```

Chương trình cho kết quả như sau:

```
Gia tri ban dau :
150
200
Gia tri sau khi thay doi:
200
150
```

2.7. Dữ liệu thành phần tĩnh và hàm thành phần tĩnh

2.7.1. Dữ liệu thành phần tĩnh

Dữ liệu thành phần tĩnh được khai báo bằng từ khoá static và được cấp phát một vùng nhí cố định, nó tồn tại ngay cả khi lớp chưa có một đối tượng nào cả. Dữ liệu thành phần tĩnh là chung cho cả lớp, nó không phải là riêng của mỗi đối tượng, ví dụ:

```
class A
{
private:
static int ts; // Thành phần tĩnh
int x;
...
};
A u, v; // Khai báo 2 đối tượng
```

Giữa các thành phần x và ts có sự khác nhau như sau: u.x và v.x có 2 vùng nhớ khác nhau, trong khi u.ts và v.ts chỉ là một, chúng cùng biểu thị một vùng nhớ, thành phần ts tồn tại ngay khi u và v chưa khai báo.

Để biểu thị thành phần tĩnh, ta có thể dùng tên lớp, ví dụ: A::ts

Khai báo và khởi gán giá trị cho thành phần tĩnh: Thành phần tĩnh sẽ được cấp phát bộ nhớ và khởi gán giá trị đầu bằng một câu lệnh khai báo đặt sau định nghĩa lớp theo mẫu như sau:

```
int A::ts;              // Khởi gán cho ts giá trị 0
int A::ts = 1234;       // Khởi gán cho ts giá trị 1234
```

Chú ý: Khi chưa khai báo thì thành phần tĩnh chưa tồn tại. Hãy xem chương trình sau:

Ví dụ 2.10

```
#include <conio.h>
#include <iostream.h>
class HDBH
{
private:
char *tenhang;
double tienban;
static int tshd;
static double tstienban;
public:
```

```
static void in()
{
cout <<"\n" << tshd;
cout <<"\n" << tstienban;
}
};
void main ()
{
HDBH::in();
getch();
}
```

Các thành phần tĩnh tshd và tstienban chưa khai báo, nên chưa tồn tại. Vì vậy các câu lệnh in giá trị các thành phần này trong hàm in() là không thể được. Khi dịch chương trình, sẽ nhận được các thông báo lỗi. Có thể sửa chương trình trên bằng cách đưa vào các lệnh khai báo các thành phần tĩnh tshd và tstienban như sau:

Ví dụ 2.11

```
#include <conio.h>
#include <iostream.h>
class HDBH
{
private:
int shd;
char *tenhang;
double tienban;
static int tshd;
static double tstienban;
public:
static void in()
{
cout <<"\n" <<tshd;
cout <<"\n" <<tstienban;
}
};
int HDBH::tshd=5
double HDBH::tstienban=20000.0;
void main()
```

```
{
HDBH::in();
getch();
}
```

2.7.2. Hàm thành phần tĩnh

Hàm thành phần tĩnh được viết theo một trong hai cách:

- Dùng từ khoá static đặt trước định nghĩa hàm thành phần viết bên trong định nghĩa lớp.

- Nếu hàm thành phần xây dựng bên ngoài định nghĩa lớp, thì dùng từ khoá static đặt trước khai báo hàm thành phần bên trong định nghĩa lớp. Không cho phép dùng từ khoá static đặt trước định nghĩa hàm thành phần viết bên ngoài định nghĩa lớp.

Các đặc tính của hàm thành phần tĩnh:

- Hàm thành phần tĩnh là chung cho toàn bộ lớp và không lệ thuộc vào một đối tượng cụ thể, nó tồn tại ngay khi lớp chưa có đối tượng nào.

- Lời gọi hàm thành phần tĩnh như sau:

Tên lớp :: Tên hàm thành phần tĩnh(các tham số thực sự)

- Vì hàm thành phần tĩnh là độc lập với các đối tượng, nên không thể dùng hàm thành phần tĩnh để xử lý dữ liệu của các đối tượng trong lời gọi phương thức tĩnh. Nói cách khác không cho phép truy nhập các thuộc tính (trừ thuộc tính tĩnh) trong thân hàm thành phần tĩnh. Đoạn chương trình sau minh họa điều này:

```
class HDBH
{
 private:
        int shd;
        char *tenhang;
        double tienban;
        static int tshd;
        static double tstienban;
 public:
static void in()
   {
 cout <<"\n" << tshd;
 cout << "\n" << tstienban;
 cout <<"\n"<< tenhang              //loi
 cout <<"\n" << tienban;     //loi
   }
```

```
};
```

Ví dụ 2.12

```
#include <iostream.h>
#include <conio.h>
class A
 {
  int m;
  static int n; //n la bien tinh
  public: void set_m(void) { m= ++n;}
   void show_m(void)
   {
    cout << "\n Doi tuong thu:" << m << endl;
   }
   static void show_n(void)
   {
    cout << " m = " << n << endl;
   }
 };
 int A::n=1; //khoi gan gia tri ban dau 1 cho bien tinh n
void  main()
  {
   clrscr();
   A t1, t2;
   t1.set_m();
   t2.set_m();
   A::show_n();
   A t3;
   t3.set_m();
   A::show_n();
   t1.show_m();
   t2.show_m();
   t3.show_m();
   getch();
  }
```

Kết quả chương trình trên là:

m = 3

m = 4

Doi tuong thu : 2

Doi tuong thu : 3

Doi tuong thu : 4

2.8. Hàm tạo (constructor)

Hàm tạo là một hàm thành phần đặc biệt của lớp làm nhiệm vụ tạo lập một đối tượng mới. Chương trình dịch sẽ cấp phát bộ nhớ cho đối tượng, sau đó sẽ gọi đến hàm tạo. Hàm tạo sẽ khởi gán giá trị cho các thuộc tính của đối tượng và có thể thực hiện một số công việc khác nhằm chuẩn bị cho đối tượng mới. Khi xây dựng hàm tạo cần lưu ý những đặc tính sau của hàm tạo:

- Tên hàm tạo trùng với tên của lớp.
- Hàm tạo không có kiểu trả về.
- Hàm tạo phải được khai báo trong vùng public.
- Hàm tạo có thể được xây dựng bên trong hoặc bên ngoài định nghĩa lớp.
- Hàm tạo có thể có tham số hoặc không có tham số.
- Trong một lớp có thể có nhiều hàm tạo (cùng tên nhưng khác các tham số).

Ví dụ 3.13

```
class DIEM
    {
      private:
         int x,y;
public:
         DIEM()                  //Ham tao khong tham so
         {
            x = y = 0;
          }
DIEM(int x1, int y1)  //Ham tao co tham so
          {
                x = x1;y=y1;
          }
         //Cac thanh phan khac
    };
```

Chú ý 1: Nếu lớp không có hàm tạo, chương trình dịch sẽ cung cấp một hàm tạo mặc định không đối, hàm này thực chất không làm gì cả. Như vậy một đối tượng tạo ra chỉ được cấp phát bộ nhớ, còn các thuộc tính của nó chưa được xác định.

Ví dụ 3.14

```
#include <conio.h>
#inlcule <iostream.h>
class DIEM
 {
private:
int x,y;
public:
                void in()
        {
                        cout <<"\n" << y <<" " << m;
         }
  };
void main()
  {
 DIEM d;
 d.in();
 DIEM   *p;
 p= new DIEM [10];
 clrscr();
 d.in();
 for (int i=0;1<10;++i)
   (p+i)->in();
 getch();
 }
```

Chú ý 2:

- Khi một đối tượng được khai báo thì hàm tạo của lớp tương ứng sẽ tự động thực hiện và khởi gán giá trị cho các thuộc tính của đối tượng. Dựa vào các tham số trong khai báo mà chương trình dịch sẽ biết cần gọi đến hàm tạo nào.
- Khi khai báo mảng đối tượng không cho phép dùng các tham số để khởi gán cho các thuộc tính của các đối tượng mảng.
- Câu lệnh khai báo một biến đối tượng sẽ gọi tới hàm tạo một lần.
- Câu lệnh khai báo một mảng *n* đối tượng sẽ gọi tới hàm tạo mặc định *n* lần.

• Với các hàm có các tham số kiểu lớp, thì chúng chỉ xem là các tham số hình thức, vì vậy khai báo tham số trong dòng đầu của hàm sẽ không tạo ra đối tượng mới và do đó không gọi tới các hàm tạo.

Ví dụ 3.15

```
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
class DIEM
{
private:
int   x,y;
public:
DIEM()
    {
        x = y = 0;
         }
DIEM(int x1, int y1)
         {
        x = x1; y = y1;
          }
        friend  void  in(DIEM  d)
         {
           cout <<"\n" << d.x <<" " << d.y;
         }
                void in()
              {
                   cout <<"\n" << x <<" " << y ;
               }
       };
void main()
  {
    DIEM d1;
    DIEM d2(2,3);
    DIEM  *d;
    d = new DIEM (5,6);
    clrscr();
```

```
in(d1); // Goi ham ban in()
d2.in(); // Goi ham thanh phan in()
in(*d); // Goi ham ban in()
DIEM(2,2).in();// Goi ham thanh phan in()
DIEM  t[3]; // 3 lan goi ham tao khong doi
DIEM  *q; // Goi ham tao khong doi
int n;
cout << "\n N = ";
cin >> n;
q = new DIEM [n+1]; //n+1 lan goi ham tao khong doi
for (int i=0;i<=n;++i)
q[i]=DIEM (3+i,4+i);//n+1 lan goi ham tao co doi
for (i=0;i<=n;++i)
  q[i].in();    // Goi ham thanh phan in()
for (i=0;i<=n;++i)
  DIEM(5+i,6+i).in(); //Goi ham thanh phan in()
getch();
}
```

Chú ý 3: Nếu trong lớp đã có ít nhất một hàm tạo, thì hàm tạo mặc định sẽ không được phát sinh nữa. Khi đó mọi câu lệnh xây dựng đối tượng mới đều sẽ gọi đến một hàm tạo của lớp. Nếu không tìm thấy hàm tạo cần gọi thì chương trình dịch sẽ báo lỗi. Điều này thường xãy ra khi chúng ta không xây dựng hàm tạo không đối, nhưng lại sử dụng các khai báo không tham số như ví dụ sau:

Ví dụ 3.16

```
#include <conio.h>
#include <iostream.h>
class DIEM
 {
private:
int x,y;
public:
        DIEM(int x1, int y1)
          {
            x=x1; y=y1;
          }
 void in()
```

```
        {
   cout << "\n" << x << " " << y <<" " << m;
     }
  };
void main()
{
DIEM  d1(200,200); // Goi ham tao co doi
DIEM  d2; // Loi, goi ham tao khong doi
d2= DIEM_DH (3,5); // Goi ham tao co doi
d1.in();
d2.in();
getch();
};
```

Trong ví dụ này, câu lệnh DIEM d2; trong hàm main() sẽ bị chương trình dịch báo lỗi. Bởi vì lệnh này sẽ gọi tới hàm tạo không đối, mà hàm tạo này chưa được xây dựng. Có thể khắc phục điều này bằng cách chọn một trong hai giải pháp sau:

- Xây dựng thêm hàm tạo không đối.
- Gán giá trị mặc định cho tất cả các đối x1, y1 của hàm tạo đã xây dựng ở trên.

Theo giải pháp thứ 2, chương trình trên có thể sửa lại như sau:

Ví dụ 3.17

```
#include <conio.h>
#include <iostream.h>
class DIEM
 {
private:
int x,y;
public:
     DIEM(int x1=0, int y1=0)
      {
        x = x1; y = y1;
      }
     void in()
      {
       cout << "\n" << x << " " << y <<" " << m;
       }
 };
```

```
      void main()
      {
      DIEM d1(2,3); //Goi ham tao,khong dung tham so mac dinh
      DIEM  d2; //Goi ham tao, dung tham so mac dinh
      d2= DIEM(6,7);//Goi ham tao,khong dung tham so mac dinh
      d1.in();
      d2.in();
      getch(); }
```

Ví dụ 3.18

```
      #include <iostream.h>
#include <conio.h>
      class rectangle
{
private:
    int dai;
    int rong;
    static int extra_data;
public:
    rectangle();
    void set(int new_dai, int new_rong);
   int get_area();
   int get_extra();
};
int rectangle::extra_data;
rectangle::rectangle()
{
   dai = 8;
   rong = 8;
   extra_data = 1;
}
void rectangle::set(int new_dai,int new_rong)
{
   dai = new_dai;
   rong = new_rong;
}
int rectangle::get_area()
```

```
{
   return (dai * rong);
}
int rectangle::get_extra()
{
  return extra_data++; }
void main()
{
  rectangle small, medium, large;
  small.set(5, 7);
  large.set(15, 20);
  cout<<"Dien tich la : "<<small.get_area()<<"\n";
  cout<<"Dien tich la : "<<
        medium.get_area()<<"\n";
  cout<<"Dien tich la : "<<large.get_area()<<"\n";
  cout <<"Gia tri du lieu tinh la : "<<
        small.get_extra()<<"\n";
  cout <<"Gia tri du lieu tinh la : "<<
        medium.get_extra()<<"\n";
  cout <<"Gia tri du lieu tinh la : "<<
        large.get_extra()<<"\n";
}
```

Chương trình cho kết quả như sau :

```
Dien tich la : 35
Dien tich la : 64
Dien tich la : 300
Gia tri du lieu tinh la : 1
Gia tri du lieu tinh la : 2
Gia tri du lieu tinh la : 3
```

2.9. Hàm tạo sao chép

2.9.1. Hàm tạo sao chép mặc định

Giả sử đã định nghĩa một lớp ABC nào đó. Khi đó:

- Ta có thể dùng câu lệnh khai báo hoặc cấp phát bộ nhớ để tạo các đối tượng mới, ví dụ:

```
ABC  p1, p2;
ABC *p = new ABC;
```

- Ta cũng có thể dùng lệnh khai báo để tạo một đối tượng mới từ một đối tượng đã tồn tại, ví dụ:

```
ABC  u;
ABC  v(u);  // Tạo v theo u
```

Câu lệnh này có ý nghĩa như sau:

- Nếu trong lớp ABC chưa xây dựng hàm tạo sao chép, thì câu lệnh này sẽ gọi tới một hàm tạo sao chép mặc định của C++. Hàm này sẽ sao chép nội dung từng bit của u vào các bit tương ứng của v. Như vậy các vùng nhớ của u và v sẽ có nội dung như nhau.

- Nếu trong lớp ABC đã có hàm tạo sao chép thì câu lệnh:

```
PS  v(u);
```

sẽ tạo ra đối tượng mới v, sau đó gọi tới hàm tạo sao chép để khởi gán v theo u.

Ví dụ sau minh họa cách dùng hàm tạo sao chép mặc định:

Trong chương trình đưa vào lớp PS (phân số):

+ Các thuộc tính gồm: t (tử số) và m (mẫu).

+ Trong lớp mà không có phương thức nào cả mà chỉ có hai hàm bạn là các hàm toán tử nhập (>>) và xuất (<<).

+ Nội dung chương trình là: Dùng lệnh khai báo để tạo một đối tượng u (kiểu PS) có nội dung như đối tượng đã có d.

Ví dụ 2.19

```
#include <conio.h>
#include <iostream.h>
class PS
{
 private:  int t,m;
 public:
 friend ostream& operator<< (ostream& os,const PS &p)
   {
  os<<" = "<<p.t<<"/"<<p.m;
  return os;
   }
  friend istream& operator>> (istream& is, PS &p)
   {
  cout <<" Nhap tu va mau : ";
  is>>p.t>>p.m;
  return is;
```

```
 }
};
void main()
 {
  PS d;
  cout <<"\n Nhap phan so d "; cin>>d;
  cout<<"\n PS d "<<d;
  PS u(d);
  cout<<"\n PS u "<<u;
  getch();
 }
```

2.9.2. Hàm tạo sao chép

Hàm tạo sao chép sử dụng một đối kiểu tham chiếu đối tượng để khởi gán cho đối tượng mới và được viết theo mẫu sau:

```
Tên_lớp (const Tên_lớp  &ob)
        {
            // Các câu lệnh dùng các thuộc tính của đối tượng ob để khởi gán
// cho các thuộc tính của đối tượng mới
         }
```

Ví dụ:

```
class PS
 {  private:  int  t, m;
    public:
      PS(const PS  &p)
       {
        t= p.t;
        m= p.m;
       }
      …
   };
```

Hàm tạo sao chép trong ví dụ trên không khác gì hàm tạo sao chép mặc định.

Chú ý:

- Nếu lớp không có các thuộc tính kiểu con trỏ hoặc tham chiếu thì dùng hàm tạo sao chép mặc định là đủ.

- Nếu lớp có các thuộc tính con trỏ hoặc tham chiếu, thì hàm tạo sao chép mặc định chưa đáp ứng được yêu cầu.

```
class DT
  {
    private:
      int n; // Bac da thuc
      double *a; // Tro toi vung nho chua cac he so da thuc a0, a1, …
    public:
      DT()
        {
          n = 0; a = NULL;
        }
       DT(int n1)
         {
           n = n1;
           a = new double[n1+1];
         }
       friend ostream& operator<< (ostream& os,const DT &d);
       friend istream& operator>> (istream& is,DT &d);
       …
    };
```

Bây giờ chúng ta hãy theo dõi xem việc dùng hàm tạo mặc định trong đoạn chương trình sau sẽ dẫn đến sai lầm như thế nào:

```
DT d;
cin >> d;
/* Nhập đối tượng d gồm: nhập một số nguyên dương và gán cho d.n, cấp phát vùng nhớ cho d.n, nhập các hệ số của đa thức và chứa vào vùng nhớ được cấp phát
*/
DT u(d);
/* Dùng hàm tạo mặc định để xây dựng đối tượng u theo d. Kết quả: u.n = d.n và u.a = d.a. Như vậy hai con trỏ u.a và d.a cùng trỏ đến một vùng nhớ.
*/
```

Nhận xét: Mục đích của ta là tạo ra một đối tượng u giống như d, nhưng độc lập với d. Nghĩa là khi d thay đổi thì u không bị ảnh hưởng gì. Thế nhưng mục tiêu này không đạt được, vì u và d có chung một vùng nhớ chứa hệ số của đa thức, nên khi sửa đổi các hệ số của đa thức trong d thì các hệ số của đa thức trong u cũng thay đổi theo. Còn một trường hợp nữa cũng dẫn đến lỗi là khi một trong hai đối tượng u và d bị giải

phóng (thu hồi vùng nhớ chứa đa thức) thì đối tượng còn lại cũng sẽ không còn vùng nhớ nữa.

Ví dụ sau sẽ minh họa nhận xét trên: Khi d thay đổi thì u cũng thay đổi và ngược lại khi u thay đổi thì d cũng thay đổi theo.

Ví dụ 2.20

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
class DT
 {
  private:
int n; // Bac da thuc
double *a; // Tro toi vung nho chua cac he
      //so da thuc a0,a1,...
  public:
     DT()
    {
     this->n=0; this->a=NULL;
    }
     DT(int n1)
    {
      this->n=n1;
      this->a= new double[n1+1];
    }
friend ostream& operator<< (ostream& os,const DT &d);
 friend istream& operator>> (istream& is,DT &d);
 };
 ostream& operator<< (ostream& os,const DT &d)
  {
   os <<" Cac he so ";
   for (int i=0; i<=d.n; ++i)
     os << d.a[i]<<" ";
   return os;
  }
 istream& operator>> (istream& is,DT &d)
  {
```

```
    if (d.a != NULL) delete d.a;
    cout << " \nBac da thuc:";
    cin >>d.n;
    d.a = new double[d.n+1];
    cout<<"Nhap cac he so da thuc:\n";
    for (int i=0 ; i<=d.n ; ++i)
     {
 cout<<"He so bac"<< i << "=";
        is >> d.a[i];
     }
    return is;
   }
void main()
{
  DT d;
  clrscr();
  cout<<"\nNhap da thuc d" ; cin >> d;
  DT u(d);
  cout <<"\nDa thuc d" << d ;
  cout <<"\nDa thuc u" << u ;
  cout <<"\nNhap da thuc d" << d ; cin >> d;
  cout <<"\nDa thuc d" << d ;
  cout <<"\nDa thuc u" << u ;
  cout <<"\nDa thuc u" << u ; cin >> u;
  cout <<"\nDa thuc d" << d ;
  cout <<"\nDa thuc u" << u ;
  getch();
}
```

Ví dụ 3.21 Ví dụ sau minh họa về hàm tạo sao chép:

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
class DT
{
private:
int n; // Bac da thuc
```

```
        double *a; // Tro toi vung nho chua cac da thuc
                        // a0, a1,...
        public:
                DT()
                {
                this->n=0; this->a=NULL;
                }
                DT(int n1)
                {
                this->n=n1;
                this->a= new double[n1+1];
                }
                DT(const DT &d);
friend ostream& operator<< (ostream& os,const DT &d);
  friend istream& operator>> (istream& is,DT &d);
 };
DT::DT(const DT &d)
{
 this->n = d.n;
 this->a = new double[d.n+1];
 for (int i=0;i<=d.n;++i)
 this->a[i] = d.a[i];
}
ostream& operator<< (ostream& os,const DT &d)
{
 os<<"-Cac he so (tu ao): ";
 for (int i=0 ; i<=d.n ; ++i)
 os << d.a[i] <<" ";
 return os;
}
istream& operator>> (istream& is,DT &d)
{
if (d.a != NULL) delete d.a;
cout << "\n Bac da thuc:";
cin >> d.n;
d.a = new double[d.n+1];
```

```
cout << "Nhap cac he so da thuc:\n";
for (int i=0 ; i<= d.n ; ++i)
{
  cout << "He so bac " << i << "=";
  is >> d.a[i];
}
return is;
}
void main()
{
  DT d;
  clrscr();
  cout <<"\nNhap da thuc d " ; cin >> d;
  DT u(d);
  cout << "\nDa thuc d " << d;
  cout << "\nDa thuc u " << u;
  cout << "\nNhap da thuc d "; cin >> d;
  cout << "\nDa thuc d " << d;
  cout << "\nDa thuc u " << u;
  cout << "\nNhap da thuc u " ; cin >> u;
  cout << "\nDa thuc d " << d;
  cout << "\nDa thuc u " << u;
  getch();
}
```

2.10. Hàm hủy (destructor)

Hàm huỷ là một hàm thành phần của lớp, có chức năng ngược với hàm tạo. Hàm huỷ được gọi trước khi giải phóng một đối tượng để thực hiện một số công việc có tính "dọn dẹp" trước khi đối tượng được hủy bỏ, ví dụ giải phóng một vùng nhớ mà đối tượng đang quản lý, xoá đối tượng khỏi màn hình nếu như nó đang hiển thị...Việc hủy bỏ đối tượng thường xảy ra trong 2 trường hợp sau:

- Trong các toán tử và hàm giải phóng bộ nhớ như delete, free...
- Giải phóng các biến, mảng cục bộ khi thoát khỏi hàm, phương thức.

Nếu trong lớp không định nghĩa hàm huỷ thì một hàm huỷ mặc định không làm gì cả được phát sinh. Đối với nhiều lớp thì hàm huỷ mặc định là đủ, không cần đưa vào một hàm huỷ mới. Trường hợp cần xây dựng hàm hủy thì tuân theo quy tắc sau:

- Mỗi lớp chỉ có một hàm hủy.

- Hàm hủy không có kiểu, không có giá trị trả về và không có tham số.
- Tên hàm hàm huỷ có một dấu ngã ngay trước tên lớp.

Ví dụ:

```
class A
{
private:
int n;
double *a;
public:
~A()
{
n = 0;
delete a;
}
      };
```

Ví dụ 3.22

```
#include <iostream.h>
class Count{
private:
   static int counter;
   int obj_id;
public:
  Count();
  static void display_total();
  void display();
  ~Count();
};
int Count::counter;
Count::Count()
{
   counter++;
   obj_id = counter;
}
Count::~Count()
{
  counter--;
```

```
  cout<<"Doi tuong "<<obj_id<<" duoc huy bo\n";
}
void Count::display_total()
{
cout <<"So cac doi tuong duoc tao ra la  = "<< counter << endl;
}
void Count::display()
{
  cout << "Object ID la  "<<obj_id<<endl;
}
void main()
{
  Count a1;
  Count::display_total();
  Count a2, a3;
  Count::display_total();
  a1.display();
  a2.display();
  a3.display();
}
```

Kết quả chương trình như sau:

```
    So cac doi tuong duoc tao ra la  = 1
So cac doi tuong duoc tao ra la  = 3
Object ID la  1
Object ID la  2
Object ID la  3
Doi tuong 3 duoc huy bo
Doi tuong 2 duoc huy bo
Doi tuong 1 duoc huy bo
```

BÀI TẬP

1. Xây dựng lớp thời gian **Time**. Dữ liệu thành phần bao gồm giờ, phút giây. Các hàm thành phần bao gồm: hàm tạo, hàm truy cập dữ liệu, hàm normalize() để chuẩn hóa dữ liệu nằm trong khoảng quy định của giờ ($0 \le$ giờ < 24) , phút ($0 \le$ phút <60), giây ($0 \le$ giây <60), hàm advance(int h, int m, int s) để tăng thời gian hiện hành của đối tượng đang tồn tại, hàm reset(int h, int m, int s) để chỉnh lại thời gian hiện hành của một đối tượng đang tồn tại và một hàm print() để hiển thị dữ liệu.

2. Xây dựng lớp **Date**. Dữ liệu thành phần bao gồm ngày, tháng, năm. Các hàm thành phần bao gồm: hàm tạo, hàm truy cập dữ liệu, hàm normalize() để chuẩn hóa dữ liệu nằm trong khoảng quy định của ngày ($1 \le$ ngày $<$daysIn(tháng)), tháng ($1 \le$ tháng < 12), năm (năm ≥ 1), hàm daysIn(int) trả về số ngày trong tháng, hàm advance(int y, int m, int d) để tăng ngày hiện lên các năm y, tháng m, ngày d của đối tượng đang tồn tại, hàm reset(int y, int m, int d) để đặt lại ngày cho một đối tượng đang tồn tại và một hàm print() để hiển thị dữ liệu.

3. Thực hiện một lớp **String**. Mỗi đối tượng của lớp sẽ đại diện một chuỗi ký tự. Những thành phần dữ liệu là chiều dài chuỗi, và chuỗi ký tự. Các hàm thành phần bao gồm: hàm tạo, hàm truy cập, hàm hiển thị, hàm character(int i)trả về một ký tự trong chuỗi được chỉ định bằng tham số i.

4. Xây dựng lớp ma trận có tên là **Matrix** cho các ma trận, các hàm thành phần bao gồm: hàm tạo mặc định, hàm nhập xuất ma trận, cộng, trừ, nhân hai ma trận.

5. Xây dựng lớp ma trận có tên là **Matrix** cho các ma trận vuông, các hàm thành phần bao gồm: hàm tạo mặc định, hàm nhập xuất ma trận, tính định thức và tính ma trận nghịch đảo.

6. Xây dựng lớp **Stack** cho ngăn xếp kiểu int. Các hàm thành phần bao gồm: Hàm tạo mặc định, hàm hủy, hàm isEmpty() kiểm tra stack có rỗng không, hàm isFull() kiểm tra stack có đầy không, hàm push() , pop(), hàm in nội dung ngăn xếp. Sử dụng một mảng để thực hiện.

7. Xây dựng lớp hàng đợi **Queue** chứa phần tử kiểu int. Các hàm thành phần bao gồm: hàm tạo, hàm hủy và những toán tử hàng đợi thông thường: hàm insert() để thêm phần tử vào hàng đợi, hàm remove() để loại bỏ phần tử, hàm isEmpty() kiểm tra hàng đợi có rỗng không, hàm isFull() kiểm tra hàng đợi có đầy không. Sử dụng một mảng để thực hiện.

4. Xây dựng lớp đa thức và các phương thức cộng, trừ hai đa thức.

8. Xây dựng lớp **Sinhvien** để quản lý hộ tên sinh viên, năm sinh, điểm thi 9 môn học của các sinh viên. Cho biết sinh viên nào được làm khóa luận tốt nghiệp, bao

nhiêu sinh viên thi tốt nghiệp, bao nhiêu sinh viên thi lại, tên môn thi lại> Tiêu chuẩn để xét như sau:

- Sinh viên làm khóa luận phải có điểm trung bình từ 7 trở lên, trong đó không có môn nào dưới 5.
- Sinh viên thi tố nghiệp khi điểm trung bình nhỏ hơn 7 và điểm các môn không dưới 5.
- Sinh viên thi lại môn dưới 5.

9. Xây dựng lớp **vector** để lưu trữ các vectơ gồm các số thực. Các thành phần dữ liệu bao gồm:

- Kích thước vectơ.
- Một mảng động chứa các thành phần của vectơ.

Các hàm thành phần bao gồm hàm tạo, hàm hủy, hàm tính tích vô hướng hai vectơ, tính chuẩn của vectơ (theo chuẩn bất kỳ nào đó).

10. Xây dựng lớp **Phanso** với dữ liệu thành phần là tử và mẫu số. Các hàm thành phần bao gồm:

- Cộng hai phân số, kết quả phải dược tối giản
- Trừ hai phân số, kết quả phải dược tối giản
- Nhân hai phân số, kết quả phải dược tối giản
- Chia hai phân số, kết quả phải dược tối giản

CHƯƠNG 3

TOÁN TỬ TẢI BỘI

3.1. Định nghĩa toán tử tải bội

Các toán tử cùng tên thực hiện nhiều chức năng khác nhau được gọi là toán tử tải bội. Dạng định nghĩa tổng quát của toán tử tải bội như sau:

Kiểu_trả_về operator op(danh sách tham số)

{//thân toán tử}

Trong đó: Kiểu_trả_về là kiểu kết quả thực hiện của toán tử.

op là tên toán tử tải bội

operator op(danh sách tham số) gọi là hàm toán tử tải bội, nó có thể là hàm thành phần hoặc là hàm bạn, nhưng không thể là hàm tĩnh. Danh sách tham số được khai báo tương tự khai báo biến nhưng phải tuân theo những quy định sau:

- Nếu toán tử tải bội là hàm thành phần thì: không có tham số cho toán tử một ngôi và một tham số cho toán tử hai ngôi. Cũng giống như hàm thành phần thông thường, hàm thành phần toán tử có đối đầu tiên (không tường minh) là con trỏ this .

- Nếu toán tử tải bội là hàm bạn thì: có một tham số cho toán tử một ngôi và hai tham số cho toán tử hai ngôi.

Quá trình xây dựng toán tử tải bội được thực hiện như sau:

- Định nghĩa lớp để xác định kiểu dữ liệu sẽ được sử dụng trong các toán tử tải bội

- Khai báo hàm toán tử tải bội trong vùng public của lớp

- Định nghĩa nội dung cần thực hiện

3.2. Một số lưu ý khi xây dựng toán tử tải bội

1. Trong C++ ta có thể đưa ra nhiều định nghĩa mới cho hầu hết các toán tử trong C++, ngoại trừ những toán tử sau đây:

- Toán tử xác định thành phần của lớp (‘.’)

- Toán tử phân giải miền xác định (‘::’)

- Toán tử xác định kích thước (‘sizeof’)

- Toán tử điều kiện 3 ngôi (‘?:’)

2. Mặc dầu ngữ nghĩa của toán tử được mở rộng nhưng cú pháp, các quy tắc văn phạm như số toán hạng, quyền ưu tiên và thứ tự kết hợp thực hiện của các toán tử vẫn không có gì thay đổi.

3. Không thể thay đổi ý nghĩa cơ bản của các toán tử đã định nghĩa trước, ví dụ không thể định nghĩa lại các phép toán +, - đối với các số kiểu int, float.

4. Các toán tử = , ( ) , [ ] , -> yêu cầu hàm toán tử phải là hàm thành phần của lớp, không thể dùng hàm bạn để định nghĩa toán tử tải bội.

3.3. Một số ví dụ

Ví dụ Toán tử tải bội một ngôi, dùng hàm bạn

```
#include <iostream.h>
#include <conio.h>
class Diem
  {
   private:
    float x,y,z;
   public:
    Diem() {}
    Diem(float x1,float y1,float z1)
      { x = x1; y = y1; z=z1;}
    friend Diem  operator -(Diem d)
            {
              Diem d1;
              d1.x = -d.x; d1.y = -d.y;d1.z=-d.z;
              return d1;
            }
    void hienthi()
            { cout<<"Toa do: "<<x<<" "<<y <<" "
        <<z<<endl;}
  };

void main()
  {
   clrscr();
   Diem p(2,3,-4),q;
   q = -p;
   p.hienthi();
   q.hienthi();
   getch();
  }
```

Ví dụ 4.2 Toán tử tải bội hai ngôi, dùng hàm bạn

```
#include <iostream.h>
#include <conio.h>
class Diem
```

```
  {
   private:
     float x,y,z;
   public:
     Diem() {}
     Diem(float x1,float y1,float z1)
       { x = x1; y = y1; z=z1;}
     friend Diem  operator +(Diem d1, Diem d2)
              {
                Diem tam;
                tam.x = d1.x + d2.x;
                tam.y = d1.y + d2.y;
                tam.z = d1.z + d2.z;
                return tam;
              }
     void hienthi()
              { cout<<x<<" "<<y <<" " <<z<<endl;}
  };

void main()
  {
   clrscr();
   Diem d1(3,-6,8),d2(4,3,7),d3;
   d3=d1+d2;
   d1.hienthi();
   d2.hienthi();
   cout<<"\n Tong hai diem co toa do la :";
   d3.hienthi();
   getch();
  }
```

Ví dụ Toán tử tải bội hai ngôi, dùng hàm thành phần

```
#include <iostream.h>
#include <conio.h>
class Diem
  {
   private:
```

```
    float x,y;
  public:
    Diem() {}
    Diem(float x1,float y1)
      { x = x1; y = y1;}
    Diem operator -()
          { x = -x; y = -y; z = -z;
            return (*this); }
    void hienthi()
   { cout<<"Toa do: "<<x<<" "<<y <<" z = "<<z
      <<" \n";}
 };

void main()
 {
  clrscr();
  Diem p(2,3,-4),q;
  p.hienthi();
  q = -p;
  q.hienthi();
  getch();
 }
```

Ví dụ Toán tử tải bội hai ngôi, dùng hàm thành phần

```
#include <iostream.h>
#include <conio.h>
class Diem
 {
  private:
    float x,y,z;
  public:
    Diem() {}
    Diem(float x1,float y1,float z1)
      { x = x1; y = y1; z=z1;}
    Diem  operator +(Diem d2)
          {
            x = x + d2.x;
```

```
                y = y + d2.y;
                z = z + d2.z;
                return (*this);
              }
    void hienthi()
  { cout<<"\n x="<<x<<" y= "<<y<<" z =  " << z
      <<endl;}
  };

void main()
  {
   clrscr();
   Diem d1(3,-6,8),d2(4,3,7),d3;
   d1.hienthi();
   d2.hienthi();
   d3=d1+d2;
   cout<<"\n Tong hai diem co toa do la :";
   d3.hienthi();
   getch();
  }
```

Ví dụ

```
#include <iostream.h>
#include <conio.h>
class Diem
  {
   private:
    int x,y;
   public:
    Diem() {}
    Diem(int x1,int y1)
      { x = x1; y = y1;}
    void operator -()
              {
                    x = -x; y = -y;
              }
    void hienthi()
```

```
            { cout<<"Toa do: "<<x<<" "<<y <<" \n";}
  };
void main()
  {
    clrscr();
    Diem p(2,3),q;
    p.hienthi();
    -p;
    p.hienthi();
    getch();
  }
```

Ví dụ Toán tử tải bội hai ngôi

```
#include <iostream.h>
#include <conio.h>
class sophuc
{float a,b;
 public : sophuc() {}
  sophuc(float x, float y)
    {a=x; b=y;}
  sophuc operator +(sophuc c2)
      {       sophuc c3;
              c3.a= a + c2.a ;
                    c3.b= b + c2.b ;
                    return (c3);
       }
  void  hienthi(sophuc c)
  { cout<<c.a<<" + "<<c.b<<"i"<<endl; }
};
void main()
{ clrscr();
  sophuc d1 (2.1,3.4);
  sophuc d2 (1.2,2.3) ;
  sophuc d3 ;
  d3 = d1+d2;   //d3=d1.operator +(d2);
  cout<<"d1= ";d1.hienthi(d1);
  cout<<"d2= ";d2.hienthi(d2);
```

```
 cout<<"d3= ";d3.hienthi(d3);
 getch();
 }
```

Chú ý: Trong các hàm toán tử thành phần hai ngôi (có hai toán hạng) thì con trỏ this ứng với toán hạng thứ nhất, vì vậy trong tham số của toán tử chỉ cần dùng một tham số tường minh để biểu thị toán hạng thứ hai .

Ví dụ Phiên bản 2 của ví dụ 4.6

```
#include <iostream.h>
#include <conio.h>
class sophuc
{float a,b;
 public : sophuc() {}
  sophuc(float x, float y)
    {a=x; b=y;}
  sophuc operator +(sophuc c2)
      {
        a= a + c2.a ;
              b= b + c2.b ;
              return (*this);
      }
  void  hienthi(sophuc c)
  { cout<<c.a<<" + "<<c.b<<"i"<<endl; }
};
void main()
{ clrscr();
  sophuc d1 (2.1,3.4);
  sophuc d2 (1.2,2.3) ;
  sophuc d3 ;
  cout<<"d1= ";d1.hienthi(d1);
  cout<<"d2= ";d2.hienthi(d2);
  d3 = d1+d2;   //d3=d1.operator +(d2);
  cout<<"d3= ";d3.hienthi(d3);
  getch();
  }
```

Ví dụ Phiên bản 3 của ví dụ 4.6

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

```
#include <conio.h>
class sophuc
{float a,b;
 public : sophuc() {}
  sophuc(float x, float y)
    {a=x; b=y;}
  friend sophuc operator +(sophuc c1,sophuc c2)
    { sophuc c;
       c.a= c1.a + c2.a ;
       c.b= c1.b + c2.b ;
       return (c);
     }
  void  hienthi(sophuc c)
  { cout<<c.a<<" + "<<c.b<<"i"<<endl; }
};
void main()
{ clrscr();
  sophuc d1 (2.1,3.4);
  sophuc d2 (1.2,2.3) ;
  sophuc d3 ;
  cout<<"d1= ";d1.hienthi(d1);
  cout<<"d2= ";d2.hienthi(d2);
  d3 = d1+d2;   //d3=operator +(d1,d2);
  cout<<"d3= ";d3.hienthi(d3);
  getch();
  }
```

Ví dụ Toán tử tải bội trên lớp chuỗi ký tự

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class string
{ char s[80];
  public:
    string() { *s='\0'; }
    string(char *p) { strcpy(s,p); }
    char *get() { return s;}
```

```
    string operator + (string s2);
    string operator = (string s2);
    int operator < (string s2);
    int operator > (string s2);
    int operator == (string s2);
};
string string::operator +(string s2)
{
 strcat(s,s2.s);
 return *this ;
}

string string::operator =(string s2)
{
 strcpy(s,s2.s) ;
 return *this;
}
int string::operator <(string s2)
{
  return strcmp(s,s2.s)<0 ;
}
int string::operator >(string s2)
{
  return strcmp(s,s2.s)>0 ;
}
int string::operator ==(string s2)
{
  return strcmp(s,s2.s)==0 ;
}
void main()
{ clrscr();
  string o1 ("Trung Tam "), o2 (" Tin hoc"), o3;
  cout<<"o1 = "<<o1.get()<<'\n';
  cout<<"o2 = "<<o2.get()<<'\n';
  if (o1 > o2)
    cout << "o1 > o2 \n";
```

```
 if (o1 < o2)
   cout << "o1 < o2 \n";
 if (o1 == o2)
   cout << "o1 bang o3 \n";
 o3=o1+o2;
 cout<<"o3 ="<<o3.get()<<'\n';  //Trung tam tin hoc
 o3=o2;
 cout<<"o2 = "<<o2.get()<<'\n'; //Tin hoc
 cout<<"o3 = "<<o3.get()<<'\n'; //Tin hoc
 if (o2 == o3)
   cout << "o2 bang o3 \n";
 getch();
}
```

3.4. Định nghĩa chồng các toán tử ++ , --

Ta có thể định nghĩa chồng cho các toán tử ++/-- theo quy định sau:

- Toán tử ++/-- dạng tiền tố trả về một tham chiếu đến đối tượng thuộc lớp.
- Toán tử ++/-- dạng tiền tố trả về một đối tượng thuộc lớp.

Ví dụ

```
#include <iostream.h>
#include <conio.h>
class Diem
  {
    private:
      int x,y;
    public:
      Diem() {x = y = 0;}
      Diem(int x1, int y1)
   {x = x1;
    y = y1;}
 Diem & operator ++(); //qua tai toan tu ++ tien to
 Diem operator ++(int); //qua tai toan tu ++ hau to
 Diem & operator --(); //qua tai toan tu -- tien to
 Diem operator --(int); //qua tai toan tu -- hau to
 void hienthi()
 {
  cout<<" x = "<<x<<" y = "<<y;
```

```
   }
   };
Diem & Diem::operator ++()
 {
   x++;
   y++;
   return (*this);
  }
Diem Diem::operator ++(int)
 {
   Diem temp = *this;
   ++*this;
   return temp;
  }
Diem & Diem::operator --()
 {
   x--;
   y--;
   return (*this);
  }
Diem Diem::operator --(int)
 {
   Diem temp = *this;
   --*this;
   return temp;
   }
void main()
   {
     clrscr();
     Diem d1(5,10),d2(20,25),d3(30,40),d4(50,60);
     cout<<"\nd1 : ";d1.hienthi();
     ++d1;
     cout<<"\n Sau khi tac dong cac toan tu tang
     truoc :";
     cout<<"\nd1 : ";d1.hienthi();
     cout<<"\nd2 : ";d2.hienthi();
```

```
    d2++;
    cout<<" \n Sau khi tac dong cac toan tu tang
    sau";
    cout<<"\nd2 : ";d2.hienthi();
    cout<<"\nd3 : ";d3.hienthi();
    --d3;
    cout<<"\n Sau khi tac dong cac toan tu giam
    truoc :";
    cout<<"\nd3 : ";d3.hienthi();
    cout<<"\nd4 : ";d4.hienthi();
    d4--;
    cout<<"\n Sau khi tac dong cac toan tu giam
    sau : ";
    cout<<"\nd4 : ";d4.hienthi();
    getch();
  }
```

Chương trình cho kết quả như sau:

```
d1 :  x = 5 y = 10
 Sau khi tac dong cac toan tu tang truoc :
d1 :  x = 6 y = 11
d2 :  x = 20 y = 25
 Sau khi tac dong cac toan tu tang sau
d2 :  x = 21 y = 26
d3 :  x = 30 y = 40
 Sau khi tac dong cac toan tu giam truoc :
d3 :  x = 29 y = 39
d4 :  x = 50 y = 60
 Sau khi tac dong cac toan tu giam sau :
d4 :  x = 49 y = 59
```

Chú ý: Đối số int trong dạng hậu tố là bắt buộc, dùng để phân biệt với dạng tiền tố, thường nó mang trị mặc định là 0.

3.5. Định nghĩa chồng toán tử << và >>

Ta có thể định nghĩa chồng cho hai toán tử vào/ra << và >> kết hợp với **cout** và **cin** (cout<< và cin>>), cho phép các đối tượng đứng bên phải chúng. Lúc đó ta có thể thực hiện các thao tác vào ra như nhập dữ liệu từ bàn phím cho các đối tượng, hiển thị

giá trị thành phần dữ liệu của các đối tượng ra màn hình. Hai hàm toán tử << và >> phải là hàm tự do và khai báo là hàm bạn của lớp.

Ví dụ

```
#include <iostream.h>
#include <conio.h>
class SO
 {
  private:
     int giatri;
  public:
     SO(int x=0)
     {
       giatri = x;
      }
     SO (SO &tso)
     {
giatri = tso.giatri;
      }
friend istream& operator>>(istream&,SO&);
friend ostream& operator<<(ostream&,SO&);
};
void main(){
   clrscr();
   SO so1,so2;
   cout<<"Nhap du lieu cho so1 va so2 " << endl;
   cin>>so1;
   cin>>so2;
   cout<<"Gia tri so1 la : " <<so1
       <<" so 2 la : "<<so2<<endl;
    getch();
}
istream& operator>>(istream& nhap,SO& so)
{
  cout << "Nhap gia tri so :";
  nhap>> so.giatri;
  return nhap; }
```

```
ostream& operator<<(ostream& xuat,SO& so)
{
  xuat<< so.giatri;
  return xuat;
}
```

BÀI TẬP

1. Định nghĩa các phép toán tải bội =, ==, ++, --, +=, <<, >> trên lớp Time (bài tập 1 chương 3).

2. Định nghĩa các phép toán tải bội =, ==, ++, --, +=, <<, >> trên lớp Date (bài tập 2 chương 3).

1. Định nghĩa các phép toán tải bội +, -, *, =, ==, != trên lớp các ma trận vuông.

2. Định nghĩa các phép toán tải bội +, -, * trên lớp đa thức.

3. Định nghĩa các phép toán tải bội +, -, *, /, =, ==, +=, -=, *=, /= , <, >, <=, >=, != , ++, -- trên lớp Phanso (bài tập 10 chương 3).

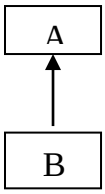
4. Ma trận được xem là một vectơ mà mỗi thành phần của nó là một vectơ. Theo nghĩa đó, hãy định nghĩa lớp **Matran** dựa trên vectơ. Tìm cách để chương trình dịch hiểu được phép truy nhập m[i][j], trong đó m là một đối tượng thuộc lớp **Matran**.

CHƯƠNG 4

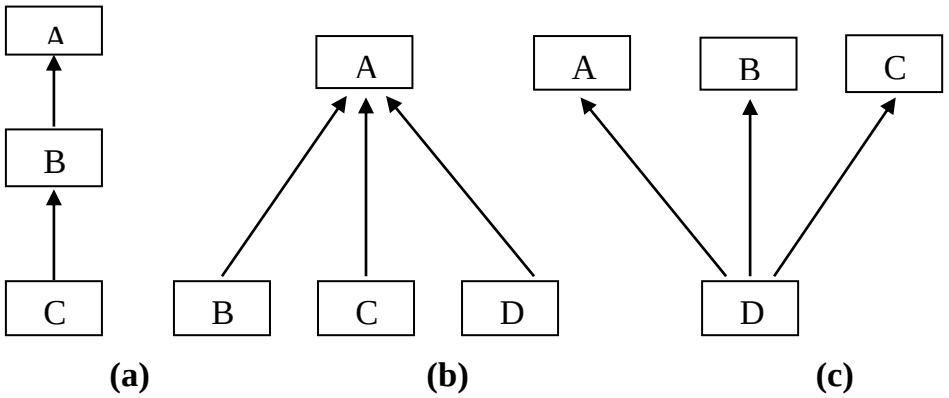
KẾ THỪA

4.1. Giới thiệu

Kế thừa là một trong các khái niệm cơ sở của phương pháp lập trình hướng đối tượng. Tính kế thừa cho phép định nghĩa các lớp mới từ các lớp đã có. Một lớp có thể là lớp cơ sở cho nhiều lớp dẫn xuất khác nhau. Lớp dẫn xuất sẽ kế thừa một số thành phần (dữ liệu và hàm) của lớp cơ sở, đồng thời có thêm những thành phần mới. Có hai loại kế thừa là: đơn kế thừa và đa kế thừa, có thể minh họa qua các hình vẽ sau đây:



Hình 4.1. Đơn kế thừa, lớp A là lớp cơ sở của lớp B



Hình 4.2. Đa kế thừa

Hình (a): Lớp A là lớp cơ sở của lớp B, lớp B là lớp cơ sở của lớp C

Hình (b): Lớp A là lớp cơ sở của các lớp B, C, D

Hình (c): Lớp A, B, C là lớp cơ sở của lớp D

4.2. Đơn kế thừa

4.2.1. Định nghĩa lớp dẫn xuất từ một lớp cơ sở

Giả sử đã định nghĩa lớp A. Cú pháp để xây dựng lớp B dẫn xuất từ lớp A như sau:

```
class  B: mode A
{
        private:
            // Khai báo các thuộc tính của lớp B
        public:
```

```
        // Định nghĩa các hàm thành phần của lớp B
};
```

Trong đó mode có thể là **private** hoặc **public** với ý nghĩa như sau:

- Kế thừa theo kiểu public thì tất cả các thành phần public của lớp cơ sở cũng là thành phần public của lớp dẫn xuất.

- Kế thừa theo kiểu private thì tất cả các thành phần public của lớp cơ sở sẽ trở thành các thành phần private của lớp dẫn xuất.

Chú ý: Trong cả hai trường hợp ở trên thì thành phần private của lớp cơ sở là không được kế thừa. Như vậy trong lớp dẫn xuất không cho phép truy nhập đến các thành phần private của lớp cơ sở.

4.2.2. Truy nhập các thành phần trong lớp dẫn xuất

Thành phần của lớp dẫn xuất bao gồm: các thành phần khai báo trong lớp dẫn xuất và các thành phần mà lớp dẫn xuất thừa kế từ các lớp cơ sở. Quy tắc sử dụng các thành phần trong lớp dẫn xuất được thực hiện theo theo mẫu như sau:

Tên đối tượng.Tên_lớp::Tên_thành_phần

Khi đó chương trình dịch C++ dễ dàng phân biệt thành phần thuộc lớp nào.

Ví dụ Giả sử có các lớp A và B như sau:

```
class A
 {
      public: int n;
 void nhap()
                  {
                        cout<<"\n Nhap n = ";
                        cin>>n;
                  }
};
class B: public A
 {
      public: int m;
void nhap()
                  {
                        cout<<"\n Nhap m = ";
                        cin>>m;
                  }
};
```

Xét khai báo: B ob;

Lúc đó: ob.B::m là thuộc tính m khai báo trong B

ob.A::n là thuộc tính n thừa kế từ lớp A

ob.B::nhap() là hàm nhap() định nghĩa trong lớp B

ob.A::nhap() là hàm nhap() định nghĩa trong lớp A

Chú ý: Để sử dụng các thành phần của lớp dẫn xuất, có thể không dùng tên lớp, chỉ dùng tên thành phần. Khi đó chương trình dịch phải tự phán đoán để biết thành phần đó thuộc lớp nào: trước tiên xem thành phần đang xét có trùng tên với các thành phần nào của lớp dẫn xuất không? Nếu trùng thì đó thành phần của lớp dẫn xuất. Nếu không trùng thì tiếp tục xét các lớp cơ sở theo thứ tự: các lớp có quan hệ gần với lớp dẫn xuất sẽ được xét trước, các lớp quan hệ xa hơn xét sau. Chú ý trường hợp thành phần đang xét có mặt đồng thời trong 2 lớp cơ sở có cùng một đẳng cấp quan hệ với lớp dẫn xuất. Trường hợp này chương trình dịch không thể quyết định được thành phần này thừa kế từ lớp nào và sẽ đưa ra một thông báo lỗi.

4.2.3. Định nghĩa lại các hàm thành phần của lớp cơ sở trong lớp dẫn xuất

Trong lớp dẫn xuất có thể định nghĩa lại hàm thành phần của lớp cơ sở. Như vậy có hai phiên bản khác nhau của hàm thành phần trong lớp dẫn xuất. Trong phạm vi lớp dẫn xuất, hàm định nghĩa lại "che khuất" hàm được định nghĩa. Việc sử dụng hàm nào cần tuân theo quy định ở trên.

Chú ý: Việc định nghĩa lại hàm thành phần khác với định nghĩa hàm quá tải. Hàm định nghĩa lại và hàm bị định nghĩa lại giống nhau về tên, tham số và giá trị trả về. Chúng chỉ khác nhau về vị trí: một hàm đặt trong lớp dẫn xuất và hàm kia thì ở trong lớp cơ sở. Trong khi đó, các hàm quá tải chỉ có cùng tên, nhưng khác nhau về danh sách tham số và tất cả chúng thuộc cùng một lớp. Định nghĩa lại hàm thành phần chính là cơ sở cho việc xây dựng tính đa hình của các hàm.

C++ cho phép đặt trùng tên thuộc tính trong các lớp cơ sở và lớp dẫn xuất. Các thành phần cùng tên này có thể cùng kiểu hay khác kiểu. Lúc này bên trong đối tượng của lớp dẫn xuất có tới hai thành phần khác nhau có cùng tên, nhưng trong phạm vi lớp dẫn xuất, tên chung đó nhằm chỉ định thành phần được khai báo lại trong lớp dẫn xuất. Khi muốn chỉ định thành phần trùng tên trong lớp cơ sở phải dùng tên lớp toán tử '::' đặt trước tên hàm thành phần.

Ví dụ Xét các lớp A và B được xây dựng như sau:

```
class A
  {
 private:
int a, b, c;
 public:
```

```
    void nhap()
     { cout<<"\na = "; cin>>a;
       cout<<"\nb = "; cin>>b;
       cout<<"\nc = "; cin>>c;
        }
void hienthi()
    { cout<<"\na = "<<a<<" b = "<<b<<" c = "<<c; }
};
class B: private A
{
 private:
double d;
public:
    void nhap_d()
     { cout<<"\nd = "; cin>>d;
        }
  };
```

Lớp B kế thừa theo kiểu private từ lớp A, các thành phần public của lớp A là các hàm nhap() và hienthi() trở thành thành phần private của lớp B.

Xét khai báo : B ob; Lúc đó các câu lệnh sau đây là lỗi :

```
ob.nhap();
ob.d=10;
ob.a = ob.b = ob.c = 5;
ob.hienthi();
```

bởi vì đối tượng ob không thể truy nhập vào các thành phần private của lớp A và B.

Ví dụ Chương trình minh họa đơn kế thừa theo kiểu public:

```
#include <iostream.h>
#include <conio.h>
class A
 {
   int a; //Bien private, khong duoc ke thua
   public:
     int b; //Bien public, se duoc ke thua
     void get_ab();
     int get_a(void);
```

```
      void show_a(void);
  };
class B: public A
  {
    int c;
    public:
      void mul(void);
      void display(void);
  };
void A::get_ab(void)
   { a = 5; b = 10;}
int A::get_a()
   { return a;}
void A::show_a()
   { cout << "a = " << a << " \n";}
void B::mul()
   { c = b*get_a();}
void B::display()
   {
     cout << "a = " << get_a() << "\n";
     cout << "b = " << b << "\n";
     cout << "c = " << c << "\n";
   }
void main()
   { clrscr();
     B d;
     d.get_ab(); //Ke thua tu A
     d.mul();
     d.show_a(); //Ke thua tu A
     d.display();
     d.b = 20;
     d.mul();
     d.display();
     getch();
   }
```

Chương trình cho kết quả:

```
a = 5
a = 5
b = 10
c = 50
a = 5
b = 20
c = 100
```

Ví dụ Chương trình sau là minh họa khác cho trường hợp kế thừa đơn:

```
#include <conio.h>
#include <iostream.h>
class Diem
 {
 private:
   double x, y;
 public:
    void nhap()
      {
       cout<<"\n x = "; cin>>x;
       cout<<"\n y = "; cin>>y;
      }
    void hienthi()
     {
       cout<<"\n x = "<<x<<" y = "<<y;
      }
 };
class Hinhtron: public Diem
 {
private:
        double r;
public:
        void nhap_r()
     {
       cout<<"\n r = "; cin>>r;
     }
double get_r()
        {
```

```
        return r;
    }
};
void main()
{
Hinhtron  h;
clrscr();
cout<<"\n Nhap toa do tam va ban kinh hinh tron";
h.nhap();
h.nhap_r();
cout<<"\n Hinh tron co tam:";h.hienthi();
cout<<"\n Co ban kinh = " << h.get_r();
getch();
  }
```

Chương trình cho kết quả:

```
Nhap toa do tam va ban kinh hinh tron
x = 2
y = 3
r = 10
Hinh tron co tam:
x = 2 y = 3
Co ban kinh = 10
```

4.2.4. Hàm tạo đối với tính kế thừa

Các hàm tạo của lớp cơ sở là không được kế thừa. Một đối tượng của lớp dẫn xuất về thực chất có thể xem là một đối tượng của lớp cơ sở, vì vậy việc gọi hàm tạo lớp dẫn xuất để tạo đối tượng của lớp dẫn xuất sẽ kéo theo việc gọi đến một hàm tạo của lớp cơ sở. Thứ tự thực hiện của các hàm tạo sẽ là: hàm tạo cho lớp cơ sở, rồi đến hàm tạo cho lớp dẫn xuất.

C++ thực hiện điều này bằng cách: trong định nghĩa của hàm tạo lớp dẫn xuất, ta mô tả một lời gọi tới hàm tạo trong lớp cơ sở. Cú pháp để truyền tham số từ lớp dẫn xuất đến lớp cơ sở như sau:

```
Tên_ lớp_dẫn_xuất(danh sách đối):Tên_ lớp_cơ_sở (danh sách đối)
        {
//thân hàm tạo của lớp dẫn xuất
        } ;
```

Trong phần lớn các trường hợp, hàm tạo của lớp dẫn xuất và hàm tạo của lớp cơ sở sẽ không dùng tham số giống nhau. Trong trường hợp cần truyền một hay nhiều tham số cho mỗi lớp, ta phải truyền cho hàm tạo của lớp dẫn xuất **tất cả** các tham số mà cả hai lớp dẫn xuất và cơ sở cần đến. Sau đó, lớp dẫn xuất chỉ truyền cho lớp cơ sở những tham số nào mà lớp cơ sở cần.

Ví dụ Chương trình sau minh họa cách truyền tham số cho hàm tạo của lớp cơ sở:

```
#include <iostream.h>
#include <conio.h>
class Diem
 {
        private:
              double x, y;
        public:
        Diem()
                 {
               x=y=0.0;
            }
    Diem(double x1, double y1)
       {
        x=x1; y=y1;
              }
           void in()
                       {
                  cout<<"\nx="<<x<<"y="<<y;
             }
};
class Hinhtron: public Diem
{
   private:
              double r;
         public:
              Hinhtron()
               {
                 r = 0.0;
          }
```

```
Hinhtron(double x1,double y1,double r1): Diem (x1, y1)
{
 r=r1;
      }
 double get_r()
       {
        return r;
       }
  };
void main()
 {
 Hinhtron  h(2.5, 3.5, 8);
 clrscr();
 cout<<"\n Hinh tron co tam:";
 h.in();
 cout<<"\n Co ban kinh =" << h.get_r();
 getch();
 }
```

Chú ý: Các tham số mà hàm tạo của lớp dẫn xuất truyền cho hàm tạo của lớp cơ sở không nhất thiết phải lấy hoàn toàn y như từ các tham số nó nhận được. Ví dụ:

```
Hinhtron(double x1,double y1,double r1):Diem (x1/2, y1/2)
```

4.2.5. Hàm hủy đối với tính kế thừa

Hàm hủy của lớp cơ sở cũng không được kế thừa. Khi cả lớp cơ sở và lớp dẫn xuất có các hàm hủy và hàm tạo, các hàm tạo thi hành theo thứ tự dẫn xuất. Các hàm hủy được thi hành theo thứ tự ngược lại. Nghĩa là, hàm tạo của lớp cơ sở thi hành trước hàm tạo của lớp dẫn xuất, hàm hủy của lớp dẫn xuất thi hành trước hàm hủy của lớp cơ sở.

Ví dụ

```
#include <iostream.h>
#include <conio.h>
class CS
{ public:
CS()
  {cout<<"\nHam tao lop co so";}
   ~CS()
  {cout<<"\nHam huy lop co so";}
```

```
};
class DX:public CS
{ public:
DX()
  {cout<<"\nHam tao lop dan xuat";}
   ~DX()
  {cout<<"\nHam huy lop dan xuat";}
};
void main()
 { clrscr();
   DX ob;
   getch();
 }
```

Chương trình này cho kết quả như sau :

```
Ham tao lop co so
Ham tao dan xuat
Ham huy lop dan xuat
Ham huy lop co so
```

4.2.6. Khai báo protected

Ta đã biết các thành phần khai báo private không được kế thừa trong lớp dẫn xuất. Có thể giải quyết vấn đề này bằng cách chuyển chúng sang vùng public. Tuy nhiên cách làm này lại phá vỡ nguyên lý che dấu thông tin của LTHĐT. C++ đưa ra cách giải quyết khác là sử dụng khai báo **protected.** Các thành phần **protected** có phạm vi truy nhập rộng hơn so với các thành phần private, nhưng hẹp hơn so với các thành phần public.

Các thành phần **protected** của lớp cơ sở hoàn toàn giống các thành phần private *ngoại trừ một điểm* là chúng có thể kế thừa từ lớp dẫn xuất trực tiếp từ lớp cơ sở. Cụ thể như sau:

- Nếu kế thừa theo kiểu public thì các thành phần proteted của lớp cơ sở sẽ trở thành các thành phần protected của lớp dẫn xuất.

- Nếu kế thừa theo kiểu private thì các thành phần proteted của lớp cơ sở sẽ trở thành các thành phần private của lớp dẫn xuất.

4.2.7. Dẫn xuất protected

Ngoài hai kiểu dẫn xuất đã biết là private và public, C++ còn đưa ra kiểu dẫn xuất **protected**. Trong dẫn xuất loại này thì các thành phần public, protected trong lớp cơ sở trở thành các thành phần **protected** trong lớp dẫn xuất.

4.3. Đa kế thừa

4.3.1. Định nghĩa lớp dẫn xuất từ nhiều lớp cơ sở

Giả sử đã định nghĩa các lớp A, B. Cú pháp để xây dựng lớp C dẫn xuất từ lớp A và B như sau:

```
        class  C: mode A, mode B
    {
       private:
            // Khai báo các thuộc tính
       public:
      // Các hàm thành phần
};
```

trong đó mode có thể là private, public hoặc protected. Ý nghĩa của kiểu dẫn xuất này giống như trường hợp đơn kế thừa.

4.3.2. Một số ví dụ về đa kế thừa

Ví dụ 5.7 Chương trình sau minh họa một lớp kế thừa từ hai lớp cơ sở:

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

class M
{ protected :
int m;
  public :
void getm(int x) {m=x;}
};
class N
{ protected :
int n;
  public :
void getn(int y) {n=y;}
};

class P : public M,public N
{
  public :
void display(void)
```

```
    { cout<<"m= "<<m<<endl;
      cout<<"n= "<<n<<endl;
      cout<<"m * n = "<<m*n<<endl;
    }
};
void main()
{ P ob;
  clrscr();
  ob.getm(10);
  ob.getn(20);
  ob.display();
  getch();
}
```

Chương trình cho kết quả như sau:

```
m = 10
n = 20
m*n = 200
```

Ví dụ Chương trình sau minh họa việc quản lý kết quả thi của một lớp không quá 100 sinh viên. Chương trình gồm 3 lớp: lớp cơ sở sinh viên (sinhvien) chỉ lưu họ tên và số báo danh, lớp điểm thi (diemthi) kế thừa lớp sinh viên và lưu kết quả môn thi 1 và môn thi 2. Lớp kết quả (ketqua) lưu tổng số điểm đạt được của sinh viên.

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

class sinhvien
{ char hoten[25];
  protected:
int sbd;
  public:
void nhap()
      {  cout<<"\nHo ten :";gets(hoten);
           cout<<"So bao danh :";cin>>sbd;
         }
   void hienthi()
       { cout<<"So bao danh : "<<sbd<<endl;
```

```
    cout<<"Ho va ten sinh vien : "<<hoten<<endl;
          }
};

class diemthi : public sinhvien
{ protected :
float d1,d2;
  public :
void nhap_diem()
    {
      cout<<"Nhap diem hai mon thi : \n";
      cin>>d1>>d2;
    }
void hienthi_diem()
       {  cout<<"Diem mon 1 :"<<d1<<endl;
       cout<<"Diem mon 2 :"<<d2<<endl;
       }
};
class ketqua : public diemthi
{
  float tong;
  public :
void display()
    { tong = d1+d2;
      hienthi();
      hienthi_diem();
      cout<<"Tong so diem :"<<tong<<endl;
    }
};
void main()
{ int i,n; ketqua sv[100];
  cout<<"\n Nhap so sinh vien : ";
  cin>>n;
  clrscr();
  for(i=0;i<n;++i)
  { sv[i].nhap();
```

```
   sv[i].nhap_diem();
  }
  for(i=0;i<n;++i)
   sv[i].display();
  getch();
}
```

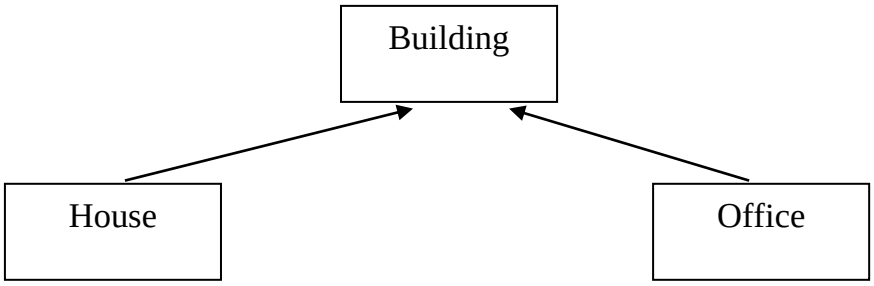
Ví dụ Chương trình sau là sự mở rộng của chương trình ở trên, trong đó ngoài kết quả hai thi, mỗi sinh viên còn có thể có điểm thưởng. Chương trình mở rộng thêm một lớp ưu tiên (uutien).

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
class sinhvien
{ char hoten[25];
  protected : int sbd;
  public    :
void nhap()
      {  cout<<"\nHo ten :";gets(hoten);
           cout<<"So bao danh :";cin>>sbd;
         }
    void hienthi()
       { cout<<"So bao danh : "<<sbd<<endl;
         cout<<"Ho va ten sinh vien : "<<hoten<<endl;
        }
};
class diemthi : public sinhvien
{ protected : float d1,d2;
  public :
void nhap_diem()
    {
      cout<<"Nhap diem hai mon thi : \n";
      cin>>d1>>d2;
    }
void hienthi_diem()
      {  cout<<"Diem mon 1 :"<<d1<<endl;
       cout<<"Diem mon 2 :"<<d2<<endl;
```

```
      }
};
class uutien
{ protected : float ut;
  public :
void nhap_ut()
    {
      cout<<"\nNhap diem uu tien :";cin>>ut;
      }
   void hienthi_ut()
    {cout<<"Diem uu tien : "<<ut<<endl; }
};
class ketqua : public diemthi, public uutien
{ float tong;
  public :
void display()
   { tong=d1+d2+ut;
     hienthi();
     hienthi_diem();
     hienthi_ut();
     cout<<"Tong so diem :"<<tong<<endl;
   }
};
void main()
{ int i,n; ketqua sv[100];
  cout<<"\n Nhap so sinh vien : ";
  cin>>n;
  clrscr();
  for(i=0;i<n;++i)
  { sv[i].nhap();
    sv[i].nhap_diem();
    sv[i].nhap_ut();
  }
  for(i=0;i<n;++i)
    sv[i].display();
  getch();
```

}

Ví dụ Xem sơ đồ kế thừa các lớp như sau:



Hình 5.3.

Trong đó lớp cơ sở Building lưu trữ số tầng của một tòa nhà, tổng số phòng và tổng diện tích của tòa nhà. Lớp dẫn xuất House kế thừa lớp Building và lưu trữ số phòng ngủ, số phòng tắm. Lớp dẫn xuất Office từ lớp Building lưu trữ số máy điện thoại và số bình cứu hỏa. Chương trình sau minh họa việc tổ chức lưu trữ theo sơ đồ kế thừa này.

```
#include <iostream.h>
#include <conio.h>
class Building
{ protected :
int floors; //tong so  tang
    int rooms; //tong so phong
    double footage; //tong so dien tich
};
class house : public Building
{       int bedrooms; //tong so phong ngu
                int bathrooms; //tong so phong tam
          public :
house(int f, int r, int ft, int br, int bth)
        { floors=f; rooms=r; footage=ft;
          bedrooms=br; bathrooms=bth;
      }
        void show()
       { cout<<'\n';
         cout<<" So tang  : " <<floors <<'\n';
         cout<<" So phong : " <<rooms <<'\n';
       cout<<" So tong dien tich : "
          <<footage<<'\n';
```

```
        cout<<" So phong ngu : " <<bedrooms <<'\n';
        cout<<" So phong tam : " <<bathrooms<<'\n';
        }
};
class office : public Building
{       int phones; //tong so may dien thoai
                int extis; //tong so binh cuu hoa
          public :
office(int f, int r, int ft, int p, int ext)
        { floors=f; rooms=r; footage=ft;
          phones=p; extis=ext;
        }
         void show()
       { cout<<'\n';
         cout<<" So tang  : " <<floors <<'\n';
         cout<<" So phong : " <<rooms <<'\n';
        cout<<" So tong dien tich : " <<footage
           <<"\n”;
        cout<<" So may dien thoai : " <<phones
           << "\n";
        cout<<" So binh cuu hoa : " <<extis<<'\n';
       }
};
void main()
{
 house h_ob(2,12,5000,6,4);
 office o_ob(4,25,12000,30,8);
 cout<<"House  : ";
 h_ob.show();
 cout<<"Office : ";
 o_ob.show();
 getch();
}
```

Chương trình cho kết quả như sau:

```
House  :
 So tang  : 2
```

So phong : 12
So tong dien tich : 5000
So phong ngu : 6
So phong tam : 4
Office :
So tang : 4
So phong : 25
So tong dien tich : 12000
So may dien thoai : 30
So binh cuu hoa : 8

4.4. Hàm ảo

4.4.1 Đặt vấn đề

Trước khi đưa ra khái niệm về hàm ảo, ta hãy thảo luận ví dụ sau:
Giả sử có 3 lớp A, B và C được xây dựng như sau:

```
class A
        {
          public:
      void xuat()
       {
        cout <<"\n Lop A";
       }
  };
class B : public A
        {
         public:
           void xuat()
           {
                cout <<"\n Lop B";
         }
  };
class C : public B
  {
         public:
           void xuat()
         {
         cout <<"\n Lop C";
```

```
        }
    };
```

Cả 3 lớp này đều có hàm thành phần là xuat(). Lớp C có hai lớp cơ sở là A, B và C kế thừa các hàm thành phần của A và B. Do đó một đối tượng của C sẽ có 3 hàm xuat(). Xem các câu lệnh sau:

```
C  ob; //  ob là đối tượng kiểu C
ob.xuat();      // Gọi tới hàm thành phần xuat() của lớp D
ob.B::xuat() ;  // Gọi tới hàm thành phần xuat() của lớp B
ob.A::xuat() ;  // Gọi tới hàm thành phần xuat() của lớp A
```

Các lời gọi hàm thành phần trong ví dụ trên đều xuất phát từ đối tượng ob và mọi lời gọi đều *xác định rõ* hàm cần gọi.

Ta xét tiếp tình huống các lời gọi không phải từ một biến đối tượng mà từ một con trỏ đối tượng. Xét các câu lệnh:

```
A   *p,  *q, *r;   // p,q,r là các con trỏ kiểu A
A  a;              // a là đối tượng kiểu A
B  b;              // b là đối tượng kiểu B
C  c;              // c là đối tượng kiểu C
```

Bởi vì con trỏ của lớp cơ sở có thể dùng để chứa địa chỉ các đối tượng của lớp dẫn xuất, nên cả 3 phép gán sau đều hợp lệ:

```
p = &a;     q = &b;     r = &c;
```

Ta xét các lời gọi hàm thành phần từ các con trỏ p, q, r:

```
p->xuat();          q->xuat();          r->xuat();
```

Cả 3 câu lệnh trên đều gọi tới hàm thành phần xuat() của lớp A, bởi vì các con trỏ p, q, r đều có kiểu lớp A. Sở dĩ như vậy là vì một lời gọi (xuất phát từ một đối tượng hay con trỏ) tới hàm thành phần **luôn luôn liên kết với một hàm thành phần cố định** và sự liên kết này xác định trong quá trình biên dịch chương trình. Ta bảo đây là **sự liên kết tĩnh**.

Có thể tóm lược cách thức gọi các hàm thành phần như sau:

1. Nếu lời gọi xuất phát từ một đối tượng của lớp nào đó, thì hàm thành phần của lớp đó sẽ được gọi.

2. Nếu lời gọi xuất phát từ một con trỏ kiểu lớp, thì hàm thành phần của lớp đó sẽ được gọi bất kể con trỏ chứa địa chỉ của đối tượng nào.

Vấn đề đặt ra là: Ta muốn tại thời điểm con trỏ đang trỏ đến đối tượng nào đó thì lời gọi hàm phải liên kết đúng hàm thành phần của lớp mà đối tượng đó thuộc vào chứ không phụ thuộc vào kiểu lớp của con trỏ. C++ giải quyết vấn đề này bằng cách dùng khái niệm *hàm ảo*.

4.4.2. Định nghĩa hàm ảo

Hàm ảo là hàm thành phần của lớp, nó được *khai báo trong lớp cơ sở* và định nghĩa lại trong lớp dẫn xuất. Để định nghĩa hàm ảo thì phần khai báo hàm phải bắt đầu bằng từ khóa *virtual*. Khi một lớp có chứa hàm ảo được kế thừa, lớp dẫn xuất sẽ định nghĩa lại hàm ảo đó cho chính mình. Các hàm ảo triển khai tư tưởng chủ đạo của tính đa hình là " một giao diện cho nhiều hàm thành phần". Hàm ảo bên trong lớp cơ sở định nghĩa hình thức giao tiếp đối với hàm đó. Việc định nghĩa lại hàm ảo ở lớp dẫn xuất là thi hành các tác vụ của hàm liên quan đến chính lớp dẫn xuất đó. Nói cách khác, định nghĩa lại hàm ảo chính là tạo ra phương thức cụ thể. Trong phần định nghĩa lại hàm ảo ở lớp dẫn xuất, không cần phải sử dụng lại từ khóa virtual.

Khi xây dựng hàm ảo, cần tuân theo những quy tắc sau :

1.Hàm ảo phải là hàm thành phần của một lớp ;

2.Những thành phần tĩnh (static) không thể khai báo ảo;

3.Sử dụng con trỏ để truy nhập tới hàm ảo;

4.Hàm ảo được định nghĩa trong lớp cơ sở, ngay khi nó không được sử dụng;

5.Mẫu của các phiên bản (ở lớp cơ sở và lớp dẫn xuất) phải giống nhau. Nếu hai hàm cùng tên nhưng có mẫu khác nhau thì C++ sẽ xem như hàm tải bội;

6.Không được tạo ra hàm tạo ảo, nhưng có thể tạo ra hàm hủy ảo;

7.Con trỏ của lớp cơ sở có thể chứa địa chỉ của đối tượng thuộc lớp dẫn xuất, nhưng ngược lại thì không được;

8.Nếu dùng con trỏ của lớp cơ sở để trỏ đến đối tượng của lớp dẫn xuất thì phép toán tăng giảm con trỏ sẽ không tác dụng đối với lớp dẫn xuất, nghĩa là không phải con trỏ sẽ trỏ tới đối tượng trước hoặc tiếp theo trong lớp dẫn xuất. Phép toán tăng giảm chỉ liên quan đến lớp cơ sở.

Ví dụ:

```
class A
        {
          ...
  virtual void hienthi()
     {
                cout<<"\nDay la lop A";
     };
};
class B : public A
        {
          ...
```

```
  void hienthi()
     {
          cout<<"\nDay la lop B";
     }
};
class C : public B
      {
        ...
  void hienthi()
     {
              cout<<"\nDay la lop C";
     }
};
class D : public A
       { ...
  void hienthi()
     {
          cout<<"\nDay la lop D";
     }
};
```

Chú ý: Từ khoá virtual không được đặt bên ngoài định nghĩa lớp. Xem ví dụ :

```
class A
 {
   ...
  virtual void hienthi();
 };
virtual void hienthi()  // sai
 {
   cout<<"\nDay la lop A";
  }
```

4.4.3. Quy tắc gọi hàm ảo

Hàm ảo chỉ khác hàm thành phần thông thường khi được gọi từ một con trỏ. Lời gọi tới hàm ảo từ một con trỏ chưa cho biết rõ hàm thành phần nào (trong số các hàm thành phần cùng tên của các lớp có quan hệ thừa kế) sẽ được gọi. Điều này sẽ *phụ thuộc vào đối tượng cụ thể* mà con trỏ đang trỏ tới: *con trỏ đang trỏ tới đối tượng của lớp nào thì hàm thành phần của lớp đó sẽ được gọi.*

4.4.5. Quy tắc gán địa chỉ đối tượng cho con trỏ lớp cơ sở

C++ cho phép gán địa chỉ đối tượng của một lớp dẫn xuất cho con trỏ của lớp cơ sở bằng cách sử dụng phép gán = và phép toán lấy địa chỉ &.

Ví dụ : Giả sử A là lớp cơ sở và B là lớp dẫn xuất từ A. Các phép gán sau là đúng:

```
A  *p;  // p là con trỏ kiểu A
A   a;  // a là biến đối tượng kiểu A
B   b;  // b là biến đối tượng kiểu B
p = &a;  // p và a cùng lớp A
p = &b;  // p là con trỏ lớp cơ sở, b là đối tượng lớp dẫn xuất
```

Chú ý: Không cho phép gán địa chỉ đối tượng của lớp cơ sở cho con trỏ của lớp dẫn xuất, chẳng hạn với khai báo B *q; A a; thì câu lệnh q = &a; là sai.

Ví dụ Chương trình sau đây minh họa việc sử dụng hàm ảo:

```
#include <iostream.h>
#include <conio.h>
class A
 {
    public:
        virtual void hienthi()
         {
               cout <<"\n Lop A";
          }
 };
class B : public A
 {
    public:
        void hienthi()
         {
               cout <<"\n Lop B";
           }
  };
class C : public B
 {
    public:
        void hienthi()
         {
```

```
            cout <<"\n Lop C";
        }
};
void main()
{ clrscr();
  A *p;
  A a; B b; C c;
  a.hienthi();      //goi ham cua lop A
  p = &b;           //p tro to doi tuong b cua lop B
  p->hienthi();     //goi ham cua lop B
  p=&c;             //p tro to doi tuong c cua lop C
  p->hienthi();     //goi ham cua lop C
  getch();
}
```

Chương trình này cho kết quả như sau:

Lop A

Lop B

Lop C

Chú ý :

- Cùng một câu lệnh p->hienthi(); được tương ứng với nhiều hàm khác nhau khác nhau khi hienthi() là hàm ảo. Đây chính là sự *tương ứng bội*. Khả năng này cho phép xử lý nhiều đối tượng khác nhau theo cùng một cách thức.
- Cũng với lời gọi: p->hienthi(); (hienthi() là hàm ảo) thì lời gọi này không liên kết với một phương thức cố định, mà tùy thuộc và nội dung con trỏ. Đó là *sự liên kết động* và phương thức được liên kết (được gọi) thay đổi mỗi khi có sự thay đổi nội dung con trỏ trong quá trình chạy chương trình.

Ví dụ Chương trình sau tạo ra một lớp cơ sở có tên là ***num*** lưu trữ một số nguyên, và một hàm ảo của lớp có tên là *shownum*(). Lớp ***num*** có hai lớp dẫn xuất là ***outhex*** và ***outoct***. Trong hai lớp này sẽ định nghĩa lại hàm ảo *shownum*() để chúng in ra số nguyên dưới dạng số hệ 16 và số hệ 8.

```
#include <iostream.h>
#include <conio.h>
class num
{
    public :
```

```
        int i;
        num(int x) { i=x; }
        virtual void shownum()
     { cout<<"\n So he 10 : ";
       cout <<dec<<i<<'\n';
         }
};
class outhex : public num
{   public :
        outhex(int n) : num(n) {}
        void shownum()
       { cout <<"\n So he 10 : "<<dec<<i<<endl;
      cout <<"\n So he 16 : "<<hex << i <<'\n';
       }
};
class outoct : public num
{   public :
        outoct(int n) :  num(n) {}
        void shownum()
       { cout <<"\n So he 10 : "<<dec<<i<<endl;
     cout <<"\n So he 8  : "<< oct << i <<'\n';
        }
};
void main()
{ clrscr();
  num n (1234);
  outoct o (342);
  outhex h (747);
  num *p;
  p=&n;
  p->shownum(); //goi ham cua lop co so, 100
  p=&o;
  p->shownum(); //goi ham cua lop dan xuat, 12
  p=&h;
  p->shownum(); //goi ham cua lop dan xuat, f
  getch();
```

}

Chương trình trên cho kết quả:

So he 10 : 1234

So he 10 : 342

So he 8 : 526

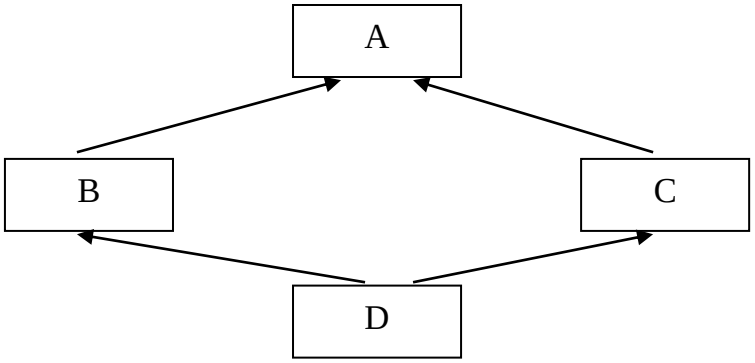
So he 10 : 747

So he 16 : 2eb

4.5. Lớp cơ sở ảo

4.5.1. Khai báo lớp cơ sở ảo

Một vấn đề tồn tại là khi nhiều lớp cơ sở được kế thừa trực tiếp bởi một lớp dẫn xuất. Để hiểu rõ hơn vấn đề này, xét tình huống các lớp kế thừa theo sơ đồ như sau:



Hình 5.4.

Ở đây, lớp A được kế thừa bởi hai lớp B và C. Lớp D kế thừa trực tiếp cả hai lớp B và C. Như vậy lớp A được kế thừa *hai lần* bởi lớp D: lần thứ nhất nó được kế thừa thông qua lớp B, và lần thứ hai được kế thừa thông qua lớp C. Bởi vì có hai bản sao của lớp A có trong lớp D nên một tham chiếu đến một thành phần của lớp A sẽ tham chiếu về lớp A được kế thừa gián tiếp thông qua lớp B hay tham chiếu về lớp A được kế thừa gián tiếp thông qua lớp C? Để giải quyết tính không rõ ràng này, C++ có một cơ chế mà nhờ đó chỉ có một bản sao của lớp A ở trong lớp D: đó là sử dụng lớp *cơ sở ảo.*

Trong ví dụ trên, C++ sử dụng từ khóa vitual để khai báo lớp A là ảo trong các lớp B và C theo mẫu như sau:

```
clacc B : virtual public A
        {...};
       clacc C : virtual public A
        {...};
clacc D : public B, public C
        {...};
```

Việc chỉ định A là ảo trong các lớp B và C nghĩa là A sẽ chỉ xuất hiện một lần trong lớp D. Khai báo này không ảnh hưởng đến các lớp B và C.

Chú ý: Từ khóa virtual có thể đặt trước hoặc sau từ khóa public, private, protected.

Ví dụ

```
#include <iostream.h>
#include <conio.h>
class A
 {   float x,y;
     public:
    void set(float x1, float y1)
     {
        x = x1; y = y1;
      }
    float getx()
     { return x;
       }
    float gety()
     { return y;
     }
 };
class B : virtual public A
 { };
class C : virtual public A
 { };
class D : public B, public C
 { };
 void main()
 { clrscr();
   D d;
   cout<<"\nd.B::set(2,3)\n";
   d.B::set(2,3);
   cout<<"\nd.C::getx() = "; cout<<d.C::getx()<<endl;
   cout<<"\nd.B::getx() = "; cout<<d.B::getx()<<endl;
   cout<<"\nd.C::gety() = "; cout<<d.C::gety()<<endl;
   cout<<"\nd.B::gety() = "; cout<<d.B::gety()<<endl;
```

```
    cout<<"\nd.C::set(2,3)\n";
    d.C::set(2,3);
    cout<<"\nd.C::getx() = "; cout<<d.C::getx()<<endl;
    cout<<"\nd.B::getx() = "; cout<<d.B::getx()<<endl;
    cout<<"\nd.C::gety() = "; cout<<d.C::gety()<<endl;
    cout<<"\nd.B::gety() = "; cout<<d.B::gety()<<endl;
    getch();
   }
```

Chương trình trên sẽ cho kết quả:

d.B::set(2,3)
d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3

d.C::set(2,3)
d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3

4.5.2. Hàm tạo và hàm hủy đối với lớp cơ sở ảo

Ta đã biết, khi khởi tạo đối tượng lớp dẫn xuất thì các hàm tạo được gọi theo thứ tự xuất hiện trong danh sách các lớp cơ sở được khai báo, rồi đến hàm tạo của lớp dẫn xuất. Thông tin được chuyển từ hàm tạo của lớp dẫn xuất sang hàm tạo của lớp cơ sở. Trong tình huống có lớp cơ sở ảo, chẳng hạn như hình vẽ 5.4., cần phải tuân theo quy định sau:

Thứ tự gọi hàm tạo: Hàm tạo của một lớp ảo luôn luôn được gọi trước các hàm tạo khác.

Với sơ đồ kế thừa như hình vẽ 5.4., thứ tự gọi hàm tạo sẽ là A, B, C và cuối cùng là D. Chương trình sau minh họa điều này:

Ví dụ

```
#include <iostream.h>
#include <conio.h>
class A
 {   float x,y;
     public:
```

```
    A() {x = 0; y = 0;}
    A(float x1, float y1)
     {
        cout<<"A::A(float,float)\n";
        x = x1; y = y1;
     }
    float getx()
     {
  return x;
      }
    float gety()
     {
          return y;
      }

   };
class B : virtual public A
 {
     public:
    B(float x1, float y1):A(x1,y1)
     {
       cout<<"B::B(float,float)\n";
      }
  };
class C : virtual public A
  {
     public:
C(float x1, float y1):A(x1,y1)
     {
       cout<<"C::C(float,float)\n";
      }
  };
class D : public B, public C
 {
     public:
D(float x1, float y1):A(x1,y1), B(10,4), C(1,1)
```

```
    {
      cout<<"D::D(float,float)\n";
    }
     };
    void main()
    { clrscr();
      D d(2,3);
      cout<<"\nD d (2,3)\n";
      cout<<"\nd.C::getx() = "; cout<<d.C::getx()<<endl;
      cout<<"\nd.B::getx() = "; cout<<d.B::getx()<<endl;
      cout<<"\nd.C::gety() = "; cout<<d.C::gety()<<endl;
      cout<<"\nd.B::gety() = "; cout<<d.B::gety()<<endl;
      cout<<"\nd1 (10,20) \n";
      D d1 (10,20);   cout<<"\nd.C::getx() = ";
cout<<d.C::getx()<<endl;
      cout<<"\nd.B::getx() = "; cout<<d.B::getx()<<endl;
      cout<<"\nd.C::gety() = "; cout<<d.C::gety()<<endl;
      cout<<"\nd.B::gety() = "; cout<<d.B::gety()<<endl;
      getch();
     }
```

Kết quả chương trình trên như sau:

A::A(float,float)
B::B(float,float)
C::C(float,float)
D::D(float,float)
D d (2,3)
d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3
d1 (10,20)
A::A(float,float)
B::B(float,float)
C::C(float,float)
D::D(float,float)

d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3

BÀI TẬP

1. Xây dựng lớp có tên là Stack với các thao tác cần thiết. Từ đó hãy dẫn xuất từ lớp stack để đổi một số nguyên dương sang hệ đếm bất kỳ.

2. Viết một phân cấp kế thừa cho các lớp hình tứ giác, hình thang, hình bình hành, hình chữ nhật, hình vuông.

3. Tạo một lớp cơ sở có tên là airship để lưu thông tin về số lượng hành khách tối đa và trọng lượng hàng hóa tối đa mà máy bay có thể chở được. Từ đó hãy tạo hai lớp dẫn xuất airplane và balloon, lớp airplane lưu thông tin về kiểu động cơ (gồm động cơ cánh quạt và động cơ phản lực), lớp balloon lưu thông tin về loại nhiên liệu sử dụng cho khí cầu (gồm hai loại là hydrogen và helium). Hãy viết chương trình minh họa.

4. Một nhà xuất bản nhận xuất bản sách. Sách có hai loại: loại có hình ảnh ở trang bìa và loại không có hình ảnh ở trang bìa. Loại có hình ảnh ở trang bìa thì phải thuê họa sĩ vẽ bìa. Viết chương trình thực hiện các yêu cầu :

- Tạo một lớp cơ sở có tên là SACH để lưu thông tin về tên sách, tác giả, số trang, giá bán và định nghĩa hàm thành phần cho phép nhập dữ liệu cho các đối tượng của lớp SACH.

- Tạo lớp BIA kế thừa từ lớp SACH để lưu các thông tin : Mã hình ảnh, tiền vẽ và định nghĩa hàm thành phần cho phép nhập dữ liệu cho các đối tượng của lớp BIA.

- Tạo lớp HOASY để lưu các thông tin họ tên, địa chỉ của họa sỹ và định nghĩa hàm thành phần cho phép nhập dữ liệu cho các đối tượng của lớp HOASY.

- Tạo lớp SACHVEBIA kế thừa từ lớp BIA và lớp HOASY và định nghĩa hàm thành phần cho phép nhập dữ liệu cho các đối tượng của lớp SACHVEBIA. Viết hàm main() cho phép nhập vào hai danh sách : danh sách các sách có vẽ bìa và danh sách các sách không có vẽ bìa (có thể dùng mảng tĩnh hoặc mảng con trỏ).

5. Xây dựng lớp hình vuông có tên là HVUONG với các dữ liệu thành phần như sau: độ dài cạnh. Các hàm thành phần để nhập dữ liệu, hiển thị dữ liệu, tính diện tích, chu vi hình vuông. Từ lớp HVUONG, xây dựng lớp dẫn xuất có tên là CHUNHAT, là lớp kế thừa của lớp HVUONG và được bổ sung thêm thuộc tính sau: độ dài cạnh thứ hai. Các hàm thành phần để nhập dữ liệu, hiển thị dữ liệu để tính diện tích và chu vi hình chữ nhật. Viết chương trình minh họa.

6. Xây dựng lớp cơ sở CANBO có dữ liệu thành phần là mã cán bộ, mã đơn vị, họ tên, ngày sinh. Các hàm thành phần bao gồm: nhập dữ liệu cán bộ, hiển thị dữ liệu. Lớp dẫn xuất LUONG kế thừa lớp CANBO và có thêm các thuộc tính: phụ cấp, hệ số lương, bảo hiểm. Hàm thành phần để tính lương cán bộ theo công thức:

lương = hệ số lương *290000 + phụ cấp - bảo hiểm

Hãy thiết kế chương trình để đáp ứng các yêu cầu:

- Nhập danh sách cán bộ

- In bảng lương các cán bộ theo từng đơn vị.

7. Nhân viên trong một cơ quan được lĩnh lương theo các dạng khác nhau. Dạng người lao động hưởng lương từ ngân sách Nhà nước gọi là cán bộ, công chức (dạng biên chế). Dạng người lao động lĩnh lương từ ngân sách của cơ quan gọi là người làm hợp đồng. Như vậy hệ thống có hai đối tượng: biên chế và hợp đồng.

- Hai loại đối tượng này có đặc tính chung là viên chức làm việc cho cơ quan. Từ đây có thể tạo nên lớp cơ sở để quản lý một viên chức (lớp **Nguoi**) bao gồm mã số, họ tên, lương.

- Hai lớp kế thừa từ lớp cơ sở trên:

+ Lớp Bienche gồm các thuộc tính: hệ số lương, tiền phụ cấp chức vụ.

+ Lớp Hopdong gồm các thuộc tính: tiền công lao động, số ngày làm việc trong tháng, hệ số vượt giờ.

Hãy thiết kế các lớp trên và viết chương trình minh họa.

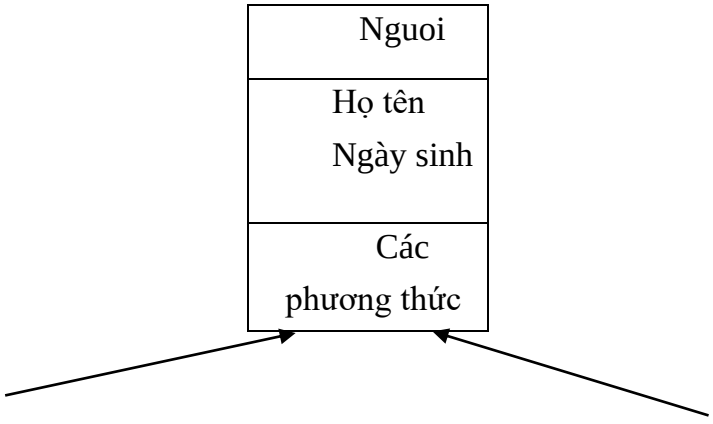
8. Viết chương trình quản lý sách báo, tạp chí của thư viện trong trường đại học, hằng tháng gởi về khoa họ tên của các giáo viên và sinh viên đã quá thời hạn mượn sách.

9. Viết chương trình tính và in bảng lương hàng tháng của giáo viên và người làm hợp đồng trong một trường đại học. Giả sử việc tính tóan tiền lương được căn cứ vào các yếu tố sau:

- Đối với giáo viên: số tiết dạy trong tháng, tiền dạy một tiết.

- Đối với người làm hợp đồng: tiền công ngày, số ngày làm việc

10. Giả sử cuối năm học cần trao phần thưởng cho các sinh viên giỏi, các giáo viên có tham gia nghiên cứu khoa học. Điều kiện khen thưởng của sinh viên là có điểm trung bình lớn hơn 8. Điều kiện khen thưởng của giáo viên là có ít nhất một bài báo nghiên cứu khoa học. Sơ đồ cấu trúc phân cấp lớp như sau:



Sinhvien
Lớp Điểm trung bình
Các phương thức

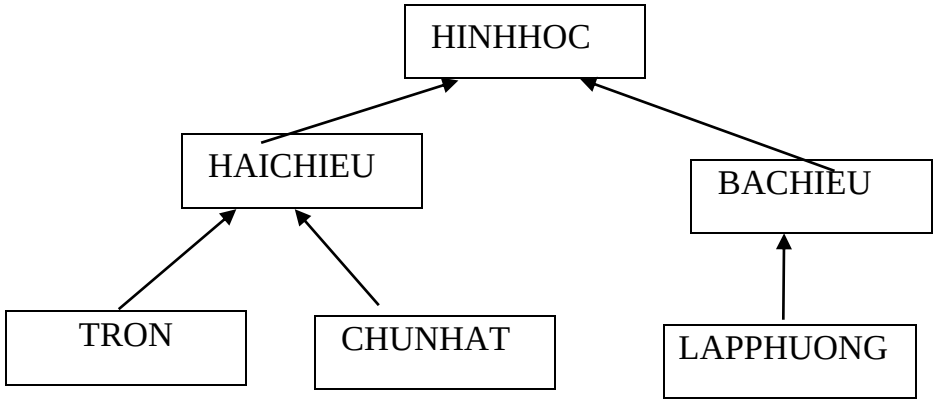
Giaovien
Bộ môn Số bài báo
Các phương thức

Yêu cầu:

- Xây dựng các lớp theo sơ đồ kề thừa ở trên, mỗi lớp có các hàm thành phần để nhập, xuất dữ liệu, hàm kiểm tra khen thưởng.

- Hãy viết hàm main() cho phép nhập vào dữ liệu của không quá 100 sinh viên và không quá 30 giáo viên. In ra danh sách sinh viên và giáo viên được khen thưởng.

11. Giả sử ta có sơ đồ kế thừa của các lớp như sau:



Yêu cầu:

- Thiết kế các lớp để có thể in ra các thông tin của các hình (tròn, chữ nhật, lập phương) bao gồm: diện tích, chu vi, thể tích.
- Viết chương trình minh họa.

12. Viết chương trình quản lý việc mượn và trả sách ở một thư viện theo phương pháp lập trình hướng đối tượng. Chương trình cho phép:

- Đăng ký bạn đọc mới với thông tin là mã và tên bạn đọc, số điện thoại
- Nhập sách mới với thông tin là mã sách, tên sách, số lượng, nhà xuất bản.
- Mượn và trả sách.
- Thống kê bạn đọc.
- Thống kê sách.

CHƯƠNG 5

KHUÔN HÌNH

5.1. Khuôn hình hàm

5.1.1. Khái niệm

Ta đã biết hàm quá tải cho phép dùng một tên duy nhất cho nhiều hàm để thực hiện các công việc khác nhau. Khái niệm khuôn hình hàm cũng cho phép sử dụng cùng một tên duy nhất để thực hiện các công việc khác nhau, tuy nhiên so với định nghĩa hàm quá tải, nó có phần mạnh hơn và chặt chẽ hơn. Mạnh hơn vì chỉ cần viết định nghĩa khuôn hình hàm một lần, rồi sau đó chương trình biên dịch làm cho nó thích ứng với các kiểu dữ liệu khác nhau. Chặt chẽ hơn bởi vì dựa theo khuôn hình hàm, tất cả các hàm thể hiện được sinh ra bởi chương trình dịch sẽ tương ứng với cùng một định nghĩa và như vậy sẽ có cùng một giải thuật.

5.1.2. Tạo một khuôn hình hàm

Giả thiết rằng chúng ta cần viết một hàm ***min*** đưa ra giá trị nhỏ nhất trong hai giá trị có cùng kiểu. Ta có thể viết một định nghĩa như thế với kiểu **int** như sau:

```
int min (int a, int b)
{
if (a<b)
          return a;
else
          return b;
 }
```

Nếu ta muốn sử dụng hàm min cho kiểu double, float, char,.... ta lại phải viết lại định nghĩa hàm **min**, ví dụ:

```
float min  (float a, float b)
{
    if  (a < b)
          return a;
    else
          return b;
 }
```

Như vậy ta phải viết rất nhiều định nghĩa hàm hoàn toàn tương tự nhau, chỉ có kiểu dữ liệu là thay đổi. Chương trình dịch C++ cho phép giải quyết đơn giản vấn đề trên bằng cách định nghĩa một khuôn hình hàm duy nhất theo cú pháp:

```
template  <danh sách tham số kiểu>  <kiểu trả về> tên hàm(khai báo tham số)
```

```
    {
     // định nghĩa hàm
}
```

trong đó <danh sách tham số kiểu> là các kiểu dữ liệu được khai báo với từ khoá *class*, cách nhau bởi dấu phẩy. Kiểu dữ liệu là một kiểu bất kỳ, kể cả kiểu *class*.

Ví dụ Xây dựng khuôn hình cho hàm tìm giá trị nhỏ nhất của hai số:

```
template <class Kieuso> Kieuso min(Kieuso a, Kieuso b)
{
if  (a<b)
          return a;
else
       return b;
}
```

5.1.3. Sử dụng khuôn hình hàm

Để sử dụng khuôn hình hàm ***min*** vừa tạo ra, chỉ cần sử dụng hàm *min* trong những điều kiện phù hợp, trong trường hợp này là hai tham số của hàm phải cùng kiểu dữ liệu. Như vậy, nếu trong một chương trình có hai tham số nguyên n và m (kiểu int) với lời gọi min(n,m) thì chương trình dịch tự động sản sinh ra hàm min(), gọi là một hàm thể hiện, tương ứng với hai tham số kiểu int. Nếu chúng ta gọi min() với hai tham số kiểu float, chương trình biên dịch cũng tự động sản sinh một hàm thể hiện min khác tương ứng với các tham số kiểu float và cứ thế với các kiểu dữ liệu khác.

Chú ý:

- Các biến truyền cho danh sách tham số của hàm phải chính xác với kiểu khai báo.

- Muốn áp dụng được với kiểu lớp thì trong lớp phải định nghĩa các toán tử tải bội tương ứng.

5.1.4. Các tham số kiểu của khuôn hình hàm

Khuôn hình hàm có thể có một hay nhiều tham số kiểu, mỗi tham số đi liền sau từ khoá class. Các tham số này có thể ở bất kỳ đâu trong định nghĩa của khuôn hình hàm, nghĩa là :

- Trong dòng tiêu đề (ở dòng đầu khai báo template).

- Trong các khai báo biến cục bộ.

- Trong các chỉ thị thực hiện.

Trong mọi trường hợp, mỗi tham số kiểu phải xuất hiện ít nhất một lần trong khai báo danh sách tham số hình thức của khuôn hình hàm. Điều đó hoàn toàn logic, bởi vì nhờ các tham số này, chương trình dịch mới có thể sản sinh ra hàm thể hiện cần thiết.

Ở khuôn hình hàm min trên, mới chỉ cho phép tìm min của hai số cùng kiểu, nếu muốn tìm min hai số khác kiểu thì khuôn hình hàm trên chưa đáp ứng được. Ví dụ sau sẽ khắc phục được điều này.

Ví dụ

```
#include <iostream.h>
template <class kieuso1,class kieuso2> kieuso1
         min(kieuso1 a,kieuso2 b)
{
  return a<b ? a : b;
}
void main(){
    float a =2.5;
    int b = 8;
    cout << "so nho nhat la :" << min(a,b);
}
```

Ví dụ Giả sử trong lớp SO các số int đã xây dựng, ta có xây dựng các toán tử tải bội < , << cho các đối tượng của class SO (xem chương 4). Nội dung file ttclsso.h như sau:

```
class SO
{
  private:
     int giatri;
  public:
     SO(int x=0)
     {
       giatri = x;
     }
     SO (SO &tso)
     {
giatri = tso.giatri;
     }
     SO (){}; //Giong nhu ham thiet lap ngam dinh
     ~SO()
```

```
      { }
int operator<(SO & s)
{
   return (giatri <s.giatri);
}

friend istream& operator>>(istream&,SO&);
friend ostream& operator<<(ostream&,SO&);
};
```

Chương trình sau đây cho phép thử hàm min trên hai đối tượng kiểu class:

Ví dụ Chương trình sau đây cho phép thử hàm min trên hai đối tượng kiểu class:

```
#include <iostream.h>
#include <ttclsso.h>
template <class kieuso1,class kieuso2> kieuso1
          min(kieuso1 a,kieuso2 b)
 {
  if (a<b)
     return a;
  else
     return b;
 }
void main(){
  float a =2.5;
  int b = 8;
  cout << "so nho nhat la :" << min(a,b)<<endl;
  SO so1(15),so2(20);
  cout << "so nho nhat la :" << min(so2,so1);
}
```

5.1.5. Định nghĩa chồng các khuôn hình hàm

Tương tự việc định nghĩa các hàm quá tải, C++ cho phép định nghĩa chồng các khuôn hình hàm, tức là có thể định nghĩa một hay nhiều khuôn hình hàm có cùng tên nhưng với các tham số khác nhau. Điều đó sẽ tạo ra nhiều họ các hàm (mỗi khuôn hình hàm tương ứng với họ các hàm).

Ví dụ có ba họ hàm min :

- Một họ gồm các hàm tìm giá trị nhỏ nhất trong hai giá trị

- Một họ gồm các hàm tìm giá trị nhỏ nhất trong ba giá trị
- Một họ gồm các hàm tìm giá trị nhỏ nhất trong một mảng giá trị.

Một cách tổng quát, ta có thể định nghĩa một hay nhiều khuôn hình cùng tên, mỗi khuôn hình có các tham số kiểu cũng như là các tham số biểu thức riêng. Hơn nữa, có thể cung cấp các hàm thông thường với cùng tên với cùng một khuôn hình hàm, trong trường hợp này ta nói đó là sự cụ thể hoá một hàm thể hiện.

Trong trường hợp tổng quát khi có đồng thời cả hàm quá tải và khuôn hình hàm, chương trình dịch lựa chọn hàm tương ứng với một lời gọi hàm dựa trên các nguyên tắc sau:

Đầu tiên, kiểm tra tất cả các hàm thông thường cùng tên và chú ý đến sự tương ứng chính xác; nếu chỉ có một hàm phù hợp, hàm đó được chọn; Còn nếu có nhiều hàm cùng thỏa mãn sẽ tạo ra một lỗi biên dịch và quá trình tìm kiếm bị gián đoạn.

Nếu không có hàm thông thường nào tương ứng chính xác với lời gọi, khi đó ta kiểm tra tất cả các khuôn hình hàm có trùng tên với lời gọi, khi đó ta kiểm tra tất cả các khuôn hình hàm có trùng tên với lời gọi; nếu chỉ có một tương ứng chính xác được tìm thấy, hàm thể hiện tương ứng được sản sinh và vấn đề được giải quyết; còn nếu có nhiều hơn một khuôn hình hàm điều đó sẽ gây ra lỗi biên dịch và quá trình dừng.

Cuối cùng, nếu không có khuôn hình hàm phù hợp, ta kiểm tra một lần nữa tất cả các hàm thông thường cùng tên với lời gọi. Trong trường hợp này chúng ta phải tìm kiếm sự tương ứng dựa vào cả các chuyển kiểu cho phép trong C/C++.

5.2. Khuôn hình lớp

5.2.1. Khái niệm

Bên cạnh khái niệm khuôn hình hàm, C++ còn cho phép định nghĩa khuôn hình lớp. Cũng giống như khuôn hình hàm, ở đây ta chỉ cần viết định nghĩa các khuôn hình lớp một lần rồi sau đó có thể áp dụng chúng với các kiểu dữ liệu khác nhau để được các lớp thể hiện khác nhau.

5.2.2. Tạo một khuôn hình lớp

Trong chương trước ta đã định nghĩa cho lớp SO, giá trị các số là kiểu **int**. Nếu ta muốn làm việc với các số kiểi **float**, **double**,... thì ta phải định nghĩa lại một lớp khác tương tự, trong đó kiểu dữ liệu **int** cho dữ liệu **giatri** sẽ được thay bằng **float,double,**...

Để tránh sự trùng lặp trong các tình huống như trên, chương trình dịch C++ cho phép định nghĩa một khuôn hình lớp và sau đó, áp dụng khuôn hình lớp này với các kiểu dữ liệu khác nhau để thu được các lớp thể hiện như mong muốn. Ví dụ :

```
template <class kieuso>  class SO
```

```
{
kieuso giatri;
     public :
    SO (kieuso x =0);
    void Hienthi();
...
};
```

Cũng giống như các khuôn hình hàm, template <class kieuso> xác định rằng đó là một khuôn hình trong đó có một tham số kỉêu kieuso . C++ sử dụng từ khoá **class** chỉ để nói rằng kieuso đại diện cho một kiểu dữ liệu nào đó.

Việc định nghĩa các hàm thành phần của khuôn hình lớp, người ta phân biệt hai trường hợp:

Khi hàm thành phần được định nghĩa bên trong định nghĩa lớp thì không có gì thay đổi.

Khi hàm thành phần được định nghĩa bên ngoài lớp, khi đó cần phải nhắc lại cho chương trình biết các tham số kiểu của khuôn hình lớp, có nghĩa là phải nhắc lại template <class kieuso> chẳng hạn, trước định nghĩa hàm. Ví dụ hàm Hienthi() được định nghĩa ngoài lớp:

```
template <class kieuso> void SO<kieuso>::Hienthi()
 {
   cout <<giatri;
  }
```

5.2.3. Sử dụng khuôn hình lớp

Sau khi một khuôn hình lớp đã được định nghĩa, nó sẽ được dùng để khai báo các đối tượng theo dạng sau :

Tên_lớp <Kiểu> Tên_đối_tượng;

Ví dụ câu lệnh khai báo SO <int> so1; sẽ khai báo một đối tượng so1 có thành phần dữ liệu giatri có kiểu nguyên int.

SO <int> có vai trò như một kiểu dữ liệu lớp; người ta gọi nó là một lớp thể hiện của khuôn hình lớp SO. Một cách tổng quát, khi áp dụng một kiểu dữ liệu nào đó với khuôn hình lớp SO ta sẽ có được một lớp thể hiện tương ứng với kiểu dữ liệu.

Tương tự với các khai báo SO <float> so2; cho phép khai báo một đối tượng so2 mà thành phần dữ liệu giatri có kiểu float.

Ví dụ

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

```
template <class kieuso>  class SO
 {
kieuso giatri;
     public :
SO (kieuso x =0);
void Hienthi(){
    cout<<"Gia tri cua so :"<<giatri<<endl;
}
};
void main(){
clrscr();
SO <int> soint(10); soint.Hienthi();
SO <float> sofl(25.4); sofl.Hienthi();
getch();
}
```

Kết quả trên màn hình là:

Gia tri cua so : 10

Gia tri cua so : 25.4

5.2.4. Các tham số trong khuôn hình lớp

Hoàn toàn giống như khuôn hình hàm, các khuôn hình lớp có thể có các tham số kiểu và tham số biểu thức.

Ví dụ một lớp mà các thành phần có các kiểu dữ liệu khác nhau được khai báo theo dạng:

```
 template <class T, class U,.... class Z>
class <ten lop>{
    T x;
    U y;
    .....
    Z fct1 (int);
.....
};
```

Một lớp thể hiện được khai báo bằng cách liệt kê đằng sau tên khuôn hình lớp các tham số thực, là tên kiểu dữ liệu, với số lượng bằng các tham số trong danh sách của khuôn hình lớp (template<...>)

5.2.5. Tóm tắt

Khuôn hình lớp/hàm là phương tiện mô tả ý nghĩa của một lớp/hàm tổng quát, còn lớp/hàm thể hiện là một bản sao của khuôn hình tổng quát với các kiểu dữ liệu cụ thể.

Các khuôn hình lớp/hàm thường được tham số hoá. Tuy nhiên vẫn có thể sử dụng các kiểu dữ liệu cụ thể trong các khuôn hình lớp/hàm nếu cần.

BÀI TẬP

1. Viết khuôn hình hàm để tìm số lớn nhất của hai số bất kỳ

2. Viết khuôn hình hàm để trả về giá trị trung bình của một mảng, các tham số hình thức của hàm này là tên mảng, kích thước mảng.

3. Cài đặt hàng đợi templete.

4. Viết khuôn hình hàm để sắp xếp kiểu dữ liệu bất kỳ.

5. Xây dựng khuôn hình lớp cho các tọa độ điểm trong mặt phẳng, các thành phần dữ liệu của lớp là toadox, toadoy.

6. Xây dựng khuôn hình lớp cho vector để quản lý các vector có thành phần có kiểu tùy ý.

MỤC LỤC