TRƯỜNG CAO ĐẲNG KỸ THUẬT CAO THẮNG

KHOA ĐIỆN TỬ - TIN HỌC

BÀI GIẢNG

PHƯƠNG PHÁP LẬP TRÌNH

HƯỚNG ĐỐI TƯỢNG

(Lưu hành nội bộ)

TP. HỒ CHÍ MINH, 2019

LỜI GIỚI THIỆU

Phương pháp lập trình hướng đối tượng (object-oriented programming - OOP) là phương pháp lập trình lấy đối tượng (object) 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 (methods) sẽ thể hiện được đúng cách mà chúng ta suy nghĩ, bao quát về thế giới thực. PPLTHĐT 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.

PPLTHĐT là môn học bắt buộc cho tất cả sinh viên ngành công nghệ thông tin, ở hầu các khung chương trình đào tạo của các trường cao đẳng, đại học ở Việt Nam và trên thế giới nói chung và tại trường Cao đẳng Kỹ thuật Cao Thắng nói riêng.

Sau nhiều năm tham gia giảng dạy môn phương pháp lập trình hướng đối tượng và thực tập phương pháp lập trình hướng đối tượng cho sinh viên cao đẳng công nghệ thông tin. Nhóm tác giả quyết định biên soạn cuốn bài giảng này nhằm phục vụ công tác giảng dạy, cũng như học tập của sinh viên chuyên ngành công nghệ thông tin.

Nội dung bài giảng tập trung vào giới thiệu tổng quan về phương pháp lập trình hướng đối tượng, và các tính chất cơ bản của nó là: tính trừu tượng hóa (abstraction), tính đóng gói (encapsulation) và che giấu thông tin (information hiding), tính kế thừa

Trong tài liệu này, chúng tôi có tham khảo và sử dụng các tài liệu, bài giảng và hình ảnh của các tác giả trong và ngoài nước, cũng như tham khảo trên các website.

Đây là lần đầu tiên xuất bản, nên chắc chắn không thể tránh khỏi những sai sót. Nhóm tác giả rất mong nhận được những ý kiến đóng góp của quý thầy cô, các đồng nghiệp và sinh viên để có thể hoàn thiện hơn bài giảng này phục vụ tốt hơn cho việc học tập của sinh viên.

Trân trọng cảm ơn./.

Tp. Hồ Chí Minh, tháng 11/2019

Nhóm tác giả

MỤC LỤC

CHƯƠNG 1

TỔNG QUAN VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Các nội dung chính:

- *Kỹ thuật lập trình hướng đối tượng*
- *Một số khái niệm cơ bản về hướng đối tượng*
- *Các tính chất của lập trình hướng đối tượng*
- *Một số ngôn ngữ lập trình hướng đối tượng*

1.1. Kỹ thuật lập trình hướng đối tượng

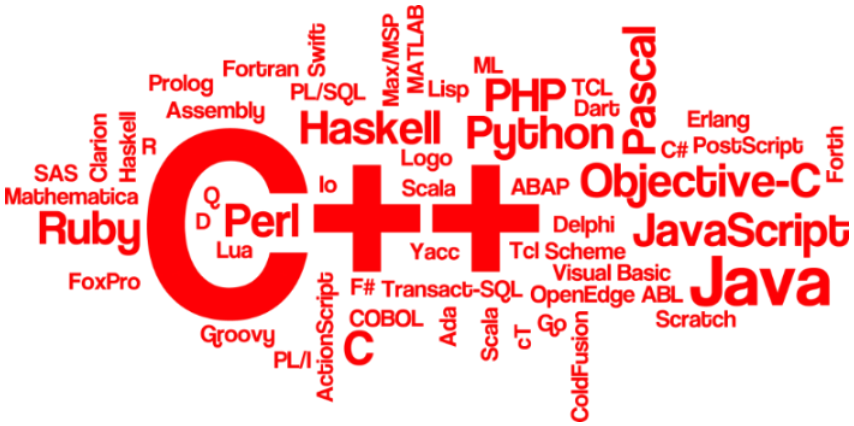
Lập trình hướng đối tượng là một kỹ thuật lập trình. Vậy kỹ thuật lập trình là gì?

1.1.1. Kỹ thuật lập trình là gì?

Kỹ thuật lập trình là kỹ thuật thực thi một giải pháp phần mềm (cấu trúc dữ liệu + giải thuật) dựa trên nền tảng một phương pháp luận (methodology) và một hoặc nhiều ngôn ngữ lập trình phù hợp với yêu cầu đặc thù của ứng dụng.

1.1.2. Ngôn ngữ lập trình

Ngôn ngữ lập trình (programming language) là dạng ngôn ngữ được chuẩn hóa theo một hệ thống các quy tắc riêng, sao cho người lập trình có thể mô tả các chương trình làm việc dành cho thiết bị điện tử mà cả con người và các thiết bị đó đều hiểu được. Sử dụng chương trình dịch tương ứng để giao tiếp với máy tính.

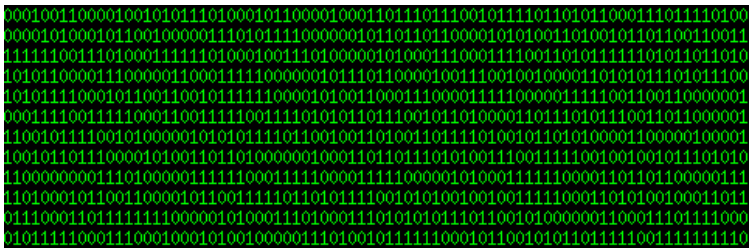


Hình 1.1. Các ngôn ngữ lập trình (Internet)

1.1.3. Phân loại ngôn ngữ lập trình

✓ Ngôn ngữ máy (machine lanuage)

Máy tính chỉ nhận các tín hiệu điện tử (có/không), tương ứng với các dòng bits. Một chương trình ở dạng đó gọi là machine code. Khởi đầu, chúng ta phải dùng machine code để viết chương trình -> nó quá phức tạp, để giải quyết các bài toán lớn là không tưởng.



Hình 1.2. Ngôn ngữ máy

✓ Hợp ngữ (assembly language)

Hợp ngữ (Assembly Language - ASM) là ngôn ngữ bậc thấp, chính xác nó là ngôn ngữ thuộc thế hệ thứ 2 (2nd generation). ASM sử dụng các từ gợi nhớ (mnemonics) để viết các chỉ thị (instructions) lập trình cho máy tính thay vì bằng những dãy 0 và 1.

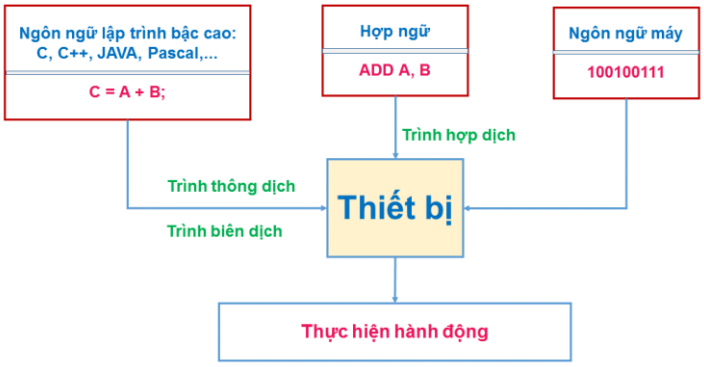
Các ASM sẽ cần một chương trình Assembler phù hợp (NASM, AS, DASM) để dịch chúng thành những file binary và một trình linker để link các thành phần lại và chỉ định nơi bắt đầu của chương trình và đây là việc bắt buộc.

ASM có thể tương tác rất sâu dưới hệ thống, chúng có thể giao tiếp trực tiếp với các phần cứng và bắt chúng hoạt động theo ý người lập trình. Vì thế mà chúng ta có hẳn một ngành mang tên là "lập trình nhúng".

Hình 1.3. Chương trình viết bằng hợp ngữ

✓ Ngôn ngữ lập trình bậc cao (high-level language)

Ngôn ngữ lập trình bậc cao là ngôn ngữ lập trình có hình thức gần với ngôn ngữ tự nhiên, có tính độc lập cao, ít phụ thuộc vào loại thiết bị (loại vi xử lý) cũng như các trình dịch. Một số ngôn ngữ lập trình bậc cao phổ biến hiện nay như: Pascal, C, C++, Java, PHP, C#, Python, Object C,...



Hình 1.4. Sơ đồ thực hiện chương trình theo ngôn ngữ lập trình

Chúng ta cũng có thể phân loại ngôn ngữ lập trình theo phương pháp lập trình:

✓ Lập trình tuần tự (tuyến tính)

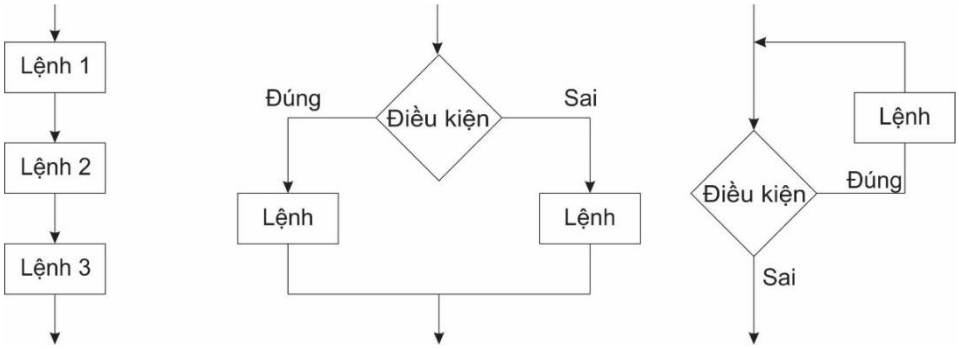
Lập trình tuyến tính là một phương pháp lập trình, kỹ thuật lập trình truyền thống. Đặc trưng của Lập trình tuyến tính là tư duy theo lối tuần tự, đơn giản và đơn luồng. Chương trình sẽ được thực hiện theo thứ tự từ đầu đến cuối, lệnh này kế tiếp lệnh kia cho đến khi kết thúc chương trình.

```
          RELOAD EQU 0E6H    ; defining reload constant for baudrate generation
          ORG 0000H          ; org directive
          SJMP START         ; jump to main program

SENDCH:   ORG 0023H          ; ISR for RI/TI interrupt
          CLR TI             ; clear transmit flag
          MOV SBUF, #'A'     ; send the ASCII value of 'A'
          RETI               ; return back to main program

START:    ANL PCON, #7FH     ; Set SMOD=0
          ANL TMOD, #0FH     ; Alter only the setting of Timer-1
          ORL TMOD, #20H     ; Timer-1 in mode-2
          MOV TH1, #RELOAD   ; Move the reload value to TH1
          SETB TR1           ; start Timer-1 for baud rate generation
          MOV SCON, #40H     ; set serial port in mode-1
          ORL IE, #90H       ; Enable serial port interrupt

          MOV SBUF, #'A'     ; Transmit a character

WAIT:     SJMP WAIT          ; wait till interrupt occurs

          END
```

Hình 1.5. Chương trình theo hướng lập trình tuyến tính

✓ Lập trình hướng thủ tục (lập trình cấu trúc)

Lập trình được cấu trúc lần đầu tiên được đề xuất bởi Corrado Bohm và Guiseppe Jacopini. Hai nhà toán học này đã chứng minh rằng bất kỳ chương trình máy tính nào cũng có thể được viết chỉ với ba cấu trúc: tuần tự, điều kiện và vòng lặp.



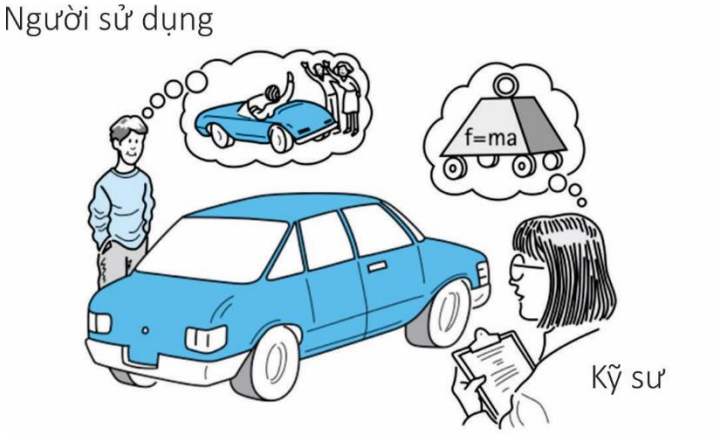
Hình 1.6. Cấu trúc trong lập trình hướng thủ tục

✓ Sự trừu tượng hóa dữ liệu (data abstraction)

Là phương pháp biểu diễn dữ liệu giúp người sử dụng có thể thao tác trên dữ liệu một cách dễ dàng mà không cần quan tâm đến các chi tiết của dữ liệu.

Ví dụ: Kiểu dữ liệu số thực dấu chấm phẩy động trong các ngôn ngữ lập trình đã được trừu tượng hóa, khi lập trình người lập trình không cần quan tâm đến cách biểu diễn nhị phân chính xác của số thực dấu chấm phẩy động và các chi tiết khác.

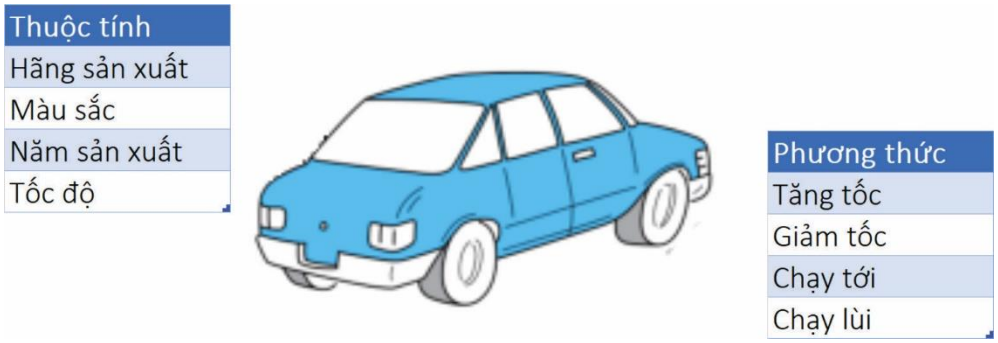
Sự phát triển của các ngôn ngữ lập trình chính là sự phát triển của quá trình trừu tượng hóa (abstraction). Lập trình hướng đối tượng là bước tiến hóa tiếp theo của lập trình cấu trúc: trừu tượng hóa dữ liệu (data abstraction), coi các dữ liệu trong chương trình là các đối tượng.



Hình 1.7. Sự trừu tượng hóa (Internet)

✓ Lập trình hướng đối tượng

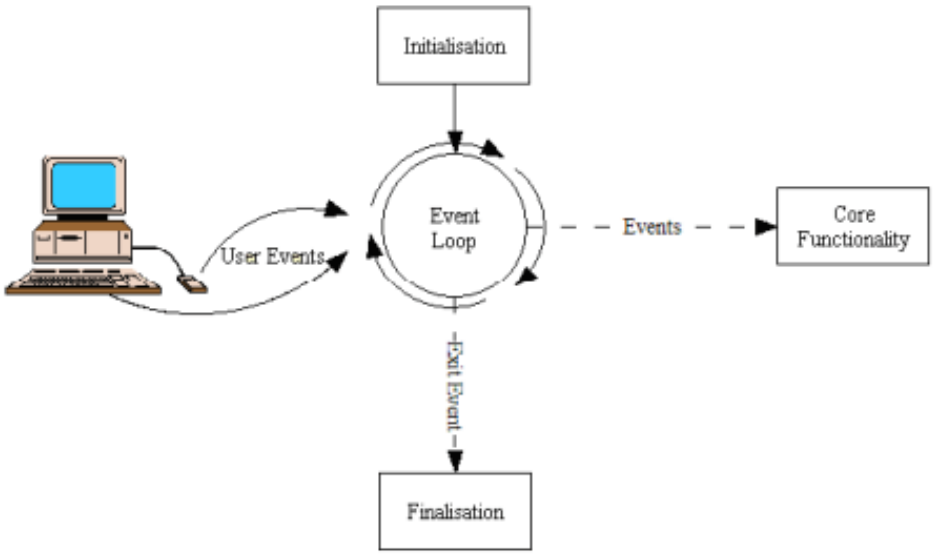
Lập trình hướng đối tượng (Object-oriented programming – OOP) phương pháp lập trình lấy khái niệm đối tượng (objects) làm nền tảng. Trong đó, đối tượng chứa đựng các dữ liệu (data), trên các trường (fields), thường được gọi là các thuộc tính (attributes), và mã nguồn (code) được tổ chức thành các phương thức (methods).



Hình 1.8. Phương pháp lập trình hướng đối tượng (Internet)

✓ Lập trình hướng sự kiện (event-driven programming)

Hướng theo các sự kiện đang xảy ra và giữ tương tác với người dùng. Lập trình hướng sự kiện cung cấp nền tảng cho lập trình trực quan. Chương trình hướng sự kiện làm việc theo dạng thụ động (passively) chờ cho một sự kiện xảy ra và kích hoạt (activate).

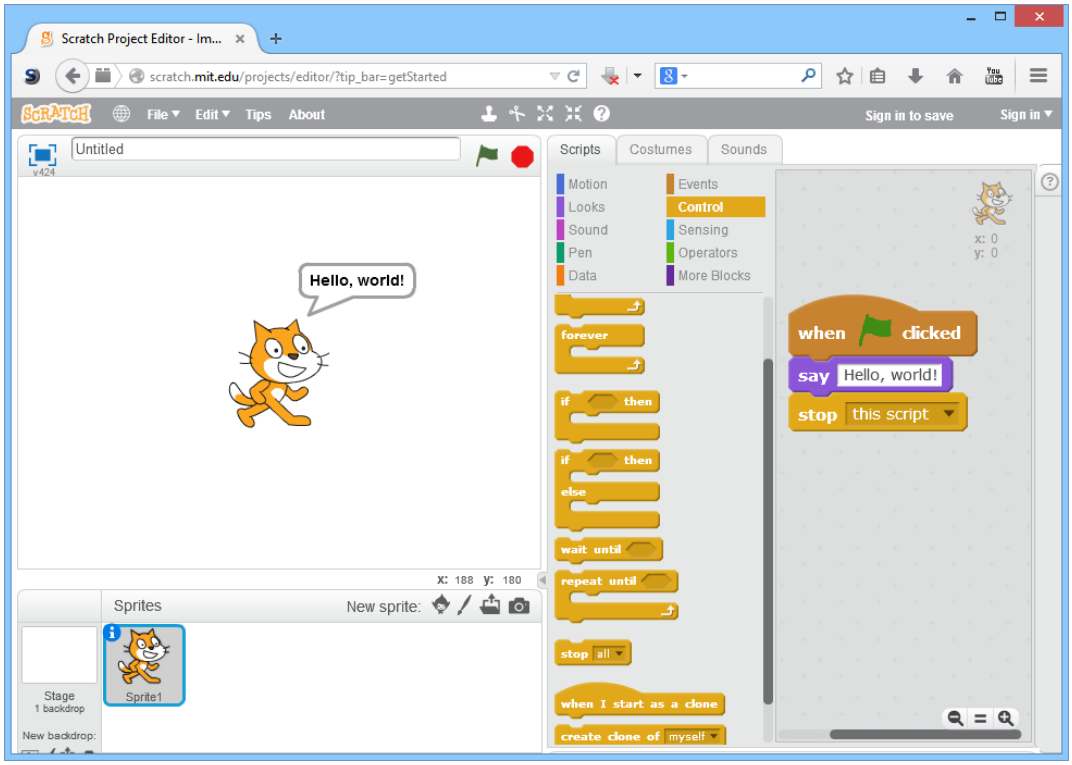


Hình 1.9. Phương pháp lập trình hướng sự kiện (Internet)

✓ Lập trình trực quan (visual programming)

Trực tiếp tạo ra các khung giao diện (interface), ứng dụng thông qua các thao tác trên màn hình dựa vào các đối tượng (object) như hộp hội thoại hoặc nút điều khiển (control button).

Ít khi phải tự viết các lệnh mà chỉ cần khai báo việc gì cần làm khi một tình huống xuất hiện. Có thể được thực hiện bằng cách sử dụng ngôn ngữ lập trình trực quan (visual programming language) hoặc một môi trường lập trình trực quan (visual programming environment)



Hình 1.10. Phương pháp lập trình trực quan

1.1.4. Trình dịch

Trình biên dịch là chương tình máy tính, có nhiệm vụ dịch chương trình nguồn (mã nguồn) thành một chương trình tương đương nhưng ở dưới dạng một ngôn ngữ mới, gọi là ngôn ngữ đích và thường là ngôn ngữ ở cấp thấp.

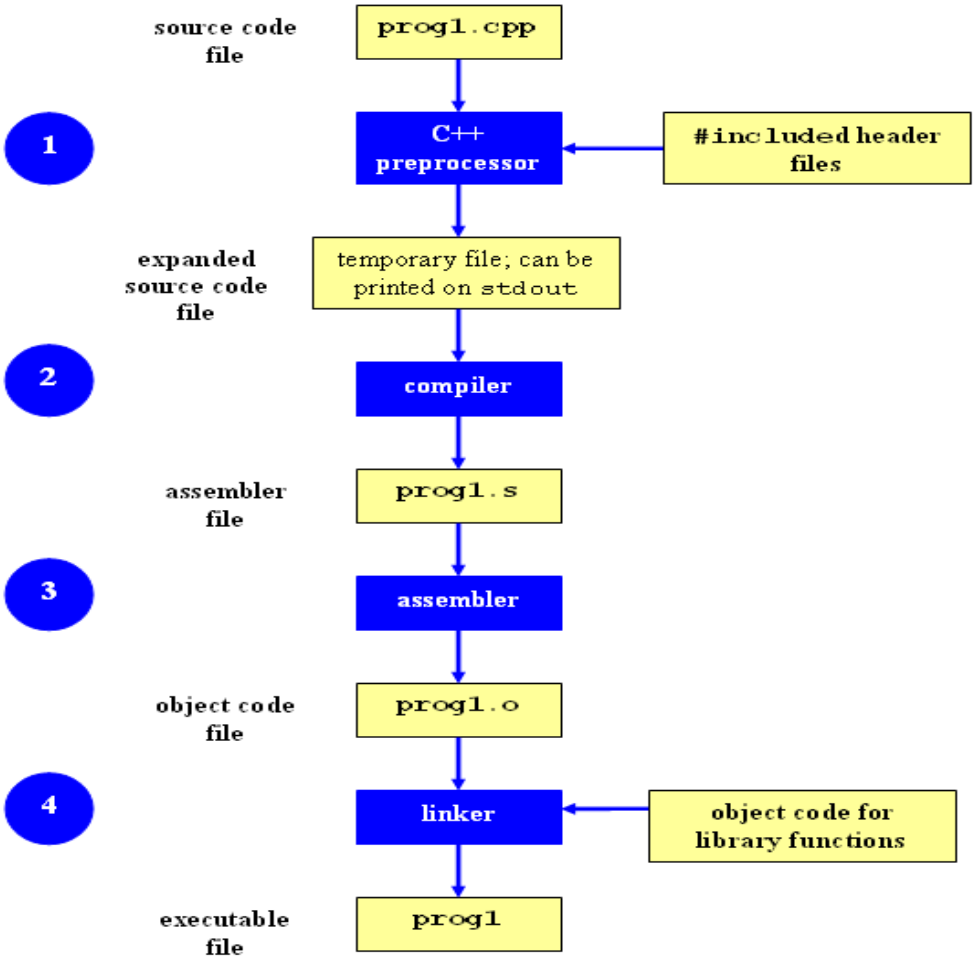
Trong quá trình biên dịch, trình biên dịch sẽ thông báo các lỗi (nếu có) để người viết có thể dựa vào đó mà kiểm tra, điều chỉnh những sai sót của chương trình nguồn (mã nguồn).

Máy tính, các thiết bị, linh kiện điện tử chỉ có thể hiểu được những thông tin dưới dạng nhị phân (0 và 1), vì vậy cần một chương trình trung gian để dịch các câu lệnh được viết bởi ngôn ngữ lập trình thành nhị phân để máy tính thể hiểu được. Có 2 loại trình dịch là biên dịch và thông dịch.

✓ Trình biên dịch (compiler)

Trình biên dịch là chương tình máy tính, có nhiệm vụ dịch chương trình nguồn (mã nguồn) thành một chương trình tương đương nhưng ở dưới dạng một ngôn ngữ mới, gọi là ngôn ngữ đích và thường là ngôn ngữ ở cấp thấp hơn (Wikipedia).

Trong quá trình biên dịch, trình biên dịch sẽ thông báo các lỗi (nếu có) để người viết có thể dựa vào đó mà kiểm tra, điều chỉnh những sai sót của chương trình nguồn (mã nguồn).

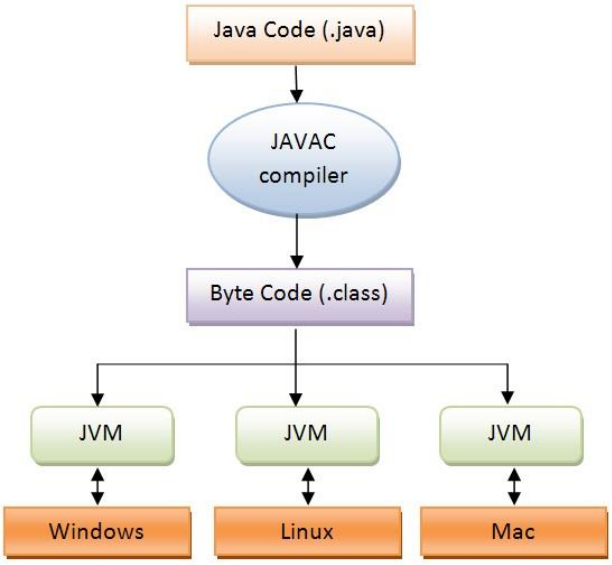


Hình 1.11. Quá trình biên dịch từ source code đến file chạy (Internet)

✓ Trình thông dịch (interpreter)

Trình thông dịch là chương tình máy tính, có nhiệm vụ dịch chương trình nguồn (mã nguồn) theo từng phân đoạn cần thiết. Tức thông dịch chỉ dịch những gì cần để thực thi mà không dịch toàn bộ file mã nguồn.

Thiết bị tích hợp trình thông dịch có thể thực thi các lệnh sau thông dịch ngay, tức là dịch tới đâu thì thực thi tới đó.



Hình 1.12. Quá trình thông dịch chương trình Java

1.1.5. Sự phát triển của ngôn ngữ lập trình[1]

Ngôn ngữ lập trình cho phép lập trình viên viết các chương trình tính toán theo những thuật toán để giải những vấn đề nào đó.

1983: ADA LOVELACE đã tạo ra ngôn ngữ lập trình máy tính đầu tiên dựa trên chiếc máy tính cơ khí đầu tiên (Analytical Engine). Hiện có trên 1,2 triệu lập trình viên máy tính và nhà phát triển phần mềm sử dụng.

1957-1959: FORTRAN: một trong những ngôn ngữ cổ xưa nhất được sử dụng. Được dùng trong những ứng siêu máy tính, phát triển AI, phần mềm kinh doanh (NASA, thẻ tín dụng, máy ATM).

[1] Nguồn: https://www.veracode.com/blog/2013/04/the-history-of-programming-languages-infographic

1970: Ngôn ngữ PASCAL (đặt theo tên nhà toán học, vật lý người Pháp – BALISE PASCAL), được tạo ra bởi NIKLAUS WIRTH. Được dùng trong dạy lập trình, sử dụng rộng rãi để phát triển ứng dụng của Windows, máy tính Apple Lisa (1983), Skype.

1972: Ngôn ngữ C (dựa vào ngôn ngữ B), được tạo ra bởi DENNIS RITCHIE. Được dùng để lập trình đa nền tảng, lập trình hệ thống, lập trinh Unix (viết lại bằng C năm 1973), phát triển game máy tính, server-client ban đầu của WWW.

1983: Ngôn ngữ C++ (trước đó gọi là C with class), là ngôn ngữ nâng cao của C, với sự cải tiến về class, hàm ảo và template, được tạo ra bởi BJARNE STROUSTRUP. Được dùng để phát triển ứng dụng thương mại, phần mềm nhúng, ứng dụng client-server, trò chơi video game (Adobe, Mozilla Firefox, Microsoft Internet Explorer).

1983: Ngôn ngữ Objective-C (lập trình hướng đối tượng mở rộng từ C), là ngôn ngữ mở rộng của C, có thêm chức năng message-passing, truyền dữ liệu từ process này sang process khác, dựa trên ngôn ngữ SMALLTALK, được tạo ra bởi BRAD COX và TOM LOVE. Đươc sử dụng trong Apple (hệ điều hành iOS và OS X).

1987: Ngôn ngữ PERL, được tạo ra bởi LARRY WALL. Được dùng để xủa lý các báo cáo trong các hệ thống UNIX, những ứng dụng chính như CGI (Common Gateway Interface), ứng dụng quản lý database, quản lý hệ thống, lập trình mạng, lập trình đồ họa (IMDb, Amazon, Priceline, Ticketmaster).

1991: Ngôn ngữ PYTHON (theo tên một đoàn hài kịch Anh – MONTY PYTHON), được tạo ra bởi GUIDO VAN ROSSUM. Được dùng để phát triên phần mềm, ứng dụng web, bảo mật thông tin (Yahoo, Google, Spotify).

1993: Ngôn ngữ RUBY, là ngôn ngữ được ảnh hưởng bởi các ngôn ngữ PERL, ADA, LISP, SMALLTALK,… được tạo ra bởi YUKIHIRO MATSUMOTO. Được dùng để phát triển ứng dụng web, framework Ruby on Rails (Twitter, Hulu, Groupon).

1995: Ngôn ngữ Java, được tạo ra bởi JAMES GOSLING. Được dùng để phát triển ứng dụng web, phát triển phần mềm, lập trình mạng, phát triển giao diện đồ họa người dùng (hệ điều hành, ứng dụng Android).

1995: Ngôn ngữ PHP (trước đây có nghĩa là Personal Homepage, ngày nay là Hypertext Preprocessor), là ngôn ngữ mã nguồn mở được sử dụng rộng rãi, được tạo ra bởi RASMUS LERDORF. Được dùng để xây dựng, bảo trì các trang web động, phát triển server (Facebook, Wikipedia, Digg, WordPress, Joomla, Codeigniter, Symfony, Yii 2, Phalcon, CakePHP, Zend, Laravel,…)

1995: Ngôn ngữ JAVASCRIP (MOCHA, LIVESSCRIPT), được tạo ra bởi BRENDAN EICH. Được dùng để mở rộng các chức năng của web, phát triển web động, ảnh động, theo dõi hoạt động người dùng, xử lý tài liệu PDF, trình duyệt web, công cụ màn hình (Gmail, Adobe, Photoshop, Mozilla Firefox).

1.2. Một số khái niệm cơ bản

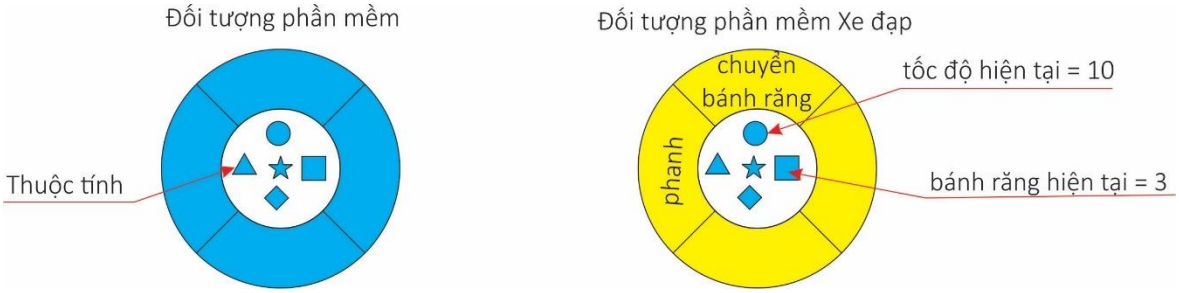
1.2.1. Đối tượng (Object)

Trong thế giới thực, đối tượng (object) là một thực thể (entity) cụ thể mà ta có thể nhìn thấy, sờ mó được. Mỗi đối tượng đều có: các thông tin, trạng thái và các hoạt dộng hành vi riêng của nó. Ví dụ:

Đối tượng	Trạng thái	Hành vi	Hình
Con chó	Tên	Sủa	
	Màu sắc	Vẫy đuôi	
	Giống chó	Ăn	
	Tuổi	Chạy	
Ô tô	Hãng sản xuất	Tăng tốc	
	Màu sắc	Giảm tốc	
	Năm sản xuất	Chạy tới	
	Tốc độ	Chạy lùi	
Tài khoản	Tên tài khoản	Đăng nhập	
	Số tài khoản	Rút tiền	
	Ngân hàng	Gửi tiền	
	Số dư	Kiểm tra số dư	

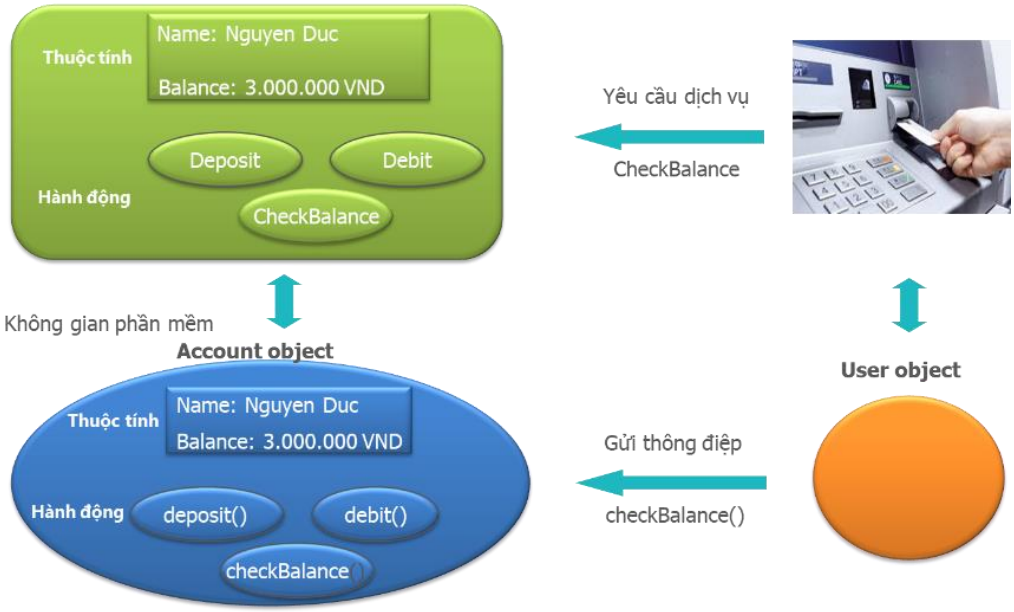
1.2.2. Đối tượng phần mềm

Trong lập trình hướng đối tượng, đối tượng là một thực thể phần mềm được sử dụng bởi máy tính, dùng để mô tả một đối tượng ở thế giới thực (một người, một sự vật, một con vật, một khái niệm), được bao bọc các thuộc tính và các phương thức liên quan.



Hình 1.13. Đối tượng phần mềm Xe đạp

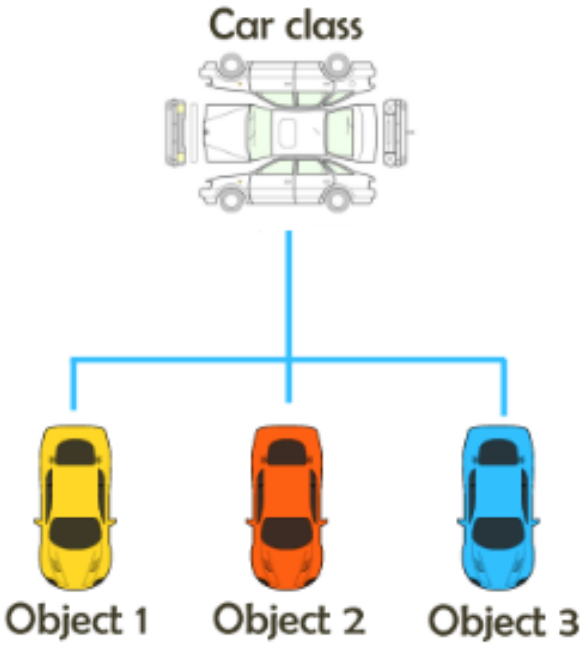
Ví dụ: Quản lý tài khoản ngân hàng



Hình 1.14. Đối tượng Tài khoản

1.2.3. Lớp (class)

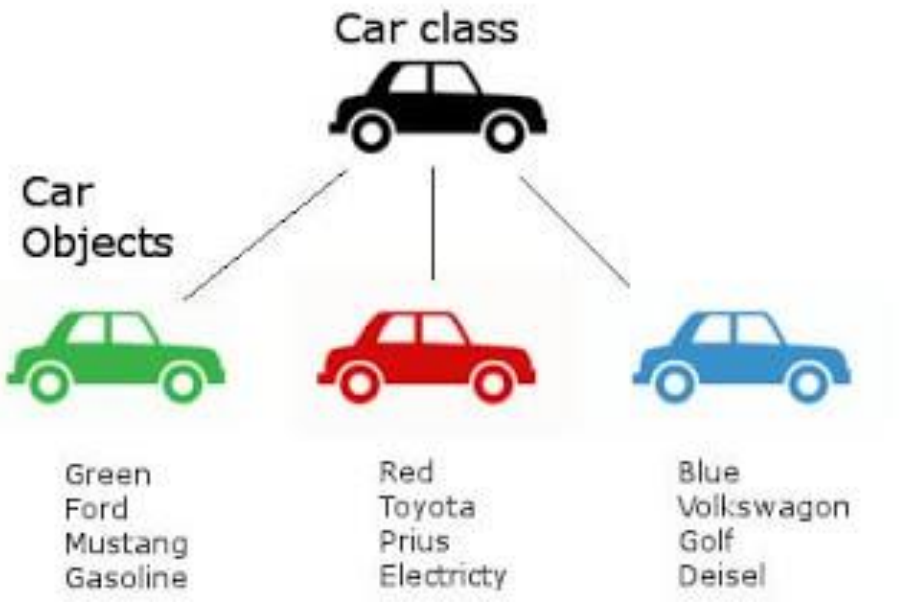
Một lớp là một thiết kế (blueprint) hay mẫu (prototype) cho các đối tượng cùng kiểu, định nghĩa các thuộc tính và các phương thức chung cho tất cả các đối tượng của cùng một loại nào đó. Một đối tượng là một thể hiện cụ thể của một lớp. Mỗi thể hiện có thể có những thuộc tính thể hiện khác nhau.



Hình 1.15. Lớp Cars

1.2.4. Lớp và đối tượng

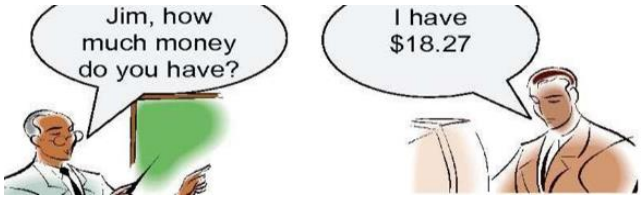
Trong OOP, lớp là một khái niệm tĩnh, có thể nhận biết từ code của chương trình (định nghĩa kiểu dữ liệu), còn đối tượng là một khái niệm động, nó chiếm một vùng nhớ trong bộ nhớ. Khi khai báo và khởi tạo, đối tượng được tạo ra để lưu trữ dữ liệu, xử lý theo yêu cầu được thiết kế và bị hủy bỏ khi hết nhiệm vụ.



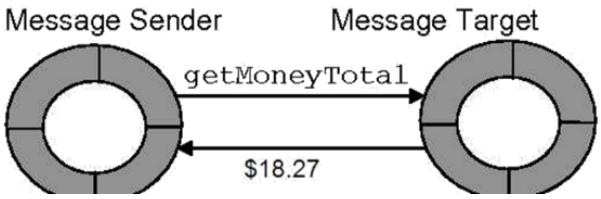
Hình 1.16. Lớp và đối tượng (internet)

1.2.5. Tương tác giữa các đối tượng

Trong thế giới thực: các đối tượng có thể tương tác được với nhau.



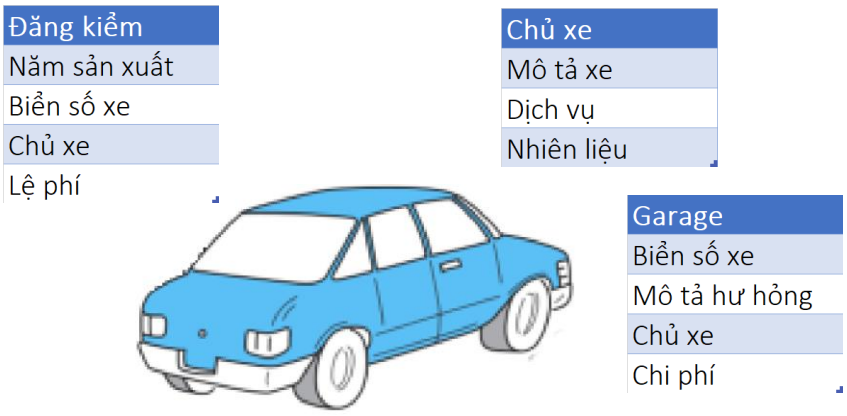
Trong lập trình: các đối tượng giao tiếp với nhau bằng cách gửi thông điệp.



1.3. Các tính chất của OOP

1.3.1. Tính trừu tượng

Là quá trình loại bỏ đi các thông tin cụ thể và giữ lại những thông tin chung. Tập trung vào các đặc điểm cơ bản của thực thể, các đặc điểm phân biệt nó với các loại thực thể khác. Trừu tượng hóa phụ thuộc vào góc nhìn, thuộc tính quan trọng trong ngữ cảnh này nhưng lại không có ý nghĩa nhiều trong ngữ cảnh khác.



Hình 1.17. Đối tượng xe hơi ở các góc nhìn khác nhau

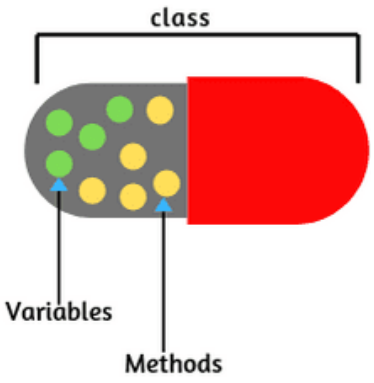
1.3.2. Tính đóng gói (encapsulation)

Tính đóng gói là cơ chế ràng buộc dữ liệu và thao tác trên dữ liệu đó thành một thể thống nhất, tránh được các tác động bất ngờ từ bên ngoài. Thể thống nhất này gọi là đối tượng.

Theo Ivar Jacobson[2], tất cả các thông tin của một hệ thống định hướng đối tượng được lưu trữ bên trong đối tượng của nó và chỉ có thể hành động khi các đối tượng đó được ra lệnh thực hiện các thao tác. Như vậy, sự đóng gói không chỉ đơn thuần là sự gom chung dữ liệu và chương trình vào trong một khối, chúng còn được hiểu theo nghĩa là sự đồng nhất giữa dữ liệu và các thao tác tác động lên dữ liệu đó.

Trong một đối tượng, dữ liệu hay thao tác hay cả hai có thể là riêng (private) hoặc chung (public) của đối tượng đó. Thao tác hay dữ liệu riêng là thuộc về đối tượng đó chỉ được truy cập bởi các thành phần của đối tượng, điều này nghĩa là thao tác hay dữ liệu riêng không thể truy cập bởi các phần khác của chương trình tồn tại ngoài đối tượng. Khi thao tác hay dữ liệu là chung, các phần khác của chương trình có thể truy cập nó mặc dù nó được định nghĩa trong một đối tượng. Các thành phần chung của một đối tượng dùng để cung cấp một giao diện có điều khiển cho các thành phần riêng của đối tượng.

Cơ chế đóng gói là phương thức tốt để thực hiện cơ chế che dấu thông tin so với các ngôn ngữ lập trình cấu trúc.



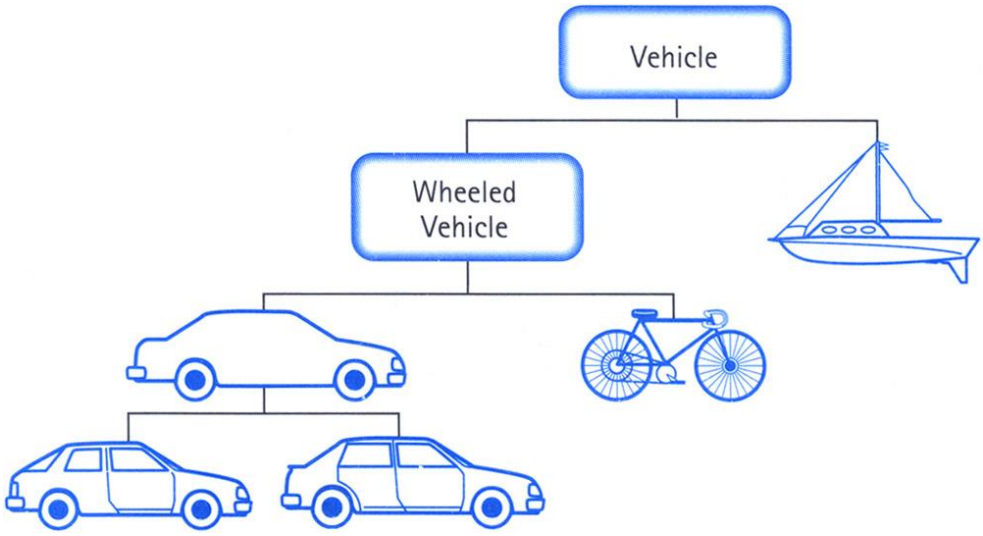
Hình 1.18. Class đóng gói các thành phần dữ liệu và các phương thức (internet)

1.3.3. Tính kế thừa (inhenritace)

Chúng ta có thể xây dựng các lớp mới từ các lớp cũ thông qua sự kế thừa. Một lớp mới còn gọi là lớp dẫn xuất (derived class), có thể thừa hưởng dữ liệu và các phương thức của lớp cơ sở (base class) ban đầu. Trong lớp này, có thể bổ sung các

[2] Ivar Jacobson: tác giả quyển sách "Object-oriented Software Engineering: A Use Case Driven Approach", xuất bản năm 1992.

thành phần dữ liệu và các phương thức mới vào những thành phần dữ liệu và các phương thức mà nó thừa hưởng từ lớp cơ sở. Mỗi lớp (kể cả lớp dẫn xuất) có thể có một số lượng bất kỳ các lớp dẫn xuất. Qua cơ cấu kế thừa này, dạng hình cây của các lớp được hình thành. Dạng cây của các lớp trông giống như các cây gia phả vì thế các lớp cơ sở còn được gọi là lớp cha (parent class) và các lớp dẫn xuất được gọi là lớp con (child class).



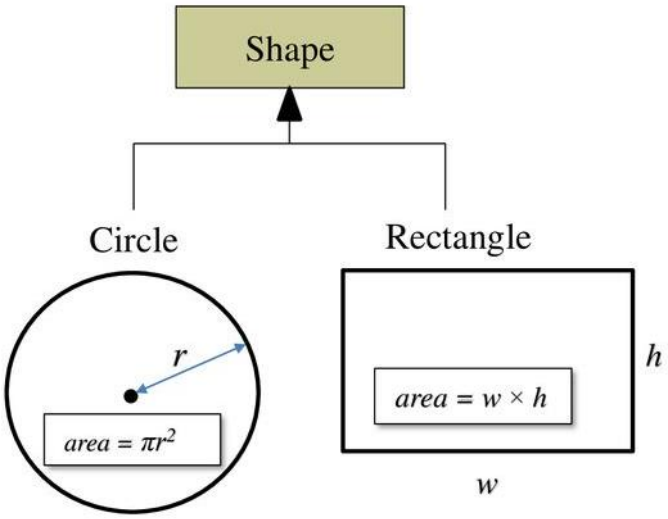
Hình 1.19. Sơ đồ kế thừa (internet)

Với tính kế thừa, chúng ta không phải mất công xây dựng lại từ đầu các lớp mới, chỉ cần bổ sung để có được trong các lớp dẫn xuất các đặc trưng cần thiết.

1.3.4. Tính đa hình (polymorphism)

Đó là khả năng để cho một thông điệp có thể thay đổi cách thực hiện của nó theo lớp cụ thể của đối tượng nhận thông điệp. Khi một lớp dẫn xuất được tạo ra, nó có thể thay đổi cách thực hiện các phương thức nào đó mà nó thừa hưởng từ lớp cơ sở của nó. Một thông điệp khi được gởi đến một đối tượng của lớp cơ sở, sẽ dùng phương thức đã định nghĩa cho nó trong lớp cơ sở. Nếu một lớp dẫn xuất định nghĩa lại một phương thức thừa hưởng từ lớp cơ sở của nó thì một thông điệp có cùng tên với phương thức này, khi được gởi tới một đối tượng của lớp dẫn xuất sẽ gọi phương thức đã định nghĩa cho lớp dẫn xuất.

Như vậy đa hình là khả năng cho phép gởi cùng một thông điệp đến những đối tượng khác nhau có cùng chung một đặc điểm, nói cách khác thông điệp được gởi đi không cần biết thực thể nhận thuộc lớp nào, chỉ biết rằng tập hợp các thực thể nhận có chung một tính chất nào đó. Chẳng hạn, thông điệp "Tính diện tích – area" được gởi đến cả hai đối tượng hình vuông và hình tròn. Trong hai đối tượng này đều có chung phương thức double area(), tuy nhiên tuỳ theo thời điểm mà đối tượng nhận thông điệp, diện tích hình tương ứng sẽ được tính.



Hình 1.20. Tính đa hình (internet)

Trong các ngôn ngữ lập trình OOP, tính đa hình thể hiện qua khả năng cho phép mô tả những phương thức có tên giống nhau trong các lớp khác nhau. Đặc điểm này giúp người lập trình không phải viết những cấu trúc điều khiển rườm rà trong chương trình, các khả năng khác nhau của thông điệp chỉ thực sự đòi hỏi khi chương trình thực hiện.

1.4. Ngôn ngữ lập trình hướng đối tượng

Xuất phát từ tư tưởng của ngôn ngữ SIMULA67, trung tâm nghiên cứu Palo Alto (PARC) của hãng XEROR đã tập trung 10 năm nghiên cứu để hoàn thiện ngôn ngữ OOP đầu tiên với tên gọi là Smalltalk. Sau đó các ngôn ngữ OOP lần lượt ra đời như Eiffel, Clos, Loops, Flavors, Object Pascal, Object C, C++, Delphi, Java,…

Chính XEROR trên cơ sở ngôn ngữ OOP đã đề ra tư tưởng giao diện biểu tượng trên màn hình (icon base screen interface), kể từ đó Apple Macintosh cũng như Microsoft Windows phát triển giao diện đồ họa như ngày nay. Trong Microsoft Windows, tư tưởng OOP được thể hiện một cách rõ nét nhất đó là "chúng ta click vào đối tượng", mỗi đối tượng có thể là control menu, control menu box, menu bar, scroll bar, button, minimize box, maximize box, … sẽ đáp ứng công việc tùy theo đặc tính của đối tượng. Turbo Vision của hãng Borland là một ứng dụng OOP tuyệt vời, giúp lập trình viên không quan tâm đến chi tiết của chương trình giao diện mà chỉ cần thực hiện các nội dung chính của vấn đề.

BÀI TẬP CHƯƠNG 1

Bài 01: Viết chương trình sau:

1. Định nghĩa cấu trúc phân số gồm tử số và mẫu số;
2. Hàm nhập, xuất một phân số.
3. Hàm tính tổng, hiệu, tích, thương (lưu ý: rút gọn phân số).
4. Hàm main, cho phép nhập 2 phân số, xuất các phân số ra màn hình, thực hiện tích tổng, hiệu, tích, thuong hai phân số và xuất kết quả.

Bài 02: Viết chương trình thực hiện các yêu cầu sau:

1. Khai báo kiểu cấu trúc Diem.
2. Khai báo kiểu cấu trúc TamGiac.
3. Hàm nhập / xuất điểm (đỉnh).
4. Hàm tính khoảng cách giữa 2 điểm.
5. Hàm kiểm tra 3 điểm có tạo thành một tam giác.
6. Nếu 3 đỉnh tạo thành tam giác, hãy cho biết là tam giác gì, tính chu vi, diện tích tam giác đó.
7. Hàm main(), thực hiện các hàm trên.

CHƯƠNG 2

GIỚI THIỆU NGÔN NGỮ LẬP TRÌNH C++

Các nội dung chính

- *Giới thiệu ngôn ngữ C++*
- *Một số mở rộng của ngôn ngữ C++ so với ngôn ngữ C*

2.1. Giới thiệu

2.1.1. Giới thiệu ngôn ngữ C++

C++ là một ngôn ngữ lập trình đa dụng, có thể dùng C++ để lập trình cho các hệ thống lớn, lập trình hệ điều hành cho đến các ứng dụng, game hay thậm chí ta có thể dùng C++ để lập trình web. Với C++, ta có thể thấy được sự mềm dẻo của nó qua việc C++ hỗ trợ cho ta các tính năng cao cấp như lập trình hướng đối tượng, lập trình generic trong khi vẫn cung cấp cho ta khả năng can thiệp sâu vào bên trong bộ nhớ máy tính thông qua con trỏ.

C++ là ngôn ngữ biên dịch, tùy thuộc vào các hệ thống khác nhau mà ta có thể có các trình biên dịch tương ứng như với Windows ta sẽ có cl (nằm trong bộ IDE Visual Studio) trong khi đó với Linux ta có gcc / g++.

2.1.2. Lịch sử hình thành C++

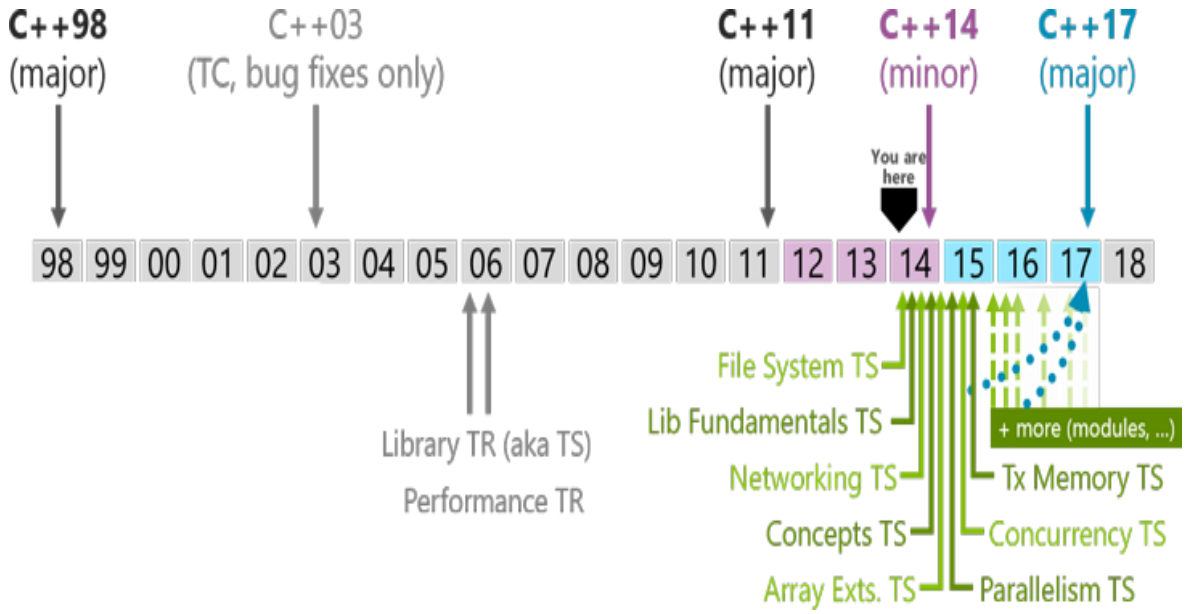
Trước C++, ngôn ngữ lập trình C được phát triển trong năm 1972 bởi Dennis Ritchie tại phòng thí nghiệm Bell Telephone, C chủ yếu là một ngôn ngữ lập trình hệ thống, một ngôn ngữ để viết ra hệ điều hành. Vào năm 1999, ủy ban ANSI đã phát hành một phiên bản mới của ngôn ngữ lập trình C, gọi là C99.

C++ được tạo ra bởi Bjarne Stroustrup - một nhà khoa máy tính người Đan Mạch tại phòng thí nghiệm AT&T Bell vào năm 1979, và gọi nó là C with classe. Sau đó, vào năm 1983, C with classes được đổi tên thành C++ (++ là toán tử tăng dần trong C). Các tính năng mới được thêm vào bao gồm: virtual functions (hàm ảo), function overloading (nạp chồng hàm) và operator overloading (nạp chồng toán tử),

references (kiểu tham chiếu), constants (hằng số), cấp phát và hủy bộ nhớ bằng toán tử new/delete, thêm tính năng chú thích trên một dòng //,...

2.1.3. Quá trình chuẩn hóa

C++ được chuẩn hóa bởi tổ chức ISO - JTC1/SC22/WG21.



Hình 2.1. Quá trình phát triển của C++ (internet)

2.1.4. Lý do chọn ngôn ngữ lập trình C++

Ngôn ngữ lập trình C++ có thể được dùng để làm những công việc sau:

– C++ được thiết kế để viết những hệ thống lớn, thậm chí C++ được dùng để tạo nên hệ điều hành máy tính (Linux, Mac OS X, Windows…).

– C++ được dùng để tạo nên các game lớn của hãng Blizzard (World of Warcraft, Diablo series, StarCraft series…). Gần như toàn bộ các game bom tấn trên thị trường hiện nay cũng dùng C++ để phát triển. Một số công cụ sử dụng trong việc lập trình game có sử dụng C++ như Unreal engine, Cocos2d-x framework,… Các ông lớn trong ngành công nghiệp game như Valve, CryTek cũng sử dụng C++.

– Các sản phẩm phần mềm nổi tiếng khác được phát triển bằng C++ như MS Office, Photoshop, Maya / 3ds, Auto CAD,…

– C++ có thể được sử dụng ở phía Web server vì C++ có thể đáp ứng được yêu cầu về tốc độ xử lý, khả năng phản hồi nhanh.

2.2. Một số mở rộng của ngôn ngữ C++ so với ngôn ngữ C

2.2.1. Các từ khóa (keyword) mới của C++

Để bổ sung các tính năng mới vào C, một số từ khóa mới đã được đưa vào C++ ngoài các từ khóa có trong C. Các chương trình bằng C nào sử dụng các tên trùng với các từ khóa cần phải thay đổi trước khi chương trình được dịch lại bằng C++.

Danh sách 32 từ khoá trong C và C++:

auto	Break	case	char	const	continue	default	do
double	Else	enum	extern	float	for	goto	if
int	Long	register	return	short	signed	sizeof	static
struct	Switch	typedef	union	unsigned	void	volatile	while

Danh sách 30 từ khoá trong C ++ nhưng không có trong ngôn ngữ C:

asm	dynamic_cast	namespace	reinterpret_cast	bool
explicit	New	static_cast	false	catch
operator	template	friend	private	class
this	Inline	public	throw	const_cast
delete	mutable	protected	true	try
typeid	typename	using	virtual	wchar_t

Danh sách một số từ khóa được thêm vào C++11:

lignas	alignof	char16_t	char32_t	constexpr
decltype	noexcept	nullptr	static_assert	thread_local

Danh sách một số từ khóa được thêm vào C++20:

char8_t	concept	consteval	constinit
co_await	co_return	co_yield	requires

2.2.2. Cách ghi chú thích trong C++

Ngoài kiểu chú thích trong C bằng /* ... */ , C++ thêm một kiểu chú thích thứ hai, đó là chú thích bắt đầu bằng //. Nói chung, kiểu chú thích /*...*/ được dùng cho các khối chú thích lớn gồm nhiều dòng, còn kiểu // được dùng cho các chú thích trên một dòng.

Ví dụ: Chương trình xuất ra màn hình "Hello world!"

```
#include <iostream>  //Thư viện nhập/xuất trong C++
using namespace std; /Khai báo không gian tên mặc định
/* Cách ghi chú thích (comment) trong C,
có thể ghi chú thích trên nhiều dòng
*/
// C++ thêm //, dùng để ghi chú thích trên một dòng
/**************
 *  có thể có nhiều dấu *, nhưng
 *  không thể lồng chú thích vào nhau
 ***************/
/* Đây là chú thích cha /* chú thích con */ báo lỗi */
// Chú thích 1 dòng /*chú thích nhiều dòng (trên 1 dòng)*/: vẫn là chú thích
void main()
{
     cout << "Hello world!\n";
     cout << endl;
}
```

Trong đó:

namespace (không gian tên): là công cụ cho phép quản lý sự xung đột về tên của các thành phần của chương trình như tên biến, tên lớp, tên hàm,…

cout và toán tử << nằm trong thư viện iostream, xuất dữ liệu ra màn hình.

"\n" và endl: kí tự xuống dòng. Trong C++, để kết thúc một dòng sang dòng mới ta có thể sử dụng endl hoặc "\n" (std::endl gửi một ký tự xuống dòng "\n" và xóa bộ nhớ đệm; còn "\n" gửi một ký tự xuống dòng, nhưng không xóa bộ đệm).

Kết quả:

```
Hello world

_
```

2.2.3. Vị trí khai báo biến

Trong C tất cả các khai báo biến bên trong một phạm vi cho trước phải được đặt ở ngay đầu của phạm vi đó. Điều này có nghĩa là tất cả các khai báo toàn cục phải đặt trước tất cả các hàm và các khai báo cục bộ phải được tiến hành trước tất cả các lệnh thực hiện. Vì vậy vị trí khai báo và vị trí sử dụng của biến có thể ở cách khá xa nhau, điều này gây khó khăn trong việc kiểm soát chương trình. C++ đã khắc phục nhược điểm này bằng cách cho phép các lệnh khai báo biến có thể đặt bất kỳ chỗ nào trong chương trình trước khi các biến được sử dụng.

Ví dụ: Khai báo biến trước khi sử dụng

```
#include <iostream>   //Thư viện nhập/xuất trong C++
using namespace std;  //Khai báo không gian tên mặc định

void main()
{
      int a;     //Khai báo biến trước khi sử dụng
      cout << "Nhap so nguyen thu nhat: ";
      cin >> a;
      int b;     //Khai báo biến trước khi sử dụng
      cout << "Nhap so nguyen thu hai: ";
      cin >> b;
      char toanTu;     //Khai báo biến trước khi sử dụng
      cout << "Nhap toan tu (+ hoac -): ";
      cin >> toanTu;
      switch (toanTu)
      {
            case '+':  cout << "Tong = " << a + b;
                       break;
            case '-':  cout << "Hieu = " << a - b;
                       break;
      }
}
```

Kết quả:

```
Nhap so nguyen thu nhat: 1
Nhap so nguyen thu hai: 2
Nhap toan tu (+ hoac -): +
Tong = 3
```

2.2.4. Chuyển đổi kiểu dữ liệu

Ngoài phép chuyển kiểu bắt buộc trong C theo cú pháp:

```
(kiểu_dữ_liệu) biểu thức.
```

C++ thêm cách chuyển kiểu mới có cú pháp như sau:

```
kiểu_dữ_liệu (biểu thức).
```

Phép chuyển kiểu này có dạng như một hàm số chuyển kiểu đang được gọi. Cách chuyển kiểu này thường được sử dụng trong thực tế.

Ví dụ: Chuyển đổi kiểu dữ liệu

```
#include <iostream>  //Thư viện nhập/xuất trong C++
using namespace std; //Khai báo không gian tên mặc định

void main()
{
	int x = 10;
	long y = (long) x;   //Chuyen doi kieu cua C
	long z = long (x);   //Chuyen doi kieu cua C++ bổ sung
	cout << "x = " << x << endl;
	cout << "y = " << y << endl;
	cout << "z = " << z;
}
```

Kết quả:

```
x = 10
y = 10
z = 10
```

2.2.5. Nhập xuất trong C++

Để nhập dữ liệu từ bàn phím và xuất dữ liệu ra màn hình, trong C++ vẫn có thể dùng hàm printf() và scanf() như trong C. Ngoài ra trong C++ có thể dùng dòng nhập xuất chuẩn để nhập xuất dữ liệu thông qua hai biến đối tượng của dòng dữ liệu (stream object) là cin cout.

2.2.5.1. Nhập dữ liệu

Cú pháp:

```
cin >> biến_1 >> biến_2 >> … >> biến_n;
```

Toán tử cin được định nghĩa trước như một đối tượng biểu diễn cho thiết bị nhập chuẩn của C++ là bàn phím, cin được sử dụng kết hợp với toán tử trích luồng >> để nhập dữ liệu từ bàn phím cho các biến 1, 2, ..., n.

Chú ý:

– Để nhập một chuỗi không quá n ký tự và lưu vào mảng một chiều a (kiểu char) có thể dùng hàm cin.get như sau:

```
cin.get(a,n);
```

– Toán tử nhập cin>> sẽ để lại ký tự chuyển dòng '\n' trong bộ đệm. Ký tự này có thể làm trôi phương thức cin.get. Để khắc phục tình trạng trên cần dùng phương thức cin.ignore(1) để bỏ qua một ký tự chuyển dòng.

– Để sử dụng các toán tử và phương thức trên, cần khai báo thư viện iostream.

2.2.5.2. Xuất dữ liệu

Cú pháp:

```
cout << biểu_thức_1 << biểu_thức_2 << … << biểu_thức_n;
```

Toán tử cout được định nghĩa trước như một đối tượng biểu diễn cho thiết bị xuất chuẩn của C++ là màn hình, cout được sử dụng kết hợp với toán tử chèn luồng << để hiển thị giá trị các biểu thức 1, 2,..., n ra màn hình.

2.2.5.3. Định dạng khi in ra màn hình

Để quy định số thực được hiển thị ra màn hình với p chữ số sau dấu chấm thập phân, ta sử dụng đồng thời các hàm sau:

```
setiosflags(ios::showpoint); //Bật cờ hiệu showpoint(p)
setprecision(p);
```

Các hàm này cần đặt trong toán tử xuất như sau:

```
cout << setiosflags(ios::showpoint) << setprecision(p);
```

Câu lệnh trên sẽ có hiệu lực đối với tất cả các toán tử xuất tiếp theo cho đến khi gặp một câu lệnh định dạng mới.

Để quy định độ rộng tối thiểu để hiển thị là k vị trí cho giá trị (nguyên, thực, chuỗi) ta dùng hàm setw(k). Hàm này cần đặt trong toán tử xuất và nó chỉ có hiệu lực cho một giá trị được xuất gần nhất. Các giá trị in ra tiếp theo sẽ có độ rộng tối thiểu mặc định là 0, như vậy câu lệnh:

```
cout << setw(10) << "Welcome" << setw(10) << "to C++!" << endl;
```

sẽ xuất ra màn hình chuỗi:

```
  Welcome    to C++!
```

Trước "Wellcome" có 2 khoảng trắng, trước "to C++!" có 3 khoảng trắng.

– Để sử dụng các toán tử và phương thức trên, cần khai báo thư viện iostream.

2.2.6. Toán tử định phạm vi

Toán tử định phạm vi (scope resolution operator) ký hiệu là ::, nó được dùng truy xuất một phần tử bị che bởi phạm vi hiện thời.

Ví dụ:

```
#include <iostream>  //Thư viện nhập/xuất trong C++
using namespace std; //Khai báo không gian tên mặc định
int x = 10;
void main()
{
     int x = 20;
     cout << "Bien x ben trong = " << x << endl;
     cout << "Bien x ben ngoai = " << ::x;
}
```

Kết quả:

```
Bien x ben trong = 20
Bien x ben ngoai = 10
```

Toán tử định phạm vi còn được dùng trong các định nghĩa hàm của các phương thức trong các lớp, để khai báo lớp chủ của các phương thức đang được định nghĩa đó. Toán tử định phạm vi còn có thể được dùng để phân biệt các thành phần trùng tên của các lớp cơ sở khác nhau.

2.2.7. Cấp phát và giải phóng bộ nhớ

Trong C, tất cả các cấp phát động bộ nhớ đều được xử lý thông qua các hàm thư viện như malloc(), calloc() và hàm free() để giải phóng vùng nhớ được cấp phát. Ngôn ngữ C++ định nghĩa một phương thức mới để thực hiện việc cấp phát động bộ nhớ bằng cách dùng hai toán tử new và delete. Sử dụng hai toán tử này sẽ linh hoạt hơn rất nhiều so với các hàm thư viện của C.

2.2.7.1. Toán tử new để cấp phát bộ nhớ

Toán tử new thay cho hàm malloc() và calloc() của C có cú pháp như sau:

```
new Kiểu_dữ _liệu;
```

hoặc

```
new (Kiểu_dữ _liệu);
```

Trong đó:

Kiểu_dữ _liệu là kiểu dữ liệu của biến con trỏ, có thể là các kiểu dữ liệu cơ sở như int, float, double, char,... hoặc các kiểu dữ liệu do người lập trình định nghĩa như mảng, cấu trúc, lớp,...

Để cấp phát bộ nhớ cho mảng một chiều, dùng cú pháp như sau:

```
Biến_con_trỏ = new kiểu_dữ_liệu[n];
```

n là số nguyên dương, xác định số phần tử của mảng.

Ví dụ:

```
float* p = new float;      //cấp phát bộ nhớ cho biến con trỏ p có kiểu float
int* a = new int[100];     //cấp phát bộ nhớ cho mảng 1 chiều a 100 phần tử
```

Khi sử dụng toán tử new để cấp phát bộ nhớ, nếu không đủ bộ nhớ để cấp phát, new sẽ trả lại giá trị NULL cho con trỏ.

Đoạn chương trình sau dùng để cấp phát vùng nhớ động của C:	Trong C++, có thể viết lại đoạn chương trình như sau:

`int* p;`
 	
`p = malloc(sizeof(int));`
`if (p == NULL)`
`  printf("Khong du bo nho!\n");`
`else`
`{`
`  *p = 100;`
`  printf("%d\n", *p);`
`  free(p);`
`}`
	`int* p;`
	
	`p = new int;`
	`if (p == NULL)`
	`  cout << "Khong du bo nho!\n";`
	`else`
	`{`
	`  *p = 100;`
	`  cout << *p << endl;`
	`  delete p;`
	`}`

2.2.7.2. Toán tử delete

Toán tử delete thay cho hàm free() của C, nó có cú pháp như sau:

```
delete con_trỏ;
```

Ví dụ:

```
void main()
{
	cout << "Nhap so phan tu mang: ";
	int length;
	cin >> length;

	int* array = new int[length]; // kích thước mảng có thể là biến số

	//Gán giá trị cho phần tử mảng
	for (int i = 0; i < length; i++)
		array[i] = rand();
	//Xuất mảng
	cout << "Mang: ";
	for (int i = 0; i < length; i++)
		cout << array[i] << " ";

	delete[] array; // trả lại vùng nhớ mảng array
}
```

Kết quả:

```
Nhap so phan tu mang: 7
Mang: 41 18467 6334 26500 19169 15724 11478
```

2.2.8. Các biến const

Trong ANSI C, muốn định nghĩa một hằng có kiểu nhất định thì chúng ta dùng biến const (vì nếu dùng #define thì tạo ra các hằng không có chứa thông tin về kiểu). Trong C++, const cũng được xem như #define nếu như chúng ta muốn dùng hằng có tên trong chương trình.

Ví dụ: dùng const để quy định kích thước của một mảng như đoạn mã sau:

```
const int arraySize = 100;
int X[arraySize];
```

Khi khai báo một biến const trong C++ thì chúng ta phải khởi tạo một giá trị ban đầu nhưng đối với ANSI C thì không nhất thiết phải làm như vậy (vì trình biên dịch

ANSI C tự động gán trị zero cho biến const nếu chúng ta không khởi tạo giá trị ban đầu cho nó).

Phạm vi của các biến const giữa ANSI C và C++ khác nhau. Trong ANSI C, các biến const được khai báo ở bên ngoài mọi hàm có phạm vi toàn cục, điều này nghĩa là chúng có thể nhìn thấy cả ở bên ngoài file mà chúng được định nghĩa, trừ khi chúng được khai báo là static. Nhưng trong C++, các biến const được hiểu mặc định là static.

2.2.9. Biến tham chiếu

Trong C có 2 loại biến là biến giá trị dùng để chứa dữ liệu (nguyên, thực, ký tự,...) và biến con trỏ dùng để chứa địa chỉ. Các biến này đều được cấp bộ nhớ và có địa chỉ.

C++ cho phép sử dụng loại biến thứ ba là biến tham chiếu. Biến tham chiếu có đặc điểm là dùng làm bí danh cho một biến đã định nghĩa trước đó và sử dụng vùng nhớ của biến này. Cú pháp khai báo biến tham chiếu như sau:

```
Kiểu_dữ_liệu& Biến_tham_chiếu = Biến;
```

Ví dụ:

```
#include <iostream>  //Thư viện nhập/xuất trong C++
using namespace std; //Khai báo không gian tên mặc định

void main()
{
	int a, &b = a;
	a = 1;	//b = 1
	cout << "b = " << b << endl; //Xuất ra b = 1
	b++; //a = 2
	cout << "a = " << a; //Xuất ra a = 2
}
```

Kết quả:

```
b = 1
a = 2
```

Với câu lệnh: int a, &b = a; thì b là bí danh của biến a và biến b dùng chung vùng nhớ của biến a. Lúc này, trong mọi câu lệnh, viết a hay b đều có ý nghĩa như nhau, vì đều truy nhập đến cùng một vùng nhớ. Mọi sự thay đổi đối với biến b đều ảnh hưởng đối với biến a và ngược lại.

Chú ý:

– Trong khai báo biến tham chiếu phải chỉ rõ tham chiếu đến biến nào.

– Biến tham chiếu có thể tham chiếu đến một phần tử mảng, nhưng không cho phép khai báo mảng tham chiếu.

– Biến tham chiếu có thể tham chiếu đến một hằng. Khi đó nó sử dụng vùng nhớ của hằng và có thể làm thay đổi giá trị chứa trong vùng nhớ này.

– Biến tham chiếu thường được sử dụng làm đối số của hàm để cho phép hàm truy nhập đến các tham biến trong lời gọi hàm.

2.2.10. Hằng tham chiếu

Cú pháp khai báo hằng tham chiếu như sau:

```
const Kiểu_dữ_liệu &Biến = Hằng/Biến;
```

Ví dụ:

```
int x = 10;
const int& y = x;
const int& z = 10;
```

Hằng tham chiếu có thể tham chiếu đến một biến hoặc một hằng.

Chú ý:

Biến tham chiếu và hằng tham chiếu khác nhau ở chỗ: không cho phép dùng hằng tham chiếu để làm thay đổi giá trị của vùng nhớ mà nó tham chiếu.

Ví dụ:

```
int x = 10, y;
const int& z = x     //Hằng tham chiếu z tham chiếu đến biến x
z = z + 10;          //Trình biên dịch sẽ thông báo lỗi
```

Hằng tham chiếu cho phép sử dụng giá trị chứa trong một vùng nhớ, nhưng không cho phép thay đổi giá trị này.

Hằng tham chiếu thường được sử dụng làm đối số của hàm để cho phép sử dụng giá trị của các tham số trong lời gọi hàm, nhưng tránh làm thay đổi giá trị tham số.

2.2.11. Truyền tham số cho hàm theo tham chiếu

2.2.11.1. Truyền tham số cho hàm theo tham chiếu (passing arguments by reference)

Trong C chỉ có một cách truyền dữ liệu cho hàm là truyền theo theo giá trị (passing arguments by value), chương trình sẽ tạo ra các bản sao của các tham số trong lời gọi hàm và sẽ thao tác trên các bản sao này chứ không xử lý trực tiếp với các tham số thực sự. Cơ chế này rất tốt nếu khi thực hiện hàm trong chương trình không cần làm thay đổi giá trị của biến gốc. Mặc dù truyền giá trị cho hàm là phương pháp thường được sử dụng nhất, vì tính linh hoạt và an toàn. Nhưng nó vẫn có 2 hạn chế:

– Gây giảm hiệu suất trong trường hợp đối số là kiểu cấu trúc (structs) hoặc các lớp (classes), đặc biệt là nếu hàm đó được gọi nhiều lần. Vì mỗi lần gọi hàm đều phải sao chép giá trị của đối số vào tham số của hàm.

– Hàm chỉ có thể trả về một giá trị duy nhất bằng câu lệnh return. Trong nhiều trường hợp, hàm cần trả về nhiều thông tin hơn, cách này không đáp ứng được.

Phương pháp truyền tham chiếu cho hàm:

– Trong C++, tham chiếu (reference) là một loại biến hoạt động như một bí danh của biến khác.
– Khai báo bằng cách sử dụng ký hiệu “&” giữa kiểu dữ liệu và tên biến.
– Mọi thay đổi trên biến tham chiếu cũng chính là thay đổi trên biến được tham chiếu.

Ví dụ: Để truyền tham chiếu cho hàm, chỉ cần khai báo các tham số (parameters) của hàm dưới dạng tham chiếu (references):

```
#include <iostream>  //Thư viện nhập/xuất trong C++
using namespace std; //Khai báo không gian tên mặc định
void callByReferences(int& a, int &b)    //a, b là biến tham chiếu
{
     cout << "Truoc khi hoan vi: a = " << a << " va b = " << b << endl;
     int temp = a;
     a = b;
     b = temp;
     cout << "Sau khi hoan vi: a = " << a << " va b = " << b << endl;
}    //biến a, b đã được giải phóng
```

```
void main()
{
	int x(5), y(10);
	cout << "Truoc khi goi ham: x = " << x << " va y = " << y << endl;

	callByReferences(x,y);

	cout << "Sau khi goi ham: x = " << x << " va y = " << y << endl;
}
```

Kết quả:

```
Truoc khi goi ham: x = 5 va y = 10
Truoc khi hoan vi: a = 5 va b = 10
Sau khi hoan vi: a = 10 va b = 5
Sau khi goi ham: x = 10 va y = 5
```

Trong chương trình trên, khi hàm callByReferences(int &a, int &b) được gọi, a, b sẽ trở thành một tham chiếu đến đối số x, y. Mọi thay đổi của biến a, b bên trong hàm callByReferences() cũng chính là thay đổi trên biến x, y.

2.2.11.2. Trả về nhiều giá trị thông qua tham số đầu ra

Đôi khi một cần trả về nhiều giá trị, trong khi, hàm chỉ có một giá trị trả về. Một trong những cách để hàm trả về nhiều giá trị là sử dụng tham số tham chiếu:

```
#include <iostream>  //Thư viện nhập/xuất trong C++
using namespace std; //Khai báo không gian tên mặc định
void calc(int x, int y, int& tong, int& hieu)
{
	tong = x + y;
	hieu = x - y;
}
void main()
{
	int a(5), b(10);
	int add, sub;
	// hàm calc sẽ trả về tong va hieu trong 2 biến add và sub
	calc(a, b, add, sub);
	cout << "a + b = " << add << endl;
	cout << "a - b = " << sub << endl;
}
```

Kết quả:

```
a + b = 15
a - b = -5
```

2.2.11.3. Truyền tham chiếu hằng (pass by const reference)

Truyền tham chiếu cho hàm đã giải quyết được vấn đề hiệu suất của truyền giá trị. Nhưng truyền tham chiếu cho phép hàm thay đổi giá trị của các đối số (arguments), điều này sẽ là nguy hiểm tiềm ẩn nếu không muốm làm thay đổi giá trị của biến tham chiếu. Khi đó, giải pháp tốt nhất là truyền tham chiếu hằng.

Tham chiếu hằng (const reference) là một tham chiếu mà không cho phép biến được tham chiếu thay đổi thông qua biến tham chiếu. Đối số của tham chiếu hằng có thể là biến số, hằng số hoặc biểu thức.

```
void xuatX(const int& x)  // x là biến tham chiếu hằng
{
     // compile error: một tham chiếu hằng không thể thay đổi giá trị
     x = x + 2;
}

int main()
{
     int x(5);

     xuatX(x);       // tham số là biến
     xuatX(15);      // tham số là hằng
     xuatX(x + 5);   // tham số là  biểu thức
}
```

Trong chương trình trên, vì x là tham chiếu hằng, giá trị của nó không thể thay đổi. Nên dòng lệnh x = x + 2; bên trong hàm xuatX(const int &x) bị lỗi biên dịch.

2.2.12. Hàm trả về giá trị tham chiếu

C++ cho phép hàm trả về giá trị là một tham chiếu, lúc này định nghĩa của hàm có dạng như sau :

```
     Kiểu_dữ_liệu& Tên_hàm(danh_sách_tham_số)
     {
          //thân hàm
          return     <biến_phạm_vi_toàn_cục>;
     }
```

Trong trường hợp này biểu thức được trả lại trong câu lệnh return phải là tên của một biến xác định từ bên ngoài hàm, bởi vì khi đó mới có thể sử dụng được giá

trị của hàm. Khi ta trả về một tham chiếu đến một biến cục bộ khai báo bên trong hàm, biến cục bộ này sẽ bị mất đi khi kết thúc thực hiện hàm. Do vậy tham chiếu của hàm sẽ không còn ý nghĩa nữa.

Khi giá trị trả về của hàm là tham chiếu, ta có thể gặp các câu lệnh gán hơi khác thường, trong đó vế trái là một lời gọi hàm chứ không phải là tên của một biến. Điều này hoàn toàn hợp lý, bởi vì bản thân hàm đó có giá trị trả về là một tham chiếu. Nói cách khác, vế trái của lệnh gán có thể là lời gọi đến một hàm có giá trị trả về là một tham chiếu. Xem các ví dụ sau:

Ví dụ:

```
#include <iostream>       //Thư viện nhập/xuất trong C++
using namespace std;      //Khai báo không gian tên mặc định

int& max(int& a, int& b)
{
     return a > b ? a : b;
}
void main()
{
     int a = 5, b = 10, c = 15;
     cout << "Max cua a, b la: "<<max(a,b) << endl;
     max(a, b)++;
     cout << "Gia tri a va b la: " << a << " va " << b << endl;
     max(b, c) = 20;
     cout << "Gia tri a, b, va c la: " << a << ", " << b << ", va " << c;
}
```

Kết quả:

```
Max cua a, b la: 10
Gia tri a va b la: 5 va 11
Gia tri a, b, va c la: 5, 11, va 20
```

2.2.13. Hàm với đối số có giá trị mặc định

Một trong các đặc tính nổi bật của C++ là khả năng định nghĩa các giá trị tham số mặc định cho các hàm. Bình thường khi gọi hàm, cần gởi một giá trị cho mỗi tham số đã được định nghĩa trong hàm đó. Trong C++, giá trị mặc định được truyền vào đối số trong nguyên mẫu hàm. Nếu các đối số không được truyền vào trong khi gọi hàm, thì giá trị mặc định đó sẽ được sử dụng.

Ví dụ:

```
#include <iostream>       //Thư viện nhập/xuất trong C++
using namespace std;      //Khai báo không gian tên mặc định

void xuat(char x = '*', char y = '*')
{
     cout << "x = " << x << endl;
     cout << "y = " << y << endl;
}

void main()
{
     cout << "Khong truyen tham so x, y:" << endl;
     xuat();
     cout << "Khong truyen tham so x:" << endl;
     xuat('x');
     cout << "Truyen ca 2 tham so x, y:" << endl;
     xuat('x', 'y');
}
```

Kết quả:

```
Khong truyen tham so x, y:
x = *
y = *
Khong truyen tham so x:
x = x
y = *
Truyen ca 2 tham so x, y:
x = x
y = y
```

2.2.14. Hàm nội tuyến trong C++ (Inline functions)

Khi một hàm được gọi, CPU sẽ lưu địa chỉ bộ nhớ của dòng lệnh hiện tại mà nó đang thực thi (để biết nơi sẽ quay lại sau lời gọi hàm), sao chép các đối số của hàm trên ngăn xếp (stack) và cuối cùng chuyển hướng điều khiển sang hàm đã chỉ định. CPU sau đó thực thi mã bên trong hàm, lưu trữ giá trị trả về của hàm trong một vùng nhớ/thanh ghi và trả lại quyền điều khiển cho vị trí lời gọi hàm. Điều này sẽ tạo ra một lượng chi phí hoạt động nhất định so với việc thực thi mã trực tiếp (không sử dụng hàm).

Đối với các hàm lớn hoặc các tác vụ phức tạp, tổng chi phí của lệnh gọi hàm thường không đáng kể so với lượng thời gian mà hàm mất để chạy. Tuy nhiên, đối với các hàm nhỏ, thường được sử dụng, thời gian cần thiết để thực hiện lệnh gọi hàm thường nhiều hơn rất nhiều so với thời gian cần thiết để thực thi mã của hàm.

Trong C, chúng ta thường sử dụng hàm macro, một kỹ thuật tối ưu hóa được sử dụng bởi trình biên dịch để giảm thời gian thực hiện. Trong C++ cung cấp một khái niệm mới tốt hơn, đó là hàm nội tuyến (inline functions).

Inline functions (hàm nội tuyến) là một loại hàm trong ngôn ngữ lập trình C++. Từ khoá inline được sử dụng để đề nghị (không phải là bắt buộc) trình biên dịch thực hiện inline expansion (khai triển nội tuyến) với hàm đó hay nói cách khác là chèn code của hàm đó tại địa chỉ mà nó được gọi.

Ví dụ:

```
#include <iostream>        //Thư viện nhập/xuất trong C++
using namespace std;       //Khai báo không gian tên mặc định

inline int max(int a, int b)
{
     return a > b ? a : b;
}

void main()
{
     cout << max(5, 10) << endl;
     cout << max(10, 5) << endl;
}
```

Khi chương trình trên được biên dịch, mã máy được tạo như sau:

```
void main()
{
     cout << (5 > 10 ? 5 : 10) << endl;
     cout << (10 > 5 ? 10 : 5) << endl;
}
```

Chương trình dịch các hàm inline tương tự như các macro, nghĩa là nó sẽ thay đổi lời gọi hàm bằng một đoạn chương trình thực hiện nhiệm vụ hàm. Cách làm này sẽ tăng tốc độ chương trình do không phải thực hiện các thao tác có tính thủ tục khi

gọi hàm nhưng lại làm tăng khối lượng bộ nhớ chương trình (nhất là đối với các hàm nội tuyến có nhiều câu lệnh). Vì vậy chỉ nên dùng hàm inline đối với các hàm có nội dung đơn giản.

Ví dụ:

```
#include <iostream>       //Thư viện nhập/xuất trong C++
using namespace std;      //Khai báo không gian tên mặc định

inline void tinhDTCV(int chieuDai, int chieuRong, int& dienTich, int& chuVi)
{
      dienTich = chieuDai * chieuRong;
      chuVi = 2 * (chieuDai + chieuRong);
}
void main()
{
      int dai[3], rong[3], cv[3], dt[3];
      for (int i = 0; i < 3; ++i)
      {
            cout << "Nhap 2 dai va chieu cua hinh thu " << i + 1 << ": ";
            cin >> dai[i] >> rong[i];
            tinhDTCV(dai[i], rong[i], dt[i], cv[i]);
      }

      for (int i = 0; i < 3; ++i)
      {
            cout << "Hinh chu nhat thu " << i + 1 << ": " << endl;
            cout << "Do dai canh: " << dai[i] << " va " << rong[i] << endl;
            cout << "Dien tich: " << dt[i] << " va Chu vi: " << cv[i] << endl;
      }
}
```

Kết quả:

```
Nhap 2 dai va chieu cua hinh thu 1: 1 2
Nhap 2 dai va chieu cua hinh thu 2: 3 4
Nhap 2 dai va chieu cua hinh thu 3: 5 6
Hinh chu nhat thu 1:
Do dai canh: 1 va 2
Dien tich: 2 va Chu vi: 6
Hinh chu nhat thu 2:
Do dai canh: 3 va 4
Dien tich: 12 va Chu vi: 14
Hinh chu nhat thu 3:
Do dai canh: 5 va 6
Dien tich: 30 va Chu vi: 22
```

2.2.15. Nạp chồng hàm (function overloading)

Nạp chồng hàm là các hàm có cùng tên nhưng có tập đối số khác nhau (số lượng hoặc kiểu dữ liệu). Khi gặp lời gọi các hàm tải bội thì trình biên dịch sẽ căn cứ vào số lượng và kiểu các tham số để gọi hàm có đúng tên và đúng các đối số tương ứng.

```
#include <iostream>       //Thư viện nhập/xuất trong C++
using namespace std;      //Khai báo không gian tên mặc định

void max(char a, char b)
{
     cout << "Day la ham max kieu char" << endl;
     //Thân hàm
}
void max(int a, int b)
{
     cout << "Day la ham max kieu int" << endl;
     //Thân hàm
}
void max(float a, float b)
{
     cout << "Day la ham max kieu float" << endl;
     //Thân hàm
}
void max(double a, double b)
{
     cout << "Day la ham max kieu double" << endl;
     //Thân hàm
}
void main()
{
     char a = 'a', b = 'b';
     max(a, b);      //Hàm max kiểu char
     max(2, 5);      //Hàm max kiểu int
     float x = 2.0, y = 5.5;
     max(x, y);      //Hàm max kiểu float
     max(2.0, 5.5);  //Hàm max kểu double
}
```

Kết quả:

```
Day la ham max kieu char
Day la ham max kieu int
Day la ham max kieu float
Day la ham max kieu double
```

BÀI TẬP CHƯƠNG 2

Bài 01: Viết chương trình minh hoạ cách dùng new để cấp phát bộ nhớ chứa n thí sinh. Mỗi thí sinh là một cấu trúc gồm các trường hoTen(Họ và tên), soBD(Số báo danh), và tongDiem(Tổng điểm). Chương trình sẽ nhập n, cấp phát bộ nhớ chứa n thí sinh, kiểm tra lỗi cấp phát bộ nhớ, nhập n thí sinh, sắp xếp thí sinh theo thứ tự giảm của tổng điểm, in danh sách thí sinh sau khi sắp xếp, giải phóng bộ nhớ đã cấp phát.

Bài 02: Viết chương trình tính điện trở tương đương của 2 điện trở mắc song song theo phương pháp hướng đối tượng. Giá trị của các điện trở được nhập từ bàn phím.

Bài 03: Mở rộng bài 2 cho việc tính điện trở của N điện trở mắc song song. Hơn nữa, khi nhập dữ liệu cho các điện trở cần kiểm tra tính hợp lệ của dữ liệu nhập vào.

Bài 04: Viết chương trình trong đó có hai hàm swap chồng nhau, một hàm cho phép hoán đổi giá trị của hai kí tự, còn hàm kia cho phép hoán đổi giá trị của hai chuỗi kí tự. Chương trình sẽ thực hiện việc hoán đổi 1 cặp kí tự và 1 cặp chuỗi có giá trị nhập từ bàn phím.

CHƯƠNG 3

LỚP VÀ ĐỐI TƯỢNG

Các nội dung chính

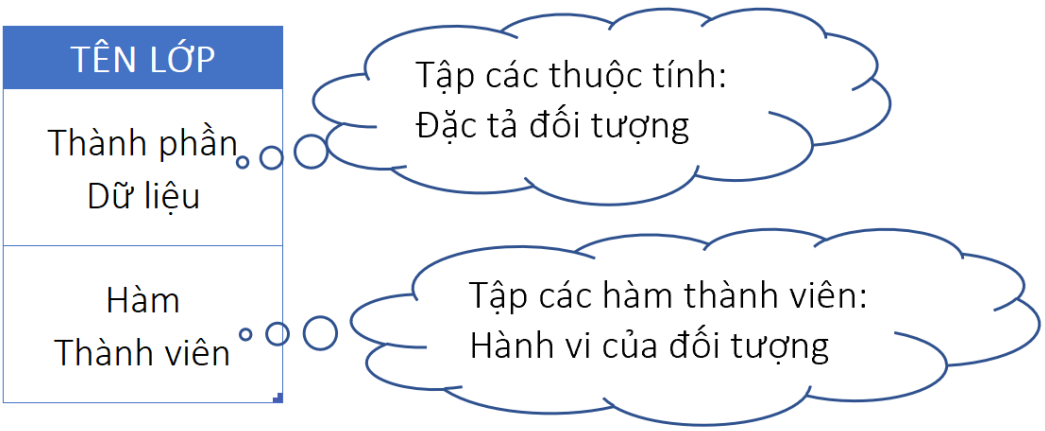
- *Lớp (class)*
- *Đối tượng (objects)*
- *Truy cập các thành phần của lớp*
- *Con trỏ đối tượng*
- *Tự tham chiếu (con trỏ this)*
- *Hàm khởi tạo, hàm hủy*
- *Các thành phần tĩnh (thành phần kiểu static)*

3.1. Lớp

3.1.1. Định nghĩa lớp

Trong lập trình hướng đối tượng, mỗi đối tượng đều phải thuộc về một lớp nào đó. Nên định nghĩa một lớp mới là xây dựng lớp đó để chuẩn bị tạo ra các đối tượng của lớp đó. Lớp là một kiểu dữ liệu trừu tượng.

Định nghĩa một lớp mới cho phép tạo ra một lớp mới, bao gồm các thành phần dữ liệu và các hàm thành viên cần thiết.



Hình 3.1. Lớp (class)

Cú pháp:

```
class <Tên_lớp>
{
//Định nghĩa các thành phần dữ liệu
<Phạm vi truy cập:>
     <Type> memberdata1;
     <Type> memberdata2;
     <Type> memberdata3;
     ...

//Định nghĩa các hàm thành viên
<Phạm vi truy cập:>
     <Type> memberFunction1(Các_tham_số);
     <Type> memberFunction2(Các_tham_số);
     <Type> memberFunction3(Các_tham_số);
     ...
};
```

<Phạm vi truy cập:>: từ khóa xác định mức độ che dấu (hay thuộc tính truy xuất): private, public hoặc protected.

<Type>: kiểu dữ liệu hoặc kiểu hàm (có thể là kiểu một lớp đã được định nghĩa).

Thuộc tính của lớp được gọi là thành phần dữ liệu và hàm được gọi là phương thức (hàm thành viên). Thuộc tính và hàm được gọi chung là các thành phần của lớp.

Ví dụ:

```
class Circle
{
private:
     const float PI = 3.1415; //Hằng số PI
     float r;  //Bán kính, thành phần dữ liệu của từng đối tượng

public:
     void setRadius(float bKinh);
     float getRadius();
     float area();
};
```

3.1.2. Khai báo lớp

Lớp có thể đặt trước hoặc sau hàm main(), nhưng không được định nghĩa một lớp trong một lớp khác.

Định nghĩa lớp trước hàm main	Định nghĩa lớp sau hàm main

class Circle
{
//Định nghĩa lớp trước hàm main
...
};
 	
void main()
{
Circle c, c2;
…
}
	class Circle; //Khai báo lớp
	void main()
	{
	Circle c, c2;
	…
	}
	class Circle
	{
	//Định nghĩa lớp sau hàm main
	...
	};

3.1.3. Khai báo thành phần dữ liệu (thuộc tính):

Cú pháp:

```
<Phạm vi truy cập> <Type> memberdata;
```

Thành phần dữ liệu của lớp được khai báo như khai báo biến, nó không thể có kiểu chính của lớp đó, nhưng có thể là kiểu con trỏ của lớp này.

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 truy cập trái phép vào dữ liệu của lớp.

Ví dụ:

```
class Circle
{
private:
     int r;     //Khai báo thuộc tính bán kính
     Circle s; //Báo lỗi, vì x có kiểu của chính lớp định nghĩa
     Circle* p;//Không báo lỗi, vì p là con trỏ
};
```

3.1.4. Khai báo thành phần hàm

Cú páp:

```
<Phạm vi truy cập> <Type> memberFunction(Các_tham_số);
```

Hàm bao gồm các hàm thành viên (còn được gọi là hàm thành phần) là hàm thuộc lớp, và sẽ thuộc về các đối tượng của lớp đó và hàm tự do là các hàm được định nghĩa bên ngoài các lớp, chính là hàm con trong C.

Các hàm thành phần khai báo là public có thể được gọi tới từ các hàm thành phần public khác trong chương trình.

Ví dụ:

```
public void setRadius(float bKinh)
{
	r = bKinh;
}
```

Các hàm thành viên 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 viên viết bên trong định nghĩa lớp được viết như hàm thông thường.

Ví dụ:

```
class Circle
{
private:
	const float PI = 3.1415; //Hằng số PI
	float r;  //Bán kính, thành phần dữ liệu của từng đối tượng

public:
	void setRadius(float bKinh)
	{
		r = bKinh;
	}
	float getRadius()
	{
		return r;
	}
};
```

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(danh sách 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.

Ví dụ:

```
class Circle
{
private:
     const float PI = 3.1415; //Hằng số PI
     float r;  //Bán kính, thành phần dữ liệu của từng đối tượng

public:
     float area();
};
float Circle::area()
{
     return PI * r * r;
}
```

Lưu ý:

– *Các thành phần private chỉ được sử dụng bên trong lớp, các thành phần public được phép sử dụng ở cả bên trong và bên ngoài lớp.*

3.2. Khai báo đối tượng

Sau khi định nghĩa lớp, để tạo đối tượng ta có thể khai báo các biến thuộc kiểu lớp. Cú pháp khai báo biến đối tượng như sau:

```
Tên_lớp Danh_sách_biến;
```

Ví dụ:

```
Circle c1, c2;
```

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

```
class Tên_lớp
{    ...
} <danh_sách_biến>;
```

Ví dụ:

```
class Circle
{    ...
} c1, c2;
```

Mỗi đối tượng sau khi khai báo sẽ được cấp phát một vùng nhớ riêng để chứa các thuộc tính của nó. Không có vùng nhớ riêng để chứa các hàm thành phần của đối tượng. Các hàm thành phần được sử dụng chung cho tất cả các đối tượng cùng lớp.

3.3. Truy cập các thành phần của lớp

Sau khi khai báo đối tượng, để truy nhập đến dữ liệu thành phần của lớp, ta sử dụng toán tử chấm ".", cú pháp như sau:

```
Tên_đối_tượng.Tên_thuộc_tính;
```

Lưu ý:

– *Thành phần dữ liệu có khai báo là private (dành riêng) chỉ có thể được truy cập bởi những hàm thành phần của cùng một lớp, đối tượng của lớp cũng không thể truy cập được.*

Và để sử dụng các hàm thành phần của lớp, ta dùng cú pháp như sau:

```
Tên_đối_tượng.Tên_hàm (giá_trị_các_khai_báo_tham_số);
```

Ví dụ:

```
//Định nghĩa lớp Circle trong ví dụ phần 3.1.1
void main()
{
      Circle c;       //Khai báo và sử dụng đối tượng thông thường
      c.setRadius(10);
      cout << "Dien tich duong tron ban kinh r = " << c.getRadius()
                                 << " la " << c.area() << endl;
}
```

3.4. Con trỏ đối tượng

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

```
Tên_lớp* Tên_con_ trỏ;
```

Ví dụ:

```
//Định nghĩa lớp Circle trong ví dụ phần 3.1
void main()
{
      Circle* pc = new Circle;    //Khai báo và khởi tạo đối tượng kiểu con trỏ
      Circle c;  //Khai báo và sử dụng đối tượng thông thường
      c.setRadius(10);
      pc = &c;   //Con trỏ pc trỏ đến đối tượng c
      cout << "Dien tich duong tron ban kinh r = " << pc->getRadius()
            << " la " << pc->area() << endl;
}
```

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

```
Tên_con_trỏ -> Tên_thuộc_tính
Tên_con_trỏ -> Tên_hàm (giá_trị_các_khai_báo_tham_số);
```

Ví dụ:

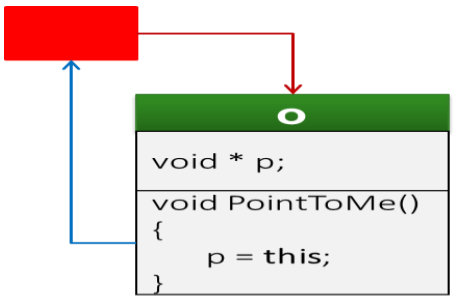
```
//Định nghĩa lớp Circle trong ví dụ phần 3.1
void main()
{
	Circle* pc = new Circle;	//Khai báo và khởi tạo đối tượng kiểu con trỏ
	Pc->setRadius(10);
	cout << "Dien tich duong tron ban kinh r = " << pc->getRadius()
		<< " la " << pc->area() << endl;
}
```

Nếu con trỏ chứa địa chỉ phần tử đầu tiên của mảng, có thể dùng con trỏ như tên mảng:

```
Circle arr[5];  //Khai báo mảng các đối thượng
Circle* pc = new Circle;  //Khai báo và sử dụng đối tượng kiểu con trỏ
pc = arr;  //Con trỏ pc tham chiếu đến mảng arr
```

3.5. Con trỏ this (tự tham chiếu)

Từ khóa this được dùng trong khi định nghĩa các hàm thành viên dùng để trỏ đến đối tượng hiện tại. Nó ít khi được sử dụng tường minh, mà thường được ngầm sử dụng khi truy cập vào các thành phần dữ liệu.



Hình 3.2. Con trỏ this

Ví dụ:

Không sử dụng con trỏ this	Sử dụng con trỏ this

`void setRadius(float r)`
`{`
`r = r; //Nhập nhằng`
`}`
	`void setRadius(float r)`
	`{`
	`this->r = r;`
	`}`

Cho phép truy cập vào đối tượng hiện tại của lớp nhằm xóa đi sự nhập nhằng giữa biến cục bộ, tham số với thành phần dữ liệu lớp. Không dùng bên trong các khối lệnh static.

3.6. Hàm khởi tạo, Hàm hủy (constructor, destructor)

3.6.1. Hàm khởi tạo

Hàm khởi tạo là hàm thành phần đặc biệt của lớp làm nhiệm vụ tạo đối tượng mới. Trình biên dịch sẽ cấp phát bộ nhớ cho đối tượng, sau đó sẽ gọi đến hàm khởi tạo. Hàm khởi tạo sẽ khởi gán giá trị cho các thuộc tính của đối tượng. Hàm khởi tạo có các đặc điểm sau:

- Tên hàm khởi tạo trùng với tên của lớp.
- Hàm khởi tạo không có kiểu dữ liệu trả về (kể cả void).
- Hàm khởi tạo phải được khai báo trong vùng public.
- Hàm khởi tạo có thể có đối số hoặc không có đối số.
- Trong một lớp có thể có nhiều hàm khởi tạo (cùng tên nhưng khác đối số).

Ví dụ:

```
#include <iostream>       //Thư viện nhập/xuất trong C++
using namespace std;      //Khai báo không gian tên mặc định
class Circle
{
private:
     const float PI = 3.1415; //Hằng số PI
     float r;  //Bán kính, thành phần dữ liệu của từng đối tượng
public:
     Circle()  //Hàm khởi tạo không có tham số
     {
          this->r = 1.0;
     }
     Circle(float r) //Hàm khởi tạo có tham số
     {
          this->r = r;
     }
     void setRadius(float r)
     {
          this->r = r;
     }
```

```
        float getRadius()
        {
            return r;
        }
        float area()
        {
            return PI * r * r;
        }
};

void main()
{
        Circle* pc1 = new Circle();    //Khai báo đối tượng không tham số
        Circle* pc2 = new Circle(10);  //Khai báo đối tượng có tham số

        cout << "Dien tich duong tron ban kinh r = " << pc1->getRadius()
            << " la " << pc1->area() << endl;
        cout << "Dien tich duong tron ban kinh r = " << pc2->getRadius()
            << " la " << pc2->area() << endl;
}
```

Kết quả:

```
Dien tich duong tron ban kinh r = 1 la 3.1415
Dien tich duong tron ban kinh r = 10 la 314.15
```

Nếu lớp không có hàm khởi tạo, trình biên dịch sẽ cung cấp một hàm khởi tạo ngầm định không đối, hàm này sẽ gán giá trị mặc định cho các tham số.

Ví dụ:

```
#include <iostream>  //Thư viện nhập/xuất trong C++
using namespace std; //Khai báo không gian tên mặc định

class Circle
{
private:
        const float PI = 3.1415; //Hằng số PI
        float r;  //Bán kính, thành phần dữ liệu của từng đối tượng
public:
        void setRadius(float r)
        {
            this->r = r;
        }
        float getRadius()
        {
            return r;
        }
```

```
        float area()
        {
            return PI * r * r;
        }
};

void main()
{
        Circle* pc = new Circle(); //Khai báo đối tượng dùng hàm tạo ngầm định

        cout << "Dien tich duong tron ban kinh r = " << pc->getRadius()
            << " la " << pc->area() << endl;
}
```

Kết quả:

```
Dien tich duong tron ban kinh r = 0 la 0
```

Nếu trong lớp đã có ít nhất một hàm khởi tạo, thì hàm khởi tạo ngầm định sẽ không còn nữa. Khi đó nếu khởi tạo đối tượng bằng hàm khởi tạo ngầm định thì trình biên dịch sẽ báo lỗi. Điều này thường xảy ra khi không xây dựng hàm khởi tạo không tham số nhưng lại sử dụng nó.

Ví dụ:

```
class Circle
{
private:
        const float PI = 3.1415; //Hằng số PI
        float r;  //Bán kính, thành phần dữ liệu của từng đối tượng

public:
        Circle(float r) //Có hàm khởi tạo, nhưng là hàm có tham số
        {
            this->r = r;
        }
};

void main()
{
        Circle c;  //Trình biên dịch sẽ báo lỗi
        Circle* pc = new Circle(); //Dùng hàm khởi tạo không tham số
                                   //Trình biên dịch sẽ báo lỗi
}
```

3.6.2. Hàm khởi tạo sao chép

Có thể khai báo và khởi tạo một đối tượng mới từ một đối tượng đã tồn tại.

```
Circle c1(10);  //Khai báo và khởi tạo đối tượng c1
Circle c2(c1);  //Khai báo đối tượng c2 dùng đối tượng c1
```

Nếu trong lớp chưa xây dựng hàm khởi tạo sao chép, thì câu lệnh Circle c2 (c1); sẽ gọi tới một hàm khởi tạo sao chép mặc định của C++. Hàm này sẽ sao chép nội dung từng bit của c1 vào các bit tương ứng của c2.

```
#include <iostream>       //Thư viện nhập/xuất trong C++
using namespace std;      //Khai báo không gian tên mặc định

class Circle
{
private:
     const float PI = 3.1415; //Hằng số PI
     float r;  //Bán kính, thành phần dữ liệu của từng đối tượng

public:
     Circle(float r)
     {
          this->r = r;
     }
     void setRadius(float r)
     {
          this->r = r;
     }
     float getRadius()
     {
          return r;
     }
     float area()
     {
          return PI * r * r;
     }
};

void main()
{
     Circle c1(10); //Khai báo và khởi tạo đối tượng c1
     Circle c2(c1);  //Khai báo đối tượng c2 dùng đối tượng c2
     cout << "Dien tich duong tron ban kinh r = " << c2.getRadius()
          << " la " << c2.area() << endl;
}
```

Kết quả:

```
Dien tich duong tron ban kinh r = 10 la 314.15
```

Nếu trong lớp Circle đã có hàm khởi tạo sao chép thì câu lệnh Circle c2 (c1); sẽ tạo ra đối tượng mới c2, sau đó gọi tới hàm tạo sao chép để khởi gán c2 theo c1.

Hàm khởi tạo sao chép sử dụng một đối kiểu tham chiếu đối tượng để khởi tạo giá trị cho đối tượng mới.

Cú pháp:

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

Ví dụ:

```
#include <iostream>       //Thư viện nhập/xuất trong C++
using namespace std;      //Khai báo không gian tên mặc định

class Circle
{
private:
     const float PI = 3.1415; //Hằng số PI
     float r;  //Bán kính, thành phần dữ liệu của từng đối tượng

public:
     Circle(float r)
     {
          this->r = r;
     }
     Circle(const Circle &c)
     {
          this->r = c.r;
     }

     void setRadius(float r)
     {
          this->r = r;
     }
     float getRadius()
     {
          return r;
     }
```

```
        float area()
        {
            return PI * r * r;
        }
};

void main()
{
        Circle c1(10);  //Khai báo và khởi tạo đối tượng c1
        Circle c2(c1);  //Khai báo đối tượng c2 dùng đối tượng c1
        cout << "Dien tich duong tron ban kinh r = " << c2.getRadius()
            << " la " << c2.area() << endl;
}
```

Kết quả:

```
Dien tich duong tron ban kinh r = 10 la 314.15
```

Lưu ý:

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

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

3.6.3. Hàm hủy

Hàm huỷ là một hàm thành phần của lớp, có chức năng ngược với hàm khởi tạo. Hàm huỷ được gọi trước khi giải phóng một đối tượng để giải phóng vùng nhớ trước khi đối tượng được hủy bỏ, xoá đối tượng khỏi màn hình nếu 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 phương thức, hàm.

Nếu trong lớp không định nghĩa hàm huỷ thì hàm huỷ mặc định được sử dụng (không làm gì cả).

– Mỗi lớp chỉ có một hàm hủy.

– Hàm hủy không có kiểu, không có giá trị trả về và không có đối số.

– Tên hàm huỷ cùng tên với tên lớp và có một dấu ngã ngay trước tên.

Ví dụ:

```
#include <iostream>       //Thư viện nhập/xuất trong C++
using namespace std;      //Khai báo không gian tên mặc định

class Circle
{
private:
     const float PI = 3.1415; //Hằng số PI
     static int counter;  //Đếm số đường tròn được tạo ra
     float r;   //Bán kính, thành phần dữ liệu của từng đối tượng

public:
     Circle(float r)
     {
          this->r = r;
          counter++;
     }
     static void demSoDT() //Hàm thành phần của lớp
     {
          cout << "So duong tron duoc tao ra la: " << counter << endl;
     }
     void hienThiDT()
     {
          cout << "Ban kinh DT la:" << r << endl;
     }
     ~Circle()
     {
          counter--;
          cout << "Duong tron ban kinh " << r << " da duoc huy!" << endl;
     }
};

int Circle::counter;

void main()
{
     Circle c1(5);
     Circle::demSoDT();
     c1.hienThiDT();
     Circle c2(10);
     Circle::demSoDT();
     c2.hienThiDT();
     Circle c3(15);
     Circle::demSoDT();
     c3.hienThiDT();
}
```

Kết quả:

```
So duong tron duoc tao ra la: 1
Ban kinh DT la:5
So duong tron duoc tao ra la: 2
Ban kinh DT la:10
So duong tron duoc tao ra la: 3
Ban kinh DT la:15
Duong tron ban kinh 15 da duoc huy!
Duong tron ban kinh 10 da duoc huy!
Duong tron ban kinh 5 da duoc huy!
```

3.6.4. Hàm bạn (friend function)

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.

Hàm bạn là loại hàm không phải là hàm thành viên của một lớp, nhưng có thể truy nhập vào các thành phần riêng tư của lớp đó. Hàm bạn có thể là hàm tự do, hoặc là hàm thành viên của lớp khác.

Để một hàm trở thành bạn của một lớp, có 2 các viết:

Cách 1: Dùng từ khóa friend để khai báo hàm trong lớp và định nghĩa hàm bên ngoài như các hàm thông thường (không dùng từ khóa friend).

```
class Tên_lớp
{
private: //Khai báo các thuộc tính
public:
     ...
     //Khai báo các hàm bạn của lớp Tên_lớp
     friend void hàm1(Các_tham_số);
     friend int hàm2(Các_tham_số);
     ...
};
//Định nghĩa các hàm bạn
void hàm1(Các_tham_số)
{
     ...
}
int hàm2(Các_tham_số)
{
     ...
}
```

Cách 2: Dùng từ khóa friend để xây dựng hàm trong định nghĩa lớp:

```
class Tên_lớp
{
private:
     //Khai báo các thuộc tính
public:
     ...
     //Khai báo các hàm bạn của lớp Tên_Lớp
     friend void hàm1(Các_tham_số)
     {
          ...
     }
     friend int hàm2(Các_tham_số)
     {
          ...
     }
};
```

Lưu ý:

– *Hàm bạn không phải là hàm thành phần của lớp. Việc truy cập tới hàm bạn được thực hiện như hàm thông thường. Trong thân hàm bạn của một lớp có thể truy cập 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 cập tới tất cả các thuộc tính của các đối tượng trong các lớp này. Để làm cho hàm func trở thành bạn của các lớp A, B và C ta viết như sau:*

```
//Trước hết, phải khai báo các lớp
class A; //Khai báo lớp A
class B; //Khai báo lớp B
//Định nghĩa lớp A
class A
{
     //Khai báo hàm "hàmBạn" là bạn của A
     friend void hàmBạn(Các_tham_số);
};
//Định nghĩa lớp B
class B
{
     //Khai báo hàm "hàmBạn" là bạn của B
     friend void hàmBạn(Các_tham_số);
};
```

```
//Xây dựng hàm hàmBạn
void hàmBạn(Các_tham_số)
{
      //Thân hàm
};
```

Ví dụ: Cộng các thành viên của hai lớp khác nhau sử dụng hàm bạn

```
#include <iostream>   //Thư viện nhập/xuất trong C++
using namespace std;  //Khai báo không gian tên mặc định

//Trước hết, phải khai báo các lớp
class A; //Khai báo lớp A
class B; //Khai báo lớp B

//Định nghĩa lớp A
class A
{
private:
      int numA;
public:
      A(int a)
      {
            numA = a;
      }
       //Khai báo hàm cong là bạn của A
      friend int cong(A, B);
};

//Định nghĩa lớp B
class B
{
private:
      int numB;
public:
      B(int b)
      {
            numB = b;
      }
      //Khai báo hàm cong là bạn của B
      friend int cong(A, B);
};

//Hàm cong() là hàm bạn của lớp A và B, có thể truy cập thành viên numA và numB
int cong(A a, B b)
{
   return (a.numA + b.numB);
}
```

```
void main()
{
    A a(10);
    B b(20);
    cout<<"Tong: "<< cong(a, b);
}
```

Kết quả:

```
Tong: 30
```

3.7. Các thành phần tĩnh (static)

3.7.1. Thành phần dữ liệu tĩnh

Thành phần dữ liệu tĩnh được khai báo bằng từ khoá static và được cấp phát một vùng nhớ cố định, nó tồn tại ngay cả khi lớp chưa có một đối tượng nào cả. Dữ liệu tĩnh là thành phần chung cho cả lớp, không của riêng từng đối tượng.

Cú pháp:

```
class A
{
private:
     static int x; //Thành phần dữ liệu tĩnh
     int y;
     ...
};
A a, b; //Khai báo 2 biến a, b
```

Giữa 2 thành phần dữ liệu x và y có sự khác nhau: a.y và b.y có 2 vùng nhớ khác nhau, trong khi a.x và b.x chỉ là một (chung một vùng nhớ), thành phần x tồn tại ngay khi a và b chưa được khai báo. Để truy cập thành phần tĩnh, ta có thể dùng tên lớp:

```
A::x;
```

Khai báo và khởi tạo giá trị cho thành phần tĩnh:

Thành phần tĩnh sẽ được cấp phát bộ nhớ và khởi tạo giá trị đầu bằng một câu lệnh khai báo đặt sau định nghĩa lớp theo cú pháp sau:

```
int A::x;       //Khởi tạo cho x giá trị 0
int A::x = 10; //Khởi tạo cho x giá trị 10
```

Ví dụ: Khai báo một biến static int dem, biến này dùng để đếm số lượng các hình chủ nhật đã được tạo ra.

```
#include <iostream>       //Thư viện nhập/xuất trong C++
using namespace std;      //Khai báo không gian tên mặc định

class Rectangle
{
private:
     int dai;
     int rong;
     static int dem;
public:
     void set(int d, int r)
     {
          dai = d;
          rong = r;
     }
     Rectangle(int x, int y)
     {
          dem++;
          set(x, y);
          cout << "So hinh CN duoc tao ra la: " << dem << endl;
     }
};

int Rectangle::dem = 0;

int main()
{
     Rectangle r1(1, 2);
     Rectangle r2(3, 4);
}
```

3.7.2. Hàm thành phần tĩnh (static)

Hàm static có vai trò cũng như biến static. Nghĩa là khi đã khai báo class nhưng chưa tạo ra đối tượng Rectangle r1(1, 2);, thì chúng ta vẫn truy cập được biến "dem" trong ví dụ trên.

Cú pháp:

```
//Hàm thành phần tĩnh, thành phần của lớp

static Kiểu_dữ_liệu Tên_hàm(Các_tham_số);
```

Lời gọi hàm thành phần tĩnh như sau:

```
Tên_lớp::Tên_hàm_thành_phần_tĩnh(Các_tham_số);
```

Ví dụ:

```
#include <iostream>       //Thư viện nhập/xuất trong C++
using namespace std;      //Khai báo không gian tên mặc định

class Rectangle
{
private:
     int dai;
     int rong;
     static int dem;
public:
     void set(int d, int r)
     {
          dai = d;
          rong = r;
     }
     Rectangle(int x, int y)
     {
          dem++;
          set(x, y);
     }
     static void demSoCN() //Hàm thành phần tĩnh, thành phần của lớp
     {
          cout << "So hinh CN duoc tao ra la: " << dem << endl;
     }

};
int Rectangle::dem = 0;
int main()
{
     Rectangle r1(1, 2);
     Rectangle::demSoCN(); //Gọi hàm thành phần tĩnh
     Rectangle r2(3, 4);
     Rectangle::demSoCN(); `
}
```

Kết quả:

```
So hinh CN duoc tao ra la: 1
So hinh CN duoc tao ra la: 2
```

Lưu ý:

– *Từ khoá static đặt trước định nghĩa hàm thành phần viết bên trong định nghĩa lớp. Nếu hàm thành phần viết bên ngoài định nghĩa lớp, thì dùng từ khoá static*

đặt trước khai báo hàm thành phần bên trong định nghĩa lớp. Không được phép dùng từ khoá static đặt trước định nghĩa hàm thành phần viết bên ngoài định nghĩa lớp.

– *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 được tạo ra.*

– *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 không phải thuộc tính tĩnh của lớp trong thân hàm thành phần tĩnh.*

BÀI TẬP CHƯƠNG 3

Bài 1: Hãy thiết kế lớp Rectangle, biểu diễn hình chữ nhật bao gồm:

- Hai thành phần dữ liệu width và height có kiểu là double xác định chiều rộng và chiều cao của hình chữ nhật, gán giá trị mặc định là 1.0 cho cả hai.
- Các phương thức set, get cho các thành phần dữ liệu.
- Phương thức khởi tạo mặc định hình chữ nhật (constructor).
- Phương thức khởi tạo hình chữ nhật với 2 tham số width và height.
- Phương thức hủy.
- Phương thức getArea() trả về diện tích hình chữ nhật.
- Phương thức getPerimeter() trả về chu vi hình chữ nhật.

Viết chương trình tạo 2 đối tượng Rectangle, một với width = 15 và height = 40, một với width = 12.5 và height = 25.2. Hiển thị chiều rộng (width), chiều dài (height), diện tích (area), và chu vi (perimeter) của mỗi hình chữ nhật.

Bài 2: Xây dựng lớp đa thức với các hệ số là các số thực với các phương thức sau:

- Phương thức khởi tạo mặc định gán đa thức bằng 0.
- Phương thức khởi tạo có tham số là một mảng hệ số và bậc của đa thức.
- Phương thức khởi tạo hủy.
- Hàm nhập/xuất đa thức.
- Các hàm cộng, trừ, nhân hai đa thức.
- Các hàm gán =, so sánh ==, !=
- Hàm tính giá trị của đa thức với giá trị biến là tham số của hàm.

Sử dụng lớp vừa cài đặt, nhập vào 2 đa thức và minh họa tất cả các phương thức đã xây dựng.

Bài 3: Xây dựng lớp phân số với các hệ số là các số nguyên với các phương thức sau:

- Phương thức khởi tạo mặc định gán tử số bằng 0 và mẫu số bằng 1.
- Phương thức khởi tạo có tham số là hai số nguyên cho tử số và mẫu số.

- Phương thức khởi tạo hủy.
- Hàm nhập/xuất phân số.
- Các hàm cộng, trừ, nhân, chia hai phân số (lưu ý rút gọn phân số).
- Các hàm gán =, so sánh ==, !=
- Hàm rút gọn phân số.

Sử dụng lớp vừa cài đặt, nhập vào 2 phân số và minh họa tất cả các phương thức đã xây dựng.

Bài 4: Xây dựng lớp ma trận có tên là Matrix cho các ma trận vuông, các hàm thành phần bao gồm:

- Phương thức khởi tạo mặc định.
- Phương thức hủy.
- Hàm nhập xuất ma trận.
- Các hàm cộng, trừ, nhân, gán trên ma trận.
- Hàm tính định thức và hàm tính ma trận nghịch đảo ma trận.

Viết chương trình tạo 2 đối tượng ma trận. Nhập, xuất 2 ma trận vừa tạo, cộng trừ, nhân 2 ma trận trên, tính định thức, ma trận nghịch đảo cho ma trận và xuất kết quả ra màn hình.

Bài 5: Ứng dụng hàm bạn và lớp bạn trong C++: Nhân Ma trận với Vector:

- Định nghĩa lớp Vector trong không gian có số chiều bất kì. Xây dựng các hàm thực hiện tính cộng trừ nhân (nhân vector với một hệ số k, tích vô hướng của 2 vector) và các phương thức cần thiết.
- Định nghĩa lớp Matrix trong không gian có kích thước bất kì. Xây dựng các hàm thực hiện tính cộng, nhân ma trận với một số k, nhân ma trận với ma trận) và các phương thức cần thiết.
- Sử dụng hàm bạn để xây dựng hàm nhân ma trận với vector.

Hướng dẫn:

Vector.h

```
#pragma once
class Matrix;
class Vector
{
private:
	double* coords;
	int n;
public:

	Vector();
	Vector(int N, double x); //tạo vector có N chiều, mỗi ô có giá trị x
	Vector(const Vector& a);
	~Vector();
	void nhap();
	void xuat();
	int cong(const Vector& a); //return 1 nếu cộng được
	void nhanK(const double& k);
	int tru(Vector a); //return 1 nếu trừ được
	double tichVoHuong(const Vector& a);
	friend Vector multiply(const Matrix& a, const Vector& b);
};
```

Matrix.h

```
#pragma once
class Vector;
class Matrix
{
private:
	int m; //dòng
	int n; //cột
	double** elements;
public:
	Matrix();
	~Matrix();
	Matrix(const Matrix& a);
	void nhap();
	void xuat();
	int cong(const Matrix& a); //return 1 nếu cộng được
	void nhanK(const double& k);//nhân với 1 số k
	int nhan(const Matrix& a);//nhân với 1 CMatrix, return 1 nếu nhân được
	friend Vector multiply(const Matrix& a, const Vector& b);
};
```

CHƯƠNG 4 – NẠP CHỒNG TOÁN TỬ (OPERATOR OVERLOADING)

Các nội dung chính

- *Khái niệm về nạp chồng toán tử*
- *Nạp chồng toán tử một ngôi*
- *Nạp chồng toán tử hai ngôi*
- *Nạp chồng các toán tử khác*
  - *Toán tử ()*
  - *Toán tử []*
  - *Nạp chồng toán tử chuyển đổi kiểu*
  - *Nạp chồng toán tử new và delete*
  - *Nạp chồng toán tử >>/<<*

4.1. Khái niệm về nạp chồng toán tử (operator overloading)

Nạp chồng toán tử, được dùng để định nghĩa toán tử có sẵn trong C++ phục vụ cho kiểu dữ liệu (class) do người lập trình tự định nghĩa, nhằm tạo ra toán tử cùng tên thực hiện các chức năng trên các lớp khác nhau.

Nạp chồng toán tử trong C++ là các hàm với tên đặc biệt: tên hàm là từ khóa operator theo sau là ký hiệu của toán tử đang được định nghĩa. Giống như bất kỳ hàm khác, một toán tử nạp chồng có một kiểu trả về và một danh sách tham số.

Cú pháp:

```
Kiểu_trả_về operator <toán tử> (danh_sách_đối_số)
{
     //Định nghĩa toán tử
}
```

Trong đó:

- Kiểu_trả_về là kiểu dữ liệu kết quả thực hiện của toán tử.
- <toán tử> là tên toán tử cần nạp chồng.

– operator <toán tử> (danh_sách_đối_số) gọi là hàm nạp chồng toán tử, nó có thể là hàm thành phần hoặc là hàm bạn, nhưng không thể là hàm tĩnh.

Danh sách các tham số được khai báo tương tự khai báo biến nhưng phải tuân theo những quy định sau:

– Nếu hàm nạp chồng toán tử là hàm thành phần thì không có đối số cho toán tử một ngôi và một đối số cho toán tử hai ngôi, toán tử có đối số đầu tiên (không tường minh) là con trỏ this.

Ví dụ:

```
Diem operator++();              //Toán tử ++: toán tử 1 ngôi
PhanSo operator+( PhanSo ps);   //Toán tử +: toán tử 2 ngôi
```

– Nếu hàm nạp chồng toán tử là hàm bạn thì có một đối số cho toán tử một ngôi và hai đối số cho toán tử hai ngôi.

Ví dụ:

```
friend PhanSo operator+(PhanSo a, PhanSo b);
friend ostream& operator << (ostream& os, PhanSo a);
```

Lưu ý:

– *Trong C++ ta có thể nạp chồng cho hầu hết các toán tử:*

Bảng 4.1. Danh sách các toán tử có thể nạp chồng

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

– *Nhưng ngoại trừ những toán tử sau đây:*

- *Toán tử xác định thành phần của lớp (‘.’, ‘->’).*
- *Toán tử phân giải miền xác định (‘::’).*

- *Toán tử điều kiện ('? :').*
- *Toán tử lấy kích thước ('sizeof').*
- *Toán tử lấy kiểu dữ liệu ('typeid').*

Lưu ý:

– *Toán tử được mở rộng (trên kiểu dữ liệu đang được định nghĩa) nhưng cú pháp, số toán hạng, quyền ưu tiên và thứ tự kết hợp thực hiện của các toán tử vẫn không có gì thay đổi.*

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

– *Các toán tử = , ( ) , [ ] yêu cầu hàm nạp chồng toán tử phải là hàm thành phần của lớp, không thể dùng hàm bạn.*

– *Toán tử phải thể hiện đúng những gì người dùng muốn định nghĩa (ví dụ tránh trường hợp nạp chồng toán tử + nhưng lại thực hiện phép * bên trong).*

– *Toán tử khi áp dụng phải đúng logic (ví dụ việc tăng thời gian (giờ, phút, giây) bằng cách + hoặc - cho giây là hợp lý nhưng * hoặc / là không đúng logic).*

– *Khi nạp chồng cho một toán tử thì nên lưu ý đến những toán tử có cùng tính chất với nó, ví dụ nếu ta nạp chồng cho toán tử > thì phải nạp chồng các toán tử như <, ==, >=, <= vì chúng cùng một tính chất là dùng để so sánh.*

4.2. Nạp chồng toán tử một ngôi

Các toán tử một ngôi là các toán tử chỉ có một toán hạng tham gia vào phép toán, như toán tử tăng (++) và toán tử giảm (--).

Ví dụ:

```
// increment counter variable with ++ operator
#include <iostream>
using namespace std;

class Counter
{
private:
     unsigned int count;                  //count
```

```
    public:
        Counter() : count(0)                    //constructor
        { }
        unsigned int getCount()                 //return count
        {
            return count;
        }
        void operator ++ ()                     //increment (prefix)
        {
            count = count + 1;
        }
    };
    int main()
    {
        Counter c1, c2;                         //define and initialize
        cout << "c1 = " << c1.getCount();       //display
        cout << "\tc2 = " << c2.getCount() << endl;
        ++c1;                                   //increment c1
        ++c2;                                   //increment c2
        ++c2;                                   //increment c2
        cout << "c1 = " << c1.getCount();       //display again
        cout << "\tc2 = " << c2.getCount() << endl;
    }
```

Kết quả:

```
c1 = 0  c2 = 0
c1 = 1  c2 = 2
```

Trong ví dụ trên, hàm operator++() trả về void, chúng ta có thể nạp chồng toán tử trả về một đối tượng thuộc lớp.

Ví dụ:

```
    // increment counter variable with ++ operator, return value
    #include <iostream>
    using namespace std;

    class Counter
    {
    private:
        unsigned int count;                     //count
    public:
        Counter() : count(0)                    //constructor
        { }
        unsigned int getCount()                 //return count
        {
            return count;
        }
```

```
        Counter operator ++ ()                  //increment count
        {
              count = count + 1;                //increment count
              Counter temp;                     //make a temporary Counter
              temp.count = count;               //give it same value as this obj
              return temp;                      //return the copy
        }
};

int main()
{
        Counter c1, c2;                         //c1=0, c2=0
        cout << "c1 = " << c1.getCount();       //display
        cout << "\tc2 = " << c2.getCount() << endl;
        ++c1;                                   //c1 = 1
        c2 = ++c1;                              //c1 = 2, c2 = 2
        cout << "c1 = " << c1.getCount();       //display again
        cout << "\tc2 = " << c2.getCount() << endl;
}
```

Kết quả:

```
c1 = 0  c2 = 0
c1 = 2  c2 = 2
```

Trong ví dụ trên, có sử dụng một đối tượng temp để cung cấp giá trị trả về cho đối tượng.

```
Counter operator ++ ()                  //increment count
{
      count = count + 1;                //increment count
      Counter temp;                     //make a temporary Counter
      temp.count = count;               //give it same value as this obj
      return temp;                      //return the copy
}
```

Tuy nhiên, chúng ta cũng có thể sử dụng đối tượng tạm không có tên, như sau:

```
Counter operator ++ () //increment count
{
      count = count + 1; //increment count
      return Counter(count);
}
```

Lưu ý: Phải khai báo phương thức khởi tạo có tham số:

```
Counter(int c) : count(c) //constructor, one arg
{ }
```

Trong các ví dụ trên, hàm oprator++() sẽ nạp chồng toán tử tiền tố (prefix) Trong trường hợp muốn nạp chồng các toán tử hậu tố (postfix), 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>
using namespace std;
class Diem
{
private:
	int x, y;
public:
	Diem() : x(0), y(0)
	{ }
	Diem(int x, int y) :
	{
		this->x = x;
		this->y = y;
	}
	Diem& operator ++(); //nạp chồng toán tử ++ tiền tố
	Diem operator ++(int);    //nạp chồng toán tử ++ hậu tố
	Diem& operator --(); //nạp chồng toán tử -- tiền tố
	Diem operator --(int);    //nạp chồng toán tử -- hậu tố

	void show()
	{
		cout << "(" << x << "," << 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;
}
int main()
{
     Diem d1(5,10), d2(15,20), d3(25,30), d4(35,40);
     cout << "Diem d1 : ";
     d1.show();
     ++d1;
     cout << " -> ++ tien to -> ";
     d1.show();
     cout << "\nDiem d2 : ";
     d2.show();
     d2++;
     cout << "-> ++ hau to -> ";
     d2.show();
     cout << "\nDiem d3 : ";
     d3.show();
     --d3;
     cout << "-> -- tien to -> ";
     d3.show();
     cout << "\nDiem d4 : ";
     d4.show();
     d4--;
     cout << "-> -- hau to -> ";
     d4.show();
}
```

Kết quả:

```
Diem d1 : (5,10) -> ++ tien to -> (6,11)
Diem d2 : (15,20)-> ++ hau to -> (16,21)
Diem d3 : (25,30)-> -- tien to -> (24,29)
Diem d4 : (35,40)-> -- hau to -> (34,39)
```

4.3. Nạp chồng toán tử hai ngôi

Các toán tử hai ngôi trong C++ nhận hai tham số. Chúng ta sử dụng toán tử hai ngôi khá thường xuyên, ví dụ như toán tử cộng (+), toán tử trừ (-) và toán tử chia (/).

Ví dụ: Nạp chồng toán tử cộng (+). Tương tự, có thể nạp chồng toán tử trừ (-) và toán tử chia (/), toán tử chia lấy phần dư (%),...

```
//Chuong trinh minh hoa nap chong toan tu hai ngoi
#include <iostream>
using namespace std;
class Box
{
      double chieudai;      // Chieu dai cua mot box
      double chieurong;     // Chieu rong cua mot box
      double chieucao;      // Chieu cao cua mot box
public:
      double tinhTheTich(void)
      {
            return chieudai * chieurong * chieucao;
      }
      void setChieuDai(double dai)
      {
            chieudai = dai;
      }

      void setChieuRong(double rong)
      {
            chieurong = rong;
      }

      void setChieuCao(double cao)
      {
            chieucao = cao;
      }
      // Nap chong toan tu + de cong hai doi tuong Box.
      Box operator+(const Box& b)
      {
            Box box;
            box.chieudai = this->chieudai + b.chieudai;
            box.chieurong = this->chieurong + b.chieurong;
            box.chieucao = this->chieucao + b.chieucao;
            return box;
      }
};

void main()
{
```

```
        Box box1;                // Khai bao box1 la cua kieu Box
        Box box2;                // Khai bao box2 la cua kieu Box
        Box box3;                // Khai bao box3 la cua kieu Box
        double thetich = 0.0; // Luu giu the tich cua mot box tai day
        // thong tin chi tiet cua box 1
        box1.setChieuDai(1.0);
        box1.setChieuRong(2.0);
        box1.setChieuCao(1.0);

        // thong tin chi tiet cua box 2
        box2.setChieuDai(2.0);
        box2.setChieuRong(2.0);
        box2.setChieuCao(2.0);

        // the tich cua box 1
        thetich = box1.tinhTheTich();
        cout << "The tich cua box1 : " << thetich << endl;

        // the tich cua box 2
        thetich = box2.tinhTheTich();
        cout << "The tich cua box2 : " << thetich << endl;

        // Cong hai doi tuong:
        box3 = box1 + box2;

        // the tich cua box 3
        thetich = box3.tinhTheTich();
        cout << "The tich cua box3 : " << thetich << endl;
    }
```

Kết quả:

```
The tich cua box1 : 2
The tich cua box2 : 8
The tich cua box3 : 36
```

Ví dụ:

```
    // overloaded '+' operator adds two Distances
    #include <iostream>
    using namespace std;

    class Distance //English Distance class
    {
    private:
        int feet;
        float inches;
    public: //constructor (no args)
```

```
        Distance() : feet(0), inches(0.0)
        { } //constructor (two args)
        Distance(int ft, float in) : feet(ft), inches(in)
        { }
        void getdist() //get length from user
        {
            cout << "Enter feet : "; cin >> feet;
            cout << "Enter inches : "; cin >> inches;
        }
        void showdist() const //display distance
        {
            cout << feet << "\' - " << inches << '\"';
        }
        Distance operator + (Distance) const; //add 2 distances
    };

    //add this distance to d2
    Distance Distance::operator + (Distance d2) const //return sum
    {
        int f = feet + d2.feet; //add the feet
        float i = inches + d2.inches; //add the inches
        if (i >= 12.0) //if total exceeds 12.0,
        { //then decrease inches
            i -= 12.0; //by 12.0 and
            f++; //increase feet by 1
        } //return a temporary Distance
        return Distance(f, i); //initialized to sum
    }

    void main()
    {
        Distance dist1, dist3, dist4; //define distances
        dist1.getdist(); //get dist1 from user
        Distance dist2(11, 6.25); //define, initialize dist2
        dist3 = dist1 + dist2; //single '+' operator
        dist4 = dist1 + dist2 + dist3; //multiple '+' operators
                                       //display all lengths
        cout << "dist1 = "; dist1.showdist(); cout << endl;
        cout << "dist2 = "; dist2.showdist(); cout << endl;
        cout << "dist3 = "; dist3.showdist(); cout << endl;
        cout << "dist4 = "; dist4.showdist(); cout << endl;
    }
```

Kết quả:

```
Enter feet : 5
Enter inches : 3
dist1 = 5' - 3"
```

```
dist2 = 11' - 6.25"
dist3 = 16' - 9.25"
dist4 = 33' - 6.5"
```

4.4. Nạp chồng các toán tử khác

4.4.1. Toán tử ()

Toán tử () trong C++ có thể được nạp chồng cho các đối tượng của kiểu lớp. Khi nạp chồng (), chúng ta không tạo một cách mới để gọi một hàm. Đúng hơn là, đang tạo một hàm toán tử mà có thể được truyền số tham số tùy ý.

```
//Chuong trinh minh hoa nap chong toan tu ()
#include <iostream>
using namespace std;

class KhoangCach
{
private:
	int met;
	int centimet;
public:
	// phan khai bao cac constructor can thiet
	KhoangCach()
	{
		met = 0;
		centimet = 0;
	}
	KhoangCach(int m, int c)
	{
		met = m;
		centimet = c;
	}

	// nap chong toan tu goi ham ()
	KhoangCach operator()(int x, int y, int z)
	{
		KhoangCach K;
		// bay gio, dat phep tinh bat ky
		K.met = x + y + 5;
		K.centimet = y - z + 20;
		return K;
	}
	// Phuong thuc de hien thi khoang cach
	void hienthiKC()
	{
```

```
            cout << "\nDo dai bang m la: " << met <<
                        "\nVa do dai bang cm la: " << centimet << endl;
        }
    };
    void main()
    {
        KhoangCach K1(24, 36), K2;

        cout << "Khoang cach dau tien la: ";
        K1.hienthiKC();

        K2 = K1(15, 15, 15); // trieu hoi toan tu ()
        cout << "\n-------------------------\n";
        cout << "Khoang cach thu hai la: ";
        K2.hienthiKC();
    }
```

Kết quả:

```
Khoang cach dau tien la:
Do dai bang m la: 24
Va do dai bang cm la: 36
```

Ví dụ sau đây cung cấp hai hàm để truy cập các phần tử mảng: putel () để chèn một giá trị vào mảng và getel () để tìm giá trị của một phần tử mảng. Cả hai hàm đều kiểm tra giá trị chỉ số (index) được cung cấp để đảm bảo nó không nằm ngoài giới hạn.

```
    // creates safe array (index values are checked before access)
    // uses separate put and get functions
    #include <iostream>
    using namespace std;
    #include <process.h> // for exit()
    const int LIMIT = 100;

    class safearay
    {
    private:
        int arr[LIMIT];

    public:
        void putel(int n, int elvalue) //set value of element
        {
            if (n< 0 || n >= LIMIT)
            {
                cout << "\nIndex out of bounds"; exit(1);
            }
            arr[n] = elvalue;
```

```
        }
        int getel(int n) const //get value of element
        {
            if (n< 0 || n >= LIMIT)
            {
                cout << "\nIndex out of bounds"; exit(1);
            }
            return arr[n];
        }
    };
    void main()
    {
        safearay sa1;
        int j;
        for (j = 0; j<LIMIT; j++) // insert elements
            sa1.putel(j, j * 10);
        for (j = 0; j<LIMIT; j++) // display elements
        {
            int temp = sa1.getel(j);
            cout << "Element " << j << " is " << temp << endl;
        }
        system("pause");
    }
```

Kết quả:

```
Element 0 is 0
Element 1 is 10
Element 2 is 20
…
Element 98 is 980
Element 99 is 990
```

4.4.2. Toán tử []

Toán tử [] thông thường được sử dụng để truy xuất các phần tử của mảng. Tuy nhiên, toán tử này có thể được nạp chồng. Điều này rất hữu ích nếu bạn muốn sửa đổi cách thức hoạt động của mảng trong C ++. Ví dụ: bạn có thể muốn tạo một mảng an toàn: mảng tự động kiểm tra các chỉ số (index) bạn sử dụng để truy cập vào mảng, để đảm bảo rằng chúng không nằm ngoài giới hạn.

Ví dụ:

```
    //Chuong trinh minh hoa nap chong toan tu []
    #include <iostream>
    using namespace std;
```

```
const int KICHCO = 15;
class ViDuMang
{
private:
	int mang[KICHCO];
public:
	ViDuMang()
	{
		register int i;
		for (i = 0; i < KICHCO; i++)
			mang[i] = i;
	}
	int &operator[](int i)
	{
		if (i > KICHCO)
		{
			cout << "======================" << endl;
			cout << "Chi muc vuot gioi han!" << endl;
			// Tra ve phan tu dau tien.
			return mang[0];
		}
		return mang[i];
	}
};
void main()
{
	ViDuMang V;

	cout << "Gia tri cua V[3] la: " << V[3] << endl;
	cout << "Gia tri cua V[6] la: " << V[6] << endl;
	cout << "Gia tri cua V[16] la: " << V[16] << endl;
}
```

Kết quả:

```
Gia tri cua V[3] la: 3
Gia tri cua V[6] la: 6
======================
Chi muc vuot gioi han!
Gia tri cua V[16] la: 0
```

4.4.3. Nạp chồng toán tử chuyển đổi kiểu

Ví dụ:

```
// conversions: Distance to meters, meters to Distance
#include <iostream>
```

```
using namespace std;
class Distance //English Distance class
{
private:
     const float MTF; //meters to feet
     int feet;
     float inches;
public: //constructor (no args)
     Distance() : feet(0), inches(0.0), MTF(3.280833F)
     { } //constructor (one arg)
     Distance(float meters) : MTF(3.280833F)
     { //convert meters to Distance
          float fltfeet = MTF * meters; //convert to float feet
          feet = int(fltfeet); //feet is integer part
          inches = 12 * (fltfeet - feet); //inches is what's left
     } //constructor (two args)
     Distance(int ft, float in) : feet(ft), inches(in), MTF(3.280833F)
     { }
     void getdist() //get length from user
     {
          cout << "\nEnter feet : "; cin >> feet;
          cout << "Enter inches : "; cin >> inches;
     }
     void showdist() const //display distance
     {
          cout << feet << "\' - " << inches << "\'";
     }
     operator float() const //conversion operator
     { //converts Distance to meters
          float fracfeet = inches / 12; //convert the inches
          fracfeet += static_cast<float>(feet); //add the feet
          return fracfeet / MTF; //convert to meters
     }
};

void main()
{
     float mtrs;
     Distance dist1 = 2.35F; //uses 1-arg constructor to
                                  //convert meters to Distance
     cout << "dist1 = "; dist1.showdist();
     mtrs = static_cast<float>(dist1); //uses conversion operator
                                   //for Distance to meters
     cout << "\ndist1 = " << mtrs << " meters";
     Distance dist2(5, 10.25); //uses 2-arg constructor
     mtrs = dist2; //also uses conversion op
     cout << "\ndist2 = " << mtrs << " meters\n";
```

```
        // dist2 = mtrs; //error, = won't convert
    }
```

Kết quả:

```
dist1 = 7' - 8.51949'
dist1 = 2.35 meters
dist2 = 1.78435 meters
```

4.4.4. Nạp chồng toán tử new và delete

```
    class Obj
    {
    public:
        void* operator new(size_t sz)
        {
            return malloc(sz);
        }
        void* operator new[](size_t sz)
        {
            return malloc(sz);
        }
            void operator delete(void* p)
        {
            free(p);
        }
        void operator delete[](void* p)
        {
            free(p);
        }
    };
```

4.4.5. Nạp chồng toán tử >>/<<

C++ là có thể input và output các kiểu dữ liệu có sẵn bởi sử dụng toán tử trích luồng >> và toán tử chèn luồng <<. Các toán tử trích luồng và chèn luồng cũng có thể được nạp chồng để thực hiện input và output cho các kiểu tự định nghĩa.

Ở đây, nó là quan trọng để tạo một hàm nạp chồng toán tử một friend của lớp, bởi vì nó sẽ được gọi mà không tạo một đối tượng.

Ví dụ:

```
    //Chuong trinh minh hoa nap chong toan tu >>/<<
    #include <iostream>
    using namespace std;
    class KhoangCach
    {
```

```
    private:
        int met;
        int centimet;
    public:
        // phan khai bao cac constructor can thiet
        KhoangCach() : met (0), centimet(0)
        { }
        KhoangCach(int m, int c) : met(m), centimet(c)
        { }
        friend ostream& operator<<(ostream& output, const KhoangCach& K)
        {
            output << "\nDo dai bang m la: " << K.met <<
                "\nVa do dai bang cm la: " << K.centimet;
            return output;
        }
        friend istream& operator>>(istream& input, KhoangCach& K)
        {
            input >> K.met >> K.centimet;
            return input;
        }
    };
    void main()
    {
        KhoangCach K1(5, 10), K2(15, 20), K3;
        cout << "Nhap gia tri cua doi tuong K3: ";
        cin >> K3;
        cout << "\nKhoang cach dau tien: " << K1;
        cout << "\n=========================";
        cout << "\nKhoang cach thu hai: " << K2;
        cout << "\n=========================";
        cout << "\nKhoang cach thu ba: " << K3;
    }
```

Kết quả:

```
Nhap gia tri cua doi tuong K3: 25 30

Khoang cach dau tien:
Do dai bang m la: 5
Va do dai bang cm la: 10
=========================
Khoang cach thu hai:
Do dai bang m la: 15
Va do dai bang cm la: 20
=========================
Khoang cach thu ba:
Do dai bang m la: 25
Va do dai bang cm la: 30
```

BÀI TẬP CHƯƠNG 4

Bài 1: Trong lớp đa thức (bài tập chương 3), hãy định nghĩa nạp chồng các toán tử +, -, *, =, ==, <<, >>,...

Bài 2: Trong lớp phân số (bài tập chương 3), hãy định nghĩa nạp chồngcác toán tử +, -, *, /, =, ==, +=, -=, *=, /= , <, >, <=, >=, != , ++, --, >>, <<,....

Bài 3: Trong lớp ma trận vuông (bài tập chương 3), hãy định nghĩa nạp chồngcác toán tử +, -, *, =, ==, <<, >>,....

Bài 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. Nạp chồng toán tử [], để trình biên trình dịch hiểu được phép truy nhập m[i][j], trong đó m là một đối tượng thuộc lớp Matran. Viết chương trình cho phép nhập, xuất ma trận.

CHƯƠNG 5

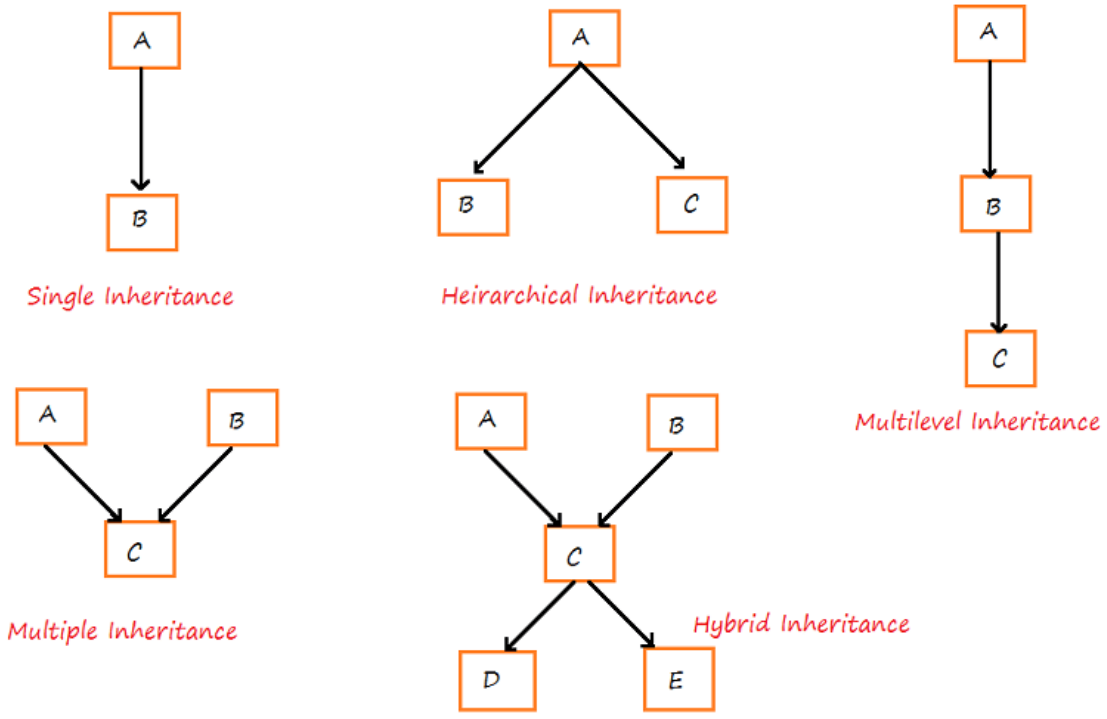
KẾ THỪA (INHERITANCE)

Các nội dung chính:

- *Giới thiệu về kế thừa*
- *Đơn kế thừa (Single Inheritance)*
- *Đa kế thừa (Multiple Inheritance)*
- *Hàm ảo (Vitual Method)*

5.1. Giới thiệu

Kế thừa là một trong những khái niệm quan trọng của phương pháp lập trình hướng đối tượng, nhằm tái sử dụng lại mã chương trình. Tính kế thừa cho phép định nghĩa các lớp mới (lớp dẫn xuất - derive class) từ các lớp đã có (lớp cơ sở - base class). 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 lớp dẫn xuất có thêm những thành phần mới. Có 2 loại kế thừa trong C++: là đơn kế thừa và đa kế thừa.

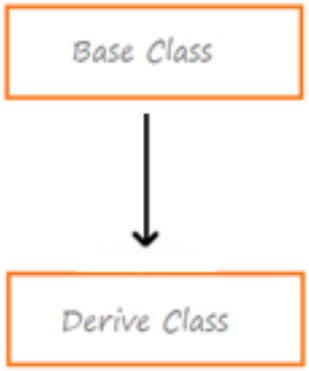


Hình 5.1. Các loại kế thừa trong C++

5.2. Đơn kế thừa (Single Inheritance)

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

Trong C ++, đơn kế thừa là một lớp dẫn xuất được kế thừa từ một và chỉ một lớp cơ sở. Do đó, hai lớp liên quan (lớp Base & lớp Derive) thể hiện mối quan hệ 1-1.



Hình 5.2. Đơn kế thừa

Cú pháp khai báo một lớp kế thừa từ một lớp khác như sau:

```
class <Tên_lớp_dẫn_xuất> : <Từ_khóa_dẫn_xuất> <Tên_lớp_cơ_sở>
{
private:
     // Khai báo các thuộc tính của lớp_dẫn_xuất
public:
     // Định nghĩa các hàm thành phần của lớp_dẫn_xuất
};
```

Trong đó <Từ_khóa_dẫn_xuất> có thể là public hoặc private:

Dẫn xuất public quy định phạm vi truy nhập như sau:

– Các thành phần private của lớp cơ sở thì không thể truy cập từ lớp dẫn xuất.

– Các thành phần public của lớp cơ sở trở thànhcác thành phần public của lớp dẫn xuất.

Dẫn xuất private quy định phạm vi truy nhập như sau:

– Các thành phần private của lớp cơ sở thì không thể truy cập từ lớp dẫn xuất.

– Các thành phần public của lớp cơ sở trở thành các thành phần private của lớp dẫn xuất.

Ví dụ:

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

class Account
{
public:
	double balance;
	Account()
	{
		balance = 0.0;
	}
	void deposit(double amt)
	{
		balance += amt;
		cout << "So du tai khoan: " << balance << endl;
	}
	void withdraw(double amt)
	{
		balance -= amt;
		cout << "So du tai khoan: " << balance << endl;
	}
};

class SavingsAccount : public Account
{
public:
	double interestRate;
	SavingsAccount()
	{
		balance = 0.0;
		interestRate = 6.0;
	}
	double getInterest()
	{
		return balance*interestRate/12;
	}
};

int main()
{
	SavingsAccount sa;
	double so_tien;
	cout << "So tien nap vao: ";
	cin >> so_tien;
	sa.deposit(so_tien);
```

```
        cout << "So tien can rut: ";
        cin >> so_tien;
        sa.withdraw(so_tien);
        cout << "Tien lai: " << sa.getInterest();
    }
```

Kết quả:

```
So tien nap vao: 10000
So du tai khoan: 10000
So tien can rut: 2000
So du tai khoan: 8000
Tien lai: 4000
```

5.2.2. Từ khóa dẫn xuất protected

Theo nguyên tắc kế thừa thì các thành phần khai báo private không được kế thừa trong lớp dẫn xuất, nên nếu muốn lớp dẫn xuất kế thừa các thành phần nào đó của lớp cơ sở, ta phải chuyển sang vùng public. Nhưng như vậy sẽ làm phá vỡ tính che dấu dữ liệu của lập trình hướng đối tượng. Vì vậy, ngôn ngữ C++ đưa ra cách giải quyết khác là thêm từ khóa dẫn xuất protected. Các thành phần protected của lớp cơ sở hoàn toàn giống các thành phần private ngoại trừ việc có thể kế thừa từ lớp dẫn xuất trực tiếp từ lớp cơ sở. Cụ thể như sau:

– 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.

– 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.

5.2.3. Dẫn xuất protected

Dẫn xuất protected quy định phạm vi truy nhập như sau:

– Các thành phần private của lớp cơ sở thì không thể truy cập được từ lớp dẫn xuất protected.

– Các thành phần protected của lớp cơ sở trở thành các thành phần protected của lớp dẫn xuất.

– Các thành phần public của lớp cơ sở cũng trở thành các thành phần protected của lớp dẫn xuất.

Bảng 5.1. Phạm vi truy vập của từ khóa dẫn xuất

Truy cập	Public	Protected	Private
Trong cùng lớp	Có	Có	Có
Lớp kế thừa	Có	Có	Không
Bên ngoài lớp	Có	Không	Không

5.2.4. 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 kế thừa từ các lớp cơ sở. Để trình biên dịch phân biệt được thành phần thuộc lớp nào, ta sử dụng cú pháp như sau:

```
Tên_đối_tượng.Tên_lớp :: Tên_thành_phần
```

Ví dụ: Cho 2 lớp A và B như sau:

```
class A
{
public:
     int n;
     void nhap()
     {    cout << "Nhap n: ";
          cin >> n;
     }
};
class B : public A
{
public:
     int m;
     void nhap()
     {    cout << "Nhap m: ";
          cin >> m;
     }
};
int main()
{
     B obj;
     obj.A::nhap();  //là hàm nhap() định nghĩa trong lớp A
     obj.B::nhap();  //là hàm nhap() định nghĩa trong lớp B
     cout << "Gia tri n thua ke tu lop A = " << obj.A::n;
     cout << "\nGia tri m khai bao trong lop B = " << obj.B::m;
}
```

Kết quả:

```
Nhap n: 5
Nhap m: 10
Gia tri n thua ke tu lop A = 5
Gia tri m khai bao trong lop B = 10
```

Lưu ý:

– Để sử dụng các thành phần của lớp dẫn xuất, có thể không cần tên lớp, chỉ cần tên thành phần. Khi đó trình dịch tự nhận biết thành phần đó thuộc lớp nào. Trình biên dịch không nhận biết được thành phần này thuộc lớp nào nó sẽ đưa ra một thông báo lỗi.

5.2.5. Định nghĩa lại 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 trong lớp dẫn xuất ghi đè lên hàm được định nghĩa trong lớp cơ sở.

Ví dụ:

```
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()
        {
            cout << "Nhap d: "; cin >> d;
            cout << "Gia tri cua d: " << d;
        }
    };
    int main()
    {
        B obj;
        obj.nhap();     //hàm nhập được định nghĩa trong B
        obj.hienthi();  //báo lỗi vì lớp B kế thừa private tử lớp A
    }
```

Trong chương trình trên, vì lớp B kế thừa private tử lớp A nên các thành phần public của lớp A là các hàm nhap() và hienthi() trở thành thành phần private của lớp B, nên không thể truy cập từ bên ngoài lớp.

5.2.6. Hàm khởi tạo đối với tính kế thừa

Các hàm khởi tạo của lớp cơ sở 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 khởi tạo lớp dẫn xuất để tạo đối tượng của lớp dẫn xuất sẽ kéo theo việc gọi đến một hàm khởi tạo của lớp cơ sở. Thứ tự thực hiện của các hàm khởi tạo của lớp cơ sở được gọi trước rồi đến hàm khởi tạo của lớp dẫn xuất.

C++ thực hiện điều này bằng cách trong định nghĩa hàm khởi tạo của lớp dẫn xuất, mô tả một lời gọi tới hàm khởi tạo của lớp cơ sở. Cú pháp để truyền đối 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)
{
    //Định nghĩa hàm khởi tạo của lớp dẫn xuất
} ;
```

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

Ví dụ:

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

class Diem
{
public:
	double x, y;
public:
	Diem()
	{
		x = y = 0.0;
	}
	Diem(double x1, double y1)
	{
		x = x1;
		y = y1;
	}
	friend istream& operator>>(istream& is, Diem& d)
	{
		cout << "x = ";
		cin >> d.x;
		cout << "y = ";
		cin >> d.y;
		return is;
	}
	friend ostream& operator<<(ostream& os, const Diem& d)
	{
		cout << "(" << d.x << "," << d.y << ")";
		return os;
	}
};

class DuongTron : public Diem
{
private:
	double r;
public:
	DuongTron()
	{
		r = 0.0;
	}
	DuongTron(double x1, double y1, double r1): Diem(x1, y1)
	{
		r = r1;
	}
```

```
        double tinhDienTich()
        {
            return 3.14 * r * r;
        }
        double getr()
        {
            return r;
        }
};
void main()
{
        DuongTron c(0.0,0.0,10.0);
        cout << "Tam duong tron: " << c << endl;
        cout << "Dtich dtron bkinh " << c.getr() << " la: " << c.tinhDienTich();
}
```

Kết quả:

```
Tam duong tron: (0,0)
Dtich dtron bkinh 10 la: 314
```

5.2.7. 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 khởi tạo và hàm hủy, các hàm khởi tạo thi hành theo thứ tự dẫn xuất. Các hàm hủy được thi hành theo thứ tự ngược lại. Nghĩa là, hàm khởi tạo của lớp cơ sở thực thi trước hàm khởi tạo của lớp dẫn xuất, hàm hủy của lớp dẫn xuất thực thi trước hàm hủy của lớp cơ sở.

Ví dụ:

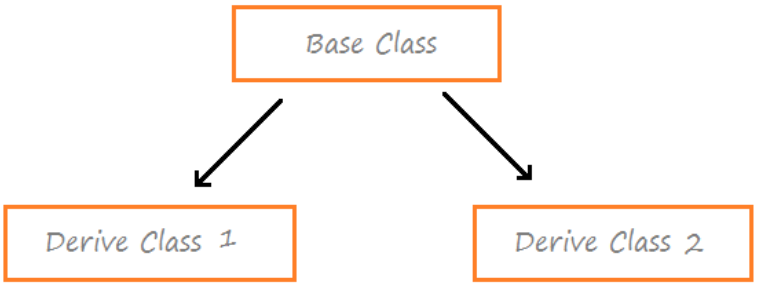
```
#include <iostream>
using namespace std;

class LopCoSo
{
public:
        LopCoSo()
        {
            cout << "Ham khoi tao cua lop co so!" << endl;
        }
        ~LopCoSo()
        {
            cout << "Ham huy cua lop co so!" << endl;
        }
};
```

```
class LopDanXuat : public LopCoSo
{
public:
	LopDanXuat()
	{
		cout << "Ham khoi tao cua lop dan xuat!" << endl;
	}
	~LopDanXuat()
	{
		cout << "Ham huy cua lop dan xuat!" << endl;
	}
};

void main()
{
	LopDanXuat obj;
}
```

Kết quả:

```
Ham khoi tao cua lop co so!
Ham khoi tao cua lop dan xuat!
Ham huy cua lop dan xuat!
Ham huy cua lop co so!
```

5.2.8. Kế thừa phân cấp (Hierarchical Inheritance)



Hình 5.3. Hierarchical Inheritance

Cú pháp:

```
class Lớp_Cơ_Sở
{
	//Định nghĩa lớp cơ sở
};

class Lớp_Dẫn_Xuất_1 : <Từ_khóa_dẫn_xuất> Lớp_Cơ_Sở
{
	//Định nghĩa lớp dẫn xuất 1
};
```

```
class Lớp_Dẫn_Xuất_2 : <Từ_khóa_dẫn_xuất> Lớp_Cơ_Sở
{
    //Định nghĩa lớp dẫn xuất 2
};
```

Ví dụ:

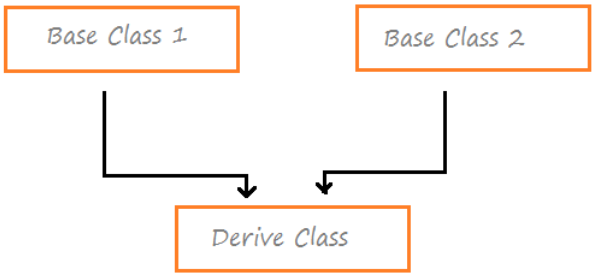
```
#include <iostream>
using namespace std;

//Lớp cơ sở
class Vehicle
{
public:
    Vehicle()
    {
        cout << "Day la Vehicle -> ";
    }
};
//Lớp dẫn xuất 1
class Car : public Vehicle
{
public:
    Car()
    {
        cout << "Car" << endl;
    }
};
//Lớp dẫn xuất 2
class Bus : public Vehicle
{
public:
    Bus()
    {
        cout << "Bus" << endl;
    }
};

int main()
{
    Car obj1;
    Bus obj2;
}
```

Kết quả:

```
Day la Vehicle -> Car
Day la Vehicle -> Bus
```

5.3. Đa kế thừa

5.3.1. Multiple Inheritance: định nghĩa lớp dẫn xuất từ nhiều lớp cơ sở



Hình 5.4. Multiple Inheritance

Cú pháp:

```
class Lớp_Cơ_Sở_1
{
     //Định nghĩa lớp cơ sở 1
};
class  Lớp_Cơ_Sở_2
{
     //Định nghĩa lớp cơ sở 2
};
class Lớp_Dẫn_Xuất : <Từ_khóa_dẫn_xuất> Lớp_Cơ_Sở_1,
                          <Từ_khóa_dẫn_xuất>Lớp_Cơ_Sở_2

{
     //Định nghĩa lớp dẫn xuất
}
```

<Từ_khóa_dẫn_xuất>: private, public hoặc protected.

Ví dụ:

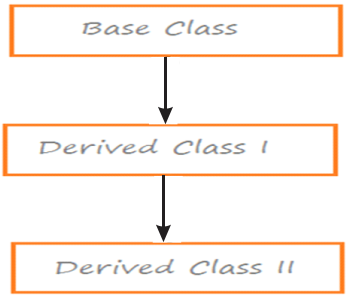
```
#include <iostream>
using namespace std;
class LopCoSo1
{
public:
     int x;
     LopCoSo1()
     {
          x = 1;
          cout << "Ham khoi tao lop co so 1!" << endl;
     }
};
```

```
class LopCoSo2
{
public:
	int y;
	LopCoSo2()
	{
		y = 2;
		cout << "Ham khoi tao lop co so 2!" << endl;
	}
};
class LopDanXuat : public LopCoSo1, public LopCoSo2
{
public:
	int z;
	LopDanXuat()
	{
		z = x + y; //x, y kế thừa từ lớp cơ sở 1 và 2
		cout << "Ham khoi tao lop dan xuat!" << endl;
	}
	void display()
	{
		cout << "Gia tri cua z = " << z << endl;
	}
};
void main()
{
	LopDanXuat obj;
	obj.display();
}
```

Kết quả:

```
Ham khoi tao lop co so 1!
Ham khoi tao lop co so 2!
Ham khoi tao lop dan xuat!
Gia tri cua z = 3
```

5.3.2. Multilevel Inheritance: kế thừa nhiều cấp



Hình 5.5. Multilevel Inheritance

Cú pháp:

```
class Lớp_Cơ_Sở
{
	//Định nghĩa lớp cơ sở 1
};

class Lớp_Dẫn_Xuất_Mức_1 : <Từ_khóa_dẫn_xuất> Lớp_Cơ_Sở
{
	//Định nghĩa lớp dẫn xuất mức 1
}
class Lớp_Dẫn_Xuất_Mức_2 : <Từ_khóa_dẫn_xuất> Lớp_Dẫn_Xuất_Mức_1
{
	//Định nghĩa lớp dẫn xuất mức 2
}
```

Ví dụ:

```
#include <iostream>
using namespace std;
//Lớp cơ sở
class Account
{
public:
	double balance;
	Account()
	{
		balance = 0;
	}
	Account(int bal)
	{
		balance = bal;
	}
	void withdraw(double amt)
	{
		balance -= amt;
		cout <<"Rut tien thanh cong:" <<amt << endl;
	}
	void deposit(double amt) {
		balance += amt;
		cout << "Nap tien thanh cong: " <<amt << endl;
	}
	void display_balance()
	{
		cout << "So du: " << balance << endl;
	}
};
```

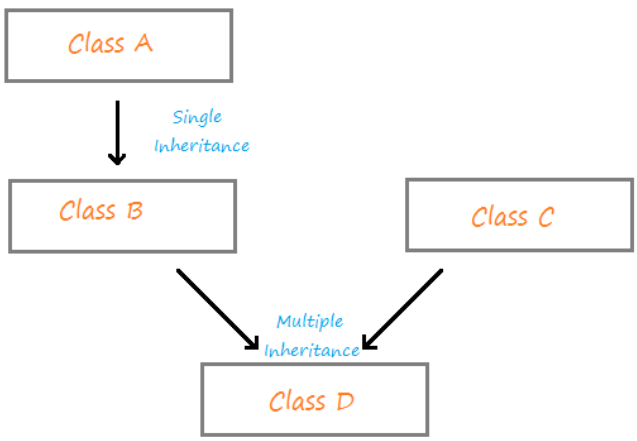
```
//Lớp dẫn xuất mức 1
class Saving_Account : public Account
{
public:
	double interest_rate;

	Saving_Account()
	{
		interest_rate = 6.0;
	}
	Saving_Account(double bal)
	{
		balance = bal;
		interest_rate = 6.0;
	}
	Saving_Account(double bal, double rate)
	{
		balance = bal;
		interest_rate = rate;
	}
	void calculate_interest(int years)
	{
		balance += (balance * interest_rate / 100 * years);
	}
};

//Lớp dẫn xuất mức 2
class Monthly_Average_Saving_Account : public Saving_Account
{
public:

	double average_balance;

	Monthly_Average_Saving_Account()
	{
		average_balance = 100;
	}
	Monthly_Average_Saving_Account(double bal)
	{
		average_balance = 500;
		balance = bal;
	}
	Monthly_Average_Saving_Account(double bal, double avg)
	{
		balance = bal;
		average_balance = avg;
	}
```

```
        void check_average()
        {
            if (balance < average_balance)
            {
                cout << "So du thap!" << endl;
            }
            else
            {
                cout << "So du trung binh!" << endl;
            }
        }
    };
    int main()
    {
        Monthly_Average_Saving_Account acc(1000);
        acc.display_balance();
        acc.deposit(100);
        acc.display_balance();
        acc.check_average();
        acc.withdraw(700);
        acc.display_balance();
        acc.check_average();
    }
```

Kết quả:

```
So du: 1000
Nap tien thanh cong: 100
So du: 1100
So du trung binh!
Rut tien thanh cong:700
So du: 400
So du thap!
```

5.3.3. Hybrid Inheritance



Hình 5.6. Hybrid Inheritance

Cú pháp:

```
class A
{
	//Định nghĩa lớp
};
class B : <Từ_khóa_dẫn_xuất> A
{
	//Định nghĩa lớp
};
class C
{
	//Định nghĩa lớp
};
class D : <Từ_khóa_dẫn_xuất> B, <Từ_khóa_dẫn_xuất> C
{
	//Định nghĩa lớp
};
```

Ví dụ:

```
#include <iostream>
using namespace std;
class Animal
{
public:
	Animal()
	{
		cout << "Da la Animal!" << endl;
	}
};
class Dog : public Animal
{
public:
	Dog()
	{
		cout << "Day la Dog!" << endl;
	}
};
class Domestic
{
public:
	Domestic()
	{
		cout << "Giong cho thuan hoa!" << endl;
	}
};
```

```
class KyKy : public Dog, public Domestic
{
public:
	KyKy()
	{
		cout << "Day la chu cho Kyky!" << endl;
	}
};

int main()
{
	KyKy my_dog;
}
```

Kết quả:

```
Da la Animal!
Day la Dog!
Giong cho thuan hoa!
Day la chu cho Kyky!
```

5.4. Hàm ảo

5.4.1. Đặt vấn đề

Trước khi đưa ra khái niệm về hàm ảo, ta hãy xem ví dụ sau:

Ví dụ: Giả sử có 3 lớp A, B và C được xây dựng như sau:

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

```
        void xuat()
        {
            cout << "Lop !" << endl;
        }
};
```

Cả 3 lớp này đều có hàm thành phần là xuat(). Lớp C có hai lớp cơ sở là A, B nên C kế thừa các hàm thành phần của A và B. Do đó một đối tượng của C sẽ có 3 hàm xuat(). Hàm main() ta khai báo biến obj và gọi các hàm:

```
void main()
{
    C obj;              // obj là đối tượng kiểu lớp C
    obj.xuat();         // Gọi tới hàm thành phần xuat() của lớp C
    obj.B::xuat();      // Gọi tới hàm thành phần xuat() của lớp B
    obj.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 obj có kiểu lớp C 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.

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

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;
```

Khi đó, ta xem 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, 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, gọi là sự liên kết tĩnh.

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.

Vậy khi 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 trỏ tới chứ không phụ thuộc vào kiểu lớp của con trỏ. Để giải quyết vấn đề này, ngôn ngữ C++ dùng khái niệm hàm ảo.

5.4.2. 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 phải định nghĩa lại hàm ảo đó. Các hàm ảo chính là 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 nạp chồng hàm;
6. Không được định nghĩa hàm khởi 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:

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

class A
{
public:
	virtual void xuat()
	{
		cout << "Lop A!" << endl;
	}
};
class B : public A
{
public:
	void xuat()
	{
		cout << "Lop B!" << endl;
	}
};
class C : public B
{
public:
	void xuat()
	{
		cout << "Lop C!" << endl;
	}
};

void main()
{
	C obj;			// obj là đối tượng kiểu lớp C
	obj.xuat();		// Gọi tới hàm thành phần xuat() của lớp C
	obj.B::xuat();	// Gọi tới hàm thành phần xuat() của lớp B
	obj.A::xuat();	// Gọi tới hàm thành phần xuat() của lớp A
```

```
        A* p, * q, * r; // p,q,r là các con trỏ kiểu lớp A
        A a; // a là đối tượng kiểu lớp A
        B b; // b là đối tượng kiểu lớp B
        C c; // c là đối tượng kiểu lớp C
        p = &a;
        q = &b;
        r = &c;
        p->xuat();
        q->xuat();
        r->xuat();
    }
```

Kết quả:

```
Lop C!
Lop B!
Lop A!
Lop A!
Lop B!
Lop C!
```

Lưu ý: Từ khoá virtual không được đặt bên ngoài định nghĩa lớp.

Ví dụ:

```
    class A
    {
        ...
        virtual void hienthi();
    };
    virtual void hienthi() //Báo lỗi
    {
        cout << "Day la lop A!" << endl;
    }
```

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 kế thừa) 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.

5.4.3. 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 &.

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
```

Lưu 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;
q = &a;    //là sai
```

Ví dụ:

```
//Định nghĩa các lớp A, B, C như trên
void main()
{
     A* p;
     A a; B b; C c;
     a.xuat(); //goi ham cua lop A
     p = &b; //p tro to doi tuong b cua lop B
     p->xuat(); //goi ham cua lop B
     p = &c; //p tro to doi tuong c cua lop C
     p->xuat(); //goi ham cua lop C
}
```

Kết quả:

```
Lop A!
Lop B!
Lop C!
```

Lưu ý:

– Cùng một câu lệnh p->xuat(); được tương ứng với nhiều hàm khác nhau khác nhau khi xuat() là hàm ảo. Đây chính là sự tương ứng bội, cho phép xử lý nhiều đối tượng khác nhau theo cùng một cách thức.

– Cũng với lời gọi: p->xuat(); (xuat() 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 in ra số nguyên dưới dạng số hệ 16 và số hệ 8.

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

class Num
{
public:
	int i;
	Num(int x)
	{
		i = x;
	}
	virtual void showNum()
	{
		cout << "So he 10: ";
		cout << dec << i << endl;
	}
};
class OutHex : public Num
{
public:
	OutHex(int n) : Num(n)
	{	}
	void showNum()
	{
		cout << "So he 10: " << dec << i << endl;
		cout << "So he 16: " << hex << i << endl;
	}
};
class OutOct : public Num
{
public:
	OutOct(int n) : Num(n)
	{	}
	void showNum()
	{
		cout << "So he 10: " << dec << i << endl;
		cout << "So he 8: " << oct << i << endl;
	}
};
```

```
void main()
{
	Num n(10);
	OutOct o(10);
	OutHex h(10);
	Num* p;
	p = &n;
	cout << "Goi ham showNum cua lop co so Num:" << endl;
	p->showNum(); //goi ham cua lop co so
	p = &o;
	cout << "Goi ham showNum cua lop dan xuat OutOct:" << endl;
	p->showNum(); //goi ham cua lop dan xuat
	p = &h;
	cout << "Goi ham showNum cua lop co so OutHex:" << endl;
	p->showNum(); //goi ham cua lop dan xuat
}
```

Kết quả:

```
Goi ham showNum cua lop co so Num:
So he 10: 10
Goi ham showNum cua lop dan xuat OutOct:
So he 10: 10
So he 8: 12
Goi ham showNum cua lop co so OutHex:
So he 10: 10
So he 16: a
```

BÀI TẬP CHƯƠNG 5

Bài 1: Viết chương trình quản lý sinh viên, giáo viên theo yêu cầu sau:

Câu 1: Định nghĩa lớp Person, bao gồm các thông tin: Tên, Phái, Ngày sinh, Địa chỉ. Với đầy đử các hàm get, set, phương thức khởi tạo mặc định, có tham số, hủy.

1. Viết phương thức nhap(), nhập thông tin Person từ bàn phím.
2. Viết phương thức xuat(), hiển thị thông tin Person ra màn hình.

Câu 2: Định nghĩa lớp Student kế thừa lớp Person, bao gồm các thông tin: Mã sinh viên (8 ký tự), Điểm trung bình, Email.

1. Viết lại phương thức nhap(), nhập thông tin Student từ bàn phím.
2. Viết lại phương thức xuat(), hiển thị thông tin Student ra màn hình.
3. Viết phương thức xét học bổng, điểm trung bình trên 8 thì được học bổng.

Câu 3: Viết chương trình, tạo một menu như sau:

```
1: Nhap n sinh vien (n la so luong, nhap tu ban phim).
2: Hien thi tat ca thong tin sinh vien ra man hinh.
3: Hien thi sinh vien co diem trung binh thap nhat va cao nhat.
4: Tim kiem sinh vien theo ma sinh vien (ma sinh vien nhap tu ban phim).
5: Sap xep sinh vien theo ten va hien thi ra man hinh.
6: Hien thi cac sinh vien co hoc bong (theo thu tu diem cao den thap).
7: Thoat.
```

Câu 4: Định nghĩa lớp Teacher kế thừa lớp Person, bao gồm các thông tin: Lớp dạy, Lương một tiết, Số tiết dạy trong tháng.

1. Viết lại phương thức nhap(), nhập thông tin Teacher từ bàn phím.
2. Viết lại phương thức xuat(), hiển thị thông tin Teacher ra màn hình.
3. Viết phương thức tính lương, lương = lương 1 tiết * số tiết dạy trong tháng.

Câu 5: Viết chương trình, tạo một menu như sau:

```
1: Nhap n giang vien (n la so luong, nhap tu ban phim).
2: Hien thi tat ca thong tin giang vien ra man hinh.
3: Hien thi giang vien co luong cao nhat.
4: Thoat.
```

Bài 2: Quản lý sở thú:

Câu 1: Định nghĩa lớp có tên Animal gồm các thông tin: Tên, Tuổi, Mô tả. Và các phương thức: xemThongTin(), hiển thị tên tuổi và mô tả của animal và một phương thức ảo keu(), biểu thị tiếng kêu của anmail.

Câu 2: Định nghĩa các phương thức khởi tạo mặc định, 1 tham số (ten), 2 tham số (tên, tuổi), 3 tham số (tên, tuổi, mô tả), phương thức hủy.

Câu 3: Định nghĩa lớp Tiger, Dog, Cat như sau:

1. Kế thừa lớp Animal
2. Ghi đè phương thức kêu, cụ thể cho từng loài vật.

Câu 4: Định nghĩa lớp Chuong, bao gồm các thông tin: Mã chuồng, Mảng các Animal.Và các phương thức them(Animal a), thêm con vật vào chuồng, xoa(String ten), xóa con vật có tên tương ứng.

Câu 5: Định nghĩa lớp Zoo, bao gồm: một mảng các chuồng, và các phương thức thêm các chuồng vào danh sách chuồng và xóa chuồng khỏi danh sách chuồng.

Câu 6: Viết chương trình, tạo một menu như sau:

```
1: Thêm chuồng.
2: Xóa chuồng.
3: Thêm con vật.
4: Xóa con vật.
5: Xem tất cả các con vật
6: Thoát.
```

Mục 3: thì yêu cầu nhập loại con vật muốn thêm (Tiger, Dog, Cat), sau đó nhập các con vật tương ứng và thêm và danh sách Animal.

Mục 5: hiển thị thông tin (bao gầm cả kêu) của từng con vật.

CHƯƠNG 6

ĐA HÌNH (POLYMORPHISM)

Các nội dung chính:

- *Giới thiệu về đa hình*
- *Sử dụng phương thức trừu tượng*
- *Nạp chồng (overloading)*
- *Ghi đè (overriding)*
- *Lớp cơ sở ảo*

6.1. Giới thiệu

Đa hình là khái niệm luôn đi kèm với kế thừa. Do tính kế thừa, một lớp có thể sử dụng lại các phương thức của lớp khác. Tuy nhiên, nếu cần thiết, lớp dẫn xuất cũng có thể định nghĩa lại một số phương thức của lớp cơ sở. Đó là sự nạp chồng phương thức trong kế thừa. Nhờ sự nạp chồng phương thức này, ta chỉ cần gọi tên phương thức bị nạp chồng từ đối tượng mà không cần quan tâm đó là đối tượng của lớp nào. Chương trình sẽ tự động kiểm tra xem đối tượng là thuộc kiểu lớp cơ sở hay thuộc lớp dẫn xuất, sau đó sẽ gọi phương thức tương ứng với lớp đó. Đó là tính đa hình.

Sự kế thừa trong C++ cho phép có sự tương ứng giữa lớp cơ sở và các lớp dẫn xuất trong sơ đồ kế thừa:

- Một con trỏ có kiểu lớp cơ sở luôn có thể trỏ đến địa chỉ của một đối tượng của lớp dẫn xuất.
- Khi thực hiện lời gọi một phương thức của lớp, trình biên dịch sẽ quan tâm đến kiểu của con trỏ chứ không phải đối tượng mà con trỏ đang trỏ tới: phương thức của lớp mà con trỏ có kiểu được gọi chứ không phải phương thức của đối tượng mà con trỏ đang trỏ tới được gọi.

Ví dụ: Lớp Bus kế thừa từ lớp Car, cả hai lớp này đều định nghĩa phương thức show()

```
class Car
{
```

```
public:
    void show();
};
class Bus : public Car
{
public:
    void show();
};
```

Khi đó, nếu ta khai báo một con trỏ lớp Car, nhưng lại trỏ vào địa chỉ của một đối tượng lớp Bus:

```
Bus my_bus;
Car* ptr_car = &my_bus;
ptr_car->show();
```

Câu lệnh Car* ptr_car = &my_bus; là đúng, nhưng khi gọi thì sẽ gọi đến phương thức show() của lớp Car (là kiểu của con trỏ ptr_car), mà không gọi tới phương thức show() của lớp Bus (là kiểu của đối tượng my_bus mà con trỏ ptr_car đang trỏ tới).

Để giải quyết vấn đề này, C++ đưa ra một khái niệm là phương thức trừu tượng. Khi gọi một phương thức từ một con trỏ đối tượng, trình biên dịch sẽ xác định kiểu của đối tượng mà con trỏ đang trỏ đến, sau đó nó sẽ gọi phương thức tương ứng với đối tượng mà con trỏ đang trỏ tới.

Khai báo phương thức trừu tượng: phương thức trừu tượng (còn gọi là phương thức ảo, hàm ảo) được khai báo với từ khoá virtual:

Nếu khai báo trong phạm vi lớp:

```
virtual <Kiểu_trả_về> <Tên_phương_thức>([<Các_tham_số>]);
```

Nếu định nghĩa ngoài phạm vi lớp:

```
virtual <Kiểu_trả_về> <Tên_lớp> :: <Tên_phương_thức>([<Các_tham_số>]){…}
```

Ví dụ: là khai báo phương thức trừu tượng show() của lớp Car: phương thức không có tham số và không cần giá trị trả về (void).

```
class Car
{
public:
    virtual void show();
};
```

Lưu ý:

– *Từ khoá virtual có thể đặt trước hay sau kiểu trả về của phương thức.*

– *Với cùng một phương thức được khai báo ở lớp cơ sở lẫn lớp dẫn xuất, chỉ cần dùng từ khoá virtual ở một trong hai lần định nghĩa phương thức đó là đủ: hoặc ở lớp cơ sở, hoặc ở lớp dẫn xuất.*

– *Trong trường hợp cây kế thừa có nhiều mức, cũng chỉ cần khai báo phương thức là trừu tượng (virtual) ở một mức bất kì. Khi đó, tất cả các phương thức trùng tên với phương thức đó ở tất cả các mức đều được coi là trừu tượng.*

– *Đôi khi không cần thiết phải định nghĩa chồng (trong lớp dẫn xuất) một phương thức đã được khai báo trừu tượng trong lớp cơ sở.*

6.2. Sử dụng phương thức trừu tượng

Một khi phương thức được khai báo là trừu tượng thì khi một con trỏ gọi đến phương thức đó, chương trình sẽ thực hiện phương thức tương ứng với đối tượng mà con trỏ đang trỏ tới, thay vì thực hiện phương thức của lớp cùng kiểu với con trỏ. Đây được gọi là hiện tượng đa hình (tương ứng bội) trong C++.

Chương trình sau ví dụ về việc sử dụng phương thức trừu tượng: lớp Bus kế thừa từ lớp Car, hai lớp này cùng định nghĩa phương thức trừu tượng show().

– Khi ta dùng một con trỏ có kiểu lớp Car trỏ vào địa chỉ của một đối tượng kiểu Car, nó sẽ gọi phương thức show() của lớp Car.

– Khi ta dùng cũng con trỏ đó, trỏ vào địa chỉ của một đối tượng kiểu Bus, nó sẽ gọi phương thức show() của lớp Bus.

Ví dụ:

```
#include<iostream>
#include<string>

using namespace std;

class Car
{
private:
```

```
        int  speed;     // Tốc độ
        string  mark;   // Nhãn hiệu
        float price;    // Giá xe
    public:
        Car();
        Car(int speed, string mark, float price);
        virtual void show(); // Giới thiệu xe, trừu tượng
        int getSpeed()
        {
            return speed;
        };
        string getMark()
        {
            return mark;
        };
        float getPrice()
        {
            return price;
        };
    };
    Car::Car()
    {
        this->speed = 0;
        this->mark = "";
        this->price = 0;
    }
    Car::Car(int speed, string mark, float price)
    {
        this->speed = speed;
        this->mark = mark;
        this->price = price;
    }
    void Car::show()     // Phương thức hiển thị xe
    {
        cout << "Day la " << mark << " co toc do " << speed
                << "km/h va gia la " << price << endl;
    }
    /* Định nghĩa lớp Bus kế thừa từ lớp Car */
    class Bus : public Car
    {
        int label;                  // Số hiệu tuyến xe
    public:
        Bus(int speed = 0, string mark = "", float price = 0, int lable = 0);
        void setLabel(int);         // Gán số hiệu tuyến xe
        int getLabel();             // Đọc số hiệu tuyến xe
        void show();
    };
```

```
// Cài đặt lớp Bus
Bus::Bus(int speed, string mark, float price, int label)
                                        : Car(speed, mark, price)
{
     this->label = label;
}

// Định nghĩa nạp chồng phương thức
void Bus::show()     // Giới thiệu xe bus
{
     cout << "Day la bus cua " << getMark() << ", tuyen so " << label <<
", co toc do " << getSpeed() << "km / h va gia la " << getPrice() << endl;
}

int main()
{
     Car* ptrCar, myCar(120, "Ford", 3000);
     Bus myBus(100, "Mercedes", 5000, 56);     // Biến đối tượng của lớp Bus
     ptrCar = &myCar;                    // Trỏ đến đối tượng lớp Car
     ptrCar->show();                     // Phương thức của lớp Car
     ptrCar = &myBus;                    // Trỏ đến đối tượng lớp Bus
     ptrCar->show();
     return 0;
}
```

Kết quả:

```
Day la Ford co toc do 120km/h va gia la 3000
Day la bus cua Mercedes, tuyen so 56, co toc do 100km / h va gia la 5000
```

6.3. Nạp chồng (Overloading)

6.3.1. Nạp chồng phương thức

Trong một lớp, ta có thể tạo ra nhiều hàm với cùng một tên gọi nhưng khác nhau các dữ liệu đầu vào hoặc tham số, đó gọi là nạp chồng phương thức. Lợi ích của việc quá tải phương thức là chúng ta có thể khai báo cùng một tên phương thức trong cùng chương trình, không cần phải khai báo tên khác cho cùng một hành động.

Ví dụ: Lớp Cal sau có hai phương thức add nhưng khác nhau về tham số:

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

class Cal
{
```

```
    public:
        static int add(int a, int b)
        {
            return a + b;
        }
        static int add(int a, int b, int c)
        {
            return a + b + c;
        }
    };

    int main()
    {
        Cal C;
        cout << C.add(5, 10) << endl;
        cout << C.add(15, 20, 25);
    }
```

Kết quả:

```
15
60
```

6.3.2. Nạp chồng toán tử

Việc nạp chồng toán tử thường được sử dụng trong C++ là định nghĩa lại toán tử của C++. Lợi ích của nạp chồng toán tử là thực hiện các thao tác khác nhau trên cùng một toán hạng.

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

    class Test
    {
    private:
        int num;
    public:
        Test() : num(8) {}
        void operator ++()
        {
            num = num + 2;
        }
        void Print()
        {
            cout << "Num : " << num;
        }
    };
```

```
int main()
{
	Test tt;
	++tt;  // calling of a function "void operator ++()"
	tt.Print();
}
```

Kết quả:

```
Num : 10
```

6.4. Ghi đè (Overriding)

Nếu lớp dẫn xuất định nghĩa cùng một phương thức đã được định nghĩa trong lớp cơ sở của nó được gọi là phương thức ghi đè trong C ++. Phương thức ghi đè cho phép cung cấp việc thực hiện cụ thể chức năng đã được lớp cơ sở của nó cung cấp. Overriding thường được sử dụng trong method ở lớp con.

Một số quy tắc sử dụng phương thức overriding:

- Các phương thức được static thì không overriden nhưng được mô tả lại.
- Các phương thức không kế thừa sẽ không được overriden.

Ví dụ: Ghi đè phương thức eat():

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

class Animal
{
public:
	void eat()
	{
		cout << "An ...";
	}
};

class Dog : public Animal
{
public:
	void eat()
	{
		cout << "An thit ...";
	}
};
```

```
int main()
{
	Dog d = Dog();
	d.eat();
}
```

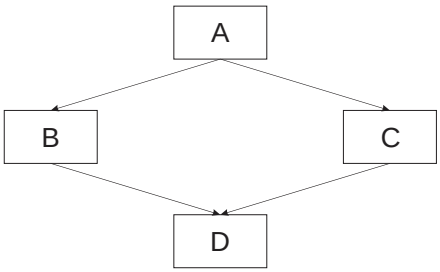
Kết quả:

```
An thit ...
```

6.5. Lớp cơ sở ảo

6.5.1. 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 6.1. Lớp cở sở ảo

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 cú pháp sau:

```
class A
{
	//Định nghĩa lớp
```

```
};
class B : virtual public A
{
     //Định nghĩa lớp
};
class C : virtual public A
{
     //Định nghĩa lớp
};
class D : public B, public C
{
     //Định nghĩa lớp
};
```

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.

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

Ví dụ:

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

class A
{
     float x, y;
public:
     void setx(float x)
     {
          this->x = x;
     }
     void sety(float y)
     {
          this->y = y;
     }
     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()
{
     D d;
     cout << "d.B::setx(1); d.B::sety(2)" << endl;
     d.B::setx(1);   d.B::sety(2);
     cout << "d.C::getx() = "; cout << d.C::getx() << endl;
     cout << "d.B::getx() = "; cout << d.B::getx() << endl;
     cout << "d.C::gety() = "; cout << d.C::gety() << endl;
     cout << "d.B::gety() = "; cout << d.B::gety() << endl;
     cout << "d.C::setx(3); d.C::sety(3)" << endl;
     d.C::setx(2);   d.C::sety(3);
     cout << "d.C::getx() = "; cout << d.C::getx() << endl;
     cout << "d.B::getx() = "; cout << d.B::getx() << endl;
     cout << "d.C::gety() = "; cout << d.C::gety() << endl;
     cout << "d.B::gety() = "; cout << d.B::gety() << endl;
}
```

Kết quả:

```
d.B::setx(1); d.B::sety(2)
d.C::getx() = 1
d.B::getx() = 1
d.C::gety() = 2
d.B::gety() = 2
d.C::setx(3); d.C::sety(3)
d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3
```

Lưu ý: Nếu lớp B và lớp C kế thừa lớp A không có virtual.

```
class B : public A { };
class C : public A { };
```

Thì kết quả như sau:

```
d.B::setx(1); d.B::sety(2)
d.C::getx() = -1.07374e+08
d.B::getx() = 1
d.C::gety() = -1.07374e+08
d.B::gety() = 2
d.C::setx(3); d.C::sety(3)
d.C::getx() = 2
d.B::getx() = 1
d.C::gety() = 3
```

```
d.B::gety() = 2
```

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

Khi khởi tạo đối tượng của lớp dẫn xuất thì các hàm khởi 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. Dữ liệu được chuyển từ hàm khởi tạo của lớp dẫn xuất sang hàm khởi tạo của lớp cơ sở. Trong tình huống có lớp cơ sở ảo, cần phải tuân theo quy định sau:

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

Ví dụ:

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

class A
{
	float x, y;
public:
	A()
	{
		x = 0; y = 0;
	}
	A(float x1, float y1)
	{
		cout << "A::A(float,float):" << endl;
		x = x1; y = y1;
	}
	~A()
	{
		cout << "Ham huy lop A!" << endl;
	}
	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)" << endl;
	}
	~B()
	{
		cout << "Ham huy lop B!" << endl;
	}
};
class C : virtual public A
{
public:
	C(float x1, float y1) :A(x1, y1)
	{
		cout << "C::C(float,float)" << endl;
	}
	~C()
	{
		cout << "Ham huy lop C!" << endl;
	}
};
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)" << endl;
	}
	~D()
	{	cout << "Ham huy lop D!" << endl;
	}
};
void main()
{
	D d1(1, 2);
	cout << "D d1 (1,2)" << endl;
	cout << "d1.C::getx() = "; cout << d1.C::getx() << endl;
	cout << "d1.B::getx() = "; cout << d1.B::getx() << endl;
	cout << "d1.C::gety() = "; cout << d1.C::gety() << endl;
	cout << "d1.B::gety() = "; cout << d1.B::gety() << endl;
	cout << "d2 (10,20)" << endl;
	D d2(10, 20);
	cout << "d2.C::getx() = "; cout << d2.C::getx() << endl;
	cout << "d2.B::getx() = "; cout << d2.B::getx() << endl;
	cout << "d2.C::gety() = "; cout << d2.C::gety() << endl;
```

```
        cout << "d2.B::gety() = "; cout << d2.B::gety() << endl;
    }
```

Kết quả:

```
A::A(float,float):
B::B(float,float)
C::C(float,float)
D::D(float,float)
D d1 (1,2)
d1.C::getx() = 1
d1.B::getx() = 1
d1.C::gety() = 2
d1.B::gety() = 2
d2 (10,20)
A::A(float,float):
B::B(float,float)
C::C(float,float)
D::D(float,float)
d2.C::getx() = 10
d2.B::getx() = 10
d2.C::gety() = 20
d2.B::gety() = 20
Ham huy lop D!
Ham huy lop C!
Ham huy lop B!
Ham huy lop A!
Ham huy lop D!
Ham huy lop C!
Ham huy lop B!
Ham huy lop A!
```

BÀI TẬP CHƯƠNG 6

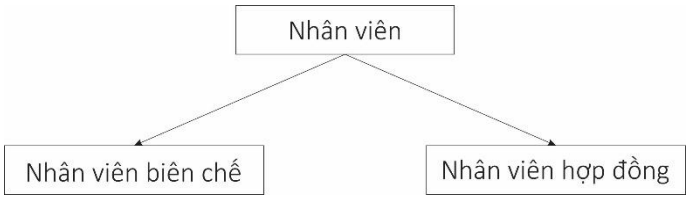
Bài 1: Hãy định nghĩa các lớp cần thiết và viết chương trình quản lý nhân viên của một công ty theo mô tả sau:

Công ty TD quản lý và tính tiền lương cho nhân viên theo 2 hình thức: biên chế và hợp đồng. Để quản lý nhân viên, cần các thông tin: mã nhân viên, họ và tên, lương. Ngoài ra, nhân viên biên chế cần có thêm: hệ số lương, phụ cấp chức vụ, nhân viên hợp đồng có thêm: tiền công một ngày, số ngày làm việc trong tháng.

Lương của nhân viên được tính theo công thức:

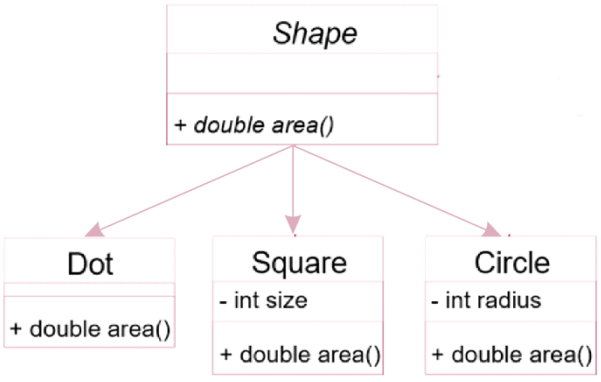
- Nhân viên biên chế = Lương cơ bản * (Hệ số lương + Phụ cấp chức vụ). Trong đó lương cơ bản = 1,390,000 đồng.
- Nhân viên hợp đồng = Tiền công một ngày * số ngày làm việc. Trong đó số ngày làm việc được tính bằng số ngày quy định + số ngày vượt quy định * 2.

Gợi ý: Thiết kế 3 lớp theo mô hình sau:



- Sử dụng hàm ảo (đa hình) để tính lương.
- Sử dụng duy nhất 1 mảng (danh sách) để lưu tất cả nhân viên trong công ty.

Bài 2: Hãy định nghĩa các lớp cần thiết và viết chương trình theo mô hình sau:



Lớp Shape:

- Lớp Shape sử dụng từ khóa abstract để khai báo đây lớp abstract.
- Lớp Shape có phương thức area() tính diện tích một hình.

Lớp Dot, Square và Circle kế thừa từ lớp Shape vậy nên cả 3 lớp này sẽ override (ghi đè) tất cả phương thức từ lớp cha. Như vậy 3 lớp trên sẽ ghi đè phương thức public abstract double area();