Mở đầu

Lập trình hướng đối tượng

Giới thiệu môn học

- Phần C++ đã học chỉ đủ để viết những chương trình C++ nhỏ.
- Sinh viên gặp nhiều khó khăn khi viết một chương trình lớn hoặc làm việc trong một nhóm cùng các lập trình viên khác? Làm thế nào để đảm bảo tính thống nhất trong chương trình lớn hoặc giữa các lập trình viên khác nhau?
- Trong môn học này, sinh viên sẽ học những kiến thức cơ bản về lập trình hướng đối tượng (Object-Oriented Programming - OOP). Mục đích là để học cách thiết kế và viết những dự án phần mềm lớn.

Lịch sử ngắn gọn

Các ngôn ngữ lập trình thời kỳ đầu như Basic, Fortran... không có cấu trúc và cho phép viết những đoạn mã rối rắm (spaghetti code). Lập trình viên sử dụng các lệnh goto" và "gosub" để nhảy đến mọi nơi trong chương trình.

```
10      k=1
20      gosub 100
30      if y > 120 goto 60
40      k = k + 1
50      goto 20
60      print k, y
70      stop
100     y = 3*k*k + 7*k - 3
110     return
```

Lệnh nhảy đến vị trí bất kỳ trong chương trình

Đoạn trình trên khó theo dõi, khó hiểu, dễ gây lỗi, khó sửa đổi.

Lịch sử ngắn gọn...

Kiểu lập trình rối rắm trên dẫn tới phong cách lập trình mới: lập trình cấu trúc, với các ngôn ngữ Algol, Pascal, C...

```
int func(int j)
{
   return (3*j*j + 7*j-3;
}

int main()
{
   int k = 1
   while (func(k) < 120)
      k++;
   printf("%d\t%d\n", k, func(k));
   return(0);
}
```

Đặc điểm của lập trình cấu trúc hay lập trình thủ tục (Procedural Programming - PP) là:

- Sử dụng các cấu trúc vòng lặp: for, while, repeat, do-while
- Chương trình là một chuỗi các hàm/ thủ tục
- Mã chương trình tập trung thể hiện thuật toán: làm như thế nào.

Hạn chế của lập trình thủ tục

- dữ liệu và phần xử lý tách biệt
- dữ liệu thụ động, xử lý chủ động
- không đảm bảo được tính nhất quán và các ràng buộc của dữ liệu
  - khó cấm mã ứng dụng sửa dữ liệu của thư viện
- khó bảo trì code
  - phần xử lý có thể nằm rải rác và phải hiểu rõ cấu trúc dữ liệu

```
struct Date
{
   int day;
   int month;
   int year;
};

void setDate(Date& date,
             int newDay, int newMonth, int newYear)
{
   date.day = newDay;
   ...
}
...
```

Chuyện gì xảy ra nếu các đối số **newDay, newMonth, newYear** tạo thành ngày tháng năm không hợp lệ?

Lập trình hướng đối tượng

- Lập trình hướng đối tượng cho phép khắc phục các hạn chế nói trên

```
class Date
{
public:
   void setDate(int newDay, int newMonth, int newYear);
   int getDay() { return day; }
   ...
private:
   int day;
   int month;
   int year;
};

void Date::setDate(int newDay, int newMonth, int newYear)
{
   //check validity of newDay, newMonth, newYear
   ...
   //set new values
   ...
}
```

Lập trình hướng đối tượng

Chú ý: Lập trình hướng đối tượng không tự dưng cho ta thiết kế chương trình tốt.

- Ví dụ: hai đoạn trình dưới đây không có gì khác nhau. Thậm chí đoạn chương trình hướng đối tượng còn tồi hơn.

```
struct Date
{
   int day;
   int month;
   int year;
};
```

```
class Date
{
public:
   int getDay { return day;}
   ...
   int setDay(int newDay) { day = newDay; }
   ...
private:
   int day;
   int month;
   int year;
};
```

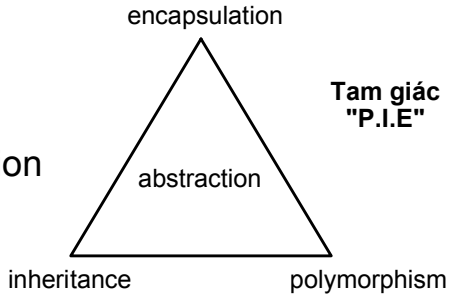

Lịch sử OOP

- Các ngôn ngữ lập trình hướng đối tượng không mới
  - Simula (1967) là ngôn ngữ đầu tiên, có lớp, thừa kế, liên kết động (hay còn gọi là hàm ảo)
- Nhưng các ngôn ngữ hướng đối tượng chậm hơn các ngôn ngữ thời kỳ đầu
  - nên chúng chỉ được dùng rộng rãi khi máy tính bắt đầu chạy nhanh (khoảng thời gian chiếc máy Pentium đầu tiên ra đời)
  - Lưu ý rằng biên dịch các chương trình hướng đối tượng cũng chậm

Hướng đối tượng là gì?

- Một số hệ thống "hướng đối tượng" thời kỳ đầu không có các lớp
  - chỉ có các "đối tượng" và các "thông điệp" (v.d. Hypertalk)
- Hiện giờ, đã có sự thống nhất rằng hướng đối tượng là:
  - lớp - class
  - thừa kế - inheritance và liên kết động - dynamic binding
- Một số đặc tính của lập trình hướng đối tượng có thể được thực hiện bằng C hoặc các ngôn ngữ lập trình thủ tục khác.
- Điểm khác biệt sự hỗ trợ và ép buộc ba khái niệm trên được cài hẳn vào trong ngôn ngữ.
- Mức độ hướng đối tượng của các ngôn ngữ không giống nhau
  - Eiffel (tuyệt đối), Java (rất cao), C++ (nửa nọ nửa kia)

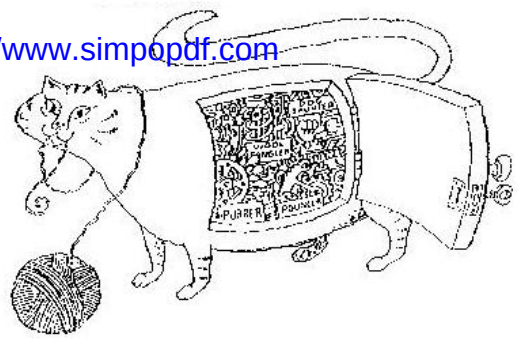
Các đặc điểm quan trọng của OO



Tam giác "P.I.E"

- Các lớp đối tượng - Classes
  - Khả năng lưu trạng thái - State retention
  - Định danh đối tượng - Object identity
  - Các thông điệp - Messages
- Đóng gói – Encapsulation
  - Che dấu thông tin - Information/implementation hiding
- Thừa kế - Inheritance
- Đa hình - Polymorphism
- Lập trình tổng quát - Genericity

Đóng gói
Che dấu thông tin



- Đóng gói: Nhóm những gì có liên quan với nhau vào làm một, để sau này có thể dùng một cái tên để gọi đến
  - Các hàm/ thủ tục đóng gói các câu lệnh
  - Các đối tượng đóng gói dữ liệu của chúng và các thủ tục có liên quan
- Che dấu thông tin: đóng gói để che một số thông tin và chi tiết cài đặt nội bộ để bên ngoài không nhìn thấy
  - mục tiêu là để khách hàng của ta (thường là các lập trình viên khác) coi các đối tượng của ta là các hộp đen

Đối tượng

- ***Lưu giữ trạng thái***: mỗi đối tượng có trạng thái (dữ liệu của nó) và các thao tác
  - có thể nói đối tượng có một dạng "ký ức" về quá khứ
  - các thao tác của đối tượng có thể sửa trạng thái của đối tượng đó.
- ***Định danh***: Mỗi đối tượng bất kể đang ở trạng thái nào đều có định danh và được đối xử như một thực thể riêng biệt.
  - mỗi đối tượng có một *handle* (trong C++ là địa chỉ)
  - hai đối tượng có thể có giá trị giống nhau nhưng *handle* khác nhau
  - ngôn ngữ Lisp phân biệt hai phép so sánh **`eq`** và **`equal`**

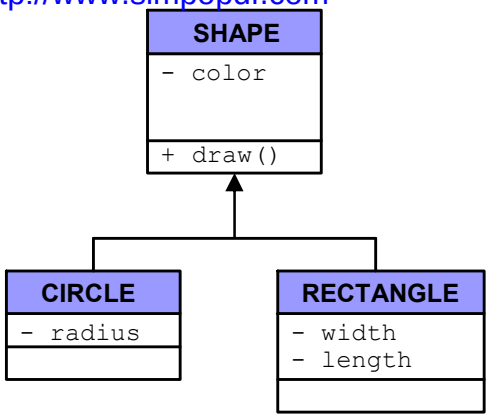
Đối tượng

- ***Thông điệp***: là phương tiện để một đối tượng A chuyển tới đối tượng B yêu cầu B thực hiện một trong số các thao tác của B.
- một thông điệp gồm 3 phần:
  - handle của đối tượng đích (đối tượng chủ)
  - tên thao tác cần thực hiện
  - các thông tin cần thiết khác (các đối số)
- thực ra là một lời gọi hàm với một đối số ẩn là đối tượng chủ
- tuy nhiên, khái niệm thông điệp có ý nghĩa lớn đối với OOP
  - dữ liệu trở nên chủ động
  - đối lập với quan điểm cũ về lập trình rằng
    - mọi hành động được điều khiển tập trung
    - dữ liệu luôn luôn bị động, các thủ tục thao tác dữ liệu

Lớp đối tượng - class

- Lớp: là khuôn mẫu để tạo các đối tượng (tạo các thể hiện). Mỗi đối tượng có cấu trúc và hành vi giống như lớp đối tượng mà nó được tạo từ đó
  - VD. Lớp VánCờ, các ván cờ cụ thể là các đối tượng VánCờ
- Lớp là cái ta thiết kế và lập trình
- Đối tượng là cái ta tạo (từ một lớp) tại thời gian chạy

Thừa kế



- là cơ chế cho phép một lớp **D** có được các thuộc tính và thao tác của lớp **C**, như thể các thuộc tính và thao tác đó đã được định nghĩa lại lớp **D**.
- cho phép các phần mềm sử dụng quan hệ "là"
- giúp ta thiết kế các dịch vụ tổng quát rồi chuyên môn hóa chúng

Đa hình

- Đa hình hàm - Functional polymorphism
  - cơ chế cho phép một tên thao tác hoặc thuộc tính có thể được định nghĩa tại nhiều lớp và có thể có nhiều cài đặt khác nhau tại mỗi lớp trong các lớp đó
    - v.d. lớp Date cài 2 phương thức setDate(),một nhận tham số là một đối tượng Date, phương thức kia nhận 3 tham số day, month, year.
- Đa hình đối tượng - Object polymorphism
  - các đối tượng thuộc các lớp khác nhau có khả năng hiểu cùng một thông điệp theo các cách khác nhau
    - vd. khi nhận được cùng một thông điệp draw(), các đối tượng Rectangle và Triangle hiểu và thực hiện các thao tác khác nhau

Lập trình tổng quát – genericity

- khả năng xây dựng một lớp **C** sao cho một hoặc nhiều lớp được sử dụng bên trong C sẽ được cung cấp tại thời gian chạy
  - vd. kiểu ngăn xếp tổng quát, có thể dùng để chứa các đối tượng Date, có thể dùng cho các string…
- Thực tế với C++:
  - cài đặt bằng khuôn mẫu – template hay các kiểu có tham số
  - Lựa chọn khuôn mẫu được quyết định tại thời điểm biên dịch của chương trình khách hàng - user
    - không phải tại thời điểm biên dịch của chương trình của người thiết kế - thư viện

Biên dịch
Biên dịch riêng rẽ

Lập trình hướng đối tượng

Biên dịch

- Chỉ hướng dẫn biên dịch trong môi trường Unix, sinh viên tự tìm hiểu đối với các môi trường lập trình khác.
- Ta sẽ sử dụng **g++** để dịch các chương trình C++.

```
g++ foo.cpp
```

- biên dịch **foo.cpp** cho kết quả là file chạy được **a.out**

```
g++ -o foo foo.cpp
```

- biên dịch **foo.cpp** cho kết quả là file chạy được **foo**

Biên dịch riêng rẽ

- VD: biên dịch chương trình **`program.cpp`** trong đó sử dụng một lớp có tên **`Picture`** để thao tác các hình vẽ
- Nên lưu phần cài đặt của lớp **`Picture`** trong một file riêng, chẳng hạn **`picture.cpp`**, để:
  - tạo thuận lợi cho việc sử dụng lớp này trong một ứng dụng khác
  - hai lập trình viên có thể dễ dàng cùng làm việc: một người cài đặt lớp **`Picture`**, người kia viết chương trình chính **`program.cpp`**
  - khi chương trình thay đổi, chỉ cần dịch lại file **`program.cpp`**, như vậy, quá trình biên dịch nhanh hơn. Đối với các chương trình lớn, điều này tạo sự khác biệt rất lớn.
- *Chú ý*: Theo thông lệ, các file chương trình C++ thường có kiểu mở rộng ".cpp", ".cc", ".C", hoặc ".cxx".

File header của lớp: ".h"

- Nếu ta không muốn người viết **`program.cpp`** biết chi tiết của lớp **`Picture`** (vì đó có thể là bí mật thương mại), ta cần tách giao diện của lớp (phần khai báo) ra khỏi cài đặt của lớp.
- Mặt khác, để có thể biên dịch được, chương trình chính **`program.cpp`** cũng cần biết về định nghĩa của lớp **`Picture`** và các phương thức của lớp đó.
- Giải pháp là mô tả lớp **`Picture`** tại hai file
  - picture.h các định nghĩa và khai báo (giao diện)
  - picture.cpp cài đặt

File header của lớp: ".h"

File chứa cài đặt

```
/* picture.h */
class Picture
{
  //...
  Picture* frame(const Picture&);
}
```

```
/* picture.cpp */
#include "picture.h"
Picture* Picture::frame (const Picture& x)
{
   //mã để đóng khung một hình ảnh
}
```

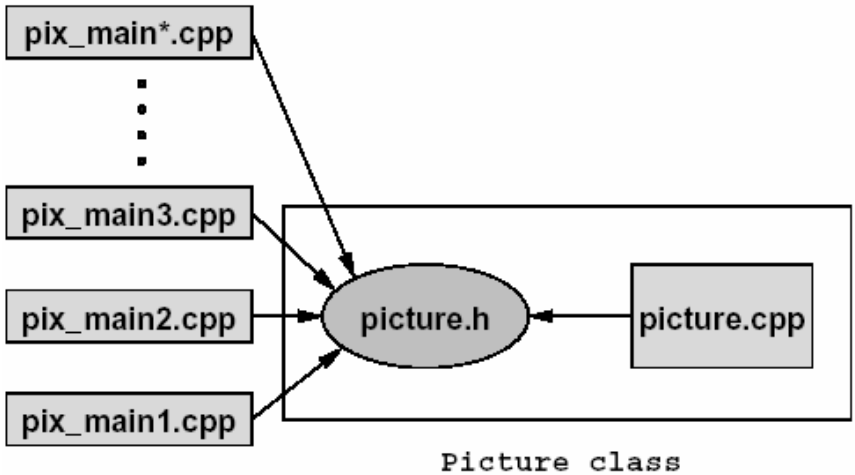
File header chứa giao diện

```
/* program.cpp */
#include "picture.h"
int main()
{
   //thao tác các hình ảnh
}
```

Client/user

File header của lớp: ".h"

Như vậy, ta có thể viết nhiều chương trình sử dụng lớp **`Picture`** có sẵn một cách tiện lợi



Picture class

Biên dịch riêng rẽ

- biên dịch chương trình như sau:

```
1> g++ -c picture.cpp
2> g++ -c program.cpp
3> g++ -o program program.o picture.o
```

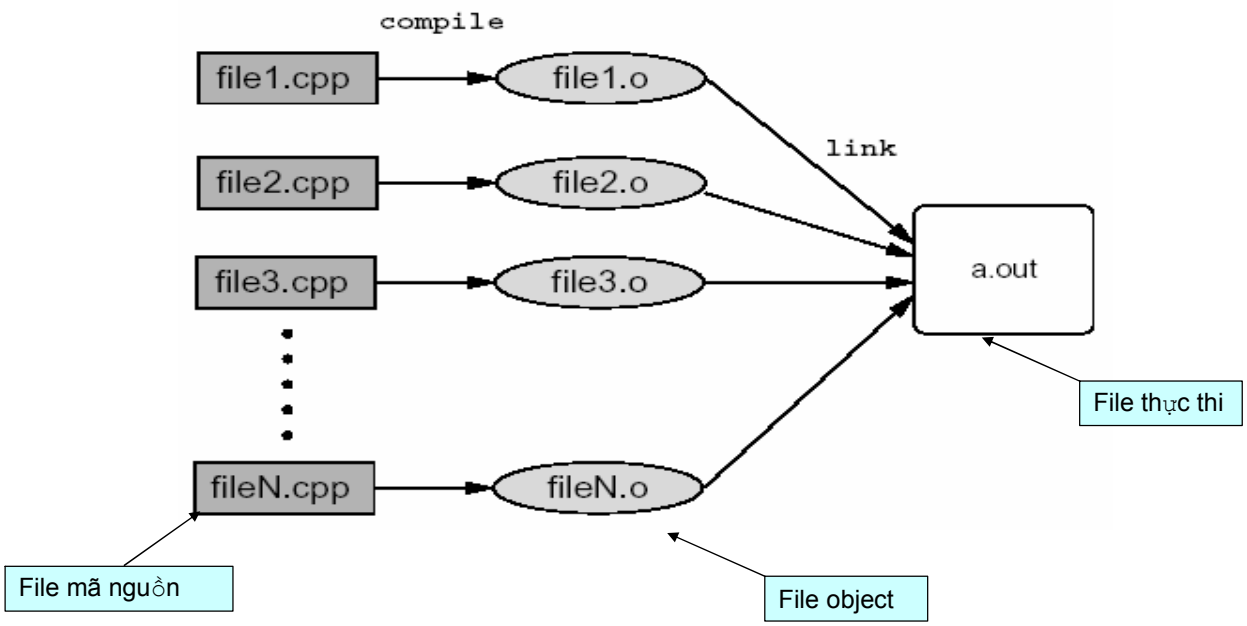
  - khóa chuyển **-c** tại dòng 1 và 2 tạo các object file **program.o** và **picture.o**. Dòng 3 tạo file chạy được có tên **program** với khóa chuyển **-o** bằng cách liên kết các object file với nhau.

- Hoặc

```
1> g++ -c picture.cpp
2> g++ -o program program.cpp picture.o
```

- Nếu **program.cpp** bị thay đổi nhưng **Picture** vẫn giữ nguyên, thì khi biên dịch lại, dòng 1 là không cần thiết.

Liên kết object file



Các định hướng tiền xử lý

- Các định hướng tiền xử lý là các lệnh có tính năng đặc biệt
- Được thực hiện bởi trình tiền xử lý trước khi mã nguồn được biên dịch.
- Trong C++, các định hướng tiền xử lý bắt đầu bằng một dấu #
- #include
- #define, #ifndef, #endif

Định hướng tiền xử lý #include

- Định hướng **`#include`** đọc nội dung của file được nêu tên vào nơi đặt định hướng

```
#include <standard_file.h>
#include "my_file.h"
```

- Cặp ngoặc nhọn < > dùng cho các file header chuẩn được tìm kiếm trong các thư mục thư viện chuẩn.
- Cặp dấu nháy " " dùng cho các file header của người dùng, sẽ được tìm kiếm trước hết trong thư mục hiện tại.
  - Có thể dùng khoá chuyển **`-I (g++ -I)`** để thay đổi đường dẫn tìm kiếm. Ví dụ:

```
g++ program.cpp -I/home/tmct/my_include/
```

trong đó, `/home/tmct/my_include/` là đường dẫn đầy đủ đến các thư mục chứa các file .h cần tìm

Các thư viện

- Để tạo một file thực thi (executable file), trình liên kết (linker) cần kết nối mã của các hàm được khai báo trong các file header chuẩn C++ (iostream.h, string.h, v.v..) Các đoạn mã tương ứng có thể được tìm thấy trong các thư viện chuẩn C++
- Một thư viện là một tập hợp các object file.
- Trình liên kết lựa chọn mã object từ các thư viện chứa định nghĩa các hàm được sử dụng trong các file chương trình và kết nối chúng vào file thực thi (executable file).
- Một số thư viện được trình liên kết C++ tự động sử dụng, chẳng hạn thư viện chuẩn C++. Các thư viện khác phải được chỉ rõ trong quá trình liên kết bằng khoá chuyển **`-l`**. Ví dụ, trong một số môi trường lập trình, cần lệnh sau để liên kết với thư viện toán học chuẩn **`libm.a`**

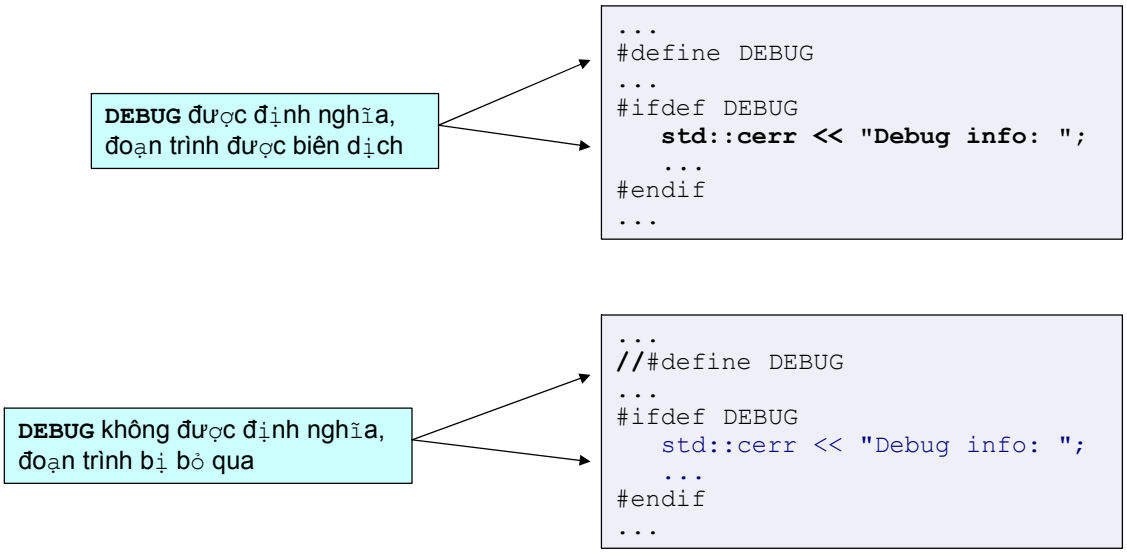
```
g++ -o myprog myprog.o -lm
```


#define, #ifdef, #ifndef, #endif

- **`#define`** định nghĩa một định danh
  - `#define MAX 100 // từ đây, MAX sẽ có giá trị 100`
  - `#define DEBUG // định nghĩa DEBUG`
- **`#ifdef`** định hướng điều kiện "nếu đã định nghĩa" (if defined)
  - `#ifdef DEBUG // nếu DEBUG đã được định nghĩa`
- **`#ifndef`** định hướng điều kiện "nếu chưa định nghĩa" (if not defined)
  - `#ifndef DEBUG // nếu DEBUG chưa được định nghĩa`
- **`#endif`** kết thúc khối mở đầu bằng **`#ifndef`** hoặc **`#ifdef`** gần nhất
  - nếu điều kiện tại định hướng mở đầu khối thỏa mãn thì biên dịch đoạn lệnh nằm trong khối

#define, #ifdef, #ifndef, #endif

- Ví dụ sử dụng



#define, #ifdef, #ifndef, #endif

```
/* program.h */
#include "b.h"
#include "c.h"
…
```

```
/* b.h */
#include "a.h"
#include "d.h"
…
```

```
/* c.h */
#include "a.h"
#include "e.h"
…
```

- Do các định hướng #include có thể lồng nhau, một file header có thể được kết nối hai lần. Hậu quả là
  - file đó được xử lý nhiều lần **à** tốn thời gian,
  - các hằng, macro, kiểu dữ liệu, nguyên mẫu hàm… được khai báo nhiều lần **à** lỗi biên dịch.
- Do vậy, ta cần các định hướng điều kiện (conditional directive) trong mọi file header

```
#ifndef PICTURE_H
#define PICTURE_H
// các khai báo đối tượng, định nghĩa lớp, hàm…
#endif      //PICTURE_H
```

Operator Overloading

Lập trình hướng đối tượng

Tài liệu đọc

- Eckel, Bruce. *Thinking in C++, 2nd Ed.* Vol. 1.
  - Chapter 8: Operator Overloading
- Dietel. *C++ How to Program, 4th Ed.*
  - Chapter 8: Operator Overloading

Operator Overloading

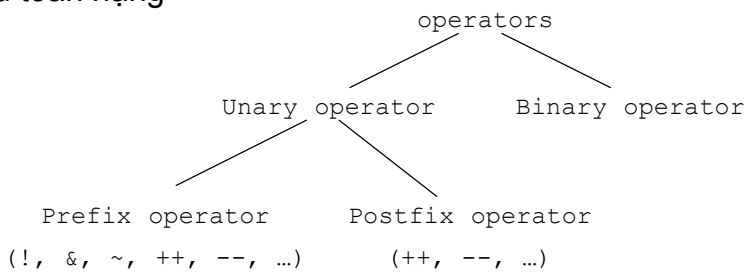
- Giới thiệu
- Các toán tử của C++
- Lý thuyết về operator overloading
- Cú pháp operator overloading
- Định nghĩa các toán tử thành viên
- Phép gán
- Định nghĩa các toán tử toàn cục
- Làm việc với tính đóng gói
- friend
- Tại sao sử dụng toán tử toàn cục
- Phép chèn ("<<")
- Phép tăng ("++")
- Các tham số và kiểu trả về
- Thành viên hay hàm toàn cục?
- Chuyển đổi kiểu tự động

Giới thiệu

- Các toán tử cho phép ta sử dụng cú pháp toán học đối với các kiểu dữ liệu của C++ thay vì gọi hàm (tuy bản chất vẫn là gọi hàm).
  - Ví dụ thay `a.set(b.add(c));` bằng `a = b + c;`
  - gần với kiểu trình bày mà con người quen dùng
  - đơn giản hóa mã chương trình
- C/C++ đã làm sẵn cho các kiểu cài sẵn (int, float…)
- Đối với các kiểu dữ liệu người dùng: C++ cho phép định nghĩa các toán tử cho các thao tác đối với các kiểu dữ liệu người dùng.
- Đó là operator overload
  - một toán tử có thể dùng cho nhiều kiểu dữ liệu
- Như vậy, ta có thể tạo các kiểu dữ liệu đóng gói hoàn chỉnh (fully-encapsulated) để kết hợp với ngôn ngữ như các kiểu dữ liệu cài sẵn

Các toán tử của C++

- Các toán tử được chia thành hai loại theo số toán hạng nó chấp nhận
  - **Toán tử đơn** nhận một toán hạng
  - **Toán tử đôi** nhận hai toán hạng
- Các toán tử đơn lại được chia thành hai loại
  - **Toán tử trước** đặt trước toán hạng
  - **Toán tử sau** đặt sau toán hạng



Các toán tử của C++.

- Một số toán tử đơn có thể được dùng làm cả toán tử trước và toán tử sau
  - Ví dụ phép tăng ("`++`") và phép giảm ("`--`")
- Một số toán tử có thể được dùng làm cả toán tử đơn và toán tử đôi
  - Ví dụ, "`*`" là toán tử đơn trong phép truy nhập con trỏ, là toán tử đôi trong phép nhân
- Toán tử chỉ mục ("`[…]`") là toán tử đôi, mặc dù một trong hai toán hạng nằm trong ngoặc
  - Phép lấy chỉ mục có dạng "`arg1[arg2]`"
- Các từ khoá "`new`" và "`delete`" cũng được coi là toán tử và có thể được định nghĩa lại (overload)

Các toán tử overload được

- Phần lớn các toán tử của C++ đều có thể overload được, bao gồm:

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


Các toán tử không overload được

- Các toán tử C++ không cho phép overload

```
.                   .*
::                  :?
typeid              sizeof
const_cast          dynamic_cast
reinterpret_cast    static_cast
```

Các hạn chế đối với việc overload toán tử

- Không thể tạo toán tử mới hoặc kết hợp các toán tử có sẵn theo kiểu mà trước đó chưa được định nghĩa
- Không thể thay đổi thứ tự ưu tiên của các toán tử
- Không thể tạo cú pháp mới cho toán tử
- Không thể định nghĩa lại một định nghĩa có sẵn của một toán tử
  - Ví dụ: không thể thay đổi định nghĩa có sẵn của phép ("+") đối với hai số kiểu int
  - Như vậy, khi tạo định nghĩa mới cho một toán tử, ít nhất một trong số các tham số (toán hạng) của toán tử đó phải là một kiểu dữ liệu người dùng

Lưu ý khi định nghĩa lại toán tử

- Tôn trọng ý nghĩa của toán tử gốc, cung cấp chức năng mà người dùng mong đợi/chấp nhận
  - không ai nghĩ rằng phép "+" sẽ in kết quả ra màn hình hoặc thực hiện phép chia
  - Sử dụng "+" để nối hai xâu có thể hơi lạ đối với người chỉ quen dùng "+" cho các số, nhưng nó vẫn tuân theo khái niệm chung của phép cộng
- Nên cho kiểu trả về của toán tử khớp với định nghĩa cho các kiểu cài sẵn
  - không nên trả về giá trị int từ phép so sánh == của mảng số, nên trả về bool
- Cố gắng tái sử dụng mã nguồn một cách tối đa
  - Ta sẽ thường xuyên định nghĩa các toán tử sử dụng các định nghĩa có sẵn

Cú pháp của Operator Overloading

```
class MyNumber {
   public:
      MyNumber(int value = 0);
      ~MyNumber();
      …
   private:
      int value;
};
```

- Ta sẽ sử dụng ví dụ trên:
  - Đây là lớp bọc ngoài (wrapper class) cho kiểu int
- Ta sẽ overload các toán tử để cho phép cộng, trừ, so sánh, … các đối tượng của lớp

Cú pháp của Operator Overloading

- Khai báo và định nghĩa toán tử thực chất không khác với việc khai báo và định nghĩa nghĩa một loại hàm bất kỳ nào khác
- sử dụng tên hàm là "operator@" cho toán tử "@"
  - để overload phép "+", ta dùng tên hàm "operator+"
- Số lượng tham số tại khai báo phụ thuộc hai yếu tố:
  - Toán tử là toán tử đơn hay đôi
  - Toán tử được khai báo là hàm toàn cục hay phương thức của lớp

```
aa@bb    è aa.operator@(bb)  hoặc operator@(aa,bb)
@aa      è aa.operator@( )   hoặc operator@(aa)
aa@      è aa.operator@(int) hoặc operator@(aa,int)
```

là phương thức của lớp

là hàm toàn cục

Cú pháp của Operator Overloading

- Ví dụ: Sử dụng toán tử "+" để cộng hai đối tượng **`MyNumber`** và trả về kết quả là một **`MyNumber`**

```
MyNumber x(5);
MyNumber y(10);
...
z = x + y;
```

- Ta có thể khai báo hàm toàn cục sau

```
const MyNumber operator+(const MyNumber& num1, const MyNumber& num2);
```

  - "`x+y`" sẽ được hiểu là "`operator+(x,y)`"
  - dùng từ khoá const để đảm bảo các toán hạng gốc không bị thay đổi

- Hoặc khai báo toán tử dưới dạng thành viên của **`MyNumber`**:

```
const MyNumber operator+(const MyNumber& num);
```

  - đối tượng chủ của phương thức được hiểu là toán hạng thứ nhất của toán tử.
  - "`x+y`" sẽ được hiểu là "`x.operator+(y)`"

Cú pháp của Operator Overloading

- Sau khi đã khai báo toán tử bị overload (là phương thức hay hàm toàn cục), cú pháp định nghĩa không có gì khó
  - Định nghĩa toán tử dạng phương thức không khác với định nghĩa phương thức bất kỳ khác
  - Định nghĩa toán tử dạng hàm toàn cục không khác với định nghĩa hàm toàn cục bất kỳ khác
- Tuy nhiên, có một số vấn đề liên quan đến hướng đối tượng (đặc biệt là tính đóng gói) mà ta cần xem xét

Toán tử là hàm thành viên

- Để bắt đầu, xét định nghĩa toán tử bên trong giới hạn lớp
- Ví dụ: định nghĩa phép cộng:

```
const MyNumber MyNumber::operator+(const MyNumber& num) {
   MyNumber result(this->value + num.value);
   return result;
}
```

  - Constructor cho **`MyNumber`** và định nghĩa có sẵn của phép cộng cho **`int`** được tái sử dụng
  - Tạo một đối tượng **`MyNumber`** sử dụng constructor với giá trị là tổng hai giá trị thành viên của các tham số tính bằng phép cộng đã có sẵn cho kiểu **`int`**.
- Ta sẽ lấy một ví dụ thường gặp khác là phép gán

Phép gán "="

- Một trong những toán tử hay được overload nhất
  - Cho phép gán cho đối tượng này một giá trị dựa trên một đối tượng khác
  - Copy constructor cũng thực hiện việc tương tự, cho nên, định nghĩa toán tử gán gần như giống hệt định nghĩa của copy constructor
- Ta có thể khai báo phép gán cho lớp **`MyNumber`** như sau:

```
const MyNumber& operator=(const MyNumber& num);
```

  - Phép gán nên luôn luôn trả về một tham chiếu tới đối tượng đích (đối tượng được gán trị cho)
  - Tham chiếu được trả về phải là **`const`** để tránh trường hợp **`a`** bị thay đổi bằng lệnh **`"(a = b) = c;"`** (lệnh đó không tương thích với định nghĩa gốc của phép gán)

Phép gán "="

```
const MyNumber& MyNumber::operator=(const MyNumber& num) {
   if (this != &num) {
      this->value = num.value;
   }
   return *this;
}
```

- Định nghĩa trên có thể dùng cho phép gán
  - Lệnh **`if`** dùng để ngăn chặn các vấn đề có thể nảy sinh khi một đối tượng được gán cho chính nó (thí dụ khi sử dụng bộ nhớ động để lưu trữ các thành viên)
  - Ngay cả khi gán một đối tượng cho chính nó là an toàn, lệnh **`if`** trên đảm bảo không thực hiện các công việc thừa khi gán

Phép gán "="

- Khi nói về copy constructor, ta đã biết rằng C++ luôn cung cấp một copy constructor mặc định, nhưng nó chỉ thực hiện sao chép đơn giản (sao chép nông)
- Đối với phép gán cũng vậy
- Vậy, ta chỉ cần định nghĩa lại phép gán nếu:
  - Ta cần thực hiện phép gán giữa các đối tượng
  - Phép gán nông (memberwise assignment) không đủ dùng vì
    - ta cần sao chép sâu - chẳng hạn sử dụng bộ nhớ động
    - Khi sao chép đòi hỏi cả tính toán - chẳng hạn gán một số hiệu có giá trị duy nhất hoặc tăng số đếm

Toán tử là hàm toàn cục

- Quay lại với ví dụ về phép cộng cho **`MyNumber`**, ta có thể khai báo hàm định nghĩa phép cộng tại mức toàn cục:

```
const MyNumber operator+(const MyNumber& num1,
                         const MyNumber& num2);
```

- Khi đó, ta có thể định nghĩa toán tử đó như sau:

```
const MyNumber operator+(const MyNumber& num1,
                         const MyNumber& num2) {
   MyNumber result(num1.value + num2.value);
   return result;
}
```

truy nhập các thành viên private **`value`**

- Ở đây có vấn đề....

Làm việc với tính đóng gói

- Rắc rối: hàm toàn cục muốn truy nhập thành viên private của lớp
  - thông thường: không được quyền
  - đôi khi bắt buộc phải overload bằng hàm toàn cục
  - không thể hy sinh tính đóng gói của hướng đối tượng để chuyển các thành viên private thành public
- Giải pháp là dạng cuối cùng của quyền truy nhập: **`friend`**

friend

- Khái niệm **friend** cho phép một lớp cấp quyền truy nhập tới các phần nội bộ của lớp đó cho một số cấu trúc được chọn
- C++ có 3 kiểu `friend`
  - Hàm `friend` (trong đó có các toán tử được overload)
  - Lớp `friend`
  - Phương thức `friend` (hàm thành viên)
- Các tính chất của quan hệ **friend**
  - Phải được cho, không tự nhận
  - Không đối xứng
  - Không bắc cầu

Hàm friend

- Khi khai báo một hàm bên ngoài là `friend` của một lớp, hàm đó được cấp quyền truy nhập *tương đương* quyền của các phương thức của lớp đó
  - Như vậy, một hàm `friend` có thể truy nhập cả các thành viên `private` và `protected` của lớp đó
- Để khai báo một hàm là `friend` của một lớp, ta phải khai báo hàm đó bên trong khai báo lớp và đặt từ khoá `friend` lên đầu khai báo. Ví dụ:

```
class MyNumber {
   public:
      MyNumber(int value = 0);
      ~MyNumber();
      ...
      friend const MyNumber operator+(const MyNumber& num1,
                                      const MyNumber& num2);
   ...
};
```

Hàm **friend**

- Lưu ý: tuy khai báo của hàm friend được đặt trong khai báo lớp và hàm đó có quyền truy nhập ngang với các phương thức của lớp, hàm đó *không phải* phương thức của lớp
- Không cần thêm sửa đổi gì cho định nghĩa của hàm đã được khai báo là **friend**.
  - Định nghĩa trước của phép cộng vẫn giữ nguyên

```
const MyNumber operator+(const MyNumber& num1,
                         const MyNumber& num2) {
   MyNumber result(num1.value + num2.value);
   return result;
}
```


Tại sao dùng toán tử toàn cục?

- Đối với toán tử được khai báo là phương thức của lớp, đối tượng chủ (xác định bởi con trỏ **this**) luôn được hiểu là toán hạng đầu tiên (trái nhất) của phép toán.
  - Nếu muốn dùng cách này, ta phải được quyền bổ sung phương thức vào định nghĩa của lớp/kiểu của toán hạng trái
- Không phải lúc nào cũng có thể overload toán tử bằng phương thức
  - phép cộng giữa `MyNumber` và `int` cần cả hai cách

    `MyNumber + int` và `int + MyNumber`
  - **cout** `<< obj;`
  - không thể sửa định nghĩa kiểu **int** hay kiểu của **cout**
  - lựa chọn duy nhất: overload toán tử bằng hàm toàn cục

Toán tử chèn (‘<<‘)

- prototype như thế nào? xét ví dụ:
  - **`cout << num;`** // num là đối tượng thuộc lớp MyNumber
- Toán hạng trái **`cout`** thuộc lớp **`ostream`**, không thể sửa định nghĩa lớp này nên ta overload bằng hàm toàn cục
- Tham số thứ nhất : tham chiếu tới **`ostream`**
- Tham số thứ hai : kiểu **`MyNumber`**,
  - **`const`** (do không có lý do gì để sửa đối tượng được in ra)
- giá trị trả về: tham chiếu tới **`ostream`**
  (để thực hiện được **`cout << num1 << num2;`**)

- Kết luận:
  `ostream& operator<<(ostream& os, const MyNumber& num)`

Toán tử chèn ("<<")

- Khai báo toán tử được overload là **`friend`** của lớp **`MyNumber`**

```
class MyNumber {
   public:
      MyNumber(int value = 0);
      ~MyNumber();
      ...
      friend ostream& operator<<( ostream& os, const MyNumber& num);
      ...
};
```

Toán tử chèn ("<<")

- Định nghĩa toán tử

```
ostream& operator<<(ostream& os, const MyNumber& num) {
   os << num.value;    // Use version of insertion operator defined for int
   return os;          // Return a reference to the modified stream
};
```

- Tuỳ theo độ phức tạp của lớp được chuyển sang chuỗi ký tự, định nghĩa của toán tử này có thể dài hơn
- Toán tử tách (">>") được overload tương tự, tuy nhiên, định nghĩa thường phức tạp hơn
  - do có thể phải xử lý input để kiểm tra tính hợp lệ tuỳ theo cách ta quy định như thế nào khi in một đối tượng ra thành một chuỗi ký tự

Phép tăng ("++")

@aa è aa.operator@( ) hoặc **operator@(aa)**
aa@ è aa.operator@(int) hoặc **operator@(aa,int)**

- Khi gặp phép tăng trong một lệnh, trình biên dịch sẽ sinh một trong 4 lời gọi hàm trên, tuỳ theo toán tử là toán tử trước (prefix) hay toán tử sau (postfix), là phương thức hay hàm toàn cục.
- Giả sử ta overload phép tăng dưới dạng phương thức của **`MyNumber`** và overload cả hai dạng đặt trước và đặt sau
  - Nếu gặp biểu thức dạng "**`x++`**", trình biên dịch sẽ sinh lời gọi **`MyNumber::operator++()`**
  - Nếu gặp biểu thức dạng "**`++x`**", trình biên dịch sẽ sinh lời gọi **`MyNumber::operator++(int)`**
  - Tham số **`int`** chỉ dành để phân biệt danh sách tham số của hai dạng prefix và postfix

Phép tăng ("++")

- giá trị trả về
  - tăng trước **++num**
    - trả về tham chiếu (**MyNumber &**)
    - giá trị trái - lvalue (có thể được gán trị)
  - tăng sau **num++**
    - trả về giá trị (giá trị cũ trước khi tăng)
    - trả về đối tượng tạm thời chứa giá trị cũ.
    - giá trị phải - rvalue (không thể làm đích của phép gán)
- prototype
  - tăng trước: **MyNumber& MyNumber::operator++()**
  - tăng sau: **const MyNumber MyNumber::operator++(int)**

Phép tăng ("++")

- Nhớ lại rằng phép tăng trước tăng giá trị trước khi trả kết quả, trong khi phép tăng sau trả lại giá trị trước khi tăng
- Ta định nghĩa từng phiên bản của phép tăng như sau:

```
MyNumber& MyNumber::operator++() { // Prefix
   this->value++;          // Increment value
   return *this;           // Return current MyNumber
}

const MyNumber MyNumber::operator++(int) { // Postfix
   MyNumber before(this->value); // Create temporary MyNumber
                                 // with current value
   this->value++;          // Increment value
   return before;          // Return MyNumber before increment
}
```

before là một đối tượng địa phương của phương thức và sẽ chấm dứt tồn tại khi lời gọi hàm kết thúc
Khi đó, tham chiếu tới nó trở thành bất hợp lệ

Không thể trả về tham chiếu

Tham số và kiểu trả về

- Cũng như khi overload các hàm khác, khi overload một toán tử, ta cũng có nhiều lựa chọn về việc truyền tham số và kiểu trả về
  - chỉ có hạn chế rằng ít nhất một trong các tham số phải thuộc kiểu người dùng tự định nghĩa
- Ở đây, ta có một số lời khuyên về các lựa chọn

Tham số và kiểu trả về

- Các toán hạng:
  - Nên sử dụng tham chiếu mỗi khi có thể (đặc biệt là khi làm việc với các đối tượng lớn)
  - Luôn luôn sử dụng tham số là hằng tham chiếu khi đối số sẽ không bị sửa đổi

```
bool String::operator==(const String &right) const
```

    - Đối với các toán tử là phương thức, điều đó có nghĩa ta nên khai báo toán tử là hằng thành viên nếu toán hạng đầu tiên sẽ không bị sửa đổi
    - Phần lớn các toán tử (tính toán và so sánh) không sửa đổi các toán hạng của nó, do đó ta sẽ rất hay dùng đến hằng tham chiếu

Tham số và kiểu trả về

- Giá trị trả về
  - không có hạn chế về kiểu trả về đối với toán tử được overload, nhưng nên cố gắng tuân theo tinh thần của các cài đặt có sẵn của toán tử
    - Ví dụ, các phép so sánh (==, !=…) thường trả về giá trị kiểu **`bool`**, nên các phiên bản overload cũng nên trả về **`bool`**
  - là tham chiếu (tới đối tượng kết quả hoặc một trong các toán hạng) hay một vùng lưu trữ mới
  - Hằng hay không phải hằng

Tham số và kiểu trả về

- Giá trị trả về …
  - Các toán tử sinh một giá trị mới cần có kết quả trả về là một giá trị (thay vì tham chiếu), và là const (để đảm bảo kết quả đó không thể bị sửa đổi như một l-value)
    - Hầu hết các phép toán số học đều sinh giá trị mới
    - ta đã thấy, các phép tăng sau, giảm sau tuân theo hướng dẫn trên
  - Các toán tử trả về một tham chiếu tới đối tượng ban đầu (đã bị sửa đổi), chẳng hạn phép gán và phép tăng trước, nên trả về tham chiếu không phải là hằng
    - để kết quả có thể được tiếp tục sửa đổi tại các thao tác tiếp theo

```
const MyNumber MyNumber::operator+(const MyNumber& right) const

MyNumber& MyNumber::operator+=(const MyNumber& right)
```

Tham số và kiểu trả về

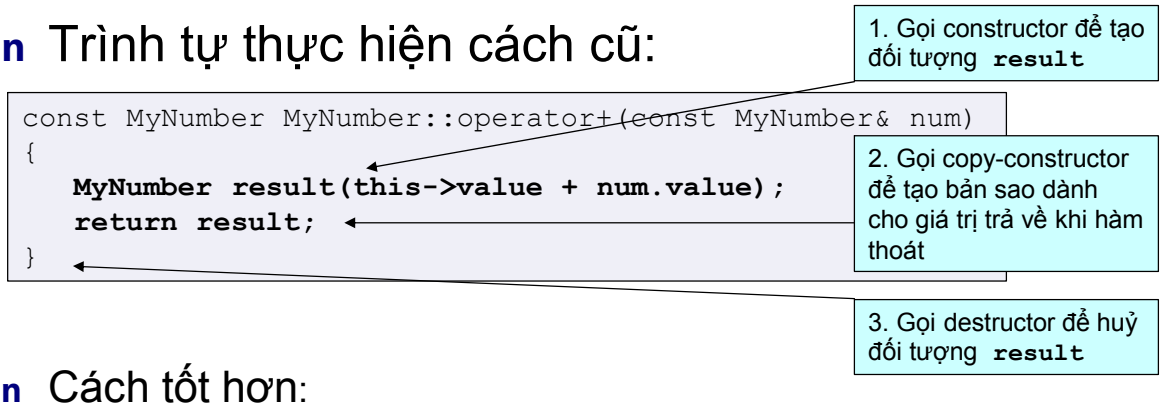
- Lời khuyên cuối cùng:
  - Xem lại cách ta đã dùng để trả về kết quả của toán tử:

```
const MyNumber MyNumber::operator+(const MyNumber& num)
{
   MyNumber result(this->value + num.value);
   return result;
}
```

  - Cách trên không sai, nhưng C++ cung cấp một cách hiệu quả hơn

Tham số và kiểu trả về

- Trình tự thực hiện cách cũ:



- Cách tốt hơn:

```
const MyNumber MyNumber::operator+(const MyNumber& num)
{
   return MyNumber(this->value + num.value);
}
```

Tham số và kiểu trả về

```
return MyNumber(this->value + num.value);
```

- Cú pháp của ví dụ trước tạo một đối tượng tạm thời (temporary object)
- Khi trình biên dịch gặp đoạn mã này, nó hiểu đối tượng được tạo chỉ nhằm mục đích làm giá trị trả về, nên nó tạo thẳng một đối tượng bên ngoài (để trả về) - bỏ qua việc tạo và huỷ đối tượng bên trong lời gọi hàm
- Vậy, chỉ có một lời gọi duy nhất đến constructor của `MyNumber` (không phải copy-constructor) thay vì dãy lời gọi trước
- Quá trình này được gọi là tối ưu hoá giá trị trả về
- Ghi nhớ rằng quá trình này không chỉ áp dụng được đối với các toán tử. Ta nên sử dụng mỗi khi tạo một đối tượng chỉ để trả về

Phương thức hay hàm toàn cục?

Khi lựa chọn overload toán tử tại lớp hoặc tại mức toàn cục, trường hợp nào nên chọn kiểu nào?

- Một số toán tử ***phải*** là thành viên:
  - "=", "`[]`", "`()`", và "`->`", "`->*`" phải là thành viên
- Các toán tử đơn ***nên*** là thành viên(để đảm bảo tính đóng gói)
- Khi toán hạng trái có thể được gán trị, toán tử ***nên*** là thành viên ("+=", "-=", "/=",…)
- Mọi toán tử đôi khác không nên là thành viên
  - Trừ khi ta muốn các toán tử này là hàm ảo trong cây thừa kế

Phương thức hay hàm toàn cục?

- Các toán tử là thành viên nên là ***hằng hàm*** mỗi khi có thể
  - Điều này cho phép tính mềm dẻo khi làm việc với hằng
- Nếu ta cảm thấy không nên cho phép sử dụng một toán tử nào đó với lớp của ta (và không muốn các nhà thiết kế khác định nghĩa nó),
  ta khai báo toán tử đó dạng `private` (và không cài đặt toán tử đó)

Ví du: Kiểu `Date`

- **`Date`** class
  - Overload phép tăng
    - thay đổi ngày, tháng, năm
  - Overloaded **+=**
  - hàm kiểm tra năm nhuận
  - hàm kiểm tra xem một ngày có phải cuối tháng

```
1  // Fig. 8.10: date1.h
2  // Date class definition
3  #ifndef DATE1_H
4  #define DATE1_H
5  #include <iostream>
6
7  using std::ostream;
8
9  class Date {
10    friend ostream &operator<<( ostream &, const Date & );
11
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // constructor
14    void setDate( int, int, int ); // set the date
15
16    Date &operator++();           // prei
17    Date operator++( int );       // postincrement operator
18
19    const Date &operator+=( int ); // add days, modify object
20
21    bool leapYear( int ) const;   // is this a leap year?
22    bool endOfMonth( int ) const; // is this end of month?
```

Lưu ý sự khác nhau giữa tăng trước và tăng sau.

```
23
24    private:
25    int month;
26    int day;
27    int year;
28
29    static const int days[];      // array of days per month
30    void helpIncrement();         // utility function
31
32    }; // end class Date
33
34    #endif
```

```
1  // Fig. 8.11: date1.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include "date1.h"
5
6  // initialize static member at file scope;
7  // one class-wide copy
8  const int Date::days[] =
9     { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
10
11 // Date constructor
12 Date::Date( int m, int d, int y )
13 {
14    setDate( m, d, y );
15
16 } // end Date constructor
17
18 // set month, day and year
19 void Date::setDate( int mm, int dd, int yy )
20 {
21    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
22    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
23
```

```
24    // test for a leap year
25    if ( month == 2 && leapYear( year ) )
26       day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
27    else
28       day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
29
30    } // end function setDate
31
32 // overloaded preincrement operator
33 Date &Date::operator++()
34 {
35    helpIncrement();
36
37    return *this;  // reference return to create an lvalue
38
39 } // end function operator++
40
41 // overloaded postincrement operator
42 // integer parameter does not have a
43 Date Date::operator++( int )
44 {
45    Date temp = *this;  // hold current state of object
46    helpIncrement();
47
48    // return unincremented, saved, temporary object
49    return temp;   // value return; not a reference return
50
51 } // end function operator++
```

Lưu ý: biến int không có tên.

Phép tăng sau sửa giá trị của đối tượng và trả về một bản sao của đối tượng ban đầu. Không trả về tham số tới biến tạm vì đó là một biến địa phương và sẽ bị hủy.

```
53 // add specified number of days to date
54 Date &Date::operator+=( int additionalDays )
55 {
56    for ( int i = 0; i < additionalDays; i++ )
57       helpIncrement();
58
59    return *this;    // enables cascading
60
61 } // end function operator+=
62
```

```
63 // if the year is a leap year, return true;
64 // otherwise, return false
65 bool Date::leapYear( int testYear ) const
66 {
67    if ( testYear % 400 == 0 ||
68       ( testYear % 100 != 0 && testYear % 4 == 0 ) )
69       return true;   // a leap year
70    else
71       return false;  // not a leap year
72
73 } // end function leapYear
74
75    // determine whether the day is the last day of the month
76    bool Date::endOfMonth( int testDay ) const
77    {
78       if ( month == 2 && leapYear( year ) )
79          return testDay == 29; // last day of Feb. in leap year
80       else
81          return testDay == days[ month ];
82
83    } // end function endOfMonth
84
```

```
85    // function to help increment the date
86    void Date::helpIncrement()
87    {
88       // day is not end of month
89       if ( !endOfMonth( day ) )
90          ++day;
91
92       else
93
94          // day is end of month and month < 12
95          if ( month < 12 ) {
96             ++month;
97             day = 1;
98          }
99
100         // last day of year
101         else {
102            ++year;
103            month = 1;
104            day = 1;
105         }
106
107   } // end function helpIncrement
108
```

```
109   // overloaded output operator
110   ostream &operator<<( ostream &output, const Date &d )
111   {
112      static char *monthName[ 13 ] = { "", "January",
113         "February", "March", "April", "May", "June",
114         "July", "August", "September", "October",
115         "November", "December" };
116
117      output << monthName[ d.month ] << ' '
118             << d.day << ", " << d.year;
119
120      return output;   // enables cascading
121
122   } // end function operator<<
```

```
1     // Fig. 8.12: fig08_12.cpp
2     // Merge and Split Unregistered
3     #include <iostream>
4
5     using std::cout;
6     using std::endl;
7
8     #include "date1.h"  // Date class definition
9
10    int main()
11    {
12       Date d1;  // defaults to January 1, 1900
13       Date d2( 12, 27, 1992 );
14       Date d3( 0, 99, 8045 );  // invalid date
15
16       cout << "d1 is " << d1 << "\nd2 is " << d2
17            << "\nd3 is " << d3;
18
19       cout << "\n\nd2 += 7 is " << ( d2 += 7 );
20
21       d3.setDate( 2, 28, 1992 );
22       cout << "\n\n  d3 is " << d3;
23       cout << "\n++d3 is " << ++d3;
24
25       Date d4( 7, 13, 2002 );
```

```
26
27       cout << "\n\nTesting the preincrement operator:\n"
28            << "  d4 is " << d4 << '\n';
29       cout << "++d4 is " << ++d4 << '\n';
30       cout << "  d4 is " << d4;
31
32       cout << "\n\nTesting the postincrement operator:\n"
33            << "  d4 is " << d4 << '\n';
34       cout << "d4++ is " << d4++ << '\n';
35       cout << "  d4 is " << d4 << endl;
36
37       return 0;
38
39    } // end main
```

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postincrement operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
  d4 is July 15, 2002
```


Đổi kiểu tự động
Automatic type conversion

- Ta đã nói về nhiều công việc mà đôi khi C++ ngầm thực hiện
- Một trong những việc đó là thực hiện đổi kiểu tự động
  - Ví dụ, nếu ta gọi một hàm đòi hỏi một kiểu dữ liệu có sẵn nhưng ta cho hàm một kiểu hơi khác, C++ đôi khi sẽ tự đổi kiểu tham số truyền vào

```
void foo(double x);
...
foo(2);
```

- Đối với các lớp dữ liệu người dùng, C++ cũng cung cấp khả năng định nghĩa các phép chuyển đổi tự động

Đổi kiểu tự động

- Có hai cách định nghĩa phép đổi kiểu tự động
  - Khai báo và định nghĩa một constructor lấy một tham số duy nhất

Đổi từ **int** sang **MyNumber**

```
class MyNumber {
   public:
      MyNumber(int value = 0);
   ...
```

  - Khai báo và định nghĩa một toán tử đặc biệt (special overloaded operator)

Đổi từ **MyOtherNumber** sang **MyNumber**

```
class MyOtherNumber {
   public:
      MyOtherNumber(…);
      …
      operator MyNumber() const;
```


Đổi kiểu tự động - constructor

- Cách 1: định nghĩa một constructor lấy một tham số duy nhất (giá trị hoặc tham chiếu) thuộc một kiểu nào đó, constructor đó sẽ được ngầm gọi khi cần đổi kiểu.

```
class MyNumber {
   public:
      MyNumber(int value = 0);
   ...
```

```
void foo(MyNumber num) {
…
}
```

```
...
int x = 5;
foo(x); // Automatically converts the argument
...
```

Đổi kiểu tự động - constructor

- Nếu ta muốn ngăn chặn đổi kiểu tự động kiểu này, ta có thể dùng từ khoá **`explicit`** khi khai báo constructor

```
class MyNumber {
   public:
      explicit MyNumber(int value = 0);
   ...
```

- Kết quả là việc đổi kiểu chỉ xảy ra khi được gọi một cách tường minh

```
int x = 5;

Foo(x); // This will generate an error
Foo(MyNumber(x)); // Explicit conversion - ok
```


Đổi kiểu tự động – toán tử

- Cách 2: định nghĩa phép đổi kiểu sử dụng một toán tử đặc biệt
  - Tạo một toán tử là thành viên của lớp cần đổi, toán tử này sẽ chuyển thành viên của lớp này sang kiểu mong muốn
  - Toán tử này hơi đặc biệt: không có kiểu trả về
    - Thực ra, tên của toán tử chính là kiểu trả về

Đổi kiểu tự động – toán tử

- Giả sử cần một phép chuyển đổi tự động từ kiểu MyOtherNumber sang kiểu **`MyNumber`**
  - khai báo một toán tử có tên **`MyNumber`** :

```
class MyOtherNumber {
   public:
      MyOtherNumber(…);
      …
      operator MyNumber() const;
```

  - Thân toán tử chỉ cần tạo và trả về một thể hiện của lớp đích
    - Nên là hàm inline

```
operator MyNumber() const { return MyNumber(…); }
```

Các tham số cần thiết để tạo thể hiện mới

Đổi kiểu tự động - đổi kiểu ẩn

```
class Fee {
   public:
      Fee(int) {}
};

class Fo {
      int i;
   public:
      Fo(int x = 0) : i(x) {}
      operator Fee() const { return Fee(i); }
};

int main() {
   Fo fo;
   Fee fee = fo;
} ///:~
```

3. Không có copy constructor để tạo **`Fee`** từ **`Fee`**, nhưng lại có copy constructor mặc định do trình biên dịch tự sinh (phép gán mặc định)

2. Tuy nhiên, có phép chuyển đổi tự động từ **`Fo`** sang **`Fee`**

1. Không có constructor tạo **`Fee`** từ **`Fo`**

Đổi kiểu tự động

- Nói chung, nên tránh đổi kiểu tự động, do chúng có thể dẫn đến các kết quả người dùng không mong đợi
  - Nếu có constructor lấy đúng một tham số khác kiểu lớp chủ, hiện tượng đổi kiểu tự động sẽ xảy ra
  - Chính sách an toàn nhất là khai báo các constructor là **`explicit`**, trừ khi kết quả của chúng đúng là cái mà người dùng mong đợi

Template

Lập trình hướng đối tượng

Tài liệu đọc

- Eckel, Bruce. *Thinking in C++, 2nd Ed.* Vol. 1.
  - Chapter 16: Introduction to Templates
- Dietel. *C++ How to Program, 4th Ed.*
  - Chapter 11: Templates

Giới thiệu về khuôn mẫu

- Giới thiệu
- Lập trình tổng quát (generic programming)
- Lập trình tổng quát trong C
- C++ template
- Khuôn mẫu hàm
- Khuôn mẫu lớp
- Các tham số template khác
- Template sử dụng template

Giới thiệu

- Trong suốt khoá học, ta đã nói về phương pháp hướng đối tượng như là một cơ chế cho việc trừu tượng hoá
  - Nhóm các đối tượng có cùng tập hành vi và thuộc tính lại với nhau, và tương tác với chúng theo cùng kiểu
- Với đa hình và thừa kế, ta có thể biểu diễn mối quan hệ giữa các lớp đối tượng tương tự và tương tác với chúng một cách thống nhất
  - Ta có thể lệnh cho một chiếc xe chạy bằng máy thực hiện việc "drive", và hành vi thích hợp sẽ được gọi tuỳ theo loại xe đang nói đến
- Để kết thúc, ta sẽ giới thiệu một kiểu trừu tượng hoá khác

Lập trình tổng quát

- Ta đã tập trung vào việc trừu tượng hoá các chi tiết để tạo được các lớp đối tượng - gồm những thứ có cùng tập thuộc tính và hành vi
- Mọi ý tưởng được bàn đến đều xoay quanh đối tượng - biểu diễn khái niệm hoặc biểu diễn trong C++
- Bây giờ, ta nói về một phương pháp lập trình mà trừu tượng hoá chính các lớp đối tượng.
- Đây là mức tiếp theo trong quá trình logic phát triển của trừu tượng hoá.

Lập trình tổng quát

- Lập trình tổng quát là phương pháp lập trình độc lập với chi tiết biểu diễn dữ liệu
  - Tư tưởng là ta định nghĩa một khái niệm không phụ thuộc một biểu diễn cụ thể nào, và sau đó mới chỉ ra kiểu dữ liệu thích hợp làm tham số
- Qua các ví dụ, ta sẽ thấy đây là một phương pháp tự nhiên tuân theo khuôn mẫu hướng đối tượng theo nhiều kiểu

Lập trình tổng quát

- Ta đã quen với ý tưởng có một phương thức được định nghĩa sao cho khi sử dụng với các lớp khác nhau, nó sẽ đáp ứng một cách thích hợp
  - Khi nói về đa hình, nếu phương thức "draw" được gọi cho một đối tượng bất kỳ trong cây thừa kế Shape, định nghĩa tương ứng sẽ được gọi để đối tượng được vẽ đúng
  - Trong trường hợp này, mỗi hình đòi hỏi một định nghĩa phương thức hơi khác nhau để đảm bảo sẽ vẽ ra hình đúng
- Nhưng nếu định nghĩa hàm cho các kiểu dữ liệu khác nhau nhưng không cần phải khác nhau thì sao?

Lập trình tổng quát

- Ví dụ, xét hàm sau:

```
void swap(int& a, int& b) {
   int temp;
   temp = a; a = b; b = temp;
}
```

  - Hàm
  - Nếu ta muốn thực hiện việc tương tự cho một kiểu dữ liệu khác, chẳng hạn `float`?

```
void swap(float& a, float& b) {
   float temp;
   temp = a; a = b; b = temp;
}
```

  - Có thực sự cần đến cả hai phiên bản không?

Lập trình tổng quát

- Ví dụ khác: ta định nghĩa một lớp biểu diễn cấu trúc ngăn xếp cho kiểu `int`

```
class Stack {
   public:
      Stack();
      ~Stack();
      void push(const int& i);
      void pop(int& i);
      bool isEmpty() const;
   ...
};
```

- Ta thấy khai báo và định nghĩa của Stack phụ thuộc tại một mức độ nào đó vào kiểu dữ liệu int
  - Một số phương thức lấy tham số và trả về kiểu `int`
  - Nếu ta muốn tạo ngăn xếp cho một kiểu dữ liệu khác thì sao?
  - Ta có nên định nghĩa lại hoàn toàn lớp `Stack` (kết quả sẽ tạo ra nhiều lớp chẳng hạn `IntStack`, `FloatStack`, ...) hay không?

Lập trình tổng quát

- Ta thấy, trong một số trường hợp, đưa chi tiết về kiểu dữ liệu vào trong định nghĩa hàm hoặc lớp là điều không có lợi
  - Trong khi ta cần các định nghĩa khác nhau cho "draw" của Point hay Circle, vấn đề khác hẳn với trường hợp một hàm chỉ có nhiệm vụ hoán đổi hai giá trị
- Thực ra, khái niệm lập trình tổng quát học theo sự sử dụng một phương pháp của lớp cơ sở cho các thể hiện của các lớp dẫn xuất
  - Ví dụ, trong cây thừa kế khỉ, ta muốn cùng một phương thức eatBanana() được thực thi, bất kể con trỏ/tham chiếu đang chỉ tới một Monkey hay LazyMonkey
- Với lập trình tổng quát, ta tìm cách mở rộng sự trừu tượng hoá ra ngoài địa hạt của các cây thừa kế

Lập trình tổng quát trong C

Sử dụng trình tiền xử lý của C

- Trình tiền xử lý thực hiện thay thế text trước khi dịch
- Do đó, ta có thể dùng #define để chỉ ra kiểu dữ liệu và thay đổi tại chỗ khi cần

```
#define TYPE int

void swap(TYPE & a, TYPE & b) {
   TYPE temp;
   temp = a; a = b; b = temp;
}
```

Trình tiền xử lý sẽ thay mọi "**TYPE**" bằng "**int**" trước khi thực hiện biên dịch

Hai hạn chế:

- nhàm chán và dễ lỗi
- chỉ cho phép đúng một định nghĩa trong một chương trình

C++ template

- Template (khuôn mẫu) là một cơ chế thay thế mã cho phép tạo các cấu trúc mà không phải chỉ rõ kiểu dữ liệu
- Từ khoá template được dùng trong C++ để báo cho trình biên dịch rằng đoạn mã theo sau sẽ thao tác một hoặc nhiều kiểu dữ liệu chưa xác định
  - Từ khoá template được theo sau bởi một cặp ngoặc nhọn chứa tên của các kiểu dữ liệu tuỳ ý được cung cấp

```
template <typename T>
// Declaration that makes reference to a data type "T"
```

```
template <typename T, typename U>
// Declaration that makes reference to a data type "T"
// and a datatype "U"
```

Chú ý:
Một lệnh `template` chỉ có hiệu quả đối với khai báo **ngay sau** nó

C++ template

- Hai loại khuôn mẫu cơ bản:
  - Function template – khuôn mẫu hàm cho phép định nghĩa các hàm tổng quát dùng đến các kiểu dữ liệu tuỳ ý
  - Class template – khuôn mẫu lớp cho phép định nghĩa các lớp tổng quát dùng đến các kiểu dữ liệu tuỳ ý
- Ta sẽ mô tả từng loại trước khi đi bàn đến những phức tạp của lập trình khuôn mẫu

Khuôn mẫu hàm

- Khuôn mẫu hàm là dạng khuôn mẫu đơn giản nhất cho phép ta định nghĩa các hàm dùng đến các kiểu dữ liệu tuỳ ý
- Định nghĩa hàm `swap()` bằng khuôn mẫu:

```
template <typename T>
void swap(T & a, T & b) {
   T temp;
   temp = a; a = b; b = temp;
}
```

- Phiên bản trên trông khá giống với phiên bản `swap()` bằng C sử dụng `#define`, nhưng nó mạnh hơn nhiều

Khuôn mẫu hàm

- Thực chất, khi sử dụng template, ta đã định nghĩa một tập vô hạn các hàm chồng nhau với tên **`swap()`**
- Để gọi một trong các phiên bản này, ta chỉ cần gọi nó với kiểu dữ liệu tương ứng

```
int x = 1, y = 2;
float a = 1.1, b = 2.2;
...
swap(x, y); // Invokes int version of swap()
swap(a, b); // Invokes float version of swap()
```


Khuôn mẫu hàm

- Chuyện gì xảy ra khi ta biên dịch mã?
  - Trước hết, sự thay thế "**`T`**" trong khai báo/định nghĩa hàm **`swap()`** không phải thay thế text đơn giản và cũng không được thực hiện bởi trình tiền xử lý
  - Việc chuyển phiên bản mẫu của **`swap()`** thành các cài đặt cụ thể cho **`int`** và **`float`** được thực hiện bởi trình biên dịch

Khuôn mẫu hàm

- Hãy xem xét hoạt động của trình biên dịch khi gặp lời gọi **`swap()`** thứ nhất (với hai tham số **`int`**)
  - Trước hết, trình biên dịch tìm xem có một hàm **`swap()`** được khai báo với 2 tham số kiểu **`int`** hay không
    - Nó không tìm thấy một hàm thích hợp, nhưng tìm thấy một template có thể dùng được
  - Tiếp theo, nó xem xét khai báo của template **`swap()`** để xem có thể khớp được với lời gọi hàm hay không
    - Lời gọi hàm cung cấp hai tham số thuộc cùng một kiểu (int)
    - Trình biên dịch thấy template chỉ ra hai tham số thuộc cùng kiểu T, nên nó kết luận rằng T phải là kiểu int
    - Do đó, trình biên dịch kết luận rằng template khớp với lời gọi hàm

Khuôn mẫu hàm

- Khi đã xác định được template khớp với lời gọi hàm, trình biên dịch kiểm tra xem đã có một phiên bản của `swap()` với hai tham số kiểu `int` được sinh ra từ template hay chưa
  - Nếu đã có, lời gọi được liên kết (bind) với phiên bản đã được sinh (lưu ý: khái niệm liên kết này giống với khái niệm ta đã nói đến trong đa hình tĩnh)
  - Nếu không, trình biên dịch sẽ sinh một cài đặt của `swap()` lấy hai tham số kiểu `int` (thực ra là viết đoạn mã mà ta sẽ tạo nếu ta tự mình viết) – và liên kết lời gọi hàm với phiên bản vừa sinh.

Khuôn mẫu hàm

- Vậy, đến cuối quy trình biên dịch đoạn mã trong ví dụ, sẽ có hai phiên bản của `swap()` được tạo (một cho hai tham số kiểu `int`, một cho hai tham số kiểu `float`) với các lời gọi hàm của ta được liên kết với phiên bản thích hợp
  - Vậy, ta có thể đoán rằng có chi phí phụ về thời gian biên dịch đối với việc sử dụng template
  - Ngoài ra còn có chi phí phụ về không gian liên quan đến mỗi cài đặt của `swap()` được tạo trong khi biên dịch
  - Tuy nhiên, tính hiệu quả của các cài đặt đó cũng không khác với khi ta tự cài đặt chúng.

Khuôn mẫu hàm

- Cần ghi nhớ rằng tuy trình biên dịch đã tạo các phiên bản của swap() cho các tham số int và float, không tồn tại các hàm swap(int,int) hay swap(float, float)
  - Thay vào đó, có một hàm swap<>() được dùng để tạo hai hàm swap<int>() và swap<float>()
- Khi được dùng với một cấu trúc template, cặp ngoặc nhọn được dùng để chỉ rõ kiểu dữ liệu cần đến
- Thực tế, ta có thể sửa đoạn mã trước để gọi các hàm trên một cách tường minh:

```
int x = 1, y = 2;
float a = 1.1, b = 2.2;
...
swap<int>(x, y);   // Invokes int version of Swap()
swap<float>(a, b); // Invokes float version of Swap()
```

Khuôn mẫu lớp

- Tương tự với khuôn mẫu hàm với tham số thuộc các kiểu tuỳ ý, ta cũng có thể định nghĩa khuôn mẫu lớp (class template) sử dụng các thể hiện của một hoặc nhiều kiểu dữ liệu tuỳ ý
  - Ta cũng có thể định nghĩa template cho `struct` và `union`
- Khai báo một khuôn mẫu lớp cũng tương tự với khuôn mẫu hàm

Khuôn mẫu lớp

Ví dụ, ta sẽ tạo một cấu trúc cặp đôi giữ một cặp giá trị thuộc kiểu tuỳ ý

- Trước hết, xét khai báo Pair cho một cặp giá trị kiểu int:

```
struct Pair {
   int first;
   int second;
};
```

- Ta có thể sửa khai báo trên thành một khuôn mẫu lấy kiểu tuỳ ý:
  Tuy nhiên hai thành viên first và second phải thuộc cùng kiểu

```
template <typename T>
struct Pair {
   T first;
   T second;
};
```

- Hoặc ta có thể cho phép hai thành viên nhận các kiểu dữ liệu khác nhau:

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};
```

Khuôn mẫu lớp

- Để tạo các thể hiện của template Pair, ta phải dùng ký hiệu cặp ngoặc nhọn
  - Khác với khuôn mẫu hàm khi ta có thể bỏ qua kiểu dữ liệu cho các tham số, đối với khuôn mẫu class/struct/union, chúng phải được cung cấp tường minh

```
Pair p;                    // Not permitted
Pair<int, int> q;          // Creates a pair of ints
Pair<int, float> r; // Creates a pair with an int and a float
```

- Tại sao đòi hỏi kiểu tường minh?
  - Các lệnh trên làm gì? - cấp phát bộ nhớ cho đối tượng
  - Nếu không biết các kiểu dữ liệu được sử dụng, trình biên dịch làm thế nào để biết cần đến bao nhiêu bộ nhớ?

Khuôn mẫu lớp

- Cũng như khuôn mẫu hàm, không có `struct Pair` mà chỉ có các struct có tên `Pair<int, int>, Pair<int,float>, Pair<int,char>,…`
- Quy trình tạo các phiên bản `struct Pair` từ khuôn mẫu cũng giống như đối với khuôn mẫu hàm
- Khi trình biên dịch lần đầu gặp khai báo dùng `Pair<int, int>`, nó kiểm tra xem `struct` đó đã tồn tại chưa, nếu chưa, nó sinh một khai báo tương ứng.
  - Đối với các khuôn mẫu cho `class`, trình biên dịch sẽ sinh cả các định nghĩa phương thức cần thiết để khớp với khai báo `class`.

Khuôn mẫu lớp

- Một khi đã tạo được một thể hiện của một khuôn mẫu class/struct/union, ta có thể tương tác với nó như thể nó là thể hiện của một class/struct/union thông thường.

```
Pair<int, int> q;
Pair<int, float> r;
q.first = 5;
q.second = 10;
r.first = 15;
r.second = 2.5;
```

- Tiếp theo, ta sẽ tạo một template cho lớp Stack đã được mô tả trong các slice trước

Khuôn mẫu lớp

- Khi thiết kế khuôn mẫu (cho lớp hoặc hàm), thông thường, ta nên tạo một phiên bản cụ thể trước, sau đó mới chuyển nó thành một template
  - Ví dụ, ta sẽ bắt đầu bằng việc cài đặt hoàn chỉnh Stack cho số nguyên
- Điều đó cho phép phát hiện các vấn đề về khái niệm trước khi chuyển thành phiên bản cho sử dụng tổng quát
  - khi đó, ta có thể test tương đối đầy đủ lớp Stack cho số nguyên để tìm các lỗi tổng quát mà không phải quan tâm đến các vấn đề liên quan đến template

Stack cho số nguyên

- Khai báo và định nghĩa lớp **`Stack`** cho kiểu **`int`**
  - Bắt đầu bằng một ngăn xếp đơn giản

```
class Stack {
   public:
      Stack();
      ~Stack();
      void push(const int& i) throw (logic_error);
      void pop(int& i) throw (logic_error);
      bool isEmpty() const;
      bool isFull() const;
   private:
      static const int max = 10;
      int contents[max];
      int current;
};
```



```
Stack::Stack() { this->current = 0; }

Stack::~Stack() {}

void Stack::push(const int& i) throw(logic_error) {
   if (this->current < this->max) {
      this->contents[this->current++] = i;
   }
   else {
      throw logic_error("Stack is full.");
   }
}

void Stack::pop(int& i)  throw(logic_error) {
   if (this->current > 0) {
      i = this->contents[--this->current];
   } else {
      throw logic_error("Stack is empty.");
   }
}

bool Stack::isEmpty() const { return (this->current == 0;) }

bool Stack::isFull() const { return (this->current == this->max); }
```

Template Stack

- Chuyển khai báo và định nghĩa trước thành một phiên bản tổng quát:

```
template <typename T>
class Stack {
   public:
      Stack();
      ~Stack();
      void push(const T& i) throw (logic_error);
      void pop(T& i) throw (logic_error);
      bool isEmpty() const;
      bool isFull() const;
   private:
      static const int max = 10;
      T contents[max];
      int current;
};
```

Thêm lệnh **template** để nói rằng một phần của kiểu sẽ được chỉ rõ sau


```
template <typename T>
Stack<T>::Stack() { this->current = 0; }

template <typename T>
Stack<T>::~Stack() {}

template <typename T>
void Stack<T>::push(const T& i) {
   if (this->current < this->max) {
      this->contents[this->current++] = i;
   }
   else {
      throw logic_error("Stack is full.");
   }
}

template <typename T>
void Stack<T>::pop(T& i) {
   if (this->current > 0) {
      i = this->contents[--this->current];
   } else {
      throw logic_error("Stack is empty.");
   }
}

template <typename T>
bool Stack<T>::isEmpty() const { return (this->current == 0;) }

template <typename T>
bool Stack<T>::isFull() const { return (this->current == this->max); }
```

Mỗi phương thức cần một lệnh **template** đặt trước

Mỗi khi dùng toán tử phạm vi, cần một ký hiệu ngoặc nhọn kèm theo tên kiểu

Ta đang định nghĩa một lớp **Stack<type>**, chứ không định nghĩa lớp **Stack**

Thay thế kiểu của đối tượng được lưu trong ngăn xếp (trước là **int**) bằng kiểu tuỳ ý **T**

Template Stack

- Sau đó, ta có thể tạo và sử dụng các thể hiện của các lớp được định nghĩa bởi template của ta:

```
int x = 5, y;
char c = 'a', d;

Stack<int> s;
Stack<char> t;

s.push(x);
t.push(c);
s.pop(y);
t.pop(d);
```


Các tham số khuôn mẫu khác

- Ta mới nói đến các lệnh template với tham số thuộc "kiểu" typename
- Tuy nhiên, còn có hai "kiểu" tham số khác
  - Kiểu thực sự (ví dụ: **`int`**)
  - Các template

Các tham số khuôn mẫu khác

- Nhớ lại rằng trong cài đặt `Stack`, ta có một hằng `max` quy định số lượng tối đa các đối tượng mà ngăn xếp có thể chứa
  - Như vậy, mỗi thể hiện sẽ có cùng kích thước đối với mọi kiểu của đối tượng được chứa
- Nếu ta không muốn đòi hỏi mọi `Stack` đều có kích thước tối đa như nhau?
- Ta có thể thêm một tham số vào lệnh `template` chỉ ra một số `int` (giá trị này sẽ được dùng để xác định giá trị cho `max`)

```
template <typename T, int I>
// Specifies that one arbitrary type T and one int I
// will be parameters in the following statement
```

  - Lưu ý: ta khai báo tham số `int` giống như trong các khai báo khác

Các tham số khuôn mẫu khác

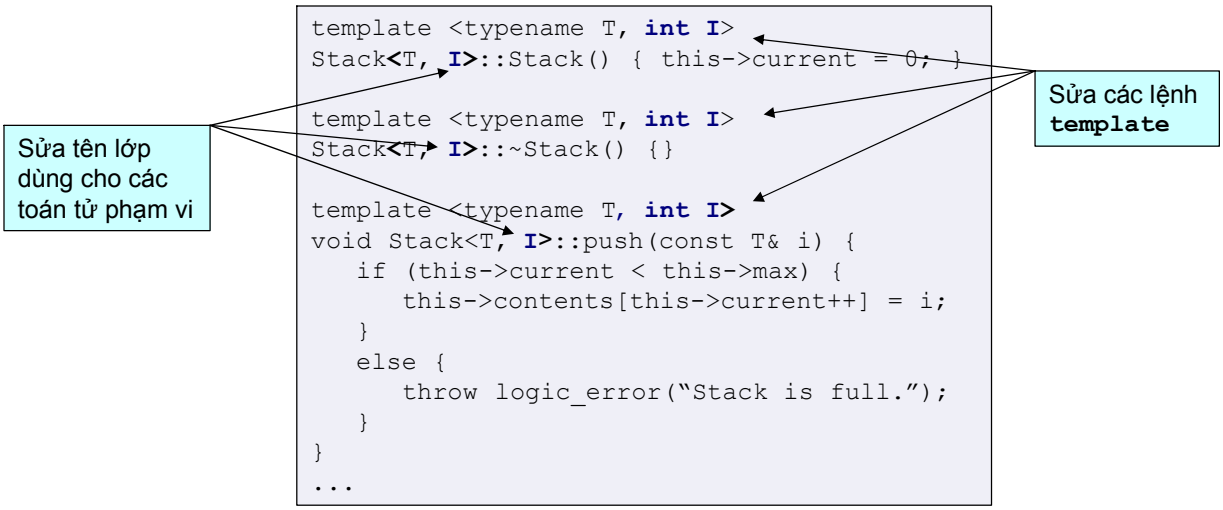
- Sửa khai báo và định nghĩa trước để sử dụng tham số mới:

```
template <typename T, int I>
class Stack {
   public:
      Stack();
      ~Stack();
      void push(const T& i) throw (logic_error);
      void pop(T& i) throw (logic_error);
      bool isEmpty() const;
      bool isFull() const;
   private:
      static const int max = I;
      T contents[max];
      int current;
};
```

Khai báo tham số mới

Sử dụng tham số mới để xác định giá trị **max** của một lớp thuộc một kiểu nào đó

Các tham số khuôn mẫu khác




```
template <typename T, int I>
Stack<T, I>::Stack() { this->current = 0; }

template <typename T, int I>
Stack<T, I>::~Stack() {}

template <typename T, int I>
void Stack<T, I>::push(const T& i) {
   if (this->current < this->max) {
      this->contents[this->current++] = i;
   }
   else {
      throw logic_error("Stack is full.");
   }
}
...
```


Các tham số khuôn mẫu khác

- Giờ ta có thể tạo các thể hiện của các lớp Stack với các kiểu dữ liệu và kích thước đa dạng

```
Stack<int, 5> s;  // Creates an instance of a Stack
                  // class of ints with max = 5
Stack<int, 10> t; // Creates an instance of a Stack
                  // class of ints with max = 10
Stack<char, 5> u; // Creates an instance of a Stack
                  // class of chars with max = 5
```

  - Lưu ý rằng các lệnh trên tạo thể hiện của 3 lớp khác nhau

Các tham số khuôn mẫu khác

- Các ràng buộc khi sử dụng các kiểu thực sự làm tham số cho lệnh template:
  - Chỉ có thể dùng các kiểu số nguyên, con trỏ, hoặc tham chiếu
  - Không được gán trị cho tham số hoặc lấy địa chỉ của tham số

Các tham số khuôn mẫu khác

- Loại tham số thứ ba cho lệnh template chính là một template
- Ví dụ, xét thiết kế khuôn mẫu cho một lớp Map (ánh xạ) ánh xạ các khoá tới các giá trị
  - Lớp này cần lưu các ánh xạ từ khoá tới giá trị, nhưng ta không muốn chỉ ra kiểu của các đối tượng được lưu trữ ngay từ đầu
  - Ta sẽ tạo Map là một khuôn mẫu sao cho có thể sử dụng các kiểu khác nhau cho khoá và giá trị
  - Tuy nhiên, ta cần chỉ ra lớp chứa (container) là một template, để nó có thể lưu trữ các khoá và giá trị là các kiểu tuỳ ý

Các tham số khuôn mẫu khác

- Ta có thể khai báo lớp `Map`:

```
template< typename Key, typename Value,
          template <typename T> Container>
class Map {
   ...
   private:
      Container<Key> keys;
      Container<Value> values;
      ...
};
```

- Sau đó có thể tạo các thể hiện của `Map` như sau:

```
Map< string, int, Stack> wordcount;
```

  - Lệnh trên tạo một thể hiện của lớp `Map<string, int, Stack>` chứa các thành viên là một tập các `string` và một tập các `int` (giả sử còn có các đoạn mã thực hiện ánh xạ mỗi từ tới một số `int` biểu diễn số lần xuất hiện của từ đó)
  - Ta đã dùng template `Stack` để làm container lưu trữ các thông tin trên

Các tham số khuôn mẫu khác

- Như vậy, khi trình biên dịch sinh các khai báo và định nghĩa thực sự cho các lớp **`Map`**, nó sẽ đọc các tham số mô tả các thành viên dữ liệu
- Khi đó, nó sẽ sử dụng khuôn mẫu Stack để sinh mã cho hai lớp **`Stack<string>`** và **`Stack<int>`**
- Đến đây, ta phải hiểu rõ tại sao container phải là một khuôn mẫu, nếu không, làm thế nào để có thể dùng nó để tạo các loại stack khác nhau?

Từ C đến C++

Lập trình hướng đối tượng

Tài liệu đọc

- Eckel, Bruce. *Thinking in C++, 2nd Ed. Volume 1*.
  - Chapter 3: The C in C++
  - Chapter 8: Constants
    - Up to p. 352: Classes
  - Chapter 10: Name Control
    - Up to p. 423: Static members in C++
  - Chapter 13: Dynamic Object Creation
    - Up to p. 566: Overloading new & delete
  - Chapter 11: References and the Copy-Constructor
    - Up to p. 452: References in Functions

Khác biệt đối với C

- Các khác biệt đối với C (ngoài các đặc điểm hướng đối tượng)
  - Chú thích
  - Các kiểu dữ liệu
  - Kiểm tra kiểu, đổi kiểu
  - Cảnh báo của trình biên dịch
  - Phạm vi và khai báo
  - Không gian tên
  - Hằng
  - Quản lý bộ nhớ
  - Tham chiếu

Chú thích

- Bên cạnh chú thích kiểu C (nhiều dòng), C++ cho phép kiểu chú thích dòng đơn

C

```
/* This is a variable */
int x;
/* This is the variable
 * being given a value */
x = 5;
```

C++

```
// This is a variable
int x;
// This is the variable
// being given a value
x = 5;
```

Chú thích

- C++ cho phép kiểu chú thích /* */ bao ngoài các chú thích dòng đơn.

C

```
/*
/* This is a variable */
int x;
/* This is the variable
 * being given a value */
x = 5;
*/
```

Chú thích có lỗi

C++

```
/*
// This is a variable
int x;
// This is the variable
// being given a value
x = 5;
*/
```


Kiểu dữ liệu

- Kiểu giá trị Boolean: **`bool`**
  - Hai giá trị: **`true`** hoặc **`false`**
  - Các toán tử logic (**!**, **&&**, ...) lấy/tạo một giá trị **`bool`**
  - Các phép toán quan hệ (**==**, **<**, ...) tạo một giá trị **`bool`**
  - Các lệnh điều kiệu (**`if`**, **`while`**, ...) đòi hỏi một giá trị **`bool`**
- Để tương thích ngược với C, C++ ngầm chuyển từ **`int`** sang **`bool`** khi cần
  - Giá trị `0` → **`false`**
  - Giá trị khác `0` → **`true`**

Kiểm tra kiểu dữ liệu

- C++ kiểm soát kiểu dữ liệu chặt chẽ hơn C
- C++ đòi hỏi hàm phải được khai báo trước khi sử dụng (mọi lời gọi hàm được kiểm tra khi biên dịch)
- C++ không cho phép gán giá trị nguyên cho các biến kiểu **enum**

```
enum Temperature {hot, cold};
enum Temperature t = 1; // Error in C++
```

- C++ không cho phép các con trỏ không kiểu (**void***) sử dụng trực tiếp tại bên phải lệnh gán hoặc một lệnh khởi tạo

```
void * vp;
int * ip = vp; // Error: Invalid conversion
```


Đổi và ép kiểu dữ liệu

- C++ cho phép người dùng đổi kiểu dữ liệu một cách khá rộng rãi
- Trình biên dịch tự động thực hiện nhiều chuyển đổi dễ thấy:
  - Gán một giá trị thuộc kiểu số học này cho một biến thuộc kiểu khác
  - Các kiểu số học khác nhau cùng có trong các biểu thức
  - Truyền đối số cho các hàm
- Nếu hiểu rõ khi nào các chuyển đổi này xảy ra và trình biên dịch đang làm gì,ta có thể giải thích được các kết quả không mong đợi

Đổi và ép kiểu dữ liệu

- Tự động chuyển đổi từ các đối tượng nhỏ thành các đối tượng lớn thì không có vấn đề gì, chiều ngược lại có thể có vấn đề
  - **short** → **long** (~16 bits → ~32 bits) không có vấn đề
  - **long** → **short** (~32 bits → ~16 bits) có thể mất dữ liệu
  - Khi chuyển từ các kiểu chấm động sang các kiểu nguyên có thể làm giảm độ chính xác của dữ liệu
- Trình biên dịch sẽ sinh cảnh báo (warning) đối với các chuyển đổi tự động có thể gây mất dữ liệu.

Đổi và ép kiểu dữ liệu

- C++ cho phép người dùng ép kiểu một cách tường minh bằng nhiều cách
  - Ép kiểu kiểu C: `myInt = (int) myFloat;`
  - Ép kiểu kiểu hàm C++: `myInt = int(myFloat);`
- Để hạn chế ép kiểu quá mức và loại trừ các lỗi do ép kiểu, C++ cung cấp một cách mới sử dụng 4 loại ép kiểu tường minh
  - `static_cast`
  - `const_cast`
  - `reinterpret_cast`
  - `dynamic_cast`
- Cú pháp `myInt = static_cast<int>(myFloat)`

Phạm vi và các Khai báo

- Trong C, các biến phải được định nghĩa tại đầu file hoặc tại bắt đầu của một khối **{…}**
- C++ cho phép khai báo sau và phạm vi của các biến được giới hạn chính xác hơn
  - Các khai báo có thể đặt tại các câu lệnh lặp for và các câu lệnh điều kiện
  - Phạm vi giới hạn bên trong vòng lặp hoặc khối điều kiện
- C++ còn bổ sung hai phạm vi mới:
  - Phạm vi không gian tên - *Namespace scope*
  - Phạm vi lớp - *Class scope*

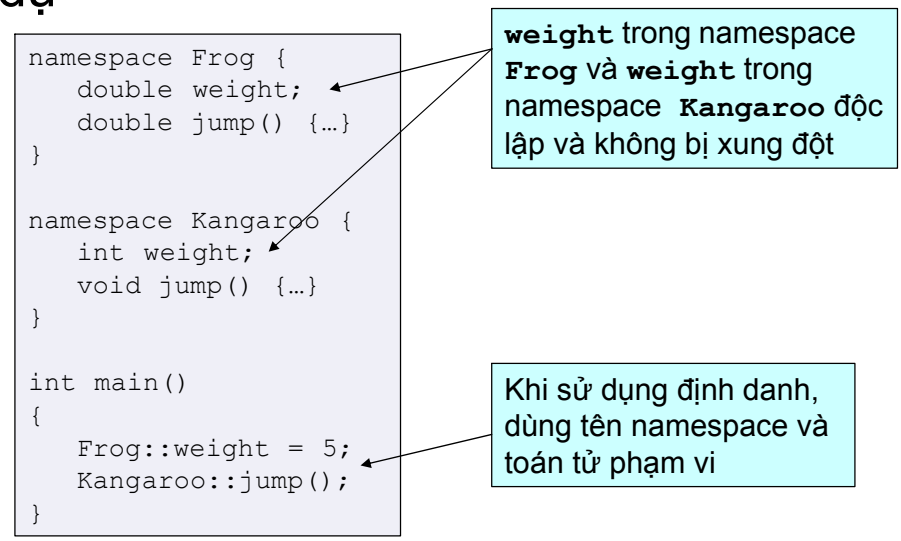
Namespace - Không gian tên

- Không gian tên được bổ sung vào C++ để biểu diễn cấu trúc logic và cung cấp khả năng quản lý phạm vi tốt hơn
- Không gian tên cung cấp một cơ chế tường minh để tạo các vùng khai báo
- Các tên khai báo trong một không gian tên
  - không xung đột với các tên được khai báo trong các không gian tên khác
  - Tránh xung đột tên biến, tên hàm
  - Nghiễm nhiên có thể được liên kết ra ngoài (external linkage)

```
namespace Frog {
   double weight;
   double jump() {…}
}

namespace Kangaroo {
   int weight;
   void jump() {…}
}
```

Namespace

- Ví dụ



Namespace

- C++ cung cấp hai cơ chế để đơn giản hóa việc sử dụng các namespaces: các khai báo using và các định hướng using.
- khai báo using (*using-declaration*) cho phép truy nhập một định danh cụ thể trong vùng khai báo tạm thời

```
using <namespace>::<name>;
```

  - Từ đây, ta có thể sử dụng tên mà không cần mỗi lần đều phải chỉ rõ namespace chứa nó.

Namespace

- Ví dụ sử dụng khai báo using

```
namespace Frog {
   double weight;
   double jump() {…}
}

int main()
{
   using Frog::weight;
   int weight; // Error (already declared locally)
   weight = 5; // Sets Frog::weight to 5
}
```

Khai báo using cho định danh **weight** trong namespace **Frog.**

Từ đây, **weight** được hiểu là **Frog::weight**

Namespace

- Khai báo using dành cho 1 tên, định hướng using cho phép truy nhập mọi định danh trong namespace

```
using namespace <namespace>;
```

- Định hướng using thường được đặt tại mức toàn cục.

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

Quản lý bộ nhớ

- Cấp phát bộ nhớ động trong C trông rối rắm và dễ lỗi
  - `myObj* obj = (myObj*)malloc(sizeof(myObj));`
- Các nhà thiết kế C++ thấy rằng:
  - Một ngôn ngữ sử dụng class sẽ hay phải sử dụng bộ nhớ động.
  - Không có lý do gì để tách cấp phát bộ nhớ động ra khỏi việc khởi tạo đối tượng (hay tách thu hồi bộ nhớ động ra khỏi việc hủy đối tượng)
- **`malloc`** và **`free`** đã được thay bằng **`new`** và **`delete`**

Quản lý bộ nhớ

- Lợi thế của **new** so với **malloc**:
  - Không cần chỉ ra lượng bộ nhớ cần cấp phát
  - Không cần đổi kiểu
  - Không cần dùng lệnh if để kiểm tra xem bộ nhớ đã hết chưa
  - Nếu bộ nhớ đang được cấp cho một đối tượng, hàm khởi tạo (constructor) của đối tượng sẽ được gọi tự động (tương tự, **delete** sẽ tự động gọi hàm hủy (destructor) của đối tượng)
- Ví dụ:

```
myObj* obj = new myObj;        //một đối tượng
delete obj;

myObj* obj = new myObj[10];       // mảng đối tượng
delete[] obj;
```

Tham chiếu – Reference

- Tham chiếu tới một đối tượng là một biệt danh tới đối tượng đó
- Có thể coi mọi thao tác trên tham chiếu đều được thực hiện trên chính đối tượng nguồn.

```
int x = 5;
int& y = x;
cout << "x = " << x << " y = " << y << ".\n"; // x = 5 y = 5
x = x + 1;
cout << "x = " << x << " y = " << y << ".\n"; // x = 6 y = 6
y = y + 1;
cout << "x = " << x << " y = " << y << ".\n"; // x = 7 y = 7

int *p = &y;
*p = 9;
cout << "x = " << x << " y = " << y << ".\n"; // x = 9 y = 9
```


Tham chiếu – Reference

- Tham chiếu có thể được dùng độc lập nhưng thường hay được dùng làm tham số cho hàm
- C truyền mọi đối số cho hàm bằng giá trị (truyền trị - call by value)
  - Khi cần, ta có thể truyền một con trỏ tới đối tượng (chính nó cũng được truyền bằng giá trị)
  - `void myFunction(myObj* obj) {…}`
- C++ cho phép các đối số hàm được truyền bằng tham chiếu (call by reference)
  - `void myFunction(myObj& obj) {…}`
  - **`myObj&`** có nghĩa "tham chiếu tới **`myObj`**"
  - đối với các đối số là các đối tượng lớn, truyền bằng tham chiếu đỡ tốn kém hơn truyền bằng giá trị (do chỉ truyền địa chỉ bộ nhớ)

Tham chiếu – Reference

- Ví dụ: tham chiếu làm đối số cho hàm

```
int f(int &i) { ++i; return i; }
int main() {
   int j = 7;   cout << f(j) << endl; cout << j <<
  endl;
}
```

- Biến i là một biến địa phương của hàm **f. i** thuộc kiểu tham chiếu **int** và được tạo khi **f** được gọi.
- Trong lời gọi **f(j), i** được tạo tương tự như trong lệnh **int &i = j;**
- Do đó trong hàm **f, i** sẽ là một tên khác của biến **j** và sẽ luôn như vậy trong suốt thời gian tồn tại của **i**

Tham chiếu – Reference

- Tham chiếu *phải* được khởi tạo
  - `int& x; // Error`
- Giá trị của tham chiếu không được thay đổi sau khi đã khởi tạo
  - không thể "chiếu" lại một tham chiếu tới đối tượng khác
  - chú ý phân biệt giữa khởi tạo tham chiếu và gán trị cho tham chiếu
- Truy nhập tới tham chiếu chính là truy nhập tới đối tượng nguồn
  - áp dụng cho cả toán tử & và phép gán
    - **cout << "The address of x is: " << &x << ".\n";**
    - **cout << "The address of y is: " << &y << ".\n";**
    - Output:
    - The address of x is 0x0056dc13.
    - The address of y is 0x0056dc13.

Tham chiếu – Reference

- Vậy tại sao dùng tham chiếu thay cho con trỏ?
- Tham chiếu sạch hơn, không dễ gây lỗi như con trỏ, đặc biệt khi dùng trong hàm
  - tham chiếu đảm bảo không chiếu tới null
- Sử dụng tham chiếu trong nguyên mẫu hàm giúp cho việc gọi hàm dễ hiểu hơn
  - không cần dùng toán tử địa chỉ

```
void func1(int *pi) { (*pi)++; }
void func2(int &ri) { ri++; }

int main() {
   int i = 1;

   func1(&i); // call using address of i
   func2(i);  // call using i
}
```


Const

- Trong C, hằng được định nghĩa bằng định hướng tiền xử lý **`#define`**

```
#define PI 3.14
```

  - Biên dịch chậm hơn (trình tiền xử lý tìm và thay thế)
  - Trình debug không biết đến các tên hằng
  - Sử dụng **#define** không gắn được kiểu dữ liệu với giá trị hằng (v.d. '**15**' là **int** hay **float?**)
- Const của ANSI-C ít dùng hơn và có nghĩa hơi khác:
  - ANSI-C **const** không được trình biên dịch chấp nhận là hằng
    - Không dùng để khai báo mảng được
    - In C++, **const** values can be used when declaring arrays:

Const

Hằng của ANSI-C và C++ có các quy tắc phạm vi khác nhau

- Các giá trị **`#define`** có phạm vi file (do trình tiền xử lý chỉ thực hiện tìm và thay thế tại file đó)
- **`const`** của ANSI-C được coi là các biến có giá trị không đổi và có phạm vi chương trình
  - Do đó ta không thể có các biến trùng tên tại hai file khác nhau
- Trong C++, các định danh **`const`** tuân theo các quy tắc phạm vi như các biến khác
  - Do đó, chúng có thể có phạm vi toàn cục, phạm vi không gian tên, hoặc phạm vi khối

const và con trỏ

- Ba loại:

```
1.  const int * pi;     // con trỏ tới hằng
2.  int * const ri = &i; // hằng con trỏ
3.  const int * const ri = &i; //hằng con trỏ tới hằng
```

- Các khái niệm cần ghi nhớ:
- Nếu một đối tượng là hằng
  - Không thể sửa đối tượng đó
  - Chỉ có con trỏ tới hằng mới được dùng để trỏ tới hằng, con trỏ thường không dùng được
- Nếu **`PI`** được khai báo là con trỏ tới hằng:
  - Có thể thay đổi **`PI`**, nhưng ***`PI`** không thể bị thay đổi.
  - **`PI`** có thể trỏ đến hằng hoặc biến thường

const và con trỏ

- Khi sử dụng một con trỏ, có hai đối tượng có liên quan: chính con trỏ đó và đối tượng nó trỏ tới
- Cú pháp cho con trỏ tới hằng và hằng con trỏ rất dễ nhầm lẫn
- Quy tắc: trong lệnh khai báo, từ khoá const *bên trái* dấu * có nghĩa đối tượng được trỏ tới là hằng, từ khoá const *bên phải* dấu * có nghĩa con trỏ là hằng
- Cách dễ hơn: đọc các khai báo từ phải sang trái
  - `char c = 'Y';` // c là char
  - `char *const cpc = &c;` //cpc là hằng con trỏ tới char
  - `const char *pcc;` // pcc là con trỏ tới hằng char
  - `const char *const cpcc = &c;` // cpcc là hằng con trỏ tới hằng char

hằng tham chiếu làm tham số hàm

Có hai lý do để truyền tham biến, nhưng nếu hàm được truyền không cần sửa đối tượng được truyền thì ta nên khai báo tham biến là **`const`**. Có hai ích lợi:

1. Nếu trong hàm, ta lỡ sửa đổi tham số thì trình biên dịch sẽ bắt lỗi.
   - ngăn chặn được một số lỗi lập trình viên có thể phạm
2. Ta có thể truyền các đối số là hằng hoặc không phải hằng cho hàm có hằng tham biến
   - Ngược lại, đối với các hàm có tham biến không phải là hằng, ta không thể truyền hằng làm đối số.

hằng tham chiếu làm tham số hàm

- Ví dụ

Lỗi: Lo là hằng nên không được sửa đổi

```
void non_constRef(LargeObj &Lo) { Lo.height +=10; }          // Fine

    void constRef(const LargeObj &Lo) { Lo.height +=10; }  // Error
    !!!
```

```
void non_constRef(LargeObj &Lo) { cout Lo.height; }

void constRef(const LargeObj &Lo) { cout Lo.height; }

int main {
   LargeObj dinosaur;   const LargeObj rocket;

   non_constRef(dinosaur);
   constRef(dinosaur);
   non_constRef(rocket);          // Error
   constRef(rocket);
}
```

Lỗi: rocket là hằng, nên không thể làm đối số không phải hằng

const: Hàm thành viên

Đối với hàm thành viên không sửa dữ liệu của đối tượng chủ, ta nên khai báo hàm đó là hằng hàm

- Đối với các đối tượng được khai báo là hằng, C++ chỉ cho phép gọi các hàm thành viên là hằng mà không cho phép gọi các hàm thành viên không phải là hằng của đối tượng đó.

```
class Date
{
   int year, month, day;
public:
   int getDay() const { return day; }
   int getMonth() const { return month; }
   void addYear(int y)            // Non-const function
};
```

Const - Tóm tắt

Nên khai báo hằng đối với:

- Các đối tượng mà ta không định sửa đổi

```
const double PI = 3.14;
const Date openDate(18,8,2003);
```

- Các tham số của hàm mà ta không định cho hàm đó sửa đổi

```
void printHeight(const LargeObj &LO)
   { cout << LO.height; }
```

- Các hàm thành viên không thay đổi đối tượng chủ

```
int Date::getDay() const { return day; }
```

Lưu ý: các yêu cầu trên áp dụng cho tất cả các bài tập, bài thi của môn học.

Ôn tập về Con trỏ

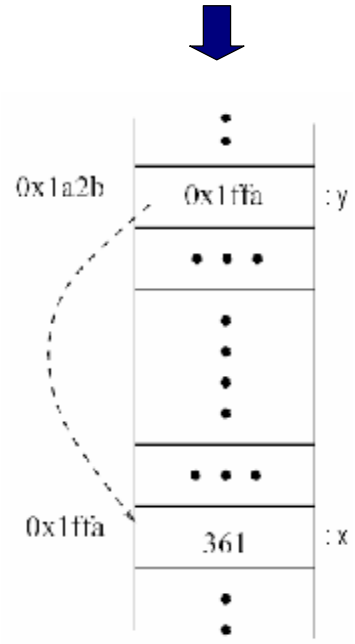
Lập trình hướng đối tượng

Ôn tập con trỏ

- Cấp phát động (Dynamic Allocation)
  `new, delete`
- Con trỏ lạc (Dangling pointers)
- Rò rỉ bộ nhớ (Memory leakage)
- Con trỏ mảng (Array Pointer)
- Các phép tính trên con trỏ (Pointer Arithmetic)
- Con trỏ tới bản ghi
- Cấp phát động mảng

Con trỏ

```
int x = 361;
int *y = &x;
```



Một con trỏ hay một biến con trỏ là:

- một biến chiếu đến một ô nhớ.
- nó lưu vị trí/địa chỉ của ô nhớ đó.

- Hai ứng dụng chính:
  - Truy nhập gián tiếp
  - Bộ nhớ động

- Vấn đề kỹ thuật:
  Nếu P là một biến con trỏ
  - Làm thế nào để trỏ P đến một ô nhớ nào đó?
  - Làm thế nào để truy nhập đến ô nhớ P trỏ đến?

Thao tác con trỏ

- Các ký hiệu, từ khóa: **`&, *, new, delete`**

```
int X, Y;
int* P;     //  P is an integer pointer variable
```

- Lệnh thứ hai khai báo một biến con trỏ P có giá trị chưa xác định nhưng khác Null. Biến con trỏ này có thể chỉ trỏ tới một ô nhớ chứa một số nguyên

```
 P = &Y;     // trỏ P tới Y (P lưu địa chỉ của Y)
*P = X;      // ghi giá trị của biến X vào vùng bộ nhớ trỏ bởi P
```

- Ví dụ

```
Y = 5;              //  variable Y stores value
P = &X;             // P points to memory location of X
*P = Y;             //  same as writing X = Y
```

Sau ví dụ trên, X = 5, Y = 5, và P trỏ tới X

Ví dụ

```
#include <iostream>
int main()
{  int x = 10; int y = 20;
   int *p1, *p2;

   p1 = &x;
   p2 = &y;
   cout << "x = " << x << endl;
   cout << "y = " << y << endl;
   cout << "*p1 = " << *p1 << endl;
   cout << "*p2 = " << *p2 << endl << endl;

   *p1 = 50;
   *p2 = 90;
   cout << "x = " << x << endl;
   cout << "y = " << y << endl;
   cout << "*p1 = " << *p1 << endl;
   cout << "*p2 = " << *p2 << endl << endl;

   p1 = p2;
   cout << "x = " << x << endl;
   cout << "y = " << y << endl;
   cout << "*p1 = " << *p1 << endl;
   cout << "*p2 = " << *p2 << endl << endl;
}
```

```
x = 10
y = 20
*p1 = 10
*p2 = 20

x = 50
y = 90
*p1 = 50
*p2 = 90

x = 50
y = 90
*p1 = 90
*p2 = 90
```


Ký hiệu

- Đọc **`*P`** là **biến mà P trỏ tới**
- Đọc **`&X`** là **địa chỉ của X**
- **`&`** là **toán tử địa chỉ** (**address of** operator)
- **`*`** là **toán tử thâm nhập** (**dereferencing** operator)
- Giả sử **`P1 = &X`** và **`P2 = &Y`**, thì **`P1`** trỏ tới **`X`** và **`P2`** trỏ tới **`Y`**

  `P1 = P2`

  Không tương đương với

  `*P1 = *P2`
- **`P1 = P2`** có hiệu quả trỏ **`P1`** tới **`Y`**, lệnh đó không thay đổi **`X`**
- Lệnh **`*P1 = *P2;`** tương đương với **`X = Y;`**

Sử dụng typedef

- Lỗi hay gặp khi sử dụng con trỏ. Phân biệt hai dòng sau:

```
int* P, Q; //  P is a pointer and Q an int
int *P, *Q;//  P and Q are both pointers
```

- Một cách tránh lỗi là sử dụng lệnh typedef để đặt tên kiểu mới. Ví dụ:

```
typedef double distance;  //distance is a new name for double

distance miles;
```

Giống như

```
double miles;
```

Có nghĩa rằng, thay vì viết

```
int *P, *Q;
```

Ta có thể viết

```
typedef int* IntPtr;       //  new name for pointers to ints
IntPtr P, Q;               //P and Q are both pointers
```


Cấp phát bộ nhớ tĩnh và động
(Static and Dynamic Allocation Of Memory)

Đoạn trình

```
int X,Y;  // X and Y are integers
int *P;   //  P is an integer pointer variable
```

Cấp phát bộ nhớ cho X, Y và P tại thời điểm biên dịch

Đó là **cấp phát tĩnh (static allocation)**

- Bộ nhớ cũng có thể được cấp phát tại thời gian chạy. Đó gọi là **Cấp phát động (dynamic allocation)**. Ví dụ:

```
P = new int;
```

  - Cấp phát một ô nhớ mới có thể chứa một số nguyên, và trỏ P tới ô nhớ đó

Ví dụ

```
//Program to demonstrate pointers
//and dynamic variables
#include <iostream>
int main()
{
   int *p1, *p2;
   p1 = new int;
   *p1 = 10;
   p2 = p1;
   cout << "*p1 = " << *p1 << endl;
   cout << "*p2 = " << *p2 << endl << endl;

   *p2 = 30;
   cout << "*p1 = " << *p1 << endl;
   cout << "*p2 = " << *p2 << endl << endl;

   p1 = new int;
   *p1 = 40;
   cout << "*p1 = " << *p1 << endl;
   cout << "*p2 = " << *p2 << endl << endl;
}
```

```
*p1 = 10
*p2 = 10

*p1 = 30
*p2 = 30

*p1 = 40
*p2 = 30
```


Cấp phát - thu hồi bộ nhớ động

- **heap:** vùng bộ nhớ đặc biệt dành riêng cho các biến động. Để tạo một biến động mới, hệ thống cấp phát không gian từ heap. Nếu không còn bộ nhớ, **`new`** không thể cấp phát bộ nhớ thì nó trả về gia trị **`Null`**
- Trong lập trình thực thụ, ta nên luôn luôn kiểm tra lỗi này

```
int *p;
p = new int;
if (p == NULL) {
    cout << "Memory Allocation Error\n";
    exit;
}
```

- Thực ra, NULL là giá trị 0, nhưng ta coi nó là một giá trị đặc biệt vì còn sử dụng cho trường hợp đặc biệt: con trỏ "rỗng".

Cấp phát - thu hồi bộ nhớ động

- Hệ thống chỉ có một lượng bộ nhớ giới hạn,
  - cần trả lại cho heap phần bộ nhớ động không còn sử dụng.
- Lệnh

```
delete P;
```

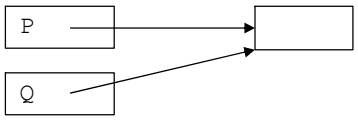
  - trả lại vùng bộ nhớ trỏ bởi **P**, nhưng không sửa giá trị của **P**.
  - Sau khi thực thi **delete P**, giá trị của **P** không xác định.

Con trỏ lạc – Dangling Pointer

- khi **delete P**, ta cần chú ý không xoá vùng bộ nhớ mà một con trỏ Q khác đang trỏ tới.

```
int *P;
int *Q;
P = new int;
Q = P;
```

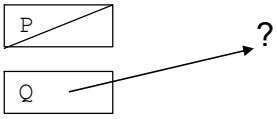
tạo



sau đó

```
delete P;
P = Null;
```

làm Q bị lạc

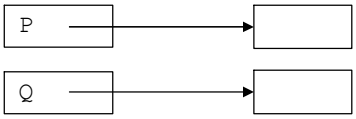


Rò rỉ bộ nhớ

- Một vấn đề liên quan: *mất* mọi con trỏ đến một vùng bộ nhớ được cấp phát. Khi đó, vùng bộ nhớ đó bị mất dấu, không thể trả lại cho heap được.

```
int *P;
int *Q;
P = new int;
Q = new int;
```

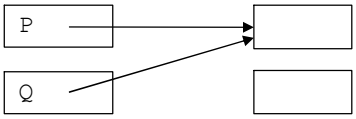
tạo



sau đó

```
Q = P;
```

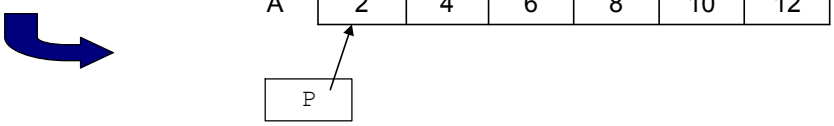
làm mất vùng nhớ đã từng được Q trỏ tới



Mảng và con trỏ

Tên mảng được coi như một *con trỏ* tới phần tử đầu tiên của mảng.

```
int A[6] = {2,4,6,8,10,12};        //  defines an array of inegers
int *P;

P = A;                             // P points to A
```

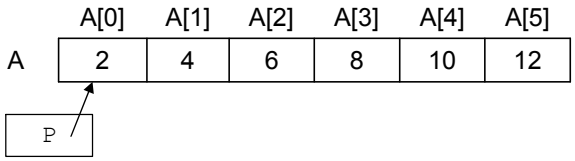


Do tên mảng và con trỏ là tương đương, ta có thể dùng P như tên mảng. Ví dụ:

`P[3] = 7;` tương đương với **`A[3] = 7;`**

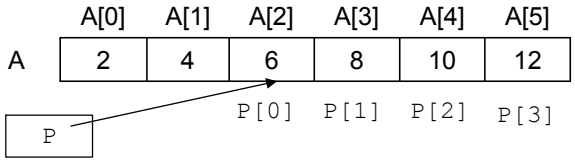
Ví dụ

Bắt đầu



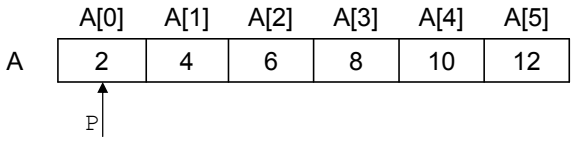
Thực hiện P = &A[2]

Bây giờ, P[0] là A[2],
p[1] là A[3],...

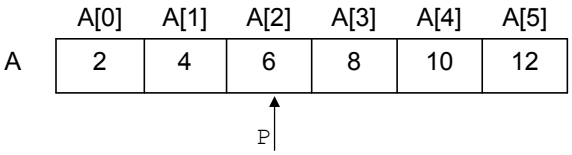


Các phép tính trên con trỏ

P = A;

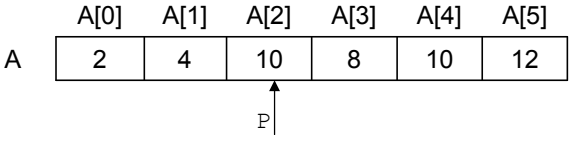


P = A + 2;

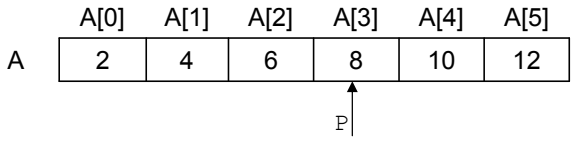


P++;

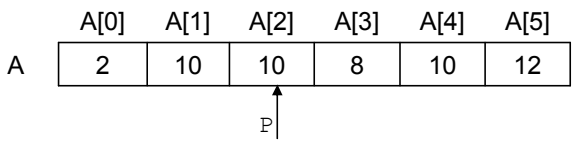
*P = *(P+1) + 2;



P = P + 2;



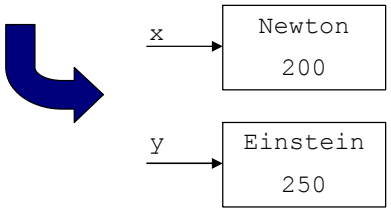
*(P-1) = *(P+2)



Con trỏ tới bản ghi: bộ nhớ động

```
#ifndef IQ1_H
#define IQ1_H

#include <iostream>

class IQ
{
private:
   char name[20];
   int score;
public:
   IQ (const char s, int k) {
      strcpy(name, s);
      score = k;
   }
   void smarter(int k) { score += k;}
   void print() const {
      cout << "(" << name << ", "
           << score << ")" << endl;
   }
}
#endif
```

```
#include <iostream>
#include "iq1.h"

int main()
{
   IQ *x = new IQ("Newton",200);
   IQ *y = new IQ("Einstein",250);

   x->print();
   y->print();

   return;
}
```



Mảng cấp phát động

- **`new T[n]`** cấp phát một mảng gồm **`n`** đối tượng kiểu **`T`** và trả về một con trỏ tới đầu mảng
- **`delete [] p`** huỷ mảng mà **`p`** trỏ tới và trả vùng bộ nhớ đó cho heap. **`P`** *phải* trỏ tới đầu mảng động, Nếu không, kết quả của **`delete`** sẽ phụ thuộc vào trình biên dịch và loại dữ liệu đang sử dụng. Ta có thể nhận được lỗi runtime error hoặc kết quả sai.
- Kích thước của mảng động không cần là hằng số mà có thể có giá trị được quyết định tại thời gian chạy

```
#include <iostream>
int main ()
{
   int size;
   cin << size;
   int* A = new int[size];     // dynamically allocate array

   A[0] = 0; A[1] = 1; A[2] = 2;
   cout << "A[1] = " << A[1] << endl;

   delete [] A;           // delete the array
}
```

Huỷ mảng động bất hợp lệ

```
#include <iostream>

int main ()
{
   int* A = new int[6];     // dynamically allocate array

   A[0] = 0; A[1] = 1; A[2] = 2;
   A[3] = 3; A[4] = 4; A[5] = 5;

   int *p = A + 2;
   cout << "A[1] = " << A[1] << endl;

   delete [] p;           // illegal!!!

   // results depend on particular compiler
   cout << "A[1] = " << A[1] << endl;
}
```

P không trỏ tới đầu mảng A

Huỷ không hợp lệ

Kết quả phụ thuộc trình biên dịch

Cấp phát động mảng đa chiều

- Cấp phát động mảng hai chiều (N+1)(M+1) gồm các đối tượng IQ:

```
IQ **a = new (IQ*) [N+1];

for (int i=0; i<N+1; i++)
   a[i] = new IQ[M+1];
```