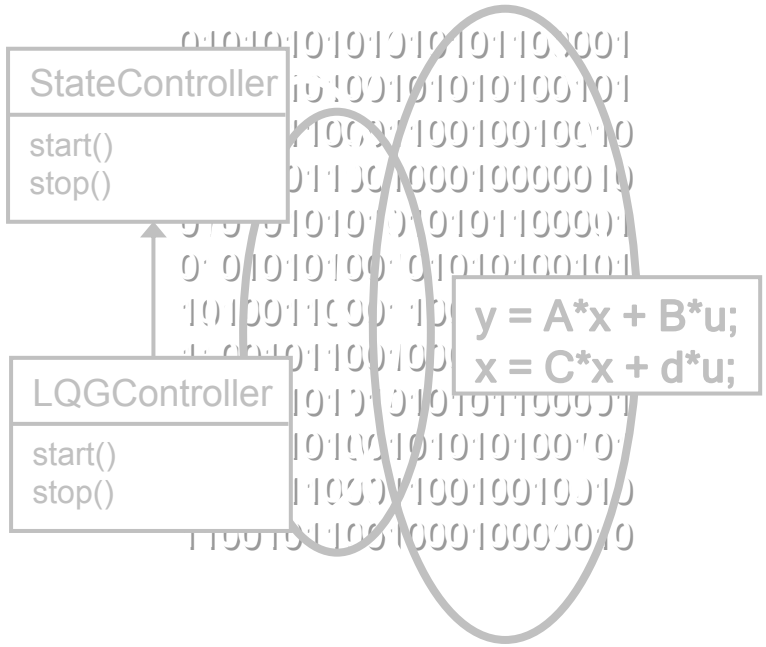
Kỹ thuật lập trình

Chương 1: Mở đầu



8/18/2005

Nội dung bài giảng

1.1 Giới thiệu nội dung môn học

1.2 Giới thiệu chung về kỹ thuật lập trình

1.3 Phương pháp luận

1.4 Qui trình phát triển phần mềm

1.5 Sơ lược về ngôn ngữ C/C++

1.1 Nội dung môn học

- Các **kỹ thuật lập trình cơ bản**, thực hiện trên các **ngôn ngữ lập trình C và C++:**
  - Lập trình có cấu trúc
  - Lập trình hướng đối tượng
  - Lập trình thời gian thực
  - Lập trình tổng quát
- Tại sao chọn C/C++:
  - Hai ngôn ngữ lập trình tiêu biểu nhất, đủ cho thực hiện các kỹ thuật lập trình quan trọng
  - Hai ngôn ngữ lập trình quan trọng nhất đối với kỹ sư điện/kỹ sư điều khiển

Quan điểm về môn học

- Đề cao kiến thức cơ bản, nền tảng:
  - Thiên về **tư duy** và **phương pháp** lập trình
  - Tạo khả năng dễ **thích ứng với các ứng dụng** khác nhau
  - Tạo khả năng dễ **thích ứng với các ngôn ngữ lập trình** khác (Java, Visual Basic, C#, MATLAB…)
  - Nhấn mạnh **tính chuyên nghiệp** trong lập trình: **hiệu quả + chất lượng**
- Những nội dung **không** có trong chương trình:
  - Lập trình hệ thống (low-level system programming)
  - Lập trình đồ họa
  - Lập trình giao tiếp với các cổng vào/ra (nối tiếp, song song)
  - Lập trình cơ sở dữ liệu
  - Lập trình thành phần, lập trình phân tán (mạng, Internet)

Phương pháp học tập

- Cách thứ nhất: Nghe giảng ➔ làm thử ➔ đọc tài liệu ➔ thảo luận ➔ luyện tập
- Cách thứ hai: Đọc tài liệu ➔ làm thử ➔ nghe giảng ➔ thảo luận ➔ luyện tập
- Nguyên tắc cơ bản: **Chủ động học thường xuyên!**
- Những điều không nên làm:
  - Chép nhiều trên lớp
  - Học thuộc lòng, học chay
  - Mong đợi nhiều vào ôn tập
  - Dựa dẫm vào các bài tập mẫu trong sách

Công cụ học tập

- Máy tính PC
- Công cụ lập trình: **Visual C++ 6.0 (Visual Studio 6.0),** Visual C++ .NET, Borland C++ Builder
- Nền ứng dụng: **Win32 Console Application**
- Tài liệu tham khảo:

1. Stanley B. Lippman, Josée Lajoie: **C++ Primer**. 3rd Edition. Addison-Wesley 1998.
2. Bjarne Stroustrup: **The C++ Programming Language**. 3rd Edition. Addison-Wesley 1997.
3. David Musser,…:**C++ Programming with Standard Template Library**. 2nd Edition, Addison-Wesley 1998.
4. Bruce Eckel: **Thinking in C++.** www.bruceeckel.com, 2003.

1.2 Tổng quan về kỹ thuật lập trình

- Kỹ thuật lập trình là gì: *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.*
- Kỹ thuật lập trình

  = Tư tưởng thiết kế + Kỹ thuật mã hóa

  = Cấu trúc dữ liệu + Giải thuật + Ngôn ngữ lập trình
- Kỹ thuật lập trình

  ≠ Phương pháp phân tích & thiết kế (A&D)

Thế nào là lập trình?

Viết chương trình tính giai thừa của 100!

Viết chương trình in ra 100 số nguyên tố đầu tiên!

Lập trình giải bài toán:
"Vừa gà vừa chó,
ba mươi sáu con,
bó lại cho tròn,
một trăm chân chẵn"

KHÔNG PHẢI LÀ LẬP TRÌNH!

Viết một hàm tính giai thừa!

Viết chương trình in ra N số nguyên tố đầu tiên!

Lập trình giải bài toán:
"Vừa gà vừa chó,
vừa vặn X con,
bó lại cho tròn,
đủ Y chân chẵn"

ĐÂY LÀ LẬP TRÌNH!

Thế nào là lập trình tốt?

- Giải đúng đề bài, được khách hàng chấp nhận
- Tin cậy
  - Chương trình chạy đúng
  - Chạy ít lỗi (số lượng lỗi ít, cường độ lỗi thấp)
  - Mức độ lỗi nhẹ
- Hiệu suất
  - Chương trình nhỏ gọn, sử dụng ít bộ nhớ
  - Tốc độ nhanh, sử dụng ít thời gian CPU
- Hiệu quả:
  - Thời gian lập trình ngắn,
  - Khả năng bảo trì dễ dàng
  - Giá trị sử dụng lại lớn
  - Sử dụng đơn giản, thân thiện
  - Nhiều chức năng tiện ích

Ví dụ minh họa: Tính giai thừa

- Viết chương trình hay **xây dựng hàm**?
  - **Hàm tính giai thừa của một số nguyên**

    `int factorial(int N);`
- Giải thuật:
  - Phương pháp đệ quy (*recursive*)

```
if (N > 1)
  return N*factorial(N-1);
return 1;
```

  - **Phương pháp lặp (*iterative*)**

```
int kq = 1;
while (N > 1)
  kq *= N--;
return kq;
```

☺„*to iterate is human, to recurse is devine!*“

Làm thế nào để lập trình tốt?

- Học cách tư duy và phương pháp lập trình
  - Tư duy toán học, tư duy logic, tư duy có cấu trúc, tư duy hướng đối tượng, tư duy tổng quát
  - Tìm hiểu về cấu trúc dữ liệu và giải thuật
- Hiểu sâu về máy tính
  - Tương tác giữa CPU, chương trình và bộ nhớ
  - Cơ chế quản lý bộ nhớ
- Nắm vững ngôn ngữ lập trình
  - Biết rõ các khả năng và hạn chế của ngôn ngữ
  - Kỹ năng lập trình (đọc thông, viết thạo)
- Tự rèn luyện trên máy tính
  - Hiểu sâu được các điểm nêu trên
  - Rèn luyện kỹ năng lập trình
  - Thúc đẩy sáng tạo

Các nguyên tắc cơ bản

- Trừu tượng hóa
  - Chắt lọc ra những yếu tố quan trọng, bỏ qua những chi tiết kém quan trọng
- Đóng gói
  - Che giấu và bảo vệ các dữ liệu quan trọng qua một giao diện có kiểm soát
- Module hóa
  - Chia nhỏ đối tượng/vấn đề thành nhiều module nhỏ để dễ can thiệp và giải quyết
- Phân cấp
  - Phân hạng hoặc sắp xếp trật tự đối tượng theo các quan hệ trên dưới

Nguyên tắc tối cao

„Keep it simple:

as simple as possible,

but no simpler!“

(Albert Einstein)

Các bài toán lập trình cho kỹ sư điện

- Lập trình phần mềm điều khiển (µC, PC, PLC, DCS)
- Lập trình phần mềm thu thập/quản lý dữ liệu quá trình
- Lập trình phần mềm giao diện người-máy (đồ họa)
- Lập trình phần mềm tích hợp hệ thống (COM, OPC,...)
- Lập trình phần mềm tính toán, thiết kế
- Lập trình phần mềm mô phỏng
- Lập trình phần mềm tối ưu hóa
- ...

1.3 Phương pháp luận

- Phương pháp: *Cách thức tiến hành một công việc để có hiệu quả cao*
- Phương pháp luận: *Một tập hợp các phương pháp được sử dụng hoặc bộ môn khoa học nghiên cứu các phương pháp đó*
- Phương pháp luận phục vụ:
  - Phân tích hệ thống
  - Thiết kế hệ thống
  - Thực hiện
  - Thử nghiệm
  - ...

Lập trình tuần tự (Sequential Programming)

- Phương pháp cổ điển nhất, bằng cách liệt kê các lệnh kế tiếp, mức trừu tượng thấp
- Kiểm soát dòng mạch thực hiện chương trình bằng các lệnh rẽ nhánh, lệnh nhảy, lệnh gọi chương trình con (subroutines)
- Ví dụ ngôn ngữ đặc thù:
  - Ngôn ngữ máy,
  - ASSEMBLY
  - BASIC
  - IL (Instruction List), STL (Statement List)
  - LD, LAD (Ladder Diagram)

Lập trình tuần tự: Ví dụ tính giai thừa

```
1:    MOV   AX, n
2:    DEC   n
3:    CMP   n, 1
4:    JMPI
5:    MUL   AX, n
6:    JMP   2
7:    MOV   n, AX
8:    RET
```

Lập trình tuần tư: Ưu điểm và nhược điểm

- Ưu điểm:
  - Tư duy đơn giản
  - Lập trình ở mức trừu tượng thấp, nên dễ kiểm soát sử dụng tài nguyên
  - Có thể có hiệu suất cao
  - Có thể thích hợp với bài toán nhỏ, lập trình nhúng, lập trình hệ thống
- Nhược điểm:
  - Chương trình khó theo dõi -> dễ mắc lỗi
  - Khó sử dụng lạI
  - Hiệu quả lập trình thấp
  - Không thích hợp với ứng dụng qui mô lớn

Lập trình có cấu trúc (structured programming)

- Cấu trúc hóa dữ liệu (xây dựng kiểu dữ liệu) và cấu trúc hóa chương trình để tránh các lệnh nhảy.
- Phân tích và thiết kế theo cách từ trên xuống (top-down)
- Thực hiện từ dưới lên (bottom-up)
- Yêu cầu của chương trình có cấu trúc: chỉ sử dụng các cấu trúc điều khiển tuần tự, tuyển chọn (if then else), lặp (while) và thoát ra (exit).
- Ví dụ các ngôn ngữ đặc thù:
  - PASCAL, ALGO, FORTRAN, C,...
  - SFC (Sequential Funtion Charts)
  - ST (Structured Text)

Lập trình có cấu trúc: Ví dụ tính giai thừa (PASCAL)

```
FUNCTION Factorial(n: INTEGER) : INTEGER
VAR X: INTERGER;
BEGIN
  X := n;
  WHILE (n > 1) DO
    BEGIN
       DEC(n);
       X := X * n;
    END
    Factorial := X;
  END
END;
```

Lập trình có cấu trúc: Ví dụ quản lý sinh viên

```
struct Date { int Day, Month, Year; };
struct Student
{
    string name;
    Date   dob;
    int    code;
};
typedef Student* Students; // cấu trúc mảng
Students create(int max_items, int item_size );
void destroy(Students lop);
void add(Students lop, Student sv);
void delete(Students lop, Student sv);
Student find(Students lop, int code);
```

Lập trình module (modular programming)

- Lập trình module là một dạng cải tiến của lập trình có cấu trúc. Chương trình được cấu trúc nghiêm ngặt hơn, dùng đơn vị cấu trúc là module.
- Module:
  - Một đơn vị cấu trúc độc lập, được chuẩn hóa dùng để tạo lập một hệ thống.
  - Mỗi module bao gồm phần giao diện (mở) và phần thực hiện (che giấu)
  - Các module giao tiếp với nhau thông qua các giao diện được đặc tả rất chính xác.
- Ví dụ ngôn ngữ tiêu biểu:
  - Modula-2, xây dựng trên cơ sở PASCAL, do Niclaus Wirth thiết kế năm 1977.

Lập trình hướng đối tượng (object-oriented programming)

- Xây dựng chương trình ứng dụng theo quan điểm dựa trên các cấu trúc dữ liệu trừu tượng (lớp), các thể nghiệm của các cấu trúc đó (đối tượng) và quan hệ giữa chúng (quan hệ lớp, quan hệ đối tượng).
- Ba nguyên lý cơ bản:
  - Đóng gói dữ liệu (data encapsulation)
  - Dẫn xuất/thừa kế (subtyping/inheritance)
  - Đa hình/đa xạ (polymorphism)
- Ví dụ ngôn ngữ hỗ trợ tiêu biểu:
  - C++, C#
  - Java,
  - ADA,
  - ...

Ví dụ minh họa: Quản lý sinh viên (C++)

```
class Date {
   int Day, Month, Year;
public:
   void setDate(int, int, int);
   ...
};
class Student {
   string name;
   Date   dob;
   int     code;
public:
   Student(string n, Date d, int c);
   ...
};
class StudentList {
   Student* list;
public:
   void addStudent(Student*);
   ...
};
```

Ví dụ minh họa: Tính toán kiểu MATLAB

```
Vector a(10, 1.0), b(10, 0.5);
Vector c = a + b;
...
Vector d = a - b + 2*c;
Matrix A(4,4), B(4,2), C(2,4), D(2,2);
Vector x(4),u(2),y(2);
...
while (true) {
  // đọc đầu vào u
  y = C*x + D*u;
  x = A*x + B*u;
  // đưa đầu ra y
}
...
CTFMatrix G = ss2tf(A,B,C,D);
...
```

Lập trình tổng quát (generic programming)

- Một tư duy lập trình mở, trên quan điểm tổng quát hóa tất cả những gì có thể nhằm đưa ra một **khuôn mẫu giải pháp** cho nhiều bài toán lập trình cụ thể.
- Ưu điểm:
  - Giảm tối đa lượng mã nguồn
  - Tăng nhiều lần giá trị sử dụng lại của phần mềm
  - Có thể kết hợp tùy ý với các phương pháp luận khác
  - Tính khả chuyển cao
- Các hình thức tổng quát hóa:
  - Kiểu dữ liệu
  - Phép toán cơ bản
  - Cấu trúc dữ liệu
  - Quản lý bộ nhớ,...

Ví dụ minh họa: Các cấu trúc toán học

```
typedef TMatrix<double> Matrix;
typedef TMatrix<complex<double> > ComplexMatrix;
Matrix a(4,4), b(4,4);
Matrix c = a*b;
ComplexMatrix a1(4,4), b1(4,4);
ComplexMatrix c1 = a1*b1;

typedef TPoly<double> Poly;
typedef TMatrix<Poly> PolyMatrix;
typedef TPoly<ComplexMatrix> ComplexMatrixPoly;

TRational<int>  IntRational;
TRational<Poly> PolyRational;
...
```

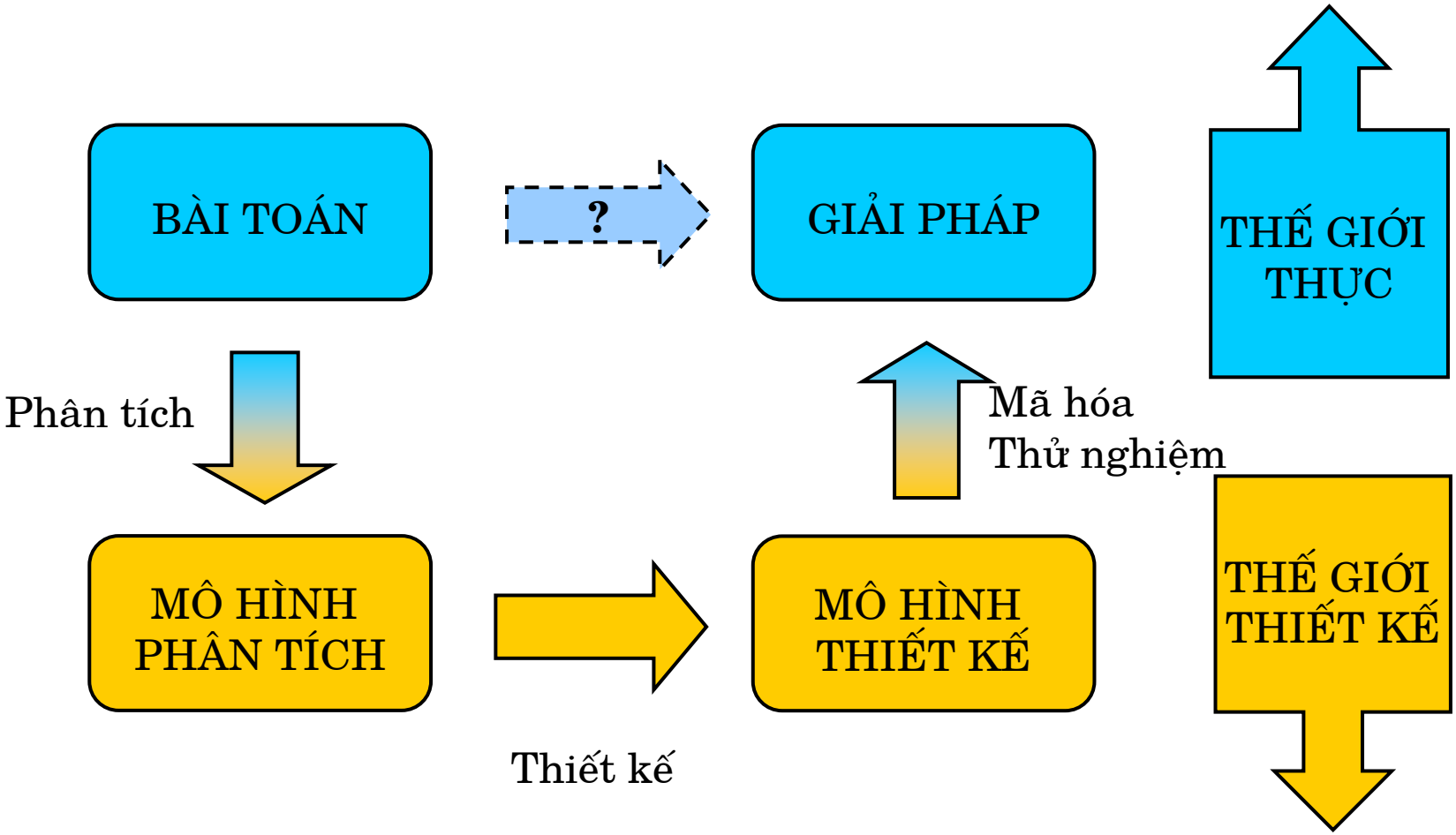
Lập trình thành phần (component-based programming)

- Phương pháp xây dựng phần mềm dựa trên các thành phần "IC" có sẵn, hoặc tạo ra các IC đó.
- Tiến hóa từ lập trình hướng đối tượng
- Hầu hết các ứng dụng Windows và ứng dụng Internet ngày nay được xây dựng theo phương pháp luận này
- Các ngôn ngữ tiêu biểu
  - C/C++, C#
  - Delphi, Visual Basic
  - Script, HMTL, XML,...
  - FBD

Lập trình thời gian thực (real-time programming)

- Xây dựng phần mềm đáp ứng tính năng thời gian thực của hệ thống, ví dụ các hệ thống điều khiển
- Đặc thù:
  - Lập trình cạnh tranh (đa nhiệm, đa luồng)
  - Cơ chế xử lý sự kiện
  - Cơ chế định thời
  - Đồng bộ hóa quá trình
  - Hiệu suất cao
- Ngôn ngữ lập trình: ASM, C/C++, ADA,...
- Cần sự hỗ trợ của nền cài đặt
  - Hệ điều hành
  - Nền phần cứng
  - Mạng truyền thông

1.4 Qui trình công nghệ phần mềm



Phân tích yêu cầu (Requirement analysis)

- Bởi vì: Khách hàng thường không biết là họ muốn gì, nhưng họ biết chắc chắn là họ không muốn gì
- Cho nên: Cần phải cùng với khách hàng làm rõ những yêu cầu về phạm chức năng, về giao diện sử dụng
- Kết quả: Mô hình đặc tả (*Specification Model*), một phần của hợp đồng
- Cần một ngôn ngữ mô hình hóa dễ hiểu để trao đổi giữa khách hàng và nhóm phân tích

⇨ Trả lời câu hỏi: **Khách hàng cần những gì?**

Phân tích hệ thống (System analysis)

- Phân tích mối liên hệ của hệ thống với môi trường xung quanh
- Tìm ra cấu trúc hệ thống và các thành phần quan trọng
- Định nghĩa chức năng cụ thể của các thành phần
- Nhận biết các đặc điểm của từng thành phần
- Phân loại các thành phần, tổng quát hóa, đặc biệt hóa
- Nhận biết mối liên hệ giữa các thành phần
- Kết quả: Mô hình hệ thống (*System model*)
- Cần một ngôn ngữ mô hình hóa để trao đổi giữa các thành viên trong nhóm phân tích và với nhóm thiết kế

⇨ Trả lời câu hỏi: **Những gì sẽ phải làm?**

Thiết kế hệ thống (System Design)

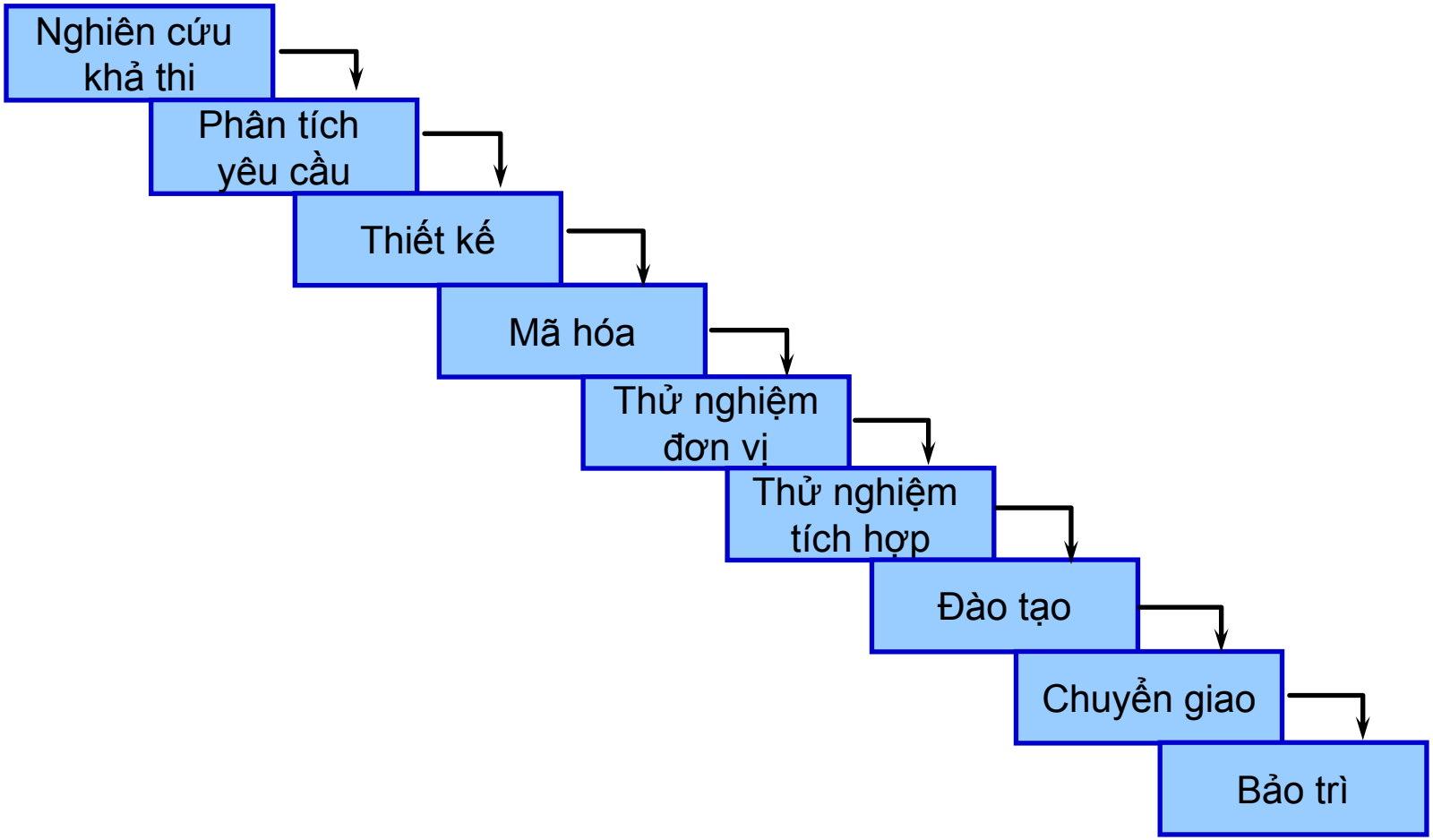
- Dựa trên mô hình hệ thống, xây dựng các mô hình chi tiết phục vụ sẵn sàng mã hóa/cài đặt
- Bao gồm:
  - Thiết kế cấu trúc (*structured design*): chương trình, kiểu dữ liệu, đối tượng, quan hệ cấu trúc giữa các đối tượng và kiểu)
  - Thiết kế tương tác (*interaction design*): quan hệ tương tác giữa các đối tượng
  - Thiết kế hành vi (*behaviour design*): sự kiện, trạng thái, phép toán, phản ứng
  - Thiết kế chức năng (*funtional design*): tiến trình hành động, hàm, thủ tục)
- Kết quả: Mô hình thiết kế (các bản vẽ và lời văn mô tả)

⇨ Trả lời câu hỏi: **Làm như thế nào?**

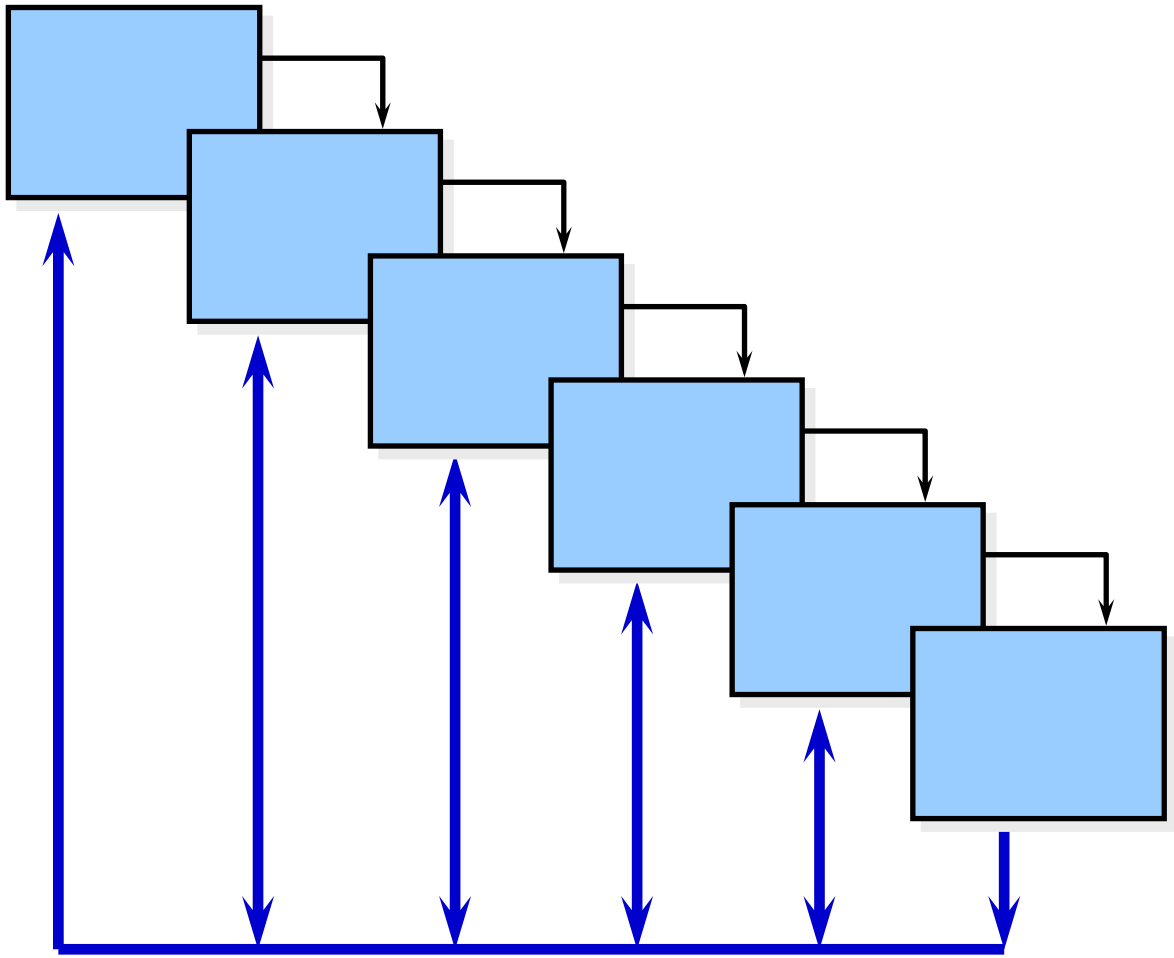
Các bước khác

- Mã hóa/cài đặt (*Coding*/*Implementation*): Thể hiện mô hình thiết kế với một ngôn ngữ lập trình cụ thể
- Thử nghiệm (*Testing, Verification*): Chạy thử, phân tích và kiểm chứng:
  - Thử đơn vị (*Unit Test*)
  - Thử tích hợp (*Integration Test*)
- Gỡ rối (*Debugging*): Tìm ra và sửa các lỗi chương trình chạy (các lỗi logic)
- Xây dựng tài liệu (*Documenting*): Xây dựng tài liệu phát triển, tài liệu hướng dẫn sử dụng
- Đào tạo, chuyển giao
- Bảo trì, bảo dưỡng

Chu trình cổ điển: "Waterfall Model"



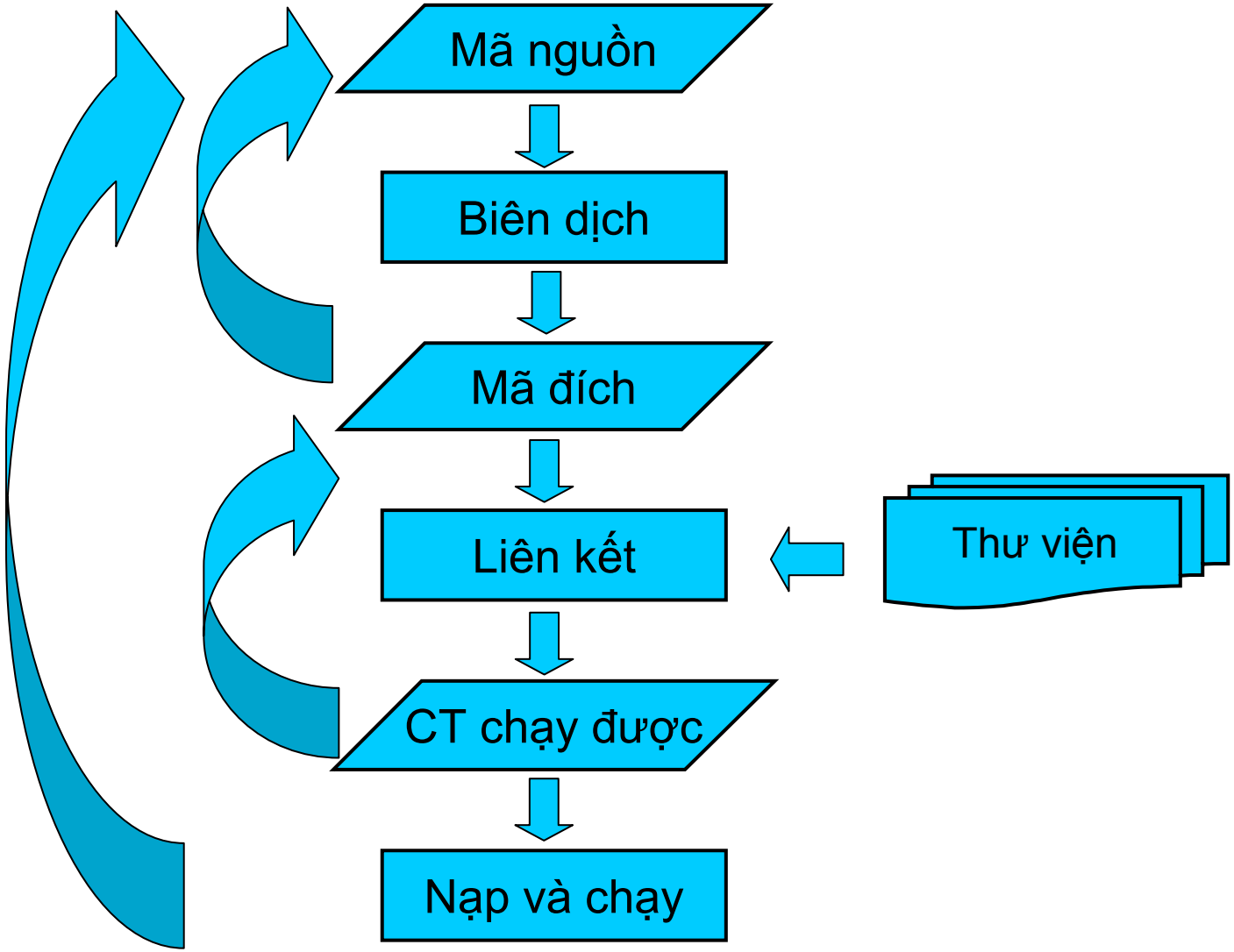
Xu thế hiện nay: Song song và lặp



Lập trình là gì, nằm ở đâu?

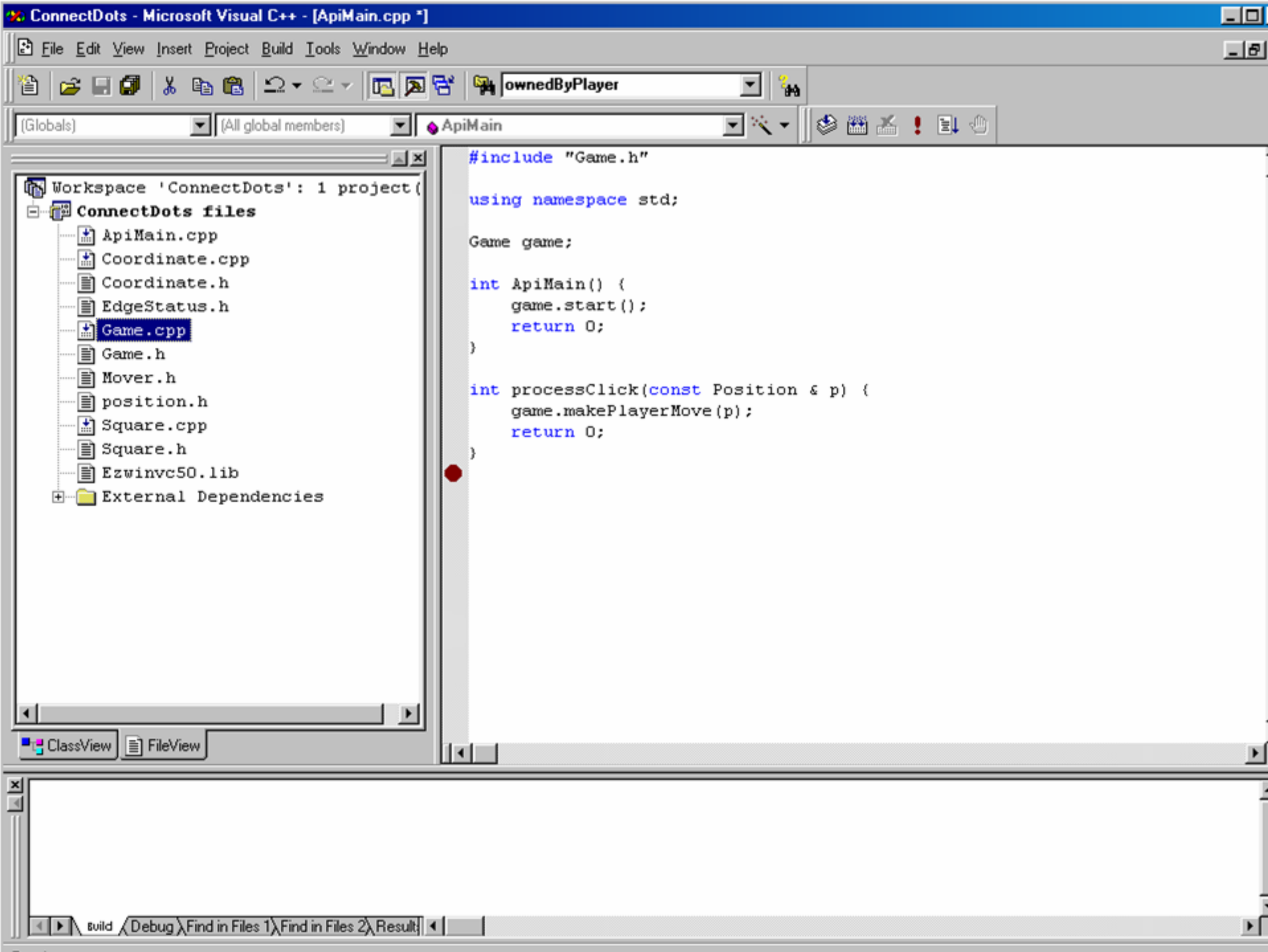
- Lập trình > Mã hóa
- Lập trình ≈ Tư tưởng thiết kế + Mã hóa + Thử nghiệm + Gỡ rối

Các bước phát triển chương trình



Môi trường/công cụ phát triển

- IDE (Integrated Development Environment)
  - Hỗ trợ toàn bộ các bước phát triển chương trình
  - Ví dụ: MS Visual C++, Borland C++ (Builder), Keil-C
- Các công cụ tiêu biểu
  - Trình soạn thảo (Editor)
  - Trình biên dịch (Compiler)
  - Trình liên kết (Linker)
  - Trình nạp (Loader)
  - Trình gỡ rối (Debugger)
  - Trình quản lý dự án (Project Manager)


ConnectDots - Microsoft Visual C++ - [ApiMain.cpp *]
File Edit View Insert Project Build Tools Window Help
ownedByPlayer
(Globals)
(All global members)
ApiMain
Workspace 'ConnectDots': 1 project(
ConnectDots files
ApiMain.cpp
Coordinate.cpp
Coordinate.h
EdgeStatus.h
Game.cpp
Game.h
Mover.h
position.h
Square.cpp
Square.h
Ezwinvc50.lib
External Dependencies
ClassView
FileView
#include "Game.h"
using namespace std;
Game game;
int ApiMain() {
game.start();
return 0;
}
int processClick(const Position & p) {
game.makePlayerMove(p);
return 0;
}
Build
Debug
Find in Files 1
Find in Files 2
Result

1.5 Sơ lược về C/C++

Lược sử ngôn ngữ C

- Tiến hóa từ hai ngôn ngữ lập trình
  - BCPL và B: Các ngôn ngữ "phi kiểu"
- Dennis Ritchie (Bell Laboratories, AT&T)
  - Bổ sung kiểu hóa dữ liệu và các yếu tố khác
- Ngôn ngữ phát triển hệ điều hành UNIX
- Không phụ thuộc phần cứng
  - Tính khả chuyển
- 1989: ANSI chuẩn hóa (ANSI-C)
- 1990: Công bố chuẩn ANSI và ISO
  - ANSI/ISO 9899: 1990

Lược sử ngôn ngữ C++

- Mở rộng, tiến hóa từ C
- Bjarne Stroustrup (Bell Laboratories)
  - Đầu những năm 1980: “C with classes”
  - 1984: Tên C++
  - 1987: “The C++ Programming Language” 1st Edition
  - 1997: “The C++ Programming Language” 3rd Edition
  - Chuẩn hóa quốc tế: ANSI/ISO 1996
- Bổ sung các đặc tính hỗ trợ:
  - Lập trình hướng đối tượng
  - Lập trình tổng quát
  - Lập trình toán học,...
- Ngôn ngữ “lai”

Tại sao chọn C/C++

- Đáp ứng các yêu cầu:
  - Gần gũi với phần cứng
  - Hiệu suất cao
  - Tương đối thân thiện với người lập trình
  - Khả chuyển
  - Chuẩn hóa quốc tế (tương lai vững chắc)
- Thế mạnh tuyệt đối của ANSI-C:
  - Phổ biến cho hầu hết các nền vi xử lý, vi điều khiển, DSP
  - Phổ biến cho "mỗi người lập trình" trên thế giới
- Thế mạnh tuyệt đối của ANSI/ISO C++:
  - Lập trình hướng đối tượng
  - Lập trình tổng quát (template)
  - Lập trình toán học (dữ liệu trừu tượng và nạp chồng toán tử)

Visual C++, .NET & C#

- Visual C++:
  - Môi trường/công cụ lập trình C++ của Microsoft
  - Mở rộng một số yếu tố
  - Thư viện lập trình Windows: Microsoft Foundation Classes (MFC), Active Template Library (ATL)
  - Các thư viện chung: GUI, graphics, networking, multithreading, …
- .NET (“dot net”)
  - Kiến trúc nền tảng phần mềm lập trình phân tán
  - Hướng tới các ứng dụng Web, phân tán trên nhiều chủng loại thiết bị khác nhau
  - Các ứng dụng trên nhiều ngôn ngữ khác nhau có thể giao tiếp một cách đơn giản trên một nền chung
  - Phương pháp luận: Lập trình thành phần

Visual C++, .NET & C#

- C#
  - Anders Hejlsberg và Scott Wiltamuth (Microsoft)
  - Thiết kế riêng cho nền .NET
  - Nguồn gốc từ C, C++ và Java
  - Điều khiển theo sự kiện, hoàn toàn hướng đối tượng, ngôn ngữ lập trình hiển thị
  - Integrated Development Environment (IDE)
  - Tương tác giữa các ngôn ngữ

Chúng ta đã học được những gì?

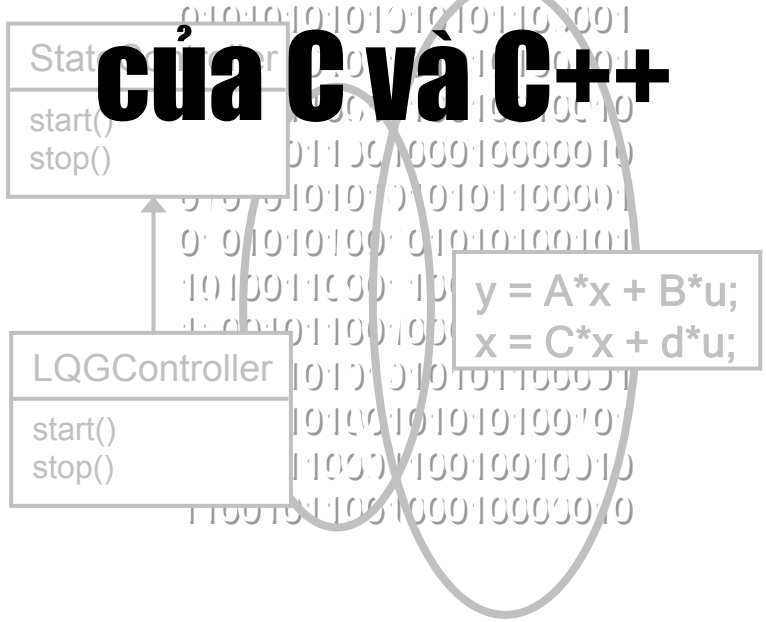
- Biết được những gì sẽ phải học, học để làm gì và phải học như thế nào
- Hàng loạt khái niệm mới xung quanh kỹ thuật lập trình và qui trình công nghệ phần mềm
- Tổng quan về các kỹ thuật lập trình
- Lược sử ngôn ngữ C/C++, thế mạnh của chúng so với các ngôn ngữ khác

Chủ đề tiếp theo: C/C++ cơ sở

- Tổ chức chương trình/bộ nhớ
- Dữ liệu và biến
- Toán tử, biểu thức và câu lệnh
- Điều khiển chương trình: vòng lặp, rẽ nhánh
- Mảng và con trỏ
- Cấu trúc

Kỹ thuật lập trình

Chương 2: Các yếu tố cơ bản của C và C++



8/29/2005

Nội dung chương 2

2.1 Tổ chức chương trình C/C++

- Cấu trúc và các phần tử cơ bản của một chương trình viết trên C/C++
- Qui trình tạo ra một chương trình chạy được:
  - Vấn đề tạo dự án
  - Qui tắc soạn thảo mã nguồn
  - Biên dịch từng phần và sửa các loại lỗi biên dịch
  - Liên kết và sử dụng thư viện, sửa lỗi liên kết
  - Chạy thử và gỡ rối (Debug)
- Sơ lược về tổ chức bộ nhớ

Chương trình tính giai thừa: Phiên bản C

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

int factorial(int);

void main() {
     char c = 'N';
     int  N = 1;
     int  kq;
     do {
        printf("\nEnter a number > 0:");    /* writing on the screen */
        scanf("%d",&N);                     /* reading from keyboard to N */
        kq = factorial(N);                  /* calling function with argument N */
        printf("\nFactorial of %d is %d", N, kq); /*write result on screen */
        printf("\nPress 'Y' to continue or any other key to stop");
        c = getch();                        /* reading a character from keyboard*/
     } while (c=='y' || c=='Y');            /* checking loop condition */
}

int factorial(int n) {
     int kq = 1;
     while (n > 1)
        kq *= n--;
     return kq;
}
```

Lệnh tiền xử lý: Khai báo sử dụng hàm thư viện

Khai báo hàm

Chương trình chính

Khai báo biến

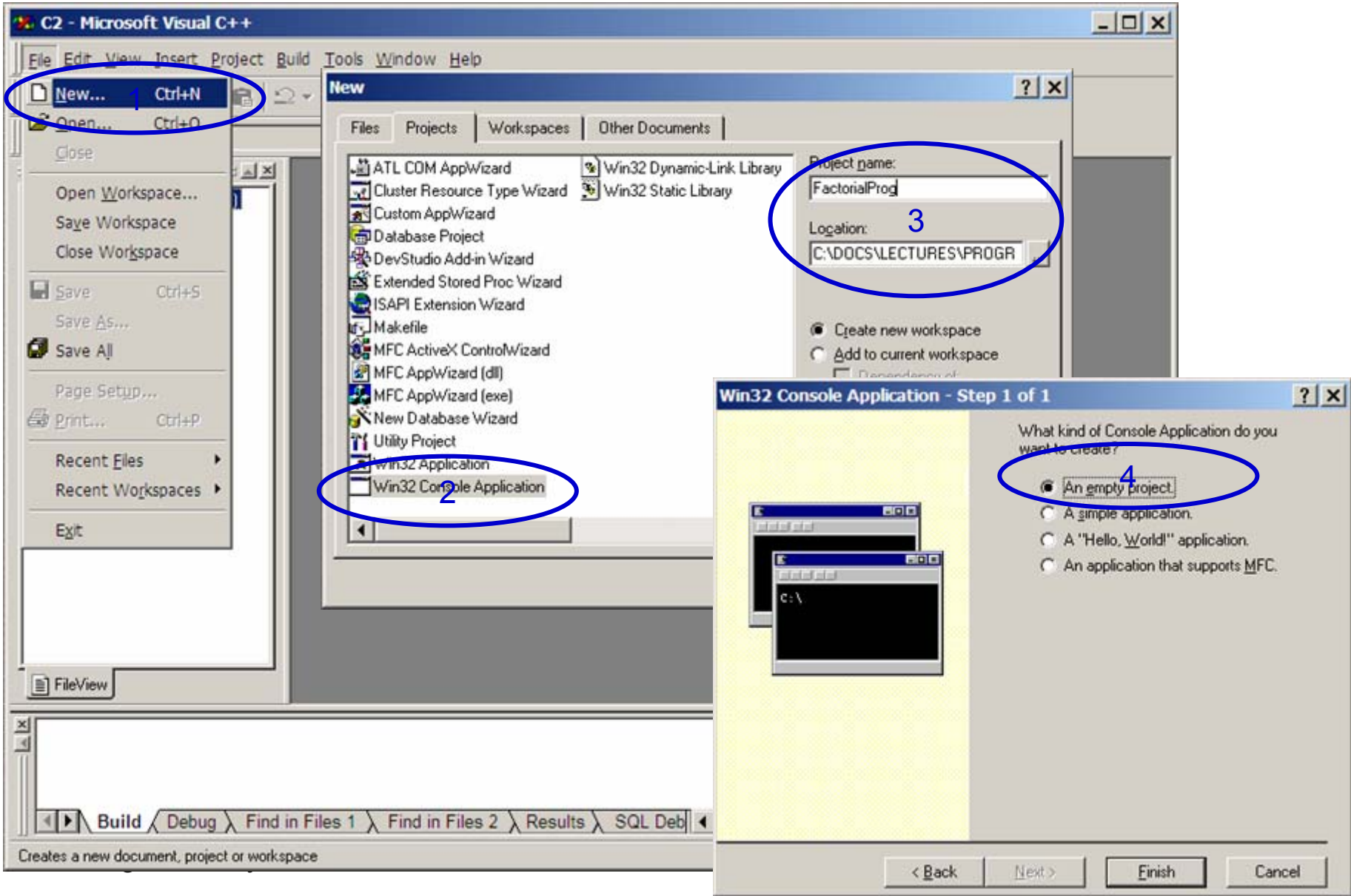
Lời chú thích

Định nghĩa hàm (thân hàm)

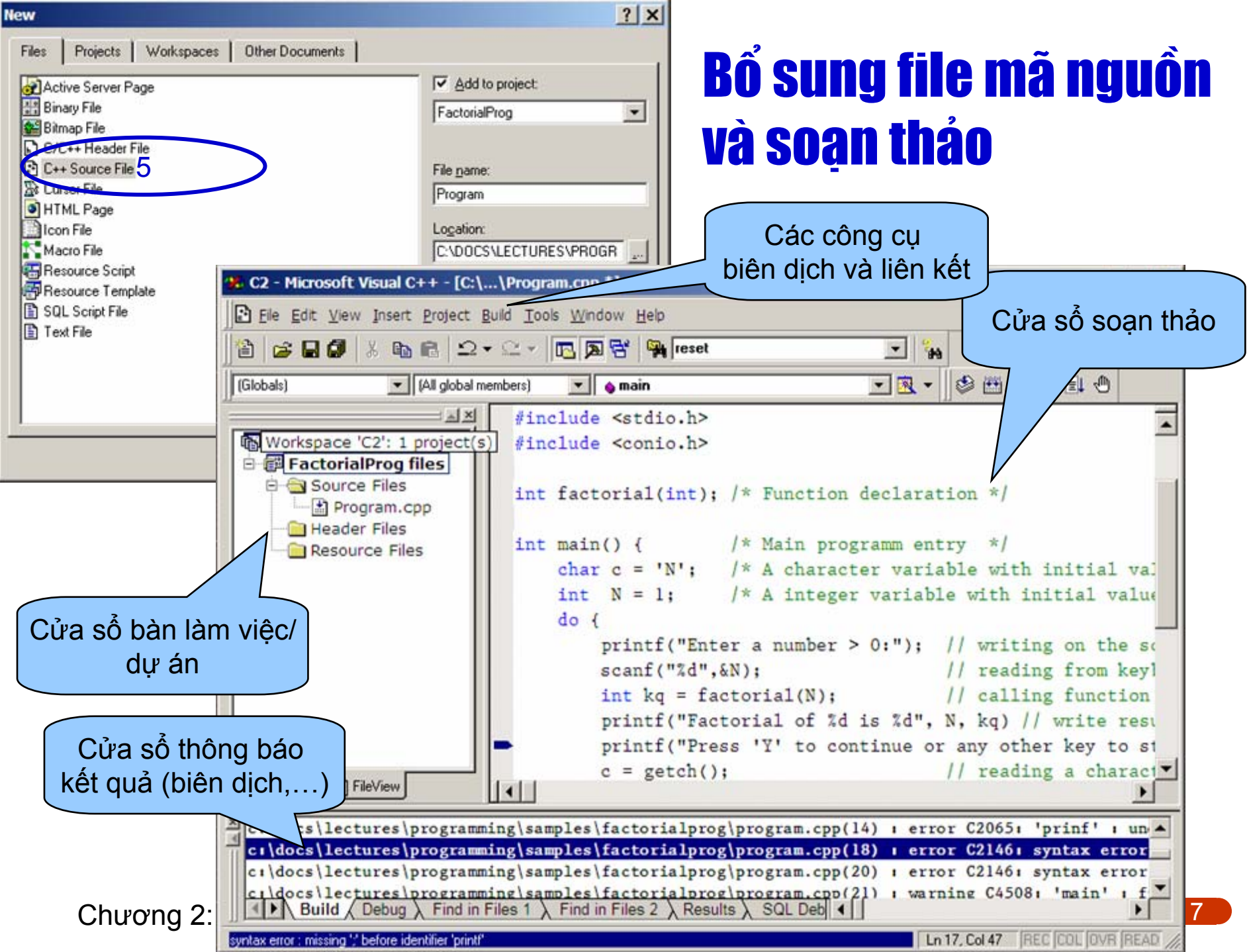
Chương trình tính giai thừa: Phiên bản C++

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

int factorial(int);

void main() {
     char c = 'N';
     int  N = 1;
     do {
          cout << “\nEnter a number > 0:“ // writing on the screen
          cin  >> N;                     // reading from keyboard to N
          int kq = factorial(N);         // calling function with argument
          cout << “\nFactorial of ” << N << “ is “ << kq
          cout << “\nPress 'Y' to continue or any other key to stop";
          c = getch();                   // reading a character from keyboard
     } while (c == 'y' || c == 'Y');     // checking loop condition
}

int factorial(int n) {
     int kq = 1;
     while (n > 1)
          kq *= n--;
     return kq;
}
```

Tạo dự án


C2 - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
New... Ctrl+N
Open... Ctrl+O
Close
Open Workspace...
Save Workspace
Close Workspace
Save Ctrl+S
Save As...
Save All
Page Setup...
Print... Ctrl+P
Recent Files
Recent Workspaces
Exit
1
New
Files
Projects
Workspaces
Other Documents
ATL COM AppWizard
Cluster Resource Type Wizard
Custom AppWizard
Database Project
DevStudio Add-in Wizard
Extended Stored Proc Wizard
ISAPI Extension Wizard
Makefile
MFC ActiveX ControlWizard
MFC AppWizard (dll)
MFC AppWizard (exe)
New Database Wizard
Utility Project
Win32 Application
Win32 Console Application
Win32 Dynamic-Link Library
Win32 Static Library
2
Project name:
FactorialProg
Location:
C:\DOCS\LECTURES\PROGR
3
Create new workspace
Add to current workspace
FileView
Build
Debug
Find in Files 1
Find in Files 2
Results
SQL Deb
Creates a new document, project or workspace
Win32 Console Application - Step 1 of 1
What kind of Console Application do you want to create?
An empty project
A simple application.
A "Hello, World!" application.
An application that supports MFC.
4
c:\
< Back
Next >
Finish
Cancel

Bổ sung file mã nguồn và soạn thảo



Qui tắc soạn thảo mã nguồn

1. Tên biến, tên hàm, tên kiểu mới:
   - Tránh sử dụng các từ khóa và tên kiểu cơ sở
   - Các ký tự dùng được: 'A'..'Z', 'a'..'z', '0'..'9', '_'
   - Phân biệt giữa chữ hoa và chữ thường: **n** khác **N**
   - Ngắn nhưng đủ khả năng phân biệt, nhận biết
   - Sử dụng tiếng Anh hoặc tiếng Việt không dấu (kể cả dòng chú thích)
2. Sau mỗi câu lệnh có chấm phảy;
3. Đoạn { … } được coi là nhóm lệnh, không có dấu chấm phảy sau đó, trừ khi khai báo kiểu
4. Cấu trúc mã nguồn theo kiểu phân cấp => dễ đọc
5. Bổ sung chú thích hợp lý (/* …*/ hoặc //)
6. Chia một file lớn thành nhiều file nhỏ

Các từ khóa trong C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

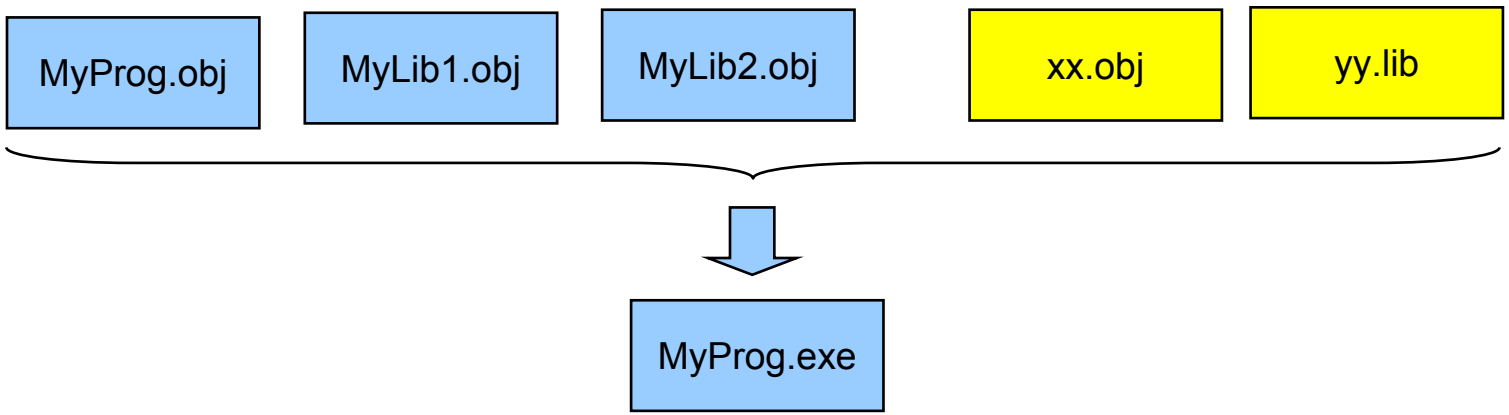
Từ khóa trong C++

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

Biên dịch (compile)

- Biên dịch từng file nguồn riêng biệt (*.c: C compiler, *.cpp: C++ compiler), kết quả => *.obj
- Trong Visual C++: Gọi **Compile** (**Ctrl + F7**) để biên dịch riêng rẽ hoặc **Build** (**F7**) để kết hợp biên dịch và liên kết cho toàn bộ dự án
- Các kiểu lỗi biên dịch (compile error):
  - Lỗi cú pháp: Sử dụng tên sai qui định hoặc chưa khai báo, thiếu dấu chấm phảy ;, dấu đóng }
  - Lỗi kiểu: Các số hạng trong biểu thức không tương thích kiểu, gọi hàm với tham số sai kiểu
  - …
- Các kiểu cảnh báo biên dịch (warning):
  - Tự động chuyển đổi kiểu làm mất chính xác
  - Hàm khai báo có kiểu trả về nhưng không trả về
  - Sử dụng dấu = trong trường hợp nghi vấn là so sánh ==
  - …

Liên kết (link)

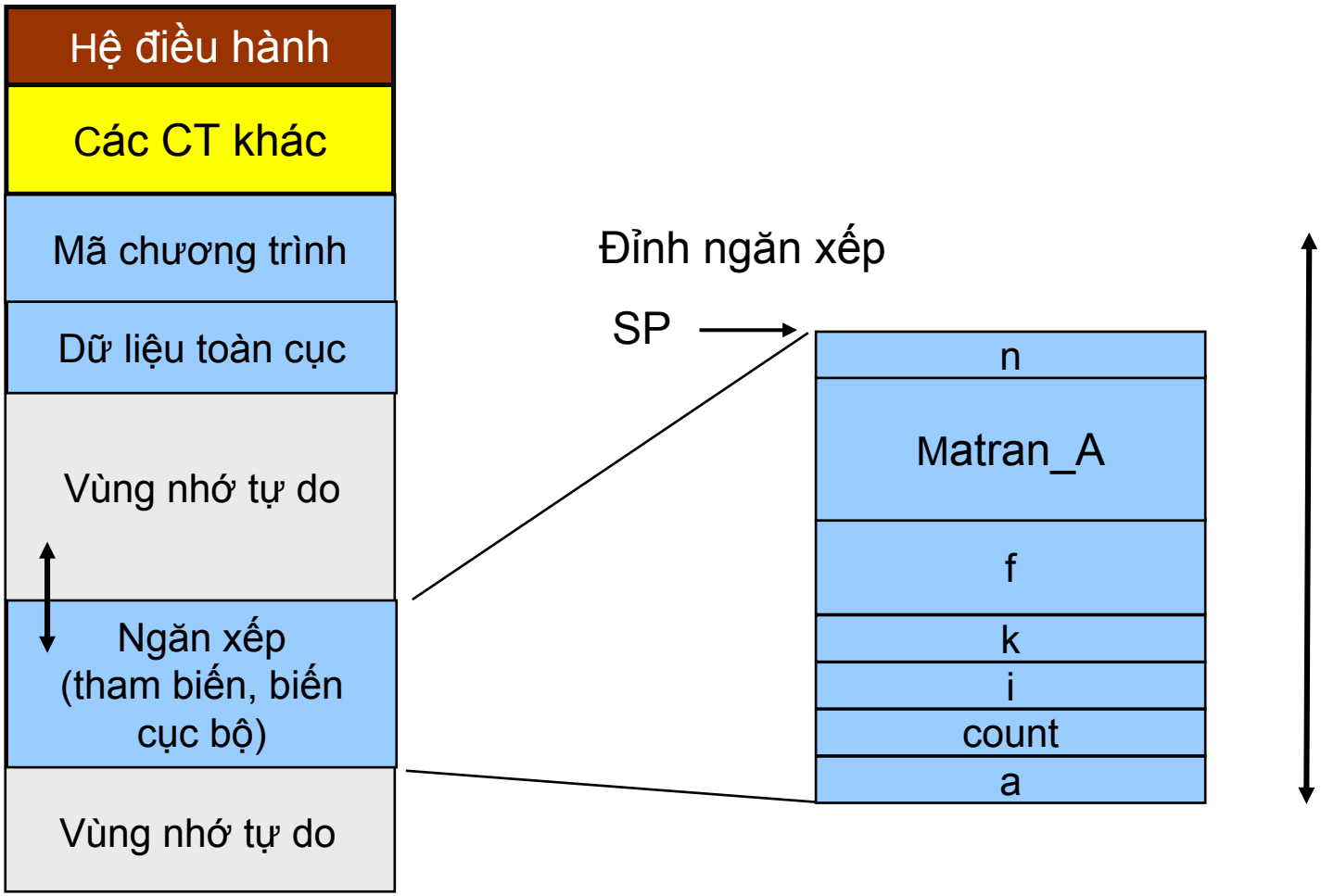


- Liên kết là quá trình ghép nhiều file đích (*.obj, *.lib) để tạo ra chương trình chạy cuối cùng *.exe
- Trong Visual C++: Gọi **Build** (**F7**)
- Lỗi liên kết có thể là do:
  - Sử dụng hàm nhưng không có định nghĩa hàm
  - Biến hoặc hàm được định nghĩa nhiều lần
  - …

Chạy thử và gỡ rối (debug)

- Chạy thử trong Visual C++: **Execute** hoặc **Ctrl+F5**
- Tìm lỗi:
  - Lỗi khi chạy là lỗi thuộc về phương pháp, tư duy, thuật toán, không phải về cú pháp
  - Lỗi khi chạy bình thường không được báo
  - Lỗi khi chạy rất khó phát hiện, vì thế trong đa số trường hợp cần tiến hành **debug**.
- Chạy Debug trong Visual C++:
  - Chạy tới chỗ đặt cursor: Ctrl+F10
  - Chạy từng dòng lệnh: F10
  - Chạy vào trong hàm: F11
  - Chạy tiếp bình thường: F5
  - Xem kết quả dưới cửa sổ Output hoặc gọi QuickWatch

Tổ chức bộ nhớ



2.2 Biến và dữ liệu

- Biểu thức = dữ liệu + phép toán + …
- Biểu diễn dữ liệu: Thông qua **biến** hoặc **hằng số,** kèm theo **kiểu**
- Nội dung trong phần này:
  - Các kiểu dữ liệu cơ bản
  - Các phép toán áp dụng
  - Tương thích và chuyển đổi kiểu
  - Khai báo biến, phân loại biến

2.2.1 Các kiểu dữ liệu cơ bản của C/C++

Kiểu	Kích cỡ thông dụng (tính bằng bit)	Phạm vi tối thiểu
char	8	-127 to 127
signed char	8	-127 .. 127
unsigned char	8	0 .. 255
int	16/32	-32767 .. 32767
signed int	16/32	-nt-
unsigned int	16/32	0 .. 65535
short	16	-32767 .. 32767
signed short	16	nt
unsigned short	16	0 .. 65535
long	32	-2147483647 .. 2147483647
signed long	32	nt
unsigned long	32	0 .. 4294967295
float	32	Độ chính xác 6 chữ số
double	64	Độ chính xác 10 chữ số
long double	80	Độ chính xác 10 chữ số
bool (C++)	-	-
wchar_t (C++)	16	-32767 .. 32767

Các phép toán cơ bản

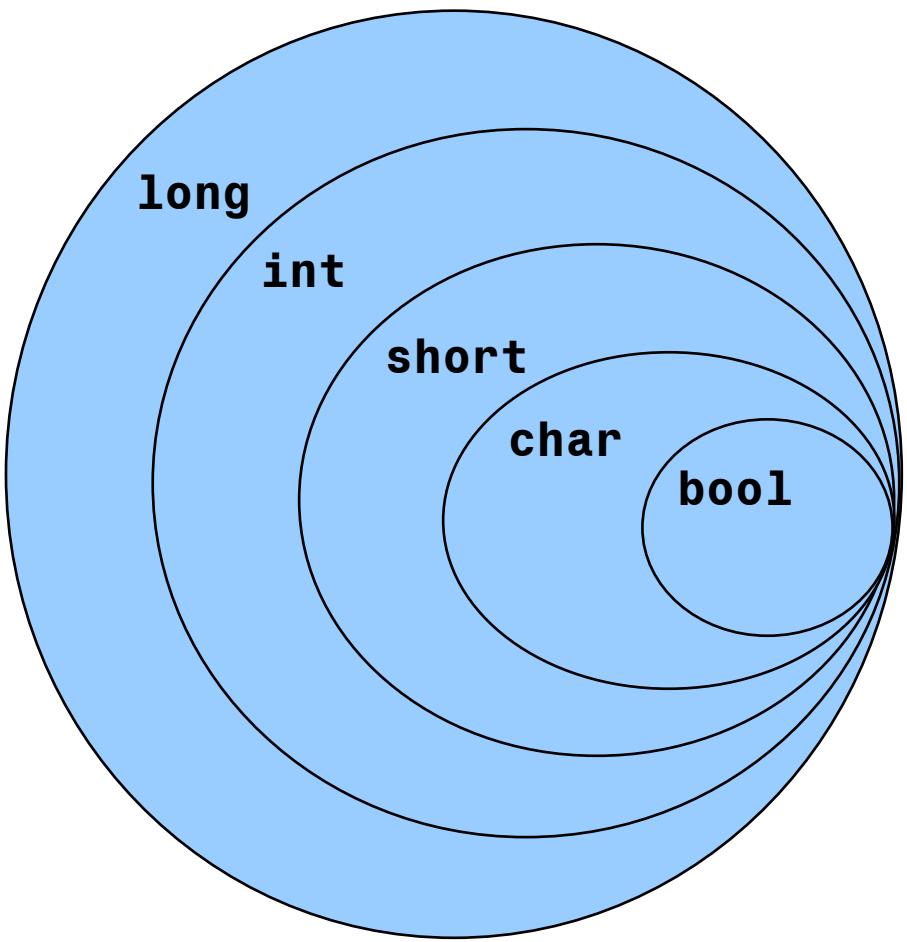
Phép toán	Ký hiệu	Kiểu nguyên	Kiểu số thực	Kiểu bool
Gán	=	X	X	X
Số học	+, -, *, /, +=, -=, *=, /=	X	X	x
	%, %=	X		x
	++, --	X		x
So sánh	>, <, >=, <=, ==, !=	X	X	X
Logic	&&, ||, !	X	X	X
Logic bit	&, |, ^, ~ &=, |=, ^=	X		x
Dịch bit	<<,>>, <<=,>>=	X		x
Lựa chọn	? :	X	X	X
Lũy thừa?	**Không có!**			

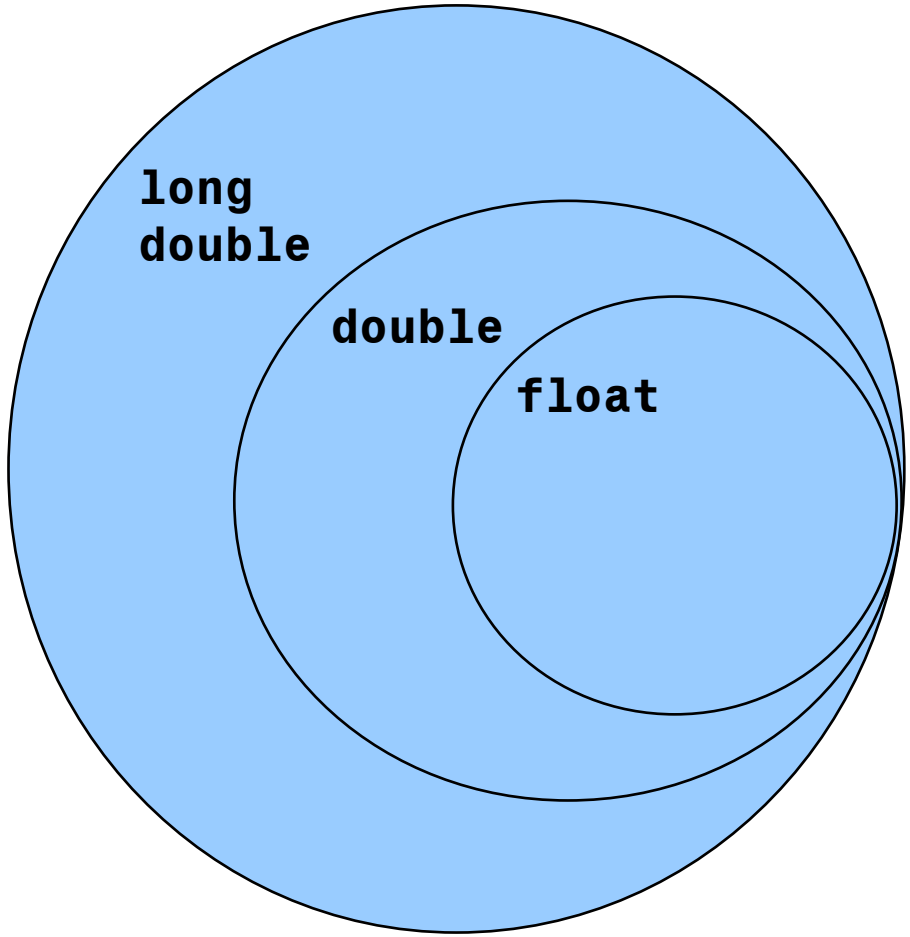
Tương thích và chuyển đổi kiểu

- Tương thích kiểu => Tự động chuyển đổi kiểu
  - Giữa các kiểu số nguyên với nhau (lưu ý phạm vi giá trị)
  - Giữa các kiểu số thực với nhau (lưu ý độ chính xác)
  - Giữa các kiểu số nguyên và số thực (lưu ý phạm vi giá trị và độ chính xác)
  - Kiểu bool sang số nguyên, số thực: true => 1, false => 0
  - Số nguyên, số thực sang kiểu bool: ≠ 0 => true, 0 => false
- Nếu có lỗi hoặc cảnh báo => khắc phục bằng cách ép chuyển đổi kiểu:
  - VD:

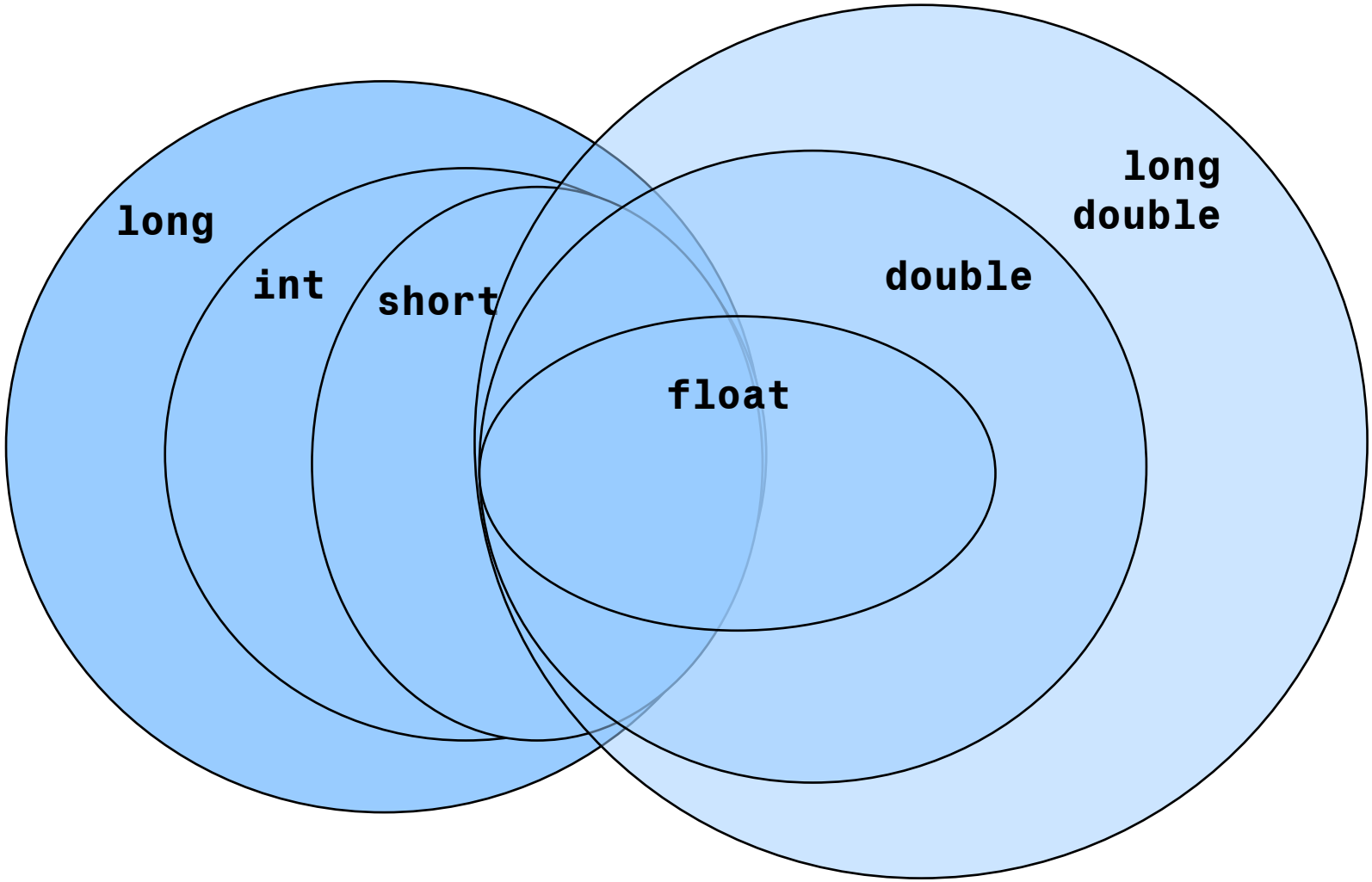
```
i = int(2.2) % 2;
j = (int)2.2 + 2; // C++
```

Nhìn nhận về chuyển đổi kiểu





Nhìn nhận về chuyển đổi kiểu



2.2.2 Khai báo biến

```
char     c = 'N';
bool     b = true;
int      kq;
double   d;
long     count, i=0;
unsigned vhexa=0x00fa;
unsigned voctal=082;
```

Khai báo và khởi tạo giá trị

Chỉ khai báo, giá trị bất định

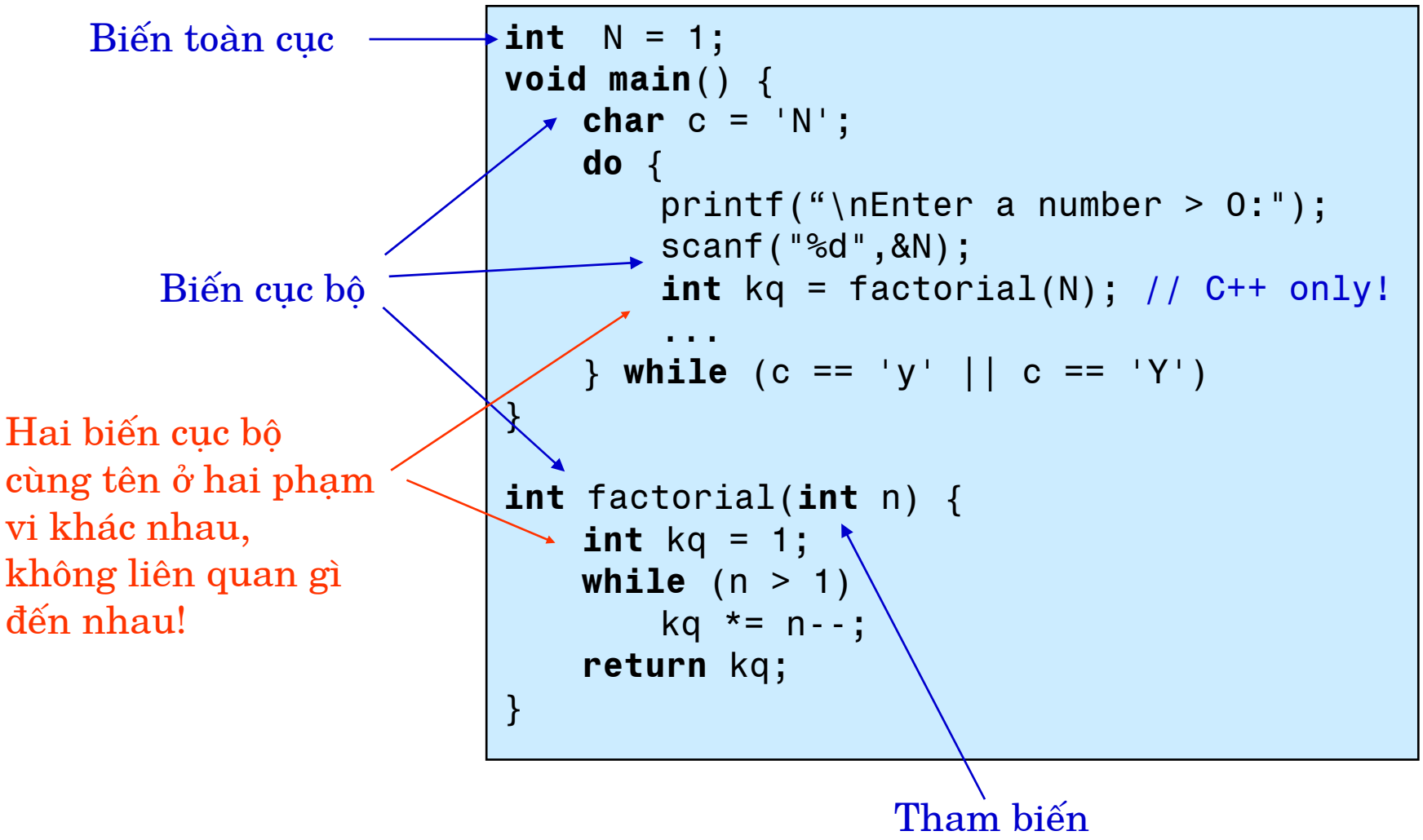
Khai báo kết hợp, chỉ i=0

Đặt giá trị đầu hexa

Đặt giá trị đầu octal -> 66 chứ không phải 82

- C: Toàn bộ biến phải khai báo ngay đầu thân hàm
- C++: Có thể khai báo tại chỗ nào cần
- Phân loại biến:
  - Biến toàn cục: Khai báo ngoài hàm, lưu giữ trong vùng nhớ dữ liệu chương trình
  - Biến cục bộ: Khai báo trong thân hàm, lưu giữ trong ngăn xếp
  - Tham biến: Khai báo trên danh sách tham số của hàm, lưu giữ trong ngăn xếp

Ví dụ khai báo các loại biến



Đặc tính lưu giữ

- Biến extern: Khai báo sử dụng biến toàn cục đã được định nghĩa trong một tập tin khác

```
/* file1.c */
int x, y;
char ch;
void main()
{
     /* ... */
}
void func1(void)
{
     x = 123;
}
```

```
/* file2.c */
extern int x, y;
extern char ch;
void func22()
{
     x = y / 10;
}
void func23()
{
     y = 10;
}
```

- Biến static: được lưu trữ trong bộ nhớ dữ liệu CT
  - Biến static cục bộ: hạn chế truy nhập từ bên ngoài hàm
  - Biến static toàn cục: hạn chế truy nhập từ file khác

2.2.3 Hằng số (trực kiện)

Kiểu	Ví dụ
int	1 123 21000 -234 0x0A 081
long int	35000L -34l -234L 0x0AL 081L
unsigned int	10000U 987u 40000u
float	123.23F 4.34e-3f .1f
double	123.23 1.0 -0.9876324 .1e-10
long double	1001.2L
char	'A' 'B' ' ' 'a' '\n' '\t' '\b'
bool	true false
wchar_t	L'A' L'B'

2.3 Các kiểu dữ liệu dẫn xuất trực tiếp

- Kiểu liệt kê
- Kiểu hằng
- Kiểu con trỏ
- Kiểu mảng
- Kiểu tham chiếu (C++)

2.3.1 Kiểu liệt kê (enum)

- Mục đích sử dụng:
  - Tương tự một kiểu nguyên, cần hạn chế phạm vi sử dụng
  - Sử dụng thuận tiện bằng tên => **hằng số nguyên**
- Ví dụ

```
enum Color {Red, Green, Blue};
enum WeekDay {
    Mon = 2,
    Tue, Wed, Thu, Fri, Sat,
    Sun = 1  };
enum {
    DI_MOTOR1_STARTED = 0x01,
    DI_MOTOR1_RUNNING = 0x02,
    DI_MOTOR2_STARTED = 0x04,
    DI_MOTOR2_RUNNING = 0x08,
    DI_PUMP1_STARTED  = 0x10,
    DI_PUMP1_RUNNING  = 0x20,
    DI_OVERLOADED     = 0x40,
    DI_VALVE1_OPENED  = 0x80
};
```

Sử dụng kiểu liệt kê

```
/* C version */
void main() {
   enum Color c = Red;          /* c = 0 */
   enum WeekDay d = Tue;        /* d = 3 */
   int i=c, j=d;                /* j=0, i=3 */
   enum Color c2 = i+1;         /* c2 = 1 */
   int di1 = 0x01;              /* OK, but... */
   int di2 = DI_MOTOR1_STARTED;/* this is better */
   ++c;                         /* c = 1 */
}
```

C:
Như một kiểu số nguyên 8 bit

```
// C++ version */
void main() {
   enum Color c = Red;          // c = Red
   WeekDay d = Tue;             // OK, d = Tue
   int i=c, j=d;                // j=0, i=3
   Color c2 = i+1;              // Error!
   Color c3 = Color(i+1);       // OK, c3 = Green
   int di1 = 0x01;              // OK, but...
   int di2 = DI_MOTOR1_STARTED;// this is better
   ++c;                         // Error!
}
```

C++
Không còn như một kiểu số nguyên!

2.3.2 Kiểu hằng (const)

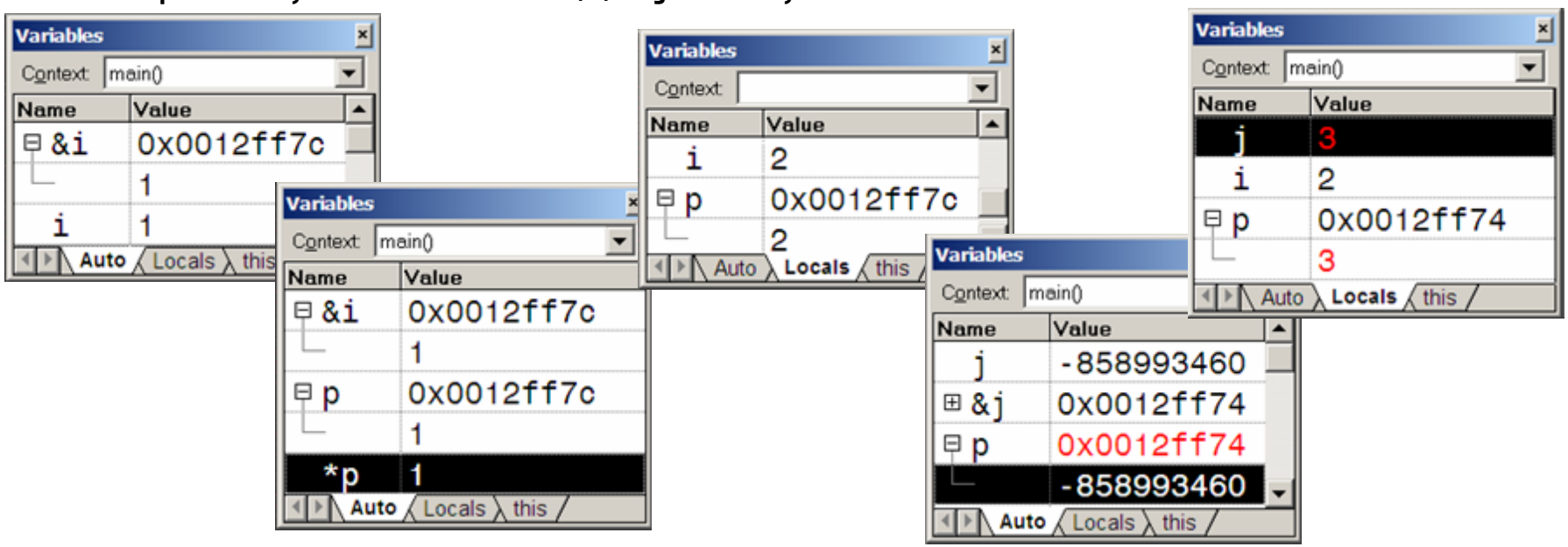
- Tác dụng làm cho một biến trở thành không thay đổi được => khai báo hằng số

```
void main() {
   const double pi = 3.1412;  // initializing is OK!
   const int ci = 1;          // initializing is OK!
   ci = 2;          // error!
   ci = 1;          // error, too!
   int i = ci;      // const int is a subset of int
   const Color cc = Red;
   cc = Green;      // error
   const double d; // potential error
}
```

2.3.3 Kiểu con trỏ

- Con trỏ thực chất là một biến mang địa chỉ của một ô nhớ (một biến khác hoặc một hàm)

```
int  i = 1;
int* p = &i;        // p has the address of i
*p = 2;             // i = 2
int  j;
p = &j;             // now p has the address of j
*p = 3;             // j = 3, i remains 2
```



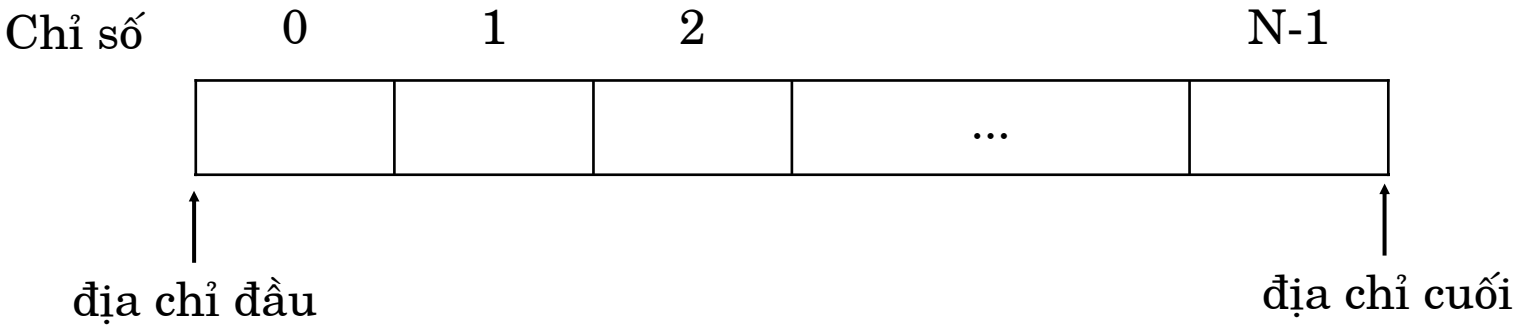
Ví dụ sử dụng kiểu con trỏ

```
void main() {
   int  i = 0;
   int* p = &i;   // p refers to the addesss of i
   int j = *p;    // j = 0
   *p = 2;        // now i = 2
   p = &j;        // now p contains the addess of j
   *p = 3;        // now j = 3, i remains 2
   double d = i; // OK, int is compatible to double
   p = &d;        // error, int* isn't compatible to double*
   p = (*int)&d; // no compile error, but dangerous,
                  // meaningles type conversion!
   double* pd=0; // p contains the address 0
   *pd = 0;       // no compile error, but fatal error
   pd = &d;       // OK
   double* pd2;   // p refers to an uncertain address
   *pd2 = 0;      // fatal error
   pd2 = &d;      // OK, pd and pd2 refer to the same addr.
}
```

Tóm tắt sơ bộ về con trỏ

- Con trỏ là một biến chứa địa chỉ byte đầu của một biến dữ liệu, được sử dụng để truy cập gián tiếp dữ liệu đó
- Sau khi khai báo mà không khởi tạo, mặc định con trỏ mang một địa chỉ bất định
- Địa chỉ con trỏ mang có thể thay đổi được => con trỏ có thể mỗi lúc đại diện cho một biến dữ liệu khác
- Toán tử lấy địa chỉ của một biến (&) trả về con trỏ vào kiểu của biến => thường gán cho biến con trỏ
- Toán tử truy nhập nội dung (*) áp dụng cho con trỏ trả về biến mà con trỏ mang địa chỉ => có thể đọc hoặc thay đổi giá trị của biến đó
- Không bao giờ sử dụng toán tử truy nhập nội dung, nếu con trỏ chưa mang một địa chỉ ô nhớ mà chương trình có quyền kiểm soát

2.3.4 Kiểu mảng



địa chỉ đầu - địa chỉ cuối = N * kích cỡ một phần tử

- Cấu trúc dữ liệu với:
  - Số lượng các phần tử cố định
  - Các phần tử có cùng kiểu
  - Các phần tử được sắp xếp kế tiếp trong bộ nhớ
  - Có thể truy nhập từng phần tử một cách tự do theo chỉ số hoặc theo địa chỉ

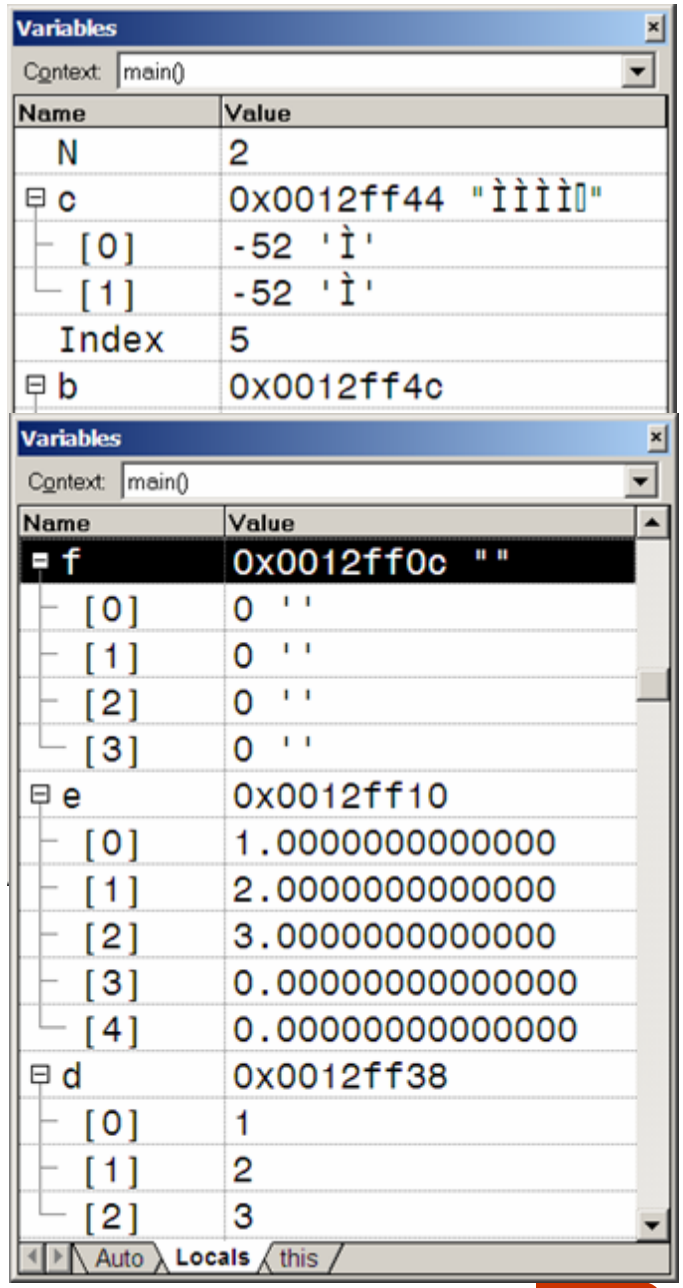
Khai báo mảng

- Số phần tử của mảng phải là hằng số nguyên (trong C phải là một trực kiện, trong C++ có thể là kiểu **const** ...)
- Khai báo không khởi tạo:

```
int         a[3];
enum        {index = 5};
double      b[index];
const int N = 2;
char        c[N]; // C++ only
```

- Khai báo với số phần tử và khởi tạo giá trị các phần tử

```
int         d[3]= {1, 2, 3};
double      e[5]= {1, 2, 3};
char        f[4]= {0};
```



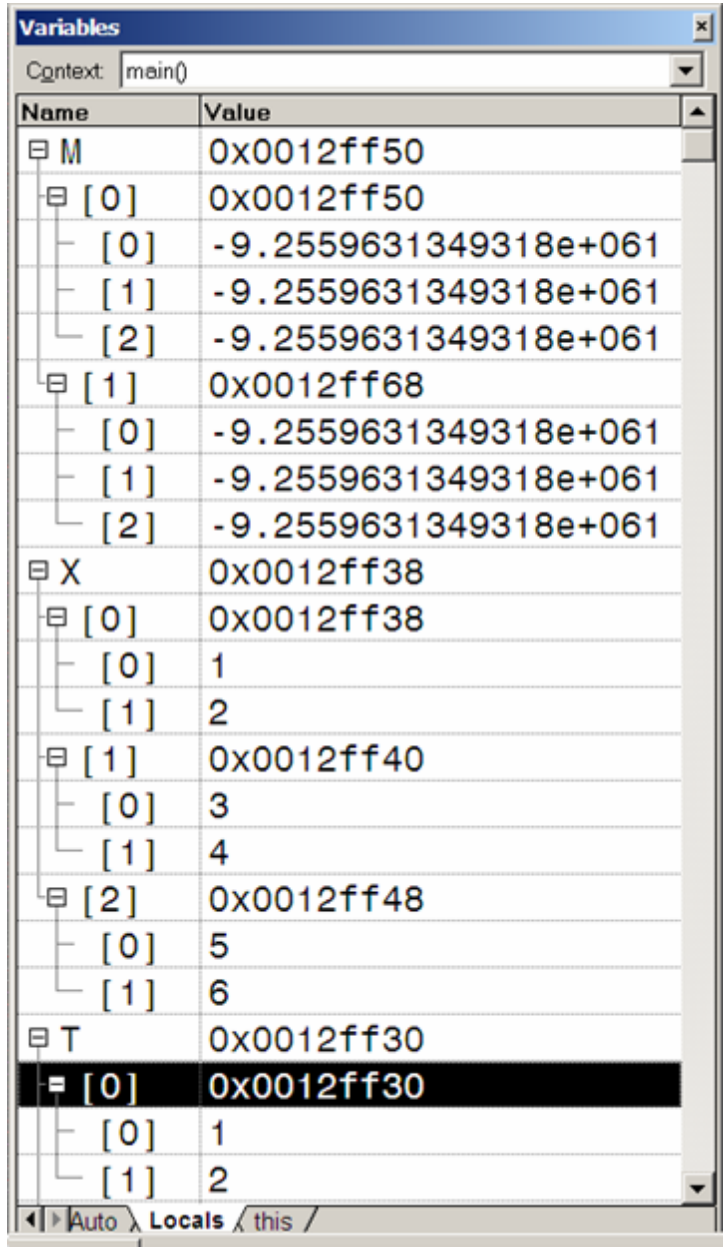
Khai báo mảng (tiếp)

- Khai báo và khởi tạo giá trị các phần tử, số phần tử được tự động xác định

```
int    a[]= {1, 2, 3, 4, 5};
double b[]= {1, 2, 3};
double c[]= {0};
char   s[]= {'a'};
```

- Khai báo mảng nhiều chiều

```
double M[2][3];
int X[2][]={{1,2},{3,4},{5,6}};
short T[2][2]={1,2,3,4,5,6};
```



Ví dụ sử dụng kiểu mảng

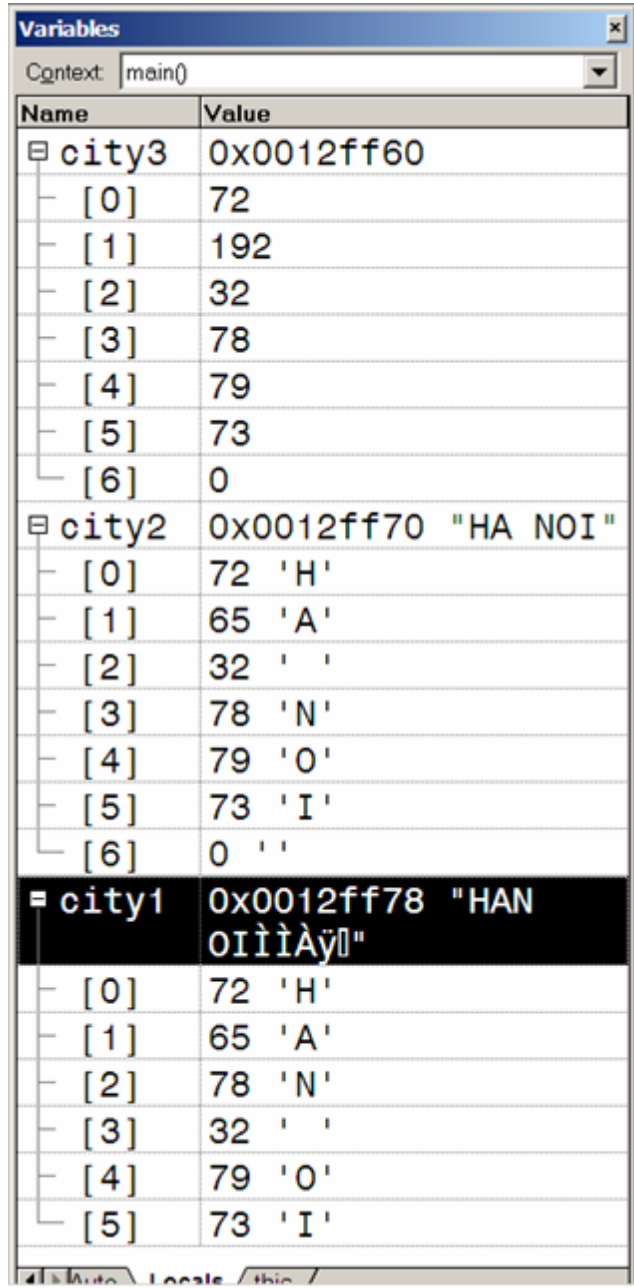
```
void main() {
   int  a[5];           // a has 5 elements with uncertain values
   int  b[5]= {1,3,5,7,9};    // 5 elements with initial values
   double c[];                // error, unspecified size
   double x = 1.0, y = 2.0;
   double d[]= {x,y,3.0};     // 3 elements with initial values
   short n = 10;
   double v[n];         // error, array size must be a constant!
   const int m=10;      // C++ OK
   double v2[m];        // C++ OK
   a[0] = 1;
   int i= 1;
   a[i] = 2;
   a[5] = 6;            // no compile error, but fatal error
   int k = a[5];        // no compile error, but fatal error
   a = {1,2,3,4,5};     // error
   a = b;               // error, cannot assign array
   int M[2][3];
   M[0][1] = 0;
   M[0][2] = 1;
}
```

Mảng đặc biệt: Chuỗi ký tự

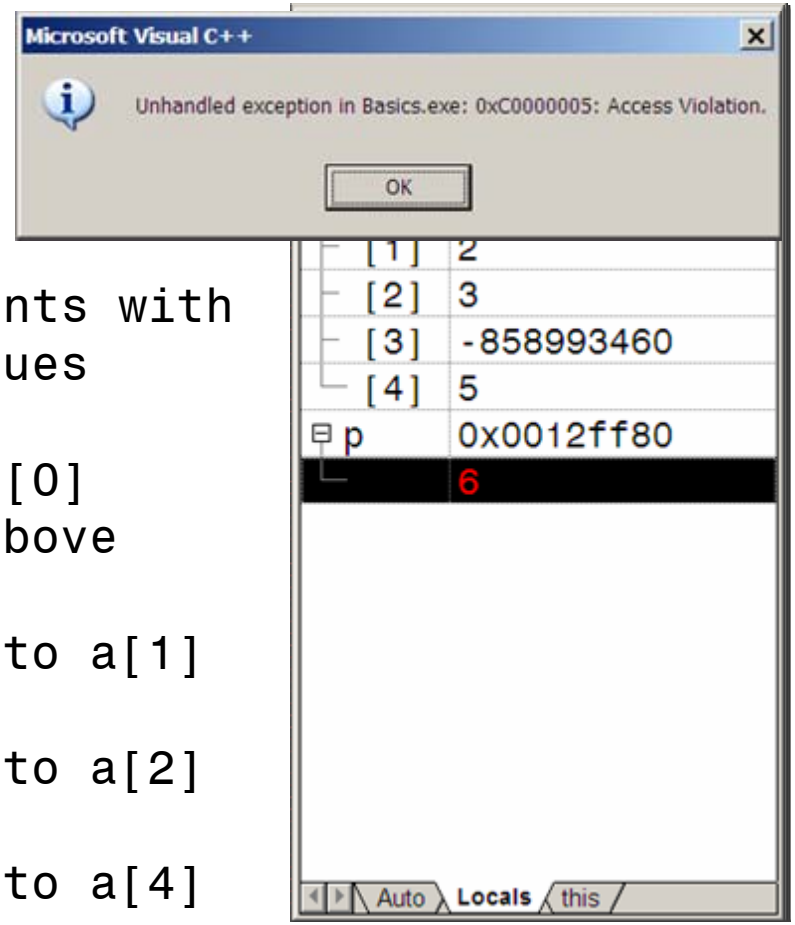
- Trong C/C++, chuỗi ký tự không phải là kiểu cơ bản, mà thực chất là một mảng
- Phân biệt chuỗi ký tự thường và **chuỗi ký tự kết 0**

```
char city1[]= {'H','A','N',' ','O','I'};
char     city2[]=  "HA NOI";
wchar_t  city3[]= L"HÀ NOI";
city2[] = "HANOI"; // error
```

- Đa số các hàm trong thư viện C làm việc với chuỗi ký tự kết 0
- Với C++, chuỗi ký tự được định nghĩa bằng lớp **string** trong thư viện chuẩn, không sử dụng byte kết 0



Mảng và con trỏ




```
void main() {
   int  a[5];    // a has 5 elements with
                 // uncertain values
   int* p;
   p = a;        // p refers to a[0]
   p = &a[0];    // the same as above
   *p = 1;       // a[0]=1
   ++p;          // now p points to a[1]
   *p = 2;       // a[1]=2
   p++;          // now p points to a[2]
   *p = 3;       // a[2]=3
   p += 2;       // now p points to a[4]
   *p = 5;       // a[4] = 5
   ++p;          // OK, no problem until we dereference it
   *p = 6;       // Now is a BIG BIG problem!
   a = p;        // error, a is like a constant pointer
}
```

Mảng và con trỏ (tiếp)

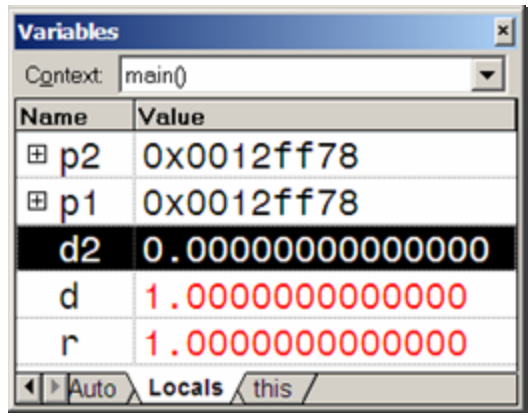
```
void main() {
   int  a[5];    // a has 5 elements with
                 // uncertain values
   int* p = a;   // p points to a[0]
   p[0] = 1;     // a[0]=1
   p[1] = 2;     // a[1]=2
   p+= 2;        // now p points to a[2]
   p[0] = 3;     // a[2]=3
   p[1] = 4;     // a[3]=4
   p[3] = 6;     // a[5]=6, Now is a BIG BIG problem!
}
```

Tóm lược về mảng

- Mảng là một tập hợp các dữ liệu cùng kiểu, sắp xếp liền kề trong bộ nhớ => các phần tử của mảng
- Có thể truy cập các phần tử mảng với biến mảng kèm theo chỉ số hoặc với biến con trỏ (theo địa chỉ của từng phần tử)
- Số phần tử của mảng là cố định (khi khai báo phải là hằng số), không bao giờ thay đổi được
- Biến mảng (tĩnh) thực chất là một con trỏ hằng, mang địa chỉ của phần tử đầu tiên
- Có thể đặt giá trị đầu cho các phần tử của mảng qua danh sách khởi tạo, không bao giờ gán được mảng cho nhau. Nếu cần sao chép hai mảng thì phải sử dụng hàm
- Không bao giờ được phép truy nhập với chỉ số nằm ngoài phạm vi, nếu N là số phần tử thì phạm vi cho phép là từ 0..N-1
- Con trỏ không bao giờ là một mảng, nó chỉ có thể mang địa chỉ của một mảng và sử dụng để quản lý mảng (dù là động hay tĩnh)

2.3.5 Kiểu tham chiếu (C++)

- Một biến tham chiếu là một biến đại diện trực tiếp cho một biến khác (thay cho con trỏ)
- Ý nghĩa sử dụng chủ yếu về sau trong truyền tham số cho hàm




```
void main() {
    double  d = 2.0;
    double& r = d;   // r represents d
    double *p1 = &d, *p2 = &r;
    r = 1.0;         // OK, d = 1.0
    double& r2;     // error, r has to be assigned to a var.
    double& r3 = 0; // error, too
    double d2 = 0;
    r = d2;          // r = 0, d=0
    r = 1.0;         // r = d = 1, d2 =0
}
```

2.3.6 Typedef

- Từ khóa **typedef** tạo ra một tên mới cho một kiểu có sẵn, không định nghĩa một kiểu mới
- Ý nghĩa: đưa tên mới dễ nhớ, phù hợp với ứng dụng cụ thể, dễ thay đổi về sau

```
typedef float REAL;
typedef int AnalogValue;
typedef int Vector[10];
typedef AnalogValue AnalogModule[8];
typedef int* IPointer;
AnalogValue av1 = 4500;
Vector x = {1,2,3,4,5,6,7,8,9,10};
AnalogModule am1 = {0};
IPointer p = &av1;
```

2.4 Định nghĩa kiểu dữ liệu mới

- Cấu trúc (**struct**): Tập hợp những dữ liệu hỗn hợp, truy nhập theo tên (biến thành viên). Thông dụng nhất trong C, ý nghĩa được mở rộng trong C++
- Hợp nhất (**union**): Một tên kiểu chung cho nhiều dữ liệu khác nhau (chiếm cùng chỗ trong bộ nhớ). Ít thông dụng trong cả C và C++
- Lớp (**class**): Chỉ có trong C++, mở rộng **struct** cũ thêm những hàm thành viên.

2.4.1 Cấu trúc (struct)

- Định nghĩa cấu trúc (bên trong hoặc ngoài các hàm)

```
struct Time
{
   int hour; // gio
   int minute;  // phut
   int second;  // giay
};

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

struct Student {
   char name[32];
   struct Date birthday;
   int  id_number;
};
```

Tên kiểu mới (không trùng lặp)

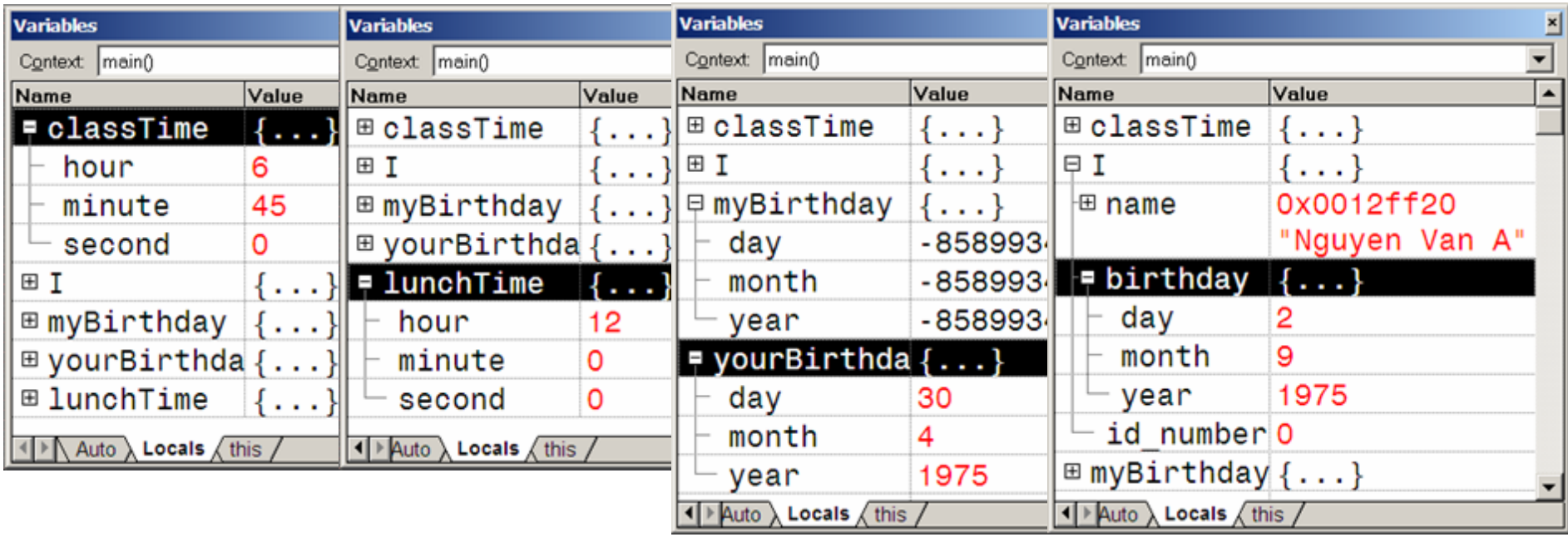
Các biến thành viên, khai báo độc lập hoặc chung kiểu

Các biến thành viên có thể cùng kiểu hoặc khác kiểu

C++

Khai báo biến cấu trúc

```
void main() {
   Time classTime  = {6,45,0};
   Time lunchTime  = {12};
   Date myBirthday, yourBirthday = {30,4,1975};
   Student I = {"Nguyen Van A", {2,9,1975}};
   //...
```



Sử dụng biến cấu trúc

```
/...
void main() {
   Time classTime  = {6,45,0};
   Time lunchTime  = {12};
   Date myBirthday, yourBirthday = {30,4,1975};
   Student I = {"Nguyen Van A", {2,9,1975}};
   lunchTime.minute = 15;
   lunchTime.hour = classTime.hour + 6;
   Student U = I;    // in C++ also possible: Student U(I);
   U.name[11] = 'B'; // "Nguyen Van B"
   U.id_number++;              // 1
   U.birthday.day = 30;        // 30-9-1975
   U.birthday.month = 4;       // 30-4-1975
   U.birthday = yourBirthday; // structs can be assigned
}
```

Phản ví dụ: khai báo và sử dụng cấu trúc

```
struct Time {
   int hour = 0;      // error, initialization not allowed
   int minute,        // error, use semicolon (;) instead
   int second         // error, missing semicolon (;)
}      // error, missing semicolon (;)
//...
void main() {
   Date d;
   d = {11,9,2001}; // error, {...} is an initialization
                    // list,not a structure
   Date.hour = 0;   // error, Date is a type, not a var.
   struct Date2 { int day, month, year; };
   Date2 d2 = d; // error, Date is not compatible to Date2
}
```

Mảng, con trỏ và cấu trúc

- Kết hợp mảng, con trỏ và cấu trúc cho phép xây dựng và sử dụng các cấu trúc dữ liệu phức tạp một cách rất linh hoạt

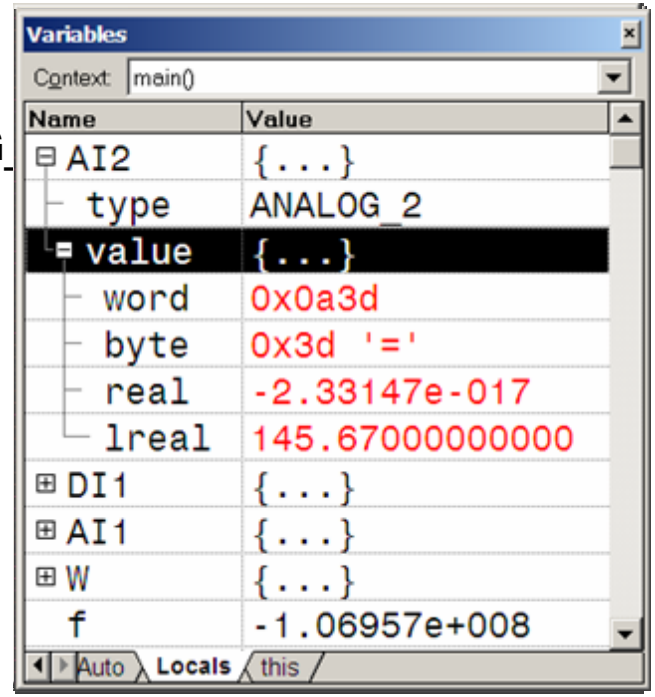
```
void main() {
//...
   Date victoryDays[]= {{19,8,1945},{7,5,1954},{30,4,1975}};
   Date saigonVictory= victoryDays[2];
   Date *p=&saigonVictory;
   (*p).year += 30;      // good
   p->year -=30;         // better
   Student studentList[45];
   for (int i=0; i < 45; ++i) {
      studentList[i].id_number= i;
      studentList[i].birthday=yourBirthday;
   }
   Student* pList = studentList;
   while (pList < studentList+45) {
      pList->id_number += 4800;
      ++pList;
   }
}
```

Tóm lược về cấu trúc (struct)

- Cấu trúc (struct) được sử dụng để nhóm các dữ liệu liên quan mô tả một đối tượng, các dữ liệu có thể cùng hoặc khác kiểu
- Định nghĩa kiểu cấu trúc bằng cách khai báo **tên các biến thành viên**. Định nghĩa kiểu cấu trúc chưa phải là định nghĩa các biến cụ thể, vì thế không được đặt giá trị đầu cho các biến
- Kích cỡ của cấu trúc >= tổng kích cỡ các thành viên
- Truy cập một biến cấu trúc thông qua tên biến, toán tử (.) và tên biến thành viên
- Các kiểu cấu trúc có thể lồng vào nhau, trong cấu trúc có thể sử dụng mảng, một mảng có thể có các phần tử là cấu trúc, v.v...
- Các biến có cùng kiểu cấu trúc có thể gán cho nhau, có thể sử dụng để khởi tạo cho nhau (khác hẳn với mảng)
- Có thể sử dụng con trỏ để truy nhập dữ liệu cấu trúc thông qua toán tử (*.) và toán tử (->)
- Hai kiểu cấu trúc có khai báo giống nhau hoàn toàn vẫn là hai kiểu cấu trúc khác nhau

2.4.2 Hợp nhất

```
enum SignalType {BINARY_8, BINARY_16, ANALOG_
union SignalValue {
   unsigned short word;
   unsigned char  byte;
   float    real;
   double   lreal;
};
struct Signal {
   SignalType  type;
   SignalValue value;
};
void main() {
   SignalValue B,W;
   B.byte = 0x01;
   W.word = 0x0101;
   unsigned char b = W.byte; // OK, the lower byte
   float f = W.real;         // meaningless
   Signal DI1 = {BINARY_8, 0x11};
   Signal AI1 = {ANALOG_1,{0}};
   Signal AI2;
   AI2.type = ANALOG_2;
   AI2.value.lreal = 145.67;
}
```



Tóm lược về hợp nhất

- Hợp nhất (union) là một tập hợp (không có cấu trúc chặt chẽ) chứa các biến sử dụng chung ô nhớ, ở mỗi ngữ cảnh chỉ sử dụng một biến riêng biệt
- Union thường được sử dụng khi dữ liệu đầu vào có thể có kiểu khác nhau
- Các thành viên của một union không liên quan đến nhau, không cùng nhau tạo thành một thực thể thống nhất
- Kích cỡ của union bằng kích cỡ của biến lớn nhất
- Khai báo kiểu union tương tự như khai báo struct, nhưng ý nghĩa khác hẳn
- Truy nhập biến thành viên cũng tương tự như struct, có thể qua biến trực tiếp hoặc qua biến con trỏ.
- Union có thể chứa struct, struct có thể chứa union, union có thể chứa mảng, các phần tử của mảng có thể là union.

2.5 Điều khiển CT: phân nhánh

- Các kiểu phân nhánh
  - **`if .. else`**: Phân nhánh lựa chọn một hoặc hai trường hợp
  - **`switch .. case`**: Phân nhánh lựa chọn nhiều trường hợp
  - **`break`**: Lệnh nhảy kết thúc (sớm) một phạm vi
  - **`return`**: Lệnh nhảy và kết thúc (sớm) một hàm
  - **`goto`**: Lệnh nhảy tới một nhãn (**không nên dùng!**)

2.5.1 Cấu trúc if .. else

- Lựa chọn một trường hợp: sử dụng **if**

```
if (npoints >= 60)
  cout << "Passed";
if (npoints >= 80 && npoints <= 90) {
  grade = 'A';
  cout << grade;
}
```

- Phân nhánh hai trường hợp: sử dụng **if .. else**

```
if (npoints >= 90)
  cout << 'A';
else if (npoints >= 80)
  cout << 'B';
  else if (npoints >= 70)
       cout << 'C';
       else if (npoints >= 60)
            cout << 'D';
            else
                 cout << 'F';
```

Ví dụ: Hàm max()

```
int max1(int a, int b) {
   int c;
   if (a > b)
        c = a;
   else
        c = b;
   return c;
}
int max2(int a, int b) {
   int c = a;
   if (a < b)
        c = b;
   return c;
}
int max3(int a, int b) {
   if (a < b)
        a = b;
   return a;
}
```

```
int max4(int a, int b) {
   if (a > b)
        return a;
   else
        return b;
}
int max5(int a, int b) {
   if (a > b)
        return a;
   return b;
}
int max6(int a, int b) {
   return (a > b)? a: b;
}
```

2.5.2 Cấu trúc switch .. case

```
Signal input;
int i = 0;
while (i++ < 8) {
   input = readInput(i); // read from input module i
   switch (input.type) {
   case BINARY_8:
        cout << input.value.byte; break;
   case BINARY_16:
        cout << input.value.word; break;
   case ANALOG_1:
        cout << input.value.real; break;
   case ANALOG_2:
        cout << input.value.lreal; break;
   default:
        cout << "Unknown signal type";
   }
}
```

2.6 Điều khiển CT: vòng lặp

- Các kiểu vòng lặp tront C/C++
  - **while** (*condition*) { }
  - **do** { } **while** (*condition*)
  - **for** (*init*;*condition*;*post_action*) { }
- Vòng lặp có thể thực hiện với **if..else + goto**, song không bao giờ nên như vậy
- Ứng dụng vòng lặp chủ yếu trong làm việc với mảng và các cấu trúc dữ liệu tổng quát khác => truy nhập qua biến mảng + chỉ số, qua con trỏ hoặc qua **iterator** (sẽ đề cập sau này)

2.6.1 Cấu trúc while..

```
#include <iostream.h>
void main() {
   char input[32];
   cout << "\nEnter your full name:";
   cin.getline(input,31);
   short nLetters=0, nSpaces=0;
   short i=0;
   while (input[i] != 0) {
     if (input[i] == ' ')
          ++nSpaces;
     else
          ++nLetters;
     ++i;
   }
   cout << "\nYour name has " << nLetters << " letters";
   cout << "\nYou have " << nSpaces - 1 << " middle name";
   cin >> i;
}
```

Cấu trúc while: Biểu thức điều kiện

```
#include <iostream.h>
void main() {
   char input[32], family_name[16]={0};
   cout << "\nEnter your full name:";
   cin.getline(input,31);
   short i=0;
   while (input[i] != 0) {
      if (input[i] == ' ')    break;
      family_name[i]= input[i];
         ++i;
   }
   cout << "\nYour family name is " << family_name;
   cin >> i;
}
```

2.6.2 Cấu trúc do while...

```
#include <iostream.h>
void main() {
   char input[32], family_name[16]={0};
   short i;
   do {
     cout << "\nEnter your full name:";
     cin.getline(input,31);
     i = 0;
     while (input[i] != 0 && input[i] != ' ') {
          family_name[i]= input[i];
          ++i;
     }
     cout << "\nYour family name is " << family_name;
     cout << "\nDo you want to continue? (Y/N):“;
     cin >> i;
   } while (i == ‘Y’ || i == ‘N’)
}
```

2.6.3 Cấu trúc for ..

```
short i =0;
while (input[i]!= 0)
{
   if (input[i]==' ')
     ++nSpaces;
   else
     ++nLetters;
   ++i;
}
```

```
for (short i=0;input[i]!=0; ++i)
{
   if (input[i] == ' ')
     ++nSpaces;
   else
     ++nLetters;
}
```

```
short i=0;
for (;input[i]!= 0;)
{
   if (input[i]==' ')
     ++nSpaces;
   else
     ++nLetters;
   ++i;
}
```

```
short i=0;
for (;input[i]!=0; ++i)
{
   if (input[i] == ' ')
     ++nSpaces;
   else
     ++nLetters;
}
```

Tóm lược các cấu trúc vòng lặp

- Các cấu trúc vòng lặp **`while`** và **`for`** tương tự như nhau, thực ra ta chỉ cần một trong hai
- Cấu trúc **`do..while`** tuy có ý nghĩa khác một chút, song cũng có thể chuyển về cấu trúc **`while`** hoặc **`for`**
- Các cấu trúc có thể lồng vào nhau tương đối tự do, tuy nhiên tránh lồng quá nhiều để còn dễ bao quát, khi cần có thể phân hoạch lại thành hàm
- Điều khiển vòng lặp có thể nằm trực tiếp trên điều kiện, hoặc có thể kết hợp bên trong vòng lặp với các lệnh **`if..else`** và **`break, return`**
- Thận trọng trong kiểm tra điều kiện vòng lặp (chỉ số mảng, con trỏ, ...)

Luyện tập ở nhà theo sườn bài giảng

- Tập tạo dự án mới với Visual C++
- Tập viết một chương trình bằng C (đặt đuôi *.c):
  - tập khai báo các loại biến, sử dụng các kiểu dữ liệu cơ bản
  - tập sử dụng các phép toán đã học
  - sử dụng toán tử **sizeof** để tìm kích cỡ các kiểu dữ liệu, in kết quả ra màn hình
  - biên dịch, chạy thử và tìm lỗi
  - tập sử dụng công cụ debugger
  - đổi đuôi file thành *.cpp và thử lại
- Tập viết một chương trình bằng C/C++ khác để tìm hiểu:
  - Cách khai báo và sử dụng kiểu hằng, kiểu liệt kê, kiểu con trỏ, kiểu mảng, kiểu tham chiếu (C++), kiểu cấu trúc
  - bản chất của con trỏ và quan hệ với kiểu mảng

Bài tập về nhà cho chương 2

1. Viết một chương trình bằng C, thực hiện lần lượt các chức năng sau đây:
   - yêu cầu người sử dụng nhập một số nguyên lớn hơn 0
   - phân tích số nguyên đó thành hàng đơn vị, hàng chục, hàm trăm, v.v... và in kết quả lần lượt ra màn hình.
   - hỏi người sử dụng có yêu cầu tiếp tục hay không, nếu có yêu cầu thì lặp lại
2. Chuyển chương trình thành C++ và đơn giản hóa các câu lệnh vào-ra bằng cách sử dụng thư viện `<iostream.h>`
3. Dựa vào kiểu Date trong bài giảng, viết một chương trình cho phép người sử dụng nhập số liệu cho một ngày, và sau đó:
   a) Kiểm tra các số liệu ngày, tháng và năm có hợp lệ hay không
   b) Kiểm tra xem ngày đó có phải là một ngày lễ trong năm hay không
   c) Xác định ngày tiếp theo là ngày nào
   d) In các kết quả thông báo ra màn hình

Bài tập lớn 1 (tuần 1-6: Lập trình cấu trúc)

1. Xây dựng một chương trình có chức năng tạo tín hiệu theo yêu cầu người sử dụng về dạng tín hiệu (bậc thang, tín hiệu dốc, xung vuông, hình sin hoặc ồn trắng), tham số của tín hiệu (tùy theo dạng tín hiệu chọn như biên độ, tần số, độ dốc, độ rộng xung,...). Yêu cầu người sử dụng nhập khoảng thời gian cần tạo giá trị tín hiệu cùng thời gian trích mẫu, sau đó ghi các giá trị gián đoạn của tín hiệu ra một file với tên do người sử dụng nhập.

   Gợi ý: sử dụng thư viện <fstream.h> cho việc thao tác với file.
2. Xây dựng một chương trình để tính tích phân của tín hiệu (hay tính diện tích dưới đường cong) bằng phương pháp xấp xỉ hình thang với các giá trị gián đoạn của tín hiệu đưa vào từ file tạo ra theo chương trình 1.
3. Suy nghĩ phân hoạch chương trình 1 và 2 thành các hàm đưa vào thư viện. Viết lại các chương trình đó theo thiết kế mới.

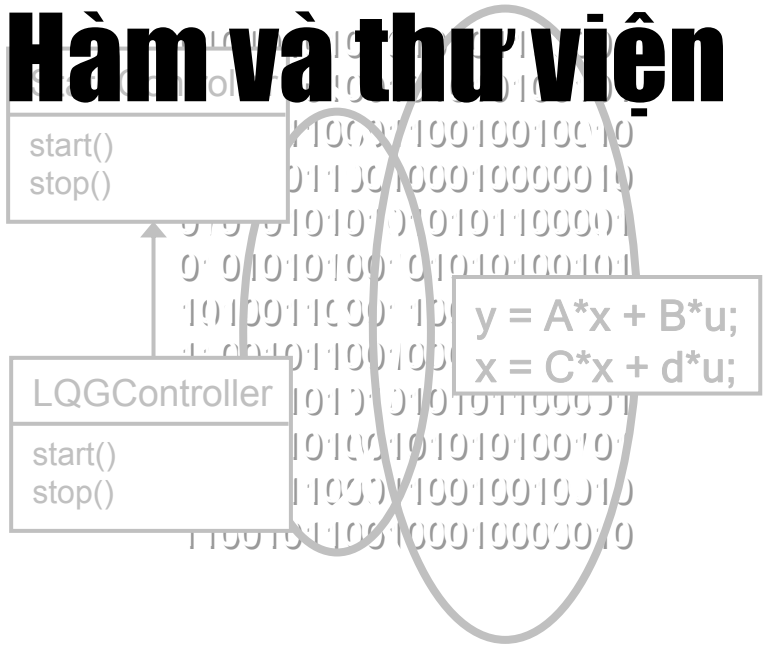
Chỉ dẫn về thực hiện bài tập lớn

- Bài tập lớn có thể thực hiện riêng hoặc theo nhóm tự chọn (tối đa 3 người/nhóm)
- Bài tập lớn 1 nộp vào cuối tuần 7, bao gồm:
  - Mô tả theo mẫu trên ít nhất 1 trang giấy về các tư tưởng phân tích, thiết kế và thực thi.
  - Toàn bộ thư mục dự án (file dự án, mã nguồn và chương trình chạy) cần nén lại dưới dạng *.zip và gửi về địa chỉ email của giáo viên: hmson-ac@mail.hut.edu.vn. Qui định tên file zip: bắt đầu bằng "P1_", tiếp theo là tên đầy đủ của người đại diện nhóm, ví dụ "P1_NguyenVanA.zip". Lưu ý trước khi nén cần xóa tất cả các file phụ trong thư mục "Debug", chỉ trừ file *.exe.
- Hoàn thành bài tập lớn không những là điều kiện dự thi học kỳ, mà điểm bài tập lớn còn được tính vào điểm cuối học kỳ theo một hệ số thích hợp

Kỹ thuật lập trình

Phần II: Lập trình có cấu trúc

Chương 3: Hàm và thư viện



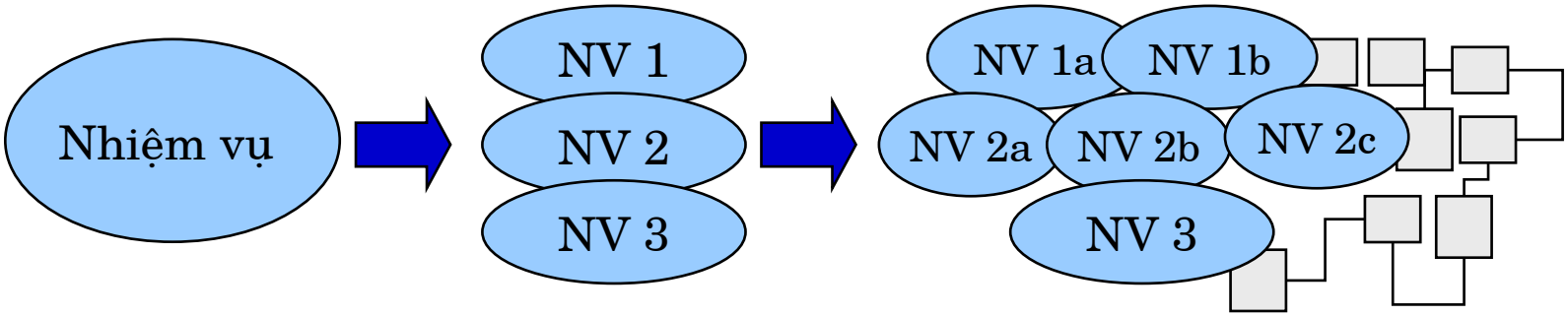
9/14/2005

Nội dung chương 3

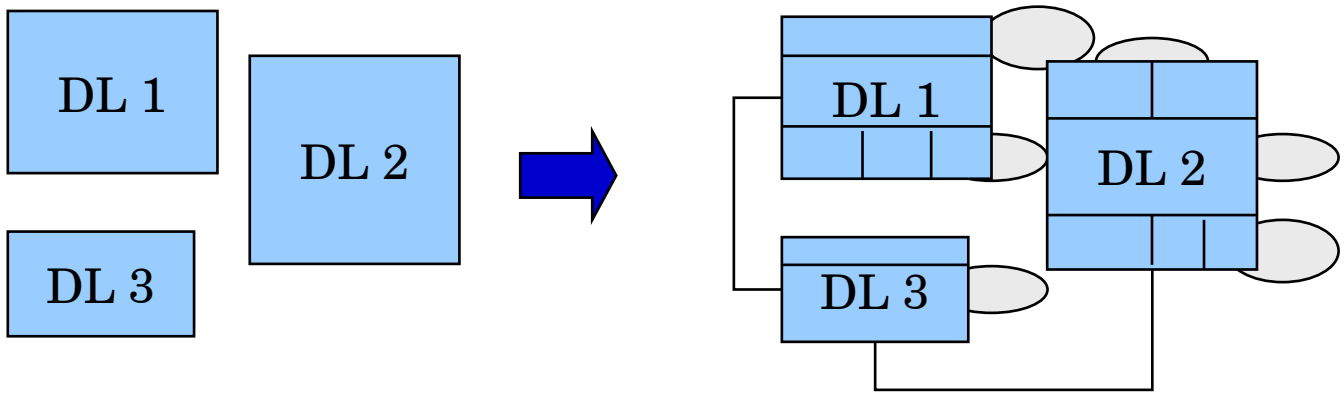
3.1 Hàm và lập trình hướng hàm

Lập trình có cấu trúc có thể dựa trên một trong hai phương pháp:

- Lập trình hướng hàm (*function-oriented*), còn gọi là hướng nhiệm vụ (*task-oriented*), hướng thủ tục (*procedure-oriented*)



- Lập trình hướng dữ liệu (*data-oriented*)



Hàm là gì?

- Tiếng Anh: function -> hàm, chức năng
- Một đơn vị tổ chức chương trình, một đoạn mã chương trình có cấu trúc để thực hiện một **chức năng** nhất định, có **giá trị sử dụng lại**
- Các hàm có quan hệ với nhau thông qua lời gọi, các biến tham số (đầu vào, đầu ra) và giá trị trả về
- Cách thực hiện cụ thể một hàm phụ thuộc nhiều vào dữ kiện (tham số, đối số của hàm):
  - Thông thường, kết quả thực hiện hàm mỗi lần đều giống nhau nếu các tham số đầu vào như nhau
  - Một hàm không có tham số thì giá trị sử dụng lại rất thấp
- Trong C/C++: Không phân biệt giữa thủ tục và hàm, cả đoạn mã chương trình chính cũng là hàm

Ví dụ phân tích

- Yêu cầu bài toán: Tính tổng một dãy số nguyên (liên tục) trong phạm vi do người sử dụng nhập. In kết quả ra màn hình.
- Các nhiệm vụ:
  - Nhập số nguyên thứ nhất:
    - Yêu cầu người sử dụng nhập
    - Nhập số vào một biến
  - Nhập số nguyên thứ hai
    - Yêu cầu người sử dụng nhập
    - Nhập số vào một biến
  - Tính tổng với vòng lặp
  - Hiển thị kết quả ra màn hình

Phương án 4 trong 1

```
#include <iostream.h>
void main() {
   int a, b;
   char c;
   do {
      cout << "Enter the first integer number: ";
      cin >> a;
      cout << "Enter the second integer number: ";
      cin >> b;
      int Total = 0;
      for (int i = a; i <= b; ++i)
        Total += i;
      cout << "The sum from " << a << " to " << b
           << " is " << Total << endl;
      cout << "Do you want to continue? (Y/N):";
      cin  >> c;
   } while (c == 'y' || c == 'Y');
}
```

Phương án phân hoạch hàm (1)

```
#include <iostream.h>

int  ReadInt();
int  SumInt(int,int);
void WriteResult(int a, int b, int kq);

void main() {
   char c;
   do {
      int a = ReadInt();
      int b = ReadInt();
      int T = SumInt(a,b);
      WriteResult(a,b,T);
      cout << "Do you want to continue? (Y/N):";
      cin  >> c;
   } while (c == 'y' || c == 'Y');
}
```

Phương án phân hoạch hàm (1)

```
int ReadInt() {
   cout << "Enter an integer number: ";
   int N;
   cin >> N;
   return N;
}

int SumInt(int a, int b) {
   int Total = 0;
   for (int i = a; i <= b; ++i)
      Total += i;
   return Total;
}

void WriteResult(int a, int b, int kq) {
   cout << "The sum from " << a << " to " << b
        << " is " << kq << endl;
}
```

Không có tham số,
Giá trị sử dụng lại?

OK,
Không thể tốt hơn!

Quá nhiều tham số,
Hiệu năng?

Phương án phân hoạch hàm (1)

- Chương trình dễ đọc hơn => dễ phát hiện lỗi
- Chương trình dễ mở rộng hơn
- Hàm SumInt có thể sử dụng lại tốt
- Mã nguồn dài hơn
- Mã chạy lớn hơn
- Chạy chậm hơn

➲ Không phải cứ phân hoạch thành nhiều hàm là tốt, mà vấn đề nằm ở cách phân hoạch và thiết kế hàm làm sao cho **tối ưu**!

Phương án phân hoạch hàm (2)

```
#include <iostream.h>

int  ReadInt(const char*);
int  SumInt(int,int);

void main() {
   char c;
   do {
      int a = ReadInt("Enter the first integer number :");
      int b = ReadInt("Enter the second integer number:");
      cout << "The sum from " << a << " to " << b
           << " is " << SumInt(a,b) << endl;
      cout << "Do you want to continue? (Y/N):";
      cin  >> c;
   } while (c == 'y' || c == 'Y');
}
```

Phương án phân hoạch hàm (2)

```
int ReadInt(const char* userPrompt) {
   cout << userPrompt;
   int N;
   cin >> N;
   return N;
}
int SumInt(int a, int b) {
   int Total = 0;
   for (int i = a; i <= b; ++i)
      Total += i;
   return Total;
}
```

OK,
Đã tốt hơn!

3.2 Khai báo và định nghĩa hàm

- Định nghĩa hàm: tạo mã thực thi hàm

Kiểu trả về Tên hàm Tham biến (hình thức)

```
int SumInt(int a, int b) {
    int Total = 0;
    for (int i = a; i <= b; ++i)
        Total += i;
    return Total;
}
```

- Khai báo hàm thuần túy: không tạo mã hàm

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

Kiểu trả về Tên hàm Kiểu tham biến

- Tại sao và khi nào cần khai báo hàm?

Khai báo hàm và lời gọi hàm

- Ý nghĩa của khai báo hàm:
  - Khi cần sử dụng hàm (gọi hàm)
  - Trình biên dịch cần lời khai báo hàm để kiểm tra lời gọi hàm đúng hay sai về cú pháp, về số lượng các tham số, kiểu các tham số và cách sử dụng giá trị trả về.

    ```
    int SumInt(int a, int b);
    ```
  - Có thể khai báo hàm độc lập với việc định nghĩa hàm (tất nhiên phải đảm bảo nhất quán)
- Gọi hàm: yêu cầu thực thi mã hàm với tham số thực tế (tham trị)

```
int x = 5;
int k = SumInt(x, 10);
```

Tên hàm Tham số (gọi hàm)

Khi biên dịch chưa cần phải có định nghĩa hàm, nhưng phải có khai báo hàm!

Khai báo hàm C/C++ ở đâu?

- Ở phạm vi toàn cục (ngoài bất cứ hàm nào)
- Một hàm phải được khai báo trước lời gọi đầu tiên trong một tệp tin mã nguồn
- Nếu sử dụng nhiều hàm thì sẽ cần rất nhiều dòng mã khai báo (mất công viết, dễ sai và mã chương trình lớn lên?):
  - Nếu người xây dựng hàm (định nghĩa hàm) đưa sẵn tất cả phần khai báo vào trong một tệp tin => **Header file** (*.h, *.hx,...) thì người sử dụng chỉ cần bổ sung dòng lệnh

    `#include <filename>`
  - Mã chương trình không lớn lên, bởi khai báo không sinh mã!
- Một hàm có thể khai báo nhiều lần tùy ý!

Định nghĩa hàm ở đâu?

- Ở phạm vi toàn cục (ngoài bất cứ hàm nào)
- Có thể định nghĩa trong cùng tệp tin với mã chương trình chính, hoặc tách ra một tệp tin riêng. Trong Visual C++:

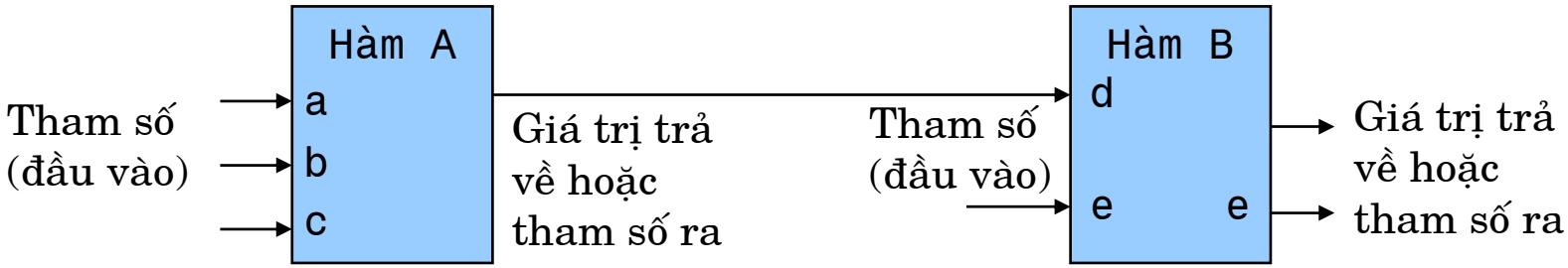
  `*.c` => C compiler,

  `*.cpp` => C++ compiler
- Một hàm đã có lời gọi thì phải được định nghĩa chính xác 1 lần trong toàn bộ (dự án) chương trình, trước khi gọi trình liên kết (lệnh Build trong Visual C++)
- Đưa tệp tin mã nguồn vào dự án, không nên:

  `#include "xxx.cpp"`
- Một hàm có được định nghĩa bằng C, C++, hợp ngữ hoặc bằng một ngôn ngữ khác và dùng trong C/C++ => Sử dụng hàm không cần mã nguồn!
- Một thư viện cho C/C++ bao gồm:
  - Header file (thường đuôi `*.h, *.hxx`, ..., nhưng không bắt buộc)
  - Tệp tin mã nguồn (`*.c, *.cpp, *.cxx,...`) hoặc mã đích (`*.obj, *.o, *.lib, *.dll, ...`)

3.3 Truyền tham số và trả về kết quả

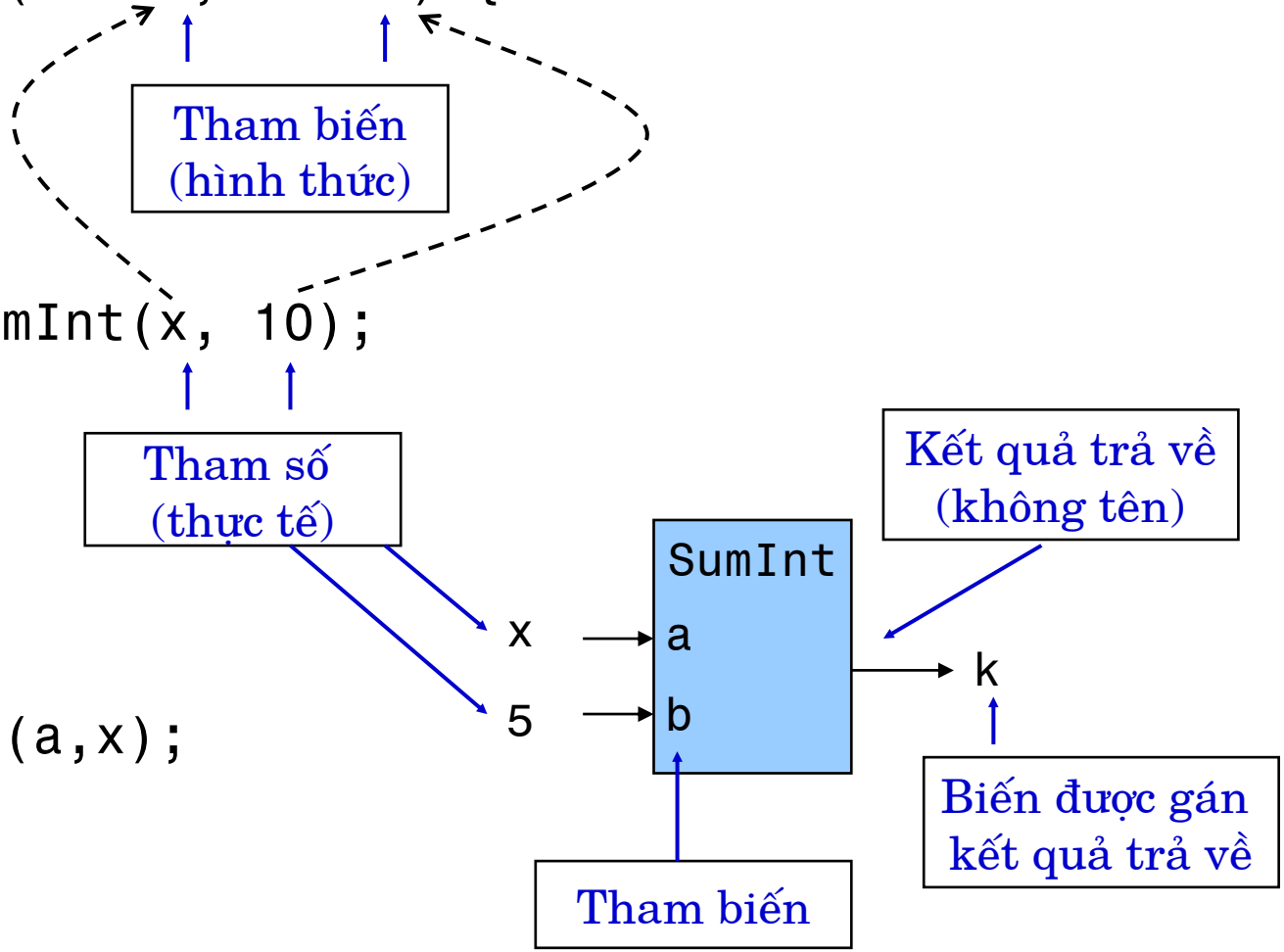
- Truyền tham số và trả về kết quả là phương pháp cơ bản để tổ chức quan hệ giữa các hàm (giữa các chức năng trong hệ thống)



- Ngoài ra, còn có các cách khác:
  - Sử dụng biến toàn cục: nói chung là không nên!
  - Sử dụng các tệp tin, streams: dù sao vẫn phải sử dụng tham số để nói rõ tệp tin nào, streams nào
  - Các cơ chế giao tiếp hệ thống khác (phụ thuộc vào hệ điều hành, nền tảng và giao thức truyền thông) => nói chung vẫn cần các tham số bổ sung
- Truyền tham số & trả về kết quả là một vấn đề cốt lõi trong xây dựng và sử dụng hàm, một trong những yếu tố ảnh hưởng quyết định tới chất lượng phần mềm!

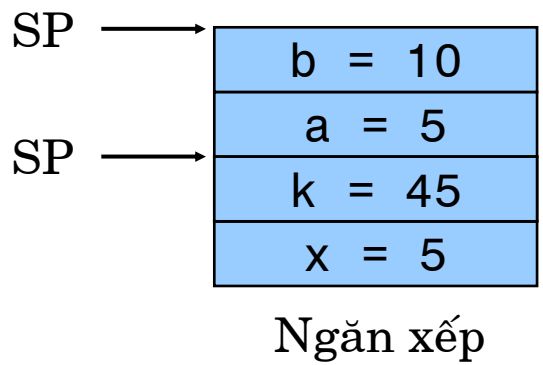
Tham biến hình thức và tham số thực tế

```
int SumInt(int a, int b) {
    ...
}

int x = 5;
int k = SumInt(x, 10);
...

int a = 2;
k = SumInt(a,x);
```



3.3.1 Truyền giá trị

```
int SumInt(int, int);

// Function call
void main() {
    int x = 5;
    int k = SumInt(x, 10);
    ...
}

// Function definition
int SumInt(int a, int b) {
    ...


}
```



Thử ví dụ đọc từ bàn phím

```
#include <iostream.h>
void ReadInt(const char* userPrompt, int N) {
   cout << userPrompt;
   cin >> N;
}

void main() {
   int x = 5;
   ReadInt("Input an integer number:", x);
   cout << "Now x is " << x;
   ...
}
```

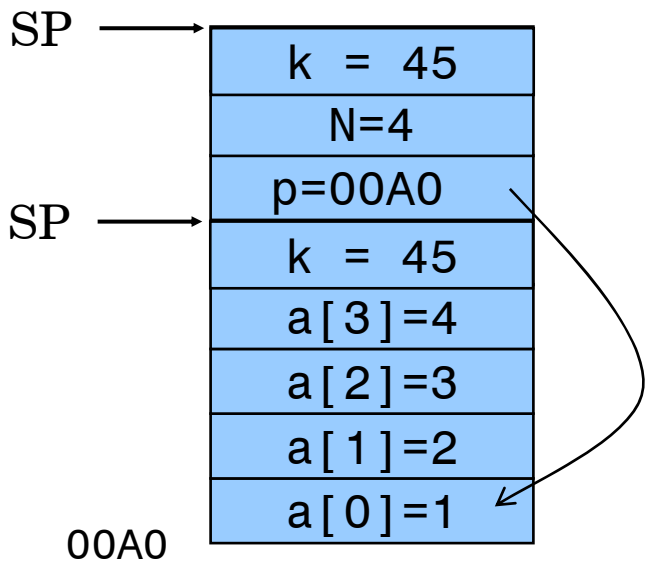
- Kết quả: x không hề thay đổi sau đó.

Truyền giá trị

- Truyền giá trị là cách thông thường trong C
- Tham biến chỉ nhận được bản sao của biến đầu vào (tham số thực tế)
- Thay đổi tham biến chỉ làm thay đổi vùng nhớ cục bộ, không làm thay đổi biến đầu vào
- Tham biến chỉ có thể mang tham số đầu vào, không chứa được kết quả (tham số ra)
- Truyền giá trị khá an toàn, tránh được một số hiệu ứng phụ
- Truyền giá trị trong nhiều trường hợp kém hiệu quả do mất công sao chép dữ liệu

3.3.2 Truyền địa chỉ

```
int SumInt(int* p, int N);
    // Function call
void main() {
    int a[] = {1, 2, 3, 4};
    int k = SumInt(a,4);
    ...
}

// Function definition
int SumInt(int* p, int N) {
    int *p2 = p + N, k = 0;
    while (p < p2)
        k += *p++;
    return k;
}
```



Truyền mảng tham số?

```
int SumInt(int p[4], int N);
    // Function call
void main() {
    int a[] = {1, 2, 3, 4};
    int k = SumInt(a,4);
    ...
}

// Function definition
int SumInt(int p[4], int N) {
    int *p2 = p + N, k = 0;
    while (p < p2)
          k += *p++;
    return k;
}
```

Bản chất như trong ví dụ trước: Truyền địa chỉ!

Thử lại ví dụ đọc từ bàn phím

```
#include <iostream.h>
void ReadInt(const char* userPrompt, int* pN) {
   cout << userPrompt;
   cin >> *pN;
}

void main() {
   int x = 5;
   ReadInt("Input an integer number:", &x);
   cout << "Now x is " << x;
   ...
}
```

- Kết quả: x thay đổi giá trị sau đó (cũng là lý do tại sao hàm scanf() lại yêu cầu kiểu tham biến là con trỏ!)

Khi nào sử dụng truyền địa chỉ?

- Khi cần thay đổi "biến đầu vào" (truy nhập trực tiếp vào ô nhớ, không qua bản sao)
- Khi kích cỡ kiểu dữ liệu lớn => tránh sao chép dữ liệu vào ngăn xếp
- Truyền tham số là một mảng => bắt buộc truyền địa chỉ

- Lưu ý: Sử dụng con trỏ để truyền địa chỉ của vùng nhớ dữ liệu đầu vào. Bản thân con trỏ có thể thay đổi được trong hàm nhưng địa chỉ vùng nhớ không thay đổi (nội dung của vùng nhớ đó thay đổi được): xem ví dụ biến p trong hàm SumInt trang 21.

3.3.3 Truyền tham chiếu (C++)

```
#include <iostream.h>
void ReadInt(const char* userPrompt, int& N) {
   cout << userPrompt;
   cin >> N;
}

void main() {
   int x = 5;
   ReadInt("Input an integer number:", x);
   cout << "Now x is " << x;
   ...
}
```

- Kết quả: x thay đổi giá trị sau đó

Thử ví dụ hàm swap

```
#include <iostream.h>
void swap(int& a, int& b) {
   int temp = a;
   a = b;
   b = temp;
}

void main() {
   int x = 5, y = 10;
   swap(x,y);
   cout << "Now x is " << x << ", y is " << y;
   ...
}
```

Khi nào sử dụng truyền tham chiếu?

- Chỉ trong C++
- Khi cần thay đổi "biến đầu vào" (truy nhập trực tiếp vào ô nhớ, không qua bản sao)
- Một tham biến tham chiếu có thể đóng vai trò là đầu ra (chứa kết quả), hoặc có thể vừa là đầu vào và đầu ra
- Khi kích cỡ kiểu dữ liệu lớn => tránh sao chép dữ liệu vào ngăn xếp, ví dụ:

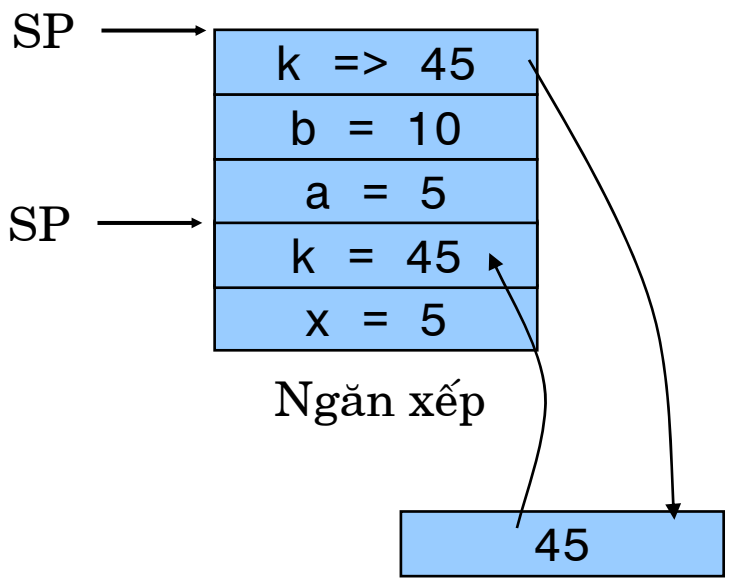
```
void copyData(const Student& sv1, Student& sv2) {
    sv2.birthday = sv1.birthday;
    ...
}
```

3.3.4 Kiểu trả về

- Kiểu trả về: gần như tùy ý, chỉ không thể trả về trực tiếp một mảng
- Về nguyên tắc, có thể trả về kiểu:
  - Giá trị
  - Con trỏ
  - Tham chiếu
- Tuy nhiên, cần rất thận trọng với trả về địa chỉ hoặc tham chiếu:
  - Không bao giờ trả về con trỏ hoặc tham chiếu vào biến cục bộ
  - Không bao giờ trả về con trỏ hoặc tham chiếu vào tham biến truyền qua giá trị
- Với người lập trình ít có kinh nghiệm: chỉ nên trả về kiểu giá trị

Cơ chế trả về

```
int SumInt(int a, int b) {
    int k = 0;
    for (int i=a; i <= b; ++i)
          k +=i;
    return k;
}

void main() {
    int x = 5, k = 0;
    k = SumInt(x,10);
    ...
}
```



Trả về con trỏ

- Viết hàm trả về địa chỉ của phần tử lớn nhất trong một mảng:

```
int* FindMax(int* p, int n) {
   int *pMax = p;
   int *p2 = p + n;
   while (p < p2) {
       if (*p > *pMax)
              pMax = p;
       ++p;
   }
   return pMax;
}
void main() {
   int s[5] = { 1, 2, 3, 4, 5};
   int *p = FindMax(s,5);
}
```

Lý do trả về con trỏ hoặc tham chiếu

- Tương tự như lý do truyền địa chỉ hoặc truyền tham chiếu:
  - Tránh sao chép dữ liệu lớn không cần thiết
  - Để có thể truy cập trực tiếp và thay đổi giá trị đầu ra
- Có thể trả về con trỏ hoặc tham chiếu vào đâu?
  - Vào biến toàn cục
  - Vào tham số truyền cho hàm qua địa chỉ hoặc qua tham chiếu
  - Nói chung: vào vùng nhớ mà còn tiếp tục tồn tại sau khi kết thúc hàm
- Con trỏ lại phức tạp thêm một chút?

Phản ví dụ: trả về con trỏ

```
int* f(int* p, int n) {
   int Max = *p;
   int *p2 = p + n;
   while (p < p2) {
       if (*p > Max)
             Max = *p;
       ++p;
   }
   return &Max;
}
void main() {
   int s[5] = { 1, 2, 3, 4, 5};
   int *p = FindMax(s,5);  // get invalid address
}
```

Các ví dụ nghiên cứu: Đúng / sai?

```
int* f1(int a) {
   ...
   return &a;
}
int& f2(int &a) {
   ...
   return a;
}
int f3(int &a) {
   ...
   return a;
}
int* f4(int *pa) {
   ...
   return pa;
}
```

```
int f5(int *pa) {
   ...
   return *pa;
}
int& f6(int *pa) {
   ...
   return *pa;
}
int& f7(int a) {
   ...
   return a;
}
int *pa;
int* f8() {
   ...
   return pa;
}
```

3.4 Thiết kế hàm và thư viện

- Viết một chương trình chạy tốt đã khó, viết một thư viện hàm tốt còn khó hơn!
- Một thư viện hàm định nghĩa:
  - một tập hợp các hàm (có liên quan theo một chủ đề chức năng)
  - những kiểu dữ liệu sử dụng trong các hàm
  - một số biến toàn cục (rất hạn chế)
- Một thư viện hàm tốt cần phải:
  - Thực hiện những chức năng hữu ích
  - Đơn giản, dễ sử dụng
  - Hiệu suất và độ tin cậy cao
  - Trọn vẹn, nhất quán và đồng bộ

Thiết kế hàm

- Phân tích yêu cầu:
  - Làm rõ các dữ kiện (đầu vào) và kết quả (đầu ra)
  - Tìm ra các chức năng cần thực hiện
- Đặt tên hàm: ngắn gọn, ý nghĩa xác đáng, tự miêu tả
  - Hàm chỉ hành động: Chọn tên hàm là một động từ kết hợp với kiểu đối tượng chủ thể, ví dụ `printVector, displayMatrix, addComplex, sortEventQueue, filterAnalogSignal`,...
  - Hàm truy nhập thuộc tính: Có thể chọn là động từ hoặc danh từ kết hợp kiểu đối tượng chủ thể, ví dụ length, size, numberOfColums, getMatrixElem, putShapeColor
  - Trong C++ nhiều hàm có thể giống tên (nạp chồng tên hàm), có thể chọn tên ngắn, ví dụ sort, print, display, add, putColor, getColor => nguyên tắc đa hình/đa xạ theo quan điểm hướng đối tượng
  - Trong C++ còn có thể định nghĩa hàm toán tử để có thể sử dụng các ký hiệu toán tử định nghĩa sẵn như *, /, +, - thay cho lời gọi hàm.

- Chọn tham số đầu vào (=> tham biến)
  - Đặc tả ý nghĩa: Thể hiện rõ vai trò tham số
  - Đặt tên: Ngắn gọn, tự mô tả
  - Chọn kiểu: Kiểu nhỏ nhất mà đủ biểu diễn
  - Chọn cách truyền tham số: cân nhắc giữa truyền giá trị hay truyền địa chỉ/tham chiếu vào kiểu hằng
- Chọn tham số đầu ra (=> tham biến truyền qua địa chỉ/qua tham chiếu hoặc sử dụng giá trị trả về)
  - Đặc tả ý nghĩa, đặt tên, chọn kiểu tương tự như tham số đầu vào
- Định nghĩa bổ sung các kiểu dữ liệu mới như cần thiết
- Mô tả rõ tiền trạng (*pre-condition*): điều kiện biên cho các tham số đầu vào và các điều kiện ngoại cảnh cho việc gọi hàm
- Mô tả rõ hậu trạng (*post-condition*): tác động của việc sử dụng hàm tới ngoại cảnh, các thao tác bắt buộc sau này,...
- Thiết kế thân hàm dựa vào các chức năng đã phân tích, sử dụng lưu đồ thuật toán với các cấu trúc điều kiện/rẽ nhánh (kể cả vòng lặp) => có thể phân chia thành các hàm con nếu cần

Ví dụ minh họa: Tìm số nguyên tố

Bài toán: Xây dựng hàm tìm N số nguyên tố đầu tiên!

- Phân tích:
  - Dữ kiện: N - số số nguyên tố đầu tiên cần tìm
  - Kết quả: Một dãy N số nguyên tố đầu tiên
  - Các chức năng cần thực hiện:
    - Nhập dữ liệu? KHÔNG!
    - Kiểm tra dữ kiện vào (N)? Có/không (Nếu kiểm tra mà N nhỏ hơn 0 thì hàm làm gì?)
    - Cho biết k số nguyên tố đầu tiên, xác định số nguyên tố tiếp theo
    - Lưu trữ kết quả mỗi lần tìm ra vào một cấu trúc dữ liệu phù hợp (dãy số cần tìm)
    - In kết quả ra màn hình? KHÔNG!

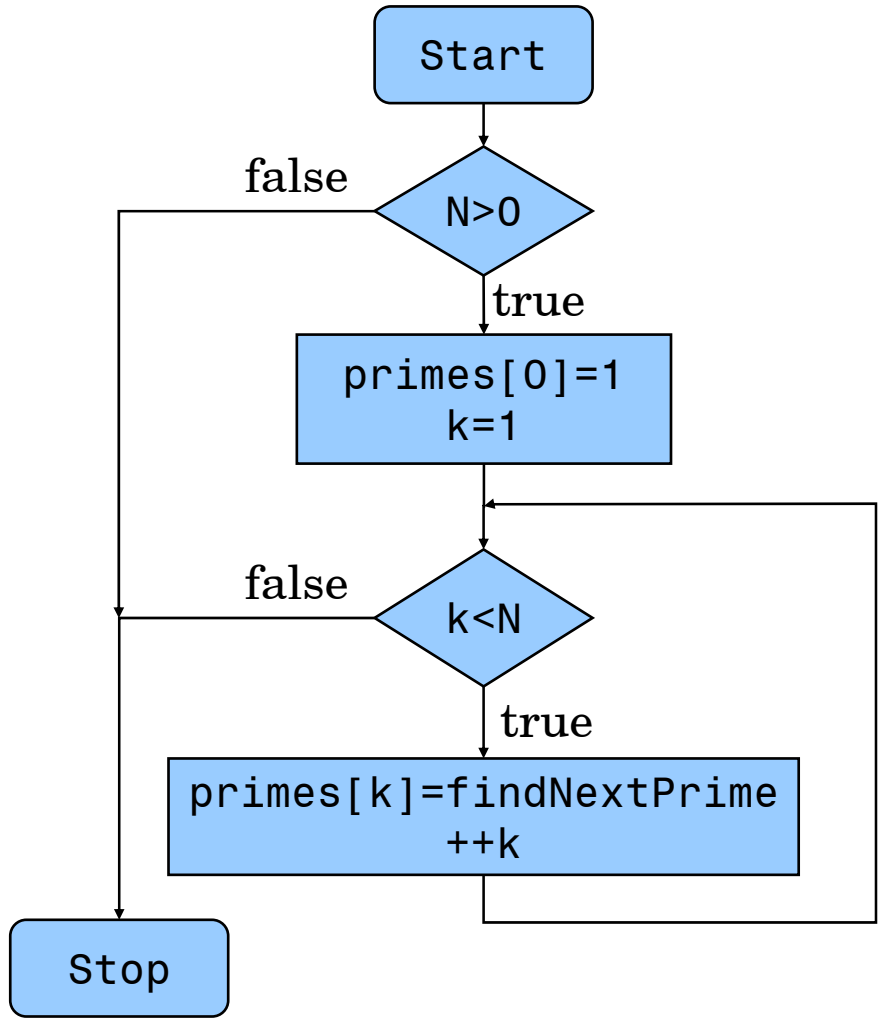
- Đặt tên hàm: `findPrimeSequence`
- Tham số vào: 1
  - Ý nghĩa: số các số nguyên tố cần tìm
  - Tên: `N`
  - Kiểu: số nguyên đủ lớn (`int/long`)
  - Truyền tham số: qua giá trị
- Tham số ra: 1
  - Ý nghĩa: dãy N số nguyên tố đầu tiên tính từ 1
  - Giá trị trả về hay tham biến? **Tham biến!**
  - Tên: `primes`
  - Kiểu: mảng số nguyên (của `int/long`)
  - Truyền tham số: qua địa chỉ (`int*` hoặc `long*`)
- Tiền trạng:
  - Tham số `N` phải là số không âm (có nên chọn kiểu `unsigned`?)
  - `primes` phải mang địa chỉ của mảng số nguyên có ít nhất N phần tử
- Hậu trạng: không có gì đặc biệt

- Khai báo hàm:

```
void findPrimeSequence(int N, int* primes);
```

- Thiết kế thân hàm
  - Lưu đồ thuật toán như hình vẽ
  - Phân chia, bổ sung một hàm mới: `findNextPrime`
- Lặp lại qui trình thiết kế hàm cho `findNextPrime`

  (Bài tập về nhà!)



3.5 Thư viện chuẩn ANSI-C

Thư viện vào/ra (nhập/xuất)	`<stdio.h>`
Xử lý ký tự và chuỗi ký tự	`<string.h>, <ctype.h>`
Thư viện hàm toán	`<math.h>, <float.h>`
Thời gian, ngày tháng	`<time.h>, <locale.h>`
Cấp phát bộ nhớ động	`<stdlib.h>`
Các hàm ký tự rộng	`<wchar.h>,<wctype.h>`
Các hàm khác	`<stdlib.h>, ...`

3.6 Làm việc với tệp tin trong C++

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

- Khai báo một biến:

```
ifstream fin;  // input
ofstream fout;         // output
fstream  fio;  // input and output
```

- Mở/tạo một tệp tin:

```
fin.open("file1.txt");
fout.open("file2.dat");
fio.open("file3.inf");
```

- Kết hợp khai báo biến và mở/tạo một tệp tin

```
ifstream fin("file1.txt"); // input
ofstream fout("file2.inf");// output
fstream  fio("file3.dat"); // input and output
```

- Ghi dữ liệu ra tệp tin
  - Tương tự như sử dụng `cout`
  - Tệp tin có thể chứa dữ liệu kiểu hỗn hợp, ví dụ:

```
fout << "Nguyen Van A" << endl;
fout << 21 << endl << false;
```

- Đọc dữ liệu từ một tệp tin
  - Tương tự như sử dụng `cin`

```
char name[32];
int  age, married;
fin.getline(name,32);
fin >> age >> married;
```

- Đóng một tệp tin:
  - Tự động khi kết thúc phạm vi { },
  - Hoặc gọi hàm thành viên `close()`:

```
fin.close();
fout.close();
fio.close();
```

Ví dụ: làm việc với tệp tin

```
#include <iostream.h>
#include <fstream.h>
void main() {
   {
     ofstream fout("file1.dat");// output
     fout << "Nguyen Van A" << endl << 21 << endl << false;
   }
   {
     ifstream fin("file1.dat"); // input
     char name[32];
     int  age;
     int  married;
     fin.getline(name,32);
     fin >> age >> married;
     cout << "Name:\t" << name << endl;
     cout << "Age:\t" << age << endl;
     cout << "Married:" << (married ? "Yes" : "No");
   }
   char c;
   cin >> c;
}
```

3.7 Nạp chồng tên hàm trong C++

- Trong C++ có thể xây dựng nhiều hàm có cùng tên, ví dụ:

```
int    max(int a, int b);
double max(double a, double b);
double max(double a, double b, double c);
double max(double *seq, int n);
```

- Mục đích của **nạp chồng tên hàm** là:
  - Đơn giản hóa cho người xây dựng hàm trong việc chọn tên (thay vì `maxInt, maxDouble, maxDouble3, maxDoubleSequence,...`)
  - Đơn giản hóa cho người sử dụng hàm, chỉ cần nhớ 1 tên quen thuộc thay cho nhiều tên phức tạp

Ví dụ: định nghĩa các hàm max()

```
int max(int a, int b) {                              // (1)
   return (a > b)? a : b;
}
double max(double a, double b) {                     // (2)
   return (a > b)? a : b;
}
double max(double a, double b, double c); {          // (3)
   if (a < b) a = b;
   if (a < c) a = c;
   return a;
}
double max(double *seq, int n) {                     // (4)
   int i = 0, kq = seq[0];
   while (i < n) {
     if (kq < seq[i])kq = seq[i];
     ++i;
   }
   return kq;
}
```

Ví dụ: sử dụng các hàm max()

```
int max(int a, int b);                          // (1)
double max(double a, double b);                 // (2)
double max(double a, double b, double c);       // (3)
double max(double *seq, int n);                 // (4)

void main() {
   int k = max(5,7);                 // call (1)
   double d = max(5.0,7.0);          // call (2)
   double a[] = {1,2,3,4,5,6};
   d = max(d, a[1], a[2]);           // call (3)
   d = max(a, 5);                    // call (4)
   d = max(5,7);                     // ?
   d = max(d, 5);                    // ?
}
```

➲ Đẩy trách nhiệm kiểm tra và tìm hàm phù hợp cho compiler!

Một số qui tắc về nạp chồng tên hàm

- Các hàm cùng tên được **định nghĩa cùng trong một file/ trong một thư viện** hoặc **sử dụng trong cùng một chương trình** phải khác nhau ít nhất về:
  - Số lượng các tham số, hoặc
  - Kiểu của ít nhất một tham số (`int` khác `short`, `const int` khác `int`, `int` khác `int&, ...`)

  ➲ Không thể chỉ khác nhau ở kiểu trả về
- Tại sao vậy?
  - Compiler cần có cơ sở để quyết định gọi hàm nào
  - Dựa vào cú pháp trong lời gọi (số lượng và kiểu các tham số thực tế) compiler sẽ chọn hàm có cú pháp phù hợp nhất
  - Khi cần compiler có thể tự động chuyển đổi kiểu theo chiều hướng **hợp lý nhất** (vd `short=>int, int => double`)

3.8 Hàm inline trong C++

- Vấn đề: Hàm tiện dụng, nhưng nhiều khi hiệu suất không cao, đặc biệt khi mã thực thi hàm ngắn
  - Các thủ tục như nhớ lại trạng thái chương trình, cấp phát bộ nhớ ngăn xếp, sao chép tham số, sao chép giá trị trả về, khôi phục trạng thái chương trình **mất nhiều thời gian**
  - Nếu mã thực thi hàm ngắn thì **sự tiện dụng không bõ so với sự lãng phí thời gian**
- Giải pháp trong C: Sử dụng macro, ví dụ

```
#define max(a,b)  a>b?a:b
```

  - Vấn đề: Macro do tiền xử lý chạy (preprocessor), không có kiểm tra kiểu, không có phân biệt ngữ cảnh => gây ra các hiệu ứng phụ không mong muốn

    Ví dụ dòng lệnh `l=max(k*5-2,l);`

    sẽ được thay thế bằng `l=k*5-2>k?k*5-2:l;  // OOPS!`
  - Những cách giải quyết như thêm dấu ngoặc chỉ làm mã khó đọc, không khắc phục triệt để các nhược điểm

Giải pháp hàm inline trong C++

- Điều duy nhất cần làm là thêm từ khóa `inline` vào đầu dòng khai báo và định nghĩa hàm

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

- Hàm inline khác gì hàm bình thường:
  - "Hàm inline" thực chất không phải là một hàm!
  - Khi gọi hàm thì lời gọi hàm được **thay thế một cách thông minh bởi mã nguồn định nghĩa hàm,** không thực hiện các thủ tục gọi hàm

Ví dụ:

```
l=max(k*5-2,l);
```

Được thay thế bằng các dòng lệnh kiểu như:

```
int x=k*5-2; // biến tạm trung gian
l=(x>l)?x:l; // OK
```

Khi nào nên dùng hàm inline

- Ưu điểm của hàm inline:
  - Tiện dụng như hàm bình thường
  - Hiệu suất như viết thẳng mã, không gọi hàm
  - Tin cậy, an toàn hơn nhiều so với sử dụng Macro
- Nhược điểm của hàm inline:
  - Nếu gọi hàm nhiều lần trong chương trình, mã chương trình có thể lớn lên nhiều (mã thực hiện hàm xuất hiện nhiều lần trong chương trình)
  - Mã định nghĩa hàm phải để mở => đưa trong header file
- Lựa chọn xây dựng và sử dụng hàm inline khi:
  - Mã định nghĩa hàm nhỏ (một vài dòng lệnh, không chứa vòng lặp)
  - Yêu cầu về **tốc độ** đặt ra trước **dung lượng bộ nhớ**

Bài tập về nhà

- Xây dựng hàm tìm N số nguyên tố đầu tiên
  - Hoàn thiện thiết kế hàm
  - Định nghĩa hàm
- Viết chương trình minh họa cách sử dụng

Kỹ thuật lập trình

Chương 4: Khái quát về cấu trúc dữ liệu

9/22/2005

Nội dung chương 4

4.1 Cấu trúc dữ liệu là gì?

4.2 Mảng và quản lý bộ nhớ động

4.2 Xây dựng cấu trúc Vector

4.3 Xây dựng cấu trúc List

4.1 Giới thiệu chung

- Phần lớn các bài toán trong thực tế liên quan tới các dữ liệu phức hợp, những kiểu dữ liệu cơ bản trong ngôn ngữ lập trình không đủ biểu diễn
- Ví dụ:
  - Dữ liệu sinh viên: Họ tên, ngày sinh, quê quán, mã số SV,...
  - Mô hình hàm truyền: Đa thức tử số, đa thức mẫu số
  - Mô hình trạng thái: Các ma trận A, B, C, D
  - Dữ liệu quá trình: Tên đại lượng, dải đo, giá trị, đơn vị, thời gian, cấp sai số, ngưỡng giá trị,...
  - Đối tượng đồ họa: Kích thước, màu sắc, đường nét, phông chữ, ...
- Phương pháp biểu diễn dữ liệu: định nghĩa kiểu dữ liệu mới sử dụng cấu trúc (struct, class, union, ...)

Vấn đề: Biểu diễn tập hợp dữ liệu

- Đa số những dữ liệu thuộc một ứng dụng có liên quan với nhau => cần biểu diễn trong một tập hợp có cấu trúc, ví dụ:
  - Danh sách sinh viên: Các dữ liệu sinh viên được sắp xếp theo thứ tự Alphabet
  - Mộ hình tổng thể cho hệ thống điều khiển: Bao gồm nhiều thành phần tương tác
  - Dữ liệu quá trình: Một tập dữ liệu có thể mang giá trị của một đại lượng vào các thời điểm gián đoạn, các dữ liệu đầu vào liên quan tới dữ liệu đầu ra
  - Đối tượng đồ họa: Một cửa sổ bao gồm nhiều đối tượng đồ họa, một bản vẽ cũng bao gồm nhiều đối tượng đồ họa
- Thông thường, các dữ liệu trong một tập hợp có cùng kiểu, hoặc ít ra là tương thích kiểu với nhau
- Kiểu mảng không phải bao giờ cũng phù hợp!

Vấn đề: Quản lý (tập hợp) dữ liệu

- Sử dụng kết hợp một cách khéo léo kiểu cấu trúc và kiểu mảng đủ để biểu diễn các tập hợp dữ liệu bất kỳ
- Các giải thuật (hàm) thao tác với dữ liệu, nhằm quản lý dữ liệu một cách hiệu quả:
  - Bổ sung một mục dữ liệu mới vào một danh sách, một bảng, một tập hợp, ...
  - Xóa một mục dữ liệu trong một danh sách, bảng, tập hợp,..
  - Tìm một mục dữ liệu trong một danh sách, bảng tập hợp,... theo một tiêu chuẩn cụ thể
  - Sắp xếp một danh sách theo một tiêu chuẩn nào đó
  - ....

Quản lý DL thế nào là hiệu quả?

- Tiết kiệm bộ nhớ: Phần "overhead" không đáng kể so với phần dữ liệu thực
- Truy nhập nhanh, thuận tiện: Thời gian cần cho bổ sung, tìm kiếm và xóa bỏ các mục dữ liệu phải ngắn
- Linh hoạt: Số lượng các mục dữ liệu không (hoặc ít) bị hạn chế cố định, không cần biết trước khi tạo cấu trúc, phù hợp với cả bài toán nhỏ và lớn
- Hiệu quả quản lý dữ liệu phụ thuộc vào
  - Cấu trúc dữ liệu được sử dụng
  - Giải thuật được áp dụng cho bổ sung, tìm kiếm, sắp xếp, xóa bỏ

Các cấu trúc dữ liệu thông dụng

- Mảng (nghĩa rộng): Tập hợp các dữ liệu có thể truy nhập tùy ý theo chỉ số
- Danh sách (list): Tập hợp các dữ liệu được móc nối đôi một với nhau và có thể truy nhập tuần tự
- Cây (tree): Tập hợp các dữ liệu được móc nối với nhau theo cấu trúc cây, có thể truy nhập tuần tự từ gốc
  - Nếu mỗi nút có tối đa hai nhánh: cây nhị phân (binary tree)
- Bìa, bảng (map): Tập hợp các dữ liệu có sắp xếp, có thể truy nhập rất nhanh theo mã khóa (key)
- Hàng đợi (queue): Tập hợp các dữ liệu có sắp xếp tuần tự, chỉ bổ sung vào từ một đầu và lấy ra từ đầu còn lại

Các cấu trúc dữ liệu thông dụng (tiếp)

- Tập hợp (set): Tập hợp các dữ liệu được sắp xếp tùy ý nhưng có thể truy nhập một cách hiệu quả
- Ngăn xếp (stack): Tập hợp các dữ liệu được sắp xếp tuần tự, chỉ truy nhập được từ một đầu
- Bảng hash (hash table): Tập hợp các dữ liệu được sắp xếp dựa theo một mã số nguyên tạo ra từ một hàm đặc biệt
- Bộ nhớ vòng (ring buffer): Tương tự như hàng đợi, nhưng dung lượng có hạn, nếu hết chỗ sẽ được ghi quay vòng
- Trong toán học và trong điều khiển: vector, ma trận, đa thức, phân thức, hàm truyền, ...

4.2 Mảng và quản lý bộ nhớ động

- Mảng cho phép biểu diễn và quản lý dữ liệu một cách khá hiệu quả:
  - Đọc và ghi dữ liệu rất nhanh qua chỉ số hoặc qua địa chỉ
  - Tiết kiệm bộ nhớ
- Các vấn đề của mảng tĩnh:

  `VD: Student student_list[100];`
  - Số phần tử phải là hằng số (biết trước khi biên dịch, người sử dụng không thể nhập số phần tử, không thể cho số phần từ là một biến) => kém linh hoạt
  - Chiếm chỗ cứng trong ngăn xếp (đối với biến cục bộ) hoặc trong bộ nhớ dữ liệu chương trình (đối với biến toàn cục) => sử dụng bộ nhớ kém hiệu quả, kém linh hoạt

Mảng động

- Mảng động là một mảng được cấp phát bộ nhớ theo yêu cầu, trong khi chương trình chạy

```
#include <stdlib.h>     /* C */
int n = 50;
...
float*  p1= (float*) malloc(n*sizeof(float)); /* C */
double* p2= new double[n];    // C++
```

- Sử dụng con trỏ để quản lý mảng động: Cách sử dụng không khác so với mảng tĩnh

```
p1[0] = 1.0f;
p2[0] = 2.0;
```

- Sau khi sử dụng xong => giải phóng bộ nhớ:

```
free(p1);         /* C */
delete [] p2;     // C++
```

Cấp phát và giải phóng bộ nhớ động

- C:
  - Hàm `malloc()` yêu cầu tham số là **số byte**, trả về con trỏ không kiểu (`void*`) mang địa chỉ vùng nhớ mới được cấp phát (nằm trong heap), trả về 0 nếu không thành công.
  - Hàm `free()` yêu cầu tham số là con trỏ không kiểu (`void*`), giải phóng vùng nhớ có địa chỉ đưa vào
- C++:
  - Toán tử `new` chấp nhận kiểu dữ liệu phần tử kèm theo số lượng phần tử của mảng cần cấp phát bộ nhớ (trong vùng heap), trả về con trỏ có kiểu, trả về 0 nếu không thành công.
  - Toán tử `delete[]` yêu cầu tham số là con trỏ có kiểu.
  - Toán tử `new` và `delete` còn có thể áp dụng cho cấp phát và giải phóng bộ nhớ cho một biến đơn, một đối tượng chứ không nhất thiết phải một mảng.

Một số điều cần lưu ý

- Con trỏ có vai trò quản lý mảng (động), chứ con trỏ không phải là mảng (động)
- Cấp phát bộ nhớ và giải phóng bộ nhớ chứ không phải cấp phát con trỏ và giải phóng con trỏ
- Chỉ giải phóng bộ nhớ một lần

```
int* p;
p[0] = 1;                  // never do it
new(p);                    // access violation!
p = new int[100];          // OK
p[0] = 1;                  // OK
int* p2=p;                 // OK
delete[] p2;               // OK
p[0] = 1;                  // access violation!
delete[] p;                // very bad!
p = new int[50];           // OK, new array
...
```

Cấp phát bộ nhớ động cho biến đơn

- **Ý nghĩa**: Các đối tượng có thể được tạo ra động, trong khi chương trình chạy (bổ sung sinh viên vào danh sách, vẽ thêm một hình trong bản vẽ, bổ sung một khâu trong hệ thống,...)
- Cú pháp

```
int* p = new int;
*p = 1;
p[0]= 2;                    // the same as above
p[1]= 1;                    // access violation!
int* p2 = new int(1);       // with initialization
delete p;
delete p2;
Student* ps = new Student;
ps->code = 1000;
...
delete ps;
```

- Một biến đơn **khác** với mảng một phần tử!

Ý nghĩa của sử dụng bộ nhớ động

- Hiệu suất:
  - Bộ nhớ được cấp phát đủ dung lượng theo yêu cầu và khi được yêu cầu trong khi chương trình đã chạy
  - Bộ nhớ được cấp phát nằm trong vùng nhớ tự do còn lại của máy tính (heap), chỉ phụ thuộc vào dung lượng bộ nhớ của máy tính
  - Bộ nhớ có thể được giải phóng khi không sử dụng tiếp.
- Linh hoạt:
  - Thời gian "sống" của bộ nhớ được cấp phát động có thể kéo dài hơn thời gian "sống" của thực thể cấp phát nó.
  - Có thể một hàm gọi lệnh cấp phát bộ nhớ, nhưng một hàm khác giải phóng bộ nhớ.
  - Sự linh hoạt cũng dễ dẫn đến những lỗi "rò rỉ bộ nhớ".

Ví dụ sử dụng bộ nhớ động trong hàm

```
Date* createDateList(int n) {
   Date* p = new Date[n];
   return p;
}
void main() {
   int n;
   cout << "Enter the number of your national holidays:";
   cin >> n;
   Date* date_list = createDateList(n);
   for (int i=0; i < n; ++i) {
        ...
   }
   for (....) { cout << ....}
   delete [] date_list;
}
```

Tham số đầu ra là con trỏ?

```
void createDateList(int n, Date* &p) {
   p = new Date[n];
}
void main() {
   int n;
   cout << "Enter the number of your national holidays:";
   cin >> n;
   Date* date_list;
   createDateList(n, date_list);
   for (int i=0; i < n; ++i) {
        ...
   }
   for (....) { cout << ....}
   delete [] date_list;
}
```

4.3 Xây dựng cấu trúc Vector

- Vấn đề: Biểu diễn một vector toán học trong C/C++?
- Giải pháp chân phương: mảng động thông thường, nhưng...
  - Sử dụng không thuận tiện: Người sử dụng tự gọi các lệnh cấp phát và giải phóng bộ nhớ, trong các hàm luôn phải đưa tham số là số chiều.
  - Sử dụng không an toàn: Nhầm lẫn nhỏ dẫn đến hậu quả nghiêm trọng

```
int n = 10;
double *v1,*v2, d;
v1 = (double*) malloc(n*sizeof(double));
v2 = (double*) malloc(n*sizeof(double));
d = scalarProd(v1,v2,n); // scalar_prod đã có
d = v1 * v2;             // OOPS!
v1.data[10] = 0;         // OOPS!
free(v1);
free(v2);
```

Định nghĩa cấu trúc Vector

- Tên file: `vector.h`
- Cấu trúc dữ liệu:

```
struct Vector {
  double *data;
  int     nelem;
};
```

- Khai báo các hàm cơ bản:

```
Vector createVector(int n, double init);
void   destroyVector(Vector);
double getElem(Vector, int i);
void   putElem(Vector, int i, double d);
Vector addVector(Vector, Vector);
Vector subVector(Vector, Vector);
double scalarProd(Vector, Vector);
...
```

Định nghĩa các hàm cơ bản

- Tên file: vector.cpp

```
#include <stdlib.h>
#include "vector.h"
Vector createVector(int n, double init) {
   Vector v;
   v.nelem = n;
   v.data = (double*) malloc(n*sizeof(double));
   while (n--) v.data[n] = init;
   return v;
}
void destroyVector(Vector v) {
   free(v.data);
}
double getElem(Vector v, int i) {
   if (i < v.nelem && i >= 0) return v.data[i];
   return 0;
}
```

```
void putElem(Vector v, int i, double d) {
   if (i >=0 && i < v.nelem) v.data[i] = d;
}
Vector addVector(Vector a, Vector b) {
   Vector c = {0,0};
   if (a.nelem == b.nelem) {
        c = createVector(a.nelem,0.0);
        for (int i=0; i < a.nelem; ++i)
          c.data[i] = a.data[i] + b.data[i];
   }
   return c;
}
Vector subVector(Vector a, Vector b) {
   Vector c = {0,0};
   ...
   return c;
}
```

Ví dụ sử dụng

```
#include "vector.h"
void main() {
   int n = 10;
   Vector a,b,c;
   a = createVector(10,1.0);
   b = createVector(10,2.0);
   c = addVector(a,b);
   //...
   destroyVector(a);
   destroyVector(b);
   destroyVector(c);
}
```

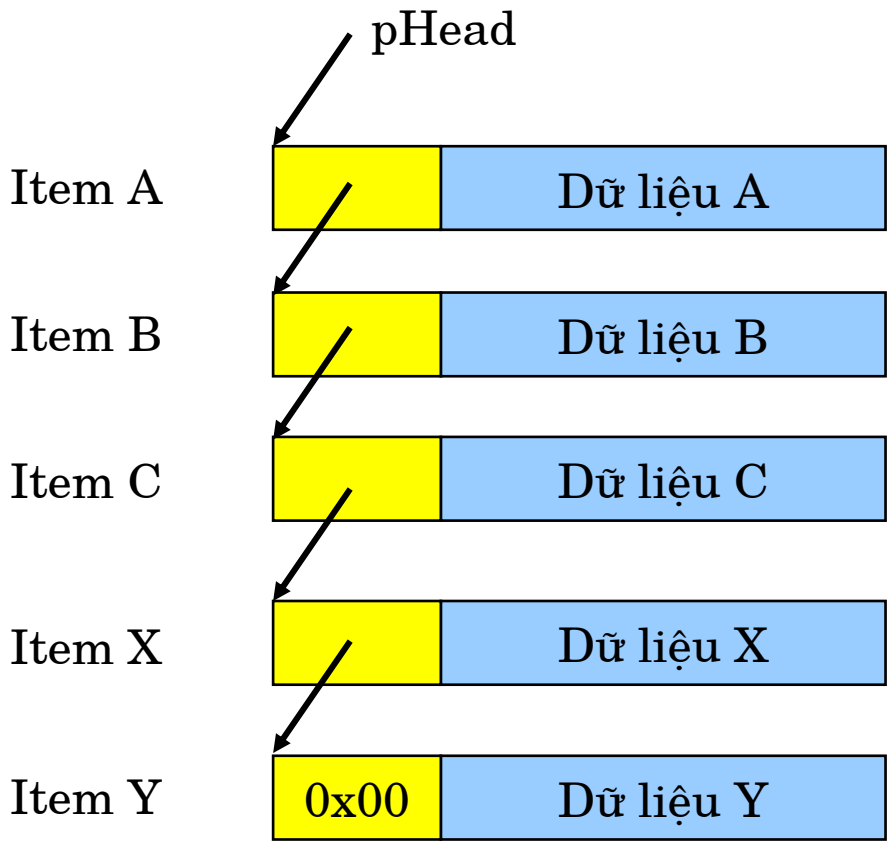
4.4 Xây dựng cấu trúc List

- Vấn đề: Xây dựng một cấu trúc để quản lý một cách hiệu quả và linh hoạt các dữ liệu động, ví dụ:
  - Hộp thư điện tử
  - Danh sách những việc cần làm
  - Các đối tượng đồ họa trên hình vẽ
  - Các khâu động học trong sơ đồ mô phỏng hệ thống (tương tự trong SIMULINK)
- Các yêu cầu đặc thù:
  - Số lượng mục dữ liệu trong danh sách có thể thay đổi thường xuyên
  - Các thao tác bổ sung hoặc xóa dữ liệu cần được thực hiện nhanh, đơn giản
  - Sử dụng tiết kiệm bộ nhớ

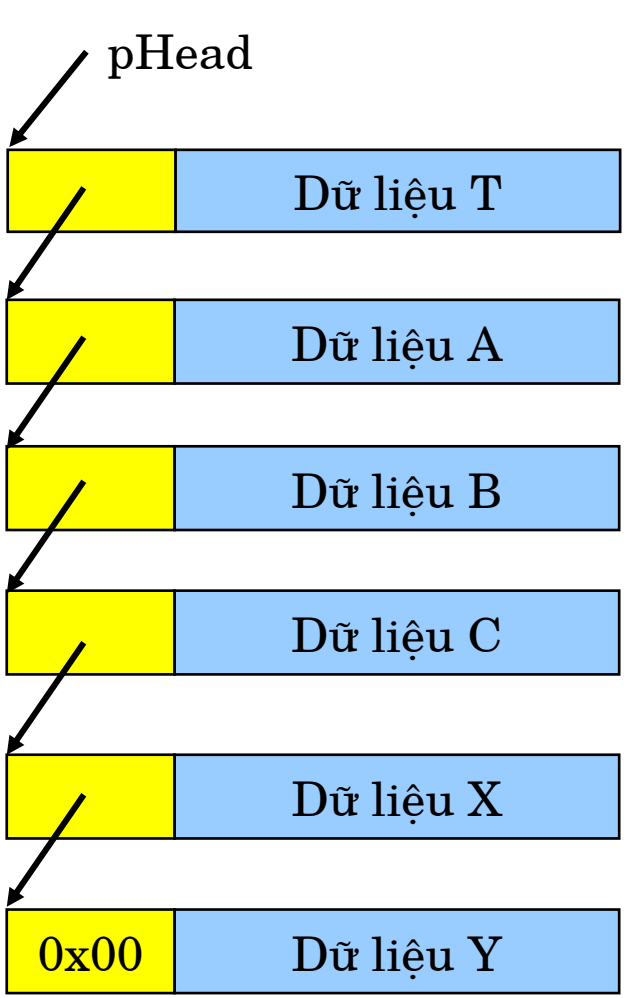
Sử dụng kiểu mảng?

- Số phần tử trong một mảng thực chất không bao giờ thay đổi được. Dung lượng bộ nhớ vào thời điểm cấp phát phải biết trước, không thực sự co giãn được.
- Nếu không thực sự sử dụng hết dung lượng đã cấp phát => lãng phí bộ nhớ
- Nếu đã sử dụng hết dung lượng và muốn bổ sung phần tử thì phải cấp phát lại và sao chép toàn bộ dữ liệu sang mảng mới => cần nhiều thời gian nếu số phần tử lớn
- Nếu muốn chèn một phần tử/xóa một phần tử ở đầu hoặc giữa mảng thì phải sao chép và dịch toàn bộ phần dữ liệu còn lại => rất mất thời gian

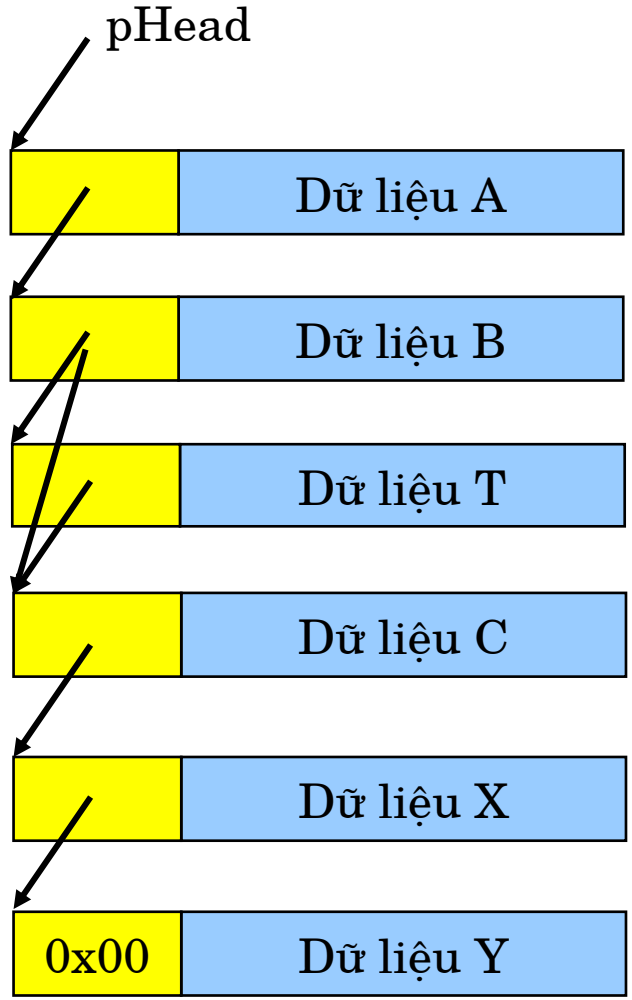
Danh sách móc nối (linked list)



Bổ sung dữ liệu

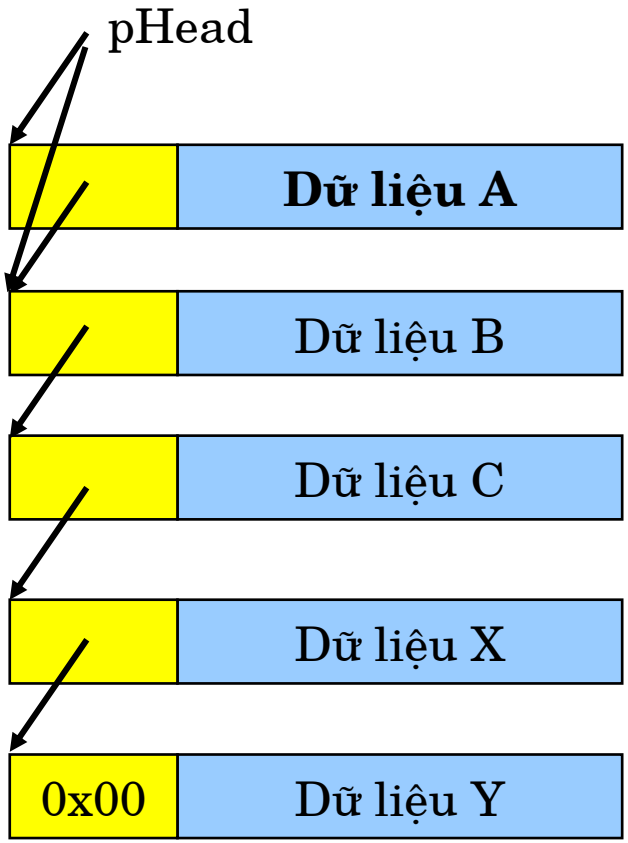


Bổ sung vào đầu danh sách

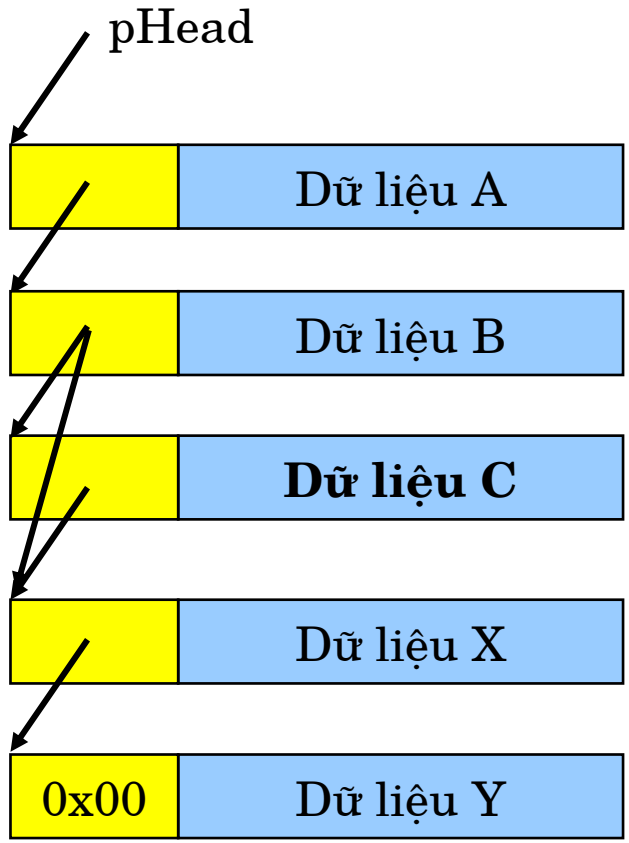


Bổ sung vào giữa danh sách

Xóa bớt dữ liệu



Xóa dữ liệu đầu danh sách



Xóa dữ liệu giữa danh sách

Các đặc điểm chính

- Ưu điểm:
  - Sử dụng rất linh hoạt, cấp phát bộ nhớ khi cần và xóa khi không cần
  - Bổ sung và xóa bỏ một dữ liệu được thực hiện thông qua chuyển con trỏ, thời gian thực hiện là hằng số, không phụ thuộc vào chiều dài và vị trí
  - Có thể truy nhập và duyệt các phần tử theo kiểu tuần tự
- Nhược điểm:
  - Mỗi dữ liệu bổ sung mới đều phải được cấp phát bộ nhớ động
  - Mỗi dữ liệu xóa bỏ đi đều phải được giải phóng bộ nhớ tương ứng
  - Nếu kiểu dữ liệu không lớn thì phần overhead chiếm tỉ lệ lớn
  - Tìm kiếm dữ liệu theo kiểu tuyến tính, mất thời gian

Ví dụ: Danh sách thông báo (hộp thư)

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

struct MessageItem {
   string subject;
   string content;
   MessageItem* pNext;
};
struct MessageList {
   MessageItem* pHead;
};
void initMessageList(MessageList& l);
void addMessage(MessageList&, const string& sj,
                const string& ct);
bool removeMessageBySubject(MessageList& l,
                            const string& sj);
void removeAllMessages(MessageList&);
```

```
#include "List.h"
void initMessageList(MessageList& l) {
   l.pHead = 0;
}
void addMessage(MessageList& l, const string& sj,
                const string& ct) {
   MessageItem* pItem = new MessageItem;
   pItem->content = ct;
   pItem->subject = sj;
   pItem->pNext = l.pHead;
   l.pHead = pItem;
}
void removeAllMessages(MessageList& l) {
   MessageItem *pItem = l.pHead;
   while (pItem != 0) {
       MessageItem* pItemNext = pItem->pNext;
       delete pItem;
       pItem = pItemNext;
   }
   l.pHead = 0;
}
```

```
bool removeMessageBySubject(MessageList& l,
                            const string& sj) {
   MessageItem* pItem = l.pHead;
   MessageItem* pItemBefore;
   while (pItem != 0 && pItem->subject != sj) {
       pItemBefore = pItem;
       pItem = pItem->pNext;
   }
   if (pItem != 0) {
       if (pItem == l.pHead)
              l.pHead = 0;
       else
              pItemBefore->pNext = pItem->pNext;
       delete pItem;
   }
   return pItem != 0;
}
```

Chương trình minh họa

```
#include <iostream>
#include "list.h"
using namespace std;
void main() {
   MessageList myMailBox;
   initMessageList(myMailBox);
   addMessage(myMailBox,"Hi","Welcome, my friend!");
   addMessage(myMailBox,"Test","Test my mailbox");
   addMessage(myMailBox,"Lecture Notes","Programming Techniques");
   removeMessageBySubject(myMailBox,"Test");
   MessageItem* pItem = myMailBox.pHead;
   while (pItem != 0) {
        cout << pItem->subject << ":" << pItem->content << '\n';
        pItem = pItem->pNext;
   }
   char c;
   cin >> c;
   removeAllMessages(myMailBox);
}
```

Bài tập về nhà

- Xây dựng kiểu danh sách móc nối chứa các ngày lễ trong năm và ý nghĩa của mỗi ngày (string), cho phép:
  - Bổ sung một ngày lễ vào đầu danh sách
  - Tìm ý nghĩa của một ngày (đưa ngày tháng là tham số)
  - Xóa bỏ đi một ngày lễ ở đầu danh sách
  - Xóa bỏ đi một ngày lễ ở giữa danh sách (đưa ngày tháng là tham số)
  - Xóa bỏ đi toàn bộ danh sách
- Viết chương trình minh họa cách sử dụng

Kỹ thuật lập trình

Phần III: Lập trình hướng đối tượng

Chương 5: Lớp và đối tượng

9/28/2005

Nội dung chương 5

5.1 Khái niệm

Đối tượng là gì?

- Thực thể phần mềm
- Mô hình/đại diện của một đối tượng vật lý:
  - Tank, Heater, Furnace
  - Motor, Pump, Valve
  - Sensor, Thermometer, Flowmeter
  - Control Loop, Control System
- Hoặc một đối tượng logic ("conceptual object):
  - Trend, Report, Button, Window
  - Matrix, Vector, Polynomial

Một đối tượng có...

- Các thuộc tính (attributes)
- Trạng thái (state)
  - Dữ liệu
  - Quan hệ
- Hành vi (behavior)
  - Các phép toán
  - Đặc tính phản ứng
- Căn cước (identity)
- Ngữ nghĩa/trách nhiệm (semantic/responsibilities)

Lớp là gì?

- Một lớp là thực thi của các đối tượng có chung
  - Ngữ nghĩa
  - Thuộc tính
  - Quan hệ
  - Hành vi
- Lớp = Đóng gói [Cấu trúc dữ liệu + hàm thao tác]
  - Lớp các vector, lớp các ma trận (dữ liệu phần tử + các phép truy nhập và phép toán cơ bản)
  - Lớp các hình chữ nhật (các dữ liệu tọa độ + phép vẽ, xóa,...)
  - Lớp các mô hình hàm truyền (các hệ số đa thức tử/mẫu, các phép toán xác định tính ổn định, xác định các điểm cực,...)
- Các dữ liệu của một lớp => biến thành viên
- Các hàm của một lớp => hàm thành viên
- Các biến của một lớp => một đối tượng, một thể nghiệm

Lập trình hướng đối tượng (object-oriented programming, OOP)

- Trừu tượng hóa (*abstraction*): giúp đơn giản hóa vấn đề, dễ sử dụng lại
- Đóng gói dữ liệu/che dấu thông tin (*data encapsulation/ information hiding*): nâng cao giá trị sử dụng lại và độ tin cậy của phần mềm
- Dẫn xuất/thừa kế (*subtyping/inheritance*): giúp dễ sử dụng lại mã phần mềm và thiết kế
- Đa hình/đa xạ (*polymorphism*): giúp phản ánh trung thực thế giới thực và nâng cao tính linh hoạt của phần mềm

Phương pháp luận hướng đối tượng cho phép tư duy ở mức ***trừu tượng cao*** *nhưng* ***gần với thế giới thực!***

5.2 Từ cấu trúc sang lớp

```
struct Time {
   int hour; // gio
   int min;  // phut
   int sec;  // giay
};
void addHour(Time& t, int h) {
  t.hour += h;
}
void addMin(Time& t, int m) {
  t.min += m;
  if (t.min > 59) {
    t.hour += t.min/60;
    t.min %= 60;
  }
  else if (t.min < 0) {
    t.hour += (t.min/60 - 1);
    t.min = (t.min % 60) + 60;
  }
}
```

```
void addSec(Time& t, int s) {
  t.sec += s;
  if (t.sec > 59) {
    addMin(t, t.sec/60);
    t.sec %= 60;
  }
  else if (t.sec < 0) {
    addMin(t, t.sec/60 - 1);
    t.sec = (t.sec % 60) + 60;
  }
}
void main() {
   Time t = {1, 0, 0};
   addMin(t,60);
   addMin(t,-5);
   addSec(t,25);
   ...
}
```

Một số vấn đề của cấu trúc

- Truy nhập dữ liệu trực tiếp, không có kiểm soát có thể dẫn đến không an toàn

```
Time t1 = {1, 61, -3};      // ??!
Time t2;                    // Uncertain values
int h = t2.hour;            // ??!
int m = 50;
t2.min = m + 15;            // ??!
```

- Không phân biệt giữa "chi tiết bên trong" và "giao diện bên ngoài", một thay đổi nhỏ ở chi tiết bên trong cũng bắt người sử dụng phải thay đổi mã sử dụng theo!

  Ví dụ: cấu trúc Time được sửa lại tên biến thành viên:

```
struct Time {
  int h, m, s;
};
```

  Đoạn mã cũ sẽ không biên dịch được:

```
Time t;
t.hour = 5;
```

Đóng gói hay "lớp hóa"

```
class Time {
   int hour; // gio
   int min;  // phut
   int sec;  // giay
public:
   Time() {hour=min=sec=0;}

   void setTime(int h, int m, int s)
   {
        hour = h;
        min = sec = 0;
        addSec(s);
        addMin(m);
   }

   int getHour() { return hour; }
   int getMin()  { return min;  }
   int getSec()  { return sec;  }

   void addHour(int h) { hour += h; }
   ...
```

Biến thành viên (member variable)

Hàm tạo (constructor)

Hàm thành viên (member functions)

```
  void addMin(int m) {
    min += m;
    if (min > 59) {
      hour += min/60;
      min %= 60;
    }
    else if (min < 0) {
      hour += (min/60 - 1);
      min = (min % 60) + 60;
    }
  }


    sec += s;
    if (sec > 59) {
      addMin(sec/60);
      sec %= 60;
    }
    else if (sec < 0) {
      addMin(sec/60 - 1);
      sec = (sec % 60) + 60;
    }
  }
};
```

```
void main()
{
   Time t;
   t.addHour(1);
   t.addMin(60);
   t.addMin(-5);
   t.addSec(25);
   t.hour = 1;  // error
   t.min  = 65; // error
    t.sec  = -3; // error
   t.setTime(1, 65, -3);
   int h = t.getHour();
   int m = t.getMin();
   int s = t.getSec();
}
```

5.3 Biến thành viên

- Khai báo biến thành viên của một lớp tương tự như cấu trúc

```
class Time {
  int hour, min, sec;
  ...
};
```

- Mặc định, các biến thành viên của một lớp không truy nhập được từ bên ngoài (biến riêng), đương nhiên cũng không khởi tạo được theo cách cổ điển:

```
Time t = {1, 0, 0};     // error!
t.hour = 2;             // error!
```

- Có thể làm cho một biến thành viên truy nhập được từ bên ngoài (biến công cộng), tuy nhiên ít khi có lý do cần làm như thế:

```
class Point {
public:
  int x,y;
};
```

- Kiểm soát việc truy nhập các biến riêng thông qua các hàm thành viên
- Cách duy nhất để khởi tạo giá trị cho các biến thành viên là sử dụng hàm tạo:

```
class Time {
...
public:
  Time() {hour=min=sec=0;}
};
Time t;   // t.hour = t.min = t.sec = 0;
```

- Một số biến thành viên có vai trò lưu trữ trạng thái bên trong của đối tượng, không nên cho truy nhập từ bên ngoài (ngay cả gián tiếp qua các hàm)

```
class PID {
  double Kp, Ti, Td;    // controller parameters
  double I;             // internal state
  ...
};
```

5.4 Hàm thành viên

Định nghĩa cấu trúc & hàm

```
struct Time {
  int hour, min, sec
};
void addHour(Time& t, int h) {
  t.hour += h;
}
...
```

Định nghĩa lớp

```
class Time {
  int hour,min,sec;
public:
  void addHour(int h) {
    hour += h;
  }
  ...
};
```

Gọi hàm với biến cấu trúc

```
Time t;
...
addHour(t,5);
```

Gọi hàm thành viên của ĐT

```
Time t;
...
t.addHour(5);
```

Ở đây có sự khác nhau về cách viết, nhưng chưa có sự khác nhau cơ bản

Khai báo và định nghĩa hàm thành viên

- Thông thường, lớp cùng các hàm thành viên được **khai báo** trong tệp tin đầu (*.h). Ví dụ trong tệp có tên "mytime.h":

```
class Time {
  int hour,min,sec;
public:
  void addHour(int h);
  void addMin(int m);
  void addSec(int s);
  ...
};
```

- Các hàm thường được **định nghĩa** trong tệp tin nguồn (*.cpp):

```
#include "mytime.h"
...
void Time::addHour(int h) {
  hour += h;
}
```

- Có thể định nghĩa một hàm thành viên trong tệp tin đầu dưới dạng một hàm inline (chỉ nên áp dụng với hàm đơn giản), ví dụ:

```
inline void Time::addHour(int h) { hour += h;}
```

- Một hàm thành viên cũng có thể được định nghĩa trong phần khai báo lớp => mặc định trở thành hàm inline, ví dụ

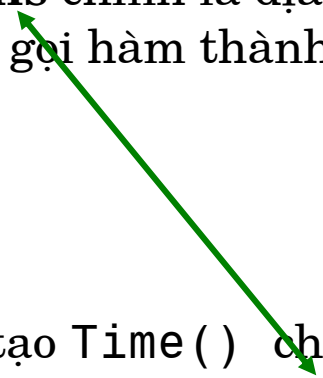
```
class Time {
  int hour,min,sec;
public:
  void addHour(int h) { hour += h; }
};
```

- Khi định nghĩa hàm thành viên, có thể sử dụng các biến thành viên và gọi hàm thành viên khác mà **không cần** (thậm chí không thể được) đưa tên biến đối tượng, ví dụ:

```
void Time::addSec(int s) {
    ...
    addMin(sec/60);
    ...
}
```

Bản chất của hàm thành viên?

```
class Time {
  int hour,min,sec;
public:
  Time() { hour=min=sec=0; }
  void addHour(int h) {
    this->hour += h;          // con trỏ this chính là địa chỉ của
  }                           // đối tượng gọi hàm thành viên
  ...
};

void main() {
  Time t1,t2;          // Tự động gọi hàm tạo Time() cho t1 và t2
  t1.addHour(5);       // Có thể hiểu như là  addHour(&t1,5);
  t2 = t1;             // OK
  t2.addHour(5);       // Có thể hiểu như là  addHour(&t2,5);
  ...
}
```



5.5 Kiểm soát truy nhập

- **public**: Các thành viên công cộng, có thể sử dụng được từ bên ngoài
- **private**: Các thành viên riêng, không thể truy nhập được từ bên ngoài, ngay cả trong lớp dẫn xuất (sẽ đề cập sau)

```
class Time {
private:
   int hour,min,sec;
   ...
};
```

- Mặc định, khi đã khai báo **class** thì các thành viên là **private**.
- **protected**: Các thành viên được bảo vệ, không thể truy nhập được từ bên ngoài, nhưng truy nhập được các lớp dẫn xuất (sẽ đề cập sau)

5.6 Con trỏ đối tượng

```
#include "mytime.h"
void main() {
  Time t;             // call constructor Time()
  t.addHour(5);
  Time *pt = &t;      // pt is identical to this pointer
  pt->addSec(70);
  pt = new Time;      // call constructor Time()
  pt->addMin(25);
  ...
  delete pt;
  pt = new Time[5]; // call constructor 5 times
  for (int i=0; i < 5; ++ i)
    pt[i].addSec(10);
  ...
  delete [] pt;
}
```

Bài tập về nhà

- Dựa trên cấu trúc Vector và các hàm liên quan đã thực hiện trong chương 4, hãy xây dựng lớp đối tượng Vector với các hàm thành viên cần thiết.

Kỹ thuật lập trình

Chương 6: Lớp và đối tượng II

10/6/2005

Nội dung chương 6

6.1 Tạo và hủy đối tượng

6.2 Xây dựng các hàm tạo và hàm hủy

6.3 Nạp chồng toán tử

6.4 Khai báo friend

6.5 Thành viên static (tự đọc)

6.1 Tạo và hủy đối tượng

Có bao nhiêu cách để tạo/hủy đối tượng?

- Tạo/hủy tự động: Định nghĩa một biến thuộc một lớp
  - Bộ nhớ của đối tượng (chứa các dữ liệu biến thành viên) được tự động cấp phát giống như với một biến thông thường
  - Bộ nhớ của đối tượng được giải phóng khi ra khỏi phạm vi định nghĩa

```
class X {
   int a, b;
   ...
};
void f( X x1) {
   if (..) {
     X x2;
     ...
   }
}
X x;
```

Đối tượng được tạo ra trong ngăn xếp

Thời điểm bộ nhớ cho x2 được giải phóng

Thời điểm bộ nhớ cho x1 được giải phóng

Đối tượng được tạo ra trong vùng dữ liệu chương trình

- Tạo/hủy đối tượng động bằng toán tử new và delete:

```
X* pX = 0;
void f(...) {
  if (..) {
    pX = new X;
      ...
    }
}
void g(...) {
  ...
  if (pX != 0) {
    delete pX;
      ...
    }
}
```

Đối tượng được tạo ra trong vùng nhớ tự do

Bộ nhớ của đối tượng trong heap được giải phóng

Vấn đề 1: Khởi tạo trạng thái đối tượng

- Sau khi được tạo ra, trạng thái của đối tượng (bao gồm dữ liệu bên trong và các mối quan hệ) thường là bất định => sử dụng kém an toàn, kém tin cậy, kém thuận tiện

```
X x;              // x.a = ?, x.b = ?
X *px = new X;    // px->a = ?, px->b = ?;
class Vector { int n; double *data; ... };
Vector v;         // v.n = ?, v.data = ?
```

- Làm sao để ngay sau khi được tạo ra, đối tượng có trạng thái ban đầu theo ý muốn của chương trình?

```
X x = {1, 2}; // Error! cannot access private members
```

- Làm sao để tạo một đối tượng là bản sao của một đối tượng có kiểu khác?

```
class Y { int c, d; };
Y y = x;  // Error, X and Y are not the same type,
          // they are not compatible
```

Vấn đề 2: Quản lý tài nguyên

- Đối với các đối tượng sử dụng bộ nhớ động, việc cấp phát và giải phóng bộ nhớ động nên thực hiện như thế nào cho an toàn?

```
class Vector {
  int nelem;
  double *data;
public:
  void create(int n) { data = new double[nelem=n];}
  void destroy()     { delete[] data; nlem = 0; }
  void putElem(int i, double d) { data[i] = d; }
};
Vector v1, v2;
v1.create(5);
// forget to call create for v2
v2.putElem(1,2.5);      // BIG problem!
// forget to call destroy for v1, also a BIG problem
```

- Vấn đề tương tự xảy ra khi sử dụng tệp tin, cổng truyền thông, và các tài nguyên khác trong máy tính

Giải pháp chung: Hàm tạo và hàm hủy

- **Một hàm tạo** luôn được tự động gọi mỗi khi đối tượng được tạo, hàm hủy luôn được gọi mỗi khi đối tượng bị hủy:

```
class X { int a,b;
public:
  X() { a = b = 0; }                    // constructor (1)
  X(int s, int t) { a = s; b = t;} // constructor (2)
  ~X() {}  // destructor
};
void f(X x1) {
  if (..) {
    X x2(1,2);
    X x3(x2);
    ...
  }
}
X *px1 = new X(1,2), *px2 = new X;
delete px1; delete px2;
```

Gọi hàm tạo (1) không tham số (hàm tạo mặc định)

Gọi hàm tạo (2)

Gọi hàm hủy cho x1

Gọi hàm hủy cho x2, x3

Gọi hàm tạo bản sao

Gọi hàm hủy cho *px1 và *px2

6.2 Xây dựng các hàm tạo và hàm hủy

- Hàm tạo là cơ hội để khởi tạo và cấp phát tài nguyên
- Hàm hủy là cơ hội để giải phóng tài nguyên đã cấp phát
- Một lớp có thể có nhiều hàm tạo (khác nhau ở số lượng các tham số hoặc kiểu các tham số)
- Mặc định, compiler tự động sinh ra một hàm tạo không tham số và một hàm tạo bản sao
  - Thông thường, mã thực thi hàm tạo mặc định do compiler sinh ra là rỗng
  - Thông thường, mã thực thi hàm tạo bản sao do compiler sinh ra sao chép dữ liệu của đối tượng theo từng bit
  - Khi xây dựng một lớp, nếu cần có thể bổ sung các hàm tạo mặc định, hàm tạo bản sao và các hàm tạo khác theo ý muốn
- Mỗi lớp có chính xác một hàm hủy, nếu hàm hủy không được định nghĩa thì compiler sẽ tự sinh ra một hàm hủy:
  - Thông thường, mã hàm hủy do compiler tạo ra là rỗng
  - Khi cần có thể định nghĩa hàm hủy để thực thi mã theo ý muốn

Ví dụ: Lớp Time cải tiến

```
class Time {
  int hour, min, sec;
public:
  Time() : hour(0), min(0), sec(0) {}
  Time(int h, int m=0, int s=0) { setTime(h,m,s); }
  Time(const Time& t)
       : hour(t.hour),min(t.min),sec(t.sec) {}
  ...
};

void main() {
  Time t1;            // 0, 0, 0
  Time t2(1,1,1);     // 1, 1, 1
  Time t3(1,1);       // 1, 1, 0
  Time t4(1);         // 1, 0, 0
  Time t5(t1);        // 0, 0, 0
  Time t6=t2;         // 1, 1, 1
  Time* pt1 = new Time(1,1); // 1, 1, 0
  ...
  delete pt1;
}
```

Hàm tạo bản sao và hàm hủy thực ra không cần định nghĩa cho lớp này!

Ví dụ: Lớp Vector cải tiến

- Yêu cầu từ người sử dụng:
  - Khai báo đơn giản như với các kiểu cơ bản
  - An toàn, người sử dụng không phải gọi các hàm cấp phát và giải phóng bộ nhớ
- Ví dụ mã sử dụng:

```
Vector v1;       // v1 has 0 elements
Vector v2(5,0); // v2 has 5 elements init. with 0
Vector v3=v2;    // v3 is a copy of v2
Vector v4(v3);  // the same as above
Vector f(Vector b) {
  double a[] = {1, 2, 3, 4};
  Vector v(4, a);
  ...
  return v;
}
// Do not care about memory management
```

Phiên bản thứ nhất

```
class Vector {
  int nelem;
  double* data;
public:
  Vector() : nelem(0), data(0) {}
  Vector(int n, double d =0.0);
  Vector(int n, double *array);
  Vector(const Vector&);
 ~Vector();
  int size() const { return nelem; }
  double getElem(int i) const    { return data[i];}
  void putElem(int i, double d)  { data[i] = d;   }
private:
  void create(int n) { data = new double[nelem=n];    }
  void destroy()     { if (data != 0) delete [] data; }
};
```

Các hàm thành viên **const** không cho phép thay đổi biến thành viên của đối tượng!

Hàm tạo: cấp phát tài nguyên và khởi tạo
Hàm hủy: dọn dẹp, giải phóng tài nguyên

```
Vector::Vector(int n, double d) {
  create(n);
  while (n-- > 0)
    data[n] = d;
}
Vector::Vector(int n, double* p) {
  create(n);
  while (n-- > 0)
    data[n] = p[n];
}
Vector::~Vector() {
   destroy();
}
```

Trường hợp đặc biệt: Hàm tạo bản sao

- Hàm tạo bản sao được gọi khi sao chép đối tượng:
  - Khi khai báo các biến `x2-x4` như sau:

```
X x1;
X x2(x1);
X x3 = x1;
X x4 = X(x1);
```

  - Khi truyền tham số qua giá trị cho một hàm, hoặc khi một hàm trả về một đối tượng

```
void f(X x) { ... }
X g(..) {
  X x1;
  f(x1);
  ...
  return x1;
}
```

Cú pháp chuẩn cho hàm tạo bản sao?

```
class X {
   int a, b;
public:
   X() : a(0), b(0) {}
   X(X x);          // (1)
   X(const X x);    // (2)
   X(X& x);         // (3)
   X(const X& x); // (4)
   ...
};
void main() {
  X x1;
  X x2(x1);
  ...
}
```

?

(1) Truyền tham số qua giá trị yêu cầu sao chép x1 sang x!!!

(2) Như (1)

(3) Không sao chép tham số, nhưng x có thể bị vô tình thay đổi trong hàm

(4) Không sao chép tham số, an toàn cho bản chính => cú pháp chuẩn!

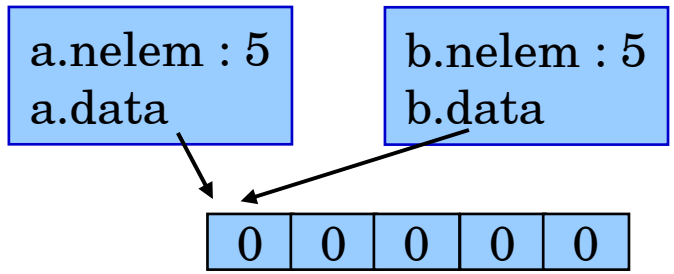
Khi nào cần định nghĩa hàm tạo bản sao?

- Khi nào hàm tạo bản sao mặc định không đáp ứng được yêu cầu.
- Ví dụ, nếu hàm tạo bản sao không được định nghĩa, mã do compiler tự động tạo ra cho lớp Vector sẽ có dạng:

```
Vector::Vector(const Vector& b)
        : nelem(b.nelem), data(b.data) {}
```

- Vấn đề: Sao chép con trỏ thuần túy, hai đối tượng cùng sử dụng chung bộ nhớ phần tử

```
Vector a(5);
Vector b(a);
```



- Trường hợp này, **phải** định nghĩa lại như sau:

```
Vector::Vector(const Vector& a) {
   create(a.nelem);
   for (int i=0; i < nelem; ++i)
     data[i] = a.data[i];
 }
```

Một số điểm cần lưu ý

- Nhiều hàm tạo nhưng chỉ có một hàm hủy => hàm hủy phải nhất quán với tất cả hàm tạo
  - Trong ví dụ lớp Vector, có hàm tạo cấp phát bộ nhớ, nhưng hàm tạo mặc định thì không => hàm hủy cần phân biệt rõ các trường hợp
- Khi nào hàm tạo có cấp phát chiếm dụng tài nguyên thì cũng cần định nghĩa lại hàm hủy
- Trong một lớp mà có định nghĩa hàm hủy thì gần như chắc chắn cũng phải định nghĩa hàm tạo bản sao (nếu như cho phép sao chép)
- Một lớp có thể cấm sao chép bằng cách khai báo hàm tạo bản sao trong phần private, ví dụ:

```
class Y { int a, b; Y(const&);
  ... };
void main() { Y y1;
  Y y2=y1;        // error!
  ... }
```

6.3 Nạp chồng toán tử

- Một trong những kỹ thuật lập trình hay nhất của C++
- Cho phép áp dụng các phép toán với số phức hoặc với vector sử dụng toán tử +, -, *, / tương tự như với các số thực. Ví dụ:

```
class Complex {
  double re, im;
public:
  Complex(double r = 0, double i =0): re(r),im(i) {}
  ...
};
Complex z1(1,1), z2(2,2);
Complex z = z1 + z2;   // ???
```

- Bản chất của vấn đề? Dòng mã cuối cùng thực ra có thể viết:

```
Complex z = z1.operator+(z2);
```

hoặc

```
Complex z = operator+(z1,z2);
```

Hàm toán tử có thể thực hiện là hàm thành viên hoặc hàm phi thành viên

Ví dụ: bổ sung các phép toán số phức

```
class Complex {
  double re, im;
public:
  Complex(double r = 0, double i =0): re(r),im(i) {}
  double real() const { return re; }
  double imag() const { return im; }
  Complex operator+(const Complex& b) const {
    Complex z(re+b.re, im+b.im);
    return z;
  }
  Complex operator-(const Complex& b) const {
    return Complex(re-b.re,im-b.im);
  }
  Complex operator*(const Complex&) const;
  Complex operator/(const Complex&) const;
  Complex& operator +=(const Complex&);
  Complex& operator -=(const Complex&);
  ...
};
```

```
#include “mycomplex.h”
Complex Complex::operator*(const Complex& b) const {
  ...// left for exercise!
}
Complex Complex::operator/(const Complex& b) const {
  ...// left for exercise!
}
Complex& Complex::operator +=(const Complex& b) {
  re += b.re; im += b.im;
  return *this;
}
Complex& operator -=(const Complex&) { ... }

bool operator==(const Complex& a, const Complex& b) {
  return a.real() == b.real() && a.imag() == b.imag();
}
void main() {
  Complex a(1,1), b(1,2);
  Complex c = a+b;
  a = c += b;  // a.operator=(c.operator+=(b));
  if (c == a) { ... }
}
```

return ?

Các toán tử nào có thể nạp chồng?

- Hầu hết các toán tử có trong C++, ví dụ
  - Các toán tử số học: `++ -- + - * / % += -= ...`
  - Các toán tử logic, logic bit: `&& || ! & &= | |= ...`
  - Các toán tử so sánh: `== != > < >= <=`
  - Các toán tử thao tác bit: `<< >> >>= <<=`
  - Các toán tử khác: `[] () -> * , ...`
- Chỉ có 4 toán tử không nạp chồng được:
  - Toán tử truy nhập phạm vi (dấu hai chấm đúp) `::`
  - Toán tử truy nhập thành viên cấu trúc (dấu chấm) `.`
  - Toán tử gọi hàm thành viên qua con trỏ `*->`
  - Toán tử điều kiện `? :`

Một số qui định

- Có thể thay đổi ngữ nghĩa của một toán tử cho các kiểu mới, nhưng không thay đổi được cú pháp (ví dụ số ngôi, trình tự ưu tiên thực hiện,...)
- Trong một phép toán định nghĩa lại, phải có ít nhất một toán hạng có kiểu mới (struct, union hoặc class) => không định nghĩa lại cho các kiểu dữ liệu cơ bản và kiểu dẫn xuất trực tiếp được!
  - Ví dụ không thể định nghĩa lại toán tử ^ là phép tính lũy thừa cho các kiểu số học cơ bản (int, float, double,...)
- Chỉ nạp chồng được các toán tử có sẵn, không đưa thêm được các toán tử mới
  - Ví dụ không thể bổ sung ký hiệu toán tử ** cho phép toán lũy thừa
- Nạp chồng toán tử thực chất là nạp chồng tên hàm => cần lưu ý các qui định về nạp chồng tên hàm
- Đa số hàm toán tử **có thể** nạp chồng **hoặc** dưới dạng hàm thành viên, **hoặc** dưới dạng hàm phi thành viên
- Một số toán tử **chỉ có thể** nạp chồng bằng hàm thành viên
- Một số toán tử **chỉ nên** nạp chồng bằng hàm phi thành viên

Nạp chồng toán tử []

- Yêu cầu: truy nhập các phần tử của một đối tượng thuộc lớp Vector với toán tử [] giống như đối với một mảng

```
Vector v(5,1.0);
double d = v[0]; // double d = v.operator[](0);
v[1] = d + 2.0;  // v.operator[](1) = d + 2.0;
const Vector vc(5,1.0);
d = vc[1];       // d = operator[](1);
```

- Giải pháp

```
class Vector {
  ...
public:
  double  operator[](int i) const { return data[i]; }
  double& operator[](int i)       { return data[i]; }
  ...
};
```

Nạp chồng toán tử gán (=)

- Giống như hàm tạo bản sao, hàm toán tử gán được compiler tự động bổ sung vào mỗi lớp đối tượng => mã hàm thực hiện gán từng bit dữ liệu
- Cú pháp chuẩn của hàm toán tử gán cho một lớp X tương tự cú pháp các phép tính và gán:

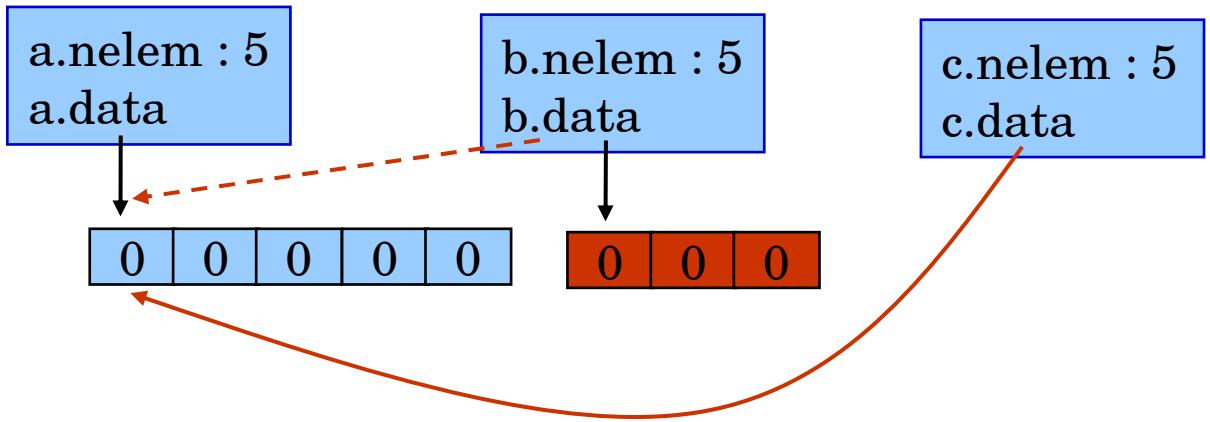
```
X& operator=(const X&);
```

- Khi nào cần định nghĩa lại hàm tạo bản sao thì cũng cần (và cũng mới nên) định nghĩa lại hàm toán tử gán
- Ví dụ, nếu hàm toán tử gán không được định nghĩa, mã do compiler tự động tạo ra cho lớp Vector sẽ có dạng:

```
Vector& Vector::operator=(const Vector& b) {
  nelem = b.nelem;
  data  = b.data
  return *this;
 }
```

- Vấn đề tương tự như hàm tạo bản sao mặc định, thậm chí còn tồi tệ hơn

```
...
{
  Vector a(5), b(3), c;
  b = a;
  c = a;
} // calling destructor for a, b and c causes
  // 3 times calling of delete[] operator for the
  // same memory space
```



Nạp chồng toán tử gán cho lớp Vector

```
Vector& Vector::operator=(const Vector& b) {
  if (nelem != b.nelem) {
    destroy();
    create(b.nelem);
  }
  for (int i=0; i < nelem; ++i)
    data[i] = b.data[i];
  return *this;
}
```

6.4 Khai báo friend

- Vấn đề: Một số hàm phi thành viên thực hiện bên ngoài, hoặc hàm thành viên của một lớp khác không truy nhập được trực tiếp vào biến riêng của một đối tượng => thực thi kém hiệu quả
- Giải pháp: Cho phép một lớp khai báo friend, có thể là một hàm phi thành viên, một hàm thành viên của một lớp khác, hoặc cả một lớp khác
- Ví dụ

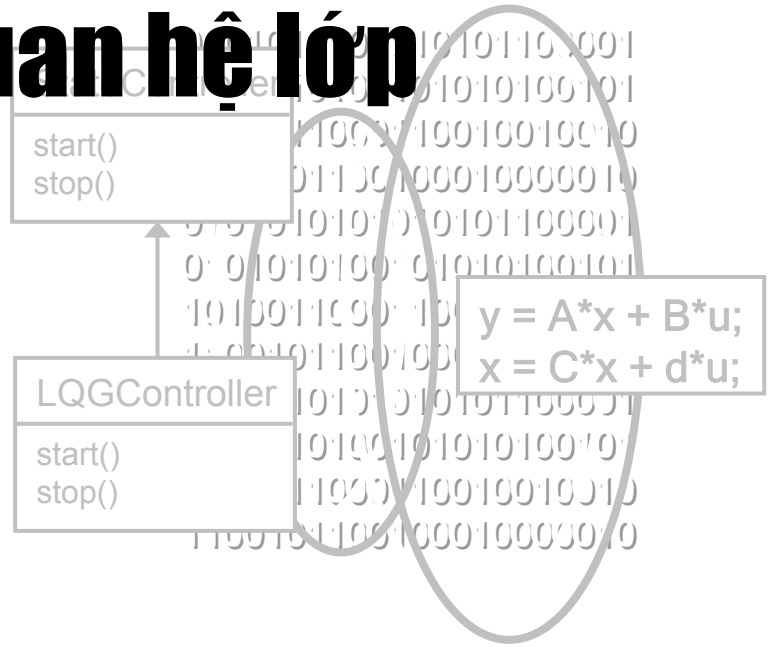
```
class Complex { ...
friend bool operator==(const Complex&,const Complex&);
friend class ComplexVector;
friend ComplexVector Matrix::eigenvalues();
...
}
bool operator==(const Complex& a, const Complex& b) {
    return a.re == b.re && a.im == b.im;
}
```

Bài tập về nhà

- Hoàn chỉnh lớp Vector với những phép toán cộng, trừ, nhân/chia với số vô hướng, nhân vô hướng và so sánh bằng nhau
- Dựa trên cấu trúc List và các hàm liên quan đã thực hiện trong chương 4, hãy xây dựng lớp đối tượng List với các hàm thành viên cần thiết.

Kỹ thuật lập trình

Chương 7: Quan hệ lớp



10/12/2005

Nội dung chương 7

7.1 Quan hệ lớp

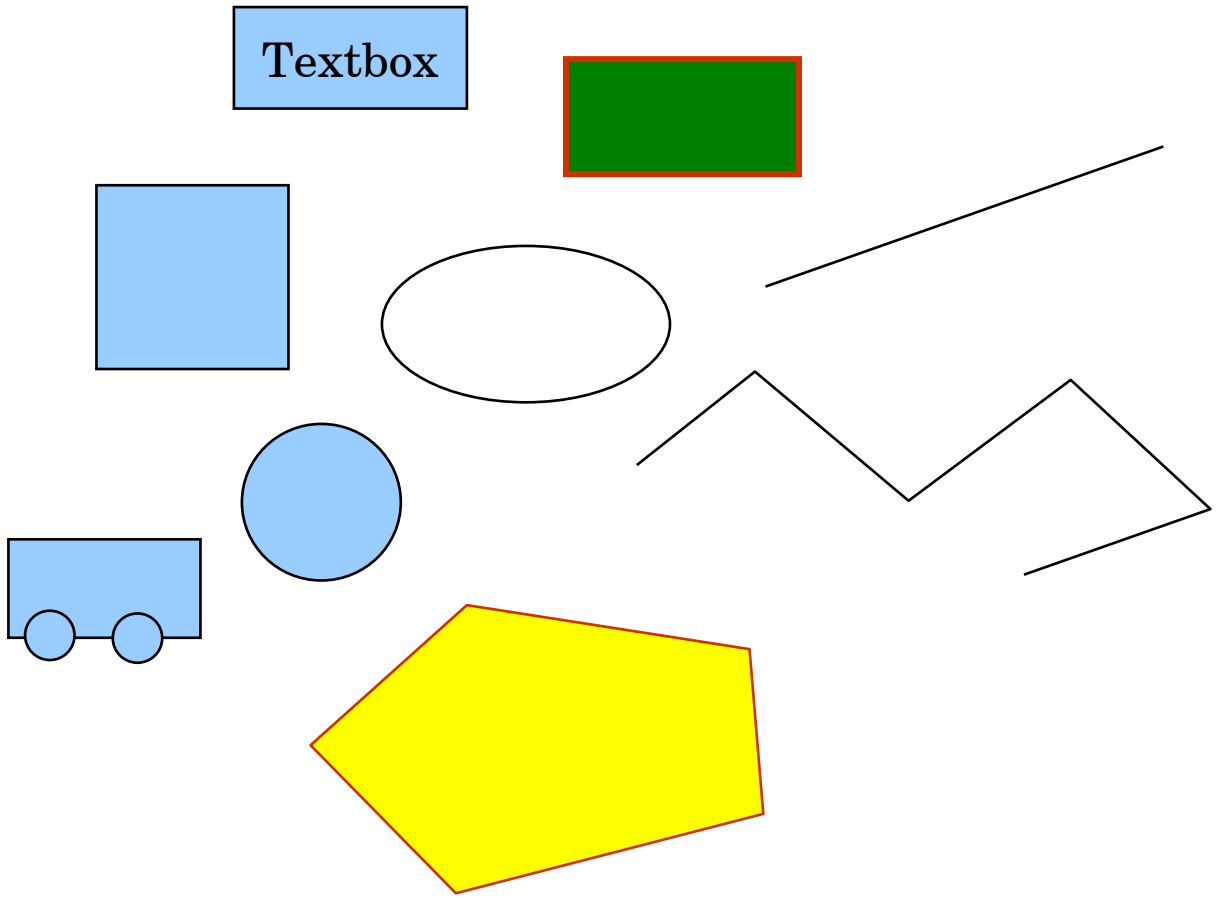
7.2 Dẫn xuất và thừa kế

7.3 Hàm ảo và nguyên lý đa hình/đa xạ

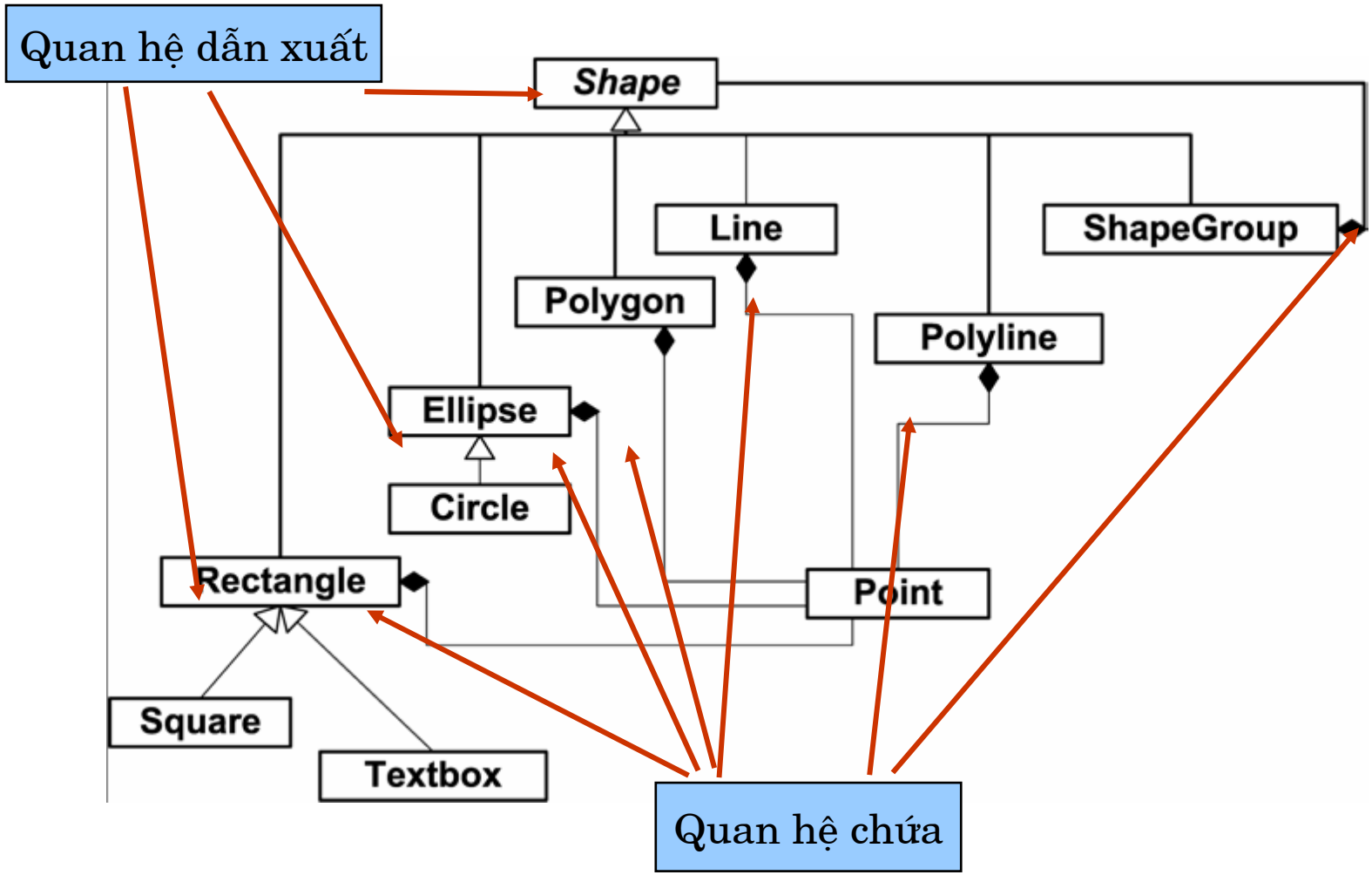
7.4 Ví dụ thư viện khối chức năng

7.1 Phân loại quan hệ lớp

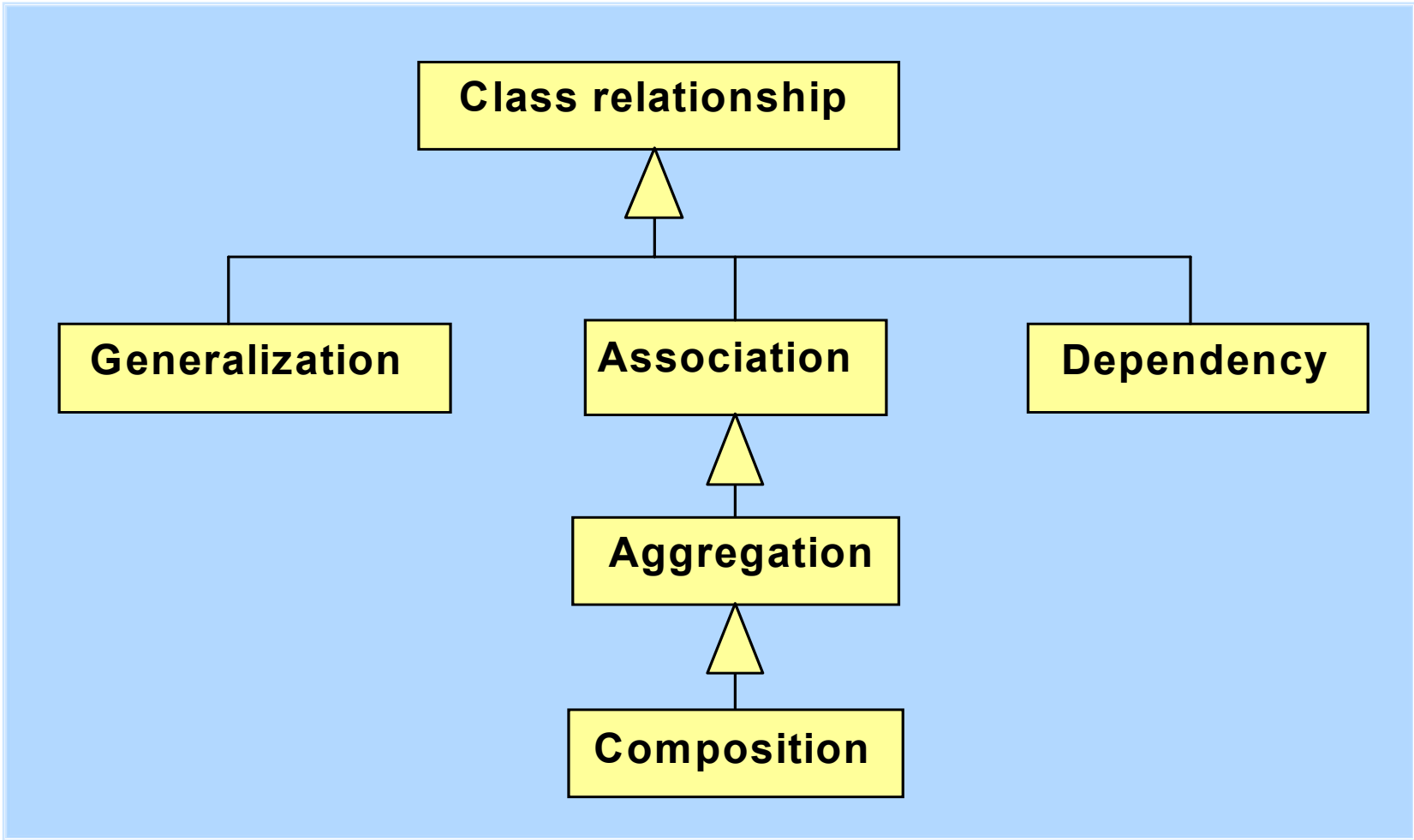
- Ví dụ minh họa: Các lớp biểu diễn các hình vẽ trong một chương trình đồ họa
  - Rectangle
  - Square
  - Ellipse
  - Circle
  - Line
  - Polygon
  - Polyline
  - Textbox
  - Group



Biểu đồ lớp (Unified Modeling Language)

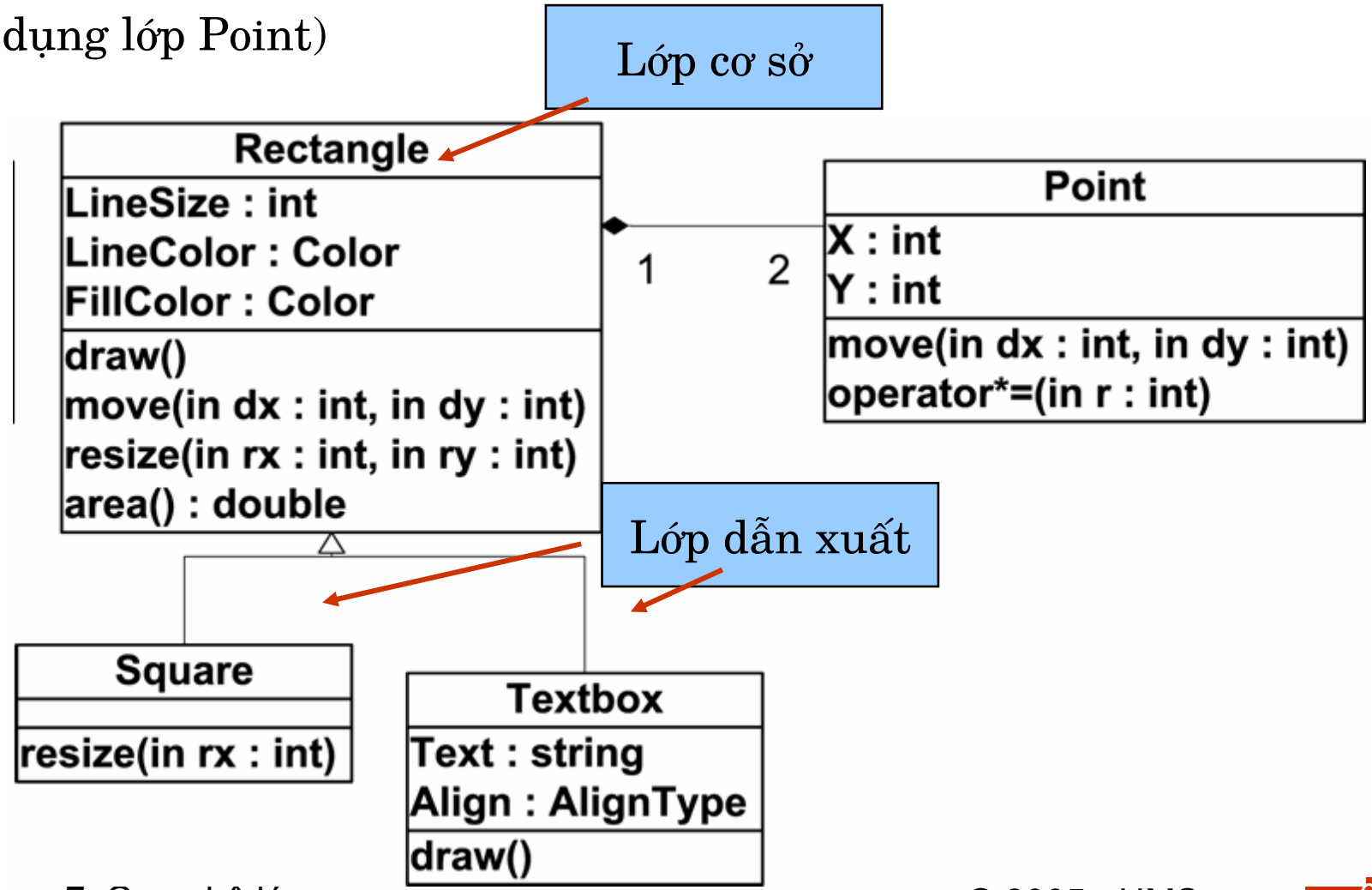


Các dạng quan hệ lớp (meta model)



7.2 Dẫn xuất và thừa kế

- Ví dụ xây dựng các lớp: Rectangle, Square và Textbox (sử dụng lớp Point)



Thực hiện trong C++: Lớp Point

```
class Point
{
   int X,Y;
public:
   Point() : X(0), Y(0) {}
   Point(int x, int y): X(x), Y(y) {}
   int x() const { return X; }
   int y() const { return Y; }
   void move(int dx, int dy) {
        X += dx;
        Y += dy;
   }
   void operator*=(int r) {
        X *= r;
        Y *= r;
   }
};
Point operator-(const Point& P1, const Point& P2) {
   return Point(P2.x()-P1.x(),P2.y()-P1.y());
}
```

Thực hiện trong C++: Lớp Rectangle

```
#include <iostream>
#include <string>
#include "Point.h"
typedef int Color;
class Rectangle
{
   Point TL, BR;
   Color LineColor, FillColor;
   int   LineSize;
public:
   Point getTL() const { return TL; }
   Point getBR() const { return BR; }
   void  setTL(const Point& tl) { TL = tl; }
   void  setBR(const Point& br) { BR = br; }
   Color getLineColor() const  { return LineColor; }
   void  setLineColor(Color c) { LineColor = c; }
   int   getLineSize() const   { return LineSize; }
   void  setLineSize(int s)    { LineSize = s; }
```

```
    Rectangle(int x1=0, int x2=10, int y1=10, int y2=10)
        : TL(x1,y1), BR(x2,y2), LineColor(256),FillColor(0) {}

                    Point& tl, const Point& br, Color lc, Color fc)
            : TL(tl), BR(br), LineColor(lc), FillColor(fc) {}
    void draw() {
      std::cout << "\nRectangle:\t[" << TL << BR << ']';
    }
    void move(int dx, int dy) {
      TL.move(dx,dy);
      BR.move(dx,dy);
      draw();
    }
    void resize(int rx, int ry) {
      TL *= rx;
      BR *= ry;
      draw();
    }
    double area() const {
      Point d = BR - TL;
      int a = d.x()*d.y();
      return a > 0 ? a : - a;
    }
};
```

Thực hiện trong C++: Lớp Square

```
#include "Rectangle.h"
class Square : public Rectangle
{
public:
   Square(int x1=1, int y1=0, int a=10)
        : Rectangle(x1,y1,x1+a,y1+a) {}

   void resize(int r) {
        Rectangle::resize(r,r);
   }
};
```

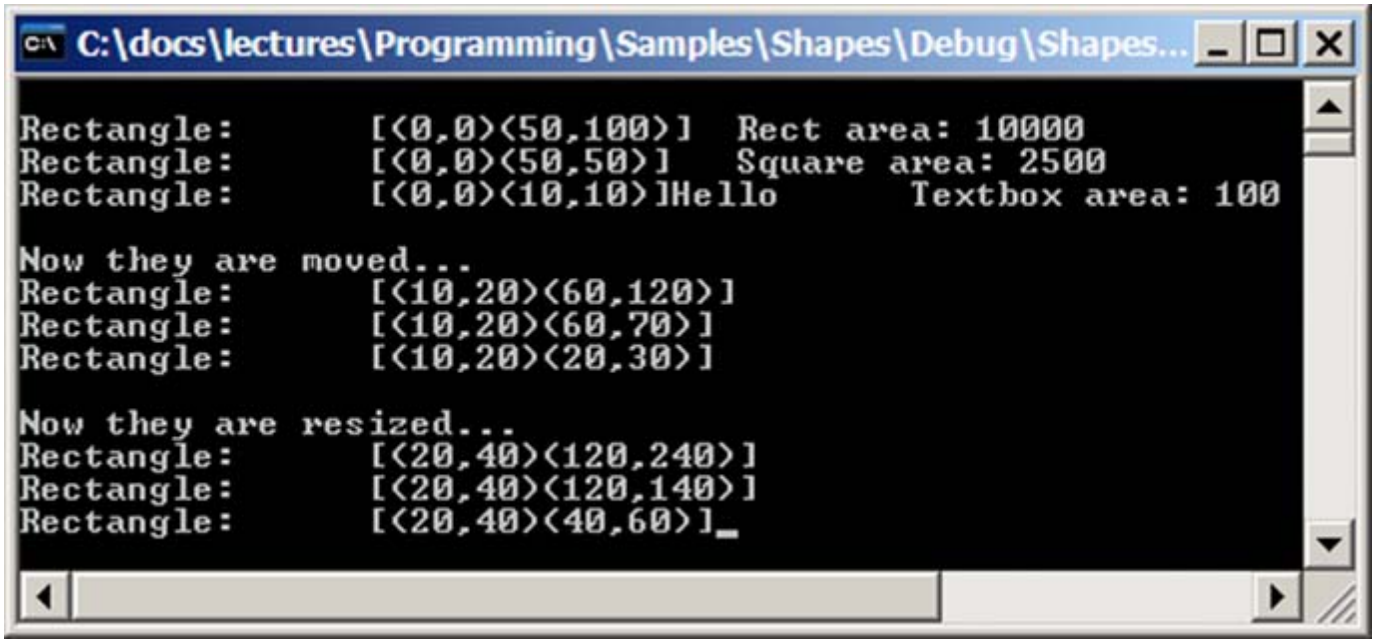
Thực hiện trong C++: Lớp Textbox

```
#include "Rectangle.h"
enum AlignType { Left, Right, Center};
class TextBox : public Rectangle
{
   std::string Text;
   AlignType   Align;
public:
   TextBox(const string& text = "Text")
        : Text(text), Align (Left) {}
   TextBox(const Point& tl, const Point& br, Color lc, Color fc,
           const string& text):
        Rectangle(tl,br,lc,fc), Text(text), Align(Left) {}

   void draw() {
        Rectangle::draw();
        std::cout << Text << '\n';
   }
};
```

Chương trình minh họa

```
#include "Rectangle.h"
#include  "Square.h"
#include "TextBox.h"
#include <conio.h>
void main()
{
   Rectangle rect(0,0,50,100);
   Square        square(0,0,50);
   TextBox       text("Hello");


   std::cout << "\t Rect area: " << rect.area();
   square.draw();
   std::cout << "\t Square area: " << square.area();
   text.draw();
   std::cout << "\t Textbox area: " << text.area();
```

```
    getch();
    std::cout << "\n\nNow they are moved...";
    rect.move(10,20);
    square.move(10,20);
    text.move(10,20);
    getch();
    std::cout << "\n\nNow they are resized...";
    rect.resize(2,2);
    square.resize(2);
    text.resize(2,2);
    getch();
}
```



Truy nhập thành viên

- Các hàm thành viên của lớp dẫn xuất có thể truy nhập thành viên "protected" định nghĩa ở lớp cơ sở, nhưng cũng không thể truy nhập các thành viên "private" định nghĩa ở lớp cơ sở

  Phản ví dụ:

```
Rectangle rect(0,0,50,100);
Square    square(0,0,50);
square.TL = 10;
```

- Lớp dẫn xuất được "thừa kế" cấu trúc dữ liệu và các phép toán đã được định nghĩa trong lớp cơ sở, nhưng không nhất thiết có quyền sử dụng trực tiếp, mà phải qua các phép toán (các hàm công cộng hoặc hàm public)
- Quyền truy nhập của các thành viên "public" và "protected" ở lớp dẫn xuất được giữ nguyên trong lớp cơ sở

7.3 Hàm ảo và cơ chế đa hình/đa xạ

- Trong quá trình liên kết, lời gọi các hàm và hàm thành viên thông thường được chuyển thành các lệnh nhảy tới địa chỉ cụ thể của mã thực hiện hàm => **"liên kết tĩnh"**
- Vấn đề thực tế:
  - Các đối tượng đa dạng, mặc dù giao diện giống nhau (phép toán giống nhau), nhưng cách thực hiện khác nhau => thực thi như thế nào?
  - Một chương trình ứng dụng chứa nhiều kiểu đối tượng (đối tượng thuộc các lớp khác nhau, có thể có cùng kiểu cơ sở) => quản lý các đối tượng như thế nào, trong một danh sách hay nhiều danh sách khác nhau?

Vấn đề của cơ chế "liên kết tĩnh"

- Xem lại chương trình trước, hàm Rectangle::draw đều in ra tên "Rectangle" => chưa hợp lý nên cần được định nghĩa lại ở các lớp dẫn xuất

```
    std::cout << "\nSquare:\t[" << getTL() << getBR() << ']';
 }
void TextBox::draw() {
    std::cout << "\nTextbox:\t[" << getTL() << getBR() << ' '
              << Text << ']';
}
```

Chương trình minh họa 1

```
void main()
{
        Rectangle rect(0,0,50,100);
        Square    square(0,0,50);
        TextBox   text("Hello");

        rect.draw();
        square.draw();
        text.draw();

        getch(); std::cout << "\n\nNow they are moved...";
        rect.move(10,20);
        square.move(10,20);
        text.move(10,20);

        getch(); std::cout << "\n\nNow they are resized...";
        rect.resize(2,2);
        square.resize(2);
        text.resize(2,2);
        getch();
}
```

Kết quả: Như ý muốn?

```
Rectangle:      [(0,0)(50,100)]
Square:         [(0,0)(50,50)]
Textbox:        [(0,0)(10,10) Hello]

Now they are moved...
Rectangle:      [(10,20)(60,120)]
Rectangle:      [(10,20)(60,70)]
Rectangle:      [(10,20)(20,30)]

Now they are resized...
Rectangle:      [(20,40)(120,240)]
Rectangle:      [(20,40)(120,140)]
Rectangle:      [(20,40)(40,60)]
```

Gọi hàm draw() của Rectangle!

Chương trình minh họa 2

```
void main()
{
        const N =3;
        Rectangle rect(0,0,50,100);
        Square    square(0,0,50);
        TextBox   text("Hello");
        Rectangle* shapes[N] = {&rect, &square, &text};

        for (int i = 0; i < N; ++i)
                shapes[i]->draw();
        getch();
}
```

Quản lý các đối tượng chung trong một danh sách nhờ cơ chế dẫn xuất!

Kết quả: các hàm thành viên của lớp dẫn xuất cũng không được gọi

```
Rectangle:      [(0,0)(50,100)]
Rectangle:      [(0,0)(50,50)]
Rectangle:      [(0,0)(10,10)]
```

Giải pháp: Hàm ảo

```
class Rectangle {
...
public:
  ...
  virtual void draw();
}
```

Kết quả: Như mong muốn!

Chương trình 1

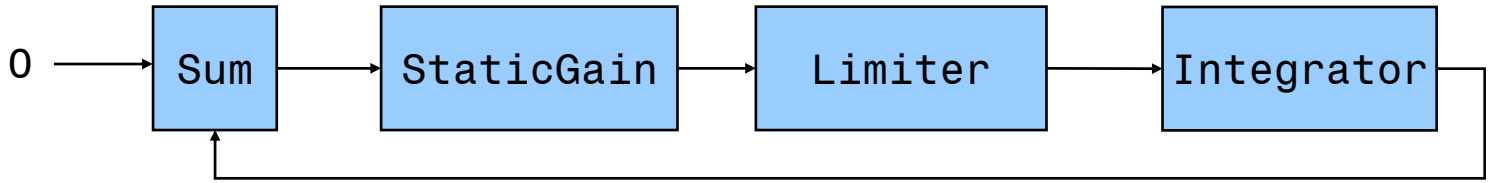
```
Rectangle:      [(0,0)(50,100)]
Square:         [(0,0)(50,50)]
Textbox:        [(0,0)(10,10) Hello]

Now they are moved...
Rectangle:      [(10,20)(60,120)]
Square:         [(10,20)(60,70)]
Textbox:        [(10,20)(20,30) Hello]

Now they are resized...
Rectangle:      [(20,40)(120,240)]
Square:         [(20,40)(120,140)]
Textbox:        [(20,40)(40,60) Hello]
```

Chương trình 2

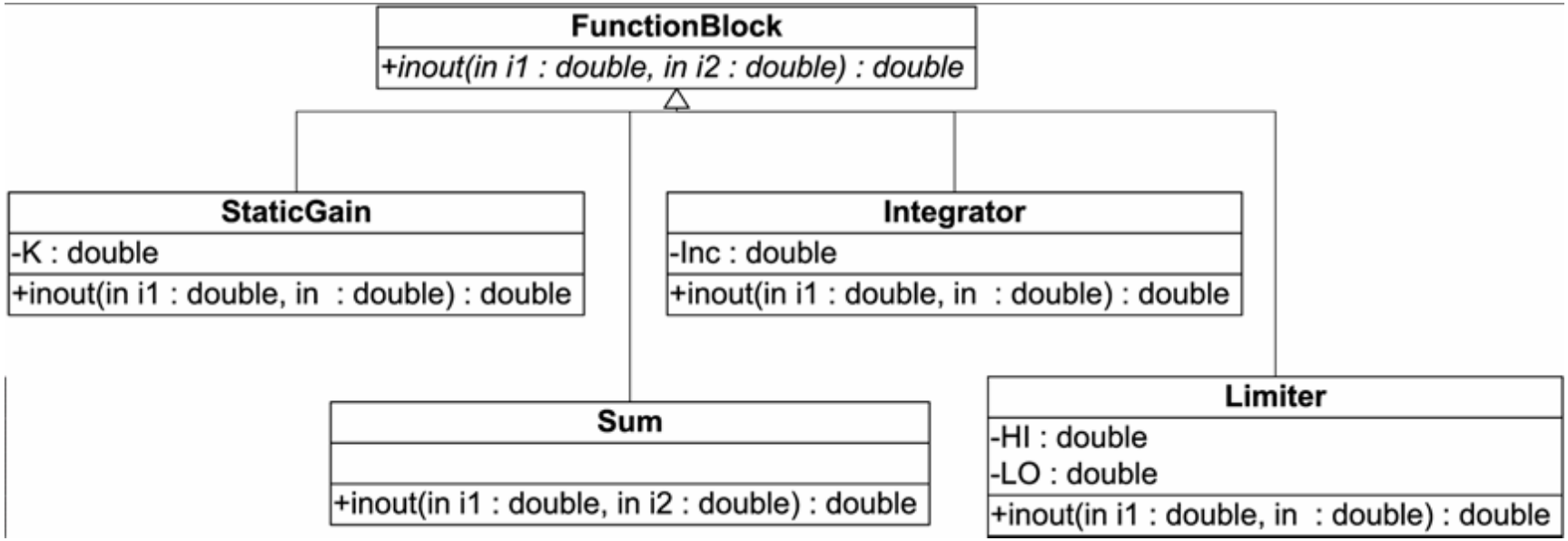
```
Rectangle:      [(0,0)(50,100)]
Square:         [(0,0)(50,50)]
Textbox:        [(0,0)(10,10) Hello]
```

7.4 Ví dụ thư viện khối chức năng

- Bài toán:
  - Xây dựng một thư viện các khối chức năng phục vụ tính toán và mô phỏng tương tự trong SIMULINK
  - Viết chương trình minh họa sử dụng đơn giản
- Ví dụ một sơ đồ khối



Biểu đồ lớp



Kỹ thuật lập trình

Chương 8: Tiến tới tư duy lập trình hướng đối tượng

10/20/2005

Nội dung chương 8

8.1 Đặt vấn đề

8.2 Giới thiệu ví dụ chương trình mô phỏng

8.3 Tư duy "rất" cổ điển

8.4 Tư duy hướng hàm

8.5 Tư duy dựa trên đối tượng (object-based)

8.6 Tư duy thực sự hướng đối tượng

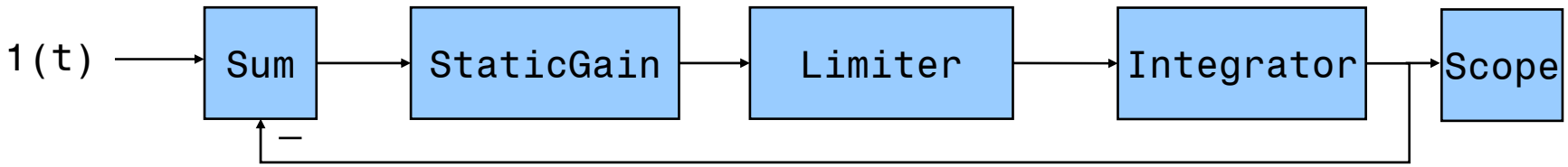
8.1 Đặt vấn đề

„Designing object-oriented software is hard, and designing reusable object-oriented software is even harder...It takes a long time for novices to learn what object-oriented design is all about. Exprienced designers evidently know something inexperienced ones don't...

One thing expert designers know *not* to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented design more flexible, elegant, and ultimately reusable...“

Erich Gamma et. al.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

8.2 Phần mềm mô phỏng kiểu FBD



Nhiệm vụ:

Xây dựng phần mềm để hỗ trợ mô phỏng thời gian thực một cách linh hoạt, mềm dẻo, đáp ứng được các yêu cầu của từng bài toán cụ thể

Trước mắt chưa cần hỗ trợ tạo ứng dụng kiểu kéo thả bằng công cụ đồ họa

8.3 Tư duy rất cổ điển

```
// SimProg1.cpp
#include <iostream.h>
#include <conio.h>
#include <windows.h>
void main() {
    double K =1,I=0, Ti = 5;
    double Hi = 10, Lo = -10;
    double Ts = 0.5;
    double r =1, y=0, e, u, ub;
    cout << "u\ty";
    while (!kbhit()) {
        e = r-y;                    // Sum block
        u = K*e;                    // Static Gain
        ub = max(min(u,Hi),Lo); // Limiter
        I += ub*Ts/Ti;              // Integrator state
        y  = I;                     // Integrator output
        cout << '\n' << u << '\t' << y;
        cout.flush();
        Sleep(long(Ts*1000));
    }
}
```

Vấn đề?

- Phần mềm dưới dạng chương trình, không có giá trị sử dụng lại
- Rất khó thay đổi hoặc mở rộng theo yêu cầu cụ thể của từng bài toán
- Toàn bộ thuật toán được gói trong một chương trình => khó theo dõi, dễ gây lỗi, không bảo vệ được chất xám

8.4 Tư duy hướng hàm

```
// SimProg2.cpp
#include <iostream.h>
#include <conio.h>
#include <windows.h>
#include "SimFun.h"
void main() {
   double K  = 5.0, double Ti = 5.0;
   double Hi = 10, Lo = -10;
   double Ts = 0.5;
   double r =1, y=0, e, u, ub;
   cout << "u\ty";
   while (!kbhit()) {
     e = sum(r,-y);       // Sum block
     u = gain(K,e);       // Static Gain
     ub= limit(Hi,Lo,u);     // Limiter
     y = integrate(Ti,Ts,ub);    // Integrator output
     cout << '\n' << u << '\t' << y;
     cout.flush();
     Sleep(long(Ts*1000));
   }
}
```

```
// SimFun.h
inline double sum(double x1, double x2) { return x1 + x2; }
inline double gain(double K, double x)  { return K * x; }
double limit(double Hi, double Lo, double x);
double integrate(double Ti, double Ts, double x);
```

```
// SimFun.cpp
double limit(double Hi, double Lo, double x) {
        if (x > Hi) x = Hi;
        if (x < Lo) x = Lo;
        return x;
}

double integrate(double Ti, double Ts, double x) {
        static double I = 0;
        I += x*Ts/Ti;
        return I;
}
```

Vấn đề?

- Vẫn chưa đủ tính linh hoạt, mềm dẻo cần thiết
- Thay đổi, mở rộng chương trình mô phỏng rất khó khăn
- Các khâu có trạng thái như khâu tích phân, khâu trễ khó thực hiện một cách "sạch sẽ" (trạng thái lưu trữ dưới dạng nào?)
- Rất khó phát triển thành phần mềm có hỗ trợ đồ họa kiểu kéo thả

8.5 Tư duy dựa đối tượng

```
// SimClass.h
class Sum {
public:
    double operator()(double x1, double x2) {
        return x1 + x2;
    }
};
class Gain {
    double K;
public:
    Gain(double k = 1) : K(k) {}
    double operator()(double x){ return K * x; }
};
class Limiter {
    double Hi, Lo;
public:
    Limiter(double h=10.0, double l= -10.0);
    double operator()(double x);
};
```

```
class Integrator {
   double Ki, Ts;
   double I;
public:
   Integrator(double ti = 1.0, double ts = 0.5);
   double operator()(double x);
};

class Delay {
   double* bufPtr;
   int      bufSize;
   double  Td, Ts;
public:
   Delay(double td = 0, double ts = 1);
   Delay(const Delay&);
   Delay& operator=(Delay&);
   ~Delay();
   double operator()(double x);
private:
   void createBuffer(int sz);
};
```

```
#include <math.h>
#include "SimClass.h"

Limiter::Limiter(double h, double l) : Hi(h), Lo(l) {
   if (Hi < Lo) Hi = Lo;
}
double Limiter::operator()(double x) {
   if (x > Hi) x = Hi;
   if (x < Lo) x = Lo;
   return x;
}
Integrator::Integrator(double ti, double ts)
   : Ts(1), Ki(1), I(0) {
   if (ts > 0)
      Ts = ts;
   if (ti > 0)
      Ki = ts/ti;
}
double Integrator::operator()(double x) {
    I += x*Ki;
    return I;
}
```

```
Delay::Delay(double td, double ts) : Td(td), Ts(ts) {
    if (Td < 0) Td = 0;
       if (Ts < 0) Ts = 1;
       createBuffer((int)ceil(Td/Ts));
}

double Delay::operator()(double x) {
       if (bufSize > 0) {
              double y = bufPtr[0];
              for (int i=0; i < bufSize-1; ++i)
                     bufPtr[i] = bufPtr[i+1];
              bufPtr[bufSize-1] = x;
              return y;
       }
       return x;
}
void Delay::createBuffer(int sz) {
       bufSize = sz;
       bufPtr =  new double[bufSize];
       for (int i=0; i < bufSize; ++i)
              bufPtr[i] = 0.0;
}
...
```

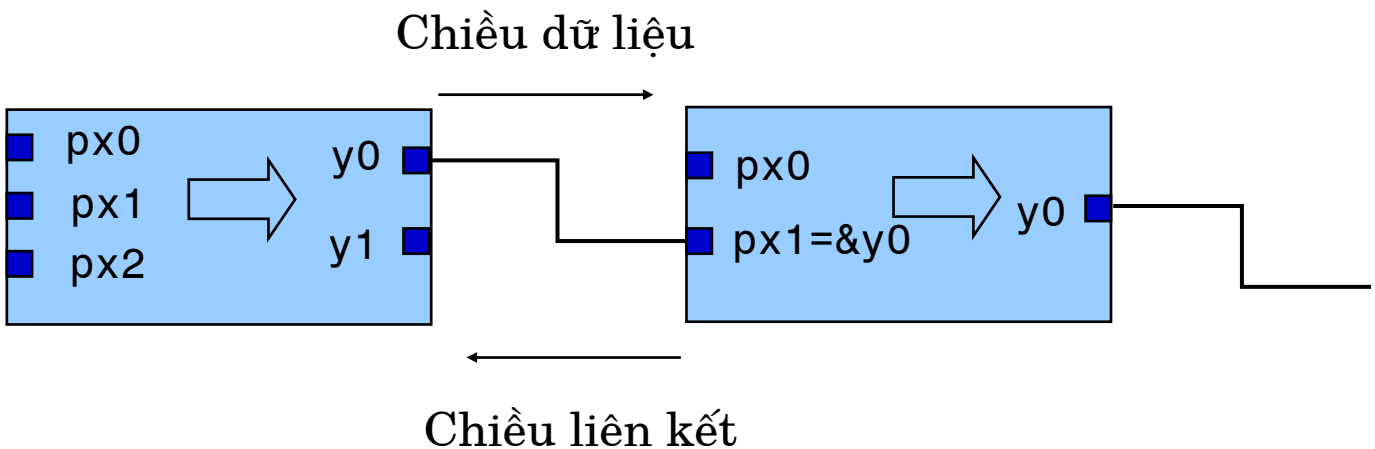
```
// SimProg3.cpp
#include <iostream.h>
#include <conio.h>
#include <windows.h>
#include "SimClass.h"

void main() {
    double Ts = 0.5;
    Sum sum;
    Gain gain(2.0);
    Limiter limit(10,-10);
    Integrator integrate(5,Ts);
    Delay delay(1.0);
    double r =1, y=0, e, u, ub;
    cout << "u\ty";
    while (!kbhit()) {
        e = sum(r,-y);     // Sum block
        u = gain(e);       // Static Gain
        ub= limit(u);      // Limiter
        y = integrate(ub);// Integrator output
        y = delay(y);
        cout << '\n' << u << '\t' << y;
        cout.flush();
        Sleep(long(Ts*1000));
    }
}
```

Vấn đề?

- Khi số lượng các khối lớn lên thì quản lý thế nào?
- Khi quan hệ giữa các khối phức tạp hơn (nhiều vào, nhiều ra) thì tổ chức quan hệ giữa các đối tượng như thế nào?
- Làm thế nào để tạo và quản lý các đối tượng một cách động (trong lúc chương trình đang chạy)?
- Lập trình dựa đối tượng mới mang lại ưu điểm về mặt an toàn, tin cậy, nhưng chưa mang lại ưu điểm về tính linh hoạt cần thiết của phần mềm => giá trị sử dụng lại chưa cao.

8.6 Tư duy hướng đối tượng

```
class FB {
public:
   virtual void execute() = 0;
private:
   virtual double* getOutputPort(int i=0) = 0;
   virtual void setInputPort(double* pFromOutputPort,
                             int i=0)= 0;
   friend class FBD;
};
```



```
class Sum : public FB {
public:
     Sum(bool plus_sign1 = true, bool plus_sign2 = false);
     void execute();
private:
     bool   sign[2];
     double *px[2];
     double y;
     double* getOutputPort(int i=0);
     void setInputPort(double* pFromOutputPort, int i=0);
};
Sum::Sum(bool plus_sign1, bool plus_sign2): y(0) {
     px[0] = px[1] = 0;
     sign[0] = plus_sign1;
     sign[1] = plus_sign2;
}
void Sum::execute() {
     if (px[0] != 0) y = sign[0] ? *(px[0]) : - *(px[0]);
     if (px[1] != 0) y += sign[1] ? *(px[1]) : - *(px[1]);
}
double* Sum::getOutputPort(int) {
     return &y;
}
void Sum::setInputPort(double* pFromOutputPort, int i) {
     if(i < 2)
          px[i] = pFromOutputPort;
}
```

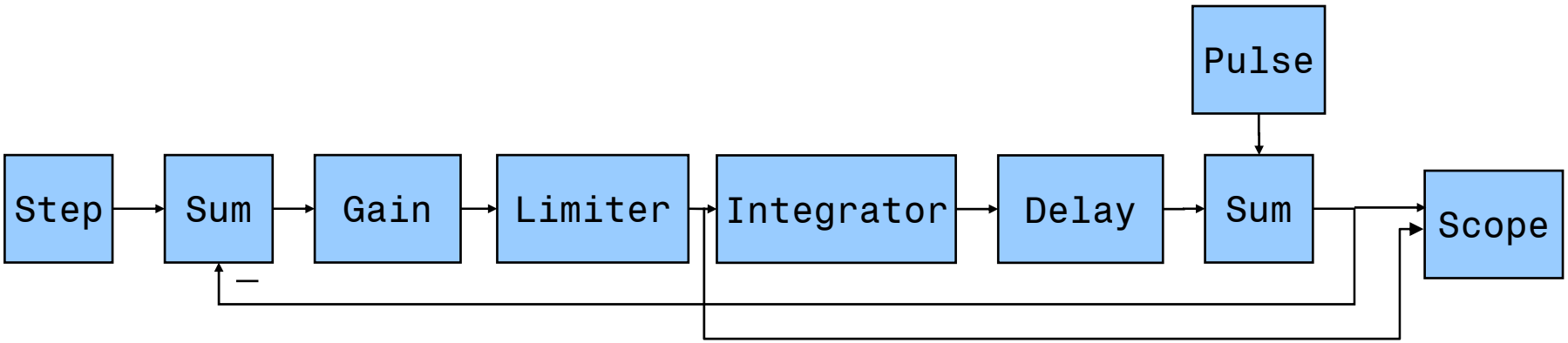
```
class Limiter: public FB {
public:
   Limiter(double h=10.0, double l = -10.0);
   void execute();
private:
   double Hi, Lo;
   double *px;
   double  y;
   double* getOutputPort(int i=0);
   void setInputPort(double* pFromOutputPort, int i=0);
};
Limiter::Limiter(double h, double l) : Hi(h), Lo(l), y(0), px(0) {
if (Hi < Lo)   Hi = Lo; }
void Limiter::execute() {
   if (px != 0) {
      y = *px;
      if (y > Hi) y = Hi;
      if (y < Lo) y = Lo;
   }
}
double* Limiter::getOutputPort(int) {
   return &y;
}
void Limiter::setInputPort(double* pFromOutputPort, int i) {
   px = pFromOutputPort;
}
```

```
#include <vector>
#include <windows.h>
class FBD : public std::vector<FB*> {
   double Ts;
   bool stopped;
public:
   FBD(double ts = 0.5): Ts (ts > 0? ts : 1), stopped(true) {}
   void addFB(FB* p) { push_back(p); }
   void connect(int i1, int i2, int oport=0, int iport = 0) {
      FB *fb1= at(i1), *fb2= at(i2);
      fb2->setInputPort(fb1->getOutputPort(oport),iport);
   }
   void start();
   ~FBD();
};
FBD::~FBD() {
   for (int i=0; i < size(); ++i)
      delete at(i);
}
void FBD::start() {
   while(!kbhit()) {
      for (int i=0; i < size(); ++i)
         at(i)->execute();
      Sleep(long(Ts*1000));
   }
}
```

```
#include <iostream>
#include "SimFB.h"
void main() {
   double Ts=0.5;
   FBD fbd(0.5);
   fbd.addFB(new Step(1.0));       // 0
   fbd.addFB(new Sum);             // 1
   fbd.addFB(new Gain(5.0));       // 2
   fbd.addFB(new Limiter(10,-10));   // 3
   fbd.addFB(new Integrator(5,Ts));  // 4
   fbd.addFB(new Delay(0.0, Ts));    // 5
   fbd.addFB(new Scope(std::cout));  // 6

   for(int i=0; i < fbd.size()-1; ++i)
      fbd.connect(i,i+1);
   fbd.connect(5,1,0,1);
   fbd.connect(3,6,0,1);

   std::cout << "y\tu";
   fbd.start();

}
```

Bài tập về nhà

- Luyện tập lại trên máy tính các ví dụ từ phần 8.3 – 8.5
- Dựa trên các ví dụ lớp đã xây dựng ở phần 8.6 (Limiter, Sum), bổ sung các lớp còn lại (Step, Scope, Gain, Integrator, Delay)
- Chạy thử lại chương trình ở phần 8.6 sau khi đã hoàn thiện các lớp cần thiết.
- Bổ sung lớp Pulse để mô phỏng tác động của nhiễu quá trình (dạng xung vuông biên độ nhỏ, chu kỳ đặt được). Mở rộng chương trình mô phỏng như minh họa trên hình vẽ.

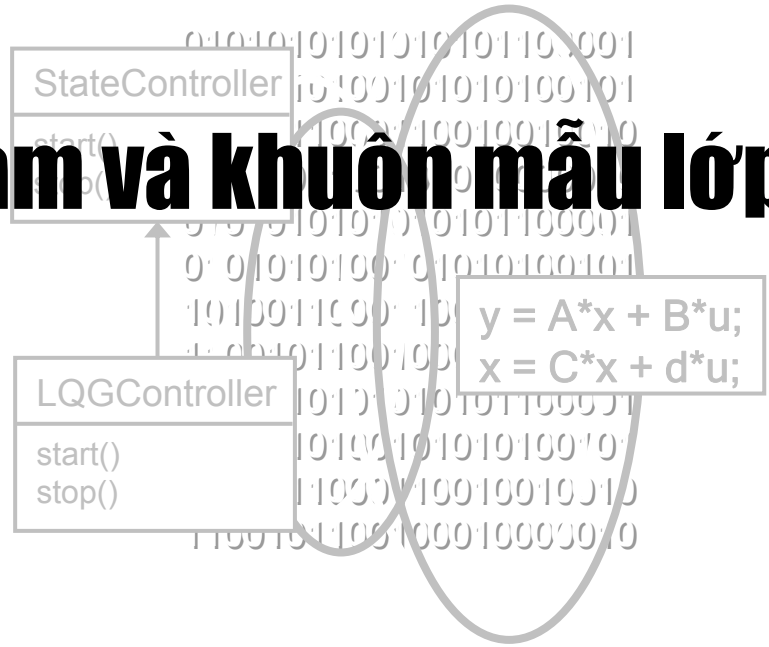


Kỹ thuật lập trình

Phần III: Lập trình tổng quát

Chương 9:

Khuôn mẫu hàm và khuôn mẫu lớp



10/25/2005

Nội dung chương 9

9.1 Khuôn mẫu hàm

- Vai trò của khuôn mẫu hàm
- Định nghĩa khuôn mẫu hàm
- Sử dụng khuôn mẫu hàm

9.2 Khuôn mẫu lớp

- Định nghĩa khuôn mẫu lớp
- Dẫn xuất khuôn mẫu lớp
- Ví dụ khuôn mẫu lớp Vector

9.1 Khuôn mẫu hàm (function template)

- Vấn đề: Nhiều hàm chỉ khác nhau về kiểu dữ liệu tham số áp dụng, không khác nhau về thuật toán
- Ví dụ:

```
int max(int a, int b) {
  return (a > b)? a : b;
}
double max(double a, double b) {
  return (a > b)? a : b;
}
...
```

- Các ví dụ khác: các hàm swap, sort, find, select,...
- Bản chất của vấn đề? Nằm ở ngôn ngữ lập trình còn thấp, chưa gần với tư duy của con người!
- Giải pháp: Tổng quát hóa các hàm chỉ khác nhau về kiểu dữ liệu áp dụng thành **khuôn mẫu hàm**.

Định nghĩa khuôn mẫu hàm

- Ví dụ tổng quát hóa hàm max để có thể áp dụng cho nhiều kiểu dữ liệu khác nhau:

```
template <typename T>
T max(T a, T b) {
  return (a > b)? a : b;
}
```

- Ví dụ tổng quát hóa hàm swap:

```
template <class X>
void (X& a, X& b) {
  X temp = a;
  a = b;
  b = temp;
}
```

Sử dụng từ khóa **typename** hoặc **class** để khai báo **tham số khuôn mẫu**

- Một khuôn mẫu hàm inline:

```
template <typename T>
inline T max(T a, T b) { return (a > b)? a : b;}
```

Khai báo và sử dụng khuôn mẫu hàm

- Ví dụ sử dụng khuôn mẫu hàm max

Khuôn mẫu hàm

```
template <class T> T max(T a, T b);
template <class T> void swap(T&, T&);

void main() {
  int N1 = 5, N2 = 7;
  double D1 = 5.0, D2 = 7.0;
  int N = max(N1,N2);     // max<int>(int,int)
  char c = max('c','a'); // max<char>(char, char)
  double D = max(D1,D2); // max<double>(double, double)
  swap(N1,N2);           // swap<int>(int&,int&)
  swap(D1,D2);           // swap<double>(double&,double&)
  D = max(D1,A1);        // error: ambiguous
  N = max('c',A1);       // error: ambiguous
  D = max<double>(D1,A1);// OK: explicit qualification
  N = max<int>('c',A);   // OK: explicit qualification
}
```

Hàm khuôn mẫu

Khả năng áp dụng khuôn mẫu hàm

- Khả năng áp dụng một khuôn mẫu hàm là vô tận, nhưng không phải áp dụng được cho tất cả các đối số khuôn mẫu

  Ví dụ: Điều kiện ràng buộc đối với kiểu dữ liệu có thể áp dụng trong khuôn mẫu hàm max là phải có phép so sánh lớn hơn (>):

```
template <class T>
inline T max(T a, T b) { return (a > b)? a : b;}
```

  => Đối với các kiểu dữ liệu mới, muốn áp dụng được thì cần phải nạp chồng toán tử so sánh >

- **Tuy nhiên**, khả năng áp dụng được chưa chắc đã có ý nghĩa
- Ví dụ: Xác định chuỗi ký tự đứng sau trong hai chuỗi cho trước theo vần ABC

```
char city1[] = "Ha Noi", city2[] = "Hai Phong";
char* city = max(city1,city2); // ???
                               // max<char*>(char*,char*)
```

Nạp chồng khuôn mẫu hàm

- Một khuôn mẫu hàm có thể được nạp chồng bằng hàm cùng tên...

```
char* max(char* a, char* b) { if (strcmp(a,b))... }
void f() {
  char c = max('H','K');          // max<char>(char,char)
  char city1[] = "Ha Noi", city2[] = "Hai Phong";
  char* city = max(city1,city2); // max(char*,char*)
   ...
}
```

- ...hoặc bằng một khuôn mẫu hàm cùng tên (khác số lượng các tham số hoặc kiểu của ít nhất một tham số), ví dụ:

```
template <class T> T max(T a, T b, T c)    {...}
template <class T> T max(T* a, int n)      {...}
```

nhưng không được như thế này:

```
template <class X> X max(X a, X b)         {...}
```

Tham số khuôn mẫu

- Tham số khuôn mẫu hàm có thể là một kiểu cơ bản hoặc một kiểu dẫn xuất, nhưng không thể là một biến hoặc một hằng số:

```
template <class T> max(T a, Tb) { ... }    // OK
template <int N> max(int* a)    { ... }    // error
```

- Một khuôn mẫu hàm có thể có hơn một tham số kiểu:

```
template <class A, class B> void swap(A& a, B& b) {
  A t = a;
  a = b;  // valid as long as B is compatible to A
  b = t;  // valid as long as A is compatible to B
}
void f() {
  double a = 2.0;
  int b = 3;
  swap(a,b);     // swap<double,int>(double&,int&)
  swap(b,a);     // swap<int,double)(int&, double&)
}
```

- Thông thường, tham số khuôn mẫu xuất hiện ít nhất một lần là kiểu hoặc kiểu dẫn xuất trực tiếp của các tham biến:

```
template <class X> void f1(X a, int b) {...}
template <class X> void f2(X* b) {...}
template <class X,class Y> void f3(Y& a, X b) {...}
```

- Theo chuẩn ANSI/ISO C++, tham số khuôn mẫu không bắt buộc phải xuất hiện trong danh sách tham biến, nhưng cần lưu ý khi sử dụng. Ví dụ

```
#include <stdlib.h>
template <class X> X* array_alloc(int nelem) {
  return (X*) malloc(nelem*sizeof(X));
}
void main() {
  double* p1 = array_alloc(5);              // error!
  double* p2 = array_alloc<double>(5);      // OK!
  ...
  free(p2);
}
```

Khuôn mẫu hàm và hàm khuôn mẫu

- Khuôn mẫu hàm chỉ đưa ra cách thức thực hiện và sử dụng một thuật toán nào đó một cách tổng quát
- Trong khi biên dịch khuôn mẫu hàm, compiler chỉ kiểm tra về cú pháp, không dịch sang mã đích
- Mã hàm khuôn mẫu được compiler tạo ra (dựa trên khuôn mẫu hàm) khi và chỉ khi khuôn mẫu hàm được sử dụng với kiểu cụ thể
- Nếu một khuôn mẫu hàm được sử dụng nhiều lần với các kiểu khác nhau, nhiều hàm khuôn mẫu sẽ được tạo ra tương ứng
- Nếu một khuôn mẫu hàm được sử dụng nhiều lần với các kiểu tương ứng giống nhau, compiler chỉ tạo ra một hàm khuôn mẫu.

Ưu điểm của khuôn mẫu hàm

- Tiết kiệm được mã nguồn => dễ bao quát, dễ kiểm soát lỗi, nâng cao hiệu quả lập trình
- Đảm bảo được tính chặt chẽ về kiểm tra kiểu mạnh trong ngôn ngữ lập trình (hơn hẳn sử dụng macro trong C)
- Tính mở, nâng cao giá trị sử dụng lại của phần mềm: thuật toán viết một lần, sử dụng vô số lần
- Đảm bảo hiệu suất tương đương như viết tách thành từng hàm riêng biệt
- Cho phép xây dựng các thư viện chuẩn rất mạnh (các thuật toán thông dụng như sao chép, tìm kiếm, sắp xếp, lựa chọn, ...)

Nhược điểm của khuôn mẫu hàm

- Nếu muốn đảm bảo tính mở hoàn toàn thì người sử dụng khuôn mẫu hàm cũng phải có mã nguồn thực thi
  - Mã nguồn thực thi cần được đặt trong header file
  - Khó bảo vệ chất xám
- Việc theo dõi, tìm lỗi biên dịch nhiều khi gặp khó khăn
  - Lỗi nhiều khi nằm ở mã sử dụng, nhưng lại được báo trong mã định nghĩa khuôn mẫu hàm
  - Ví dụ: Compiler không báo lỗi ở dòng lệnh sau đây, mà báo lỗi ở phần định nghĩa hàm max, tại phép toán so sánh lớn hơn không được định nghĩa cho kiểu Complex:

```
Complex a, b;
   ...
Complex c = max(a,b);
```

- Định nghĩa và sử dụng không đúng cách có thể dẫn tới gia tăng lớn về mã đích, bởi số lượng hàm khuôn mẫu có thể được tạo ra quá nhiều không cần thiết.

Ví dụ: khuôn mẫu hàm copy

```
template <class S, class D>
void copy(const S * s, D* d, int n) {
  while (n--)
     *d++ = *s++;
}

void main() {
   int a[] = {1,2,3,4,5,6,7};
   double b[10];
   float  c[5];
   copy(a,b,7);      // copy<int,double>(a,b,7)
   copy(b,c,5);      // copy<double,float>(b,c,5);
    ...
}
```

9.2 Khuôn mẫu lớp (class template)

- Nhiều cấu trúc dữ liệu như Point, Complex, Vector, List, Map,... trước kia vẫn phải được định nghĩa riêng cho từng kiểu dữ liệu phần tử cụ thể, ví dụ DoubleComplex, FloatComplex, DoubleVector, IntVector, ComplexVector, DateList, MessageList, ...
- Cách thực hiện mỗi cấu trúc thực ra giống nhau, nói chung không phụ thuộc vào kiểu phần tử cụ thể

```
class IntPoint { int x,y;
public: IntPoint(int x0, int y0) : x(x0), y(y0) {}
   ...
};
class DoublePoint { double x,y;
public: DoublePoint(double x0, double y0) : x(x0), y(y0) {}
   ...
};
```

Định nghĩa khuôn mẫu lớp

```
// Point.h
template <typename T>
class Point {
   T x, y;
public:
   Point(): x(0), y(0) {}
   Point(T x0, T y0) : x(x0), y(y0) {}
   Point(const Point&);
   void move(T dx, T dy) { x += dx; y+= dy; }
   bool inRect(Point p1, Point p2);
   //...
};
template <class T>
Point<T>::Point(const Point<T>& b) : x(b.x), y(b.y) {}

template <class Coord>
bool Point<Coord>::inRect(Point<Coord> p1, Point<Coord> p2) {
   return (x >= p1.x && x <= p2.x || x >= p2.x && x <= p1.x) &&
          (y >= p1.y && y <= p2.y || y >= p2.y && x <= p1.y);
}
```

Tham số khuôn mẫu: Kiểu hoặc hằng số

Mỗi hàm thành viên của một khuôn mẫu lớp là một khuôn mẫu hàm

Sử dụng khuôn mẫu lớp: Lớp khuôn mẫu

```
#include "Point.h"
void main() {
   Point<int> A1(5,5),A2(10,10);
   Point<int> A3(A1);
   while (A3.inRect(A1,A2))
       A3.move(2,3);
   typedef Point<float> FPoint;
   FPoint B1(5.0,5.0), B2(10.0,10.0);
   FPoint B3(B1);
   while (B3.inRect(B1,B2))
       B3.move(2,3);
   //...
   Point<double> C1(B1);    // error
   if (A3.inRect(B1,B2))    // error
       ; //...
}
```

Những kiểu nào có thể áp dụng?

- Khả năng áp dụng của kiểu là vô tận, tuy nhiên không có nghĩa là áp dụng được cho tất cả các kiểu
- Một kiểu muốn áp dụng được phải hỗ trợ các phép toán được sử dụng trong mã thực thi khuôn mẫu lớp.
- Ví dụ khuôn mẫu lớp Point yêu cầu kiểu tọa độ phải áp dụng được các phép toán sau đây:
  - Chuyển đổi từ số nguyên (trong hàm tạo mặc định)
  - Sao chép (trong hàm tạo thứ hai và hàm tạo bản sao)
  - Toán tử += (trong hàm move)
  - Các phép so sánh >=, <= (trong hàm inRect)
- Việc kiểm tra kiểu được tiến hành khi sử dụng hàm thành viên của lớp khuôn mẫu, nếu có lỗi thì sẽ được báo tại **mã nguồn thực thi khuôn mẫu lớp**

Tham số khuôn mẫu: kiểu hoặc hằng số

```
template <class T = double, int N = 10>
class Array {
  T data[N];
public:
   Array(const T& x = T(0));
   int size() const { return N; }
   T  operator[](int i) const { return data[i]; }
   T& operator[](int i)       { return data[i]; }
   //...
};
template <class T, int N> Array<T,N>::Array(const T& x) {
   for (int i=0; i < N; ++ i) data[i] = x;
}
void main() {
   Array<double,10> a;
   Array<double>    b; // same as above
   Array<>          c; // same as above
   //...
}
```

Tham số mặc định

Dẫn xuất từ khuôn mẫu lớp

```
template <class T> class IOBuffer : public Array<T,8> {
public:
   IOBuffer(T x) : Array<T,8>(x) {}
   //...
};
class DigitalIO : public IOBuffer<bool> {
public:
   DigitalIO(bool x) : IOBuffer<bool>(x) {}
   //...
};
class AnalogIO : public IOBuffer<unsigned short> {
   typedef IOBuffer<unsigned short> BaseClass;
public:
   AnalogIO(unsigned short x) : BaseClass(x) {}
   //...
};
void main() {
   IOBuffer<double> delayBuf(0);
   DigitalIO di(false);
   AnalogIO  ao(0); //...
}
```

Ví dụ khuôn mẫu lớp Vector

```
template <class T>
class Vector {
  int nelem;
  T* data;
public:
  Vector() : nelem(0), data(0) {}
  Vector(int n, T d);
  Vector(int n, T *array);
  Vector(const Vector<T>&);
 ~Vector();
  int size() const { return nelem; }
  T  operator[](int i) const     { return data[i]; }
  T& operator[](int i)           { return data[i]; }
private:
  void create(int n) { data = new T[nelem=n];    }
  void destroy()     { if (data != 0) delete [] data; }
};
```

```
template <class T> Vector<T>::Vector(int n, T d) {
  create(n);
  while (n-- > 0)
    data[n] = d;
}
template <class T> Vector<T>::Vector(int n, T* p) {
  create(n);
  while (n-- > 0)
    data[n] = p[n];
}
template <class T> Vector<T>::~Vector() { destroy(); }

template <class T>
Vector<T>::Vector(const Vector<T>& a) {
   create(a.nelem);
   for (int i=0; i < nelem; ++i)
     data[i] = a.data[i];
}
```

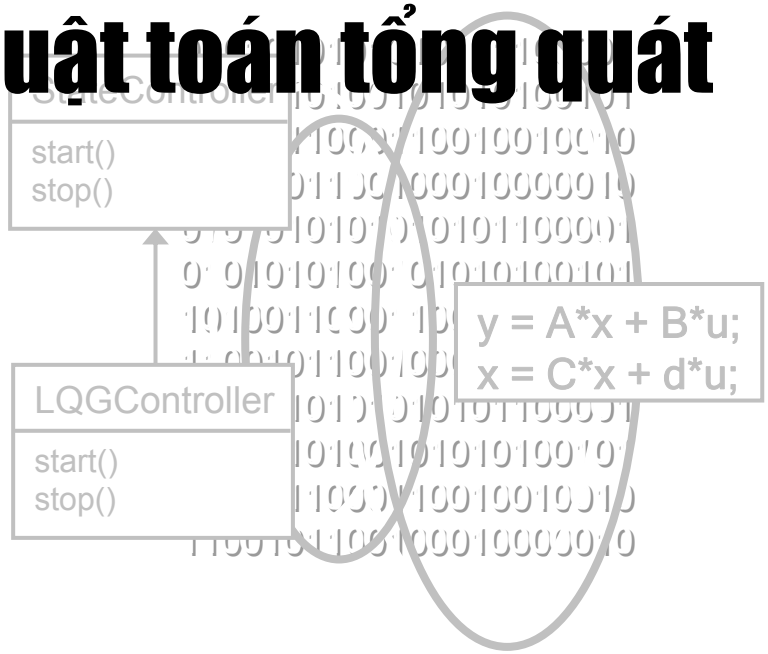
```
#include "Point.h"
#include "Vector.h"
void main()
    Vector<double> v(5,1.0);
    double d = v[0];
    v[1] = d + 2.0;
    Vector<double> v2(v);
    //...
    int b[] = {1,2,3,4,5};
    Vector<int> a(5,b);
    int n = a[0];
    a[1] = n + 2;
    Vector<int> a2(a);
    //...
    typedef Vector<Point<double> > Points;
    Points lines(5,Point<double>(0.0,0.0));
    lines[0] = Point<double>(5.0,5.0);
    //...
}
```

Bài tập về nhà

- Xây dựng một khuôn mẫu hàm xác định vị trí (địa chỉ) của phần tử có giá trị lớn nhất xuất hiện đầu tiên trong một dãy số. Viết chương trình minh họa sử dụng cho hai kiểu số liệu cụ thể.
- Từ bài tập xây dựng lớp MessageList, tổng quát hóa thành một khuôn mẫu lớp có tên là List với các phép toán (hàm thành viên) cần thiết

Kỹ thuật lập trình

Chương 10: Thuật toán tổng quát



11/2/2005

Nội dung chương 10

10.1 Tổng quát hóa kiểu dữ liệu phần tử

- Thực tế:
  - Khoảng 80% thời gian làm việc của một người thư ký văn phòng trước đây (và hiện nay ở nhiều nơi) sử dụng cho công việc tìm kiếm, sắp xếp, đối chiếu, so sánh,.... tài liệu và hồ sơ
  - Trung bình, khoảng 80% mã chương trình và thời gian thực hiện chương trình dành cho thực hiện các thuật toán ít liên quan trực tiếp tới bài toán ứng dụng cụ thể, mà liên quan tới tìm kiếm, sắp xếp, lựa chọn, so sánh... dữ liệu
- Dữ liệu được quản lý tốt nhất trong các cấu trúc dạng "container" (vector, list, map, tree, queue,...)
- Vấn đề xây dựng hàm áp dụng cho các "container": Nhiều hàm chỉ khác nhau về kiểu dữ liệu tham số áp dụng, không khác nhau về thuật toán
- Giải pháp: Xây dựng khuôn mẫu hàm, tổng quát hóa kiểu dữ liệu phần tử

- Ví dụ: Thuật toán tìm địa chỉ phần tử đầu tiên trong một mảng có giá trị lớn hơn một số cho trước:

```
template <typename T>
T* find_elem(T *first, T* last, T k) {
  while (first != last && !(*first > k))
     ++first;
  return first;
}
void main() {
  int a[] = { 1, 3, 5, 2, 7, 9, 6 };
  int *p = find_elem(a,a+7,4);
  if (p != a+7) {
    cout << "First number > 4 :" << *p;
    p = find_elem(p+1,a+7,4);
    if (p != a+7) cout << "Second number > 4:" << *p;
  }
  double b[] = { 1.5, 3.2, 5.1, 2.4, 7.6, 9.7, 6.5 };
  double *q = find_elem(b+2,b+6,7.0);
  *q = 7.0;
  ...
}
```

- Ví dụ: Thuật toán cộng hai vector, kết quả lưu vào vector thứ ba

```
#include <assert.h>
#include "myvector.h"
template <typename T>
void addVector(const Vector<T>& a, const Vector<T>& b,
               Vector<T>& c) {
  assert(a.size() == b.size() && a.size() == c.size());
  for (int i= 0; i < a.size(); ++i)
    c[i] = a[i] + b[i];
}

template <typename T>
Vector<T> operator+(const Vector<T>&a, const Vector<T>& b) {
  Vector<T> c(a.size());
  addVector(a,b,c);
  return c;
}
```

10.2 Tổng quát hóa phép toán cơ sở

- Vấn đề: Nhiều thuật toán chỉ khác nhau ở một vài phép toán (cơ sở) trong khi thực hiện hàm
- Ví dụ:
  - Các thuật toán tìm địa chỉ phần tử đầu tiên trong một mảng số nguyên có giá trị lớn hơn, nhỏ hơn, lớn hơn hoặc bằng, nhỏ hơn hoặc bằng, ... một số cho trước
  - Các thuật toán cộng, trừ, nhân, chia,... từng phần tử của hai mảng số thực, kết quả lưu vào một mảng mới
  - Các thuật toán cộng, trừ, nhân, chia,... từng phần tử của hai vector (hoặc của hai danh sách, hai ma trận, ...)
- Giải pháp: Tổng quát hóa thuật toán cho các phép toán cơ sở khác nhau!

```
template <typename COMP>
int* find_elem(int* first, int* last, int k, COMP comp) {
   while (first != last && !comp(*first, k))
     ++first;
   return first;
}
bool is_greater(int a, int b) { return a > b; }
bool is_less(int a, int b)    { return a < b; }
bool is_equal(int a, int b)   { return a == b;}
void main() {
  int a[] = { 1, 3, 5, 2, 7, 9, 6 };
  int* alast = a+7;
  int* p1 = find_elem(a,alast,4,is_greater);
  int* p2 = find_elem(a,alast,4,is_less);
  int* p3 = find_elem(a,alast,4,is_equal);
  if (p1 != alast) cout << "First number > 4 is " << *p1;
  if (p2 != alast) cout << "First number < 4 is " << *p2;
  if (p3 != alast) cout << "First number = 4 is at index "
                        <<  p3 - a;
  char c; cin >> c;
}
```

Tham số khuôn mẫu cho phép toán

- Có thể là một hàm, ví dụ

```
bool is_greater(int a, int b){ return a > b; }
bool is_less(int a, int b)   { return a < b; }
int  add(int a, int b)       { return a + b; }
int  sub(int a, int b)       { return a - b; }
...
```

- Hoặc tốt hơn hết là một đối tượng thuộc một lớp có hỗ trợ (nạp chồng) toán tử gọi hàm => **đối tượng hàm**, ví dụ

```
struct Greater {
  bool operator()(int a, int b) { return a > b; }
};
struct Less {
  bool operator()(int a, int b) { return a < b; }
};
struct Add {
  int operator()(int a, int b) { return a + b; }
};
...
```

- Ví dụ sử dụng **đối tượng hàm**

```
void main() {
  int a[] = { 1, 3, 5, 2, 7, 9, 6 };
  int* alast = a+7;
  Greater greater;
  Less    less;
  int* p1 = find_elem(a,alast,4,greater);
  int* p2 = find_elem(a,alast,4,less);
  if (p1 != alast) cout << "First number > 4 is " << *p1;
  if (p2 != alast) cout << "First number < 4 is " << *p2;

                                Greater());
  p2 = find_elem(a,alast,4,Less());
  char c; cin >> c;
}
```

Ưu điểm của đối tượng hàm

- Đối tượng hàm có thể chứa trạng thái
- Hàm toán tử () có thể định nghĩa inline => tăng hiệu suất

```
template <typename OP>
void apply(int* first, int* last, OP& op) {
  while (first != last) {
     op(*first);
     ++first;
  }
}
class Sum {
  int val;
public:
  Sum(int init = 0) : val(init) {}
  void operator()(int k) { val += k;   }
  int  value() const     { return val; }
};
```

```
class Prod {
  int val;
public:
  Prod(int init=1): val(init) {}
  void operator()(int k) { val *= k;   }
  int  value() const     { return val; }
};
struct Negate {void operator()(int& k) { k = -k;} };
struct Print { void operator()(int& k) { cout << k << ' ';} };
void main() {
  int a[] = {1, 2, 3, 4, 5, 6, 7};
  Sum sum_op;
  Prod prod_op;
  apply(a,a+7,sum_op);  cout << sum_op.value() << endl;
  apply(a,a+7,prod_op); cout << prod_op.value() << endl;
  apply(a,a+7,Negate());
  apply(a,a+7,Print());
  char c; cin >> c;
}
```

Kết hợp 2 bước tổng quát hóa

```
template <typename T, typename COMP>
T* find_elem(T* first, T* last, T k, COMP comp) {
    while (first != last && !comp(*first, k))
      ++first;
    return first;
}
template <typename T, typename OP>
void apply(T* first, T* last, OP& op) {
  while (first != last) {
     op(*first);
     ++first;
  }
}
```

Khuôn mẫu lớp cho các đối tượng hàm

```
template <typename T> struct Greater{
   bool operator()(const T& a, const T& b)
   { return a > b; }
};
template <typename T> struct Less{
   bool operator()(const T& a, const T& b)
   { return a > b; }
};
template <typename T> class Sum {
  T val;
public:
  Sum(const T& init = T(0)) : val(init) {}
  void operator()(const T& k) { val += k;   }
  T  value() const     { return val; }
};
```

```
template <typename T> struct Negate {
  void operator()(T& k) { k = -k;}
};
template <typename T> struct Print {
  void operator()(const T& k) { cout << k << ' ';}
};
void main() {
  int a[] = { 1, 3, 5, 2, 7, 9, 6 };
  int* alast = a+7;
  int* p1 = find_elem(a,alast,4,Greater<int>());
  int* p2 = find_elem(a,alast,4,Less<int>());
  if (p1 != alast) cout << "\nFirst number > 4 is " << *p1;
  if (p2 != alast) cout << "\nFirst number < 4 is " << *p2;
  Sum<int> sum_op; apply(a,a+7,sum_op);
  cout<< "\nSum of the sequence " << sum_op.value() << endl;
  apply(a,a+7,Negate<int>());
  apply(a,a+7,Print<int>());
  char c; cin >> c;
}
```

10.3 Tổng quát hóa truy lặp phần tử

- Vấn đề 1: Một thuật toán (tìm kiếm, lựa chọn, phân loại, tính tổng, ...) áp dụng cho một mảng, một vector, một danh sách hoặc một cấu trúc khác thực chất chỉ khác nhau ở cách truy lặp phần tử
- Vấn đề 2: Theo phương pháp truyền thống, để truy lặp phần tử của một cấu trúc "container", nói chung ta cần biết cấu trúc đó được xây dựng như thế nào
  - Mảng: Truy lặp qua chỉ số hoặc qua con trỏ
  - Vector: Truy lặp qua chỉ số
  - List: Truy lặp qua quan hệ móc nối (sử dụng con trỏ)
  - ...

Ví dụ thuật toán copy

- Áp dụng cho kiểu mảng thô

```
template <class T> void copy(const T* s, T* d, int n) {
  while (n--) { *d = *s; ++s; ++d; }
}
```

- Áp dụng cho kiểu Vector

```
template <class T>
void copy(const Vector<T>& s, Vector<T>& d) {
  for (int i=0; i < s.size(); ++i) d[i] = s[i];
}
```

- Áp dụng cho kiểu List

```
template <class T>
void copy(const List<T>& s, List<T>& d) {
  ListItem<T> *sItem=s.getHead(), *dItem=d.getHead();
  while (sItem != 0) {
    dItem->data = sItem->data;
    dIem = dItem->getNext(); sItem=sItem->getNext();
  }
}
```

Ví dụ thuật toán find_max

- Áp dụng cho kiểu mảng thô

```
template <typename T> T* find_max(T* first, T* last) {
  T* pMax = first;
  while (first != last) {
    if (*first > *pMax) pMax = first;
    ++first;
  }
  return pMax;
}
```

- Áp dụng cho kiểu Vector

```
template <typename T> T* find_max(const Vector<T>& v) {
  int iMax = 0;
  for (int i=0; i < v.size(); ++ i)
    if (v[i] > v[iMax]) iMax = i;
  return &v[iMax];
}
```

- Áp dụng cho kiểu List (đã làm quen):

```
template <typename T>
ListItem<T>* find_max(List<T>& l) {
  ListItem<T> *pItem = l.getHead();
  ListItem<T> *pMaxItem = pItem;
  while (pItem != 0) {
    if (pItem->data > pMaxItem->data) pMaxItem = pItem;
    pItem = pItem->getNext();
  }
  return pMaxItem;
}
```

➲ Cần tổng quát hóa phương pháp truy lặp phần tử!

Bộ truy lặp (iterator)

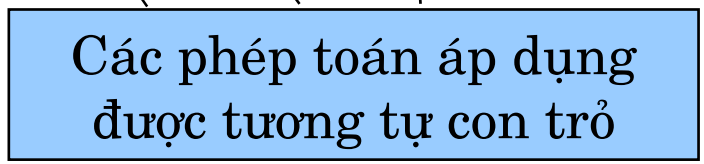
- Mục đích: Tạo một cơ chế thống nhất cho việc truy lặp phần tử cho các cấu trúc dữ liệu mà không cần biết chi tiết thực thi bên trong từng cấu trúc
- Ý tưởng: Mỗi cấu trúc dữ liệu cung cấp một kiểu bộ truy lặp riêng, có **đặc tính tương tự như một con trỏ** (trong trường hợp đặc biệt có thể là một con trỏ thực)
- Tổng quát hóa thuật toán **copy**:

```
template <class Iterator1, class Iterator2>
void copy(Iterator1 s, Iterator2 d, int n) {
  while (n--) {
    *d = *s;
    ++s;
    ++d;
  }
}
```

Các phép toán áp dụng được tương tự con trỏ

- Tổng quát hóa thuật toán **find_max**:

```
template <typename ITERATOR>
ITERATOR find_max(ITERATOR first, ITERATOR last) {
  ITERATOR pMax = first;
  while (first != last) {
    if (*first > *pMax) pMax = first;
    ++first;
  }
  return pMax;
}
```



Bổ sung bộ truy lặp cho kiểu Vector

- Kiểu Vector lưu trữ dữ liệu dưới dạng một mảng => có thể sử dụng bộ truy lặp dưới dạng con trỏ!

```
template <class T> class Vector {
  int nelem;
  T* data;
public:
  ...
  typedef T* Iterator;
  Iteratator begin() { return data; }
  Iteratator end()   { return data + nElem; }
};
void main() {
  Vector<double> a(5,1.0),b(6);
  copy(a.begin(),b.begin(),a.size());
  ...
}
```

Bổ sung bộ truy lặp cho kiểu List

```
template <class T> class ListIterator {
  ListItem<T> *pItem;
  ListIterator(ListItem<T>* p = 0) : pItem(p) {}
  friend class List<T>;
public:
  T& operator*() { return pItem->data; }
  ListIterator<T>& operator++() {
    if (pItem != 0) pItem = pItem->getNext();
    return *this;
  }
  friend bool operator!=(ListIterator<T> a,
                         ListIterator<T> b) {
     return a.pItem != b.pItem;
  }
};
```

Khuôn mẫu List cải tiến

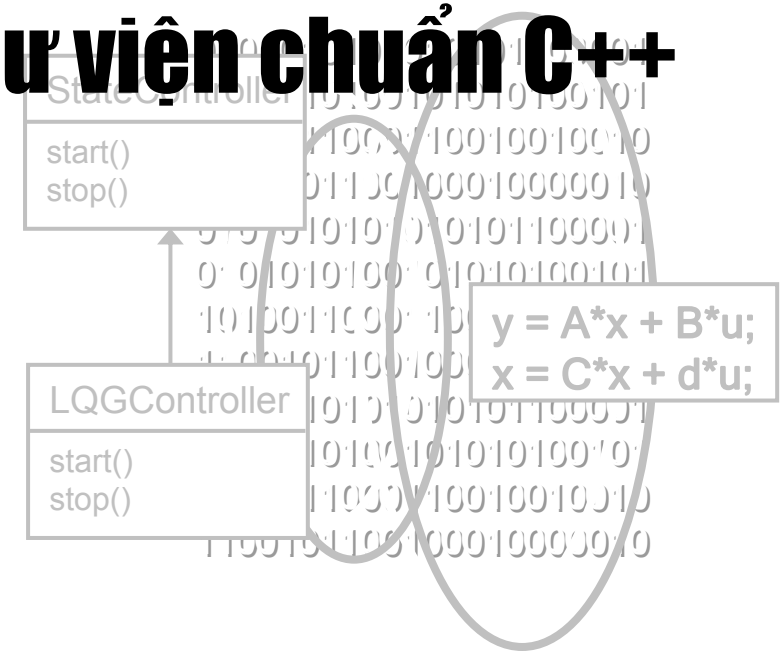
```
template <class T> class List {
  ListItem<T> *pHead;
public:
  ...
  ListIterator<T> begin() {
     return ListIterator<T>(pHead);
  }
  ListIterator<T> end() {
     return ListIterator<T>(0);
  }
};
```

Bài tập về nhà

- Xây dựng thuật toán sắp xếp tổng quát để có thể áp dụng cho nhiều cấu trúc dữ liệu tập hợp khác nhau cũng như nhiều tiêu chuẩn sắp xếp khác nhau. Viết chương trình minh họa.
- Xây dựng thuật toán cộng/trừ/nhân/chia từng phần tử của hai cấu trúc dữ liệu tập hợp bất kỳ. Viết chương trình minh họa.

Kỹ thuật lập trình

Chương 11: Thư viện chuẩn C++



11/9/2005

Nội dung chương 11

11.1 Cấu trúc thư viện chuẩn C++

11.2 Standard Template Library

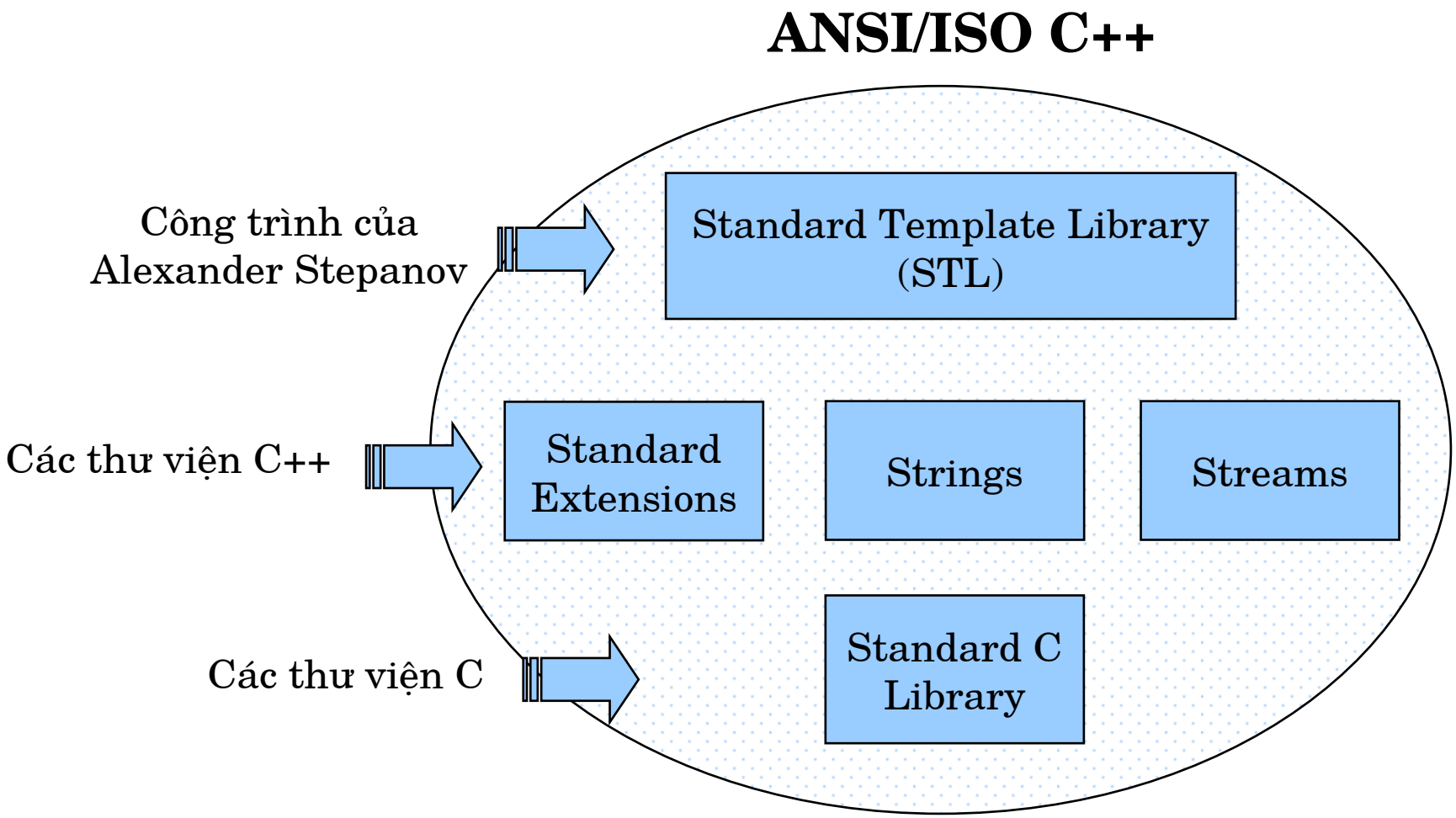
Giới thiệu chung

Các cấu trúc dữ liệu chuẩn

Thuật toán tổng quát

Các bộ truy lặp và đối tượng hàm

11.1 Cấu trúc thư viện chuẩn C++



Tất cả thư viện chuẩn C++ nằm trong phạm vi tên **std**

Standard C Library

<cassert>	Tiện ích giúp gỡ rối
<cctype>	Các thao tác với kiểu ký tự hẹp
<cwctype>	Các thao tác với kiểu ký tự rộng
<cerrno>	Mã lỗi của các hàm trong thư viện chuẩn C
<cfloat>	Các tính chất của kiểu số thực dấu phảy động
<ciso646>	Hỗ trợ lập trình với tập ký tự theo ISO 646
<climits>	Các tính chất của kiểu dữ liệu số nguyên
<clocale>	Hỗ trợ lập trình với các yêu cầu bản địa
<cmath>	Các hàm toán thông dụng
<csetjmp>	Hỗ trợ các lệnh nhảy "xa"

Standard C Library (tiếp)

<csignal> Hỗ trợ kiểm tra các trường hợp ngoại lệ
<cstdarg> Phục vụ truy nhập các tham số gọi hàm
<cstddef> Định nghĩa một số macro và kiểu thông dụng
<cstdio> Phục vụ nhập/xuất dữ liệu thông dụng
<cstdlib> Các hàm thông dụng
<cstring> Các hàm thao tác với chuỗi ký tự
<ctime> Các hàm và cấu truc thời gian và ngày tháng
<cwchar> Các hàm thao tác với chuỗi ký tự rộng

Standard Extensions

<bitset>	Khuôn mẫu lớp cho quản lý dãy bit
<complex>	Khuôn mẫu lớp cho các kiểu số phức
<exception>	Lớp cơ sở hỗ trợ lập trình với ngoại lệ
<new>	Khai báo các hàm toán tử new và delete
<stdexcept>	Định nghĩa một số lớp ngoại lệ thông dụng
<typeinfo>	Hỗ trợ khai thác thông tin kiểu động
<locale>	Mở rộng hỗ trợ lập trình bản địa
<valarray>	Các khuôn mẫu lớp và lớp mảng giá trị

Strings

- Header file: <string>
  - Khuôn mẫu lớp: basic_string <T,...>
  - Lớp string: typedef basic_string<char,...> string
  - Lớp wstring: typedef basic_string<wchar_t,...> wstring
- Cho phép lập trình với các chuỗi ký tự một cách rất thuận tiện
  - Không cần quan tâm tới quản lý bộ nhớ động
  - Có thể sao chép, gán giống như các kiểu dữ liệu cơ bản
  - Có thể truy nhập ký tự qua chỉ số toán tử [] giống như chuỗi ký tự thô
  - Có thể áp dụng các phép toán +, ==, !=, > , <, ...
  - Có thể truy nhập chuỗi con, tìm kiếm, thay thế ký tự,...
  - Có thể áp dụng các thuật toán tổng quát (string, wstring cũng được coi là các container, có các hàm begin(), end())

Sử dụng string

- Khai báo biến **string**

```
string s1( "Hello" );
string s2( 8, 'x' );        // 8 'x' characters
string month = "March";     // Implicitly calls constructor
```

- Các hàm truy nhập thuộc tính

```
n=s1.size();        // Number of characters in string
n=s1.length();      // Same as size()
n=s1.capacity();    // Number of elements that can be
                    //  stored without reallocation
n=s1.max_size(); // Maximum possible string size
if (s1.empty())     // Returns true if empty
   ...
s1.resize(newlength);      // Resizes string to newlength
```

- Phép gán

```
s2 = s1;              // Makes a separate copy
s2.assign(s1);        // Same as s2 = s1;
myString.assign(s, start, N); // Copies N characters from s
                              //  beginning at index start
s2[0] = s3[2];        // Individual characters
```

- Ghép chuỗi ký tự

```
s3.append( "pet" );
s3 += "pet";          // Both add "pet" to end of s3
s3.append( s1, start, N ); // Appends N characters from s1,
                           // beginning at index start
```

- So sánh chuỗi ký tự (theo vần ABC)

```
==, !=, <, >, <=,>=
```

- Truy nhập chuỗi con

```
s = s1.substr( start, N );
```

- Hoán đổi hai chuỗi ký tự:

```
s1.swap(s2);
```

- Các hàm tìm kiếm: trả về chỉ số nếu tìm thấy và **`string::npos`** nếu không tìm thấy

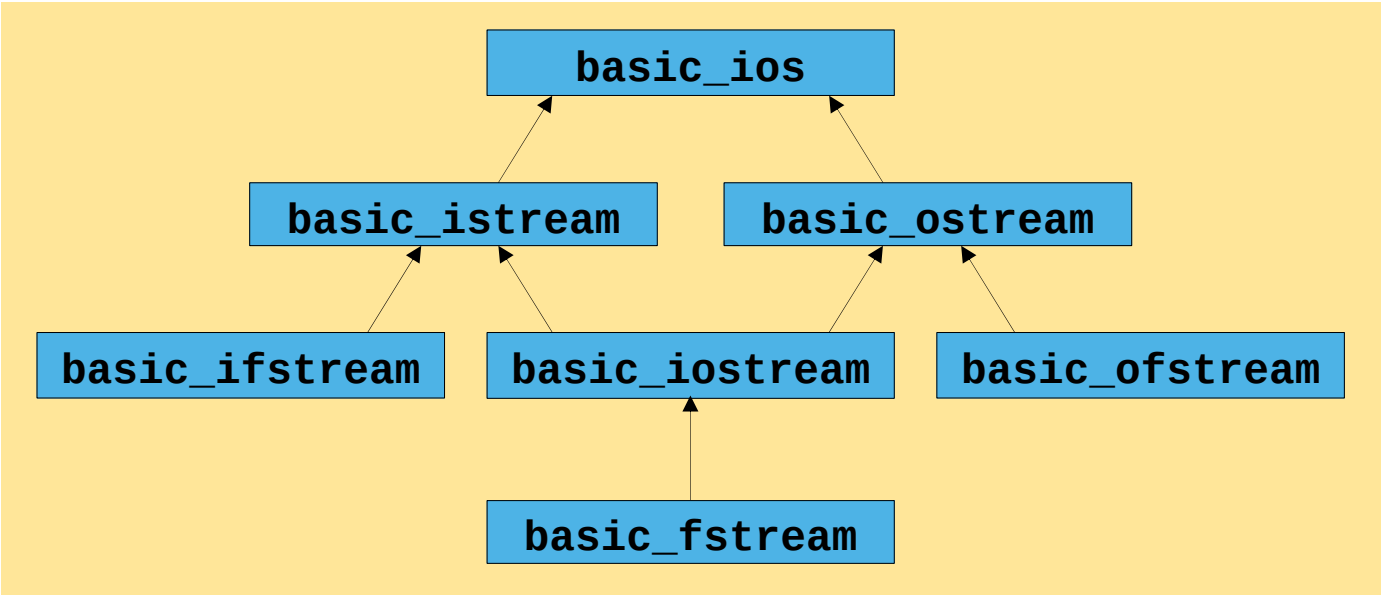
```
i=s1.find(s2);
i=s1.rfind(s2);              // Searches right-to-left
i=s1.find_first_of(s2);      // Returns first occurrence of any
                             //  character in s2
i=s1.find_first_of("abcd"); // Returns index of first 'a',
                                        // 'b', 'c' or 'd'
i=s1.find_last_of(s2); // Finds last occurrence of any char in s2
i=s1.find_first_not_of(s2);  // Finds first character NOT in s2
i=s1.find_last_not_of(s2);   // Finds last character NOT in s2
```

- Các hàm xóa và thay thế

```
s1.erase(start);    // Erase from start to end of string
s1.replace(begin,N,s2); // begin: index in s1 to start replacing
                        // N: number of characters to replace
                        // s2: replacement string
```

Streams

- Header files: <iostream>, <fstream>, <strstream>,...
- Được xây dựng lại hoàn toàn
  - Trên cơ sở khuôn mẫu lớp
  - Về cơ bản là tương thích với các thư viện trước đây
  - Linh hoạt và hiệu suất cao hơn so với các thư viện trước đây
  - Hỗ trợ kỹ thuật lập trình tổng quát



11.2 Standard Template Library (STL)

- STL được Alexander Stepanov lần đầu tiên xây dựng, khi từ Nga sang Mỹ làm việc cho Hewlet Packard
- STL sử dụng kỹ thuật lập trình tổng quát và cung cấp:
  - Các cấu trúc dữ liệu chuẩn (Containers)
  - Các bộ truy lặp (Iterators)
  - Các thuật toán tổng quát (Algorithms)
- Ý tưởng chính:
  - Với N kiểu dữ liệu, M cấu trúc dữ liệu, và K thuật toán, lẽ ra cần xây dựng N*M cấu trúc cụ thể và N *M *K hàm cụ thể
  - STL (with C++ templates): **M cấu trúc + K thuật toán**

Ví dụ 1

- Đọc dãy số nguyên từ một tệp tin “numbers.txt”, tính toán và hiển thị các giá trị
  - Nhỏ nhất, lớn nhất
  - Đứng giữa
  - Trung bình cộng
  - Trung bình nhân (y1 * y2 * … yn)^(1/n)
- Bình thường ta cần bao nhiêu dòng lệnh để thực hiện?
  - Cấp phát bộ nhớ động (lớn bao nhiêu?)
  - Vòng lặp đọc các giá trị từ tệp tin
  - Các vòng lặp tính toán
  - …

Giải pháp sử dụng STL

```
vector<int> v;

copy(istream_iterator<int>(ifstream("numbers.txt")),
     istream_iterator<int>(), back_inserter(v));

sort(v.begin(), v.end());

cout << "min/max:" << v.front() << " " << v.back() << endl;
cout << "median : " << *(v.begin() + (v.size()/2)) << endl;
cout << "average: " << accumulate(v.begin(), v.end(), 0.0)
                       / v.size() << endl;
cout << "geomean: " << pow(accumulate(v.begin(),v.end(),
         1.0, multiplies<double>()),1.0/v.size()) << endl;
```

Ví dụ 2

- Viết một chương trình in ra các từ và tần suất xuất hiện trong một tệp tin, sắp xếp theo thứ tự ABC

```
vector<string> v;
map<string, int> m;

copy(istream_iterator<string>(ifstream("words.txt")),
     istream_iterator<string>(), back_inserter(v));

for (vector<string>::iterator vi = v.begin();
     vi != v.end(); ++vi)
   ++m[*vi];

for (map<string, int>::iterator mi = m.begin();
     mi != m.end(); ++mi)
   cout << mi->first << ": " << mi->second << endl;
```

STL containers

- Các cấu trúc dãy (sequence containers)
  - vector
  - list (móc nối hai chiều)
  - deque
- Các cấu trúc sắp xếp liên hệ (associative containers)
  - map, multimap,
  - set, multiset
- Các cấu trúc dẫn xuất (container adapters)
  - queue, priority_queue,
  - stack

Các hàm thành viên chung

- Hàm thành viên cho tất cả các cấu trúc
  - Hàm tạo mặc định, hàm tạo bản sao
  - Hàm hủy
  - **`empty`**
  - **`max_size`**, **`size`**
  - **`= < <= > >= == !=`**
  - **`swap`**
- Hàm thành viên chỉ cho các cấu trúc dãy
  - **`begin`**, **`end`**
  - **`rbegin`**, **`rend`**
  - **`erase`**, **`clear`**

vector<T>

- #include <vector>
- Một mảng động thực sự: truy nhập tùy ý, co dãn được
- Khai báo đơn giản:

```
vector<int> v(10);
```

- Hàm tạo:
  - `vector<T>()`
  - `vector<T>(size_t num_elements)`
  - `vector<T>(size_t num, T init)`
- Thuộc tính:
  - `v.empty()`
  - `v.size()`
  - `v.capacity()`
  - `v.begin()`
  - `V.end()`

- Bổ sung phần tử
  - **v.push_back(42);**
  - **v.insert(iter before, T val)**
  - **v.insert(iter before, iter start, iter end)**
- Truy nhập phần tử
  - **v.at(i)** // reference, range checked!
  - **v[i]** // reference, not range checked
  - **v.front()** // reference to first element
  - **v.back()** // reference to last element
    - **v.back() = 4;** // legal if size > 0
- Loại bỏ phần tử
  - **v.pop_back()** // removes last element
  - **v.clear()** // removes everything
  - **v.erase(iterator i)**
  - **v.erase(iter start, iter end)**

Ví dụ sử dụng vector<T>

```
vector<char> v;
for (int i = 0; i < 10; ++i)
     v.push_back('A' + i);
cout << v[0] << v.back() << endl;        // AJ
v.pop_back();
cout << v.size() << v.back() << endl;    // 9I
for (size_t i = 0; i < v.size(); ++i)
   cout << v[i]; // ABCDEFGHI
cout << endl;
for (vector<char>::iterator p = v.begin(); p != v.end();
   ++p)
   cout << *p; // ABCDEFGHI
```

list<T>

- Danh sách móc nối hai chiều
- Chỉ truy nhập tuần tự
- Hàm tạo
  - `list<T>()`
  - `list<T>(size_t num_elements)`
  - `list<T>(size_t num, T init)`
- Tính chất
  - `l.empty()`
  - `l.size()`
  - `l.begin()`
  - `l.end()`

- Bổ sung phần tử
  - `l.push_back(43);`
  - `l.push_front(31);`
  - `l.insert(iter b, iter s, iter s)/`
- Truy nhập phần tử
  - `l.front() // T &`
  - `l.back() // T &`
- Loại bỏ phần tử
  - `l.pop_back()`
  - `l.pop_front()`
  - `l.erase(iterator i)`
  - `l.erase(iter start, iter end)`

Ví dụ sử dụng list<T>

```
list<char> l;
for (int i = 0; i < 4; ++i)
{
   l.push_front(i + 'A');
   l.push_back(i + 'A');
}
for (list<char>::iterator i = l.begin(); i != l.end();
   ++i)
   cout << *i; // DCBAABCD
```

Ví dụ sử dụng map<key,value>

```
#pragma warning(disable: 4786)
#include <iostream>
#include <map>
#include <string>
using namespace std;

void main() {
   map<string,long> telbook;
   telbook["Bush"]  = 1234567;
   telbook["Putin"] = 7654321;
   telbook["Chirac"]= 1231231;
   //...
   string name;
   while (cin >> name) {
        if(telbook.find(name) != telbook.end())
               cout << "The phone number for " << name
                    << " is " << telbook[name] << '\n';
        else
               cout << "No such name exists\n";
   }
}
```

Thuật toán tổng quát

- Các thuật toán thông dụng nhất, bao gồm:
  - Các thuật toán quản lý dữ liệu: copy, count, find, binary_search, sort, merge, replace,..
  - Các thuật toán tập hợp: set_union, set_difference, set_intersect,...
  - Các thuật toán số học: transform, accumulate, generate, for_each...
- Các thuật toán yêu cầu đầu vào là các bộ truy lặp (iterators), kết quả đầu ra nhiều khi cũng là bộ truy lặp
- Mỗi thuật toán có thể áp dụng cho nhiều cấu trúc dữ liệu khác nhau, nhưng không phải tất cả

Khuôn mẫu đối tượng hàm

STL function objects	Ý nghĩa
`plus< T >`	Cộng
`minus< T >`	Trừ
`multiplies< T >`	Nhân
`divides< T >`	Chia
`modulus< T >`	Phần dư
`negate< T >`	Đảo dấu
`equal_to< T >`	Bằng nhau
`greater< T >`	Lớn hơn
`greater_equal< T >`	Lớn hơn hoặc bằng
`less< T >`	Nhỏ hơn
`less_equal< T >`	Nhỏ hơn hoặc bằng
`not_equal_to< T >`	relational
`logical_and< T >`	AND
`logical_not< T >`	NOT
`logical_or< T >`	OR

Các loại iterators

- Truy lặp ngẫu nhiên (Random Access Iterators): có thể áp dụng *, ++, --, [], !=, ==, >, <, >=, <=
- Truy lặp hai chiều (Binary Iterators): có thể áp dụng *, ++, --, !=, ==
- Truy lặp tiến (Forward Iterators): có thể áp dụng *, ++, !=, ==
- Truy lặp lùi (Backward Iterators): có thể áp dụng *, --, !=, ==
- Nhập (Input Iterators): Đọc/nhập dữ liệu, dịch một chiều (tiến)
- Xuất (Output Iterators): Ghi/xuất dữ liệu, dịch một chiều (tiến)

Bài tập về nhà

- Hoàn thiện bài tập chương 10
- Ôn tập các phần đã học
- Chuẩn bị sẵn các câu hỏi thảo luận