# CHƯƠNG I. CÁC KHÁI NIỆM CƠ BẢN

## I.1. Các khái niệm cơ bản

### I.1.1. Ngôn ngữ lập trình

Khi bạn muốn điều khiển máy tính, bạn cần biết cách để nói chuyện với nó. Không giống như những con vật cưng của bạn như chó, mèo. Chúng có những ngôn ngữ bí hiểm, để nói chuyện với máy tính, bạn dùng ngôn ngữ lập trình được tạo bởi con người. Một chương trình máy tính là một đoạn văn bản - như sách, nhưng nó có cấu trúc riêng biệt. Ngôn ngữ lập trình, thường dễ hiểu đối với con người, được cấu trúc chặt chẽ hơn so với ngôn ngữ bình thường và kho từ vựng cũng nhỏ hơn. C++ là một trong những ngôn ngữ này.

Một khi bạn đã viết một chương trình máy tính, bạn cần một cách cho máy tính chạy nó để giải thích những gì bạn đã viết. Điều này thường được gọi là *thực hiện chương trình* của bạn. Cách bạn làm điều này sẽ phụ thuộc vào ngôn ngữ lập trình và môi trường của bạn. Chúng ta sẽ nói sau về việc làm thế nào để thực hiện chương trình.

Có nhiều ngôn ngữ lập trình, mỗi ngôn ngữ có cấu trúc và từ vựng riêng, nhưng chúng có nhiều điểm giống nhau. Như vậy, khi bạn đã học được một ngôn ngữ, thì học ngôn ngữ khác sau đó sẽ dễ dàng hơn.

### I.1.2. Các thuật ngữ

#### *a. Lập trình*

*Lập trình* là thao tác viết các lệnh theo cách mà cho phép máy tính hiểu và thực hiện những lệnh đó. Các lệnh này được gọi là *mã nguồn*. Đó là những gì bạn sẽ viết. Chúng ta sẽ thấy một số mã nguồn trong một vài trang tới.

#### *b. Tập tin thi hành*

Kết quả cuối cùng của chương trình là bạn sẽ có một tập tin thi hành. Tập tin thi hành là một tập tin máy tính có thể chạy. Nếu bạn sử dụng HĐH Windows, bạn sẽ thấy những file này có đuôi như EXE. Một chương trình máy tính như MS Word là một file thi hành. Một số chương trình có tập tin bổ sung (file đồ họa, file nhạc, vv.) nhưng mỗi chương trình yêu cầu một tập tin thi hành. Để làm cho một tập tin thi hành, bạn cần một trình biên dịch, có khả năng dịch các tập tin mã nguồn thành một tập tin thi hành. Nếu không có trình biên dịch, bạn không thể làm bất cứ điều gì ngoại trừ nhìn vào mã nguồn của bạn. Tiếp theo, chúng ta sẽ giúp bạn làm quen với một trình biên dịch.

#### *c. Biên soạn và biên dịch tập tin nguồn*

Mã nguồn là tập tin chứa các lệnh của chương trình và được lưu theo dạng văn bản phẳng (plaint text). Các tập tin văn bản phẳng không chứa thông tin gì khác ngoài nội dung văn bản của tập tin. Một tập tin được tạo ra bằng Microsoft Word (hoặc sản phẩm tương tự) không phải là một tập tin văn bản phẳng bởi vì nó chứa thông tin về các font chữ được sử dụng, kích thước của văn bản, định dạng, ... mà bạn đã áp dụng cho tập tin. Trong Windows, phần mềm Notepad là một trình soạn thảo văn bản phẳng.

Các trình biên tập mã nguồn thường cung cấp hai tính năng: làm nổi bật cú pháp và tự động thụt đầu dòng. Hai tính năng này giúp bạn dễ dàng nhận ra các phần tử khác nhau của chương trình.

### I.1.3. Bảng ký tự của C++

Bảng kí tự được phép dùng trong các câu lệnh của ngôn ngữ C++ gồm:

- Các chữ cái la tinh (viết thường và viết hoa): a .. z và A .. Z. Cùng một chữ cái nhưng viết thường phân biệt với viết hoa. Ví dụ chữ cái 'a' là khác với 'A'.
- Dấu gạch dưới: _
- Các chữ số thập phân: `0, 1, ..., 9`
- Các ký hiệu toán học: `+, -, *, /, % , &, ||, !, >, <, = ...`
- Các ký hiệu đặc biệt khác: `, ; : [ ], {}, #,` dấu cách,`v.v...`

### I.1.4. Tên gọi (Identifier)

Để phân biệt các đối tượng trong chương trình, mỗi đối tượng cần có một tên gọi. Hầu hết mọi đối tượng được viết ra trong chương trình thuộc 2 dạng:

- Dạng 1: Tên gọi đã có sẵn trong ngôn ngữ: Từ khoá, tên các hàm chuẩn ...
- Dạng 2: Tên gọi người sử dụng (NSD) tạo ra để đặt tên cho hằng, biến, kiểu, hàm ... các tên gọi do NSD tự đặt phải tuân theo một số qui tắc sau:
  - Là dãy ký tự liên tiếp (không chứa dấu cách) và phải bắt đầu bằng chữ cái hoặc dấu gạch dưới _
  - Tên gọi có phân biệt chữ cái in hoa và in thường.
  - Tên gọi không được trùng với từ khóa, tên hàm.
  - Số lượng ký tự trong tên gọi là tuỳ ý.

*Chú ý*: Các tên gọi có sẵn của C++ cũng tuân thủ theo đúng qui tắc trên.

Trong một chương trình nếu NSD đặt tên sai thì trong quá trình xử lý sơ bộ (trước khi chạy chương trình) máy sẽ báo lỗi (gọi là lỗi văn phạm).

*Ví dụ 1.1*:

- Các tên gọi sau đây là đúng (được phép): *i, i1, j, tinhoc, tin_hoc, luu_luong, mohd, zara, abc, move_name, a_123, myname50, _temp, a23b9, retVal, ...*
- Các tên gọi sau đây là sai (không được phép): *1i, tin hoc, luu-luong-nuoc*
- Các tên gọi sau đây là khác nhau: *ha_noi, Ha_noi, HA_Noi, HA_NOI, ...*

### I.1.5. Từ khoá

Một từ khoá là loại tên gọi được qui định trước trong NNLT với một ý nghĩa cố định, thường dùng để chỉ các loại dữ liệu hoặc các mệnh đề cấu trúc. NSD có thể tạo ra những tên mới để chỉ các đối tượng của mình nhưng không được phép trùng với từ khoá. Dưới đây bảng từ khoá thường gặp. Ý nghĩa của các từ này, sẽ được trình bày dần trong các mục liên quan.

| | | | |
|---|---|---|---|
| `asm` | `else` | `new` | `this` |
| `auto` | `enum` | `operator` | `throw` |

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

Một đặc trưng của C++ là các từ khoá luôn luôn được viết bằng chữ thường.

### I.1.6. Chú thích trong chương trình

Một chương trình thường được viết một cách ngắn gọn, do vậy thông thường bên cạnh các câu lệnh chính thức của chương trình, NSD còn được phép viết vào chương trình các câu chú thích, nhằm giải thích ý nghĩa các câu lệnh. Một chú thích có thể giải thích về nhiệm vụ, mục đích, cách thức của thành phần đang được chú thích như biến, hằng, hàm hoặc công dụng của một đoạn lệnh ... Các chú thích làm cho chương trình sáng sủa, dễ đọc, dễ hiểu và vì vậy dễ bảo trì, sửa chữa về sau.

Có 2 cách báo cho C++ biết một đoạn văn bản là chú thích:

- Nếu chú thích là một đoạn văn bản liên tiếp trong một hoặc trên nhiều dòng: Ta đặt đoạn chú thích đó giữa cặp dấu mở /* và đóng */.
- Nếu chú thích là đoạn văn bản nằm trên một dòng hoặc bắt đầu từ một vị trí nào đó cho đến hết dòng: Ta đặt dấu // ở vị trí bắt đầu có chú thích.

Khi biên dịch chương trình, các đoạn chú thích sẽ được bỏ qua.

*Chú ý*: Cặp dấu chú thích /* ... */ không được phép viết lồng nhau, ví dụ dòng chú thích sau là không được phép:

```
/* Đây là chú thích /* chứa chú thích này */ như chú thích con */
```

Cần phải sửa lại như sau:

```
/* Đây là chú thích chứa chú thích này như chú thích con */
```

hoặc là phân đoạn như sau:

```
/* Đây là chú thích */ /*chứa chú thích này*/ /*như chú thích con */
```

## I.2. Phần mềm miễn phí Code::Blocks

### I.2.1. Code::Blocks là gì?

Code::Blocks là một phần mềm mã nguồn mở, miễn phí. Nó có vai trò là một môi trường phát triển tích hợp cho phép viết mã nguồn và biên dịch chương trình của bạn ngay trong môi trường của nó. Lưu ý, Code::Blocks không phải một trình biên dịch phần mềm.

Trong giáo trình này, chúng ta sử dụng Code::Blocks kết hợp với phần mềm biên dịch GCC từ MinGW, một trình biên dịch miễn phí cho Windows. Code::Blocks sẽ lo việc xử lý tất cả các chi tiết lộn xộn của việc thiết lập và gọi trình biên dịch, gỡ lỗi.

### I.2.2. Tải và cài đặt phần mềm Code::Blocks

Trong phần này, ta đi tìm hiểu cách tải và cài đặt phần mềm Code::Blocks kèm MinGW từ Internet. Quá trình có thể thực hiện theo các bước sau:

**Bước 1: Tải Code::Blocks**

- Vào địa chỉ website: *http://www.codeblocks.org/downloads*
- Theo liên kết "*Download the binary release*"
- Chọn trong phần dành cho Windows 2000 / XP / Vista / 7
- Tìm tệp có bao gồm chuỗi *mingw* trong tên. Có dạng *codeblocks-10.05mingw-setup.exe*; (số hiệu có thể khác).
- Tải tệp này về máy tính của bạn.

Bạn có thể thường xuyên vào trang web này để xem các thông tin cập nhật và tải các bản cập nhật mới của Code::Blocks về để sử dụng.

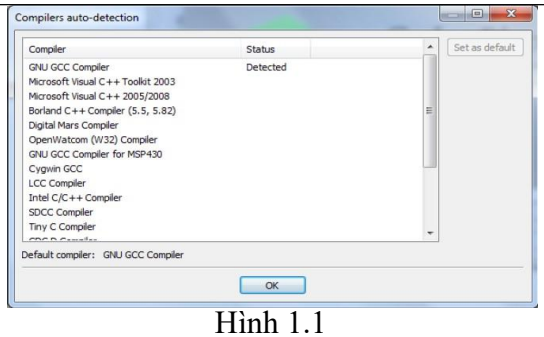
**Bước 2: Cài Code::Blocks**

- Bấm đúp chuột vào tệp đã tải trong bước 1 để cài đặt.
- Bấm *next* vài lần. Hướng dẫn cài đặt sẽ giả sử bạn cài đặt theo đường dẫn mặc định là: **C:\ProgramFiles\CodeBlocks**, nhưng bạn có thể cài theo đường dẫn khác nếu muốn
- Để cài đặt đầy đủ, chọn *Full: All plugins, all tools, just everything* từ menu thả xuống *Select the type of install.*

**Bước 3: Chạy Code::Blocks**

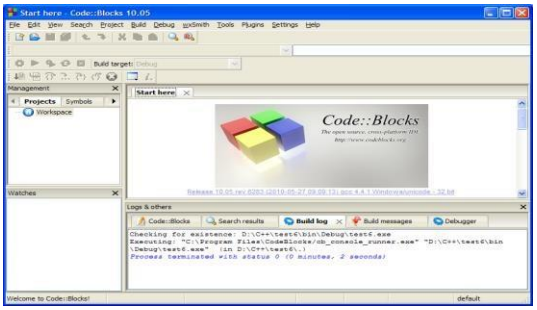
Để tạo một chương trình nguồn C++ mới, biên dịch và chạy thử, bạn có thể làm theo hướng dẫn dưới đây:

- Chạy phần mềm Code::Blocks từ menu *Start* của Windows hoặc Shortcut của nó trên màn hình Desktop (thường có sau khi cài đặt Code::Blocks). Bạn sẽ thấy xuất hiện hộp thoại *Compilers auto-detection* thông báo rằng Code::Blocks đã tự dò thấy phần mềm biên dịch kết hợp với nó. Như hình 1.1 là *GNU GCC Compiler*:




Hình 1.1

Bấm *OK*. Code::Blocks có thể yêu cầu bạn chọn kết hợp nó với trình mặc định để mở các tệp C/C++. Sau đó thì màn hình khởi đầu xuất hiện như hình 1.2:



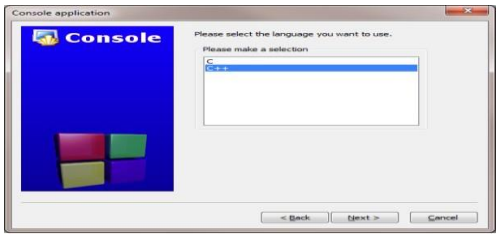

Hình 1.2

Chọn menu *File -> New -> Project...* xuất hiện cửa sổ (hình 1.3):


Hình 1.3

Chọn "*Console Application*" và nhấn nút "*Go*". Tất cả các chương trình ví dụ nguồn trong cuốn sách đều có thể chạy dưới dạng Console Application.

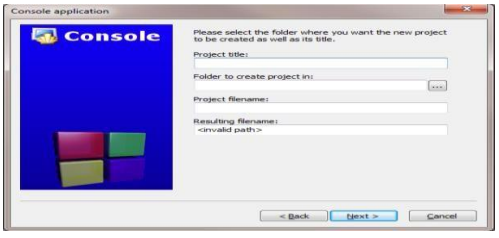
Bấm *next* và xuất hiện hộp thoại *Language Selection* (Hình 1.4):




Hình 1.4

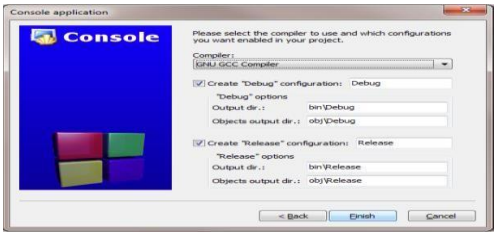
Bạn có thể được yêu cầu chọn C hoặc C++. Vì ta học C++, nên bạn chọn *C++* và bấm *Next*.

Mỗi chương trình bạn viết, được coi như một (dự án) Project và sau khi bạn bấm *Next*, Code::Blocks sẽ yêu cầu bạn xác nhận thư mục lưu và tên tệp dự án của mình như hình 1.5.



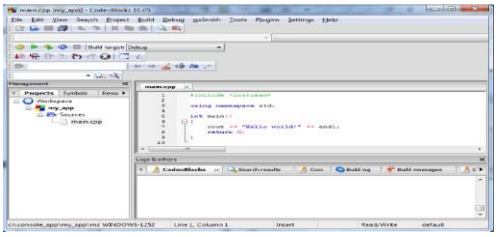

Hình 1.5

Trong màn hình này, bạn phải nhập tên tệp dự án vào hộp *Project Title*. Chọn vị trí trên đĩa để lưu project bằng cách bấm chuột vào nút ... phía phải, hoặc tự gõ đường dẫn vào hộp *Folder to create project in.*Sau này các tệp chương trình trong dự án của bạn đều được lưu tại ví trí đó. Bấm *next*. Xuất hiện màn hình Console Application (hình 1.6), bạn cứ việc bấm Finish.



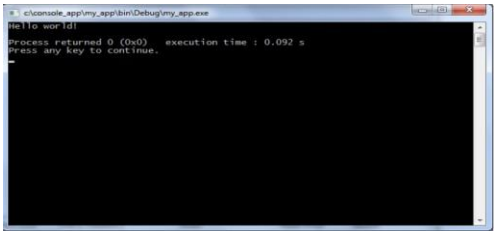

Hình 1.6

Xuất hiện cửa sổ làm việc với dự án của bạn, như hình 1.7. Tìm tệp main.cpp trong cây thư mục bên trái cửa sổ. Nếu không thấy thì bấm chuột vào nút dấu + bên trái biểu tượng Folder Source. Bấm đúp chuột vào tên tệp *main.cpp* bạn sẽ thấy cửa sổ bên phải hiện ra đoạn mã có sẵn như hình 1.8:



Hình 1.8

Tại thời điểm này, Code::Blocks đã viết sẵn cho bạn tệp nguồn mặc định *main.cpp*, và bạn có thể thay đổi nội dung và cả tên tệp này nếu muốn. Chú ý phần mở rộng tập tin: *.cpp* là phần mở rộng chuẩn cho tập tin nguồn C++ mặc dù các file *.cpp* là văn bản phẳng. Đến bây giờ, chương trình mẫu của Code::Bloack chỉ có một lệnh in ra màn hình câu "*Hello World*!", bạn có thể chạy nó bằng cách nhấn phím F9 để biên dịch và chạy thử chương trình, hoặc chọn menu *Build | Build and Run*.




Hình 1.9

Bây giờ bạn có một cửa sổ màu nền đen, với dòng chữ *Hello World!* màu trắng. Đó là giao diện của chương trình đang chạy của bạn! Dòng bên dưới, chương trình có đoạn thông báo tự động, rằng chương trình của bạn không có lỗi (mã lỗi = 0) và thời gian chạy chương trình tính bằng giây. Bạn có thể bấm Enter để quay lại màn hình biên tập mã. Bạn có thể chỉnh sửa nội dung *main.cpp* và sau đó nhấn F9 để biên dịch và chạy lại chương trình.

### I.2.3. Xử lý một vài sự cố nhỏ

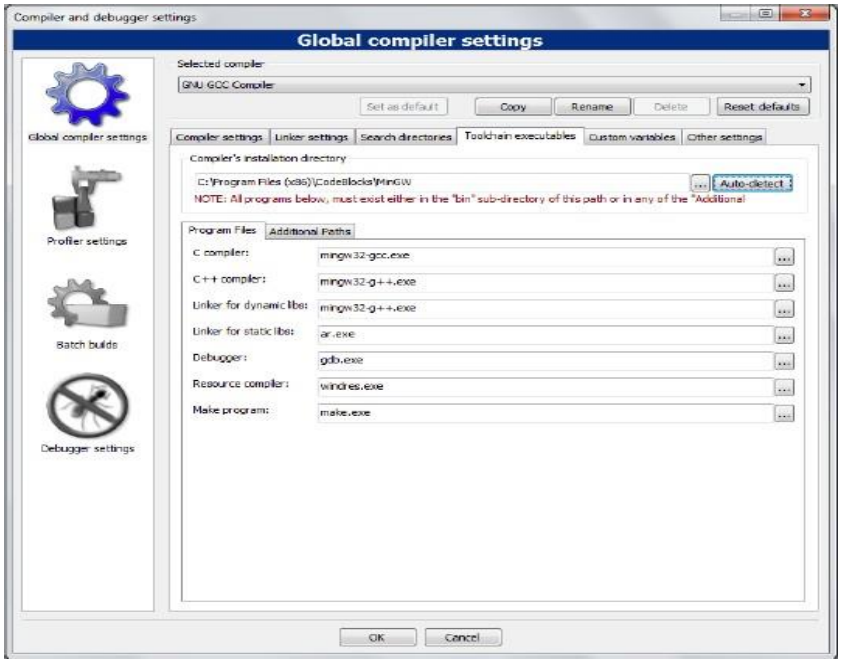
Nếu vì một lý do nào đó bạn không chạy được chương trình, thì có thể là có lỗi biên dịch hoặc môi trường đã không được thiết lập đúng.

### *a. Do môi trường cài đặt*

Những lỗi phổ biến nhất cho thấy nếu mọi thứ không làm việc là một thông điệp như "*CB01 - Debug*" rằng trình biên dịch không hợp lệ. Có lẽ đường dẫn *toolchain* trong tùy chọn trình biên dịch không được thiết lập một cách chính xác?

Trước tiên, hãy chắc chắn rằng bạn đã tải về đúng phiên bản Code:: Blocks, trong đó bao gồm trình biên dịch của MinGW. Nếu điều đó không giải quyết được vấn đề, thì có thể là do Code::Blocks không tự tìm thấy vị trí đặt trình biên dịch trên đĩa. Để kiểm tra trạng thái "auto-detected" của Code::Blocks, bạn chọn menu "*Settings|Compiler and Debugger ...*". Tiếp theo, ở phần cửa sổ bên trái, chọn"*Global Compiler Settings*" (nó có biểu tượng dụng cụ) và cửa sổ bên phải, chọn tab "*Toolchain executables*".

Tab này có một nút "*Auto-detected*" và bạn có thể bấm chuột vào nó để Code::Blocks tự tìm lại trình biên dịch của MinGW. Việc đó có thể sửa chữa vấn đề. Nếu không, bạn có thể tự nhập đường dẫn nơi lưu thư mục MinGW vào hộp văn bản *Compiler's Installation Directory*, sau đó bấm OK. Dưới đây là một ảnh chụp màn hình thể hiện những gì trông giống như trên hệ thống của tôi. Thông thường thì đường dẫn đến thư mục chứa MinGW sẽ như hình 1.10.



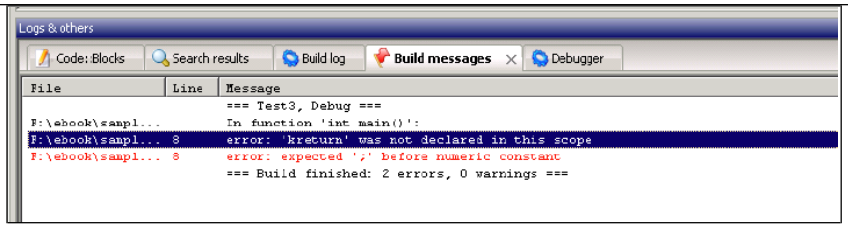

Hình 1.10

Khi thiết lập xong, quay về với chương trình của mình. Bạn có thể bấm F9 để làm lại.

### *b. Một số lỗi cú pháp*

Lỗi trình biên dịch có thể xảy ra nếu bạn đã sửa đổi file *main.cpp* sai cách và làm trình biên dịch không hiểu.

Để biết những gì sai, hãy xem cửa sổ "*Build messages*" hoặc "*Build log*" ở phía dưới màn hình soạn thảo. Cửa sổ "*Build messages*" sẽ cho bạn biết về lỗi biên dịch, "*Build log*" hiển thị cho bạn vài thông tin khác. Hình 1.11 là một mô tả lỗi:




Hình 1.11

Trong trường hợp này, nó cho bạn thấy tên của tập tin, số hiệu dòng có lỗi, và một chuỗi giải thích lỗi ngắn gọn. Ở đây, do tôi đã thay đổi dòng *return 0;* thành *kreturn 0;* và đó là một lệnh C++ không hợp lệ, vì vậy tôi có một lỗi.

Trong suốt cuốn sách này, bạn sẽ thấy rất nhiều mã ví dụ. Đối với mỗi ví dụ, bạn có thể tạo ra một Console Project mới và gõ chương trình ví dụ đó vào. Các ví dụ thường ngắn nên bạn chỉ cần viết nó vào file *main.cpp* của Project mới tạo, sau đó biên dịch và chạy thử.

Trong phần phụ lục của cuốn giáo trình, có trình bày cách sử dụng chức năng Debug của Code::Blocks để dò và gỡ lỗi cho chương trình của bạn. Hiện giờ, chưa phải là lúc sử dụng chức năng này với các chương trình đơn giản chỉ gồm vài dòng lệnh.

### I.2.4. Môi trường soạn thảo của Code::Blocks

Trong quá trình soạn thảo mã nguồn C++ trong Code::Blocks, bạn có thể sử dụng các chức năng soạn thảo gần tương tự như bất kỳ phần mềm soạn thảo nào (MS Word chẳng hạn).

- Các chức năng quản lý tệp chương trình nằm trong menu File gồm: New Project, Open Project, Save, Save As, Close Project, Print, Exit, ...
- Các chức năng soạn thảo văn bản: Undo, Redo, Copy, Cut, Paste, Select, Select all, ... với hệ thống phím tắt tương tự như MS Word, ngoài ra bạn có thể tham khảo thêm các chức năng khác trong menu Edit.
- Các chức năng Tìm kiếm và Thay thế trong menu Search: Find, Replace, Goto line, ...
- Các chức năng biên dịch, chạy chương trình trong menu Build: Build, Run, Compile
- Các chức năng chạy dò và gỡ lỗi trong menu Debug

Bạn có thể tham khảo thêm về các chức năng này trên Internet.

### I.2.5. Viết chương trình C++ đầu tiên

Một chương trình C++ có thể được đặt trong một hoặc nhiều file văn bản khác nhau. Mỗi file văn bản chứa một số phần nào đó của chương trình (trong Code::Blocks, mỗi chương trình được tổ chức thành một dự án - Project). Với những chương trình đơn giản và ngắn thường chỉ cần đặt chúng trên một file.

Một chương trình gồm nhiều hàm, mỗi hàm phụ trách một công việc khác nhau của chương trình. Đặc biệt trong các hàm này, có một hàm duy nhất có tên hàm là *main()*. Khi chạy chương trình, các câu lệnh trong hàm *main()* sẽ được thực hiện đầu tiên. Trong hàm *main()* có thể có các câu lệnh gọi đến các hàm khác khi cần thiết, và các hàm này khi chạy lại có thể gọi đến các hàm khác nữa đã được viết trong chương trình (trừ việc gọi quay lại hàm *main()*). Sau khi thực hiện đến lệnh cuối cùng của hàm *main()* chương trình sẽ kết thúc.

Thông thường, một chương trình gồm có các phần sau:

- *Phần khai báo các tệp tiêu đề*: khai báo tên các tệp chứa những thành phần có sẵn (như các hằng chuẩn, kiểu chuẩn và các hàm chuẩn) mà NSD dùng trong chương trình.
- *Phần khai báo các kiểu dữ liệu, các biến, hằng...*: do NSD định nghĩa và được dùng chung trong toàn bộ chương trình.
- *Danh sách các khai báo nguyên mẫu hàm của chương trình*: do NSD viết, bao gồm cả hàm *main()*. Chi tiết về hàm sẽ được đề cập đến trong chương 4.

Dưới đây là một đoạn chương trình đơn giản chỉ gồm một hàm *main()*. Nhiệm vụ của chương trình là in ra màn hình dòng chữ: *Hello World*.

```
#include <iostream>       // khai báo tệp tiêu đề iostream để dùng cout<<
using namespace std;
main()
{
    cout << "Hello World\n";
}
```

Dòng đầu là câu lệnh *#include* báo cho trình biên dịch phải gộp mã từ tập tin tiêu đề có tên là *iostream* vào chương trình của bạn trước khi tạo file thi hành. Tệp tiêu đề *iostream* đi kèm với trình biên dịch của bạn và cho phép bạn thực hiện lệnh nhập/xuất dữ liệu (Input/Output). Bằng các sử dụng các lệnh *#include* bạn có thể gộp các tệp tiêu đề vào chương trình của bạn. Qua đó, bạn có quyền truy cập vào rất nhiều hàm được cung cấp sẵn bởi trình biên dịch.

Bất cứ khi nào cần truy cập vào các hàm cơ bản, ta sẽ phải bao gồm tập tin tiêu đề cho phép truy cập các hàm đó, hầu hết các hàm chúng ta cần hiện nay đều ở trong tệp tiêu đề *iostream*, và bạn sẽ thấy nó ở đầu trong tất cả các chương trình, và gần như mọi chương trình bạn viết sẽ bắt đầu với một hoặc nhiều lệnh *include*. Tiếp theo là dòng:

```
using namespace std;
```

Đây là mã soạn có sẵn mỗi khi bạn mở một dự án mới trong Code::Blocks và nó luôn xuất hiện trong hầu như trong tất cả các chương trình C++. Chỉ cần sử dụng nó ở phía trên cùng của tất cả các chương trình của bạn, theo sau các lệnh *include*. Khai báo này làm cho việc viết các lệnh trong chương trình ngắn hơn so với nguyên bản của lệnh. Chúng ta sẽ nói chuyện đó sau, bây giờ chỉ cần bạn đừng quên viết nó.

Chú ý, dòng lệnh này kết thúc bằng dấu chấm phẩy ;. Dấu chấm phẩy là một phần trong cú pháp của C++. Nó nói với trình biên dịch rằng đây là vị trí kết thúc một lệnh. Dấu chấm phẩy được sử dụng để kết thúc hầu hết các lệnh trong C++. Thiếu dấu chấm phẩy là một trong những lỗi phổ biến nhất của các lập trình viên mới. Nếu chương trình của bạn gặp lỗi khi biên dịch, hãy chắc rằng bạn đã không thiếu một dấu chấm phẩy nào đó. Bất cứ khi nào giới thiệu khái niệm mới, tôi sẽ nói cho bạn biết có cần phải sử dụng một dấu chấm phẩy hay không.

Tiếp theo là hàm *main()*, hiện giờ bạn chỉ cần biết đây là nơi chương trình bắt đầu:

```
main()
```

Dòng tiếp theo của chương trình có vẻ kỳ lạ, với ký hiệu << buồn cười.

```
cout << "Hello World\n";
```

C++ sử dụng đối tượng *cout* (phát âm là "C out") để hiển thị thông tin ra màn hình. Việc sử dụng *cout* là lý do cần khai báo *#include* tệp tin tiêu đề *iostream*. *cout* sử dụng ký hiệu <<, gọi là

"phép chèn", để chỉ những gì sẽ được in ra màn hình. Trong chương trình này, *cout* in ra chuỗi văn bản mà chúng ta cung cấp.

Chuỗi văn bản của bạn được bao trong cặp dấu nháy kép. Cặp dấu nháy này báo cho trình biên dịch là bạn muốn in ra chuỗi văn bản viết giữa các dấu nháy kép, ngoại trừ một số chuỗi đặc biệt. Dãy \n là một trong các chuỗi đặc biệt, nó được coi như ký tự đại diện cho một dòng mới, giống như ta gõ phím Enter (chúng ta sẽ nói về điều này chi tiết hơn ở phần sau). Chuỗi \n di chuyển con trỏ trên màn hình của bạn sang dòng tiếp theo. Đôi khi bạn cũng thấy giá trị đặc biệt *endl* được sử dụng thay cho chuỗi \n: Lệnh *cout << "Hello World"<<endl* cơ bản là tương đương với lệnh trong chương trình mẫu.

Cuối cùng, hãy chú ý dấu chấm phẩy ; chúng ta cần phải đặt nó ở cuối dòng lệnh.

Dấu đóng ngoặc nhọn } thông báo kết thúc hàm *main()* ở đây. Bạn hãy biên dịch chương trình này và chạy nó.

Bạn có thể thử sửa chương trình làm cho nó in ra màn hình thêm một vài chuỗi văn bản khác, bằng cách thêm vào chương trình các lệnh *cout*<<.

## I.4. Các lệnh xuất và nhập dữ liệu của C++

C++ cung cấp một tập lớn các đối tượng xuất / nhập dữ liệu. Tuy nhiên, giáo trình này viết với mục đích dạy học lập trình cơ bản và hướng cấu trúc cho HS phổ thông, chúng tôi chỉ giới thiệu một số lệnh xuất / nhập thiết yếu. Bạn đọc có thể tự tìm hiểu thêm qua các tài liệu khác.

### 1.4.1. Lệnh cin nhập dữ liệu vào từ bàn phím

Để nhập dữ liệu vào cho các biến có tên *var1, var2, var3* chúng ta sử dụng câu lệnh:

```
cin >> var1 ;
cin >> var2 ;
cin >> var3 ;
```

hoặc:

```
cin >> var1 >> var2 >> var3 ;
```

Các biến *var1, var2, var3* được sử dụng để lưu trữ các giá trị NSD nhập vào từ bàn phím. Hiển nhiên có thể nhập nhiều biến hơn bằng cách viết tiếp tiếp các tên biến vào bên phải các dấu >> của câu lệnh.

Khi chạy chương trình, nếu gặp các câu lệnh trên, chương trình sẽ "tạm dừng" để chờ NSD nhập dữ liệu vào từ bàn phím cho các biến. Sau khi NSD nhập xong dữ liệu (và bấm ENTER), chương trình sẽ tiếp tục chạy tới câu lệnh tiếp theo.

Cách thức nhập dữ liệu của NSD phụ thuộc vào loại giá trị của biến cần nhập mà ta gọi là kiểu dữ liệu, ví dụ nhập một số khác với nhập một chuỗi kí tự. Giả sử cần nhập độ dài hai cạnh của một hình chữ nhật, trong đó cạnh dài được qui ước bởi tên biến *cd* và chiều rộng được qui ước bởi tên biến *cr*. Câu lệnh nhập sẽ như sau:

```
cin >> cd >> cr ;
```

Khi máy dừng chờ nhập dữ liệu NSD sẽ gõ giá trị cụ thể của các chiều dài, rộng theo đúng thứ tự trong câu lệnh. Các giá trị này cần cách nhau bởi ít nhất một dấu trắng (ta qui ước gọi dấu trắng là một trong 3 loại ký tự nhập bởi các phím: *dấu cách, phím tab hoặc phím Enter*. Các giá trị NSD nhập vào cũng được hiển thị trên màn hình để dễ theo dõi.

*Ví dụ 1.1*: Chương trình tính chu vi và diện tích hình chữ nhật.

```
#include <iostream>
using namespace std;
main()
{
    int cd, cr;
    cout << "Nhap hai canh hinh chu nhat: ";
    cin >> cd >> cr;
    cout << "Dien tich = "<<cd*cr<<" Chu vi = "<<(cd+cr)*2;
}
```

### 1.4.2. Các phương thức nhập dữ liệu của cin

Toán tử nhập >> chủ yếu làm việc với dữ liệu kiểu số. Để nhập kí tự hoặc xâu kí tự, C++ cung cấp các phương thức (hàm) của *cin* sau đây:

1) **`cin.get()`:** Hàm trả về một kí tự (kể cả dấu cách, dấu ↵). Ví dụ:

```
char ch;
ch = cin.get();
```

- nếu nhập AB↵, *ch* nhận giá trị 'A', trong *cin* còn B↵.
- nếu nhập A↵, *ch* nhận giá trị 'A', trong *cin* còn ↵.
- nếu nhập ↵, *ch* nhận giá trị '↵', trong *cin* rỗng.

2) **`cin.get(ch)`:** Hàm nhập kí tự cho *ch* và trả lại một tham chiếu tới *cin*. Do hàm trả lại tham chiếu tới *cin* nên có thể viết các phương thức nhập này liên tiếp trên một đối tượng *cin*. Ví dụ:

```
char c, d;
cin.get(c).get(d);
```

nếu nhập ABC↵ thì *c* nhận giá trị 'A' và *d* nhận giá trị 'B'. Trong cin còn 'C↵'.

3) **`cin.get(s, n, fchar)`:** Phương thức này dùng để nhập chuỗi kí tự từ *cin* vào biến chuỗi *s*, tính từ kí tự đầu tiên trong *cin* cho đến khi đã đủ *n-1* kí tự hoặc gặp kí tự báo kết thúc *fchar*. Kí tự kết thúc này được ngầm định là dấu xuống dòng nếu biến *fchar* bị bỏ qua trong danh sách đối. Tức có thể viết câu lệnh trên dưới dạng *cin.get(s, n),* khi đó xâu *s* sẽ nhận dãy kí tự nhập cho đến khi đủ *n-1* kí tự hoặc đến khi NSD kết thúc nhập (bằng phím Enter).

*Chú ý* :

- Lệnh sẽ tự động gán dấu kết thúc xâu ('\0') vào cho xâu *s* sau khi nhập xong.
- Các lệnh có thể viết nối nhau, ví dụ: *cin.get(s1, n1).get(s2,n2);*
- Kí tự kết thúc *fchar* (hoặc ↵) vẫn nằm lại trong *cin*. Điều này có thể làm trôi các lệnh *get()* tiếp theo.

4) **`cin.ignore(n,fchar)`:** Phương thức này dùng để đọc và loại bỏ *n* kí tự còn trong bộ đệm (dòng nhập cin) hoặc cho đến khi gặp ký tự *fchar*. Để không bị hiệu ứng trôi lệnh cho trong bộ đệm bàn phím còn chứa ký tự thừa, ta có thể dùng lệnh này.

*Ví dụ*: `cin.ignore(10,' ')` sẽ loại bỏ 9 ký tự còn lưu trong bộ đệm hoặc cho đến khi gặp ký tự dấu cách.

5) **cin.getline(s,n,ch):** Phương thức này hoạt động hoàn toàn tương tự phương thức *cin.get(s, n, fchar)*, tuy nhiên nó có thể khắc phục "lỗi Enter" của phương thức trên. Cụ thể, sau khi gán nội dung nhập cho biến *s, cin.getline()* sẽ xóa kí tự Enter khỏi bộ đệm và do vậy NSD không cần phải sử dụng thêm các câu lệnh phụ trợ (*cin.get(), cin.ignore(1)*) để loại Enter ra khỏi bộ đệm.

Ví dụ đoạn lệnh sau cho phép nhập một số nguyên *x* (bằng toán tử >>) và một kí tự *c* (bằng phương thức *cin.get(c)*):

```
int x;
char c;
cin >> x; cin.ignore(1);
cin.get(c);
```

*Ví dụ 1.2*: chương trình minh họa sử dụng lệnh *cin.get()* và *cin.ignore():* Lấy chữ cái đầu trong chuỗi họ và tên.

```
#include <iostream>
using namespace std;
main ()
{
    char first, last;
    cout << "Nhap ho va ten: "; // ví dụ LE NA
    first = cin.get();      // lay 1 chu cai dau L
    cin.ignore(256,' ');    // bo qua cho den khi gap dau cach
    last = std::cin.get(); // Lay mot ky tu N
    cout << "Cac chu cai dau la: " << first <<" "<< last << '\n';
}
```

Nếu nhập: `Le Na↵`

Chương trình in ra: `Cac chu cai dau la L N`

### 1.4.3. Lệnh cout xuất dữ liệu ra màn hình

Để in giá trị của các biểu thức ra màn hình ta dùng câu lệnh sau:

```
cout << bt_1 ;
cout << bt_2 ;
cout << bt_3 ;
```

hoặc:

```
cout << bt_1 << bt_2 << bt_3 ;
```

Cũng giống câu lệnh nhập *cin*, ở đây chúng ta cũng có thể mở rộng lệnh in ra màn hình với nhiều biểu thức. Câu lệnh trên cho phép in giá trị của các biểu thức *bt_1, bt_2, bt_3*. Các giá trị này có thể là tên biến, kết quả biểu thức hoặc chuỗi ký tự.

*Ví dụ 1.3*: In câu "*Chieu dai la* " và số *23* và tiếp theo là chữ " *met*", ta sử dụng 3 lệnh sau:

```
cout << "Chieu dai la " ;
cout << 23 ;
cout << " met";
```

hoặc có thể chỉ bằng 1 lệnh:

```
cout << "Chieu dai la 23 met" ;
```

Trường hợp chưa biết giá trị cụ thể của chiều dài, chỉ biết hiện tại giá trị này đã được lưu trong biến *cd* (ví dụ bạn nhập số 23 từ bàn phím bởi câu lệnh *cin >> cd* trước đó) và ta cần biết giá trị này là bao nhiêu thì có thể sử dụng câu lệnh:

```
cout << "Chieu dai la " << cd << " met" ;
```

Một giá trị cần in không chỉ là một biến như *cd, cr*, ... mà còn có thể là một biểu thức, ví dụ yêu cầu máy in ra diện tích và chu vi của hình chữ nhật khi đã biết *cd* và *cr* như *ví dụ 1.1*.

### 1.4.4. Các phương thức và cờ định dạng xuất dữ liệu

#### *a. Các phương thức định dạng xuất dữ liệu*

*1)* **`cout.width(n)`**: Chỉ định độ rộng in ra màn hình là *n*

Phương thức này cho phép in ra các giá trị với độ rộng *n* cột trên màn hình bằng lệnh *cout*<< liền sau đó. Nếu *n* bé hơn độ rộng thực sự của giá trị, thì in giá trị với số cột bằng với độ rộng thực sự. Nếu *n* lớn hơn độ rộng thực, thì in giá trị canh theo lề phải, và để trống các cột thừa phía trước giá trị được in.

Phương thức này chỉ có tác dụng với giá trị cần in ngay sau nó. Ví dụ:

```
int a = 12; b = 345;      // độ rộng thực của a là 2, của b là 3
cout << a;             // chiếm 2 cột màn hình
cout.width(7);         // đặt độ rộng giá trị in tiếp theo là 7
cout  << b;            // b in trong 7 cột với 4 dấu cách đứng trước
```

Kết quả in ra sẽ là: `12     345`

2) **`cout.precision(n)`**: Chỉ định độ chính xác cần in với các giá trị thực (số chữ số sau dấu chấm thập phân).

Phương thức này yêu cầu các số thực in ra sau đó sẽ có *n-1* chữ số lẻ. Các số thực trước khi in ra sẽ được làm tròn đến chữ số lẻ thứ *n-1*.

Phương thức này có tác dụng cho đến khi gặp một chỉ định mới. Ví dụ:

```
int a = 12.3; b = 345.678;    // độ rộng thực của a là 4, của b là 7
cout << a;                 // chiếm 4 cột màn hình
cout.width(10);               // đặt độ rộng giá trị in tiếp theo là 10
cout.precision(3);            // đặt độ chính xác đến 2 số lẻ
cout  << b;                // b in trong 10 cột với 4 dấu cách đứng trước
```

Kết quả in ra sẽ là: `12.3     345.68`

#### *b. Các cờ định dạng in*

Một số các qui định về định dạng thường được gắn liền với các giá trị gọi là "cờ". Thông thường nếu một số định dạng được sử dụng trong suốt quá trình chạy chương trình hoặc trong một khoảng thời gian dài trước khi gỡ bỏ thì ta "bật" các cờ tương ứng với nó. Các cờ được bật sẽ có tác dụng cho đến khi cờ định dạng khác được bật. Các cờ được khai báo trong tệp tiêu đề <*iostream*>.

Để bật/tắt các cờ ta sử dụng các phương thức sau:

1) **`cout.setf(danh sách cờ)`**: Bật các cờ trong danh sách

2) **`cout.unsetf(danh sách cờ)`**: Tắt các cờ trong danh sách

Các cờ trong danh sách được viết phân cách bởi phép toán hợp bit (|). Ví dụ lệnh *cout.setf(ios::left | ios::scientific)* sẽ bật các cờ *ios::left* và *ios::scientific*. Phương thức *cout.unsetf(ios::right | ios::fixed)* sẽ tắt các cờ *ios::right | ios::fixed*.

Dưới đây là danh sách một số cờ cho trong <*iostream*>.

*1. Cờ canh lề*

- **ios::left**: nếu bật thì giá trị in nằm bên trái vùng in ra (kí tự độn phía nằm sau).
- **ios::right**: giá trị in nằm bên phải vùng in ra (kí tự độn nằm phía trước), đây là cờ ngầm định nếu ta không sử dụng cờ cụ thể thì cờ này có hiệu lực.

*Ví dụ 1.4*:

```
int a = 12.3; b = -345.678; // độ rộng thực của a là 4, của b là 8
cout << a;                  // chiếm 4 cột màn hình
cout.width(10);             // đặt độ rộng giá trị in tiếp theo là 10
cout.precision(3);          // đặt độ chính xác đến 2 số lẻ
cout.setf(ios::left) ;      // bật cờ ios::left
cout << b;                  // kết qủa: 12.3-345.68
cout.setf(ios::right) ;     // bật cờ ios::right
cout << b;                  // kết qủa: 12.3   -345.68
```

*2) Cờ định dạng số nguyên*

- **ios::dec** : in số nguyên dưới dạng thập phân (ngầm định)
- **ios::oct** : in số nguyên dưới dạng cơ số 8
- **ios::hex** : in số nguyên dưới dạng cơ số 16

*3) Nhóm cờ định dạng số thực*

- **ios::fixed**: in số thực dạng dấu phảy tĩnh (ngầm định)
- **ios::scientific**: in số thực dạng dấu phảy động
- **ios::showpoint**: in đủ n chữ số lẻ của phần thập phân, nếu tắt (ngầm định) thì không in các số 0 cuối của phần thập phân.

*Ví dụ*: Độ chính xác được đặt với 3 số lẻ bằng lệnh *cout.precision(4)*

+ nếu *fixed* bật + *showpoint* bật :

| | | |
|---|---|---|
| 123.2500 | được in thành | 123.250 |
| 123.2599 | được in thành | 123.260 |
| 123.2 | được in thành | 123.200 |

+ nếu *fixed* bật + *showpoint* tắt :

| | | |
|---|---|---|
| 123.2500 | được in thành | 123.25 |
| 123.2599 | được in thành | 123.26 |
| 123.2 | được in thành | 123.2 |

+ nếu *scientific* bật + *showpoint* bật :

| | | |
|---|---|---|
| 12.3 | được in thành | 1.230e+01 |
| 2.32599 | được in thành | 2.326e+00 |
| 324 | được in thành | 3.240e+02 |

+ nếu *scientific* bật + *showpoint* tắt :

| | | |
|---|---|---|
| `12.3` | được in thành | `1.23e+01` |
| `2.32599` | được in thành | `2.326e+00` |
| `324` | được in thành | `3.24e+02` |

*4) Cờ định dạng hiển thị*

- **`ios::showpos`**: nếu tắt (ngầm định) thì không in dấu cộng (+) trước số dương. Nếu bật trước mỗi số dương sẽ in thêm dấu +.
- **`ios::showbase`**: nếu bật sẽ in số 0 trước các số nguyên hệ 8 và in 0x trước số hệ 16. Nếu tắt (ngầm định) sẽ không in.
- **`ios::uppercase`**: nếu bật thì các kí tự biểu diễn số trong hệ 16 (A..F) sẽ viết hoa, nếu tắt (ngầm định) sẽ viết thường.

### *c. Một số hàm định dạng in dữ liệu trong thư viện iomanip*

Để sử dụng các hàm này cần khai báo tệp tiêu đề *#include <iomanip>*

```
setw(n)              // tương tự cout.width(n)
setprecision(n)      // tương tự cout.precision(n)
setiosflags(l)       // tương tự cout.setf(l)
resetiosflags(l)     // tương tự cout.unsetf(l)
```

## 1.6. Câu hỏi và Bài tập

1. Những tên gọi nào sau đây là hợp lệ:

| | | |
|---|---|---|
| A. x | E. so-dem | I. Radius |
| B. 123variabe | F. RADIUS | J. nam2000 |
| C. tin_hoc | G. one.0 | |
| D. toan tin | H. number# | |

2. Khi chương trình thực hiện xong sẽ trả về hệ thống giá trị nào dưới đây?

| | |
|---|---|
| A. -1 | C. 0 |
| B. 1 | D. Không có giá trị nào. |

2. Hàm nào luôn phải có trong các chương trình C++?

| | | | |
|---|---|---|---|
| A. start() | B. system() | C. main() | D. program() |

3. Cặp ký hiệu nào cho biết bắt đầu và kết thúc khối lệnh?

| | |
|---|---|
| A. { } | C. BEGIN and END |
| B. -> and <- | D. ( and ) |

4. Ký hiệu nào cho biết đã đến cuối một lệnh trong C++?

| | | | |
|---|---|---|---|
| A. . | B. ; | C. : | D. ' |

5. Ghi chú nào dưới đây là đúng trong chương trình C++?

| | |
|---|---|
| A. */ Comments */ | B. ** Comment ** |

C. /* Comment */ D. { Comment }

6. Tệp tin tiêu đề nào cần khai báo để sử dụng phương thức *cout*?

A. stream C. iostream

B. nothing, it is available by default D. using namespace std;

7. Tìm các lỗi cú pháp trong chương trình sau:

```
#include (iostream.h)
main();
{
    cout << 'Day la chuong trinh: Gptb1.\nXin chao cac ban';
    cin.get();
}
```

8. Viết chương trình in nội dung một bài thơ nào đó.

9. Viết chương trình in ra 4 dòng, 2 cột gồm các số sau và gióng cột:

| | | |
|---|---|---|
| thẳng theo lề trái | 0.63 | 64.1 |
| thẳng theo lề phải | 12.78 | -11.678 |
| thẳng theo dấu chấm thập phân | -124.6 | 59.002 |
| | 65.7 | -1200.654 |

10. Hãy viết và chạy các chương trình trong các ví dụ của chương I.

11. Chương trình sau khai báo 5 biến kí tự a, b, c, d, e và một biến số nam. Hãy điền thêm các câu lệnh vào các dòng … để chương trình thực hiện nhiệm vụ sau:

- Nhập giá trị cho biến nam
- Nhập giá trị cho các biến kí tự a, b, c, d, e.
- In ra màn hình dòng chữ được ghép bởi 5 kí tự đã nhập và chữ "năm" sau đó in số đã nhập (nam). Ví dụ nếu 5 chữ cái đã nhập là 'H', 'A', 'N', 'O', 'I' và nam được nhap là 2000, thì màn hình in ra dòng chữ: HANOI năm 2000.

# CHƯƠNG II. KIỂU DỮ LIỆU CHUẨN, BIỂU THỨC, CÂU LỆNH VÀ CẤU TRÚC ĐIỀU KHIỂN

## II.1. Kiểu dữ liệu chuẩn

Dưới đây là bảng tóm tắt các kiểu chuẩn và các thông số của nó được sử dụng trong C++.

| Kiểu dữ liệu | Tên kiểu | Kích thước/ precision |
|---|---|---|
| Ký tự | **char** | 8 bit |
| | **char16_t** | 16 bit |
| | **char32_t** | Không nhỏ hơn kiểu char16_t. Tối thiểu là 32 bit. |
| | **wchar_t** | Có thể biểu diễn cho các bộ ký tự được hỗ trợ lớn nhất. |
| Số nguyên có dấu | **signed char** | 8 bit. |
| | ***signed* short *int*** | 16 bits. |
| | ***signed* int** | Không nhỏ hơn short. Tối thiểu 16 bit. |
| | ***signed* long *int*** | Không nhỏ hơn int. Tối thiểu 32 bit. |
| | ***signed* long long *int*** | Không nhỏ hơn long. Tối thiểu 64 bit. |
| Số nguyên không dấu | **unsigned char** | Tương tự các số nguyên có dấu cùng kiểu |
| | **unsigned short *int*** | |
| | **unsigned *int*** | |
| | **unsigned long *int*** | |
| | **unsigned long long *int*** | |
| Số thực (dấu chấm động) | **float** | 4 byte (1.2E-38 -> 3.4E+38), 6 chữ số thập phân |
| | **double** | 8 byte (2.3E-308 -> 1.7E+308), 15 chữ số thập phân |
| | **long double** | 10 byte (3.4E-4932 -> 1.1E+4932), 19 chữ số thập phân |
| Lô gic | **bool** | |
| Rỗng | **void** | Không lưu trữ |
| Con trỏ rỗng | **decltype(nullptr)** | |

(*Những từ in nghiêng trong tên kiểu không viết cũng được*)

*Ví dụ 2.1*: Chương trình sau minh họa việc dùng biến.

```
#include <iostream>
using namespace std;
main ()
{
    int a =5, b = 2, result;
    a = a + 1;
    result = a - b;
    cout << result;
}
```

## II.2. Khai báo và sử dụng hằng

### II.2.1. Các loại hằng

Hằng là một giá trị cố định nào đó được sử dụng trong chương trình. Ví dụ: 3 (hằng nguyên), 'A' (hằng kí tự), 5.0 (hằng thực), "Ha Noi" (hằng xâu kí tự). Một giá trị có thể được hiểu dưới nhiều kiểu khác nhau, do vậy khi viết hằng, ta cũng cần có dạng viết thích hợp.

***a. Hằng nguyên***

- kiểu short, int: 3, -7, ...
- kiểu unsigned: 3, 123456, ...
- kiểu long, long int: 3l, -7lu, 123456ul, ... (viết l vào cuối mỗi giá trị)

Các cách viết trên là thể hiện của số nguyên trong hệ thập phân, ngoài ra chúng còn được viết dưới các hệ đếm khác như hệ cơ số 8 hoặc hệ cơ số 16. Một số nguyên trong cơ số 8 phải viết số 0 ở đầu, trong cơ số 16 phải viết 0x ở đầu. Ví dụ số 65 trong cơ số 8 là 101 và trong cơ số 16 là 41, do đó 3 cách viết 65, 0101, 0x41 cùng biểu diễn giá trị số 65.

***b. Hằng thực***

Một số thực có thể được khai báo dưới dạng kiểu float hoặc double và các giá trị của nó có thể được viết dưới một trong hai dạng.

*+ Dạng dấu phảy tĩnh*

Theo cách viết thông thường. Ví dụ: 3.0, -7.0, 3.1416, ...

*+ Dạng dấu phảy động*

Một số thực x có thể được viết dưới dạng: *men* hoặc *mEn*, trong đó *m* được gọi là phần định trị, *n* gọi là phần bậc (hay mũ). Số *men* biểu thị giá trị x = m x 10n. Ví dụ số π = 3.1416 có thể được viết:

```
    π = … = 0.031416e2 = 0.31416e1 = 3.1416e0 = 31.416e1 = 314.16e2 = …
vì  π = 0.031416 x 102 = 0.31416 x 101 = 3.1416 x 100 = …
```

Như vậy một số *x* có thể được viết dưới dạng *mEn* với nhiều giá trị m, n khác nhau, phụ thuộc vào phần nguyên *m* và phần thập phân *n* của số đó. Do vậy cách viết này gọi là dạng dấu phảy động.

***c. Hằng kí tự***

*+ Cách viết :* Có 2 cách viết hằng kí tự:

- Với các kí tự có mặt chữ ta sử dụng cách viết thông thường gõ ký tự đó giữa 2 dấu nháy đơn: 'A', '3', ' ' (dấu cách) ... hoặc sử dụng mã ASCII của chúng. Ví dụ các giá trị tương ứng của các kí tự trên là 65, 51 và 32.

- Với một số kí tự không có mặt chữ ta buộc phải dùng mã ASCII của chúng, mã 27 thay cho phím Escape, mã 13 thay cho phím Enter ...

Để biểu diễn kí tự bằng mã ASCII ta có thể viết trực tiếp (không dùng cặp dấu nháy đơn) giá trị đó dưới dạng hệ số 10 (như trên) hoặc đặt chúng vào cặp dấu nháy đơn, trường hợp này chỉ dùng cho giá trị viết dưới dạng hệ 8 hoặc hệ 16 theo mẫu sau:

'\kkk': không quá 3 chữ số trong hệ 8. Ví dụ '\11' biểu diễn kí tự có mã 9.

'\xkk': không quá 2 chữ số trong hệ 16. Ví dụ '\x1B' biểu diễn kí tự có mã 27.

Tóm lại, một kí tự có thể có nhiều cách viết, chẳng hạn 'A' có giá trị là 65 (hệ 10) hoặc 101 (hệ 8) hoặc 41 (hệ 16), do đó kí tự 'A' có thể viết bởi một trong các dạng sau:

65, 0101, 0x41 hoặc 'A' , '\101' , '\x41'

Tương tự, dấu kết thúc xâu có giá trị 0 nên có thể viết bởi 0 hoặc '\0' hoặc '\x0'. Cách viết '\0' được dùng thông dụng nhất.

*+ Một số hằng ký tự thông dụng*

Đối với một số hằng kí tự thường dùng nhưng không có mặt chữ tương ứng, hoặc các kí tự được dành riêng với nhiệm vụ khác, khi đó thay vì phải nhớ giá trị của chúng ta có thể viết theo qui ước sau:

```
'\n'    :   biểu thị kí tự xuống dòng (tương đương với endl)
'\t'    :   kí tự tab
'\\'    :   dấu \
'\?'    :   dấu chấm hỏi ?
'\''    :   dấu nháy đơn '
'\"'    :   dấu nháy kép "
'\kkk'  :   kí tự có mã là kkk trong hệ 8
'\xkk'  :   kí tự có mã là kk trong hệ 16
```

*Ví dụ 2.2:*

```
cout << "Hôm nay trời\t nắng\n" ;
```

sẽ in ra màn hình dòng chữ "*Hôm nay trời*" sau đó bỏ một khoảng cách bằng một tab (khoảng 8 dấu cách) rồi in tiếp chữ "*nắng*" và cuối cùng con trỏ trên màn hình sẽ nhảy xuống đầu dòng mới.

Dấu cách (phím spacebar) không có mặt chữ, ví dụ trong giáo trình này dấu cách (có giá trị là 32) và được viết là ' ' hoặc '\040' hoặc '\x20'.

### *d. Hằng xâu kí tự*

Hằng xâu ký tự là dãy kí tự bất kỳ đặt giữa cặp dấu nháy kép. Ví dụ: "*Lop 10Tin*", "*12Anh*", "*A*", " ", "" là các hằng xâu kí tự. Chú ý:

+ Phân biệt cách viết '*A*' và "*A*", cùng biểu diễn chữ cái *A* nhưng '*A*' là hằng kí tự còn "*A*" hằng xâu kí tự (do vậy chúng được đối xử khác nhau).

+ Không được viết '' (2 dấu nháy đơn liền nhau) vì không kí tự "rỗng". Xâu rỗng là "" (2 dấu nháy kép liền nhau).

## II.2.2. Khai báo hằng

Một giá trị cố định (hằng) được sử dụng nhiều lần trong chương trình thường sẽ thuận lợi hơn nếu ta đặt cho nó một tên gọi, thao tác này được gọi là khai báo hằng.

Để khai báo hằng ta dùng các câu khai báo sau:

```
<#define>  tên_hằng  giá_trị_hằng ;
```

hoặc:

```
<const>  tên_hằng = giá_trị_hằng ;
```

*Ví dụ 2.3*:

```
#define sohs 50 ;
#define MAX 100 ;
const sohs = 50 ;
```

Khai báo trên chưa nói kiểu dữ liệu của nó (có thể gây sự nhập nhằng khi sử dụng). Ta nên khai báo rõ ràng hơn bằng cách thêm tên kiểu trước tên hằng trong khai báo *const*, các hằng khai báo như vậy gọi là hằng có kiểu.

*Ví dụ 2.4:*

```
const int sohs = 50 ;
const float nhiet_do_soi = 100.0 ;
```

## II.3. Khai báo và sử dụng biến

### II.3.1. Khai báo biến

Một biến trong chương trình là một số ô nhớ liên tiếp trong bộ nhớ, dùng để lưu trữ dữ liệu (vào, ra hay kết quả trung gian) trong quá trình hoạt động của chương trình và được gắn với một tên do NSD khai báo. Để sử dụng biến, NSD khai báo: tên biến và kiểu của dữ liệu chứa trong biến. Nguyên tắc: *chỉ có các dữ liệu cùng kiểu với nhau mới được phép làm toán với nhau*. Do đó, khi đề cập đến một kiểu dữ liệu chuẩn, ta sẽ xét đến các yếu tố sau:

- *Tên kiểu giá trị*: là một từ dành riêng để chỉ định kiểu của dữ liệu.
- *Số byte lưu trữ trong bộ nhớ*: Thông thường số byte này phụ thuộc vào các trình biên dịch và hệ thống máy khác nhau, ở đây ta chỉ xét đến hệ thống máy PC thông dụng hiện nay.
- *Miền giá trị của kiểu*: Quy định một đơn vị dữ liệu thuộc kiểu này sẽ có thể chứa giá trị trong phạm vi nào. NSD cần nhớ đến miền giá trị này để khai báo kiểu cho các biến cần sử dụng một cách thích hợp.

Biến có thể lưu giá trị dữ liệu ban đầu (khởi tạo), giá trị trung gian trong quá trình tính toán hoặc giá trị kết quả cuối cùng. Khác với hằng, giá trị của biến có thể thay đổi trong quá trình làm việc bằng các lệnh đọc vào từ bàn phím hoặc lệnh gán.

Mọi biến phải được khai báo trước khi sử dụng. Một khai báo biến sẽ cho chương trình biết về một biến mới gồm có: *tên của biến*, *kiểu của biến*. Thông thường với nhiều NNLT, tất cả các biến phải được khai báo ngay từ đầu chương trình hay đầu của hàm, tuy nhiên để thuận tiện C++ cho phép khai báo biến ngay bên trong chương trình hoặc hàm, ở vị trí bất kỳ. NSD thấy cần sử dụng biến mới, họ có quyền khai báo và sử dụng nó từ vịtrí đó trở đi.

Cú pháp khai báo biến gồm *tên kiểu, tên biến* và có thể có hay không khởi tạo giá trị ban đầu cho biến. Để khởi tạo hoặc thay đổi giá trị của biến ta dùng lệnh gán (=).

***a. Khai báo không khởi tạo***

```
<tên_kiểu>  tên_biến_1 ;
<tên_kiểu>  tên_biến_2 ;
<tên_kiểu>  tên_biến_3 ;
```

Nhiều biến cùng kiểu có thể được khai báo trên cùng một dòng:

```
<tên_kiểu> tên_biến_1, tên_biến_2, tên_biến_3;
```

*Ví dụ 2.5:*

```
main()
{
    int i, j ;      // khai báo 2 biến i, j có kiểu nguyên
    float x ;       // khai báo biến thực x
    char c, d[100] ;   // biến kí tự c, xâu d chứa tối đa 100 kí tự
    unsigned int u ;   // biến nguyên không dấu u
    …
}
```

***b. Khai báo có khởi tạo***

Trong câu lệnh khai báo, các biến có thể được gán ngay giá trị ban đầu bởi phép toán gán (=) theo cú pháp:

**<tên_kiểu>  tbiến_1 = gt_1, tbiến_2 = gt_2, tbiến_3 = gt_3;**

trong đó các giá trị gt_1, gt_2, gt_3 có thể là các hằng, biến hoặc biểu thức.

*Ví dụ 2.6:*

```
const int n = 10 ;
main()
{
    int i = 2, j , k = n + 5;
    float eps = 1.0e-6 ;
    char c = 'Z';
    char d[100] = "Tin học";
    ...
}
```

***c. Phạm vi hiệu lực của biến***

Chương trình C++ là một tập hợp các hàm, các câu lệnh cũng như các khai báo. Phạm vi hiệu lực của một biến là nơi mà biến có tác dụng. Tức là biến thuộc hàm hay khối lệnh nào? câu lệnh nào được phép sử dụng biến đó? Một biến xuất hiện trong chương trình có thể được sử dụng bởi hàm này nhưng không sử dụng được bởi hàm khác hoặc có thể sử dụng bởi cả hai, điều này phụ thuộc chặt chẽ vào vị trí nơi biến được khai báo. Một nguyên tắc đầu tiên là biến sẽ có tác dụng kể từ vị trí nó được khai báo cho đến hết khối lệnh chứa nó. Chi tiết cụ thể sẽ được trình bày trong chương 4.

### II.3.2. Gán giá trị cho biến

***a. Phép gán thông thường***

*Cú pháp:*

**<tên_biến> = <biểu thức ;>**

Khi gặp phép gán chương trình sẽ tính toán giá trị của biểu thức bên vế phải và gán giá trị này cho biến ở vế trái. Ví dụ:

```
int n, i = 3;                       // khởi tạo i bằng 3
n = 10;                             // gán cho n giá trị 10
cout << n <<", " << i << endl;      // in ra: 10, 3
i = n / 2;                          // gán i bằng n/2 = 5
cout << n <<", " << i << endl;      // in ra: 10, 5
```

***b. Phép gán có điều kiện***

*Cú pháp:*

**<biến> = (<điều_kiện>) ? gt1 : gt2 ;**

*điều_kiện* là một biểu thức logic, *gt1*, *gt2* là các biểu thức bất kỳ cùng kiểu với kiểu của *biến*. Phép toán này gán giá trị *gt1* cho *biến* nếu điều kiện đúng và *gt2* nếu ngược lại.

*Ví dụ 2.7*:

```
x = (3 + 4 < 7) ? 10: 20 // x = 20 vì 3 + 4 < 7 là sai
x = (3 + 4) ? 10: 20      // x = 10 vì 3 + 4 != 0, tức điều kiện đúng
x = (a > b) ? a: b        // x = số lớn nhất trong 2 số a, b.
```

***c. Một số lưu ý về phép gán***

• *C++ cho phép chúng ta gán "kép" nhiều biến nhận cùng một giá trị bởi cú pháp:*

**biến_1 = biến_2 = … = biến_n = gt ;**

với cách gán này tất cả các biến sẽ nhận cùng giá trị *gt*. Ví dụ:

```
int i, j, k ;
i = j = k = 1;
```

Ngoài việc gán kép như trên, phép toán gán còn được phép xuất hiện trong bất kỳ biểu thức nào, điều này cho phép một biểu thức chứa phép gán, nó không chỉ tính toán mà còn gán giá trị cho các biến, ví dụ *n = 3 + (i = 2)* sẽ cho ta *i = 2* và *n = 5*.

*Chú ý*: khi sử dụng nhiều chức năng gộp trong một câu lệnh làm cho chương trình gọn gàng hơn (trong một số trường hợp) nhưng câu lệnh trở nên khó hiểu, ví dụ câu lệnh trên nên tách thành 2 câu lệnh: *i = 2; n = 3 + i;* sẽ dễ đọc hơn, nhất là với các bạn mới học lập trình.

• *Viết gọn của phép gán:*

Ví dụ:

Lệnh: x = x + 2 có thể viết gọn là : x += 2;

Các lệnh: x = x/2 ; x = x*2 có thể viết gọn là: x /= 2; x *= 2;

Cách viết gọn này có nhiều thuận lợi khi viết và đọc chương trình nhất là khi tên biến quá dài hoặc đi kèm nhiều chỉ số … Ví dụ, thay vì viết:

```
ngay_quoc_te_lao_dong = ngay_quoc_te_lao_dong + 365;
```

có thể viết gọn hơn bởi:

```
ngay_quoc_te_lao_dong += 365;
```

## II.4. Phép toán, biểu thức và câu lệnh

### II.4.1. Phép toán

C++ có rất nhiều phép toán loại 1 ngôi, 2 ngôi và 3 ngôi. Để có hệ thống, ta tạm phân chia thành các lớp và chỉ trình bày một số trong chúng. Các phép toán còn lại sẽ được tìm hiểu dần trong các phần sau của giáo trình. Các thành phần tên gọi tham gia trong phép toán còn gọi là *toán hạng*, các kí hiệu phép toán còn gọi là *toán tử*. Ví dụ trong phép toán *a + b; a, b* được gọi là toán hạng và + là toán tử. Phép toán 1 ngôi là phép toán chỉ có một toán hạng, ví dụ: *-a* (đổi dấu số *a*), *&x* (lấy địa chỉ của biến *x*) … Một số kí hiệu phép toán cũng được sử dụng chung cho cả 1 ngôi lẫn 2 ngôi (hiển nhiên với ngữ nghĩa khác nhau), ví dụ kí hiệu - được sử dụng cho phép toán trừ 2 ngôi $a - b$, hoặc phép & còn được sử dụng cho phép toán lấy hội các bit (*a & b*) của 2 số nguyên *a* và *b* …

***a. Các phép toán số học: +, -, *, /, %***

- Các phép toán + (cộng), - (trừ), * (nhân) được hiểu theo nghĩa thông thường trong số học.

- Phép toán a / b (chia) được thực hiện theo kiểu của các toán hạng, tức là nếu cả hai toán hạng là số nguyên thì kết quả của phép chia chỉ lấy phần nguyên, ngược lại nếu 1 trong 2 toán hạng là thực thì kết quả là số thực. Ví dụ:

```
13/5 = 2                              // do 13 và 5 là 2 số nguyên
13.0/5 = 13/5.0 = 13.0/5.0 = 2.6  // có ít nhất 1 toán hạng là thực
```

- Phép toán a % b trả lại phần dư của phép chia a/b, trong đó a và b là 2 số nguyên. Ví dụ:

```
13%5 = 3 ;                  // phần dư của 13/5
5%13 = 5 ;                  // phần dư của 5/13
```

***b. Các phép toán tự tăng, giảm: i++ , ++i , i-- , --i***

- Phép toán ++i và i++ sẽ cùng tăng i lên 1 đơn vị tức tương đương với câu lệnh i = i+1. Tuy nhiên nếu 2 phép toán này nằm trong câu lệnh hoặc biểu thức thì ++i khác với i++. Cụ thể ++i sẽ tăng i, sau đó i mới được tham gia vào tính toán trong biểu thức. Ngược lại i++ sẽ tăng i sau khi biểu thức được tính toán xong (với giá trị i cũ). Điểm khác biệt này được minh hoạ thông qua ví dụ sau, giả sử i = 3, j = 15.

| Phép toán | Tương đương | Kết quả |
|---|---|---|
| `i = ++j ;` // tăng trước | `j = j + 1; i = j ;` | `i = 16 , j = 16` |
| `i = j++ ;` // tăng sau | `i = j ; j = j + 1 ;` | `i = 15 , j = 16` |
| `j = ++i + 5 ;` | `i = i + 1 ; j = i + 5 ;` | `i = 4, j = 9` |
| `j = i++ + 5 ;` | `j = i + 5; i = i + 1;` | `i = 4, j = 8` |

*Chú ý*: Việc lạm dụng kết hợp nhiều phép toán tự tăng giảm vào trong biểu thức hoặc câu lệnh sẽ làm chương trình khó hiểu hơn.

***c. Các phép toán so sánh và lôgic***

Đây là các phép toán mà giá trị trả lại là *đúng* hoặc *sai*. Nếu giá trị của biểu thức là *đúng* thì biểu thức nhận giá trị 1, ngược lại là *sai* thì biểu thức nhận giá trị 0. Nói cách khác 1 và 0 là giá trị cụ thể của 2 khái niệm "đúng", "sai".

C++ quan niệm một giá trị bất kỳ khác 0 là "*đúng*" và giá trị 0 là "*sai*".

*+ Các phép toán so sánh*

```
== (bằng nhau), != (khác nhau),
> (lớn hơn), < (nhỏ hơn),
>= (lớn hơn hoặc bằng), <= (nhỏ hơn hoặc bằng).
```

Hai toán hạng của các phép toán này phải cùng kiểu. Ví dụ:

```
3 == 3 hoặc 3 == (4 - 1)     // nhận giá trị 1 vì đúng
3 == 5                       // = 0 vì sai
3 != 5                       // = 1
3 + (5 < 2)                  // = 3 vì 5<2 bằng 0
3 + (5 >= 2)                 // = 4 vì 5>=2 bằng 1
```

*Chú ý*: Cần phân biệt phép toán gán (=) và phép toán so sánh (==). Phép gán vừa gán giá trị cho biến vừa trả lại giá trị bất kỳ (là giá trị của toán hạng bên phải), trong khi phép so sánh luôn luôn trả lại giá trị 1 hoặc 0.

*+ Các phép toán lôgic:*

```
&& (và), || (hoặc ), ! (không, phủ định)
```

Hai toán hạng của loại phép toán này phải có kiểu lôgic tức chỉ nhận một trong hai giá trị "đúng" (số nguyên khác 0) hoặc "sai" (số 0). Khi đó giá trị trả lại của phép toán là 1 hoặc 0 và được cho trong bảng sau:

| a | b | a && b | a \|\| b | ! a |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 |

*Tóm lại:*

- Phép toán && là đúng khi và chỉ khi hai toán hạng cùng đúng
- Phép toán || là sai khi và chỉ khi hai toán hạng cùng sai
- Phép toán ! là đúng khi và chỉ khi toán hạng của nó sai.

*Ví dụ 2.8:*

```
3 && (4 > 5)            // = 0 vì có hạng thức (4>5) sai
(3 >= 1) && (7)         // = 1 vì cả hai hạng thức cùng đúng
!(4 + 3 < 7)            // = 1 vì (4+3<7) bằng 0
5 || (4 >= 6)           // = 1 vì có một hạng thức (5) đúng
(5 < !0) || (4 >= 6)    // = 0 vì cả hai hạng thức đều sai
```

*Chú ý:* việc đánh giá biểu thức được tiến hành từ trái sang phải và sẽ dừng khi biết kết quả mà không chờ đánh giá hết biểu thức. Cách đánh giá này sẽ cho những kết quả phụ khác nhau nếu trong biểu thức ta đưa thêm vào các phép toán tự tăng giảm. Ví dụ cho $i = 2$, $j = 3$, xét 2 biểu thức sau đây:

```
x = (++i < 4 && ++j > 5) //  cho kết quả x = 0 , i = 3 , j = 4
y = (++j > 5 && ++i < 4) //  cho kết quả y = 0 , i = 2 , j = 4
```

cách viết hai biểu thức là như nhau (ngoại trừ hoán đổi vị trí 2 toán hạng của phép toán &&). Với giả thiết $i = 2$ và $j = 3$ ta thấy cả hai biểu thức trên cùng nhận giá trị 0. Tuy nhiên các giá trị của $i$ và $j$ sau khi thực hiện xong hai biểu thức này sẽ có kết quả khác nhau. Với biểu thức đầu vì $++i < 4$ là *đúng* nên chương trình phải tiếp tục tính tiếp $++j > 5$ để đánh giá biểu thức. Do vậy, sau khi đánh giá xong cả $i$ và $j$ đều được tăng 1 ($i$=3, $j$=4). Trong khi, biểu thức sau do $++j > 5$ là *sai* nên chương trình có thể kết luận toàn bộ biểu thức là *sai* mà không cần tính tiếp $++i < 4$. Ví dụ này một lần nữa nhắc ta chú ý kiểm soát kỹ việc sử dụng các phép toán tự tăng giảm trong biểu thức và trong câu lệnh.

### *d. Phép toán bitwise*

Đây là các phép toán thao tác với các bit của số nguyên và chỉ áp dụng trên kiểu số nguyên:

| Phép toán | Ý nghĩa | Kết quả |
|---|---|---|
| & | Phép AND bit | 1&1 = 0, 1&0 = 0&1 = 0&0= 0. |
| \| | Phép OR bit | 0\|0 = 0, 1\|0 = 0\|1 = 1\|1= 1. |
| ^ | Phép XOR bit | 0^0 = 1^1 = 0, 1^0 = 0^1 = 1. |
| << | Phép dịch trái | i << 2 tương đương phép gán i *=2; |
| >> | Phép dịch phải | i >> 2 tương đương phép gán i /=2; |
| ~ | Phép đảo bit | ~1 = 0, ~0 = 1 |

Chú ý phép dịch bit trái hoặc phải bảo toàn bit dấu (bit cực trái).

### II.4.2. Biểu thức

Biểu thức là dãy kí hiệu kết hợp giữa các toán hạng, toán tử và các cặp dấu () theo một qui tắc nhất định. Các toán hạng là hằng, biến, hàm hoặc biểu thức khác. Biểu thức cung cấp một cách thức để tính giá trị mới dựa trên các toán hạng và toán tử trong biểu thức. Ví dụ:

```
(x + y) * 2 - 4 ; 3 - x + sqrt(y) ; (-b + sqrt(delta))/(2*a);
```

#### *a. Thứ tự ưu tiên của các phép toán*

C++ qui định trật tự tính toán theo các mức độ ưu tiên như sau:

1. Các biểu thức trong cặp dấu ngoặc ()
2. Các phép toán 1 ngôi (tự tăng, giảm, lấy địa chỉ, lấy nội dung con trỏ …)
3. Các phép toán số học.
4. Các phép toán quan hệ, logic.
5. Các phép gán.

*Ví dụ:*

Để tính $\Delta = b^2 - 4ac$ ta viết *delta = b * b - 4 * a * c ;*

Để tính nghiệm phương trình bậc 2: $x = \frac{-b + \sqrt{\Delta}}{2a}$ viết : *x = -b + sqrt(delta) / 2*a;* là sai vì theo mức độ ưu tiên x sẽ được tính là *-b + ((sqrt(delta)/2) * a)* (thứ tự tính sẽ là phép toán 1 ngôi đổi dấu -b, đến phép chia, phép nhân và cuối cùng là phép cộng). Để tính chính xác cần phải *viết (-b + sqrt(delta)) / (2*a).*

#### *b. Phép chuyển đổi kiểu*

Khi tính toán một biểu thức, phần lớn các phép toán đều yêu cầu các toán hạng phải cùng kiểu. Ví dụ phép gán yêu cầu giá trị của biểu thức phải bên phải có cùng kiểu với biến bên trái dấu gán. Trong trường hợp kiểu giá trị của biểu thức khác với kiểu biến trong phép gán thì hoặc chương trình sẽ tự động chuyển kiểu giá trị biểu thức về kiểu của biến được gán (nếu được) hoặc sẽ báo lỗi. Do vậy khi cần thiết NSD phải dùng câu lệnh chuyển kiểu của biểu thức cho phù hợp với kiểu biến.

*Chuyển kiểu tự động*: về mặt nguyên tắc, khi cần thiết các kiểu có giá trị thấp sẽ được chương trình tự động chuyển lên kiểu cao hơn cho phù hợp với phép toán. Cụ thể phép chuyển kiểu có thể được thực hiện theo sơ đồ như sau:

```
char ↔ int ↔ long int ↔ float ↔ double
```

*Ví dụ 2.9:*

```
int i = 3;
float f ;
f = i + 2;
```

trong ví dụ trên i có kiểu nguyên và vì vậy i + 2 cũng có kiểu nguyên trong khi f có kiểu thực. Tuy vậy phép toán gán này là hợp lệ vì chương trình sẽ tự động chuyển kiểu cuả i + 2 (bằng 5) sang kiểu thực (bằng 5.0) rồi mới gán cho f.

*Ép kiểu*: trong chuyển kiểu tự động, chương trình chuyển các kiểu từ thấp đến cao, tuy nhiên chiều ngược lại không thể thực hiện được vì nó có thể gây mất dữ liệu. Do đó khi cần, ta phải ra lệnh cho chương trình chuyển kiểu. Ví dụ:

```
int i;
float f = 3 ;  // tự động chuyển 3 thành 3.0 và gán cho f
i = f + 2 ;    // sai vì mặc dù f + 2 = 5 nhưng không gán được cho i
```

Trong ví dụ trên để câu lệnh *i = f + 2* thực hiện được ta phải ép kiểu của biểu thức *f + 2* về thành kiểu nguyên. Cú pháp tổng quát như sau:

```
(tên_kiểu)biểu_thức      // cú pháp cũ trong C
```

hoặc:

```
tên_kiểu(biểu_thức)      // cú pháp mới trong C++
```

trong đó *tên_kiểu* là kiểu cần được chuyển sang. Như vậy câu lệnh trên phải được viết lại:

```
i = int(f + 2) ;
```

khi đó *f + 2* (bằng 5.0) được chuyển thành 5 và gán cho i.

Dưới đây ta sẽ xét một số ví dụ về lợi ích của việc ép kiểu.

Phép ép kiểu từ một số thực về số nguyên sẽ cắt bỏ tất cả phần thập phân của số thực, chỉ để lại phần nguyên. Như vậy để tính phần nguyên của một số thực x ta chỉ cần ép kiểu của x về thành kiểu nguyên, có nghĩa int(x) là phần nguyên của số thực x bất kỳ.

Ví dụ kiểm tra một số nguyên n có phải là số chính phương không, ta làm như sau:

```
int n = 10 ;
float x = sqrt(n) ;
if (int(x) == x) cout << "n chính phương" ;
else cout << "n không chính phương" ;
```

Để biết mã ASCII của một kí tự ta chỉ cần chuyển kí tự đó sang kiểu nguyên.

```
char c ;
cin >> c ;
cout << "Mã của kí tự vừa nhập là " << int(c) ;
```

*Ghi chú*: Xét ví dụ sau:

```
int i = 3 , j = 5 ;
float x ;
x = i / j * 10;
cout << x ;
```

trong ví dụ này mặc dù x là kiểu float nhưng kết quả in ra sẽ là 0 thay vì 6.0. Lý do, vì phép chia giữa 2 số nguyên i và j cho kết quả nguyên, i / j = 3 / 5 = 0. Từ đó x = 0*10 = 0. Để phép chia ra kết quả thực ta cần phải ép kiểu hoặc i hoặc j hoặc cả 2 thành số thực, khi đó phép chia sẽ cho kết quả thực và x được tính đúng giá trị. Cụ thể:

```
x = float(i) / j * 10 ;           // đúng
x = i / float(j) * 10   ;         // đúng
x = float(i) / float(j) * 10 ;    // đúng
x = float(i/j) * 10 ;             // sai
```

Phép ép kiểu: *x = float(i / j) * 10* ; vẫn cho kết quả sai vì trong dấu ngoặc phép chia *i / j* vẫn là phép chia nguyên, kết quả *x* vẫn là 0.

### II.4.3. Câu lệnh và khối lệnh

Một câu lệnh trong C++ được thiết lập từ các từ khoá và các biểu thức … và luôn luôn được kết thúc bằng dấu chấm phẩy ;. Các ví dụ vào/ra hoặc các phép gán tạo thành những câu lệnh đơn giản như:

```
cin >> x >> y ;
x = 3 + x ; y = (x = sqrt(x)) + 1 ;
cout << x ;
cout << y ;
```

Các câu lệnh được phép viết trên một hoặc nhiều dòng. Một số câu lệnh được gọi là lệnh có cấu trúc, tức là nó chứa dãy nhiều lệnh khác. Dãy lệnh này phải được bao giữa cặp dấu ngoặc {} và được gọi là *khối lệnh*. Ví dụ, tất cả các lệnh trong một hàm (như hàm *main()*) luôn luôn là một khối lệnh. Một đặc điểm của khối lệnh là các biến được khai báo trong khối lệnh nào thì chỉ có tác dụng trong khối lệnh đó. Chi tiết hơn về các đặc điểm của lệnh và khối lệnh sẽ được trình bày trong các chương tiếp theo của giáo trình.

## II.5. Cấu trúc điều khiển chương trình

Việc thực hiện chương trình là hoạt động tuần tự. Chương trình thực hiện từng lệnh một từ câu lệnh bắt đầu của chương trình cho đến câu lệnh cuối cùng. Tuy nhiên, để việc lập trình hiệu quả hơn hầu hết các NNLT bậc cao đều có các câu lệnh rẽ nhánh và các câu lệnh lặp cho phép thực hiện các câu lệnh của chương trình không theo trình tự tuần tự như trong văn bản.

Phần này sẽ trình bày các cấu trúc cho phép chương trình rẽ nhánh và lặp.

### II.5.1. Cấu trúc rẽ nhánh

#### *a. Câu lệnh if*

*+ Ý nghĩa*

Một câu lệnh *if* cho phép chương trình có thể chọn thực hiện khối lệnh này hay khối lệnh khác phụ thuộc vào điều kiện được viết trong câu lệnh là *đúng* hay *sai*.

*+ Cú pháp*

- Kiểu 1: **`if (điều kiện) { khối lệnh; }`**
- Kiểu 2: **`if (điều kiện) { khối lệnh 1; } else { khối lệnh 2; }`**

Trong cú pháp trên câu lệnh *if* có hai dạng: có *else* và không có *else*. *điều kiện* là một biểu thức lôgic, tức là nó có giá trị đúng (khác 0) hoặc sai (bằng 0).

Khi gặp câu lệnh *if*, chương trình sẽ tính biểu thức *điều kiện*. Nếu *điều kiện* đúng chương trình sẽ tiếp tục thực hiện các lệnh trong *khối lệnh* (kiểu 1) hoặc *khối lệnh 1* (kiểu 2), nếu *điều kiện* sai, chương trình sẽ bỏ qua *if* (đối với kiểu 1) và thực hiện *khối lệnh 2* (đối với kiểu 2).

*+ Đặc điểm*

- Đặc điểm chung của các câu lệnh có cấu trúc là nó có thể chứa các câu lệnh khác. Điều này cho phép các câu lệnh *if* có thể đặt lồng vào nhau.
- Nếu nhiều câu lệnh *if* (có *else* và không *else*) lồng nhau, cần chú ý việc hiểu *if* và *else* nào đi với nhau. Qui tắc là *else* sẽ đi với *if* gần nó nhất mà chưa được ghép cặp với *else* khác. Ví dụ câu lệnh:

```
if (n>0) if (a>b) c = a; else c = b;
```

tương đương với:

```
if (n>0) { if (a>b) c = a; else c = b;}
```

*Ví dụ 2.10*: Bằng phép toán gán có điều kiện có thể tìm số lớn nhất max trong 2 số a, b như sau: `max = (a > b) ? a: b ;`

hoặc max được tính bởi dùng câu lệnh if:

```
if (a > b) max = a; else max = b;
```

*Ví dụ 2.11*: Tính năm nhuận. Năm thứ n là nhuận nếu nó chia hết cho 4, nhưng không chia hết cho 100 hoặc chia hết 400.

```
#include <iostream>
using namespace std;
int main()
{
    int year;
    cout << "Nam = " ; cin >> year ;
    if (year%4 == 0 && year%100 != 0 || year%400 == 0)
        cout << year << " la nam nhuan";
    else
        cout << year << " la nam khong nhuan";
    return 0;
}
```

*Ví dụ 2.12*: Giải phương trình bậc 2: $ax^2 + bx + c = 0$, biết $a \neq 0$, b, c ?

```
#include <iostream>
#include <cmath>            // để sử dụng hàm sqrt()
using namespace std;
int main()
{
    float a, b, c, delta, x1, x2;
    cout << "Nhap a , b, c: " ; cin >> a >> b >> c ;
    if (a == 0){
        cout<<"Ban da nhap a = 0"; return 0;
    }
    delta = b*b - 4*a*c ;
    if (delta < 0)
        cout << "PT vo nghiem"<<endl ;
    else if (delta == 0)
        cout<<"PT co nghiem kep x = " << -b/(2*a) <<endl;
    else
    {
        x1 = (-b+sqrt(delta))/(2*a);
        x2 = (-b-sqrt(delta))/(2*a);
        cout << "x1 = " << x1 << " x2 = " << x2 ;
    }
    return 0;
}
```

***b. Câu lệnh lựa chọn trường hợp swith***

*+ Ý nghĩa*

Câu lệnh *if* cho ta khả năng được lựa chọn một trong hai nhánh để thực hiện, do đó nếu sử dụng nhiều lệnh *if* lồng nhau sẽ cung cấp khả năng được rẽ theo nhiều nhánh. Tuy nhiên, như vậy chương trình sẽ khó đọc. Trong một số trường hợp cụ thể, C++ cung cấp câu lệnh cấu trúc *switch* cho phép chương trình có thể chọn một hoặc một số trong nhiều trường hợp để thực hiện.

*+ Cú pháp*

```
switch (biểu thức điều khiển)
{
    case biểu_thức_1: dãy lệnh 1 ;
    case biểu_thức_2: dãy lệnh 2 ;
    ...
    case biểu_thức_n: dãy lệnh n ;
    default: dãy lệnh n+1;
}
```

- *biểu thức điều khiển*: phải có kiểu nguyên hoặc kí tự,
- Các *biểu_thức_i* : phải là các hằng nguyên hoặc kí tự,
- Các *dãy lệnh* có thể rỗng. Không cần bao *dãy lệnh* bởi cặp dấu {},
- Nhánh *default* có thể có hoặc không và nó có thể nằm ở vị trí bất kỳ trong cấu trúc (không nhất thiết phải nằm cuối cùng).

*+ Cách thực hiện*

Khi gặp câu lệnh *switch*, đầu tiên chương trình tính giá trị của *biểu thức điều khiển* (*btđk*), sau đó so sánh kết quả của *btđk* với giá trị của các *biểu thức* bên dưới. Lần lượt từ *biểu_thức_1* đến *biểu_thức_n*, nếu giá trị của *btđk* bằng giá trị của *biểu_thức_i* đầu tiên nào đó thì chương trình sẽ thực hiện *dãy lệnh i.* Sau đó chương trình lại tiếp tục kiểm tra và thực hiện như vậy với các dãy lệnh còn lại (từ *dãy lệnh i+1*) cho đến hết (gặp dấu ngoặc đóng } của lệnh switch). Nếu quá trình so sánh không gặp biểu thức nào bằng với giá trị của *btđk* thì chương trình thực hiện *dãy lệnh n+1* sau *default* và tiếp tục cho đến hết (nếu sau *default* còn những nhánh case khác). Trường hợp câu lệnh *switch* không có nhánh *default* và *btđk* không khớp với bất cứ nhánh case nào thì chương trình không làm gì, coi như đã thực hiện xong lệnh *switch*.

Nếu muốn lệnh *switch* chỉ thực hiện nhánh thứ i (khi *btđk = biểu_thức_i*) mà không thực hiện tiếp các lệnh tiếp theo thì cuối *dãy lệnh thứ i*, ta đặt thêm lệnh *break*; đây là lệnh cho phép thoát ra khỏi một lệnh cấu trúc bất kỳ.

*Ví dụ 2.13* : In số ngày của một tháng bất kỳ nào đó được nhập từ bàn phím.

```
#include <iostream>
using namespace std;
int main()
{
    int th,n;
    cout << "Cho biet thang va nam : " ; cin >> th >>n ;
    cout << "thang "<<th<<"/"<<n<<" : ";
    switch (th)
    {
        case 1:  case 3:  case 5: case 7: case 8: case 10:
        case 12:  cout<<"31 ngay"; break ;
        case 2:{
             if(n%4 == 0 && n%100 !=0 || n%400 == 0)cout<< "29 ngay";
             else cout<<"28 ngay"; break;
        }
        case 4:   case 6:    case 9:
         case 11: cout << "30 ngay" ; break;
        default: cout << "Ban da nhap sai thang hoac nam" ;
    }
    return 0;
}
```

*Ví dụ 2.14* : Nhập 2 số a và b vào từ bàn phím. Nhập kí tự thể hiện một trong bốn phép toán: cộng, trừ, nhân, chia. In ra kết quả thực hiện phép toán đó trên 2 số a, b.

```
#include <iostream>
#include <iomanip>  // su dung lenh setiosflags(ios::showpoint)
using namespace std;

int main()
{
    float a, b, c ;       // các toán hạng a, b và kết quả c
    char dau ;            // phép toán được cho dưới dạng kí tự
    cout << "Nhap a, b: " ; cin >> a >> b ;
    cout << "Nhap dau phep toan (+,-, x,/) : " ; cin >> dau ;
    switch (dau)
    {
        case '+': c = a + b ; break ;
        case '-': c = a - b ; break ;
        case 'x': case '.': case '*': c = a * b ; break ;
        case ':': case '/': c = a / b ; break ;
    }
    cout << setiosflags(ios::showpoint)<<setprecision(4);
    cout << "Ket qua la: " << c ;
        return 0;
}
```

Trong chương trình trên ta chấp nhận các kí tự *x*, . , * thể hiện cho phép toán nhân và *:* , / thể hiện phép toán chia.

### *c. Câu lệnh nhảy goto*

Một dạng khác của rẽ nhánh là câu lệnh nhảy *goto,* cho phép chương trình chuyển đến thực hiện một đoạn lệnh khác bắt đầu từ một điểm được đánh dấu bởi một nhãn trong chương trình. Nhãn là một tên gọi do NSD tự đặt theo các qui tắt đặt tên của C++. Lệnh *goto* thường được sử dụng để tạo vòng lặp. Tuy nhiên, việc xuất hiện nhiều lệnh *goto* dẫn đến việc khó theo dõi trình tự thực hiện chương trình, vì vậy lệnh này ít được dùng.

*+ Cú pháp*

**goto <nhãn:> ;**

Vị trí chương trình chuyển đến thực hiện là đoạn lệnh đứng sau nhãn và dấu hai chấm (:).

*Ví dụ 2.15* : Nhân 2 số nguyên theo phương pháp Ấn độ.

Phương pháp Ấn độ cho phép nhân 2 số nguyên bằng cách chỉ dùng các phép toán nhân 2, chia 2 và cộng. Các phép nhân 2 và chia 2 thực chất là phép toán bitwise dịch trái (nhân 2) hoặc dịch phải (chia 2). Giả sử cần nhân m với n. Thuật toán như sau:

**Mô tả thuật toán**

```
Bước 0: kq = 0;
Bước 1: Nếu m lẻ kq += n;
Bước 2: n >> 1; m << 1;
Bước 3: Nếu m != 0 sang
Bước 1.
Bước 4: in kq, dừng TT
```

| Bước | m (chia 2) | n (nhân 2) | kq (khởi tạo kq = 0) |
|---|---|---|---|
| 1 | 21 | 11 | m lẻ, cộng thêm 11 vào kq = 0 + 11 = 11 |
| 2 | 10 | 22 | m chẵn, bỏ qua |
| 3 | 5 | 44 | m lẻ, cộng thêm 44 vào kq = 11 + 44 = 55 |
| 4 | 2 | 88 | m chẵn, bỏ qua |
| 5 | 1 | 176 | m lẻ, cộng thêm 176 vào kq = 55 + 176 = 231 |
| 6 | 0 | | m = 0, dừng cho kết quả kq = 231 |

Sau đây là chương trình C++ với câu lệnh goto giải bài toán trên.

```
#include <iostream>
using namespace std;
int main()
{
    long m, n, kq = 0;        // Các số cần nhân và kết quả kq
    cout << "Nhap m và n: " ; cin >> m >> n ;
    cout <<m<<" x "<<n<<" = ";
    lap:                      // đây là nhãn để chương trình quay lại
    if (m%2) kq += n;         // nếu m lẻ thì cộng thêm n vào kq
    m = m >> 1;               // dịch m sang phải 1 bit ~ m = m / 2
    n = n << 1;               // dịch m sang trái 1 bit ~ m = m * 2
    if (m) goto lap;          // quay lại nếu m != 0
    cout << kq ;
     return 0;
}
```

### II.5.2. Cấu trúc lặp

Một trong những cấu trúc quan trọng của C++ là các câu lệnh lặp. Các lệnh lặp cho phép lặp nhiều lần một đoạn lệnh nào đó của chương trình.

C++ cung cấp cho chúng ta 3 lệnh lặp. Về thực chất 3 lệnh này là tương đương (và có thể thay bằng goto). Tuy nhiên, để chương trình được sáng sủa, rõ ràng, C++ đã cung cấp nhiều phương án cho NSD lựa chọn câu lệnh lặp để viết chương trình cho phù hợp.

Mỗi bài toán lặp có một đặc trưng riêng. Việc sử dụng lệnh lặp phù hợp, làm chương trình dễ đọc và dễ bảo trì hơn.

#### *a. Lệnh lặp for*

*+ Cú pháp*

**for (<day_bt_1>; <dk_lap>; <day_bt_2>) {khoi_lenh_lap;}**

- Trong các thành phần <*day_bt_1*> và <*day_bt_2*> có thể chứa nhiều biểu thức, cách nhau bởi dấu phảy (,).

- <*dk_lap*>: là biểu thức lôgic có giá trị đúng/sai để quyết định có tiếp tục thực hiện *khoi_lenh_lap* nữa không.

- Các thành phần <*day_bt_1*> và <*day_bt_2*> và / hoặc <*dk_lap*> có thể để trống tuy nhiên vẫn phải giữ lại các dấu chấm phảy (;) ngăn cách các thành phần với nhau.

*+ Cách thực hiện*

Khi gặp câu lệnh for, trình tự thực hiện của chương trình như sau:

- Thực hiện <*day_bt_1*> (thường là các lệnh khởi tạo cho một số biến),

- Kiểm tra <*dk_lap*>, nếu đúng thì thực hiện *khoi_lenh_lap* → thực hiện <*day_bt_1*> → quay lại kiểm tra <*dk_lap*>, quá trình trên lặp lại cho đến khi <*dk_lap*> sai thì dừng vòng lặp.

*Ví dụ 2.16* : Nhân 2 số nguyên theo phương pháp Ấn độ

```
#include <iostream>
using namespace std;
int main()
{
    long m, n, kq;    // Các số cần nhân và kết quả kq
```

```
    cout << "Nhap m va n: " ; cin >> m >> n ;
    cout << m << " * " << n <<" = ";
    for (kq = 0 ; m ; m >>= 1, n <<= 1)
        if (m % 2) kq += n ;
    cout<< kq ;
    return 0;
}
```

So sánh ví dụ này với ví dụ 2.15 dùng *goto* ta thấy chương trình được viết rất gọn.

*Ví dụ 2.17* : Tính tổng của dãy các số từ 1 đến 100.

Chương trình dùng một biến đếm i được khởi tạo từ 1, và một biến kq để chứa tổng. Mỗi bước lặp chương trình cộng i vào kq và sau đó tăng i lên 1 đơn vị. Chương trình còn lặp khi nào i còn chưa vượt qua 100. Khi i lớn hơn 100 chương trình dừng.

```
#include <iostream>
using namespace std;
int main()
{
    int i, kq = 0;
    for (i = 1 ; i <= 100 ; i ++) kq += i ;
    cout << "Tong = " << kq;
    return 0;
}
```

*Ví dụ 2.18* : In ra màn hình dãy số lẻ bé hơn một số n nào đó được nhập vào từ bàn phím.

Chương trình dùng một biến đếm i được khởi tạo từ 1, mỗi bước lặp chương trình sẽ in i sau đó tăng i lên 2 đơn vị. Chương trình còn lặp khi nào i còn chưa vượt qua n. Khi i lớn hơn n chương trình dừng.

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

+ *Chú ý:*

Các thành phần của *for* có thể để trống, tuy nhiên các dấu chấm phẩy vẫn giữ lại để ngăn cách các thành phần với nhau.

Ví dụ câu lệnh

```
for (kq = 0 ; m ; m >>= 1, n <<= 1) if (m%2) kq += n ;
```

trong ví dụ 2.16 có thể viết lại như sau:

```
kq = 0; for ( ; m ; ) { if (m%2) kq += n; m >>= 1; n <<= 1; }
```

câu lệnh

```
for (i = 1 ; i <= 100 ; i ++) kq += i ;
```

trong ví dụ 2.17 có thể được viết lại như sau:

```
i = 1; for ( ; i <= 100 ; ) kq += i++ ;
```

Biểu thức điều kiện trong for cũng để trống điều kiện lặp, chương trình sẽ ngầm định là điều kiện luôn luôn đúng, tức vòng lặp sẽ lặp vô hạn lần (!). Trong trường hợp này để dừng vòng lặp trong khối lệnh cần có câu lệnh kiểm tra dừng và câu lệnh break.

Ví dụ câu lệnh

```
for (i = 1 ; i <= 100 ; i ++) kq += i ;
```

được viết lại như sau:

```
i = 1;
for ( ; ; ){  kq += i++; if (i > 100) break; }
```

*+ Lệnh for lồng nhau*

Trong dãy lệnh lặp có thể chứa cả lệnh for, tức là các lệnh for cũng được phép lồng nhau như các câu lệnh có cấu trúc khác.

*Ví dụ 2.19* : Bài toán cổ: vừa gà vừa chó có 36 con và 100 chân. Hỏi có mấy gà và mấy chó.

```
#include <iostream>
using namespace std;
int main()
{
    int ga, cho ;
    for (ga = 0 ; ga <= 36 ; ga++)
        for (cho = 0; cho <= 36; cho++)
            if (ga*2 + cho*4 == 100 && ga + cho == 36)
                cout<<"ga= "<<ga<<" cho= "<<cho<<endl;
    return 0;
}
```

*Chú ý*: Cách giải trên chưa tối ưu với bài toán. Bạn đọc có thể tự nghĩ các giải khác để giảm số vòng lặp đến ít nhất.

*Ví dụ 2.20* : Tìm tất cả các phương án để có 100đ từ các tờ giấy bạc loại 10đ, 20đ và 50đ.

```
#include <iostream>
using namespace std;
int main()
{
    int t10, t20, t50;    // số tờ 10đ, 20đ, 50đ
    int sopa = 0;         // số phương án
    for (t10 = 0 ; t10 <= 10 ; t10++)
    for (t20 = 0 ; t20 <= 5 ; t20++)
    for (t50 = 0 ; t50 <= 2 ; t50++)
    if (t10*10 + t20*20 + t50*50 == 100) // nếu thoả đk thì
    {
        sopa++;                          // tăng số phương án
        cout << t10 << " to 10d + " ;    // in số tờ 10đ
        cout << t20 << " to 20d + " ;    // thêm số tờ 20đ
        cout << t50 << " to 50d " ;      // thêm số tờ 50đ
        cout << '\n' ;
    }
    cout << "Tong so phuong an = " << sopa ;
    return 0;
}
```

### *b. Lệnh lặp while*

*+ Cú pháp*

```
while (<dk_lap>) { khoi_lenh_lap; }
```

*+ Cách thực hiện*

Khi gặp lệnh *while*, chương trình kiểm tra <*dk_lap*>, nếu đúng thì thực hiện *khoi_lenh_lap*, sau đó quay lại kiểm tra <*dk_lap*> và cứ thế tiếp tục. Nếu <*dk_lap*> sai thì dừng vòng lặp.

+ *Đặc điểm*

- *khoi_lenh_lap* có thể không được thực hiện lần nào nếu <*dk_lap*> sai ngay từ đầu.
- Lệnh lặp có thể vô hạn khi trong *khoi_lenh_lap* không có câu lệnh nào tác động đến kết quả của <*dk_lap*>, làm cho <*dk_lap*> đang đúng trở thành sai.
- Nếu <*dk_lap*> luôn luôn nhận giá trị đúng (ví dụ <*dk_lap*> == 1) thì *khoi_lenh_lap* phải chứa câu lệnh kiểm tra và dừng vòng lặp bằng lệnh *break*.

*Ví dụ 2.21* : Nhân 2 số nguyên theo phương pháp Ấn độ

```
#include <iostream>
using namespace std;
int main()
{
    long m, n, kq;
    cout << "Nhap m va n: " ; cin >> m >> n ;
    cout << m << " x "<< n <<" = ";
    kq = 0 ;
    while (m)
    {
        if (m%2) kq += n ;
        m >>= 1; n <<= 1;
    }
    cout << kq ;
    return 0;
}
```

*Ví dụ 2.22* : Một phiên bản khác của bài toán cổ gà và chó.

```
#include <iostream>
using namespace std;
int main()
{
    int g =0, c ;
    while (g <= 36) {
        c = 36-g ;
        if (2*g + 4*c == 100) cout <<"ga = "<< g <<" cho = "<< c ;
        g++;
    }
    return 0;
}
```

*Ví dụ 2.23* : Tìm ước chung lớn nhất (UCLN) của 2 số nguyên m và n.

```
#include <iostream>
using namespace std;
int main()
{
    int m, n, r;
    cout << "Nhap m, n: " ; cin >> m >> n ;
    while (m != n) {
        if(m>n) m = m - n ;
        if(n>m) n = n - m ;
    }
    cout << "UCLN = " << m ;
    return 0;
}
```

### *c. Lệnh lặp do ... while*

+ *Cú pháp*

**do { khoi_lenh_lap; } while (<dk_lap>);**

*+ Cách thực hiện*

Khi gặp lệnh *do while,* chương trình sẽ thực hiện *khoi_lenh_lap*, sau đó kiểm tra <*dk_lap*>, nếu <*dk_lap*> đúng thì thực hiện lại *khoi_lenh_lap,* cứ thế cho đến khi <*dk_lap*> sai thì dừng.

*+ Đặc điểm*

Các đặc điểm của câu lệnh *do ... while* cũng giống với câu lệnh lặp *while* trừ điểm khác biệt, đó là *khoi_lenh_lap* trong *do ... while* sẽ được thực hiện ít nhất một lần (vì lệnh *do while* kiểm tra <*dk_lap*> sau khi đã thực hiện *khoi_lenh_lap*, do đó khi <*dk_lap*> sai thì *khoi_lenh_lap* đã được thực hiện ít nhất một lần).

*Ví dụ 2.24* : Kiểm tra một số n có là số nguyên tố.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    int n, i = 2;
    cout << "Nhap so can kiem tra: " ; cin >> n ;
    do {
        if (n % i == 0) {
            cout << n << " la hop so" ;
            return 0;
        }
        i++;
    } while (i*i <= n);
    cout << n << " la so nguyen to" ;
    return 0;
}
```

*Ví dụ 2.25* : Nhập dãy kí tự và thống kê các loại chữ hoa, thường, chữ số và các loại khác cho đến khi gặp ENTER thì dừng.

```
#include <iostream>
using namespace std;
int main()
{
    char c;
    int chuthuong, chuhoa, chuso, khac ;
    chuthuong = chuhoa = chuso = khac = 0;
    cout << "Nhap day ki tu: \n" ;
    do
    {
        cin.get(c);       // nhập 1 kí tự
        cin.ignore();     // xóa ký tự Enter trong bộ đệm
        if ('a' <= c && c <= 'z') chuthuong++;
        else if ('A' <= c && c <= 'Z') chuhoa++;
        else if ('0' <= c && c <= '9') chuso++;
        else khac++;
    } while (c != 10) ;
    cout << "chu in thuong = "<<chuthuong << endl;
    cout << "chu in hoa = "<<chuhoa << endl;
    cout << "chu so = "<<chuso << endl;
    cout << "Ki tu khac = "<<khac<< endl;
    return 0;
}
```

***d. Lệnh break, continue***

+ *Lệnh* ***break***

Công dụng của lệnh *break* là để thoát ra khỏi (chấm dứt) các câu lệnh cấu trúc, chương trình sẽ tiếp tục thực hiện các câu lệnh tiếp sau câu lệnh cấu trúc vừa thoát.

+ *Lệnh* ***continue***

Khi gặp lệnh này trong vòng lặp, chương trình sẽ bỏ qua các lệnh còn lại của khối lệnh lặp hiện tại mà chưa thực hiện, để thực hiện lần lặp tiếp theo.

*Ví dụ 2.26* : In ra các số nguyên i từ 1 đến 100 là số không chính phương.

```
#include <iostream>
#include <cmath>      // sử dụng hàm sqrt()
using namespace std;
int main()
{
    for (int i=1; i<=100; i++){
        if (sqrt(i)==int(sqrt(i))) continue;
        cout<< i <<" ";
    }
    return 0;
}
```

**II.5.3. So sánh cách dùng các câu lệnh lặp**

Khả năng thực hiện các câu lệnh lặp cho một phép toán lặp là như nhau. Tuy nhiên, tùy ngữ cảnh cụ thể, ta chọn viết câu lệnh lặp phù hợp làm chương trình sáng sủa, rõ ràng và tăng độ tin cậy. Sau đây là một số gợi ý dùng các lệnh lặp trong một số ngữ cảnh cụ thể:

- *for* thường được sử dụng trong những vòng lặp mà số lần lặp được biết trước, nghĩa là vòng lặp thường được tổ chức dưới dạng một (hoặc nhiều) biến đếm chạy từ một giá trị xuất phát nào đó và đến khi đạt được đến một giá trị tới hạn cho trước thì dừng. Người ta hay gọi for là *lặp xác định*.
- Ngược lại với *for, while* và *do ..while* thường dùng cho các vòng lặp mà số lần lặp không biết trước, việc lặp tiếp hay dừng phụ thuộc vào một biểu thức lôgic. Người ta hay gọi là *lặp không xác định.*
- Với lệnh lặp *while*, biểu thức điều kiện lặp được kiểm tra trước khi thực hiện khối lệnh lặp và người ta hay gọi là *lặp không xác định kiểm tra điều kiện trước*. Trong khi, *do… while* lại thực hiện khối lệnh lặp trước sau đó mới kiểm tra biểu thức điều kiện và người ta hay gọi là *lặp không xác định kiểm tra điều kiện sau*.

## II.5. Câu hỏi và bài tập

**a. Câu hỏi**

1. Kiểu biến nào của C++ dưới đây lưu trữ được giá trị số 3.1415?

A. int B. char C. double D. string

2. Để so sánh giá trị hai biến trong C++ người ta dùng phép toán nào dưới đây?

A. := B. = C. equal D. ==

3. Tên nào dưới đây không phải là kiểu dữ liệu trong C++?

A. double
C. int
B. real
D. char

4. Màn hình hiển thị giá trị nào dưới đây khi thực hiện lệnh C++ sau: *cout << 1234/2000*?

A. 0
C. Khoảng 0.617, nhưng không chính xác lắm
B. 0.617
D. Tùy thuộc vào giá trị hai bên dấu /

5. Vì sao C++ cần có kiểu char nếu đã có kiểu int?

A. Vì một kiểu là số, một kiểu là kí tự. Chúng khác nhau hoàn toàn
B. Để tương thích với ngôn ngữ C
C. Để có thể dễ dàng đọc và in ra, mặc lưu trữ như số nhưng char là kiểu ký tự
D. Để được hỗ trợ quốc tế, xử lý các ngôn ngữ như Trung Quốc và Nhật Bản, có nhiều ký tự

6. Viết câu lệnh khai báo biến để lưu các giá trị sau:

a. Tuổi của một người
b. Số lượng cây trong thành phố
c. Độ dài cạnh một tam giác
d. Khoảng cách giữa các hành tinh
e. Một chữ số
f. Nghiệm x của phương trình bậc 1
g. Một chữ cái
h. Biệt thức Δ của phương trình bậc 2

7. Cho n = x = y và bằng:

A. 1 B. 2 C. 3 D. 4

Hãy cho biết giá trị của x, y sau khi chạy xong câu lệnh:

```
if (n % 2 == 0) if (x > 3) x = 0;
else y = 0;
```

8. Giá trị của i bằng bao nhiêu sau khi thực hiện cấu trúc for sau:

```
for (i = 0; i < 100; i++);
```

9. Giá trị của x bằng bao nhiêu sau khi thực hiện cấu trúc for sau:

```
for (x = 2; i < 10; x+=3) ;
```

10. Bạn bổ sung gì vào lệnh for sau: `for ( ; nam < 1997 ; );` để kết thúc `nam` có giá trị 2000.

11. Bao nhiêu kí tự 'X' được in ra màn hình khi thực hiện đoạn chương trình sau:

```
for (x = 0; x < 10; x ++) for (y = 5; y > 0; y --) cout << 'X';
```

**b. Bài tập**

1. Viết chương trình có các câu lệnh nhập vào 4 giá trị lần lượt là số thực, nguyên, nguyên dài và kí tự. In ra màn hình các giá trị này để kiểm tra.
2. Viết chương trình có các câu lệnh in ra màn hình các dòng như sau:

```
Bộ Giáo dục và Đào tạo          Cộng hoà xã hội chủ nghĩa Việt Nam
 Sở Giáo dục Hòa Bình              Độc lập - Tự do - Hạnh phúc
```

3. Viết chương trình nhập vào một kí tự. In ra kí tự đó và mã ascii của nó.
4. Viết chương trình nhập vào hai số thực và in các số đó với 2 chữ số lẻ, cách nhau 5 cột.
5. Nhập, chạy và giải thích kết quả đạt được của đoạn chương trình sau:

```
#include <iostream>
using namespace std;
int main()
{
     char c1 = 200; unsigned char c2 = 200 ;
     cout << "c1 = " << c1 << ", c2 = " << c2 << "\n" ;
```

```
        cout << "c1+100 = " << c1+100 << ", c2+100 = " << c2+100 ;
        cin.get();
        return 0;
}
```

6. Nhập a, b, c. In ra màn hình dòng chữ phương trình có dạng $ax^2 + bx + c = 0$, trong đó các giá trị a, b, c. Chỉ in 2 số lẻ.
7. Viết chương trình tính và in ra giá trị các biểu thức sau với 2 số lẻ:

   a. $\sqrt{3+\sqrt{3+\sqrt{3}}}$ b. $\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2}}}$
8. Nhập a, b, c là các số thực. In ra giá trị của các biểu thức sau với 3 số lẻ:

   a. $a^2 - 2b + ab/c$ c. $3a - b^3 - 2\sqrt{c}$

   b. $\dfrac{b^2 - 4ac}{2a}$ d. $\sqrt{a^2/b - 4a/bc + 1}$
9. In ra tổng, tích, hiệu và thương của 2 số được nhập vào từ bàn phím.
10. In ra trung bình cộng, trung bình nhân của 3 số được nhập vào từ bàn phím.
11. Viết chương trình nhập cạnh, bán kính và in ra diện tích, chu vi của các hình: vuông, chữ nhật, tròn.
12. Tính diện tích và thể tích của hình cầu bán kính R theo công thức: $S = 4\pi R^2$ ; $V = RS/3$
13. Nhập vào 4 chữ số. In ra tổng của 4 chữ số này và chữ số hàng chục, hàng đơn vị của tổng (ví dụ 4 chữ số 3, 1, 8, 5 có tổng là 17 và chữ số hàng chục là 1 và hàng đơn vị là 7, cần in ra 17, 1, 7).
14. Nhập vào một số nguyên (có 4 chữ số). In ra tổng của 4 chữ số này và chữ số đầu, chữ số cuối (ví dụ số 3185 có tổng các chữ số là 17, đầu và cuối là 3 và 5, kết quả in ra là: 17, 3, 5).
15. Hãy nhập 2 số a và b. Viết chương trình tráo đổi giá trị của a và b theo 2 cách:
    a. dùng biến phụ t: t = a; a = b; b = t;
    b. không dùng biến phụ: a = a + b; b = a - b; a = a - b;
    In kết quả ra màn hình để kiểm tra.
16. Viết chương trình đoán số của người chơi đang nghĩ, bằng cách yêu cầu người chơi nghĩ một số, sau đó thực hiện một loạt các tính toán trên số đã nghĩ rồi cho biết kết quả. Máy sẽ in ra số mà người chơi đã nghĩ. (ví dụ yêu cầu người chơi lấy số đã nghĩ nhân đôi, trừ 4, bình phương, chia 2 và trừ 7 rồi cho biết kết quả, máy sẽ in ra số người chơi đã nghĩ).
17. Nhập một kí tự. Cho biết kí tự đó có phải là chữ cái hay không.
18. Nhập vào một số nguyên. Trả lời số nguyên đó: âm hay dương, chẵn hay lẻ ?
19. Viết chương trình giải hệ phương trình bậc nhất 2 ẩn: $\begin{cases} ax + by = c \\ dx + ey = f \end{cases}$
20. Nhập 2 số a, b. In ra max, min của 2 số đó. Mở rộng với 3 số, 4 số ?
21. Nhập 3 số a, b, c. Hãy cho biết 3 số trên có thể là độ dài 3 cạnh của một tam giác ? Nếu là một tam giác thì đó là tam giác gì: vuông, đều, cân, vuông cân hay tam giác thường ?
22. Nhập vào một số, in ra thứ tương ứng với số đó (qui ước 2 là thứ hai, …, 8 là chủ nhật).

23. Nhập 2 số tháng và năm. In ra số ngày của tháng năm đó (có kiểm tra năm nhuận).
24. Lấy ngày tháng hiện tại làm chuẩn. Hãy nhập một ngày bất kỳ trong tháng. Cho biết thứ của ngày vừa nhập ?
25. Nhập vào tuổi cha và tuổi con hiện nay sao cho tuổi cha lớn hơn 2 lần tuổi con. Tìm xem bao nhiêu năm nữa tuổi cha sẽ bằng đúng 2 lần tuổi con (ví dụ 30 và 12, sau 6 năm nữa tuổi cha là 36 gấp đôi tuổi con là 18).
26. Nhập số nguyên dương N. Tính:

    a. $S_1 = \dfrac{1+2+3+...+N}{N}$

    b. $S_2 = \sqrt{1^2+2^2+3^2+...+N^2}$

27. Nhập số nguyên dương n. Tính:

    a. $S_1 = \sqrt{3+\sqrt{3+\sqrt{3+...+\sqrt{3}}}}$ n dấu căn

    b. $S_2 = \dfrac{1}{2+\dfrac{1}{2+\dfrac{1}{2+..\dfrac{1}{2}}}}$ n dấu chia

28. Nhập số tự nhiên n. In ra màn hình biểu diễn của n ở dạng nhị phân.
29. In ra màn hình các số có 2 chữ số sao cho tích của 2 chữ số này bằng 2 lần tổng của 2 chữ số đó (ví dụ số 36 có tích 3*6 = 18 gấp 2 lần tổng của nó là 3 + 6 = 9).
30. Số hoàn chỉnh là số bằng tổng mọi ước của nó (không kể chính nó). Ví dụ 6 = 1 + 2 + 3 là một số hoàn chỉnh. Hãy in ra màn hình tất cả các số hoàn chỉnh < 1000.
31. Các số sinh đôi là các số nguyên tố mà khoảng cách giữa chúng là 2. Hãy in tất cả cặp số sinh đôi < 1000.
32. In ra mã của phím bất kỳ được nhấn. Chương trình lặp cho đến khi nhấn ESC để thoát.

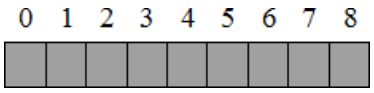
# CHƯƠNG III. CON TRỎ - MẢNG - XÂU KÍ TỰ

## III.1. Kiểu dữ liệu mảng

### III.1.1. Mảng một chiều

***a. Ý nghĩa***

Mảng là một dãy các phần tử có cùng kiểu được sắp kề nhau liên tục trong bộ nhớ. Tất cả các phần tử đều có cùng tên và đó là tên của mảng. Để phân biệt các phần tử với nhau, người ta gán chỉ số từ 0 cho đến hết mảng. Khi cần nói đến thành phần cụ thể nào của mảng, ta sẽ dùng tên mảng và kèm theo chỉ số của phần tử đó. Hình 3.1 minh họa một mảng có 9 phần tử, các phần tử được gán số từ 0 đến 8.




Hình 3.1

***b. Khai báo biến mảng***

Cách 1: Khai báo nhưng không khởi tạo giá trị

**<tên_kiểu> <tên_biến>[số_pt] ;**

Cách 2: Khai báo nhưng có khởi tạo giá trị

**<tên_kiểu> <tên_biến>[số_pt] = { dãy giá trị };**

Cách 3: Khai báo không có kích thước, có khởi tạo giá trị

**<tên_kiểu> <tên_biến>[] = { dãy giá trị } ;**

Trong đó *<tên_kiểu>* là kiểu dữ liệu của các phần tử, các phần tử này có kiểu giống nhau.

Cách 1, như cách khai báo biến bình thường nhưng thêm *<số_pt>* là số phần tử của mảng đặt giữa cặp dấu ngoặc vuông [], số phần tử còn được gọi là kích thước của mảng. Phần tử đầu tiên được gán chỉ số là 0, phần tử tiếp theo là 1, và tiếp tục cho đến hết. Như vậy, nếu mảng có n phần tử thì phần tử cuối cùng được gán chỉ số là n-1.

Cách 2, khai báo biến mảng, đồng thời khởi gán giá trị các phần tử mảng bằng dãy giá trị trong cặp dấu {}, mỗi giá trị cách nhau bởi dấu phảy, các giá trị được gán lần lượt bắt đầu từ phần tử thứ 0 cho đến hết dãy. Số giá trị có thể ít hơn số phần tử. Nhữn phần tử chưa có giá trị sẽ không được xác định cho đến khi được gán một giá trị nào đó.

Cách 3, khia báo biến mảng và cho phép vắng mặt *<số_pt>*, trường hợp này số phần tử mảng được xác định bởi số các giá trị của dãy khởi tạo. Việc vắng mặt cả dãy khởi tạo là không được phép (như vậy, khai báo *int a[]* là sai).

*Ví dụ 3.1*:

```
float x[100] , y[100]; // Khai báo biến chứa tọa độ 100 điểm trong mặt phẳng
long L[100] ; // Khai báo mảng L chứa được tối đa 100 số nguyên dài
char dong[80] ;//Khai báo mảng dong, mỗi dòng chứa được tối đa 80 kí tự
double data[] = {0,0,0,0,0};// dãy data chứa 5 số thực, khởi tạo giá trị là 0
```

***c. Cách sử dụng***

Để chỉ phần tử thứ i của một mảng ta viết tên mảng với chỉ số trong ngoặc vuông [].

Không thể thao tác trực tiếp với mảng mà phải thao tác với từng phần tử của mảng. Ví dụ, ta không thể nhập dữ liệu cho mảng a[10] bằng câu lệnh:

```
cin >> a ;     // sai
```

mà phải nhập cho từng phần tử từ a[0] đến a[9] của a. Trong trường hợp này, chúng ta phải cần đến lệnh lặp, ví dụ:

```
for (int i = 0 ; i < 10 ; i++) cin >> a[i] ;
```

***d. Ví dụ***

*Ví dụ 3.2*: Nhập dãy số nguyên, đếm số lượng: số dương, số âm, số 0 của dãy.

```
#include <iostream>
using namespace std;
int main()
{
    float a[50] ;     // a chứa tối đa 50 số
    int i, n, sd, sa, s0;
    cout << "Nhap so phan tu cua day: " ; cin >> n;
    for (i = 0; i < n; i++) {
        cout << "a[" << i << "] = " ; cin >> a[i];
    }
    sd = sa = s0 = 0 ;
    for (i=1; i<n; i++) {
        if (a[i] > 0 ) sd++;
        if (a[i] < 0 ) sa++;
        if (a[i] == 0 ) s0++;
    }
    cout << "Co "<<sd << " so duong; " << sa <<" so am; " ;
    cout << s0<<" so 0 ";
    return 0;
}
```

*Ví dụ 3.3*: Tìm số bé nhất của một dãy số. In ra số này và vị trí của nó trong dãy.

```
#include <iostream>
using namespace std;
int main()
{
    float a[100], min ;
    int i, n,  k;
    cout << "Nhap so phan tu: " ; cin >> n;
    for (i=0; i<n; i++) {
        cout << "a[" << i << "] = " ; cin >> a[i];
    }
    min = a[0]; k = 0; i=1;
    while (i<n){
        if (a[i] < min ) { min = a[i]; k = i; }
        i++;
    }
    cout << "So be nhat la " << min << " o vi tri " << k;
    return 0;
}
```

*Ví dụ 3.4*: Nhập và sắp xếp tăng dần một dãy số. Chương trình thực hiện sắp xếp bằng thuật toán sắp xếp đổi chỗ.

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

```
    float a[100], tmp;
    int i, j, n;
    cout << "Nhap so phan tu cua day: " ; cin >> n;
    for (i=0; i < n; i++) {
        cout << "a[" << i << "] = "; cin >> a[i];
    }
    for (i = 0; i < n - 1; i++)
        for (j = i + 1;j < n;j++)
           if (a[i]>a[j]){
               tmp = a[i]; a[i] = a[j];a[j] = tmp;
           }
    cout<<"Day so sau khi sap xep \n";
    for (i =0; i < n; i++) cout<<a[i]<<" ";
    return 0;
}
```

***e. Một số hàm hay dùng với mảng***

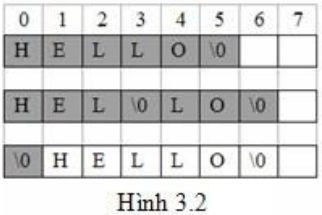
| Mẫu hàm | Ý nghĩa | Ví dụ |
|---|---|---|
| `sizeof(a)` | Cho biết kích thước a theo byte | `int a[6];`<br>`cout<<sizeof(a);`<br>`//24 = 6*4 byte` |
| `memcmp(p1,p2,n)` | So sánh n byte đầu hai khối nhớ trỏ bởi p1 và p2<br>- trả về 0 nếu hai khối bằng nhau<br>- trả về -1 nếu khối p1 < khối p2<br>- trả về 1 nếu khối p1 > khối p2 | *Lưu ý:*<br>- Để sử dụng các hàm này cần khai báo #include <cstring><br>- Các hàm này không tự kiểm tra lỗi tràn nhớ khi vùng đích không đủ chỗ chứa lượng dữ liệu. NSD phải tự tính toán. |
| `memcpy(p1,p2,n)` | Sao chép trực tiếp n byte trỏ bởi p2 sang vị trí trỏ bởi p1 | |
| `memmove(p1,p2,n)` | Sao chép gián tiếp n byte trỏ bởi p2 sang vị trí trỏ bởi p1. Dùng bộ nhớ trung gian. | |
| `memset(p,v,n)` | Điền vào vùng nhớ trỏ bởi p cùng một giá trị *unsigned char* v, phạm vi điền n byte. | |

*Ví dụ 3.5*: Minh họa cách dùng các hàm *memcpy, memmove, memset*

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
  int x[10] = {1,2,3,4,5,6,7,8,9,10},a[10];
  memset(a,0,sizeof(a));
  for(int i = 0; i < 10 ; i++) cout <<a[i]<<" "; cout<<"\n";
  memcpy ( a, x, sizeof(int)*10 );
  for(int i = 0; i < 10 ; i++) cout <<a[i]<<" "; cout<<"\n";
  memmove(a+5,a,sizeof(int)*5);
  for(int i = 0; i < 10 ; i++) cout <<a[i]<<" ";cout<<"\n";
  cout << memcmp(a,a+5,sizeof(int)*5)<<"\n";
  return 0;
}
```

### III.1.2. Mảng ký tự (C-string)

Đây là kiểu xâu của ngôn ngữ C nên còn có tên gọi là *C-string* để phân biệt với đối tượng *string* của C++. Cũng như C, C++ coi một mảng các ký tự như một *xâu kí tự*. Để C++ nhận biết được mảng kí tự là một xâu, mảng phải có kí tự kết thúc xâu, theo qui ước là kí tự có mã 0 (tức '\0'). Khi đó xâu là dãy kí tự bắt đầu từ phần tử đầu tiên (thứ 0) đến kí tự kết thúc xâu đầu tiên (không kể các kí tự còn lại trong mảng).



Hình 3.2

Hình 3.2 minh hoạ 3 xâu, mỗi xâu được chứa trong mảng kí tự có độ dài tối đa là 7. Xâu thứ nhất "*Hello*" có độ dài thực tế là 5 kí tự, chiếm 6 ô trong mảng (tính cả kí tự '\0'). Xâu thứ hai có nội dung "Hel" với độ dài 3 và xâu cuối cùng biểu thị một xâu rỗng (chiếm 1 ô).

### *a. Khai báo C-string*

```
Cách 1: char <tên xâu>[độ dài] ;                // không khởi tạo
Cách 2: char <tên xâu>[độ dài] = xâu kí tự ; // có khởi tạo
Cách 3: char <tên xâu>[] = xâu kí tự ;          // có khởi tạo
```

Độ dài xâu C-string là số kí tự tối đa có thể chứa trong xâu (không kể dấu kết thúc xâu '\0'). Do đó, khi khai báo độ dài xâu cần nhớ khai báo thừa ra một phần tử (độ dài tối đa của xâu + 1).

Ví dụ khai báo mảng s chứa được xâu có độ dài tối đa 80 kí tự, ta khai báo *char s[81]*.

Cách 2, kèm theo khởi tạo xâu, là dãy kí tự trong cặp dấu nháy kép. Ví dụ:

```
char hoten[26] ;      // xâu họ tên chứa tối đa 25 kí tự
char monhoc[31] = "NNLT C++" ;
```

xâu *monhoc* chứa tối đa 30 kí tự, được khởi tạo với nội dung "NNLT C++" với độ dài thực sự là 10 kí tự (chiếm 11 ô đầu tiên trong xâu *monhoc[31]*).

Cách khai báo thứ 3, C++ tự quyết định độ dài xâu bằng độ dài chuỗi khởi tạo. Ví dụ:

```
char thang[] = "Muoi hai" ;        // độ dài xâu = 9
```

### *b. Cách sử dụng C-string*

Xâu C-string có những đặc trưng như mảng, tuy nhiên chúng cũng có những điểm khác biệt. Dưới đây là các điểm giống và khác nhau đó.

- Truy nhập một kí tự trong xâu, cú pháp giống như mảng. Ví dụ:

```
char s[50] = "I\'m a student" ;    //chú ý kí tự ' phải được viết là \'
cout << s[0] ;                     // in kí tự đầu tiên, tức kí tự 'I'
s[1] = 'a' ;                       // gán kí tự thứ 2 là 'a'
```

- Không được thực hiện các phép toán trực tiếp trên xâu, ví dụ:

```
char s[20] = "Hello", t[20]; // đúng, khai báo hai xâu s và t, khởi tạo s
t = "Hello" ;                // sai, chỉ gán được khi khai báo khởi tạo
t = s ;                      // sai, không gán được
if (s < t) …                 // sai, không so sánh được hai xâu
…
```

### *c. Phương thức nhập xâu C-string*

Có thể dùng toán tử *cin* >>, nhưng hạn chế là xâu chỉ nhận được các ký tự trước dấu cách.

Ngoài ra, C++ cung cấp hàm *cin.getline(s,n,c)* để nhập xâu kí tự. Hàm có 2 đối: *s* là xâu cần nhập nội dung và *n-1* là số kí tự tối đa xâu có thể nhận. Hàm này trả về chuỗi s có n-1 ký tự, nếu nhập đủ. Nếu có tham số là ký tự c và dãy nhập vào có chứa ký tự c (ký tự giới hạn) thì s chỉ nhận được chuỗi ký tự từ đầu cho đến trước ký tự c.

*Ví dụ 3.6*:

```
char s[12] ;
cin.getline(s, 12);      // nhập Hello World ↵
cout << s << endl ;      // in ra Hell World
cin.getline(s, 12,'o'); // nhập Hello World
cout << s << endl ;      // in ra Hell
```

*Ví dụ 3.7*: Nhập một ngày tháng dạng Anh Mỹ (mm/dd/yy), đổi sang ngày tháng dạng Việt Pháp rồi in ra màn hình.

```
#include <iostream>
using namespace std;
int main()
{
    char US[9], VN[9] = "  /  /  " ; // khởi tạo trước hai dấu /
    cin.getline(US, 9) ;              // nhập ngày tháng, ví dụ "05/01/99"
    VN[0] = US[3]; VN[1] = US[4] ;    // ngày
    VN[3] = US[0]; VN[4] = US[1] ;    // tháng
    VN[6] = US[6]; VN[7] = US[7] ;    // năm
    cout << VN << endl ;
    return 0;
}
```

### *d. Một số hàm xử lí xâu C-string*

Không có các phép toán thao tác trực tiếp với xâu, nên chương trình dịch cung cấp sẵn các hàm thư viện. Các hàm này giải quyết hầu hết các thao tác trên xâu như: gán xâu, so sánh xâu, sao chép xâu, tính độ dài xâu, nhập/xuất …

Để sử dụng các hàm này, cần khai báo tệp tiêu đề *<cstring>*

| Mẫu hàm | Ý nghĩa | Ví dụ |
|---|---|---|
| `strcpy(s,t)` | sao chép các ký tự trong t sang s. Chú ý độ dài xâu s phải đủ chứa. | `t="Hello"; strcpy(s,t)`<br>`cout<<s; //Hello` |
| `strncpy(s,t,n)` | sao chép n ký tự đầu trong t sang s. NSD phải tự thêm '\0' vào cuối s trong trường hợp n<độ dài của t. Chú ý độ dài xâu s phải đủ chứa. | `char t[6]="Hello";`<br>`strncpy(s,t,4); s[5]='\0';`<br>`cout<<s; //Hell`<br>`strncpy(s,t+3,2);s[2]='\0';`<br>`cout<<s; //ll` |
| `strcat(s,t)` | sao chép các ký tự trong t nối vào cuối s. Chú ý độ dài xâu s phải đủ chứa. | `char s[6]="Hello",t[6]="World";`<br>`strcat(s,t);`<br>→ `s = ''HelloWorld"` |
| `strcmp(s,t)` | so sánh hai xâu, trả lại hiệu hai ký tự khác nhau đầu tiên trong s và t. Phân biệt hoa, thường<br>s < t thì hàm trả giá trị <0<br>s ==t thì hàm trả giá trị 0<br>s > t thì hàm trả giá trị >0 | `char s[6]="Hello",t[6]="World";`<br>`cout<<strcmp(s,t); // -1`<br>`cout<<strcmp(t,s); // 1`<br>`cout<<strcmp(s,"Hello"); // 0` |
| `strcmpi(s,t)` | Như strcmp() nhưng không phân biệt hoa, thường | |
| `strupr(s)` | Đổi các chữ cái trong xâu s thành in hoa | `char s[6]="Hello7"; strupr(s);`<br>`cout<<s; //HELLO7` |
| `strlwr(s)` | Đổi các chữ cái trong xâu s thành in thường | `char s[6]="Hello7"; strupr(s);`<br>`cout<<s; //hello7` |
| `strlen(s)` | Trả về độ dài xâu | `char s[6]="Hello";`<br>`cout<<strlen(s); //5` |
| `sizeof(s)` | Cho biết kích thước s theo byte | `char s[6]="Hello";`<br>`cout<<sizeof(s); //6,gồm cả'0\'` |

*Lưu ý:* Các hàm dùng cho mảng nêu trong phần *III.1.1.e* đều có thể dùng cho C-string.

*Ví dụ 3.8*: Thống kê số chữ 'a' xuất hiện trong xâu s.

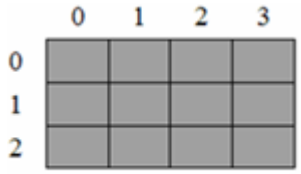
```
#include <iostream>
#include <cstring>
using namespace std;
int main()
```

```
{
    const int MAX = 100;
    char s[MAX+1];
    int sokitu = 0;
    cin.getline(s, MAX+1);
    for (int i=0; i < strlen(s); i++)
        if (s[i] = 'a') sokitu++;
    cout << "so ki tu a = " << sokitu << endl ;
    return 0;
}
```

*Ví dụ 3.9*: Sao chép xâu s sang xâu t (tương đương với hàm *strcpy(t,s)*)

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char s[100], t[100];
    cin.getline(s, 100);     // nhập xâu s
    int i=0;
    while ((t[i] = s[i]) != '\0') i++; //copy cả dấu kết thúc xâu '\0'
    cout << t << endl ;
    return 0;
}
```

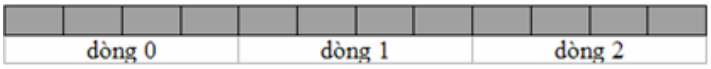
### III.1.3. Mảng hai chiều



Hình 3.3

Để thuận tiện trong việc biểu diễn các loại dữ liệu phức tạp như ma trận hoặc các bảng biểu có nhiều chỉ tiêu, C++ đưa ra kiểu dữ liệu mảng nhiều chiều. Tuy nhiên, sử dụng mảng nhiều chiều thường khó kiểm soát. Trong mục này chỉ trình bày mảng hai chiều. Nếu một mảng một chiều có m phần tử, mỗi phần tử của nó lại là mảng một chiều n phần tử thì ta gọi nó là mảng hai chiều với số phần tử (hay kích thước) của mảng là m*n. Ma trận là một minh hoạ cho hình ảnh của mảng hai chiều, nó gồm m dòng và n cột, chứa m*n phần tử. Các phần tử này phải có cùng kiểu. Tuy nhiên, về mặt bản chất mảng hai chiều không phải là một tập hợp với m*n phần tử cùng kiểu mà là tập hợp có m phần tử, trong đó mỗi phần tử là một mảng một chiều với n phần tử. Điểm nhấn mạnh này sẽ được giải thích cụ thể hơn trong các phần trình bày về con trỏ.

Hình 3.3 minh hoạ hình thức một mảng hai chiều với 3 dòng, 4 cột. Thực chất trong bộ nhớ 12 phần tử của mảng được sắp liên tiếp theo từng dòng của mảng như minh hoạ trong hình 3.4.



Hình 3.4

#### *a. Khai báo*

**<kiểu phần tử> <tên mảng>[m][n] ;**

- m là số hàng, n số cột của mảng.
- *kiểu phần tử* là kiểu của m x n phần tử trong mảng.
- Trong khai báo cũng có thể được khởi tạo bằng dãy các dòng giá trị, các dòng cách nhau bởi dấu phẩy, mỗi dòng được bao bởi cặp ngoặc {} và toàn bộ giá trị khởi tạo nằm trong cặp dấu {}.

***b. Sử dụng***

- Tương tự mảng một chiều, các chiều trong mảng cũng được đánh số từ 0.
- Không được thao tác trực tiếp với biến mảng mà phải thao tác với từng phần tử của mảng.
- Để truy nhập phần tử của mảng ta sử dụng tên mảng kèm theo 2 chỉ số chỉ vị trí hàng và cột của phần tử. Các chỉ số này có thể là các biểu thức thực, C++ tự chuyển kiểu sang nguyên.

*Ví dụ 3.10:* Khai báo 2 ma trận a và b có 3 hàng, 4 cột chứa các số nguyên như hình 3.5:

```
int a[3][4], b[3][4] ;
```

Khai báo có khởi tạo:

```
int a[3][4] = { {1,2,3,4}, {3,2,1,4}, {0,1,1,0} };
```

với khởi tạo này ta có ma trận, trong đó:

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 3 | 2 | 1 | 4 |
| 0 | 1 | 1 | 0 |

Hình 3.5

```
a[0][0] = 1, a[0][1] = 2, a[1][0] = 3, a[2][3] = 0
```

Trong khai báo có thể vắng số hàng (không được vắng số cột), số hàng phải được xác định thông qua khởi tạo.

```
float a[][3] = { {1,2,3}, {0,1,0} } ;
```

trong ví dụ trên, chương trình tự động xác định số hàng là 2. Phép khai báo và khởi tạo sau đây là cũng hợp lệ:

```
float a[][3] = { {1,2}, {0} } ;
```

chương trình cũng xác định số hàng là 2 và số cột (bắt buộc phải khai báo) là 3 mặc dù trong khởi tạo không xác định rõ số cột. Các phần tử chưa khởi tạo sẽ chưa được xác định cho đến khi nào nó được nhập hoặc gán giá trị cụ thể. Trong ví dụ trên các phần tử a[0][2], a[1][1] và a[1][2] là chưa được xác định.

***c. Ví dụ minh hoạ***

*Ví dụ 3.11*: Nhập, in và tìm phần tử lớn nhất của một ma trận.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    float a[10][10] ;   // mang hai chieu 10 hang, 10 cot
    int m, n ;
    int i, j ;
    int amax, imax, jmax ;
    cout << "Nhap so hang va cot: "; cin >> m >> n;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)   {
            cout << "a[" << i << "," << j << "]= ";
            cin >> a[i][j] ;
        }
    amax = a[0][0]; imax = 0; jmax = 0;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            if (amax < a[i][j]) {
                amax = a[i][j]; imax = i; jmax = j;
            }
     cout << setiosflags(ios::showpoint) << setprecision(2) ;
     cout << "Ma tran da nhap\n";
```

```
        for (i = 0; i < m; i++){
            for (j = 0; j < n; j++) cout << setw(8) << a[i][j] ;
            cout << "\n";
        }
        cout << "So lon nhat la: " << amax << endl;
        cout << "tai vi tri (" << imax << "," << jmax << ")";
        return 0;
    }
```

## III.2. Xâu ký tự của C++

Ngoài việc chấp nhận mảng ký tự như một xâu kiểu C-string trong phần trước. C++ cung cấp đối tượng *string* biểu diễn một xâu ký tự.

Kiểu *string* trong C++ là một đối tượng nằm trong thư viện STL (phần II của giáo trình sẽ trình bày về STL). Vì vậy, *string* có các phép toán và hàm riêng của nó.

Khi sử dụng ta vẫn coi biến kiểu *string* như một biến thông thường khác, biến *string* chứa một dãy các kí tự có phân biệt thứ tự, các phần tử không nhất thiết phải khác nhau. Để sử dụng kiểu string trong chương trình, bạn cần khai báo tệp tiêu đề *#include <string>*.

### *a. Khai báo sử dụng thư viện string*

```
#include <string>
```

### *b. Khai báo biến kiểu string*

- Cách 1: **`string <tên_biến>;`** `//Khai báo không khởi tạo`
- Cách 2: **`string <tên_biến>(str);`** `//Khai báo và khởi tạo giá trị là string str`
- Cách 3: **`string <tên_biến>(s, n);`** `//Khai báo, gán g.trị là C-string s với n ktự`

*Ví dụ 3.12:*

```
string myStr; // khai báo chuỗi myStr chưa có giá trị (rỗng).
string myStr("hello!"); // khai báo chuỗi myStr và gán cho giá trị "hello!"
char* myChar = new char[10];
string myStr(myChar, 10); //khai báo myStr và gán cho chuỗi C-string myChar
```

### *c. Các phép toán của string*

| Phép toán | Cú pháp, ý nghĩa | Ví dụ |
|---|---|---|
| = | <s1> = <s2>;<br>- Gán giá trị string *s2* cho string *s1*<br>- Có thể gán trực tiếp các kiểu cơ sở cho chuỗi. Phép gán có độ ưu tiên thấp hơn phép toán số học | string s1, s2="Hello"; s1=s2;<br>cout<< s1; //Hello<br>s1=12;<br>cout<< s1; //12 dạng chuỗi<br>s1=12+1.5;<br>cout<< s1; //13.5 dạng chuỗi |
| + hoặc += | <s1> = <s1> + <s2>;<br><s1> += <s2>;<br>- Nối giá trị s2 vào cuối s1 | string s1(12), s2(3);<br>s1+= s2;<br>cout<<s1; // 123 dạng chuỗi |
| >, <, >=, <=, ==, != | Là các phép so sánh trả về giá trị bool.<br>Các phép này so sánh mã ASCII của từng ký tự tương ứng giữa hai chuỗi tham gia so sánh cho đến khi có sự khác nhau. | string s1("Hello"), s2="Hell";<br>cout<<(s1<s2); //0 (false) |
| <<, >> | Xuất chuỗi và nhập chuỗi. Phép nhập chuỗi >> có hạn chế, nên thay bằng getline() | string s;<br>cin >> s; cout << |

***d. Phép duyệt string***

+ *Cách 1*: Thực hiện như phép duyệt C-string.

*Ví dụ 3.13*:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s1 = "Hello";
    for(int i = 0; i < s1.length(); i++) cout << s1[i] <<" ";
    return 0;
}
```

Chương trình in ra màn hình dòng ký tự: `H e l l o`

+ *Cách 2*: Sử dụng con trỏ duyệt *iterator* của string và các hàm thành viên *begin, end*.

*Ví dụ 3.14:* Chương trình sau cho kết quả tương tự ví dụ 3.13

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s1 = "Hello";
    string::iterator i;
    for(i = s1.begin(); i != s1.end() ; ++i)
        cout << *i <<" ";
}
```

Ngoài ra ta còn có thể dùng con trỏ *reverse_iterator* và các hàm thành viên *rbegin, rend* để duyệt ngược string.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s1 = "Hello";
    string::reverse_iterator i;
    for (i = s1.rbegin(); i != s1.rend(); ++i)  std::cout << *i;
}
```

Chương trình in ra màn hình xâu ký tự: `olleH`

***e. Các hàm thành viên của string***

Dưới đây là một số hàm hay dùng của string, trong phần nói về STL của giáo trình có nêu nhiều hàm hơn

| Nhóm hàm | Tên hàm | Ý nghĩa |
|---|---|---|
| Kích thước | **`size`** hoặc **`length`** | trả về số lượng phần tử của string. |
| | **`empty`** | trả về true(1) nếu string rỗng, ngược lại là false(0) |
| | **`clear`** | xóa string |
| | **`s[i]`** | trả về phần tử thứ i của biến string s. Không kiểm tra phạm vi |

| | | |
|---|---|---|
| | **at(i)** | trả về phần tử thứ i string s. Có kiểm tra phạm vi |
| Thêm bớt phần tử | **toán tử +/+=** | nối hai string. Ví dụ: *s1 = s1+ s2* hoặc *s1 += s2;* |
| | **push_back(x)** | thêm ký tự x vào cuối string |
| | **insert(i,x)** | chèn *x* vào trước vị trí *i* của string (x có thể là kiểu string/C string) |
| | **erase(i,n)** | xóa n phần tử tính từ vị trí i của string |
| | **replace(i,n,s)** | Thay thế n phần tử tính từ vị trí i của string bằng s (s có thể là kiểu string/C string/char) |
| Thao tác khác với string | **c_str** | trả về chuỗi giá trị của string ở dạng C string (đổi từ str → C string) |
| | **copy(s,n,i)** | trả về một chuỗi s dạng C string là một phần của string, gồm n ký tự, tính từ vị trí i (mặc định i = 0). Không tự thêm '\0' vào cuối s. |
| | **find(s,i)** | trả về vị trí đầu tiên xuất hiện chuỗi s trong string, tính từ vị trí i (mặc định i=0, s có thể là kiểu string/C string/char) |
| | **rfind(s,i)** | tương tự hàm find nhưng tìm từ cuối string trở lại |
| hằng | **npos** | Cho biết kết thúc chuỗi. Thường có giá trị -1, thường để chỉ kết quả tìm kiếm thất bại (không tìm thấy) |

*Ví dụ 3.15:* Để minh hoạ một số thao tác cơ bản trên lớp chuỗi, ví dụ này thực hiện việc chuẩn hóa một chuỗi fname họ tên người Việt (loại bỏ dấu cách thừa, viết hoa đầu các từ) và tách riêng họ, đệm, tên.

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;
main ()
{
    int i, j;
    string fname, first, last, sur;
    first = last = sur = "";
    cout<<"Nhap chuoi ho ten nguoi Viet:"; getline(cin,fname,'\n');
    cout<<"Chuoi ho ten truoc chuan hoa:" << fname <<"\n";
    //Cat bo cac dau cach thua trong fname
    while((j=fname.find(' ')) == 0) fname.erase(j,1);
    while((j=fname.rfind(' ')) == fname.size()-1) fname.erase(j,1);
    while((j=fname.find("  ")) != string::npos) fname.erase(j,1);
    //Viet hoa dau tu
    fname.replace(0,1,1,toupper(fname[0]));
    for (j = 0; j < fname.size(); j++)
        if(fname[j] == ' ')
            fname.replace(j+1,1,1,toupper(fname[j+1]));
    cout<<"Chuoi ho ten da chuan hoa:";
    cout << fname <<" "<<'\n';
    j = fname.rfind(' ');
    last = fname.substr(j+1,fname.size()-j-1); //tach lay ten
    i = fname.find(' ');
    if (j != i) first = fname.substr(0,i);    //tach lay ho
    if (i != fname.npos && j!=fname.npos)
        sur = fname.substr(i+1,j-i);          //tach lay dem
    cout<<"\nHo="<<first<<"\nDem="<<sur<<"\nTen="<<last;
}
```

## III.3. Con trỏ và địa chỉ

Trong phần này, chúng ta sẽ nói về một loại biến mới gọi là con trỏ, ý nghĩa, công dụng và cách dùng. Con trỏ là một đặc trưng mạnh của C++, nó cho phép ta thâm nhập trực tiếp vào bộ nhớ để xử lý thao tác phức tạp bằng chỉ vài câu lệnh đơn giản của chương trình. Điều này góp phần làm cho C++ trở thành ngôn ngữ gần gũi với các ngôn ngữ cấp thấp như hợp ngữ. Tuy nhiên, vì tính đơn giản, ngắn gọn nên việc sử dụng con trỏ đòi hỏi sự cẩn thận và kinh nghiệm trong lập trình.

### III.3.1. Địa chỉ, phép toán &

Mỗi biến (với tên biến) của chương trình được gắn với một địa chỉ của byte đầu tiên mà biến đó được phân phối. Số lượng các byte phân phối cho biến là khác nhau (nhưng đặt liền nhau từ thấp đến cao) tuỳ thuộc kiểu dữ liệu của biến (và tuỳ thuộc từng NNLT). Bạn chỉ cần biết tên biến hoặc địa chỉ của biến là có thể đọc/viết dữ liệu vào/ra các biến đó. Như vậy, ngoài việc sử dụng biến thông qua tên, ta còn có thể sử dụng biến thông qua địa chỉ của chúng. Tóm lại biến, ô nhớ và địa chỉ có quan hệ khăng khít với nhau. C++ cung cấp toán tử một ngôi & để lấy địa chỉ của các biến. Nếu *x* là một biến thì *&x* trả về địa chỉ của *x*. Lệnh sau cho ta biết địa chỉ của biến *x* trong bộ nhớ:

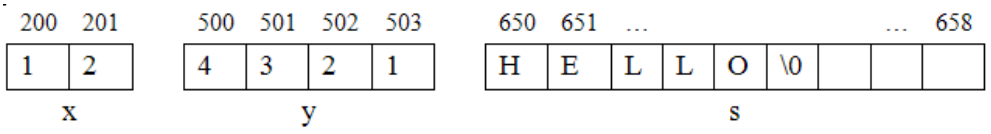
```
int x ;
cout << &x ;      // địa chỉ hiện dưới dạng cơ số 16. Ví dụ 0xfff4
```

Với biến kiểu mảng, thì tên mảng chính là địa chỉ của mảng, do đó không cần dùng đến toán tử &. Ví dụ địa chỉ của mảng *a* chính là *a* (không phải là *&a*). Mặt khác địa chỉ của mảng *a* cũng chính là địa chỉ của byte đầu tiên mà mảng *a* chiếm và nó cũng chính là địa chỉ của phần tử đầu tiên của mảng *a*. Do vậy địa chỉ của mảng *a* là địa chỉ của phần tử *a[0]* và là *&a[0]*.

Tóm lại, cần nhớ:

```
int x;                // khai báo biến nguyên x
long y;               // khai báo biến nguyên dài y
cout << &x << &y;     // in địa chỉ các biến x, y
char s[9];            // khai báo mảng kí tự s
cout << a;            // in địa chỉ mảng s
cout << &a[0];        // in địa chỉ mảng s (tức địa chỉ s[0])
cout << &a[2];        // in địa chỉ kí tự s[2]
```

Hình 3.6 minh hoạ một vài biến và địa chỉ của nó trong bộ nhớ.




Biến x chiếm 2 byte, có địa chỉ là 200, biến y có địa chỉ là 500 và chiếm 4 byte. Xâu s chiếm 9 byte tại địa chỉ 650. Các byte bộ nhớ của một biến là liên tục.

Hình 3.6

Các phép toán liên quan đến địa chỉ được gọi là *số học địa chỉ*. Tuy nhiên, chúng ta không được phép thao tác trực tiếp trên các địa chỉ, như: đặt biến vào địa chỉ này hay khác (*công việc này do chương trình dịch đảm nhiệm*), hay việc cộng, trừ hai địa chỉ với nhau là vô nghĩa… Các thao tác được phép với địa chỉ phải thông qua biến trung gian chứa địa chỉ, được gọi là *biến con trỏ*.

## III.3.2. Con trỏ

### *a. Ý nghĩa*

- Con trỏ là một biến chứa địa chỉ của biến khác. Nếu *p* là con trỏ chứa địa chỉ của biến *x* ta bảo *p* trỏ tới *x* và *x* được trỏ bởi *p*. Thông qua con trỏ ta có thể làm việc được với nội dung của những ô nhớ mà *p* trỏ đến.
- Để con trỏ *p* trỏ tới *x* ta phải gán địa chỉ của *x* cho *p*.
- Để làm việc với địa chỉ của các biến cần phải thông qua các biến con trỏ trỏ đến biến đó.

### *b. Khai báo biến con trỏ*

```
<kiểu được trỏ> <*tên biến> ;
```

Địa chỉ của một biến là địa chỉ byte nhớ đầu tiên của biến đó. Để lấy được nội dung của biến, con trỏ phải quản lý được số byte của biến, tức kiểu của biến mà nó trỏ tới. Kiểu này cũng được gọi là kiểu của con trỏ. Như vậy khai báo biến con trỏ cũng giống như khai báo một biến thường ngoại trừ cần thêm dấu * trước tên biến (hoặc sau tên kiểu). Ví dụ:

```
int *p ;          // khai báo p là biến con trỏ trỏ đến kiểu dữ liệu nguyên.
float *q, *r ;    // hai con trỏ thực q và r.
```

### *c. Sử dụng con trỏ, phép toán **

- Để con trỏ *p* trỏ đến biến *x* ta phải dùng phép gán *p = địa chỉ của x*.
  - Nếu x không phải là mảng ta viết: *p = &x*
  - Nếu x là mảng ta viết: *p = x* hoặc *p = &x[0]*
- Không gán *p* cho một hằng địa chỉ cụ thể. Ví dụ viết *p* = 200 là sai.
- Phép toán * cho phép lấy nội dung nơi *p* trỏ đến, ví dụ để gán nội dung nơi *p* trỏ đến cho biến *f* ta viết *f = *p*.
- & và * là 2 phép toán ngược nhau. Cụ thể nếu *p = &x* thì *x = *p*. Từ đó nếu *p* trỏ đến *x* thì bất kỳ nơi nào xuất hiện *x* đều có thể thay được bởi **p* và ngược lại.

*Ví dụ 3.16:*

```
int i, j ;            // khai báo 2 biến nguyên i, j
int *p, *q ;          // khai báo 2 con trỏ nguyên p, q
p = &i;               // cho p trỏ tới i
q = &j;               // cho q trỏ tới j
cout << &i ;          // hỏi địa chỉ biến i
cout << q ;           // hỏi địa chỉ biến j (thông qua q)
i = 2;                // gán i bằng 2
*q = 5;               // gán j bằng 5 (thông qua q)
i++ ; cout << i ;     // tăng i và hỏi i, i = 3
(*q)++ ; cout << j ; // tăng j (thông qua q) và hỏi j, j = 6
(*p) = (*q) * 2 + 1; // gán lại i (thông qua p)
cout << i ;           // 13
```

Qua ví dụ trên ta thấy mọi thao tác với *i* là tương đương với **p*, với *j* là tương đương với **q* và ngược lại.

***d. Các phép toán với con trỏ***

Trên đây ta đã trình bày về 2 phép toán một ngôi liên quan đến địa chỉ và con trỏ là & và *. Phần này chúng ta tiếp tục xét với các phép toán khác làm việc với con trỏ.

***+ Phép gán***

- Gán con trỏ với địa chỉ một biến: *p = &x* ;
- Gán con trỏ với con trỏ khác: *p = q* ; (sau phép toán gán này *p, q* chứa cùng một địa chỉ, cùng trỏ đến một nơi).

*Ví dụ 3.17:*

```
int i = 10 ;                    // khai báo và khởi tạo biến i = 10
int *p, *q, *r ;                // khai báo 3 con trỏ nguyên p, q, r
p = q = r = &i ;                // cùng trỏ tới i
*p = q**q + 2**r + 1 ;          // i = 10*10 + 2*10 + 1
cout << i ;                     // 121
```

***+ Phép toán tăng / giảm địa chỉ***

`p ±n:` con trỏ trỏ đến thành phần thứ *n* sau (trước) *p*.

Một đơn vị tăng giảm của con trỏ bằng kích thước của biến được trỏ. Ví dụ giả sử *p* là con trỏ nguyên (2 byte) đang trỏ đến địa chỉ 200 thì *p+1* là con trỏ trỏ đến địa chỉ 202. Tương tự, *p + 5* là con trỏ trỏ đến địa chỉ 210. *p – 3* trỏ đến địa chỉ 194.

| 194 | 195 | 196 | 197 | 198 | 199 | 200 | 201 | 202 |
|---|---|---|---|---|---|---|---|---|
| p - 3 | | | | | | p | | p + 1 |

Hình 3.7

Như vậy, phép toán tăng, giảm con trỏ cho phép làm việc thuận lợi trên mảng. Nếu con trỏ đang trỏ đến mảng (tức đang chứa địa chỉ phần tử đầu tiên của mảng), việc tăng con trỏ lên 1 đơn vị sẽ dịch chuyển con trỏ trỏ đến phần tử thứ hai, … Từ đó ta có thể cho con trỏ chạy từ đầu đến cuối mảng bằng cách tăng con trỏ lên từng đơn vị như trong câu lệnh for dưới đây.

*Ví dụ 3.18:*

```
int a[100] = { 1, 2, 3, 4, 5, 6, 7 }, *p, *q;
p = a; cout << *p ;           // cho p trỏ đến mảng a, *p = a[0] = 1
p += 5; cout << *p ;          // *p = a[5] = 6 ;
q = p - 4 ; cout << *q ;      // q = a[1] = 2 ;
for (int i=0; i<100; i++) cout << *(p+i); // in toàn bộ mảng a
```

***+ Phép toán tự tăng / giảm***

`p++, p--, ++p, --p`: tương tự p+1 và p-1, có chú ý đến tăng (giảm) trước, sau.

*Ví dụ 3.19:* Minh hoạ kết quả kết hợp phép tự tăng giảm với lấy giá trị nơi con trỏ trỏ đến. *a* là một mảng có 2 phần tử, *p* là con trỏ trỏ đến mảng *a*. Các lệnh dưới đây được qui ước là độc lập với nhau (tức lệnh sau không bị ảnh hưởng bởi lệnh trước, đối với mỗi lệnh *p* luôn luôn trỏ đến phần tử đầu (*a[0]*) của *a*.

```
int a[2] = {3, 7}, *p = a;
(*p)++ ;   // tăng (sau) giá trị nơi p trỏ ≡ tăng a[0] thành 4
++(*p) ;   // tăng (trước) giá trị nơi p trỏ ≡ tăng a[0] thành 4
*(p++) ;   // lấy giá trị nơi p trỏ (3) và tăng trỏ p (tăng sau), p → a[1]
*(++p) ;   // tăng trỏ p (tăng trước), p → a[1] và lấy giá trị nơi p trỏ (7)
```

*Chú ý:*

- Phân biệt *p+1* và *p++* (hoặc *++p*):
  - *p+1* được xem như một con trỏ khác với *p*. *p+1* trỏ đến phần tử sau *p*.
  - *p++* là con trỏ *p* nhưng trỏ đến phần tử khác. *p++* trỏ đến phần tử đứng sau phần tử *p* trỏ đến ban đầu.
- Phân biệt **(p++)* và **(++p)*:

Các phép toán tự tăng giảm cũng là một ngôi, mức ưu tiên của chúng là cao hơn các phép toán hai ngôi khác và cao hơn phép lấy giá trị (*). Cụ thể:

```
*p++      ≡   *(p++)
*++p      ≡   *(++p)
++*p      ≡   ++(*p)
```

Cũng giống các biến nguyên việc kết hợp các phép toán này với nhau rất dễ gây nhầm lẫn, do vậy cần sử dụng cặp dấu ngoặc để qui định trình tự tính toán.

***+ Hiệu của 2 con trỏ***

Phép toán hiệu của con trỏ *p* với con trỏ *q* này chỉ thực hiện được khi *p* và *q* cùng trỏ đến các phần tử của một dãy dữ liệu nào đó trong bộ nhớ (ví dụ cùng trỏ đến một mảng). Khi đó hiệu *p - q* là số thành phần giữa *p* và *q* (*p - q* không phải hiệu của 2 địa chỉ mà là số thành phần giữa *p* và *q*).

Ví dụ, giả sử p và q là 2 con trỏ nguyên (2 byte), *p* có địa chỉ 200 và *q* có địa chỉ 208. Khi đó *p - q = 4* và *q - p = 4* (4 là số thành phần nguyên từ địa chỉ 200 đến 208).

***+ Phép toán so sánh***

Các phép toán so sánh cũng được áp dụng đối với con trỏ, thực chất là so sánh giữa địa chỉ của hai nơi được trỏ bởi các con trỏ này. Thông thường các phép so sánh <, <=, >, >= chỉ áp dụng cho hai con trỏ trỏ đến phần tử của cùng một mảng dữ liệu nào đó. Thực chất của phép so sánh này chính là so sánh chỉ số của 2 phần tử được trỏ bởi 2 con trỏ đó.

*Ví dụ 3.20:*

```
float a[100], *p, *q ;
p = a ;                    // p trỏ đến mảng (tức p trỏ đến a[0])
q = &a[3] ;                // q trỏ đến phần tử thứ 3 (a[3]) của mảng
cout << (p < q) ;          // 1
cout << (p + 3 == q) ; // 1
cout << (p > q - 1) ; // 0
cout << (p >= q - 2) ; //0
for (p=a ; p < a+100; p++) cout << *p ;// in toàn bộ mảng a
```

### III.3.3. Cấp phát động, toán tử cấp phát bộ nhớ new và thu hồi bộ nhớ delete

Khi chạy chương trình, chương trình dịch bố trí các ô nhớ cụ thể và cố định cho các biến được khai báo trong chương trình. Các ô nhớ này tồn tại trong suốt thời gian chạy chương trình, chúng xem như đã bị chiếm dụng và sẽ không được sử dụng vào mục đích khác và chỉ được giải phóng sau khi chấm dứt chương trình. Việc phân bổ bộ nhớ như vậy gọi là *cấp phát tĩnh*. Ví dụ nếu khai báo: *int a[1000];* thì *a* sẽ chiếm một vùng nhớ liên tục 4000 byte để chứa dữ liệu. Dù chương trình chỉ sử dụng và làm việc với một vài phần tử của *a* thì phần mảng còn lại vẫn không được sử dụng vào việc khác. Đây là hạn chế thứ nhất mảng tĩnh. Một vấn đề khác, tại thời điểm nào đó

chương trình lại cần làm việc với hơn 1000 số nguyên. Khi đó vùng nhớ đã cấp cho *a* là không đủ dùng. Đây là hạn chế thứ hai của mảng tĩnh.

### *a. Cấp phát động*

Khắc phục các hạn chế trên của kiểu mảng, ta không khai báo mảng với kích thước cố định như vậy. Kích thước cụ thể sẽ được cấp phát ngay trong quá trình chạy chương trình theo đúng yêu cầu của NSD. Nhờ vậy chúng ta có đủ số ô nhớ để làm việc mà vẫn tiết kiệm được bộ nhớ, và khi không dùng nữa ta có thể thu hồi (còn gọi là giải phóng) số ô nhớ này để chương trình sử dụng vào việc khác. Hai công việc cấp phát và thu hồi này được thực hiện thông qua các toán tử *new*, *delete* và con trỏ *p*. Thông qua *p* ta có thể làm việc với bất kỳ địa chỉ nào của vùng được cấp phát. Cách thức bố trí bộ nhớ như thế này được gọi là *cấp phát động*.

### *b. Cú pháp của câu lệnh new cấp phát bộ nhớ động*

```
p = new <kiểu> ;              // cấp phát 1 phần tử
p = new <kiểu>[n] ;           // cấp phát n phần tử
```

*Ví dụ 3.21:*

```
int *p ;
p = new int   ;       // cấp phát vùng nhớ chứa được 1 số nguyên
p = float int[100] ; // cấp phát vùng nhớ chứa được 100 số thực
```

Khi gặp toán tử *new*, chương trình sẽ tìm trong bộ nhớ một lượng ô nhớ còn rỗi và liên tục với số lượng đủ theo yêu cầu và cho *p* trỏ đến địa chỉ (byte đầu tiên) của vùng nhớ này. Nếu không có vùng nhớ với số lượng như vậy thì việc cấp phát thất bại và *p* == NULL (NULL là một địa chỉ rỗng, không xác định). Để kiểm tra việc cấp phát có thành công hay không, ta kiểm tra con trỏ p bằng hay khác NULL.

*Ví dụ 3.22*:

```
float *p ;
int n ;
cout << "Số lượng cần cấp phát = ";
cin >> n;
p = new double[n];
if (p == NULL) {
    cout << "Không đủ bộ nhớ" ;
    return 0 ;
}
```

### *b. Cú pháp của câu lệnh delete thu hồi bộ nhớ động.*

Để giải phóng bộ nhớ đã cấp phát cho một biến, ta sử dụng câu lệnh *delete*. Cú pháp:

```
delete p ;          // p là con trỏ được sử dụng trong new
```

và để giải phóng toàn bộ mảng được cấp pháp thông qua con trỏ p ta dùng cú pháp:

```
delete[] p ;        // p là con trỏ trỏ đến mảng
```

Dưới đây là ví dụ sử dụng tổng hợp các phép toán trên con trỏ.

*Ví dụ 3.23:* Nhập dãy số (không dùng mảng), sắp xếp và in ra màn hình.

Trong ví dụ này chương trình xin cấp phát bộ nhớ đủ chứa *n* số nguyên và được trỏ bởi con trỏ *head*. Khi đó địa chỉ của số nguyên đầu tiên và cuối cùng sẽ là *head* và *head+n-1*. Với hai con trỏ *p* và *q* trỏ đến dãy số, chương trình sau sẽ sắp thành dãy n số nguyên theo trật tự giá trị tăng dần và in ra kết quả.

```
#include <iostream>
using namespace std;
int main()
{
    int *head, *p, *q, n, tam;     // head trỏ đến đầu dãy
    cout << "Cho biet so so hang cua day: "; cin >> n ;
    head = new int[n] ;            // cấp phát bộ nhớ chứa n số nguyên
    for (p = head; p < head + n; p++){
        cout << "Nhap so thu " << p - head + 1 << ": " ;
        cin >> *p ;
    }
    for (p = head; p < head + n - 1; p++)     // sắp xếp
    for (q = p + 1; q < head + n; q++)
        if (*q < *p) { tam = *p; *p = *q; *q = tam; } // đổi chỗ
    for (p = head; p < head + n; p++)
        cout << *p <<" " ;          // in kết quả
    return 0;
}
```

### III.3.4. Con trỏ và mảng, xâu kí tự

#### *a. Con trỏ và mảng 1 chiều*

Việc cho con trỏ trỏ đến mảng cũng tương tự trỏ đến các biến khác, tức gán địa chỉ của mảng (chính là tên mảng) cho con trỏ. Chú ý rằng địa chỉ của mảng cũng là địa chỉ của thành phần thứ 0 nên *a+i* sẽ là địa chỉ thành phần thứ *i* của mảng. Tương tự, nếu *p* trỏ đến mảng *a* thì *p+i* là địa chỉ thành phần thứ *i* của mảng *a* và do đó **(p+i) = a[i] = *(a+i)*.

Chú ý khi viết **(p+1) = *(a+1)* ta thấy vai trò của *p* và *a* trong biểu thức này là như nhau, cùng truy cập đến giá trị của phần tử *a[1]*. Tuy nhiên khi viết **(p++)* thì lại khác với **(a++)*, cụ thể viết *p++* là hợp lệ còn *a++* là không được phép. Lý do là tuy *p* và *a* cùng thể hiện địa chỉ của mảng *a* nhưng *p* thực sự là một biến, nó có thể thay đổi được giá trị còn *a* là một hằng, giá trị không được phép thay đổi. Ví dụ viết *x* = 3 và sau đó có thể tăng *x* bằng *x++* nhưng không thể viết *x* = 3++.

*Ví dụ 3.24:* In toàn bộ mảng thông qua con trỏ.

```
int a[5] = {1,2,3,4,5}, *p, i;
```

Phương án 1:

```
p = a; for (i=1; i<=5; i++) cout << *(p+i); // p không thay đổi
```

hoặc:

Phương án 2:

```
for (p=a; p<=a+4; p++) cout << *p ;     // thay đổi p
```

Trong phương án 1, con trỏ p không thay đổi trong suốt quá trình làm việc của lệnh *for*, để truy nhập đến phần tử thứ *i* của mảng *a* ta sử dụng cú pháp **(p+i)*.

Đối với phương án 2, con trỏ sẽ dịch chuyển dọc theo mảng *a* bắt đầu từ địa chỉ *a* (phần tử đầu tiên) đến phần tử cuối cùng. Tại bước thứ *i*, *p* sẽ trỏ vào phần tử *a[i]*, do đó ta chỉ cần in giá trị **p*. Để kiểm tra khi nào *p* đạt đến phần tử cuối cùng, ta có thể so sánh *p* với địa chỉ cuối mảng chính là địa chỉ đầu mảng cộng thêm số phần tử trong *a* trừ đi 1 (tức *a*+4 trong ví dụ trên).

***b. Con trỏ và xâu kí tự***

Một con trỏ kí tự có thể xem như một biến xâu kí tự, trong đó xâu là tất cả các kí tự kể từ byte đầu tiên có con trỏ trỏ đến cho đến byte '\0' nó gặp đầu tiên. Vì vậy ta có thể khai báo xâu dưới dạng con trỏ kí tự như sau.

```
char *s ;
char *s = "Hello" ;
```

Các hàm trên xâu vẫn được sử dụng như khi ta khai báo nó dưới dạng mảng kí tự. Ngoài ra, khác với mảng kí tự, ta được phép sử dụng phép gán cho hai xâu dưới dạng con trỏ, ví dụ:

```
char *s, *t = "Tin học" ; s = t;  // thay cho hàm strcpy(s, t) ;
```

Thực chất phép gán trên chỉ là gán 2 con trỏ với nhau, nó cho phép *s* bây giờ cũng được trỏ đến nơi mà *t* trỏ (tức dãy kí tự "*Tin học*" đã có sẵn trong bộ nhớ).

Khi khai báo xâu dạng con trỏ nó vẫn chưa có bộ nhớ cụ thể, vì vậy thông thường kèm theo khai báo ta cần phải xin cấp phát bộ nhớ cho xâu với độ dài cần thiết.

*Ví dụ 3.25*:

```
char *s = new char[30], *t ;
strcpy(s, "Hello") ; // trong trường hợp này không cần cấp phát bộ
t = s ;              // nhớ cho t vì t và s cùng sử dụng chung vùng nhớ
```

nhưng:

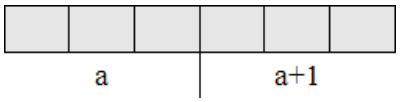
```
char *s = new char[30], *t ;
strcpy(s, "Hello") ;
t = new char[30];    // trong trường hợp này phải cấp bộ nhớ cho t vì
strcpy(t, s) ;       // có chỗ để strcpy sao chép sang nội dung của s.
```

***c. Con trỏ và mảng hai chiều***

Để dễ hiểu việc sử dụng con trỏ trỏ đến mảng hai chiều, chúng ta nhắc lại về mảng 2 chiều thông qua ví dụ. Giả sử ta có khai báo:

```
float a[2][3], *p;
```

khi đó *a* được bố trí trong bộ nhớ như là một dãy 6 phần tử float như hình 3.8:




Hình 3.8

tuy nhiên *a* không được xem là mảng 1 chiều với 6 phần tử mà được quan niệm như mảng một chiều gồm 2 phần tử, mỗi phần tử là 1 mảng 1 chiều có 3 phần tử là số thực. Do đó địa chỉ của mảng *a* chính là địa chỉ của phần tử đầu tiên a[0][0], và *a+1* không phải là địa chỉ của phần tử tiếp theo *a[0][1]* mà là địa chỉ của phần tử *a[1][0]*. Nói cách khác *a+1* là tăng địa chỉ của *a* lên một thành phần, nhưng 1 thành phần ở đây được hiểu là cả một dãy gồm 3 phần tử.

Mặt khác, việc lấy địa chỉ của từng phần tử (float) trong *a* thường là không chính xác. Ví dụ: viết *&a[i][j]* (địa chỉ của phần tử dòng i cột j) là được đối với mảng nguyên nhưng lại không đúng đối với mảng thực.

Từ các thảo luận trên, phép gán *p = a* là dễ gây nhầm lẫn vì *p* là con trỏ float còn *a* là địa chỉ mảng (1 chiều). Do vậy trước khi gán ta cần ép kiểu của *a* về kiểu float. Tóm lại cách gán địa chỉ của *a* cho con trỏ *p* được thực hiện như sau:

Các cách đúng:

```
p = (float*)a; // ép kiểu của a về con trỏ float (cũng là kiểu của p)
p = a[0];      // gán với địa chỉ của mảng a[0]
p = &a[0][0];  // gán với địa chỉ số thực đầu tiên trong a
```

Cách sai:

```
p = a ;        // sai vì khác kiểu
```

trong đó cách dùng *p = (float*)a*; là trực quan và đúng trong mọi trường hợp nên được dùng thông dụng hơn cả.

Sau khi gán *a* cho *p* (*p* là con trỏ thực), việc tăng giảm *p* chính là dịch chuyển con trỏ trên từng phần tử (thực) của *a*. Tức:

```
p       trỏ tới   a[0][0]
p+1     trỏ tới   a[0][1]
p+2     trỏ tới   a[0][2]
p+3     trỏ tới   a[1][0]
p+4     trỏ tới   a[1][1]
p+5     trỏ tới   a[1][2]
```

Tổng quát, đối với mảng *m x n* phần tử:

`p + i*n + j` trỏ tới `a[i][j]` hoặc `a[i][j] = *(p + i*n + j)`

Từ đó để truy nhập đến phần tử *a[i][j]* thông qua con trỏ *p* ta nên sử dụng cách viết sau:

```
p = (float*)a;
cin >> *(p+i*n+j) ;      // nhập cho a[i][j]
cout << *(p+i*n+j);      // in a[i][j]
```

*Ví dụ 3.26:* Nhập và in một mảng 2 chiều m*n (m dòng, n cột) thông qua con trỏ p. Nhập liên tiếp m*n số vào mảng và in thành ma trận m dòng, n cột.

```
#include <iostream>
using namespace std;
int main()
{
    int i, j, m=2, n=4;
    float a[m][n], *p;
    p = (float*) a;
    for (i = 0; i < m * n; i++) cin >> *(p+i); // nhập mảng mxn phần tử
    *(p + 2) = *(p + n + 2) = 100;  // gán a[0,2] = a[1][2] = 100
     for (i = 0; i < m; i++)         // in lại dưới dạng ma trận
     {
         for (j=0; j<n; j++) cout << *(p + i * n + j)<<" ";
         cout << endl;
     }
    return 0;
}
```

***d. Mảng con trỏ***

Cũng giống như các biến thông thường, nhiều biến cùng kiểu có thể tổ chức thành một mảng với tên gọi chung, nhiều con trỏ cùng kiểu cũng tổ chức được thành mảng. Như vậy mỗi phần tử của mảng con trỏ là một con trỏ trỏ đến một mảng nào đó. Nói cách khác một mảng con trỏ cho phép quản lý nhiều mảng dữ liệu cùng kiểu. Cách khai báo:

**<kiểu> <*con_trỏ>[size];**

Ví dụ: `int *a[10];` khai báo một mảng chứa 10 con trỏ. Mỗi con trỏ a[i] chứa địa chỉ của một mảng nguyên.

***e. Mảng xâu kí tự***

Là trường hợp riêng của mảng con trỏ nói chung, trong đó kiểu cụ thể là char. Mỗi thành phần mảng là một con trỏ trỏ đến một xâu kí tự, có nghĩa các thao tác tiến hành trên *a[i] như đối với một xâu kí tự.

*Ví dụ 3.27:* Nhập vào và in ra một bài thơ.

```
#include <iostream>
using namespace std;
int main()
{
    char *dong[100]; int i, n;
    cout << "so dong = "; cin >> n ;    // nhập số dòng thực sự
    cin.ignore();         // loại dấu ENTER trong lệnh cin ở trên
    for (i=0; i<n; i++) {
        dong[i] = new char[80];   // cấp bộ nhớ cho dòng i
        cin.getline(dong[i],80);  // nhập dòng i
    }
    for (i=0; i<n; i++) cout << dong[i] << endl;  // in kết quả
    return 0;
}
```

## III.4. Bài tập

### III.4.1. Mảng

**1.** Nhập vào dãy n số thực. Tính tổng dãy, trung bình dãy, tổng các số âm, dương và tổng các số ở vị trí chẵn, vị trí lẻ trong dãy. Tìm phần tử gần số trung bình nhất của dãy.

**2.** Tìm và chỉ ra vị trí xuất hiện đầu tiên của phần tử x trong dãy.

**3.** Nhập vào dãy n số. Hãy in ra số lớn nhất, bé nhất của dãy.

**4.** Nhập vào dãy số. In ra dãy đã được sắp xếp tăng dần, giảm dần.

**5.** Cho dãy đã được sắp tăng dần. Chèn thêm vào dãy phần tử x sao cho dãy vẫn sắp xếp tăng dần.

**6.** Hãy nhập vào 16 số nguyên. In ra thành 4 dòng, 4 cột.

**7.** Nhập ma trận A và in ra ma trận đối xứng của nó.

**8.** Cho một ma trận nguyên kích thước m*n. Tính:

- Tổng tất cả các phần tử của ma trận.
- Tổng tất cả các phần tử dương của ma trận.
- Tổng tất cả các phần tử âm của ma trận.
- Tổng tất cả các phần tử chẵn của ma trận.
- Tổng tất cả các phần tử lẻ của ma trận.

**9.** Cho một ma trận thực kích thước m*n. Tìm:

- Số nhỏ nhất, lớn nhất (kèm chỉ số) của ma trận.
- Số nhỏ nhất, lớn nhất (kèm chỉ số) của từng hàng của ma trận.
- Số nhỏ nhất, lớn nhất (kèm chỉ số) của từng cột của ma trận.
- Số nhỏ nhất, lớn nhất (kèm chỉ số) của đường chéo chính của ma trận.
- Số nhỏ nhất, lớn nhất (kèm chỉ số) của đường chéo phụ của ma trận.

**10.** Nhập 2 ma trận vuông cấp n A và B. Tính A + B, A – B.

### III.4.2. Xâu kí tự

Hãy lập trình bằng cả hai kiểu xâu ký tự với các bài toán sau:

**1.** Hãy nhập một xâu kí tự. In ra màn hình đảo ngược của xâu đó.

**2.** Nhập xâu. Thống kê số các chữ số '0', số chữ số '1', …, số chữ số '9' trong xâu.

**3.** In ra vị trí kí tự trắng đầu tiên từ bên trái (phải) một xâu kí tự.

**4.** Nhập xâu. In ra tất các các vị trí của chữ 'a' trong xâu và tổng số lần xuất hiện của nó.

**5.** Nhập xâu. Tính số từ có trong xâu. In mỗi dòng một từ.

**6.** Nhập xâu họ tên, in ra họ, tên dưới dạng viết hoa chữ đầu, các chữ còn lại viết thường và không có dấu cách thừa trong xâu.

**7.** Thay kí tự x trong xâu s bởi kí tự y (s, x, y được đọc vào từ bàn phím)

**8.** Xoá mọi kí tự x có trong xâu s (s, x được đọc vào từ bàn phím). (Gợi ý: nên xoá ngược từ cuối xâu về đầu xâu).

**9.** Nhập xâu. Không phân biệt viết hoa hay viết thường, hãy in ra các kí tự có mặt trong xâu và số lần xuất hiện của nó (ví dụ xâu “Trach □ Van – Doanh” có chữ a xuất hiện 3 lần, c(1), d(1), h(2), n(2), o(1), r(1), t(1), □(2), space(4)).

### III.4.3. Con trỏ

1. Hãy khai báo biến kí tự ch và con trỏ kiểu kí tự pc trỏ vào biến ch. Viết ra các cách gán giá trị ‘A’ cho biến ch.
2. Cho mảng nguyên cost. Viết ra các cách gán giá trị 100 cho phần tử thứ 3 của mảng.
3. Cho p, q là các con trỏ cùng trỏ đến kí tự c. Đặt *p = *q + 1. Có thể khẳng định: *q = *p - 1 ?
4. Cho p, q là các con trỏ trỏ đến biến nguyên x = 5. Đặt *p = *q + 1; Hỏi *q ?
5. Cho p, q, r, s là các con trỏ trỏ đến biến nguyên x = 10. Đặt *q = *p + 1; *r = *q + 1; *s = *r + 1. Hỏi giá trị của biến x ?
6. Chọn câu đúng nhất trong các câu sau:
   A. Địa chỉ của một biến là số thứ tự của byte đầu tiên máy dành cho biến đó.
   B. Địa chỉ của một biến là một số nguyên.
   C. Số học địa chỉ là các phép toán làm việc trên các số nguyên biểu diễn địa chỉ của biến
   D. a và b đúng
7. Chọn câu sai trong các câu sau:
   A. Các con trỏ có thể phân biệt nhau bởi kiểu của biến mà nó trỏ đến.
   B. Hai con trỏ trỏ đến các kiểu khác nhau sẽ có kích thước khác nhau.
   C. Một con trỏ kiểu void có thể được gán bởi con trỏ có kiểu bất kỳ (cần ép kiểu).
   D. Hai con trỏ cùng trỏ đến kiểu cấu trúc có thể gán cho nhau.
8. Cho con trỏ p trỏ đến biến x kiểu float. Có thể khẳng định?
   A. p là một biến và *p cũng là một biến
   B. p là một biến và *p là một giá trị hằng
   C. Để sử dụng được p cần phải khai báo float *p; và gán *p = x;
   D. Cũng có thể khai báo void *p; và gán (float)p = &x;

9. Cho khai báo float x, y, z, *px, *py; và các lệnh px = &x; py = &y; Có thể khẳng định ?
   A. Nếu x = *px thì y = *py
   B. Nếu x = y + z thì *px = y + z
   C. Nếu *px = y + z thì *px = *py + z
   D. a, b, c đúng
10. Cho khai báo float x, y, z, *px, *py; và các lệnh px = &x; py = &y; Có thể khẳng định ?
   A. Nếu *px = x thì *py = y
   B. Nếu *px = *py - z thì *px = y - z
   C. Nếu *px = y - z thì x = y - z
   D. a, b, c đúng
11. Không dùng mảng, hãy nhập một dãy số nguyên và in ngược dãy ra màn hình.
12. Không dùng mảng, hãy nhập một dãy số nguyên và chỉ ra vị trí của số bé nhất, lớn nhất.
13. Không dùng mảng, hãy nhập một dãy số nguyên và in ra dãy đã được sắp xếp.
14. Không dùng mảng, hãy nhập một dãy kí tự. Thay mỗi kí tự ‘a’ trong dãy thành kí tự ‘b’ và in kết quả ra màn hình.

### III.4.2. Con trỏ và xâu kí tự

**1.** Giả sử p là một con trỏ kiểu kí tự trỏ đến xâu "Tin học". Chọn câu đúng nhất trong các câu sau:

A. cout << p sẽ in ra dòng "Tin học"
B. cout << p sẽ in ra dòng "Tin học"
C. cout << p sẽ in ra chữ cái 'T'
D. b và c đúng

**2.** Xét chương trình (không kể các khai báo file nguyên mẫu):

```
char st[] = "tin học";
main() {
     char *p; p = new char[10];
     for (int i=0; st[i] != '\0'; i++) p[i] = st[i];
}
```

Chương trình trên chưa hoàn chỉnh vì:

A: Sử dụng sai cú pháp toán tử new
B: Sử dụng sai cú pháp p[i] (đúng ra là *(p+i))
C: Xâu p chưa có kết thúc
D: Cả a, b, c, đều sai

**3.** Để tính độ dài xâu, có người viết đoạn chương trình sau:

```
char *st;
main()
{
  int len = 0; gets(st);
  while (st++ != '\0') len++; printf("%d",len);
}
```

Hãy chọn câu đúng nhất:

A. Chương trình trên là hoàn chỉnh
B. Cần thay len++ bởi ++len
C. Cần thay st++ bởi *st++
D. Cần thay st++ != '\0' bởi st++ == '\0'

**4.** Cho xâu kí tự (dạng con trỏ) s. Hãy in ngược xâu ra màn hình.

**5.** Cho xâu kí tự (dạng con trỏ) s. Hãy copy từ s sang xâu t một đoạn bắt đầu tại vị trí m với độ dài n.

**6.** Cho xâu kí tự (dạng con trỏ) s. Hãy thống kê tần xuất xuất hiện của các kí tự có trong s. In ra màn hình theo thứ tự giảm dần của các tần xuất (tần xuất là tỉ lệ % số lần xuất hiện của x trên tổng số kí tự trong s).

# CHƯƠNG IV. HÀM - TỔ CHỨC CHƯƠNG TRÌNH

## IV.1. Hàm

Hàm là một chương trình con trong chương trình lớn. Hàm có thể có (hoặc khôngcó) đối số và có thể trả lại (hoặc không trả lại) giá trị cho chương trình gọi nó. Một chương trình là tập các hàm, trong đó có một hàm chính với tên gọi *main()*, khi chạy chương trình, hàm *main()* sẽ được chạy đầu tiên và gọi đến hàm khác. Kết thúc hàm *main()* cũng là kết thúc chương trình.

Hàm giúp cho việc phân đoạn chương trình thành những môđun riêng rẽ, hoạt động độc lập với ngữ nghĩa của chương trình lớn. Như vậy, một hàm có thể được sử dụng trong chương trình này mà cũng có thể được sử dụng trong chương trình khác, dễ cho việc kiểm tra và bảo trì chương trình. Hàm có một số đặc trưng:

- Nằm trong hoặc ngoài văn bản có chương trình gọi đến hàm. Trong một tệp mã nguồn có thể chứa nhiều hàm,
- Được gọi từ chương trình chính (main), từ hàm khác hoặc từ chính nó (đệ quy),
- Không lồng nhau.
- Có 3 cách truyền giá trị: Truyền theo tham trị, tham biến và tham trỏ.

### IV.1.1. Khai báo và định nghĩa hàm

#### *a. Khai báo*

Một hàm thường làm chức năng: tính toán trên các tham đối và cho lại giá trị kết quả, hoặc chỉ đơn thuần thực hiện một nhiệm vụ nào đó và không trả lại kết quả tính toán. Kiểu giá trị trả lại được gọi là kiểu của hàm. Các hàm thường được khai báo ở đầu chương trình. Các hàm viết sẵn được khai báo trong các file tiêu đề. Do đó, để sử dụng được các hàm này, cần khai báo lệnh #include <tên tệp tiêu đề> ở ngay đầu chương trình. Ví dụ để sử dụng các hàm toán học ta cần khai báo file tiêu đề <math>. Đối với các hàm do NSD tự viết, cũng nên khai báo. Cú pháp khai báo một hàm như sau:

```
<kiểu giá trị trả lại> <tên hàm>(d/s kiểu đối) ;
```

trong đó, *kiểu giá trị trả lại* gọi là kiểu hàm và có thể là kiểu dữ liệu chuẩn bất kỳ của C++ và cả kiểu của NSD tự tạo. Đặc biệt nếu hàm không trả lại giá trị thì kiểu của giá trị trả lại được khai báo là *void*.

*Ví dụ 4.1:*

```
int bp(int);       // Khai báo hàm bp, có đối kiểu int và kiểu hàm là int
int rand100();     // Không đối, kiểu hàm (giá trị trả lại) là int
void alltrim(char[]); // đối là xâu kí tự, hàm không trả lại giá trị.
```

#### *b. Định nghĩa hàm*

Cấu trúc một hàm bất kỳ được bố trí giống như hàm *main()*. Cụ thể:

```
<kiểu hàm> <tên hàm>(danh sách tham đối hình thức)
{
    Khai báo cục bộ của hàm ;  // chỉ dùng riêng cho hàm này
    Dãy lệnh của hàm ;
    return (biểu thức trả về); // có thể nằm đâu đó trong dãy lệnh.
}
```

- *Danh sách tham đối hình thức* còn được gọi ngắn gọn là danh sách đối gồm dãy các đối cách nhau bởi dấu phẩy, đối có thể là một biến thường, biến tham chiếu hoặc biến con trỏ, hai loại biến sau ta sẽ trình bày trong các phần tới. Mỗi đối được khai báo giống như khai báo biến, tức là cặp gồm <*kiểu đối*> <*tên đối*>.

- Với hàm có trả lại giá trị cần có câu lệnh *return* kèm theo sau là một biểu thức. Kiểu giá trị của biểu thức phải tương thích với *kiểu của hàm* đã được khai báo. Câu lệnh *return* có thể nằm ở vị trí bất kỳ trong phần câu lệnh, tuỳ thuộc mục đích của hàm. Khi gặp câu lệnh *return* chương trình tức khắc thoát khỏi hàm và trả về giá trị của biểu thức sau *return* như giá trị của hàm.

*Ví dụ 4.2:*

Chương trình sau định nghĩa hàm *luythua()*. Hàm này có hai đối: x kiểu thực và số mũ n nguyên. Hàm trả về giá trị $x^n$ có kiểu thực.

```
double luythua(float x, int n)
{
    int i ;                          // biến chỉ số
    double kq = 1 ;                  // để lưu kết quả
    for (i = 1; i <= n; i++) kq *= x ;
    return kq;
}
```

- Nếu hàm không trả lại giá trị (tức kiểu hàm là void), thì có thể có hoặc không có câu lệnh *return*, hoặc chỉ có lệnh *return* mà không có biểu thức giá trị trả về.

*Ví dụ 4.3:*

Hàm in ra màn hình 100 số tự nhiên đầu tiên, hàm chỉ làm công việc in ra màn hình mà không có giá trị gì để trả về.

```
void inkq()
{
    int i;
    for (i = 1; i <= 100; i++) cout << i << " ";
    return ;
}
```

Hàm *main()* thường có giá trị trả về cho hệ điều hành khi chương trình chạy xong, vì vậy ta thường khai báo kiểu hàm là *int main()* và câu lệnh cuối cùng trong hàm thường là *return 0*.

### *c. Chú ý về khai báo và định nghĩa hàm*

- Danh sách đối trong khai báo hàm có thể chứa hoặc không chứa tên đối, thông thường ta chỉ khai báo kiểu đối chứ không cần khai báo tên đối, trong khi ở dòng đầu tiên của định nghĩa hàm phải có tên đối đầy đủ.

- Cuối lệnh khai báo hàm phải có dấu chấm phẩy ; , trong khi cuối dòng đầu tiên của định nghĩa hàm không có dấu chấm phẩy.

- Hàm có thể không có đối (danh sách đối rỗng), tuy nhiên cặp dấu ngoặc sau tên hàm vẫn phải được viết. Ví dụ *main(), lamtho(), vietgiaotrinh(), ...*

- Một hàm có thể không cần phải khai báo nếu nó được định nghĩa phía trước hàm gọi nó. Ví dụ có thể viết hàm *main()* trước (trong văn bản chương trình), rồi sau đó mới viết đến các hàm "con". Do trong hàm *main()* chắc chắn sẽ gọi đến hàm con này nên danh sách của chúng phải được khai báo trước hàm *main()*. Trường hợp ngược lại nếu các hàm con được định nghĩa trước

hàm *main()* thì không cần phải khai báo chúng nữa. Nguyên tắc này áp dụng cho hai hàm A, B bất kỳ chứ không riêng cho hàm *main()*, nghĩa là nếu B gọi đến A thì trước đó A phải được định nghĩa hoặc ít nhất cũng có dòng khai báo A.

### IV.1.2. Lời gọi và sử dụng hàm

Lời gọi hàm được phép xuất hiện trong bất kỳ biểu thức, câu lệnh của hàm khác … Nếu lời gọi hàm lại nằm trong chính bản thân hàm đó thì ta gọi là đệ quy. Để gọi hàm ta chỉ cần viết tên hàm và danh sách các giá trị cụ thể truyền cho các đối đặt trong cặp dấu ngoặc tròn ().

**tên hàm(danh sách tham đối thực sự) ;**

- Danh sách tham đối thực sự gồm các giá trị cụ thể để gán lần lượt cho các đối hình thức của hàm. Khi hàm được gọi thì tất cả những vị trí xuất hiện của đối hình thức sẽ được gán cho giá trị cụ thể của đối thực sự tương ứng trong danh sách.

- Danh sách tham đối thực sự truyền cho hàm phải có số lượng bằng với số lượng đối hình thức trong hàm và được truyền theo đúng thứ tự tương ứng. Các tham đối thực sự có thể là các hằng, các biến hoặc biểu thức. Ví dụ ta có hàm in *n* lần kí tự *c* với tên là *inkitu(int n, char c);* và lời gọi hàm *inkitu(12, 'A');* thì *n* và *c* là các đối hình thức, *12* và *'A'* là các đối thực sự. Các đối hình thức *n* và *c* sẽ lần lượt được gán bằng các giá trị tương ứng là *12* và *'A'* trước khi tiến hành các câu lệnh trong thân hàm. Giả sử hàm in kí tự được khai báo lại thành *inkitu(char c, int n);* thì lời gọi hàm cũng phải được đổi lại thành *inkitu('A', 12).*

- Các giá trị tương ứng được truyền cho đối phải có kiểu cùng với kiểu đối (hoặc C++ có thể tự động chuyển kiểu được về kiểu của đối nếu phù hợp).

- Khi một hàm được gọi, nơi gọi hàm sẽ tạm thời chuyển quyền điều khiển cho hàm. Sau khi hàm thực hiện xong công việc của nó, quyền điều khiển lại được trả về nơi gọi để thực hiện tiếp câu lệnh sau lệnh gọi hàm.

*Ví dụ 4.4*: Giả sử ta cần tính giá trị của biểu thức $2x^3 - 5x^2 - 4x + 1$, thay cho việc tính trực tiếp $x^3$ và $x^2$, ta có thể gọi hàm *luythua()* trong ví dụ 4.2, để tính các giá trị này bằng cách gọi nó trong hàm *main()* như sau:

```
#include <iostream>
#include <iomanip>
double luythua(float x, int n)          // trả lại giá trị x^n
{
    int i ;                             // biến chỉ số
    double kq = 1 ;                     // để lưu kết quả
    for (i=1; i<=n; i++) kết quả *= x ;
    return kq;
}

int main()                  // tính giá trị 2x3 - 5x2 - 4x + 1
{
    float x ;               // tên biến có thể trùng với đối của hàm
    double f ;              // để lưu kết quả
    cout << "x = " ; cin >> x;
    f = 2*luythua(x,3) - 5*luythua(x,2) - 4*x + 1;
    cout << setprecision(2) << f << endl ;
    return 0;
}
```

Qua ví dụ này ta thấy lập trình có hàm làm cho chương trình gọn hơn, dễ đọc hơn. Ở đây, hàm *luythua()* chỉ được viết một lần nhưng có thể sử dụng nó nhiều lần (2 lần trong ví dụ này) bằng một lệnh gọi đơn giản cho mỗi lần sử dụng, không phải viết lại nhiều lần đoạn lệnh tính luỹ thừa.

### IV.1.3. Hàm với đối mặc định

Trong phần trước chúng ta đã khẳng định số lượng tham đối thực sự phải bằng số lượng tham đối hình thức khi gọi hàm. Tuy nhiên, nhiều khi hàm được gọi với các giá trị mặc định trước một số tham đối hình thức. Từ thực tế đó C++ cung cấp một cú pháp mới để khi gọi hàm, ta không nhất thiết phải viết đầy đủ danh sách tham đối, nếu một số trong chúng đã có những giá trị định trước. Trường hợp này, được gọi là *hàm với tham đối mặc định* và có khai báo với cú pháp như sau:

<kiểu hàm> <tên hàm>(đ$_1$, …, đ$_n$, đmđ$_1$ = gt$_1$, …, đmđ$_m$ = gt$_m$) ;

- Các đối *đ$_1$, ..., đ$_n$* và đối mặc định *đmđ$_1$, ..., đmđ$_m$* đều được khai báo như bình thường nghĩa là gồm có kiểu đối và tên đối.
- Riêng các đối mặc định *đmđ$_1$, ..., đmđ$_m$* có thêm lệnh gán giá trị mặc định cho đối. Một lời gọi bất kỳ đến hàm này đều phải có đầy đủ các tham đối thực sự ứng với các đối *đ$_1$, ..., đ$_m$* nhưng có thể không có các tham đối thực sự ứng với các đối mặc định *đmđ$_1$, ..., đmđ$_m$*. Tham đối hình thức nào không có tham đối thực sự ứng thì nó sẽ tự động nhận giá trị mặc định đã khai báo.

Ví dụ, xét hàm *int luythua(float x, int n = 2);* Hàm này có một tham đối mặc định là số mũ *n*, nếu lời gọi hàm bỏ qua số mũ này thì chương trình hiểu là tính bình phương của *x* (*n* = 2). Ví dụ lời gọi *luythua(4, 3)* được hiểu là tính $4^3$ còn nếu gọi *luythua(4)* thì hiểu là tính $4^2$.

Hàm tính tổng 4 số nguyên: *int tong(int m, int n, int i = 0; int j = 0);* khi đó có thể tính tổng của 5+2+3+7 bằng lời gọi hàm *tong(5,2,3,7)* hoặc có thể chỉ tính tổng 3 số 4, 2, 1 bằng lời gọi tong(4, 2,1) hoặc cũng có thể gọi tong(6,4) chỉ để tính tổng của 2 số 6 và 4.

*Chú ý:* Các đối ngầm định phải được khai báo liên tục và và phải nằm ở cuối danh sách đối.

*Ví dụ 4.5:*

```
// sai vì các đối mặc định không liên tục
int tong(int x, int y=2, int z, int t=1)
// sai vì đối mặc định không ở cuối
void xoa(int x=0, int y)
```

### IV.1.4. Khai báo hàm chồng tên

Đây là một kỹ thuật trong lập trình hướng đối tượng, cho phép sử dụng cùng một tên gọi cho nhiều hàm có cùng mục đích, nhưng tính toán trên các kiểu dữ liệu khác nhau hoặc trên số lượng dữ liệu khác nhau. Ví dụ hàm sau tìm số lớn nhất trong 2 số nguyên:

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

Nếu đặt c = max(3, 5) ta sẽ có c = 5. Tuy nhiên, nếu gọi c = max(3.0, 5.0), chương trình sẽ bị lỗi vì các giá trị (float) không phù hợp với kiểu (int) của đối trong hàm max. Trong trường hợp như vậy chúng ta phải viết hàm mới để tính max của 2 số thực. Mục đích, cách làm việc của hàm này hoàn toàn giống hàm trước (trong C và các NNLT cổ điển khác) chúng ta buộc phải sử dụng một tên mới cho hàm "mới" này. Ví dụ:

```
float fmax(float a, float b) { return (a > b) ? a: b ; }
```

Tương tự để tuận tiện ta sẽ viết thêm các hàm:

```
char cmax(char a, char b) { return (a > b) ? a: b ; }
long lmax(long a, long b) { return (a > b) ? a: b ; }
```

```
double dmax(double a, double b) { return (a > b) ? a: b ; }
```

Tóm lại ta sẽ có 5 hàm: *max, cmax, fmax, lmax, dmax*, việc sử dụng tên như vậy sẽ gây nhầm lẫn khi cần gọi hàm. C++ cho phép ta có thể khai báo và định nghĩa cả 5 hàm trên với cùng 1 tên gọi ví dụ là *max* chẳng hạn. Khi đó ta có 5 hàm:

```
1:  int max(int a, int b) { return (a > b) ? a: b ; }
2:  float max(float a, float b) { return (a > b) ? a: b ; }
3:  char max(char a, char b) { return (a > b) ? a: b ; }
4:  long max(long a, long b) { return (a > b) ? a: b ; }
5:  double max(double a, double b) { return (a > b) ? a: b ; }
```

Và lời gọi hàm bất kỳ dạng nào như max(3,5), max(3.0,5), max('O', 'K') đều được đáp ứng. Vậy làm sao mà chương trình làm sao phân biệt được? Đó là, vì chương trình sẽ dựa vào kiểu của các đối khi gọi để quyết định chạy hàm nào. Ví dụ lời gọi max(3,5) có 2 đối đều là kiểu nguyên nên chương trình sẽ gọi hàm 1, lời gọi max(3.0,5) hướng đến hàm số 2 và tương tự chương trình sẽ chạy hàm số 3 khi gặp lời gọi max('O','K'). Như vậy, có một yêu cầu khi viết các hàm trùng tên đó là trong danh sách đối của chúng phải có ít nhất một cặp đối nào đó khác kiểu nhau hoặc số lượng đối phải khác nhau (nếu kiểu của chúng là giống nhau).

Ví dụ việc vẽ các hình: thẳng, tam giác, vuông, chữ nhật trên màn hình là giống nhau, chúng chỉ phụ thuộc vào số lượng các điểm nối và toạ độ của chúng. Do vậy ta có thể khai báo và định nghĩa 4 hàm vẽ nói trên với cùng chung tên gọi.

*Ví dụ 4.6:*

```
void ve(Diem A, Diem B) ;                       // vẽ đường thẳng AB
void ve(Diem A, Diem B, Diem C) ;               // vẽ tam giác ABC
void ve(Diem A, Diem B, Diem C, Diem D) ;       // vẽ tứ giác ABCD
```

Tóm lại nhiều hàm có thể được định nghĩa chồng (với cùng tên gọi giống nhau) nếu chúng thoả các điều kiện sau:

- Số lượng các tham đối trong hàm là khác nhau, hoặc
- Kiểu của tham đối trong hàm là khác nhau.

### IV.1.5. Biến, đối tham chiếu

Một biến có thể được gán cho một bí danh mới, và khi đó chỗ nào xuất hiện biến thì cũng tương đương như dùng bí danh và ngược lại. Một bí danh như vậy được gọi là một biến tham chiếu, ý nghĩa thực tế của nó là cho phép "tham chiếu" tới một biến khác cùng kiểu của nó, tức sử dụng biến khác nhưng bằng tên của biến tham chiếu.

Giống khai báo biến bình thường, tuy nhiên trước tên biến ta thêm dấu và (&). Có thể tạm phân biến thành 3 loại: biến thường với tên thường, biến con trỏ với dấu * trước tên và biến tham chiếu với dấu &.

```
<kiểu biến> &<tên biến tham chiếu> = <tên biến được tham chiếu>;
```

Cú pháp khai báo này cho phép ta tạo ra một biến tham chiếu mới và cho nó tham chiếu đến biến được tham chiếu (cùng kiểu và phải được khai báo từ trước). Khi đó biến tham chiếu còn được gọi là bí danh của biến được tham chiếu. Chú ý không có cú pháp khai báo chỉ tên biến tham chiếu mà không kèm theo khởi tạo.

*Ví dụ 4.7:*

```
int hung, dung ; // khai báo các biến nguyên hung, dung
```

```
int &ti = hung;   // khai báo biến tham chiếu ti, teo tham chieu đến
int &teo = dung; // hung dung. ti, teo là bí danh của hung, dung
```

Từ vị trí này trở đi việc sử dụng các tên *hung, ti* hoặc *dung, teo* là như nhau.

*Ví dụ 4.8:*

```
hung = 2 ;
ti++;                       // tương đương hung ++;
cout << hung << ti ;        // 3 3
teo = ti + hung ;           // tương đương dung = hung + hung
dung++ ;                    // tương đương teo ++
cout << dung << teo ;       // 7 7
```

Vậy sử dụng thêm biến tham chiếu để làm gì ?

Cách tổ chức bên trong của một biến tham chiếu khác với biến thường ở chỗ nội dung của nó là địa chỉ của biến mà nó đại diện (giống biến con trỏ), ví dụ câu lệnh

```
cout << teo ;               // 7
```

in ra giá trị 7 nhưng thực chất đây không phải là nội dung của biến *teo*, nội dung của *teo* là địa chỉ của *dung*, khi cần in *teo*, chương trình sẽ tham chiếu đến *dung* và in ra nội dung của *dung* (7). Các hoạt động khác trên *teo* cũng vậy (ví dụ *teo++*), thực chất là tăng một đơn vị nội dung của *dung* (chứ không phải của *teo*). Từ cách tổ chức của biến tham chiếu ta thấy chúng giống con trỏ nhưng thuận lợi hơn ở chỗ khi truy cập đến giá trị của biến được tham chiếu (*dung*) ta chỉ cần ghi tên biến tham chiếu (*teo*) chứ không cần thêm toán tử (*) ở trước như trường hợp dùng con trỏ. Điểm khác biệt này có ích khi được sử dụng để truyền đối cho các hàm với mục đích làm thay đổi nội dung của biến ngoài. Tư tưởng này được trình bày rõ ràng hơn trong mục sau.

*Chú ý:*

- Biến tham chiếu phải được khởi tạo khi khai báo.
- Tuy giống con trỏ nhưng không dùng được các phép toán con trỏ cho biến tham chiếu. Nói chung chỉ nên dùng chúng trong việc truyền đối cho hàm.

### IV.1.6. Các cách truyền tham đối

Có 3 cách truyền tham đối thực sự cho các tham đối hình thức trong lời gọi hàm. Trong đó cách ta đã dùng cho đến thời điểm hiện nay được gọi là truyền theo tham trị, tức các đối hình thức sẽ nhận các giá trị cụ thể từ lời gọi hàm và tiến hành tính toán rồi trả lại giá trị. Để dễ hiểu các cách truyền đối chúng ta sẽ xem qua cách thức chương trình thực hiện với các đối khi thực hiện hàm.

#### *a. Truyền theo tham trị*

Ta xét lại ví dụ hàm *luythua(float x, int n)* tính $x^n$. Giả sử trong chương trình chính ta có các biến a, b, f đang chứa các giá trị a = 2, b = 3, và f chưa có giá trị. Để tính $a^b$ và gán giá trị tính được cho f, ta có thể gọi f = luythua(a,b). Khi gặp lời gọi này, chương trình sẽ tổ chức như sau:

- Tạo 2 biến mới (tức 2 ô nhớ trong bộ nhớ) có tên x và n. Gán nội dung các ô nhớ này bằng các giá trị trong lời gọi, tức gán 2 (a) cho x và 3 (b) cho n.
- Tới phần khai báo (của hàm), chương trình tạo thêm các ô nhớ mang tên kq và i.
- Tiến hành tính toán (gán lại kết quả cho kq).

- Cuối cùng lấy kết quả trong kq gán cho ô nhớ f (là ô nhớ có sẵn đã được khai báo trước, nằm bên ngoài hàm).
- Kết thúc hàm quay về chương trình gọi. Do hàm luythua đã hoàn thành xong việc tính toán nên các ô nhớ được tạo ra trong khi thực hiện hàm (x, n, kq, i) sẽ được xoá khỏi bộ nhớ. Kết quả tính toán được lưu giữ trong ô nhớ f (không bị xoá vì không liên quan gì đến hàm).

Trên đây là truyền đối theo cách thông thường. Vấn đề đặt ra là giả sử ngoài việc tính f, ta còn muốn thay đối các giá trị của các ô nhớ a, b (khi truyền nó cho hàm) thì có thể thực hiện được không ? Để giải quyết bài toán này ta cần theo một kỹ thuật khác, nhờ vào vai trò của biến con trỏ và tham chiếu.

***b. Truyền theo tham trỏ***

Xét ví dụ tráo đổi giá trị của 2 biến. Đây là một yêu cầu nhỏ nhưng được gặp nhiều lần trong chương trình, như sắp xếp một danh sách. Do vậy cần viết một hàm để thực hiện yêu cầu trên. Hàm không trả kết quả. Do các biến cần trao đổi là chưa được biết trước tại thời điểm viết hàm, nên ta phải đưa chúng vào hàm như các tham đối, tức hàm có hai tham đối x, y đại diện cho các biến sẽ thay đổi giá trị sau này.

*Ví dụ 4.9:*

Giả sử trong hàm *main()* ta có 2 biến *x, y* chứa các giá trị lần lượt là 2, 5. Ta cần đổi nội dung 2 biến này sao cho *x* = 5 còn *y* = 2 bằng cách gọi đến hàm *swap(x, y)*.

```
void swap(int *a, int *b) // hàm tráo đổi giá trị x, y
{
    int t ;  t = *a ; *a = *b ; *b = t ;
}
main()
{
    int x = 2; int y = 5;
    swap(&x, &y) ;
    cout << x <<" "<< y ;    // 5 2
}
```

Thay vì truyền giá trị của các biến ngoài cho đối, ta truyền địa chỉ của nó cho đối, và các thay đổi sẽ phải thực hiện trên nội dung của các địa chỉ này. Đó chính là lý do ta phải sử dụng con trỏ để làm tham đối thay cho biến thường. Từ đó lời gọi hàm sẽ là *swap(&x, &y)* (tức truyền địa chỉ của x cho p, p trỏ tới x và tương tự q trỏ tới y).

Như vậy có thể tóm tắt 3 đặc trưng để viết một hàm làm thay đổi giá trị biến ngoài như sau:

- Đối của hàm phải là con trỏ (ví dụ int *p)
- Các thao tác liên quan đến đối này (trong thân hàm) phải thực hiện tại nơi nó trỏ đến (ví dụ *p = …)
- Lời gọi hàm phải chuyển địa chỉ cho p (ví dụ &x).

*Ví dụ 4.10:* Hàm giải phương trình bậc 2: không thể trả giá trị của nghiệm qua hàm vì hàm chỉ trả lại 1 giá trị trong khi ta cần đến 2 nghiệm. Do vậy, ta khai báo 2 biến "ngoài" trong chương trình để chứa các nghiệm, và hàm phải làm thay đổi 2 biến chứa giá trị nghiệm. Hàm được viết với 5 đối, trong đó 3 đối a, b, c đại diện cho các hệ số, không thay đổi và 2 biến x1, x2 chứa nghiệm được khai báo dạng con trỏ. Ngoài ra, phương trình có thể vô nghiệm, 1 nghiệm hoặc 2 nghiệm do

vậy hàm sẽ trả lại giá trị là số nghiệm của phương trình, trong trường hợp 1 nghiệm (nghiệm kép), giá trị nghiệm sẽ được trả vào x1.

*Ví dụ 4.11:* Dưới đây là một dạng đơn giản của hàm giải phương trình bậc 2.

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

int gptb2(float a, float b, float c, float *x1, float *x2)
{
    float d ;
    d = (b*b) - 4*a*c ;
    if (d < 0) return 0 ;
    else if (d == 0) { *x1 = -b/(2*a) ; return 1 ; }
    else {
        *x1 = (-b + sqrt(d))/(2*a) ;
        *x2 = (-b - sqrt(d))/(2*a) ;
        return 2 ;
    }
}

int main()
{
    float a, b, c ;
    float x1, x2 ;
    cout << "Nhap cac he so a, b, c: " ; cin >> a >> b >> c;
    switch (gptb2(a, b, c, &x1, &x2)) {
         case 0: cout << "PT VN" ; break;
         case 1: cout << "PT co nghiem kep x = " << x1 ; break ;
         case 2: cout << "PT co 2 nghiem phan biet:" << endl ;
         cout << "x1 = " << x1 << " ; x2 = " << x2 << endl ; break;
    }
    return 0;
}
```

Trên đây chúng ta đã trình bày cách xây dựng các hàm cho phép thay đổi giá trị của biến ngoài. Một đặc trưng dễ nhận thấy là cách viết hàm tương đối phức tạp. C++ phát triển một cách viết khác dựa trên đối tham chiếu và việc truyền đối cho hàm được gọi là truyền theo tham chiếu.

### *c. Truyền theo tham chiếu*

Một hàm viết dưới dạng đối tham chiếu sẽ đơn giản hơn rất nhiều so với đối con trỏ và giống với cách viết bình thường (truyền theo tham trị), trong đó chỉ có một khác biệt đó là các đối khai báo dưới dạng tham chiếu.

Để so sánh, ta nhắc lại đặc điểm khi viết hàm theo đối con trỏ, đó là:

- Đối của hàm phải là con trỏ (ví dụ int *p)
- Các thao tác liên quan đến đối này trong thân hàm phải thực hiện tại nơi nó trỏ đến (ví dụ *p = …)
- Lời gọi hàm phải chuyển địa chỉ cho đối (ví dụ &x).

Với cách viết hàm theo đối tham chiếu:

- Đối của hàm phải là tham chiếu (ví dụ int &p)

- Các thao tác liên quan đến đối này phải thực hiện tại nơi nó trỏ đến. Vì đối là biến tham chiếu nên thao tác trên nó tác động đến nơi nó tham chiếu (thay vì *p chỉ cần viết p)
- Lời gọi hàm chỉ cần ghi tên biến, ví dụ x (thay vì &x như đối với dẫn trỏ).

Tóm lại, đối với hàm viết theo tham chiếu chỉ thay đổi ở đối (là các tham chiếu) còn lại mọi nơi khác đều viết đơn giản như cách viết truyền theo tham trị.

*Ví dụ 4.12*: Đổi giá trị 2 biến

```
void swap(int &x, int &y)
{
    int t = x; x = y; y = t;
}
main(){
    int a = 5, b = 3;
    swap(a, b);
    cout << a << b;
}
```

Bảng dưới đây minh hoạ tóm tắt 3 cách viết hàm thông qua ví dụ đổi biến ở trên.

| | **Tham trị** | **Tham chiếu** | **Dẫn trỏ** |
|---|---|---|---|
| Khai báo đối | void swap(int x, int y) | void swap(int &x, int &y) | void swap(int *x, int *y) |
| Câu lệnh | t = x; x = y; y = t; | t = x; x = y; y = t; | t = *x; *x = *y; *y = t; |
| Lời gọi | swap(a, b); | swap(a, b); | swap(&a, &b); |
| Tác dụng | a, b không thay đổi | a, b có thay đổi | a, b có thay đổi |

### IV.1.7. Hàm và mảng dữ liệu

#### *a. Truyền mảng 1 chiều cho hàm*

Thông thường chúng ta hay xây dựng các hàm làm việc trên mảng như vectơ hay ma trận gồm các phần tử. Khi đó tham đối thực sự của hàm sẽ là các mảng dữ liệu này. Trong trường hợp này ta có 2 cách khai báo đối. Cách thứ nhất đối được khai báo bình thường như khai báo biến mảng nhưng không cần có số phần tử kèm theo, ví dụ:

```
int x[];
float x[];
```

Cách thứ hai khai báo đối như một con trỏ kiểu phần tử mảng, ví dụ:

```
int *p;
float *p
```

Trong lời gọi hàm tên mảng a sẽ được viết vào danh sách tham đối thực sự, vì *a* là địa chỉ của phần tử đầu tiên của mảng *a*, nên khi hàm được gọi địa chỉ này sẽ gán cho con trỏ *p*. Vì vậy giá trị của phần tử thứ *i* của *a* có thể được truy cập bởi *x[i]* (theo khai báo 1) hoặc **(p+i)* (theo khai báo 2) và nó cũng có thể được thay đổi thực sự (do đây cũng là cách truyền theo tham trỏ).

Sau đây là chương trình nhập và in vectơ, minh hoạ cho cả 2 kiểu khai báo đối.

*Ví dụ 4.13:* Hàm nhập và in giá trị 1 vectơ

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

void nhap(int x[], int n)    // n: số phần tử
{
    int i;
```

```
    for (i=0; i<n; i++) cin >> x[i];    // hoặc cin >> *(x+i)
}
void in(int *p, int n)
{
    int i;
    for (i=0; i<n; i++) cout << *(p+i)<<" ";
}
int main()
{
    int a[10] ;               // mảng a chứa tối đa 10 phần tử
    nhap(a,7);                // vào 7 phần tử đầu tiên cho a
    in(a,3);                  // ra 3 phần tử đầu tiên của a
    return 0;
}
```

***b. Truyền mảng 2 chiều cho hàm***

Đối với mảng 2 chiều, khai báo đối và gọi hàm, phức tạp hơn nhiều so với mảng 1 chiều. Ta có hai cách khai báo đối như sau:

- Khai báo theo đúng bản chất của mảng 2 chiều float *x[m][n]* do C++ qui định, tức x là mảng 1 chiều m phần tử, mỗi phần tử của nó là một mảng một chiều float[n]. Từ đó, đối được khai báo như một mảng hình thức 1 chiều (không cần số phần tử - ở đây là số dòng) của kiểu float[n]. Tức có thể khai báo như sau:

```
float x[][n] ; // mảng với số phần tử không định trước, mỗi phần tử là n số
float (*x)[n] ;// một con trỏ, có kiểu là mảng n số (float[n])
```

Để truy nhập đến đến phần tử thứ (*i, j)* ta vẫn sử dụng cú pháp *x[i][j]*. Tên của mảng a được viết bình thường trong lời gọi hàm. Nói chung theo cách khai báo này việc truy nhập là đơn giản nhưng phương pháp cũng có hạn chế đó là số cột của mảng truyền cho hàm phải cố định bằng n.

- Xem mảng *float x[m][n]* thực sự là mảng một chiều *float x[m*n]* và sử dụng cách khai báo như trong mảng một chiều, đó là sử dụng con trỏ *float *p* để truy cập được đến từng phần tử của mảng. Cách này có hạn chế trong lời gọi: địa chỉ truyền cho hàm không phải là mảng *a* mà cần phải ép kiểu về (float*) (để phù hợp với *p*). Với cách này, gọi *k* là thứ tự của phần tử *a[i][j]* trong mảng một chiều (*m*n*), ta có quan hệ giữa *k, i, j* như sau:

```
k = *(p + i*n + j)
i = k/n
j = k%n
```

trong đó *n* là số cột của mảng truyền cho hàm. Nghĩa là, để truy cập đến *a[i][j]* ta có thể viết **(p+i*n+j)*, ngược lại biết chỉ số *k* có thể tính được dòng *i*, cột *j* của phần tử này. Ưu điểm của cách khai báo này là ta có thể truyền mảng với kích thước bất kỳ (số cột không cần định trước) cho hàm.

Sau đây là các ví dụ minh hoạ cho 2 cách khai báo trên.

*Ví dụ 4.14*: Tính tổng các số hạng trong ma trận

```
#include <iostream>
#include <cmath>
using namespace std;
float tong(float x[][10], int m, int n){
    float t = 0;
    for (int i=0; i<m; i++)
        for (int j=0; j<n; j++) t += x[i][j] ;
    return t;
}
```

```
void nhap(float x[][10],int &m, int &n){
    cout << "nhap so dong, so cot ma tran: " ; cin >> m >> n;
    for (int i=0; i<m; i++)
        for (int j=0; j<n; j++){
             cout << "[" << i << "," << j << "] = " ;
             cin >> x[i][j] ;
        }
}

int main()
{
    float a[10][10], b[10][10] ;
    int ma, na, mb, nb;
    nhap(a,ma,na);
    nhap(b,mb,nb);
    cout << tong(a, ma, na);       // in tổng các số trong ma trận
    return 0;
}
```

*Ví dụ 4.15*: Tìm phần tử bé nhất của ma trận

```
#include <iostream>
using namespace std;
float min(float *x, int m, int n) {
    float t = *x;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            if(t > *(x + i * m + j)) t = *(x + i * m + j);
    return t;
}
void nhap(float *x, int &m, int &n){
    cout << "nhap so dong, so cot ma tran: " ;
    cin >> m >> n;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++){
            cout << "[" << i << "," << j << "] = " ;
            cin >> *(x + i * m + j);
        }
}
int main()
{
    float  a[10][10];
    int m, n;
    nhap((float*)a,m,n);
    cout << min((float*)a, m, n);
    return 0;
}
```

Xu hướng chung là chúng ta xem mảng (1 hoặc 2 chiều) như là một dãy liên tiếp các số trong bộ nhớ, tức một ma trận là một đối con trỏ trỏ đến thành phần của mảng. Đối với mảng 2 chiều m*n khi truyền đối địa chỉ của ma trận cần phải ép kiểu về kiểu con trỏ. Ngoài ra bước chạy k của con trỏ (từ 0 đến m*n-1) tương ứng với các toạ độ của phần tử a[i][j] trong mảng như sau:

- `k = *(p + i*n + j)`
- `i = k/n`
- `j = k%n`

từ đó, chúng ta có thể viết các hàm mà không cần phải băn khoăn gì về kích thước của ma trận sẽ truyền cho hàm.

### *c. Giá trị trả lại của hàm là một mảng*

Không có cách nào để giá trị trả lại của một hàm là mảng. Tuy nhiên thực sự mỗi mảng cũng chính là một con trỏ, vì vậy việc hàm trả lại một con trỏ trỏ đến dãy dữ liệu kết quả là tương đương với việc trả lại mảng. Ngoài ra còn một cách dễ dùng hơn đối với mảng 2 chiều là mảng kết quả được trả lại vào trong tham đối của hàm. Ở đây chúng ta sẽ lần lượt xét 2 cách làm việc này.

Giá trị trả lại là con trỏ trỏ đến mảng kết quả. Trước hết chúng ta xét ví dụ nhỏ sau đây.

*Ví dụ 4.16:*

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

int* tragiatri1() // giá trị trả lại là con trỏ trỏ đến dãy số nguyên
{
    int kq[3] = { 1, 2, 3 }; // tạo mảng kết quả với 3 giá trị 1, 2, 3
    return kq ;                // trả lại địa chỉ cho con trỏ kết quả hàm
}

int* tragiatri2()    // giá trị trả lại là con trỏ trỏ đến dãy số nguyên
{
    int *kq = new int[3];          // cấp phát 3 ô nhớ nguyên
    // tạo mảng kết quả với 3 giá trị 1, 2, 3
    *kq =1; *(kq+1) =2; *(kq+2) = 3 ;
    return kq ;           // trả lại địa chỉ cho con trỏ kết quả hàm
}

int main()
{
    int *a, i;
    a = tragiatri1();
    for (i=0; i < 3; i++) cout<<*(a+i)<<" "; //3 giá trị không xác định
    cout<<endl;
    a = tragiatri2();
    for (i=0; i<3; i++) cout<<*(a+i)<<" " ;  // 1, 2, 3
    return 0;
}
```

Qua ví dụ trên ta thấy hai hàm trả giá trị đều tạo bên trong nó một mảng 3 số nguyên và trả lại địa chỉ mảng này cho con trỏ kết quả hàm. Tuy nhiên, chỉ có *tragiatri2()* là cho lại kết quả đúng. Tại sao ? Xét mảng *kq* được khai báo và khởi tạo trong *tragiatri1()*, đây là một mảng cục bộ (được tạo bên trong hàm) như sau này chúng ta sẽ thấy, các loại biến "tạm thời" này (và cả các tham đối) chỉ tồn tại trong quá trình hàm hoạt động. Khi hàm kết thúc các biến này sẽ mất đi. Do vậy tuy hàm đã trả lại địa chỉ của *kq* trước khi nó kết thúc, thế nhưng sau khi hàm thực hiện xong, toàn bộ *kq* sẽ được xoá khỏi bộ nhớ và vì vậy con trỏ kết quả hàm đã trỏ đến vùng nhớ không còn các giá trị như *kq* đã có. Từ điều này việc sử dụng hàm trả lại con trỏ là phải hết sức cẩn thận. Muốn trả lại con trỏ cho hàm thì con trỏ này phải trỏ đến dãy dữ liệu nào sao cho nó không mất đi sau khi hàm kết thúc, hay nói khác hơn đó phải là những dãy dữ liệu được khởi tạo bên ngoài hàm hoặc có thể sử dụng theo phương pháp trong hàm *tragiatri2()*. Trong *tragiatri2()* một mảng kết quả 3 số cũng được tạo ra nhưng bằng cách xin cấp phát vùng nhớ. Vùng nhớ được cấp phát này sẽ vẫn còn tồn tại sau khi hàm kết thúc (nó chỉ bị xoá đi khi sử dụng toán tử *delete*). Do vậy hoạt động của *tragiatri2()* là chính xác.

Tóm lại, ví dụ trên cho thấy nếu muốn trả lại giá trị con trỏ thì vùng dữ liệu mà nó trỏ đến phải được cấp phát một cách tường minh (bằng toán tử *new*), chứ không để chương trình tự động cấp phát và tự động thu hồi.

*Ví dụ 4.17*: minh họa hàm cộng 2 vectơ và trả lại vectơ kết quả (thực chất là con trỏ trỏ đến vùng nhớ đặt kết quả)

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

int* congvt(int x[10], int y[10], int n)
{
    int* z = new int[n]; // xin cấp phát bộ nhớ
    for (int i=0; i<n; i++) z[i] = x[i] + y[i];
    return z;
}
void nhap(int x[10],int n){
    cout<<"nhap "<<n<<" so nguyen\n";
    for (int i=0 ; i<n; i++) cin>>x[i];
}
main()
{
    int i, n, a[10], b[10], *c ;
    cout << "n = " ; cin >> n;          // nhập số phần tử
    nhap(a,n) ; // nhập vectơ a
    nhap(b,n) ; // nhập vectơ b
    c = congvt(a, b, n);
    for (i=0; i<n; i++) cout << c[i]<<" " ;  // in kết quả
    // hoặc viết thế này: for (i=0; i<n; i++) cout << *(c+i)<<" " ;
}
```

Chú ý: *a[i], b[i], c[i]* còn được viết dưới dạng tương đương **(a+i), *(b+i), *(c+i).*

Trong cách này, mảng cần trả lại được khai báo như một tham đối trong danh sách đối của hàm. Tham đối này là một con trỏ nên hiển nhiên khi truyền mảng đã khai báo sẵn (để chứa kết quả) từ ngoài vào cho hàm thì mảng sẽ thực sự nhận được nội dung kết quả (tức có thay đổi trước và sau khi gọi hàm - xem mục truyền tham đối thực sự theo dẫn trỏ). Ở đây ta xét lại ví dụ cộng 2 vectơ trong ví dụ trước.

*Ví dụ 4.18*: Cộng 2 vectơ, vectơ kết quả trả lại trong tham đối của hàm. So với ví dụ trước giá trị trả lại là void (không trả lại giá trị) còn danh sách đối có thêm con trỏ z để chứa kết quả.

```
void congvt(int *x, int *y, int *z, int n)   // z lưu kết quả
{
    for (int i=0; i<n; i++) z[i] = x[i] + y[i];
}

main()
{
    int i, n, a[10], b[10], c[10] ;
    cout << "n = " ; cin >> n;                // nhập số phần tử
    for (i=0; i<n; i++) cin >> a[i] ;        // nhập vectơ a
    for (i=0; i<n; i++) cin >> b[i] ;        // nhập vectơ b
    congvt(a, b, c, n);
    for (i=0; i<n; i++) cout << c[i] ;       // in kết quả
}
```

***d. Đối và giá trị trả lại là xâu kí tự***

Giống các trường hợp đã xét với mảng 1 chiều, đối của các hàm xâu kí tự có thể khai báo dưới 2 dạng: mảng kí tự hoặc con trỏ kí tự. Giá trị trả lại luôn luôn là con trỏ kí tự. Ngoài ra hàm cũng có thể trả lại giá trị vào trong các đối con trỏ trong danh sách đối.

*Ví dụ 4.19:* tách họ, tên của một xâu họ và tên. Ví dụ gồm 3 hàm. Hàm họ trả lại xâu họ (con trỏ kí tự) với đối là xâu họ và tên được khai báo dạng mảng. Hàm tên trả lại xâu tên (con trỏ kí tự) với đối là xâu họ và tên được khai báo dạng con trỏ kí tự. Thực chất đối họ và tên trong hai hàm họ, tên có thể được khai báo theo cùng cách thức, ở đây chương trình muốn minh hoạ các cách khai báo đối khác nhau (đã đề cập đến trong phần đối mảng 1 chiều). Hàm thứ ba cũng trả lại họ, tên nhưng cho vào trong danh sách tham đối, do vậy hàm không trả lại giá trị (void). Để đơn giản ta qui ước xâu họ và tên không chứa các dấu cách đầu và cuối xâu, trong đó họ là dãy kí tự từ đầu cho đến khi gặp dấu cách đầu tiên và tên là dãy kí tự từ sau dấu cách cuối cùng đến kí tự cuối xâu.

```
#include <iostream>
#include <cstring>
using namespace std;
char* ho(char hoten[])                    // hàm trả lại họ
{
    char* kq = new char[10];              // cấp bộ nhớ để chứa họ
    int i=0;
    while (hoten[i] != '\40') i++; // i dừng tại dấu cách đầu tiên
    strncpy(kq, hoten,i) ;          // copy i kí tự của hoten vào kq
    kq[i]='\0';                     // thêm kí tự kết thúc chuỗi
    return kq;
}
char* ten(char* hoten)              // hàm trả lại tên
{
    char* kq = new char[10];        // cấp bộ nhớ để chứa tên
    int i=strlen(hoten);
    while (hoten[i] != '\40') i--; // i dừng tại dấu cách cuối cùng
    strncpy(kq, hoten+i+1, strlen(hoten)-i-1) ;    // copy tên vào kq
    kq[strlen(hoten)-i-1]='\0';    // thêm kí tự kết chuỗi
    return kq;
}
void tachht(char* hoten, char* ho, char* ten)
{
int i=0;
    while (hoten[i] != '\40') i++; // i dừng tại dấu cách đầu tiên
    strncpy(ho, hoten, i) ; // copy i kí tự của hoten vào ho
    i=strlen(hoten);
    while (hoten[i] != '\40') i--; // i dừng tại dấu cách cuối cùng
    strncpy(ten, hoten+i+1, strlen(hoten)-i-1) ; // copy tên vào ten
}

int main()
{
    char ht[30], *h, *t ;           // các biến họ tên, họ, tên
    cout << "Ho va ten = " ; cin.getline(ht,30) ; // nhập họ tên
    h = ho(ht); t = ten(ht);
    cout << "Ho = " << h <<", ten = " << t << endl;
    tachht(ht, h, t);
    cout << "Ho = " << h << ", ten = " << t << endl;
    return 0;
}
```

***e. Đối là hằng con trỏ***

Như đã nói trong phần truyền đối cho hàm, ta đã biết để thay đổi biến ngoài, đối tương ứng phải được khai báo dưới dạng con trỏ. Tuy nhiên, trong nhiều trường hợp các biến ngoài không có nhu cầu thay đổi nhưng đối tương ứng với nó vẫn phải khai báo dưới dạng con trỏ (ví dụ đối là mảng hoặc xâu kí tự). Điều này có khả năng do nhầm lẫn, các biến ngoài này sẽ bị thay đổi ngoài ý muốn. Trong trường hợp như vậy để cẩn thận, các đối con trỏ nếu không muốn thay đổi (chỉ lấy giá trị) cần được khai báo như là một hằng con trỏ bằng cách thêm trước khai báo kiểu của chúng từ khoá const. Từ khoá này khẳng định biến tuy là con trỏ nhưng nó là một hằng không thay đổi được giá trị. Nếu trong thân hàm ta cố tình thay đổi chúng thì chương trình sẽ báo lỗi. Ví dụ đối hoten trong cả 3 hàm ở trên có thể được khai báo dạng const char* hoten.

*Ví dụ 4.20:* Đối là hằng con trỏ. In hoa một xâu kí tự

```
#include <iostream>
#include <cstring>
using namespace std;
void inhoa(const char* s){
    char *t;
    strcpy(t, s);
    cout << s << strupr(t);  // không dùng được strupr(s)
}
int main(){
    char *s = "abcde" ;
    inhoa(s);                // abcdeABCDE
    return 0;
}
```

## IV.2. Đệ quy

### IV.2.1. Khái niệm đệ qui

Một hàm gọi đến hàm khác là bình thường, nhưng nếu hàm lại gọi đến chính nó thì ta gọi hàm là đệ qui. Khi thực hiện một hàm đệ qui, hàm sẽ phải chạy rất nhiều lần, trong mỗi lần chạy chương trình sẽ tạo nên một tập biến cục bộ mới trên khối nhớ stack (các đối, các biến riêng khai báo trong hàm) độc lập với lần chạy trước đó, việc này rất dễ gây tràn stack (stack là khối bộ nhớ lưu giữa các biến được khai báo trong hàm). Vì vậy đối với những bài toán có thể giải được bằng phương pháp lặp thì không nên dùng đệ qui.

*Ví dụ 4.21:* Xét hàm tính n giai thừa. Để tính n! ta có thể dùng phương pháp lặp như sau:

```
main()
{
    int n; doule kq = 1;
    cout << "n = " ; cin >> n;
    for (int i=1; i<=n; i++) kq *= i;
    cout << n << "! = " << kq;
}
```

Mặt khác, n! giai thừa cũng được tính thông qua (n-1)! bởi công thức truy hồi

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

do đó ta có thể xây dựng hàm đệ qui tính n! như sau:

```
double gt(int n)
{
```

```
    if (n==0) return 1;
    else return gt(n-1)*n;
}

main()
{
    int n;
    cout << "n = " ; cin >> n;
    cout << gt(n);
}
```

Trong hàm main() giả sử ta nhập 3 cho n, khi đó để thực hiện câu lệnh cout << gt(3) để in 3! đầu tiên chương trình sẽ gọi chạy hàm gt(3). Do 3 ≠ 0 nên hàm gt(3) sẽ trả lại giá trị gt(2)*3, tức lại gọi hàm gt với tham đối thực sự ở bước này là n = 2. Tương tự gt(2) = gt(1)*2 và gt(1) = gt(0)*1. Khi thực hiện gt(0) ta có đối n = 0 nên hàm trả lại giá trị 1, từ đó gt(1) = 1*1 = 1 và suy ngược trở lại ta có gt(2) = gt(1)*2 = 1*2 = 2, gt(3) = gt(2)*3 = 2*3 = 6, chương trình in ra kết quả 6.

Từ ví dụ trên ta thấy hàm đệ qui có đặc điểm:

- Chương trình viết rất gọn,
- Việc thực hiện gọi đi gọi lại hàm nhiều lần, số lần phụ thuộc vào độ lớn của đầu vào. Chẳng hạn trong ví dụ trên hàm được gọi n lần, mỗi lần như vậy chương trình sẽ mất thời gian để lưu giữ các thông tin của hàm gọi trước khi chuyển điều khiển đến thực hiện hàm được gọi. Mặt khác các thông tin này được lưu trữ nhiều lần trong stack sẽ dẫn đến tràn stack nếu n lớn.
- Tuy nhiên, đệ qui là cách viết rất gọn, dễ viết và đọc chương trình, mặt khác có nhiều bài toán hầu như tìm một thuật toán lặp cho nó là rất khó trong khi viết theo thuật toán đệ qui thì lại rất dễ dàng.

### IV.2.2. Lớp các bài toán giải được bằng đệ qui

Phương pháp đệ qui thường được dùng để giải các bài toán có đặc điểm:

- Giải quyết được dễ dàng trong các trường hợp riêng gọi là trường hợp suy biến hay cơ sở, trong trường hợp này hàm được tính bình thường mà không cần gọi lại chính nó,
- Đối với trường hợp tổng quát, bài toán có thể giải được bằng bài toán cùng dạng nhưng với tham đối khác có kích thước nhỏ hơn tham đối ban đầu. Và sau một số bước hữu hạn biến đổi cùng dạng, bài toán đưa được về trường hợp suy biến.

Như vậy trong trường hợp tính n! nếu n = 0 hàm cho ngay giá trị 1 mà không cần phải gọi lại chính nó, đây chính là trường hợp suy biến. Trường hợp n > 0 hàm sẽ gọi lại chính nó nhưng với n giảm 1 đơn vị. Việc gọi này được lặp lại cho đến khi n = 0.

Một lớp rất rộng của bài toán dạng này là các bài toán có thể định nghĩa được dưới dạng đệ qui như các bài toán lặp với số bước hữu hạn biết trước, các bài toán UCLN, tháp Hà Nội, ...

### IV.2.3. Cấu trúc chung của hàm đệ qui

Dạng thức chung của một chương trình đệ qui thường như sau:

```
if (trường hợp suy biến)
{
    trình bày cách giải      // giả định đã có cách giải
}
```

```
else                        // trường hợp tổng quát
{
    gọi lại hàm với tham đối "bé" hơn
}
```

### IV.2.4. Các ví dụ

*Ví dụ 4.22:* Tìm UCLN của 2 số a, b.

Dựa vào thuật toán Ơclít, bài toán có thể được định nghĩa dưới dạng đệ qui như sau:

- nếu a = b thì UCLN = a
- nếu a > b thì UCLN(a, b) = UCLN(a-b, b)
- nếu a < b thì UCLN(a, b) = UCLN(a, b-a)

Từ đó ta có chương trình đệ qui để tính UCLN của a và b như sau.

```
int UCLN(int a, int b)       // qui uoc a, b > 0
{
    if (a < b) UCLN(a, b-a);
    if (a == b) return a;
    if (a > b) UCLN(a-b, b);
}
```

*Ví dụ 4.23*: Tính số hạng thứ n của dãy Fibonaci là dãy f(n) được định nghĩa:

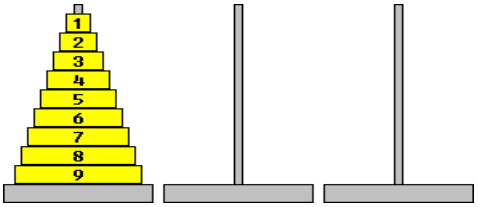
- f(0) = f(1) = 1
- f(n) = f(n-1) + f(n-2) với ∀n ≥ 2.

```
long fib(int n)
{
    long kq;
    if (n==0 || n==1) kq = 1; else kq = fib(n-1) + fib(n-2);
    return kq;
}
```

*Ví dụ 4.24*: Chuyển tháp là bài toán cổ nổi tiếng, nội dung như sau: Cho n chiếc đĩa xếp tại vị trí cọc 1. Yêu cầu bài toán là hãy chuyển toàn bộ tháp sang cọc 2 (cho phép sử dụng cọc trung gian 3) theo các điều kiện sau đây:

- mỗi lần chỉ được chuyển một đĩa trên cùng của chồng đĩa,
- tại bất kỳ thời điểm tại cả 3 vị trí các tầng tháp lớn hơn phải nằm dưới các tầng tháp nhỏ hơn.

Bài toán chuyển đĩa được minh hoạ bởi hình vẽ dưới đây.




Bài toán có thể được đặt ra tổng quát hơn như sau: Gọi tên các cọc lần lượt từ trái sang phải là a, b, c, ta có thể xây dựng một cách chuyển đệ qui như sau:

- chuyển n-1 đĩa từ cọc a sang cọc c;

- chuyển 1 đĩa còn lại từ a sang b,
- chuyển trả n-1 đĩa từ cọc c sang vị trí cọc b

hiển nhiên nếu số đĩa là 1 thì ta chỉ phải thực hiện một phép chuyển đến cọc b.

Mỗi lần chuyển 1 đĩa từ vị trí i đến j ta kí hiệu i $\rightarrow$ j. Chương trình sẽ nhập vào input là số tầng và in ra các bước chuyển theo kí hiệu trên.

Từ đó ta có thể xây dựng hàm đệ qui sau đây ;

```
#include <iostream>
using namespace std;
void chuyen(int n, int a, int b, int c) {
    if (n==1)
        cout <<"chuyen 1 dia tu "<<a<<" sang " << b << endl;
    else {
        chuyen(n-1, a, c, b) ;
        cout <<"chuyen 1 dia tu "<<a<<" sang " << b << endl;
        chuyen(n-1, c, b, a) ;
    }
}
main()
{
    int sotang ;
    cout << "So dia = " ; cin >> sotang;
    chuyen(sotang, 1, 2, 3);
}
```

Ví dụ nếu số tầng bằng 3 thì chương trình in ra kết quả là dãy các phép chuyển sau đây:

$$a \rightarrow b\ ,\ a \rightarrow c\ ,\ b \rightarrow c\ ,\ a \rightarrow b\ ,\ c \rightarrow a\ ,\ c \rightarrow b\ ,\ a \rightarrow b.$$

có thể tính được số lần chuyển là $2^n$ - 1 với n là số tầng.

## IV.3. Tổ chức chương trình

### IV.3.1. Các loại biến và phạm vi

#### *a. Biến cục bộ*

Biến cục bộ là các biến được khai báo trong thân của hàm nào đó và chỉ có tác dụng trong phàm vi hàm này, kể cả các biến khai báo trong hàm *main()* cũng chỉ có tác dụng riêng trong hàm *main()*. Vì vậy, được phép sử dụng tên biến trùng nhau giữa các hàm. Các biến của mỗi hàm chỉ tồn tại trong bộ nhớ vào lúc hàm đó hoạt động. Khi bắt đầu thực hiện hàm, các biến này được tự động tạo ra. Khi hàm kết thúc, các biến này bị xóa khỏi bộ nhớ.

Tham đối của các hàm cũng được xem như biến cục bộ.

Trong C++, biến cục bộ còn có thể được khai báo ngay trong khối lệnh, các biến này chỉ có hiệu lực trong khối lệnh đó và cũng bị xóa khi khối lệnh kết thúc.

*Ví dụ 4.25*: một chương trình nhỏ có hai hàm: *luythua()* và *main()*. Mục đích để minh hoạ biến cục bộ trong hàm và trong khối lệnh.

```
#include <iostream>
using namespace std;
float luythua(float x, int n)// hàm trả giá trị xn
{
    float kq = 1;
```

```
    for (int i=1; i<=n; i++) kq *= x;
    return kq;
}
int main()
{
    float x; int n;
    cout << "Nhap x va n: "; cin >> x >> n;
    cout << luythua(x, n);
    return 0;
}
```

Trong ví dụ trên, biến *kq* là biến cục bộ của hàm *luythua()*, biến *i* là biến cục bộ chỉ trong khối lệnh *for()* của hàm *luythua()*, biến *x* và *n* là đối của hàm *luythua()*. Các biến cục bộ của hàm *main()* cũng được khai báo là *x, n*. Tất cả khai báo trên đều hợp lệ và đều được xem như khác nhau.

*Ví dụ 4.26*: một chương trình khác minh hoạ biến cục bộ trong hàm và khối lệnh của hàm.

```
#include <iostream>
using namespace std;
int main()
{
    int max = 0; int j = 1000;
    for (int j = 1; j<=100; j++) cout<< j <<"" ;
    cout <<"\n"<<j;
    return 0;
}
```

Trong ví dụ trên, hàm *main()* có biến cục bộ *j*, khối lệnh *for* cũng có biến cục bộ *j* của riêng nó. Hai biến *j* này hoạt động độc lập. Ở đầu hàm *main(), j* của *main()* được gán giá trị là 10000. Khi thực hiện *for*, chương trình tạo một biến j mới làm biến đếm vòng lặp *for*. Khi *for* kết thúc (sau khi in ra màn hình các số tự nhiên từ 1..100) thì biến *j* của *for* bị xóa bỏ. Biến *j* của *main()* vẫn lưu giá trị 1000.

### *b. Biến toàn cục*

Ngược lại với biến cục bộ, các biến toàn cục được khai báo bên ngoài tất cả các hàm. Vị trí khai báo của chúng có thể ở đầu chương trình hoặc tại một một vị trí bất kỳ nào đó giữa thân chương trình. Thời gian tồn tại của biến toàn cục là từ lúc chương trình bắt đầu chạy đến khi kết thúc chương trình giống như các biến trong hàm *main()*. Tuy nhiên, phạm vi tác dụng của biến toàn cục rộng khắp tới các hàm thuộc chương trình từ khi chúng được tạo ra. Tất cả các hàm thuộc chương trình đều có thể sử dụng và thay đổi giá trị của chúng. Như vậy nếu các biến toàn cục được khai báo từ đầu chương trình sẽ có tác dụng lên toàn bộ chương trình. Tất cả các hàm đều sử dụng được các biến này nếu trong hàm đó không có biến khai báo trùng tên. Một hàm có biến cục bộ (hoặc khối lệnh chứa biến cục bộ) trùng tên với biến toàn cục thì biến toàn cục bị che (không có hiệu lực trong hàm này).

*Ví dụ 4.27*: Chúng ta xét lại các hàm *luythua()* và *clrscr()*. Chú ý rằng trong cả hai hàm này đều có biến *i*. Vì vậy trong ví dụ này, ta khai báo *i* như một biến ngoài (*dùng chung cho cả hai hàm*), ngoài ra *x, n* cũng có thể được khai báo như biến ngoài. Cụ thể:

```
#include <iostream>
using namespace std;
int i, n; float x;
float luythua(float x, int n)// hàm trả giá trị xn
{
    float kq = 1;
```

```
    for (i=1; i<=n; i++) kq *= x;
    return kq;
}

int main()
{
    x = 2; n = 3;
    luythua(x, n);    // in x^i
    return 0;
}
```

Trong ví dụ này ta thấy các biến *x, i, n* đều là các biến toàn cục. Khi ta muốn sử dụng biến toàn cục trong hàm, chẳng hạn biến *i* trong hàm *luythua()*, thì không được khai báo biến cục bộ *i* trong hàm *luythua()* nữa.

Việc sử dụng biến cục bộ và toàn cục trùng tên nhau có thể gây đổ vỡ chương trình của bạn. Tốt nhất, nếu không có lý do gì quan trọng, bạn nên dùng các tên khác nhau.

Một vài nguyên tắc lập trình tránh lỗi xung đột tên biến và làm kết quả chương trình không như ý muốn:

- Nếu một biến chỉ sử dụng vì mục đích riêng của một hàm thì nên khai báo biến đó như biến cục bộ trong hàm.
- Với các biến mang tính chất sử dụng chung rõ nét (đặc biệt với những biến kích thước lớn) mà nhiều hàm cùng sử dụng chúng với mục đích giống nhau thì nên khai báo chúng như biến toàn cục. Điều này tiết kiệm được thời gian cho người lập trình vì không phải khai báo chúng nhiều lần trong nhiều hàm, tiết kiệm bộ nhớ vì không phải tạo chúng tạm thời mỗi khi chạy các hàm, tiết kiệm được thời gian chạy chương trình vì không phải tổ chức bộ nhớ để lưu trữ và giải phóng chúng.
- Cố gắng tạo hàm một cách độc lập, khép kín, không chịu ảnh hưởng của các hàm khác và không gây ảnh hưởng đến hoạt động của các hàm khác đến mức có thể.

### IV.3.2. Các chỉ thị tiền xử lý

Trước khi chạy chương trình (bắt đầu từ chương trình nguồn) C++ sẽ dịch chương trình ra tệp mã máy còn gọi là chương trình đích. Thao tác dịch chương trình nói chung gồm có 2 phần: xử lý sơ bộ chương trình và dịch. Phần xử lý sơ bộ được gọi là tiền xử lý, trong đó có các công việc liên quan đến các chỉ thị được đặt ở đầu tệp chương trình nguồn như *#include, #define* …

#### *a. Chỉ thị bao hàm tệp #include*

Cho phép ghép nội dung các tệp đã có khác vào chương trình trước khi dịch. Các tệp cần ghép thêm vào chương trình thường là các tệp chứa khai báo nguyên mẫu của các hằng, biến, hàm … có sẵn trong C hoặc các hàm do lập trình viên tự viết. Có hai dạng viết chỉ thị này.

```
1: #include <tệp>
2: #include "đường dẫn\tệp"
```

Dạng khai báo 1 cho phép C++ ngầm định tìm tệp tại thư mục định sẵn và các tệp nguyên mẫu của thư viện C++.

Dạng khai báo 2 cho phép tìm tệp theo đường dẫn, nếu không có đường dẫn sẽ tìm trong thư mục hiện tại. Các tệp này thường là tệp (thư viện) được tạo bởi lập trình viên và được đặt trong

cùng thư mục chứa chương trình. Cú pháp này cho phép lập trình viên chia một chương trình thành nhiều môđun đặt trên một số tệp khác nhau để dễ quản lý. Nó đặc biệt hữu ích khi lập trình viên muốn tạo các thư viện riêng cho mình.

***b. Chỉ thị macro #define***

**`#define tên_macro xaukitu`**

Trước khi dịch bộ tiền xử lý sẽ tìm trong chương trình và thay thế bất kỳ vị trí xuất hiện nào của *tên_macro* bởi xâu kí tự. Ta thường sử dụng *macro* để định nghĩa các hằng hoặc thay cụm từ này bằng cụm từ khác dễ nhớ hơn, ví dụ:

```
#define then                       // thay then bằng dấu cách
#define begin {                    // thay begin bằng dấu {
#define end }                      // thay end bằng dấu }
#define MAX 100                    // thay MAX bằng 100
#define TRUE 1                     // thay TRUE bằng 1
```

từ đó trong chương trình ta có thể viết những đoạn lệnh như:

```
if (i < MAX) then
begin
  ok = TRUE;
  cout << i ;
end
```

trước khi dịch bộ tiền xử lý sẽ chuyển đoạn chương trình trên thành

```
if (i < 100)
{
  ok = 1;
  cout << i ;
}
```

theo đúng cú pháp của C++ và rồi mới tiến hành dịch.

Ngoài việc chỉ thị *#define* cho phép thay *tên_macro* bởi một xâu kí tự bất kỳ, nó còn cũng được phép viết dưới dạng có đối. Ví dụ, để tìm số lớn nhất của 2 số, thay vì ta phải viết nhiều hàm max (mỗi hàm ứng với một kiểu số khác nhau), bây giờ ta chỉ cần thay chúng bởi một macro có đối đơn giản như sau:

```
#define max(A,B) ((A) > (B) ? (A): (B))
```

khi đó trong chương trình nếu có dòng x = max(a, b) thì nó sẽ được thay bởi: x = ((a) > (b) ? (a): (b))

*Chú ý:*

- Tên macro phải được viết liền với dấu ngoặc của danh sách đối. Ví dụ không viết `max (A,B)`.
- Cũng tương tự viết `#define max(A,B) (A > B ? A: B)` là sai. Luôn phải bao bọc các đối bởi dấu ngoặc.

## IV.4. Bài tập

### IV.4.1. Hàm

**1.** Chọn câu sai trong các câu sau đây:

A. Hàm không trả lại giá trị thì không cần khai báo kiểu giá trị của hàm.

B. Các biến được khai báo trong hàm là cục bộ, tự xoá khi hàm thực hiện xong

C. Hàm không trả lại giá trị sẽ có kiểu giá trị ngầm định là void.

D. Hàm là đơn vị độc lập, không được khai báo hàm lồng nhau.

**2.** Chọn câu đúng nhất trong các câu sau đây:

A. Hàm phải được kết thúc với 1 câu lệnh return

B. Phải có ít nhất 1 câu lệnh return cho hàm

C. Các câu lệnh return được phép nằm ở vị trí bất kỳ trong thân hàm

D. Không cần khai báo kiểu giá trị trả lại của hàm nếu hàm không có lệnh return

**3.** Chọn câu sai trong các câu sau đây:

A. Số tham số thực sự phải bằng số tham số hình thức trong lời gọi hàm

B. Các biến cục bộ trong thân hàm được chương trình dịch cấp phát bộ nhớ

C. Các tham số hình thức sẽ được cấp phát bộ nhớ tạm thời khi hàm được gọi

D. Kiểu của tham số thực sự phải bằng kiểu của tham số hình thức tương ứng với nó trong lời gọi hàm

**4.** Để thay đổi giá trị của tham biến, các đối của hàm cần khai báo dưới dạng:

A. biến bình thường và tham đối được truyền theo giá trị

B. biến con trỏ và tham đối được truyền theo giá trị

C. biến bình thường và tham đối được truyền theo địa chỉ

D. biến tham chiếu và tham đối được truyền theo giá trị

**5.** Viết hàm tìm UCLN của 2 số. áp dụng hàm này để tìm UCLN của 4 số nhập từ bàn phím.

**6.** Viết hàm kiểm tra số nguyên n có là số nguyên tố. In ra các số nguyên tố bé hơn 1000.

**7.** Viết hàm kiểm tra một số nguyên n có là số nguyên tố. In các cặp số sinh đôi < 1000. (Các số "sinh đôi" là các số nguyên tố mà khoảng cách giữa chúng là 2).

**8.** Viết hàm kiểm tra một năm có là năm nhuận. In ra các năm nhuận từ năm 1000 đến 2000.

**9.** Viết hàm xoá dấu cách đầu tiên trong một xâu. Xoá tất cả dấu cách trong xâu.

**10.** Viết hàm thay 2 dấu cách bởi 1 dấu cách. Xóa các dấu cách "thừa" trong xâu.

**11.** Viết hàm tráo đổi giá trị của 2 số. Sắp xếp dãy số.

**12.** Viết hàm giải phương trình bậc 2. Dùng chương trình con này tìm nghiệm của một phương trình chính phương bậc 4.

**13.** Số hoàn chỉnh là số bằng tổng mọi ước của nó (Ví dụ 6 = 1 + 2 + 3). Hãy in ra mọi số hoàn chỉnh từ 1 đến 100.

**14.** Tính tổng của dãy phân số. In ra màn hình kết quả dưới dạng phân số tối giản.

**15.** Nhập số tự nhiên chẵn n > 2. Hãy kiểm tra số này có thể biểu diễn được dưới dạng tổng của 2 số nguyên tố hay không ?.

**16.** In tổng của n số nguyên tố đầu tiên.

**17.** Tính phần diện tích giới hạn bởi hình tròn bán kính R và hình vuông ngoại tiếp của nó.

**18.** Chuẩn hoá một xâu (cắt kí tự trắng 2 đầu, cắt bớt các dấu trắng (chỉ để lại 1) giữa các từ, viết hoa đầu từ).

**19.** Viết chương trình nhập số nguyên lớn (không quá một tỷ), hãy đọc giá trị của nó bằng cách in ra xâu kí tự tương ứng với giá trị của nó. Ví dụ 1094507 là “Một triệu, (không trăm) chín tư nghìn, năm trăm linh bảy đơn vị”.

**20.** Viết chương trình sắp xếp theo tên một mảng họ tên nào đó.

**21.** Tìm tất cả số tự nhiên có 4 chữ số mà trong mỗi số không có 2 chữ số nào giống nhau.

**22.** Nhập số tự nhiên n. Hãy biểu diễn n dưới dạng tích của các thừa số nguyên tố.

### IV.4.2. Đệ qui

**1.** Nhập số nguyên dương N. Viết hàm đệ qui tính:

a) $S_1 = \dfrac{1+2+3+...+N}{N}$

b) $S_2 = \sqrt{1^2+2^2+3^2+...+N^2}$

**2.** Nhập số nguyên dương n. Viết hàm đệ qui tính:

a) $S_1 = \sqrt{3+\sqrt{3+\sqrt{3+...+\sqrt{3}}}}$ n dấu căn

b) $S_2 = \dfrac{1}{2+\dfrac{1}{2+\dfrac{1}{2+..\dfrac{1}{2}}}}$ n dấu chia

**3.** Viết hàm đệ qui tính n!. áp dụng chương trình con này tính tổ hợp chập k theo công thức truy hồi: C(n, k) = n!/(k! (n-k)!)

**4.** Viết hàm đệ qui tính số fibonaci thứ n. Dùng hàm này tính f(2) + f(4) + f(6) + f(8).

**5.** Viết dưới dạng đệ qui các hàm

a) daoxau b. UCLN c. Fibonaci d. Tháp Hà Nội

**6.** Viết macro tráo đổi nội dung 2 biến. Sắp xếp dãy số.

# CHƯƠNG V. DỮ LIỆU KIỂU CẤU TRÚC - HỢP - LIỆT KÊ

## V.1. Kiểu cấu trúc

Để lưu trữ các giá trị gồm nhiều thành phần dữ liệu giống nhau ta có kiểu biến mảng. Thực tế rất nhiều dữ liệu là tập các kiểu dữ liệu khác nhau tập hợp lại, để quản lý dữ liệu kiểu này C++ đưa ra kiểu dữ liệu cấu trúc. Một ví dụ của dữ liệu kiểu cấu trúc là một danh sách lý lịch các nhân sự, trong đó mỗi nhân sự gồm nhiều thông tin khác nhau như họ tên, tuổi, giới tính, mức lương …

### V.1.1. Khai báo, khởi tạo

Để tạo ra một kiểu cấu trúc, NSD cần phải khai báo tên của kiểu (do NSD tự đặt), cùng với các thành phần dữ liệu trong kiểu cấu trúc này. Một kiểu cấu trúc được khai báo theo mẫu sau:

```
struct <tên kiểu>
{
    <các thành phần> ;
} <danh sách biến>;
```

- Mỗi thành phần giống như một biến riêng của kiểu, nó gồm tên kiểu thành phần và tên thành phần. Một thành phần còn được gọi là trường.
- Phần tên của kiểu cấu trúc và phần danh sách biến có thể có hoặc không. Tuy nhiên cuối khai báo phải có dấu chấm phẩy (;).
- Các kiểu cấu trúc được phép khai báo lồng nhau, nghĩa là một thành phần của kiểu cấu trúc có thể lại là một trường có kiểu cấu trúc.
- Một biến có kiểu cấu trúc sẽ được phân bố bộ nhớ sao cho các thực hiện của nó được sắp liên tục theo thứ tự xuất hiện trong khai báo.
- Khai báo biến kiểu cấu trúc cũng giống như khai báo các biến kiểu cơ sở dưới dạng:

  ```
  struct <tên cấu trúc> <danh sách biến> ;      // kiểu C
  ```

  hoặc

  ```
  <tên cấu trúc> <danh sách biến> ;             // kiểu C++
  ```

  Các biến được khai báo cũng có thể đi kèm khởi tạo:

  ```
  <tên cấu trúc> biến = { giá trị khởi tạo } ;
  ```

  *Ví dụ 5.1:*

- Khai báo kiểu cấu trúc chứa phân số gồm 2 thành phần nguyên chứa tử số và mẫu số.

  ```
  struct Phanso
  {
      int tu ;
      int mau ;
  };
  ```

  hoặc:

  ```
  struct Phanso { int tu, mau ; }
  ```

- Kiểu ngày tháng gồm 3 thành phần nguyên chứa ngày, tháng, năm.

  ```
  struct Ngaythang {
      int ng, th, nam ;
  } holiday = { 1,5,2000 } ;
  ```

- Kiểu *Lop* dùng chứa thông tin về một lớp học gồm tên lớp và sĩ số học sinh. Các biến kiểu *Lop* được khai báo là *lop10* và *lop11*, trong đó *lop10* được khởi tạo bởi bộ giá trị *{"10Tin", 37}* với ý nghĩa tên lớp *lop10* là *10Tin* và sĩ số là 37 học sinh, *lop11* không được khởi tạo giá trị.

```
struct Lop {
    char tenlop[10],
    int soluong;
};
struct Lop lop10 = {"10Tin", 60}, lop11;
```

hoặc:

```
Lop lop10 = {"10Tin", 60}, lop11;
```

- Kiểu *Hocsinh* gồm có các trường *hoten, ngaysinh, gioitinh* (qui ước 1: nam, 2: nữ) và cuối cùng là trường *dtb* lưu trữ điểm trung bình của học sinh. Các trường trên đều có kiểu khác nhau.

```
struct Hocsinh {
    char hoten[25] ;
    Ngaythang ngaysinh;
    int  gioitinh;
    float dtb;
} x, *p, l10tin[37];
Hocsinh y = {"NVA", {1,1,1998}, 1} ;
```

### V.1.2. Truy nhập các thành phần kiểu cấu trúc

Để truy nhập vào các thành phần kiểu cấu trúc ta sử dụng cú pháp:

**<tên biến>.<tên thành phần>;**

hoặc

**<tên biến>-><tên thành phần>;** `//đối với biến con trỏ cấu trúc.`

*Ví dụ 5.2:*

- Đối với biến thường: `<tên biến>.<tên thành phần>`

```
struct Lop {
  char tenlop[10];
  int siso;
} ;
lop l10 = "Lop10Tin", l11 ;
l11.tenlop = l10.tenlop ;
l11.siso++;
```

- Đối với biến con trỏ: `<tên biến>-><tên thành phần>`

```
struct Hocsinh {
    char hoten[25] ;
    Ngaythang ngaysinh;
    int  gioitinh;
    float diem ;
} x, *p, l10tin[37];
Hocsinh y = {"NVA", {1,1,1980}, 1} ;
y.diem = 5.5 ;               // gán điểm thi cho học sinh y
p = new Hocsinh;             // cấp bộ nhớ chứa 1 học sinh
strcpy(p->hoten, y.hoten) ; // gán họ tên của y cho sv trỏ bởi p
cout << p->hoten << y.hoten;// in hoten của y và con trỏ p
```

- Đối với biến mảng: truy nhập thành phần mảng rồi đến thành phần cấu trúc.

```
strcpy(l10tin[0].hoten, p->hoten) ; // gán họ tên cho sv đầu tiên của lớp
l10tin[1].diem = 7.0 ;              // gán điểm cho hs đầu tiên
```

- Đối với cấu trúc lồng nhau. Truy nhập thành phần ngoài rồi đến thành phần của cấu trúc bên trong, sử dụng các phép toán . hoặc -> (các phép toán lấy thành phần) một cách thích hợp.

```
x.ngaysinh.ng = y.ngaysinh.ng ;   // gán ngày,
x.ngaysinh.th = y.ngaysinh.th ;   // tháng,
x.ngaysinh.nam = y.ngaysinh.nam ; // năm sinh của y cho x.
```

### V.1.3. Phép toán gán cấu trúc

Cũng giống các biến mảng, để làm việc với một biến cấu trúc chúng ta phải thực hiện thao tác trên từng thành phần của chúng. Ví dụ vào/ra một biến cấu trúc phải viết câu lệnh vào/ra từng cho từng thành phần.

*Ví dụ 5.3:*

```
struct Hocsinh{
    char hoten[25] ;
    Ngaythang ngaysinh;
    int  gioitinh;
    float diem ;
} x, y;
cout << " Nhap du lieu cho hoc sinh x:" << endl ;
cin.getline(x.hoten, 25);
cin >> x. ngaysinh.ng >> x.ngaysinh.th >> x. ngaysinh.nam;
cin >> x.gioitinh; cin >> x.diem;
cout << "Thong tin ve hoc sinh x la:" << endl ;
cout << "Ho va ten: " << x.hoten << endl;
cout << "Sinh ngyy: " << x.ngaysinh.ng << "/";
cout << x. ngaysinh.th << "/" << x. ngaysinh.nam ;
cout << "Gioi tinh: " << (x.gioitinh == 1) ? "Nam": "Nu";
cout << x.diem;
```

Tuy nhiên, khác với biến mảng, đối với cấu trúc chúng ta có thể gán giá trị của 2 biến cho nhau. Phép gán này cũng tương đương với việc gán từng thành phần của cấu trúc.

*Ví dụ 5.4:*

```
struct Hocsinh{
    char hoten[25] ;
    Ngaythang ngaysinh;
    int  gioitinh;
    float diem ;
} x, y, *p ;
cout << " Nhap du lieu cho hoc sinh x:" << endl ;
cin.getline(x.hoten, 25);
cin >> x.ngaysinh.ng >> x. ngaysinh.th >> x.ngaysinh.nam;
cin >> x.gioitinh;
cin >> x.diem;

y = x ; // Đối với biến mảng, phép gán này là không thực hiện được
p = new Hocsinh[1] ; *p = x ;

cout << "Thong tin sinh vien y la:" << endl ;
cout << "Ho va ten: " << y.hoten << endl;
cout << "Sinh ngay: " << y.ngaysinh.ng << "/" ;
cout << y. ngaysinh.th << "/" << y. ngaysinh.nam ;
cout << "Gioi tinh: " << (y.gt = 1) ? "Nam": "Nu ;
cout << y.diem;
```

*Chú ý*: không gán bộ giá trị cụ thể cho biến cấu trúc. Cách gán này chỉ thực hiện được khi khởi tạo.

*Ví dụ 5.5:*

```
Hocsinh x = { "NVA", {1,1,1980}, 1, 7.0}, y ;      // được
y = {"NVA", {1,1,1980}, 1, 7.0};                   // không được
y = x;                                             // được
```

### V.1.4. Các ví dụ minh họa

*Ví dụ 5.6* : Cộng, trừ, nhân, chia hai phân số.

```
#include <iostream>
using namespace std;
struct Phanso {
    int tu ;
    int mau ;
} a, b, c ;
int main()
{
    cout << "Nhap phan so a:" << endl ;
    cout << "Tu:"; cin >> a.tu;
    cout << "Mau:"; cin >> a.mau;
     cout << "Nhap phan so b:" << endl ;
    cout << "Tu:"; cin >> b.tu;
    cout << "Mau:"; cin >> b.mau;
    c.tu = a.tu*b.mau + a.mau*b.tu;
    c.mau = a.mau*b.mau;
    cout << "a + b = " << c.tu << "/" << c.mau<<endl;
    c.tu = a.tu*b.mau - a.mau*b.tu;
    c.mau = a.mau*b.mau;
    cout << "a - b = " << c.tu << "/" << c.mau<<endl;
    c.tu = a.tu*b.tu;
    cout << "a + b = " << c.tu << "/" << c.mau<<endl;
    c.tu = a.tu*b.mau;
    cout << "a + b = " << c.tu << "/" << c.mau<<endl;
    return 0;
}
```

*Ví dụ 5.7*: Nhập mảng l10tin. Tính điểm trung bình của học sinh nam, nữ. Hiện danh sách của học sinh có điểm trung bình cao nhất.

```
#include <iostream>
using namespace std;
struct Ngaythang{
    int ng;
    int th;
    int nam;
};
struct Hocsinh{
    char hoten[25];
    Ngaythang ngaysinh;
    int  gioitinh;
    float dtb;
};
int main()
{
    Hocsinh x, l10tin[37];
    int i, n;
    cout << "Nhap so sinh vien: ";
```

```
        cin >> n; cin.ignore();        //xoa bo dem ban phim
        for (i = 1; i <= n; i++)
        {
            cout <<i<< "> Ho ten: " ;
            cin.getline(x.hoten,25);
            cout << "Ngay sinh: " ;
            cin >> x.ngaysinh.ng >> x.ngaysinh.th >>x.ngaysinh.nam ;
            cout << "Gioi tinh: " ; cin >> x.gioitinh;
            cout << "DTB: " ; cin >> x.dtb ;
            cin.ignore(); //xoa bo dem ban phim
            l10tin[i] = x ;
        }
        float tbnam = 0, tbnu = 0;
        int sonam = 0, sonu = 0 ;
        for (i=1; i<=n; i++)
            if (l10tin[i].gioitinh == 1)
            {
                sonam++ ; tbnam += l10tin[i].dtb ;
            }
            else
            {
                sonu++ ; tbnu += l10tin[i].dtb ;
            }
        cout << "DTB cua hoc sinh nam la: " << tbnam/sonam<<endl;
        cout << "DTB cua hoc sinh nu la: " << tbnu/sonu<<endl ;

        float diemmax = 0;
        for (i=1; i<=n; i++)
            if (diemmax < l10tin[i].dtb) diemmax = l10tin[i].dtb ;

        for (i=1; i<=n; i++)
        {
            if (l10tin[i].dtb < diemmax) continue ;
            x = l10tin[i] ;
            cout <<i<<">"<< x.hoten << '\t' ;
            cout << x.ngaysinh.ng << "/" << x.ngaysinh.th << "/";
            cout << x.ngaysinh.nam << '\t' ;
            cout<<(x.gioitinh==1)?"Nam":"Nu";
            cout <<'\t'<< x.dtb << endl;
        }
        return 0;
    }
```

### V.1.5. Hàm với cấu trúc

#### *a. Con trỏ và địa chỉ cấu trúc*

Một con trỏ cấu trúc cũng giống như con trỏ trỏ đến các kiểu dữ liệu khác, có nghĩa nó chứa địa chỉ của một biến cấu trúc hoặc một vùng nhớ có kiểu cấu trúc nào đó. Một con trỏ cấu trúc được khởi tạo bởi:

- Gán địa chỉ của một biến cấu trúc, một thành phần của mảng, tương tự nếu địa chỉ của mảng gán cho con trỏ thì ta cũng gọi là con trỏ mảng cấu trúc.

*Ví dụ 5.8:*

```
struct Hocsinh{
    char hoten[25];
    Ngaythang ngaysinh;
```

```
    int gioitinh;
    float dtb;
}x, y, *p, lop[60];

p = &x ;              // cho con trỏ p trỏ tới biến cấu trúc x
p->diem = 5.0;        // gán giá trị 5.0 cho điểm của biến x
p = &lop[10] ;        // cho p trỏ tới học sinh thứ 10 của lớp
cout << p->hoten;     // hiện họ tên của học sinh này
*p = y ;              // gán lại học sinh thứ 10 là y
(*p).gt = 2;          // sửa lại giới tính của học sinh thứ 10 là nữ
```

*Chú ý:* thông qua ví dụ này ta còn thấy một cách khác nữa để truy nhập các thành phần của x được trỏ bởi con trỏ p. Khi đó *p là tương đương với x, do vậy ta dùng lại cú pháp sử dụng toán tử. sau *p để lấy thành phần như (*p).hoten, (*p).diem, …

- Con trỏ được khởi tạo do xin cấp phát bộ nhớ.

```
Hocsinh *p, *q ;
p = new Hocsinh[1];
q = new Hocsinh[60];
```

trong ví dụ này *p có thể thay cho một biến kiểu Hocsinh (tương đương biến x ở trên) còn q có thể được dùng để quản lý một danh sách có tối đa là 60 học sinh (ví dụ *(p+10) là học sinh thứ 10 trong danh sách).

- Đối với con trỏ p trỏ đến mảng a, chúng ta có thể sử dụng một số cách sau để truy nhập đến các trường của các thành phần trong mảng, ví dụ để truy cập hoten của thành phần thứ i của mảng a ta có thể viết:

```
p[i].hoten
(p+i)->hoten
*(p+i).hoten
```

Nói chung các cách viết trên đều dễ nhớ do suy từ kiểu mảng và con trỏ mảng. Cụ thể trong đó p[i] là thành phần thứ i của mảng a, tức a[i]. (p+i) là con trỏ trỏ đến thành phần thứ i và *(p+i) chính là a[i]. Ví dụ sau gán giá trị cho thành phần thứ 10 của mảng lop, sau đó in ra màn hình các thông tin này. Ví dụ dùng để minh hoạ các cách truy nhập trường dữ liệu của thành phần trong mảng lop.

*Ví dụ 5.9 :*

```
struct Hocsinh {
char hoten[25] ;
    Ngaythang ns;
    int gt;
    float diem ;
} lop[60] ;

strcpy(lop[10].hoten, "NVA");
lop[10].gt = 1;
lop[10].diem = 9.0 ;
Hocsinh *p ;              // khai báo thêm biến con trỏ Học sinh
p = &lop ;                // cho con trỏ p trỏ tới mảng lop
cout << p[10].hoten ;     // in họ tên học sinh thứ 10
cout << (p+10)->gt ;      // in giới tính của học sinh thứ 10
cout << (*(p+10)).diem ; // in điểm của học sinh thứ10
```

*Chú ý:* Độ ưu tiên của toán tử lấy thành phần (dấu chấm) là cao hơn các toán tử lấy địa chỉ (&) và lấy giá trị (*) nên cần phải viết các dấu ngoặc đúng cách.

***b. Địa chỉ của các thành phần của cấu trúc***

Các thành phần của một cấu trúc cũng giống như các biến, do vậy cách lấy địa chỉ của các thành phần này cũng tương tự như đối với biến bình thường. Chẳng hạn địa chỉ của thành phần giới tính của biến cấu trúc x là &x.gt (lưu ý độ ưu tiên của . cao hơn &, nên &x.gt là cũng tương đương với &(x.gt)), địa chỉ của trường hoten của thành phần thứ 10 trong mảng lớp là lop[10].hoten (hoten là xâu kí tự), tương tự địa chỉ của thành phần diem của biến được trỏ bởi p là &(p->diem).

*Ví dụ 5.10:*

```
struct Hocsinh{
    char hoten[25] ;
    Ngaythang ns;
    int gt;
    float diem ;
} lop[60], *p, x = { "NVA", {1,1,1980}, 1, 9.0) };
lop[10] = x; p = &lop[10] ; // p trỏ đến học sinh thứ 10 trong lop
char *ht; int *gt; float *d;// các con trỏ kiểu thành phần
ht = x.ht ;                 // cho ht trỏ đến thành phần hoten của x
gt = &(lop[10].gt) ;        // gt trỏ đến gt của học sinh thứ 10
d = &(p->diem) ;          // p trỏ đến diem của sv p đang trỏ
cout << ht ;              // in họ tên học sinh x
cout << *gt ;             // in giới tính của học sinh thứ 10
cout << *d ;              // in điểm của học sinh p đang trỏ.
```

***c. Đối của hàm là cấu trúc***

Một cấu trúc có thể được sử dụng để làm đối của hàm dưới các dạng sau đây:

- Là một biến cấu trúc, khi đó tham đối thực sự là một cấu trúc.
- Là một con trỏ cấu trúc, tham đối thực sự là địa chỉ của một cấu trúc.
- Là một tham chiếu cấu trúc, tham đối thực sự là một cấu trúc.
- Là một mảng cấu trúc hình thức hoặc con trỏ mảng, tham đối thực sự là tên mảng cấu trúc.

*Ví dụ 5.11* : Ví dụ sau đây cho phép tính chính xác khoảng cách của 2 ngày tháng bất kỳ, từ đó có thể suy ra thứ của một ngày tháng bất kỳ. Đối của các hàm là một biến cấu trúc.

```
struct DATE {
    int ngay ;
    int thang;
    int nam ;
};
// Số ngày của mỗi tháng
int n[13] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
```

- Hàm tính năm nhuận hay không nhuận, trả lại 1 nếu năm nhuận, ngược lại trả 0.

```
int Nhuan(int nm)
{
    return (nam%4==0 && nam%100!=0 || nam%400==0) ? 1 : 0;
}
```

- Hàm trả lại số ngày của một tháng bất kỳ. Nếu năm nhuận và là tháng hai số ngày của tháng hai (28) được cộng thêm 1.

```
int Ngayct(int thang, int nam)
{
    return n[thang] + ((thang==2) ? Nhuan(nam): 0);
}
```

- Hàm trả lại số ngày tính từ ngày 1 tháng 1 năm 1 bằng cách cộng dồn số ngày của từng năm từ năm 1 đến năm hiện tại, tiếp theo cộng dồn số ngày từng tháng của năm hiện tại cho đến tháng hiện tại và cuối cùng cộng thêm số ngày hiện tại.

```
long Tongngay(DATE d)
{
    long i, kq = 0;
    for (i=1; i<d.nam; i++) kq += 365 + Nhuan(i);
    for (i=1; i<d.thang; i++) kq += Ngayct(i,d.nam);
    kq += d.ngay;
    return kq;
}
```

- Hàm trả lại khoảng cách giữa 2 ngày bất kỳ.

```
long Khoangcach(DATE d1, DATE d2)
{
    return Tongngay(d1)-Tongngay(d2);
}
```

- Hàm trả lại thứ của một ngày bất kỳ. Qui ước 1 là chủ nhật, 2 là thứ hai, … Để tính thứ hàm dựa trên một ngày chuẩn nào đó (ở đây là ngày 1/1/2000, được biết là thứ Bảy). Từ ngày chuẩn này nếu cộng hoặc trừ 7 sẽ cho ra ngày mới cũng là thứ bảy. Từ đó, tính khoảng cách giữa ngày cần tính thứ và ngày chuẩn. Tìm phần dư của phép chia khoảng cách này với 7, nếu phần dư là 0 thì thứ là bảy, phần dư là 1 thì thứ là chủ nhật …

```
int Thu(DATE d)
{
    DATE curdate = {1,1,2000};          // ngày 1/1/2000 là thứ bảy
    long kc = Khoangcach(d, curdate);
    int du = kc % 7; if (du < 0) du += 7;
    return du;
}
```

- Hàm dịch một số dư sang thứ trong tuần

```
char* Dich(int t)
{
    char* kq = new char[10];
    switch (t) {
        case 0: strcpy(kq, "thứ bảy"); break;
        case 1: strcpy(kq, "chủ nhật"); break;
        case 2: strcpy(kq, "thứ hai"); break;
        case 3: strcpy(kq, "thứ ba"); break;
        case 4: strcpy(kq, "thứ tư"); break;
        case 5: strcpy(kq, "thứ năm"); break;
        case 6: strcpy(kq, "thứ sáu"); break;
    }
    return kq;
}
```

- Hàm *main()*

```
void main()
{
    DATE d;
    cout << "Nhap ngay thang nam: " ;
    cin >> d.ngay >> d.thang >> d.nam ;
    cout << "Ngày " << d.ngay << "/" << d.thang << "/" << d.nam ;
    cout << " là " << Dich(Thu(d));
}
```

*Ví dụ 5.12* : Chương trình quản lý học sinh đơn giản.

```
struct Hocsinh{
    char hoten[25] ;
    Ngaythang ns;
    int gt;
};
Hocsinh lop[40];          // lớp chứa tối đa 40 học sinh
```

- Hàm in thông tin học sinh sử dụng biến cấu trúc làm đối.

```
void in(Hocsinh x)
{
    cout << x.hoten << "\t" ;
    cout << x.ns.ng << "/" << x.ns.th << "/" << x.ns.nam << "\t" ;
    cout << x.gt << "\t";
    cout << x.diem << endl;
}
```

- Hàm nhập thông tin học sinh sử dụng con trỏ học sinh làm đối. Trong lời gọi sử dụng địa chỉ của một cấu trúc để truyền cho hàm.

```
void nhap(Hocsinh *p)
{
    cin.ignore();
    cout << "Họ tên: "; cin.getline(p->hoten, 25) ; cin.inogre();
    cout << "Ngày sinh: ";
    cin >> (p->ns).ng >> (p->ns).th >> (p->ns).nam ;
    cout << "Giới tính: "; cin >> (p->gt) ;
    cout << "Điểm: "; cin >> (p->diem) ; cin.inogre();
}
```

- Hàm sửa thông tin về học sinh sử dụng tham chiếu cấu trúc làm đối. Trong lời gọi sử dụng biến cấu trúc để truyền cho hàm.

```
void sua(Hocsinh &r)
{
    int chon;
    do {
        cout << "1: Sửa họ tên" << endl ;
        cout << "2: Sửa ngày sinh" << endl ;
        cout << "3: Sửa giới tính" << endl ;
        cout << "4: Sửa điểm" << endl ;
        cout << "0: Thôi" << endl ;
        cout << "Sửa (0/1/2/3/4) ? ; cin >> chon ; cin.ignore();
        switch (chon) {
           case 1: cin.getline(r.hoten, 25) ; cin.ignore(); break;
           case 2: cin >> r.ns.ng >> r.ns.th >> r.ns.nam ; break;
           case 3: cin >> r.gt ; break;
           case 4: cin >> r.diem ; break;
        }
    } while (chon) ;
}
```

- Hàm nhập thông tin của mọi học sinh trong mảng, sử dụng con trỏ mảng Hocsinh làm tham đối hình thức. Trong lời gọi sử dụng tên mảng để truyền cho hàm.

```
void nhapds(Hocsinh *a)
{
    int sosv = sizeof(lop) / sizeof(Hocsinh) -1;   // bỏ phần tử 0
    for (int i=1; i<=sosv; i++) nhap(&a[i]) ;
}
```

- Hàm sửa thông tin của học sinh trong mảng, sử dụng con trỏ mảng Hocsinh làm tham đối hình thức. Trong lời gọi sử dụng tên mảng để truyền cho hàm.

```
void suads(Hocsinh *a)
{
    int chon;
    cout << "Chọn học sinh cần sửa: " ; cin >> chon ; cin.ignore();
    sua(a[chon]) ;
}
```

- Hàm in thông tin của mọi học sinh trong mảng, sử dụng hằng con trỏ mảng Hocsinh làm tham đối hình thức. Trong lời gọi sử dụng tên mảng để truyền cho hàm.

```
void hien(const Hocsinh *a)
{
    int sosv = sizeof(lop) / sizeof(Hocsinh)-1;    // bỏ phần tử 0
    for (int i=1; i<=sosv; i++) in(a[i]) ;
}
```

- Hàm *main():*

```
void main()
{
    nhapds(lop) ;
    inds(lop);
    suads(lop);
    inds(lop);
}
```

***d. Giá trị hàm là cấu trúc***

Cũng tương tự như các kiểu dữ liệu cơ bản, giá trị trả lại của một hàm cũng có thể là các cấu trúc dưới các dạng sau:

- là một biến cấu trúc.
- là một con trỏ cấu trúc.
- là một tham chiếu cấu trúc.

Sau đây là các ví dụ minh hoạ giá trị cấu trúc của hàm.

*Ví dụ 5.13* : Chương trình nhập và in thông tin một lớp, và học sinh có điểm cao nhất lớp.

- Khai báo kiểu dữ liệu Hocsinh và biến mảng lop.

```
struct Hocsinh {
    char *hoten ;
    float diem ;
} lop[4] ;
```

- Hàm nhập học sinh, giá trị trả lại là một con trỏ trỏ đến dữ liệu vừa nhập.

```
Hocsinh* nhap()
{
    Hocsinh* kq = new Hocsinh[1];       // nhớ cấp phát vùng nhớ
    kq->hoten = new char[15];           // cho cả con trỏ hoten
    cout << "Họ tên: "; cin.getline(kq->hoten,30); cin.ignore();
    cout << "Điểm: "; cin >> kq->diem; cin.ignore();
    return kq;                          // trả lại con trỏ kq
}
```

- Hàm tìm học sinh có điểm cao nhất, giá trị trả lại là một tham chiếu đến học sinh tìm được.

```
Hocsinh& svmax()
{
    int sosv = sizeof(lop)/sizeof(Hocsinh)-1; // bỏ thành phần thứ 0
    float maxdiem = 0;
    int kmax;                                  // chỉ số hs có điểm max
    for (int i = 1; i < sosv; i++)
    if (maxdiem < lop[i].diem)
    {
        maxdiem = lop[i].diem ;
        kmax = i;
    }
    return lop[kmax];                    // trả lại hs có điểm max
}
```

- Hàm in thông tin của một học sinh x

```
void in(Hocsinh x)
{
    cout << x.hoten << "\t";
    cout << x.diem << endl;
}
```

- Hàm *main()*

```
int main()
{
    int i;
    int sosv = sizeof(lop)/sizeof(Hocsinh)-1; // bỏ thành phần thứ 0
    for (i=1; i<=sosv; i++) lop[i] = *nhap(); // nhập danh sách lớp
    for (i=1; i<=sosv; i++) in(lop[i]);       // in danh sách lớp
    Hocsinh &b = svmax();    // khai báo tham chiếu b và cho
    in(b);                   // tham chiếu đến sv có điểm max
    return 0;                // in học sinh có điểm max
}
```

### V.1.6. Câu lệnh typedef

Để thuận tiện trong sử dụng, thông thường các kiểu được NSD tạo mới sẽ được gán cho một tên kiểu bằng câu lệnh typedef như sau:

```
typedef <kiểu> <tên_kiểu> ;
```

*Ví dụ 5.14*: Để tạo kiểu mới có tên Bool và chỉ chứa giá trị nguyên (thực chất chỉ cần 2 giá trị 0, 1), ta có thể khai báo:

```
typedef int Bool;
```

khai báo này cho phép xem Bool như kiểu số nguyên.

hoặc có thể đặt tên cho kiểu ngày tháng là Date với khai báo sau:

```
typedef struct Date {
    int ng;
    int th;
    int nam;
};
```

khi đó ta có thể sử dụng các tên kiểu này trong các khai báo (ví dụ tên kiểu của đối, của giá trị hàm trả lại …).

### V.1.7. Hàm sizeof()

Hàm trả lại kích thước của một biến hoặc kiểu. Ví dụ:

```
Bool a, b;
```

```
    Date x, y, z[50];
    cout << sizeof(a) << sizeof(b) << sizeof(Bool); // in 2 2 2
    cout << "So ptu cua z = " << sizeof(z) / sizeof(Date);    // in 50
```

## V.2. Cấu trúc tự trỏ và danh sách liên kết

Như đã nói ở chương III về con trỏ và cấp phát động: Để khắc phục tình thừa hoặc thiếu bộ nhớ này C++ cho phép cấp phát bộ nhớ động cho các biến mảng bằng toán tử *new,* và thu hồi bộ nhớ bằng toán tử *delete.* Tuy nhiên, bộ nhớ cấp phát động cho biến mảng vẫn là các khối nhớ liên tiếp và đôi khi không đủ khối nhớ liên tiếp để cấp phát. Mặt khác, mảng lưu trên các khối nhớ liên tiếp dẫn đến, các thao tác chèn hoặc xóa phần tử của mảng tốn nhiều thời gian dồn hoặc giãn vị trí các phần tử.

Trong phần này, ta xem xét đến một cách xin cấp phát động bộ nhớ để lưu kiểu dữ liệu danh sách (như mảng) theo nhu cầu mỗi khi chạy chương trình mà không đòi hỏi các khối nhớ liên tiếp. Tuy nhiên, cách làm này dẫn đến việc dữ liệu của một danh sách nào đó sẽ không còn nằm liên tục trong bộ nhớ như đối với biến mảng.

Mỗi lần xin cấp phát bởi toán tử *new*, chương trình sẽ tìm một vùng nhớ đang rỗi bất kỳ để cấp phát cho biến và như vậy dữ liệu sẽ nằm rải rác trong bộ nhớ. Để dễ dàng quản lý trật tự của một danh sách các dữ liệu, mỗi phần tử của danh sách phải chứa địa chỉ của phần tử tiếp theo hoặc trước nó (điều này không cần thiết đối với biến mảng vì các phần tử của mảng là liên tục, kề nhau). Như vậy, mỗi phần tử của danh sách phải là một cấu trúc, ngoài các thành phần chứa thông tin cần thiết, còn phải có thêm một hoặc nhiều con trỏ để trỏ đến các phần tử tiếp theo hay trước đó. Các cấu trúc như vậy được gọi là *cấu trúc tự trỏ* vì các thành phần con trỏ trong cấu trúc này sẽ trỏ đến các vùng dữ liệu có kiểu chính là kiểu của chúng.

### V.2.1. Cấu trúc tự trỏ

Một cấu trúc chứa ít nhất một thành phần là con trỏ có kiểu chính là kiểu của cấu trúc đang định nghĩa được gọi là cấu trúc tự trỏ. Có thể khai báo cấu trúc tự trỏ bởi một trong những cách sau:

Cách 1:

```
typedef struct <tên cấu trúc> <tên kiểu> ; // định nghĩa tên cấu trúc
struct <tên cấu trúc>
{
    các thành phần chứa thông tin … ;
    <tên kiểu> *con trỏ ;
} ;
```

*Ví dụ 5.15:*

```
typedef struct Hs Hocsinh ; // lưu ý phải có từ khoá struct
struct Hs
{
    char hoten[30] ; // thành phần chứa thông tin
    float diem ;      // thành phần chứa thông tin
    Hocsinh *tiep ;   // thành phần con trỏ chứa địa chỉ tiếp theo
} ;
```

Cách 2:

```
struct <tên cấu trúc>
{
    các thành phần chứa thông tin … ;
    <tên cấu trúc> *con trỏ ;
} ;
```

```
typedef <tên cấu trúc> <tên kiểu> ; // định nghĩa tên cấu trúc tự trỏ
```

*Ví dụ 5.16:*

```
struct Hs
{
    char hoten[30] ; // thành phần chứa thông tin
    float diem ;     // thành phần chứa thông tin
    Sv *tiep ;       // thành phần con trỏ chứa địa chỉ tiếp theo
} ;
typedef Hs Hocsinh ;
```

Cách 3:

```
typedef struct <tên kiểu>    // định nghĩa tên cấu trúc tự trỏ
{
    các thành phần chứa thông tin … ;
    <tên kiểu> *con trỏ ;
} ;
```

*Ví dụ 5.17:*

```
typedef struct Hocsinh
{
    char hoten[30] ; // thành phần chứa thông tin
    float diem ;     // thành phần chứa thông tin
    Hocsinh *tiep ;  // con trỏ chứa địa chỉ thành phần tiếp theo
} ;
```

Cách 4:

```
struct <tên kiểu>
{
    các thành phần chứa thông tin … ;
    <tên kiểu> *con trỏ ;
} ;
```

*Ví dụ 5.18:*

```
struct Hocsinh
{
    char hoten[30] ; // thành phần chứa thông tin
    float diem ;     // thành phần chứa thông tin
    Hocsinh *tiep ;  // con trỏ chứa địa chỉ của phần tử tiếp.
} ;
```

Ta thấy 2 cách khai báo cuối cùng là đơn giản nhất. C++ quan niệm các tên gọi đứng sau các từ khoá struct, union, enum là các tên kiểu (dù không có từ khoá typedef), do vậy có thể sử dụng các tên này để khai báo.

### V.2.2. Khái niệm danh sách liên kết

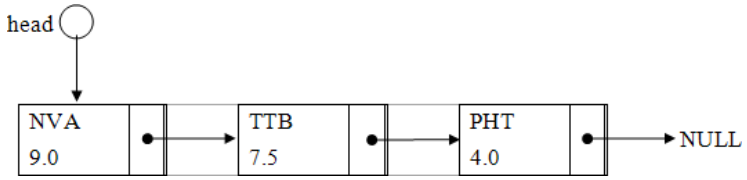
Danh sách liên kết là một cấu trúc dữ liệu cho phép thể hiện và quản lý danh sách bằng các cấu trúc liên kết với nhau thông qua các con trỏ trong cấu trúc. Có nhiều dạng danh sách liên kết phụ thuộc vào các kết nối, ví dụ:

- Danh sách liên kết đơn, mỗi cấu trúc chứa một con trỏ trỏ đến cấu trúc tiếp theo hoặc trước đó. Đối với danh sách con trỏ trỏ về trước, cấu trúc đầu tiên của danh sách sẽ trỏ về hằng con trỏ NULL, cấu trúc cuối cùng được đánh dấu bởi con trỏ last là con trỏ trỏ vào cấu trúc này. Đối với danh sách con trỏ trỏ về cấu trúc tiếp theo, cấu trúc đầu sẽ được đánh dấu bằng con trỏ head và cấu trúc cuối cùng chưa con trỏ NULL.

- Danh sách liên kết kép gồm 2 con trỏ, một trỏ đến cấu trúc trước và một trỏ đến cấu trúc sau, 2 đầu của danh sách được đánh dấu bởi các con trỏ head, last.
- Danh sách liên kết vòng gồm 1 con trỏ trỏ về sau (hoặc trước), hai đầu của danh sách được nối với nhau tạo thành vòng tròn. Chỉ cần một con trỏ head để đánh dấu đầu danh sách.

Do trong cấu trúc có chứa các con trỏ trỏ đến cấu trúc tiếp theo và/hoặc cấu trúc đứng trước nên từ một cấu trúc này chúng ta có thể truy cập đến một cấu trúc khác (trước và/hoặc sau nó). Kết hợp với các con trỏ đánh dấu 2 đầu danh sách (head, last) chúng ta sẽ dễ dàng làm việc với bất kỳ phần tử nào của danh sách. Có thể kể một số công việc thường thực hiện trên một danh sách như: bổ sung phần tử vào cuối danh sách, chèn thêm một phần tử mới, xoá một phần tử của danh sách, tìm kiếm, sắp xếp danh sách, in danh sách …

Hình 5.1 minh hoạ một danh sách liên kết đơn quản lý học sinh, thông tin chứa trong mỗi phần tử của danh sách gồm có họ tên học sinh, điểm. Ngoài ra mỗi phần tử còn chứa con trỏ *tiep* để nối với phần tử tiếp theo của nó. Phần tử cuối cùng nối với cấu trúc rỗng (NULL).




Hình 5.1

### V.2.3. Các phép toán trên danh sách liên kết

Dưới đây là mô tả tóm tắt cách thức thực hiện một số thao tác trên danh sách liên kết đơn.

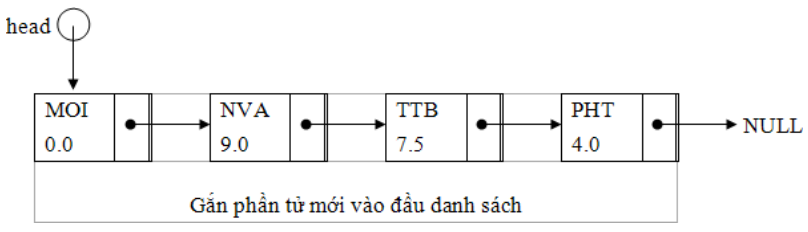
#### *a. Tạo phần tử mới*

Để tạo phần tử mới thông thường chúng ta thực hiện theo các bước sau đây:

- Dùng toán tử *new* xin cấp phát một vùng nhớ đủ chứa một phần tử của danh sách.
- Nhập thông tin cần lưu trữ vào phần tử mới. Con trỏ *next* được đặt bằng NULL.
- Gắn phần tử vừa tạo được vào danh sách. Có hai cách:

➢ hoặc gắn vào đầu danh sách, khi đó vị trí của con trỏ *head* (chỉ vào đầu danh sách) được điều chỉnh lại để chỉ vào phần tử mới.

➢ hoặc gắn vào cuối danh sách bằng cách cho con trỏ *next* của phần tử cuối danh sách (đang trỏ vào NULL) trỏ vào phần tử mới. Nếu danh sách có con trỏ *last* để chỉ vào cuối danh sách thì *last* được điều chỉnh để trỏ vào phần tử mới. Nếu danh sách không có con trỏ *last* thì để tìm được phần tử cuối chương trình phải duyệt từ đầu, bắt đầu từ con trỏ *head* cho đến khi gặp phần tử trỏ vào NULL, đó là phần tử cuối của danh sách.

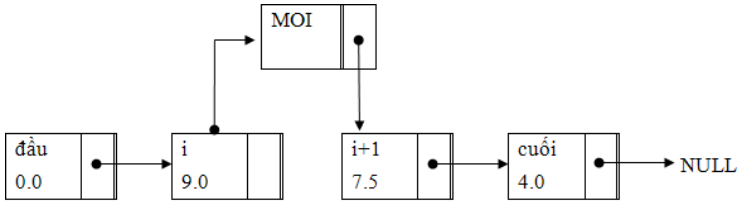



Hình 5.2

### *b. Chèn phần tử mới vào giữa*

Giả sử phần tử mới được chèn vào giữa phần tử thứ i và i+1. Để chèn ta nối phần tử thứ i vào phần tử mới và phần tử mới nối vào phần tử thứ i+1. Thuật toán sẽ như sau:

- Cho con trỏ p chạy đến phần tử thứ i.
- Cho con trỏ tiep của phần tử mới trỏ vào phần tử thứ i+1 (tuc p->tiep).
- Cho con trỏ tiep của phần tử thứ i (hiện được trỏ bởi p) thay vì trỏ vào phần tử thứ i+1 bây giờ sẽ trỏ vào phần tử mới.



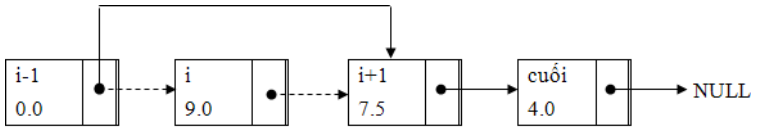

Chèn phần tử mới vào giữa phần tử i và i+1

Hình 5.3

### *c. Xoá phần tử thứ i khỏi danh sách*

Việc xoá một phần tử ra khỏi danh sách rất đơn giản bởi chỉ việc thay đổi các con trỏ. Cụ thể giả sử cần xoá phần tử thứ i ta chỉ cần cho con trỏ tiep của phần tử thứ i-1 trỏ ("vòng qua" phần tử thứ i) vào phần tử thứ i+1. Như vậy bây giờ khi chạy trên danh sách đến phần tử thứ i-1, phần tử tiếp theo là phần tử thứ i+1 chứ không còn là phần tử thư i. Nói cách khác phần tử thứ i không được nối bởi bất kỳ phần tử nào nên nó sẽ không thuộc danh sách. Có thể thực hiện các bước như sau:

- Cho con trỏ p chạy đến phần tử thứ i-1.
- Đặt phần tử thứ i vào biến x.
- Cho con trỏ tiep của phần tử thứ i-1 trỏ vào phần tử thứ i+1 bằng cách đặt tiep = x.tiep.
- Giải phóng bộ nhớ được trỏ bởi x bằng câu lệnh delete x.




Xóa phần tử thứ i

Hình 5.4

### *d. Duyệt danh sách*

Duyệt là thao tác đi qua từng phần tử của danh sách, tại mỗi phần tử chương trình thực hiện một công việc gì đó trên phần tử mà ta gọi là thăm phần tử đó. Một phép thăm có thể đơn giản là hiện nội dung thông tin của phần tử đó ra màn hình chẳng hạn. Để duyệt danh sách ta chỉ cần cho một con trỏ p chạy từ đầu đến cuối danh sách đến khi phần tử cuối có con trỏ tiep = NULL thì dừng. Câu lệnh cho con trỏ p chuyển đến phần tử tiếp theo của nó là:

```
p = p->tiep ;
```

***e. Tìm kiếm***

Cho một danh sách trong đó mỗi phần tử của danh sách đều chứa một trường gọi là trường khoá, thường là các trường có kiểu cơ sở hoặc kết hợp của một số trường như vậy. Bài toán đặt ra là tìm trên danh sách phần tử có giá trị của trường khoá bằng với một giá trị cho trước. Tiến trình thực hiện nhiệm vụ thựcchất cũng là bài toán duyệt, trong đó thao tác "thăm" chính là so sánh trường khoá của phần tử với giá trị cho trước, nếu trùng nhau ta in kết quả và dừng, Nếu đã duyệt hết mà không có phần tử nào có trường khoá trùng với giá trị cho trước thì xem danh sách không chứa giá trị này.

Ngoài các thao tác trên, nói chung còn nhiều các thao tác quen thuộc khác tuy nhiên chúng ta không trình bày ở đây vì nó không thuộc phạm vi của giáo trình này.

*Ví dụ 5.19*: Dưới đây là một ví dụ minh hoạ cho cấc cấu trúc tự trỏ, danh sách liên kết và một vài thao tác trên danh sách liên kết thông qua bài toán quản lý học sinh.

+ *Khai báo:*

```
struct DATE
{
    int day, month, year;    // ngày, tháng, năm
};
struct Hocsinh {             // cấu trúc tự trỏ
    char hoten[31];
    DATE ns;
    float diem;
    Hocsinh *next ;
};
Hocsinh *dau = NULL, *cuoi = NULL;// Các con trỏ tới đầu và cuối ds
Hocsinh *cur = NULL;              // Con trỏ tới sv hiện tại
int sosv = 0;                     // Số sv của danh sách
```

+ *Tạo học sinh mới và nhập thông tin, trả lại con trỏ trỏ đến học sinh mới.*

```
Hocsinh* Nhap1sv()       // Tạo 1 khối dữ liệu cho sv mới
{
    Hocsinh *kq = new Hocsinh[1] ; // Cấp phát bộ nhớ cho kq
    cout << "\nHoc sinh thu ", sosv+1 ;
    cout << "Ho ten = " ; cin.getline(kq->hoten);
    cout << "Ns = " ;
    cin >> kq->ns.day >> kq->ns.month >> kq->ns.year;
    cout << "Diem = " ; cin >> kq->diem ; cin.ignore() ;
    kq->next = NULL;
    return kq ;
}
```

+ *Bổ sung học sinh mới vào cuối danh sách.*

```
void Bosung()         // Bổ sung sv mới vào cuối ds
{
    cur = Nhap1sv();
    if (sosv == 0) {dau = cuoi = cur;}
    else { cuoi->next = cur; cuoi = cur; }
    sosv++;
}
```

+ *Chèn học sinh mới vào trước học sinh thứ n.*

```
void Chentruoc(int n)        // Chèn hs mới vào trước hs thứ n
{
    cur = Nhap1sv();
```

```
    if (sosv==0) { dau = cuoi = cur; sosv++; return; }
    if (sosv==1 || n==1)
    {
        cur->next = dau; dau = cur; sosv++; return;
    }
    Hocsinh *truoc, *sau;
    truoc = dau;
    sau = dau -> next;
    for (int i=1; i<n-1; i++) truoc = truoc->next;
    sau = truoc->next;
    truoc->next = cur;
    cur -> next = sau;
    sosv ++;
}
```

+ *Chèn học sinh mới vào sau học sinh thứ n.*

```
void Chensau(int n)          // Chèn hs mới vào sau hs thứ n
{
    cur = Nhap1sv();
    if (sosv==0 || sosv<n) { dau = cuoi = cur; sosv++; return; }
    Hocsinh *truoc, *sau;
    truoc = dau; sau = dau -> next;
    for (int i=1; i<n; i++) truoc = truoc->next;
    sau = truoc->next;
    truoc->next = cur;
    cur -> next = sau;
    sosv ++;
}
```

+ *Xoá học sinh thứ n.*

```
void Xoa(int n)              // Xoá học sinh thứ n
{
    if (sosv==1&&n==1) { delete dau ; dau = cuoi = NULL; sosv--;
return; }
    if (n==1) { cur = dau; dau = cur->next; delete cur; sosv--;
return; }
    Hocsinh *truoc, *sau;
    truoc = dau;
    sau = dau -> next;
    for (int i=1; i<n-1; i++) truoc = truoc->next;
    cur = truoc->next; sau = cur->next; truoc->next = sau;
    delete cur ;
    sosv --;
}
```

+ *Tạo danh sách học sinh.*

```
void Taods()             // Tạo danh sách
{
    int next = 1;
    while (next) {
        Bosung();
        cout << "Next (0/1) ? " ; cin >> next ;
    }
}
```

+ *In danh sách học sinh.*

```
void Inds()              // In danh sách
{
```

```
        cur = dau;    int i=1;
        while (cur != NULL)
        {
            cout << "\nHoc sinh thu " << i << " --------------\n") ;
            cout << "Hoten:" << cur->hoten ;
            cout << "Ngay sinh: " cout << cur -> ns.day << "/" ;
            cout << cur -> ns.month << "/" ;
            cout << cur -> ns.year ;
            cout << "Diem: " << cur->diem ;
            cur = cur->next; i++;
        }
    }
```

+ *Hàm chính.*

```
void main()
{
    Taods();
    Inds();
    getch();
}
```

## V.3. Kiểu hợp

### V.3.1. Khai báo

Giống như cấu trúc, kiểu hợp cũng có nhiều thành phần nhưng các thành phần của chúng sử dụng chung nhau một vùng nhớ. Do vậy kích thước của một kiểu hợp là độ dài của trường lớn nhất và việc thay đổi một thành phần sẽ ảnh hưởng đến tất cả các thành phần còn lại.

```
union <tên kiểu> {
    Danh sách các thành phần;
};
```

### V.3.2. Truy cập

Cú pháp truy cập đến các thành phần của hợp cũng tương tự như kiểu cấu trúc, tức cũng sử dụng toán tử lấy thành phần (dấu chấm . hoặc -> cho biến con trỏ kiểu hợp).

Dưới đây là một ví dụ minh hoạ việc sử dụng khai báo kiểu hợp để tách byte thấp, byte cao của một số nguyên.

*Ví dụ 5.20* :

```
void main()
{
    union songuyen {
        int n;
        unsigned char c[2];
    } x;
    cout << "Nhập số nguyên: " ; cin >> x.n ;
    cout << "Byte thấp của x = " << x.c[0] << endl ;
    cout << "Byte cao của x = " << x.c[1] << endl;
}
```

*Ví dụ 5.21*: Kết hợp cùng kiểu nhóm bit trong cấu trúc, chúng ta có thể tìm được các bit của một số như chương trình sau. Trong chương trình ta sử dụng một biến u có kiểu hợp. Trong kiểu hợp này có 2 thành phần là 2 cấu trúc lần lượt có tên s và f.

```
        union {
            struct { unsigned a, b ; } s;
            struct {
```

```
            unsigned n1: 1;
            unsigned: 15;
            unsigned n2: 1;
            unsigned: 7;
            unsigned n3: 8;
        } t ;
    } u;
```

với khai báo trên đây khi nhập u.s thì nó cũng ảnh hưởng đến u.t, cụ thể:

- u.t.n1 là bit đầu tiên (0) của thành phần u.s.a
- u.t.n2 là bit 0 của thành phần u.s.b
- u.t.n3 là byte cao của u.s.b

## V.4. Kiểu liệt kê

Có thể gán các giá trị nguyên liên tiếp (tính từ 0) cho các tên gọi cụ thể bằng kiểu liệt kê theo khai báo sau đây:

**`enum tên_kiểu { d/s tên các giá trị };`**

*Ví dụ 5.22*:

```
enum bool {false, true};
```

khai báo kiểu mới đặt tên Bool chỉ nhận 1 trong 2 giá trị đặt tên false và true, trong đó false ứng với giá trị 0 và true ứng với giá trị 1. Cách khai báo kiểu enum trên cũng tương đương với dãy các macro sau:

```
#define   false      0
#define   true       1
```

Với kiểu bool ta có thể khai báo một số biến như sau:

```
bool ok, found;
```

hai biến ok và found sẽ chỉ nhận 1 trong 2 giá trị false (thay cho 0) hoặc true (thay cho 1). Có nghĩa có thể gán:

```
ok = true;
```

hoặc:

```
found = false;
```

Tuy nhiên không thể gán các giá trị nguyên trực tiếp cho các biến enum mà phải thông qua ép kiểu. Ví dụ:

```
ok = 0;           // sai
ok = bool(0) ;    // đúng
ok = false ;      // đúng
```

## V.5. Câu hỏi và Bài tập

### V.5.1. Câu hỏi

1. Có thể truy nhập thành phần của cấu trúc thông qua con trỏ như sau (với p là con trỏ cấu trúc và a là thành phần của cấu trúc):

   A: (*p).a  B: *p→a  C: a và b sai  D: a và b đúng

2. Cho khai báo struct T {int x; float y;} t, *p, a[10]; Câu lệnh nào trong các câu sau không hợp lệ:

   (1) p = &t;  (2) p = &t.x;  (3) p = a; (4) p = &a (5) p = &a[5]; (6) p = &a[5].y;

   A: 1, 2 và 3  B: 4, 5 và 6  C: 1, 3 và 5  D: 2, 4 và 6

3. Cho các khai báo sau:

```
struct ngay {int ng, th, nam;} vaotruong, ratruong;
typedef struct {char hoten[25]; ngay ngaysinh;} sinhvien;
```

Hãy chọn câu đúng nhất:

A. Không được phép gán: ratruong = vaotruong;

B. sinhvien là tên cấu trúc, vaotruong, ratruong là biến cấu trúc

C. Có thể viết: vaotruong.ng, ratruong.th, sinhvien.vaotruong.nam để truy nhập đến các thành phần tương ứng.

D. A, B, C đều đúng.

4. Trong các khởi tạo giá trị cho các cấu trúc sau, khởi tạo nào đúng:

```
struct S1 {
    int ngay, thang, nam;
} s1 = {2,3};
struct S2 {
    char hoten[10];
    struct S1 ngaysinh;
} s2 = {"Ly Ly",1,2,3};
struct S3 {
    struct S2 sinhvien;
    float diem;
} s3 = {{{"Cốc cốc", {4,5,6}}, 7};
```

A. S1 và S2 đúng ; B. S2 và S3 đúng ; C. S3 và S1 đúng ; D. Cả 3 cùng đúng ;

5. Đối với kiểu cấu trúc, cách gán nào dưới đây là không được phép:

A. Gán hai biến cho nhau.

B. Gán hai phần tử mảng (kiểu cấu trúc) cho nhau

C. Gán một phần tử mảng (kiểu cấu trúc) cho một biến và ngược lại

D. Gán hai mảng cấu trúc cùng số phần tử cho nhau

6. Cho đoạn chương trình sau:

```
struct {
    int to  ;
    float soluong;
} x[10];
for (int i = 0; i < 10; i++) cin >> x[i].to >> x[i].soluong ;
```

Chọn câu đúng nhất trong các câu sau:

A. Đoạn chương trình trên có lỗi cú pháp

B. Không được phép sử dụng toán tử lấy địa chỉ đối với các thành phần to và soluong

C. Lấy địa chỉ thành phần soluong dẫn đến chương trình hoạt động không đúng đắn

D. Cả A, B, C đều sai

7. Chọn câu đúng nhất trong các câu sau:

A. Các thành phần của kiểu hợp (union) được cấp phát một vùng nhớ chung

B. Kích thước của kiểu hợp bằng kích thước của thành phần lớn nhất

C. Một biến kiểu hợp có thể được tổ chức để cho phép thay đổi được kiểu dữ liệu của biến trong qua trình chạy chương trình

D. A, B, C đều đúng

8. Cho khai báo:

```
union {
    unsigned x;
    unsigned char y[2];
} z = {0xabcd};
```

Chọn câu đúng nhất trong các câu sau:

A. Khai báo trên là sai vì thiếu tên kiểu

B. Khởi tạo biến z là sai vì chỉ có một giá trị (0xabcd)

C. z.y[0] = 0xab

D. z.y[1] = 0xab

9. Cho kiểu hợp:

```
union U {
    char x[1];
    int y[2]; float z[3];
} u;
```

Chọn câu đúng nhất trong các câu sau:

A. sizeof(U) = 1+2+3 = 6

B. sizeof(U) = max(sizeof(char), sizeof(int), sizeof(float))

C. sizeof(u) = max(sizeof(u.x), sizeof(u.y), sizeof(u.z))

D. B và C đúng

10. Cho khai báo:

```
union {
    unsigned x;
    struct {
    unsigned char a, b;
} y;
} z = {0xabcd};
```

Giá trị của z.y.a và z.y.b tương ứng:

A. 0xab, 0xcd B. 0xcd, 0xab C. 0xabcd, 0 D. 0, 0xabcd

11. Cho khai báo:

```
union {
    struct {
        unsigned char a, b;
    } y;
    unsigned x;
} z = {{1,2}};
```

Giá trị của z.x bằng:

A. 513 B. 258

C. Không xác định vì khởi tạo sai D. Khởi tạo đúng nhưng z.x chưa có giá trị

12. Xét đoạn lệnh:

```
union U {
    int x; char y;
} u;
u.x = 0; u.y = 200;
```

Tìm giá trị của u.x + u.y ?

A. 122 B. 144 C. 200 D. 400

13. Để tạo danh sách liên kết, theo bạn khai báo nào dưới đây đúng cấu trúc tự trỏ sẽ được dùng:

1. *struct HS{char ht[25]; int tuoi; struct HS *next;};*
2. *typedef struct HS node; struct SV {char ht[25]; int tuoi; node *next;};*
3. *typedef struct HS {char ht[25]; int tuoi; struct HS *next;} node;*

A. 1 B. 2 C. 2 và 3 D. 1, 2 và 3

### V.5.2. Bài tập

1. Cho số phức dưới dạng cấu trúc gồm 2 thành phần là thực và ảo. Viết chương trình nhập 2 số phức và in ra tổng, tích, hiệu, thương của chúng.
2. Cho phân số dưới dạng cấu trúc gồm 2 thành phần là tử và mẫu. Viết chương trình nhập 2 phân số, in ra tổng, tích, hiệu, thương của chúng dưới dạng tối giản.
3. Tính số ngày đã qua kể từ đầu năm cho đến ngày hiện tại. Qui ước ngày được khai báo dưới dạng cấu trúc và để đơn giản một năm bất kỳ được tính 365 ngày và tháng bất kỳ có 30 ngày.
4. Nhập một ngày tháng năm dưới dạng cấu trúc. Tính chính xác (kể cả năm nhuận) số ngày đã qua kể từ ngày 1/1/1 cho đến ngày đó.
5. Tính khoảng cách giữa 2 ngày tháng bất kỳ.
6. Hiện thứ của một ngày bất kỳ nào đó, biết rằng ngày 1/1/1 là thứ hai.
7. Hiện thứ của một ngày bất kỳ nào đó, lấy ngày thứ hiện tại để làm chuẩn.
8. Viết chương trình nhập một mảng học sinh, thông tin về mỗi học sinh gồm họ tên và ngày sinh (kiểu cấu trúc). Sắp xếp mảng theo tuổi và in ra màn hình
9. Lập danh sách liên kết chứa bảng chữ cái A, B, C … Hãy đảo phần đầu từ A .. M xuống cuối thành N, O, … Z, A, …M.
10. Viết chương trình tìm người cuối cùng trong trò chơi: 30 người xếp vòng tròn. Đếm vòng tròn (bắt đầu từ người số 1) cứ đến người thứ 7 thì người này bị loại ra khỏi vòng. Hỏi người còn lại cuối cùng ?
11. Giả sử có danh sách liên kết mà mỗi nốt của nó lưu một giá trị nguyên. Viết chương trình sắp xếp danh sách theo thứ tự giảm dần.
12. Giả sử có danh sách liên kết mà mỗi nốt của nó lưu một giá trị nguyên được sắp giảm dần. Viết chương trình cho phép chèn thêm một phần tử vào danh sách sao cho danh sách vẫn được sắp giảm dần.
13. Tạo danh sách liên kết các số thực $x_1, x_2, ..., xn$. Gọi m là trung bình cộng: $m = \frac{x_1 + x_2 + ... + x_n}{n}$

    Hãy in lần lượt ra màn hình các giá trị: m, $x_1$ - m, $x_2$ - m, ..., xn - m.
14. Sử dụng kiểu union để in ra byte thấp, byte cao của một số nguyên.

# CHƯƠNG VI. KIỂU DỮ LIỆU TỆP TIN

## VI.1. Thao tác với tệp bằng thư viện C++

### VI.1.1. Kiểu dữ liệu tệp tin

Để làm việc với file, bạn cần khai báo include tệp tin tiêu đề <fstream> ở đầu chương trình.

Làm việc với một tệp tin trên đĩa cũng được quan niệm như làm việc với các thiết bị khác của máy tính (như máy in, màn hình, bàn phím). Các đối tượng làm việc với tệp thuộc lớp ifstream hoặc ofstream hoặc fstream tùy thuộc việc ta muốn sử dụng tệp để đọc hay ghi hoặc cả hai.

Như vậy, để sử dụng một tệp dữ liệu, đầu tiên chúng ta cần khai báo biến tệp và gắn với tệp dữ liệu. Khi khai báo, ta có thể sử dụng lớp fstream hoặc hai lớp đặc biệt của ftream là ifstream và ofstream. Biến tệp sẽ được gắn với tệp cụ thể trên đĩa ngay trong quá trình khởi tạo (khởi tạo đối tượng với tham số là tên tệp) hoặc cũng có thể sau này mới được gắn với tên tệp bằng câu lệnh mở tệp. Sau khi đã gắn một biến tệp với một tệp trên đĩa, bạn có thể sử dụng biến như đối với bàn phím, màn hình. Điều này có nghĩa trong các câu lệnh in ra màn hình chỉ cần thay từ khóa *cout* bởi tên tệp, mọi dữ liệu cần in trong câu lệnh sẽ được ghi lên tệp mà biến tệp đang đại diện. Cũng tương tự nếu thay *cin* bởi tên biến tệp, dữ liệu sẽ được đọc vào từ tệp vào thay vì đọc từ bàn phím. Để tạo đối tượng:

- Chỉ để ghi dữ liệu ra tệp, ta khai báo biến tệp với lớp *ofstream*;
- Chỉ đề dùng đọc dữ liệu từ tệp, ta khai báo biến tệp với lớp *ifstream*;
- Dùng cho cả việc ghi và đọc dữ liệu với tệp, ta khai báo biến tệp với lớp *fstream*;

### VI.1.2. Khai báo biến làm việc với tệp tin

Mỗi lớp *ifstream, ofstream* và *ftream* cung cấp một số phương thức để làm việc với tệp. Ở đây chúng tôi chỉ trình bày 2 cách (2 phương thức) hay dùng.

***a. Cách khai báo 1:***

```
<Class> <tên_biến_file>;
<tên_biến_file>.open(<tên_file>, [chế_độ_mở]);
```

<Class> là một trong ba lớp *ifstream, ofstream hoặc fstream*. Biến <*tên_biến_file*> là tên do NSD tự đặt theo quy định đặt tên của C++. *Chế độ mở* là cách thức làm việc với tệp (xem ở dưới). Cách này cho phép tạo trước một đối tượng chưa gắn với tệp cụ thể nào. Sau đó, cần dùng tiếp phương thức *open()* để đồng thời mở tệp và gắn tệp với <*tên_biến_file*> vừa tạo.

*Ví dụ 6.1:*

```
ifstream f;           // tạo đối tượng có tên f để đọc hoặc
ofstream f;           // tạo đối tượng có tên f để ghi
f.open("Baitap");     // mở file Baitap và gắn với f
```

***b. Cách khai báo 2:***

```
<Class> <object>(tên_file, chế_độ_mở);
```

Cách này cho phép đồng thời mở file cụ thể và gắn file với tên đối tượng trong câu lệnh.

*Ví dụ 6.2:*

```
ifstream fi("bt.inp");   // mở file bt.inp gắn với đối tượng fi để đọc dữ liệu
```

```
ofstream fo("bt.out");   // mở file bt.out gắn với đối tượng fo để ghi dữ liệu
fstream f("bt.dat"); // mở file bt.dat gắn với đối tượng f để ghi/đọc dữ liệu
```

Sau khi mở tệp và gắn với biến tệp, mọi thao tác trên biến tệp cũng chính là làm việc với tệp dữ liệu đó.

Trong các câu lệnh trên có các *chế độ mở* để qui định cách thức làm việc của tệp. Gồm có:

| | |
|---|---|
| `ios::binary` | Chế độ mở file kiểu nhị phân. Ngầm định là kiểu văn bản. |
| `ios::in` | Chế độ mở file để đọc (ngầm định với đối tượng trong ifstream). |
| `ios::out` | Chế độ mở file để ghi (ngầm định với đối tượng trong ofstream), nếu file đã có trên đĩa thì nội dung của nó sẽ bị ghi đè (bị xóa). |
| `ios::trunc` | Chế độ mở file và xóa nội dung file đã có |
| `ios::ate` | Chế độ mở file và di chuyển chuyển con trỏ đến cuối file |
| `ios::app` | Chế độ mở file để ghi bổ sung thêm dữ liệu vào cuối file |

Có thể chỉ định nhiều chế độ mở cùng lúc bằng cách ghi chúng liên tiếp nhau với toán tử bitwise |. Ví dụ để mở file baitap.dat như một file nhị phân và ghi tiếp theo vào cuối file:

```
ofstream f("baitap.dat", ios::binary | ios::app);
```

### VI.1.3. Đóng tệp và giải phóng biến tệp

Để đóng tệp được đại diện bởi *biến_tệp* , sử dụng phương thức *close()* như sau:

**`<biến_tệp>.close();`**

Sau khi đóng tệp, đồng thời bạn cũng đã giải phóng mối liên kết giữa *biến_tệp* và tệp đó. Bạn có thể dùng *biến_tệp* để gắn và làm việc với tệp khác bằng phương thức *open()* như đã nói trên.

*Ví dụ 6.3*: Đọc một dãy số từ bàn phím và ghi lên tệp DULIEU.DAT. Tệp được xem như tệp văn bản (ngầm định), các số được ghi cách nhau 1 dấu cách.

```
#include <iostream>
#include <fstream>
using namespace std;
ofstream f;   // khai báo (tạo) biến f
int main()
{
    int x;
    f.open("DULIEU.DAT");
    for (int i = 1; i<=10; i++)
    {
        cin >> x;
        f << x << ' ';
    }
    f.close();
}
```

*Ví dụ 6.4*: Chương trình sau nhập danh sách học sinh, ghi vào file *dulieu.inp*, đọc ra mảng, sắp xếp theo tuổi và ghi ra file *ketqua.out*. Dòng đầu tiên trong file ghi số học sinh, các dòng tiếp theo ghi thông tin của học sinh gồm họ tên (độ rộng 25 kí tự), tuổi (độ rộng 4 kí tự) và điểm (độ rộng 8 kí tự).

```
#include <iostream>
#include <iomanip>
#include <fstream>
```

```
#include <cstdlib>
using namespace std;
ifstream fi;  // khai báo biến fi để sau này gắn với tệp input
ofstream fo;  // khai báo biến fo để sau này gắn với tệp output

struct Hocsinh{
    char ht[25];
    int tuoi;
    double dtb;
} ;

Hocsinh h[40];
int sl;
void nhaphs()
{
    cin >> sl;
    for(int i = 1 ; i <= sl ; i++)
    {
       cout<<"Hoc sinh thu: "<<i<<endl;
       cin.ignore();
       cout<<"Ho ten: "; cin.getline(h[i].ht,25);
       cout<<"Tuoi: "; cin >> h[i].tuoi;
       cout<<"DTB: "; cin >> h[i].dtb;
    }
}

void ghihs(char fname[])
{
    fo.open(fname);
    fo << setw(4) << sl <<endl;
    for(int i = 1 ; i <= sl ; i++)
    {
       fo << setw(25) << h[i].ht;
       fo << setw(4) << h[i].tuoi;
       fo << setw(8) << h[i].dtb<<'\n';
    }
    fo.close();
}

void dochs(char fname[])
{
    fi.open(fname);
    fi >> sl;  fi.ignore();
    for(int i = 1 ; i <= sl ; i++)
    {
       fi.get(h[i].ht,27);
       char tmp1[10], tmp2[10];
       fi.get(tmp1,5); h[i].tuoi = atoi(tmp1);
       fi.get(tmp2,9); h[i].dtb = atof(tmp2);
       fi.ignore();
    }
    fi.close();
}

void sapxep()
{
    for(int i = 1 ; i < sl ; i++)
        for (int j = i+1; j <= sl ; j++)
            if (h[i].tuoi > h[j].tuoi)
```

```
                {
                    Hocsinh t = h[i];
                    h[i] = h[j];
                    h[j] = t;
                }
}

int main() {
    nhaphs();
    ghihs("hocsinh.inp");
    dochs("hocsinh.inp");
    sapxep();
    ghihs("hocsinh.out");
    return 0;
}
```

### VI.1.4. Kiểm tra sự tồn tại của tệp, kiểm tra đã đến cuối tệp chưa

Việc mở một tệp chưa có để đọc dữ liệu, sẽ gây nên lỗi và làm dừng chương trình. Khi xảy ra lỗi mở file, giá trị trả lại của phương thức *bad()* là một số khác 0. Do vậy có thể sử dụng phương thức này để kiểm tra một file đã có trên đĩa hay chưa.

*Ví dụ 6.5:*

```
ifstream f("Bai tap");
if (f.bad()) {
    cout << "file Baitap chưa có";
    exit(1);
}
```

Khi đọc hoặc ghi, con trỏ tệp sẽ chuyển dần về cuối tệp. Khi con trỏ ở cuối tệp, phương thức *eof()* sẽ trả lại giá trị khác không. Có thể sử dụng phương thức này để kiểm tra đã hết tệp hay chưa.

Chương trình sau cho phép tính độ dài của file Baitap. File được mở theo kiểu nhị phân.

*Ví dụ 6.6*:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void main()
{
    long dodai = 0;
    char ch;
    ifstream f("Baitap", ios::in | ios::binary) ;
    if (f.bad()) {
        cout << "File Baitap không có";
        exit(1);
    }
    while (!f.eof()) {
        f.get(ch));
        dodai++;
    }
    cout << "Độ dài của file = " << dodai;
}
```

### VI.1.5. Đọc, ghi đồng thời trên tệp

Để đọc ghi đồng thời, tệp phải được gắn với biến tệp của lớp *fstream*. Khi đó chế độ mở phải bao gồm chỉ định *ios::in | ios::out*. Ví dụ:

```
fstream f("Data.dat", ios::in | ios::out) ;
```

hoặc

```
fstream f ;
f.open("Data.dat", ios::in | ios::out) ;
```

### VI.1.6. Di chuyển con trỏ file

Các phương thức sau cho phép làm việc trên đối tượng của dòng xuất (ofstream).

- **<biến_tệp>.seekp(n):** Di chuyển con trỏ đến byte thứ n (các byte được tính từ 0)
- **< biến_tệp>.seekp(n, postion):** Dịch chuyển con trỏ tệp đi *n* byte (có thể âm hoặc dương) từ vị trí `postion`. Vị trí `postion` gồm các giá trị:
  - `ios::beg` : từ đầu file
  - `ios::end`: từ cuối file
  - `ios::cur`: từ vị trí hiện tại của con trỏ.
- **<biến_tệp>.tellp(n):** Cho biết vị trí hiện tại của con trỏ.

Để làm việc với dòng nhập tên các phương thức trên được thay tương ứng bởi các tên : seekg và tellg. Đối với các dòng nhập lẫn xuất có thể sử dụng được cả 6 phương thức trên.

Ví dụ sau tính độ dài tệp, đơn giản hơn ví dụ ở trên.

```
fstream f("Baitap");
f.seekg(0, ios::end);
cout << "Độ dài bằng = " << f.tellg();
```

*Ví dụ 6.7* : Chương trình nhập và in danh sách học sinh trên ghi/đọc đồng thời.

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
#include <cstring>
#include <cctype>

using namespace std;
fstream f;
char fname[10];

void input_data()
{
    int stt = 0, tuoi;
    char hoten[26];
    double dtb;
    f << setprecision(2) << setiosflags(ios::showpoint) ;
    while (1) {
       stt++;
       cout << "\nNhap hoc sinh thu " << stt ;
       cout << "\nHo ten: ";
       cin.ignore();
       cin.getline(hoten, 25,'\n');
       if (strlen(hoten)== 0) break;
       cout << "\nTuoi: "; cin >> tuoi;
       cout << "\nDiem: "; cin >> dtb;
       f << setw(25) << hoten << setw(4);
```

```
            f << tuoi << setw(8) << dtb<<endl ;
        }
    }

    void output_data()
    {
        int stt = 0, tuoi;
        char hoten[100];
        double dtb;
        f.seekg(0) ;            // quay về đầu tep
        while (! f.eof()) {
            f.get(hoten, 26);
            if(strlen(hoten)==0) break;
            char tmp[8];
            f.get(tmp, 6);
            tuoi = atoi(tmp);
            f.get(tmp, 9);
            dtb = atof(tmp);
            f.ignore();
            stt++;
            cout<< stt << " : " << setw(25) << hoten << setw(5) << tuoi;
            cout<< setw(8)<< dtb << endl;
        }
    }

    int main() {
        cout << "Input file name: ";
        cin.getline(fname,10);
        if (strlen(fname)==0) return  0;
        f.open(fname, ios::in | ios::out | ios::ate);
        if (! f.bad())       //Nếu mở file thành công
        {
            cout << "File exist. Overwrite (Y/N)?" ; cin.ignore();
            char traloi;
            cin.get(traloi) ;
            if (toupper(traloi) == 'Y') {
               f.close(); //Đóng tệp mở ở mode ghi mới
               f.open(fname, ios::in | ios::out | ios::trunc);
               input_data();
            }
            output_data();
            f.close();
        }
        else {
             cout<<"Can't open file "<<fname;
        }
        return 0;
    }
```

### VI.1.7. Tệp văn bản và tệp nhị phân

#### *a. Khái niệm file văn bản và file nhị phân*

*+ File văn bản:*

Trong file văn bản mỗi byte được xem là một kí tự. Tuy nhiên nếu 2 byte 10 (LF), 13 (CR) đi liền nhau thì được xem là một kí tự và nó là kí tự xuống dòng. Như vậy file văn bản là một tập hợp các dòng kí tự với kí tự xuống dòng có mã là 10. Kí tự có mã 26 được xem là kí tự kết thúc file.

*+ File nhị phân:*

Thông tin lưu trong file nhị phân được xem như dãy byte bình thường. Mã kết thúc file được chọn là -1, được định nghĩa là EOF trong <cstdio>. Các thao tác trên file nhị phân thường đọc ghi từng byte một, không quan tâm ý nghĩa của byte.

Một số các thao tác nhập/xuất sẽ có hiệu quả khác nhau khi mở file dưới các dạng khác nhau.

Ví dụ, ký tự ch = 10, khi đó f << ch sẽ ghi 2 byte 10,13 lên file văn bản f, trong khi đó lệnh này chỉ khi 1 byte 10 lên file nhị phân.

Ngược lại, nếu f là file văn bản thì f.getc(ch) sẽ trả về chỉ 1 byte 10 khi đọc được 2 byte 10, 13 liên tiếp nhau.

Một file luôn ngầm định dưới dạng văn bản, do vậy để chỉ định file là nhị phân ta cần sử dụng cờ *ios::binary*.

***b. Đọc và ghi kí tự***

```
put(c);           // ghi kí tự ra file
get(c);           // đọc kí tự từ file
```

*Ví dụ 6.8*: Sao chép file dulieu1.dat sang file dulieu2.dat. Cần sao chép và ghi từng byte một do vậy để chính xác ta sẽ mở các file dưới dạng nhị phân.

```
#include <fstream>
#include <iostream>

using namespace std;
fstream fi, fo;
int main()
{
    fi.open("dulieu1.dat", ios::in | ios::binary);
    fo.open("dulieu2.dat", ios::out | ios::binary);
    char ch;
    while (!fi.eof()) {
        fi.get(ch);
       fo.put(ch);
    }
    fi.close();
    fo.close();
    return 0;
}
```

***c. Đọc và ghi dãy kí tự***

```
write(char *buf, int n);     // ghi n kí tự trong buf ra dòng xuất
read(char *buf, int n);      // nhập n kí tự từ buf vào dòng nhập
gcount();                    // cho biết số kí tự read đọc được
```

*Ví dụ 6.9* : Chương trình sao chép file ở trên có thể sử dụng các phương thức mới sau:

```
#include <fstream>
#include <iostream>

using namespace std;
fstream fi, fo;
int main()
{
    fi.open("dulieu1.dat", ios::in | ios::binary);
    fo.open("dulieu2.dat", ios::out| ios::binary);
```

```
        char buf[2000] ;
        int n = 2000;
        while (n) {
            fi.read(buf, 2000);
            n = fi.gcount();
            fo.write(buf, n);
        }
        fi.close();
        fo.close();
        return 0;
    }
```

## VI.2. Thao tác với tệp bằng thư viện C

### VI.2.1. Đóng, mở tệp, xóa vùng đệm và kiểm tra lỗi

#### *a. Khai báo biến kiểu tệp*

Để làm việc với một tệp, bạn cần khai báo một biến con trỏ kiểu FILE (nhớ viết in hoa). Chú ý, trong các chế độ đọc/ghi cần làm sạch vùng đệm trước khi chuyển từ đọc sang ghi hoặc ngược lại. Các hàm fflush và hàm di chuyển đầu từ có thể giúp việc này. Cần khai báo tệp tin tiêu đề <cstdio> khi sử dụng các hàm dưới đây

*Cú pháp*: `FILE * <bien_tep>;`

*Ví dụ*: `FILE *fi, *fo;`

#### *b. Hàm fopen mở tệp*

Mở tệp để thao tác với tệp. Nếu không thành công <bien_tep> ==NULL

*Cú pháp*: `<bien_tep> = fopen(<ten_tep>, <open_mode>);`

Trong đó, <*ten_tep*> là tên của file cần mở trên đĩa, <*open_mode*> là chế độ mở, gồm có các chế độ mở tệp như sau:

| Chế độ mở | Ý nghĩa |
|---|---|
| "r" "rt" | Mở tệp để đọc dữ liệu kiểu văn bản, sẽ báo lỗi nếu không có tệp |
| "w" "wt" | Mở tệp để ghi dữ liệu kiểu văn bản, sẽ ghi đè nếu đã có tệp này |
| "a" "at" | Mở tệp để ghi bổ sung dữ liệu kiểu văn bản, sẽ tạo mời nếu chưa có tệp |
| "rb" | Mở tệp để đọc dữ liệu kiểu nhị phân, sẽ báo lỗi nếu không có tệp |
| "wb" | Mở tệp để ghi dữ liệu kiểu nhị phân, sẽ ghi đè nếu đã có tệp này |
| "ab" | Mở tệp để ghi bổ sung dữ liệu kiểu nhị phân, sẽ tạo mới nếu đã có tệp |
| "r+" "r+t" | Mở tệp để đọc/ghi dữ liệu kiểu văn bản, sẽ báo lỗi nếu không có tệp |
| "w+" "w+t" | Mở tệp để đọc/ghi dữ liệu kiểu văn bản, sẽ tạo mới nếu không có tệp |
| "a+" "a+t" | Mở tệp để đọc/bổ sung dữ liệu kiểu văn bản, sẽ tạo mới nếu không có tệp |
| "r+b" | Mở tệp để đọc/ghi dữ liệu kiểu nhị phân, sẽ báo lỗi nếu không có tệp |
| "w+b" | Mở tệp để đọc/ghi dữ liệu kiểu nhị phân, sẽ tạo mới nếu không có tệp |
| "a+b" | Mở tệp để đọc/bổ sung dữ liệu kiểu nhị phân, sẽ tạo mới nếu không có tệp |

#### *c. Hàm fclose đóng tệp*

Đóng tệp, đẩy dữ liệu trong vùng đệm lên đĩa, xóa vùng đệm, giải phóng biến tệp (có thể dùng biến tệp cho tệp khác). Hàm trả giá trị 0 nếu thành công, trái lại hàm trả giá trị EOF.

*Cú pháp*: `fclose(<ten_bien>);`

***c. Hàm fflush làm sạch vùng đệm***

Xóa sạch vùng đệm. Hàm trả giá trị 0 nếu thành công, trái lại hàm trả giá trị EOF.

*Cú pháp*: `fflush(<ten_bien>);`

***d. Hàm ferror thông báo lỗi***

Cho biết có lỗi khi thao tác với tệp. Hàm trả giá trị 0 nếu không có lỗi, trái lại hàm trả giá trị khác 0.

*Cú pháp*: `ferror(<ten_bien>);`

***e. Hàm perror thông báo lỗi***

In ra chuỗi thông báo lỗi.

*Cú pháp*: `perror(<ten_chuoiC>);`

***f. Hàm feof kiểm tra cuối tệp***

Kiểm tra đã đến cuối tệp chưa. Hàm trả giá trị khác 0 nếu ở con trỏ tệp đã ở cuối tệp, trái lại hàm trả giá trị 0.

*Cú pháp*: `feof(<ten_bien>);`

***g. Hàm remove xóa tệp***

Xóa tệp <*ten_tep*> trên đĩa. Hàm trả giá trị khác 0 nếu thành công, trái lại hàm trả giá trị EOF.

*Cú pháp*: `remove(<ten_tep>);`

*Ví dụ 6.10*: Chương trình sau mở tệp và kiểm tra lỗi.

```
#include <stdio.h>
int main ()
{
    FILE * pFile;
    pFile=fopen("dulieu.dat","r");
    if (pFile==NULL) perror ("Error opening file");
    else {
        fputc ('x',pFile);
        if (ferror (pFile))
        printf ("Error writing to dulieu.dat\n");
        fclose (pFile);
  }
  return 0;
}
```

### VI.2.2. Xuất nhập ký tự với tệp

Các hàm đọc và ghi ký tự này dùng được cả trong kiểu nhị phân và văn bản, nhưng có tác dụng khác nhau.

***a. Các hàm putc và fputc***

Cả hai hàm đều ghi lên tệp ký tự có mã là *m=ch % 256*. Trong kiểu tệp văn bản, nếu m = 10 thì hàm sẽ ghi lên tệp hai mã 13 và 10.

*Cú pháp*: **putc(ch, <bien_tep>);** // ch là biến nguyên

**fputc(ch, <bien_tep>);** // ch là biến nguyên

*Ví dụ*: putc(-1,fp) ghi lên tệp trỏ bởi con trỏ fp ký tự có mã 255, vì dạng không dấu của -1 là 655535.

***b. Các hàm getc và fgetc***

Cả hai hàm đều đọc một ký tự từ tệp. Nếu thành công sẽ trả về giá trị nguyên (trong khoảng 0..255) đọc được. Nếu thất bại, trả về EOF. Trong kiểu tệp văn bản, hàm đọc liền mã 13 và 10 (nếu gặp) và trả về giá trị 10. Nếu gặp mã 26 (cuối tệp) thì trả về EOF.

*Cú pháp*: **getc(<bien_tep>);**

**fgetc(<bien_tep>);**

*Ví dụ*: putc(-1,fp) ghi lên tệp trỏ bởi con trỏ fp ký tự có mã 255, vì dạng không dấu của -1 là 655535.

***c. Ví dụ minh họa***

*Ví dụ 6.11*: Sao chép tệp ở chế độ nhị phân bằng *fgetc* và *fputc*.

```
#include <stdio.h>
int main ()
{
  FILE * fi, *fo;
  char tf1[14], tf2[14], c;
  printf("\nTep nguon: "); gets(tf1);
  printf("\nTep dich: "); gets(tf2);
  fi = fopen(tf1,"rb");
  if (fi==NULL)
  {
      printf("\nTep %s khong ton tai",tf1);
      return 0;
  }
  fo = fopen(tf2,"wb");
  while((c=fgetc(fi)!=EOF)) fputc(c,fo);
  fclose(fi); fclose(fo);
  return 0;
}
```

**VI.2.3. Xuất nhập văn bản với tệp**

***a. Hàm fprintf ghi dữ liệura tệp theo khuôn dạng***

Hàm ghi dữ liệu theo khuôn dạng ra tệp văn bản. Nếu thành công, hàm trả về số byte đã ghi ra tệp, ngược lại trả về EOF.

+ *Dạng hàm*: **int fprintf(<bien_tep>, [chuoi_dk], ...);**

*Ví dụ 6.12*: In ra tệp ba dòng văn bản

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

```
int main ()
{
    FILE * f;
    f = fopen("vidu.txt","wt");
    for(int i = 1; i <= 3 ; i++)
       fprintf(f,"Dong van ban thu %d\n",i);
    fclose(f);
    return 0;
}
```

***b. Hàm fscanf đọc dữ liệu từ tệp theo khuôn dạng***

Hàm đọc dữ liệu theo khuôn dạng từ tệp. Hàm trả về số trường dữ liệu đọc được.

+ *Dạng hàm*: **`int fscanf(<bien_tep>, [chuoi_dk], ...);`**

*Ví dụ 6.13*: Giả sử có tệp văn bản dulieu.txt, mô tả n điểm trong mặt phẳng, có cấu trúc:

- Dòng đầu chứa số nguyên n

- n dòng sau mỗi dòng chứa hai số nguyên x, y

Ta có thể đọc tệp dữ liệu như sau:

```
#include <stdio.h>
int main ()
{
    FILE * f;
    float x[100], y[100];
    int n;
    f = fopen("dulieu.txt","rt");
    if (f == NULL)
    {
        printf("Khong mo duoc tep dulieu.txt");
        return 0;
    }
    fscanf(f,"%d",&n);
    for(int i = 1; i <= n ; i++)
        fscanf(f,"%f%f",&x[i],&y[i]);
    for(int i = 1; i <= n ; i++)
        printf("%6.2f%6.2f\n",x[i],y[i]);
    fclose(f);
    return 0;
}
```

***c. Hàm fputs chuỗi ký tự ra tệp***

Hàm ghi chuỗi ký tự ra tệp văn bản. Nếu thành công, hàm trả về ký tự cuối đã ghi ra tệp, ngược lại trả về EOF.

+ *Dạng hàm*: **`int fputs(<chuoi_ky_tu>,<bien_tep>);`**

*Ví dụ 6.14*: Chương trình sau ghi chuỗi ký tự từ bàn phím ra tệp

```
#include <stdio.h>
int main ()
{
    FILE * f;
    int i = 0;
    char s[256];
    f = fopen("vanban.txt","wt");
    while(1){
```

```
            printf("\nDong %d (Bam Enter det ket thuc)",++i);
            gets(s);
            if (s[0]=='\0') break;
            if(i>1) fputc(10,f); //Ghi ky hieu xuong dong vao tep
            fputs(s,f);
        }
        fclose(f);
        return 0;
    }
```

***d. Hàm fgets đọc chuỗi ký tự từ tệp***

Hàm đọc chuỗi ký tự từ tệp văn bản vào biến chuỗi s. Việc đọc kết thúc khi:

- Hoặc đã đọc được n-1 ký tự.

- Hoặc đọc được ký tự hết dòng (khi đó mã 10 được đưa vào s)

- Hoặc kết thúc tệp

Xâu s sẽ được tự động thêm ký tự kết thúc chuỗi '\0'. Nếu thành công hàm trả về địa chỉ chuỗi s, ngược lại trả về NULL.

+ *Dạng hàm*: **`char *fgets(char *s,int n, <bien_tep>);`**

*Ví dụ 6.15*: Đọc các dòng văn bản trong tệp vanban.txt ra chuỗi và in ra màn hình

```
#include <stdio.h>
int main ()
{
    FILE * f;
    int i = 0;
    char s[256];
    f = fopen("vanban.txt","rt");
    while(!feof(f)){
        fgets(s,256,f);
        printf("Dong %d : %s",++i,s);
    }
    fclose(f);
    return 0;
}
```

## VI.3. Bài tập

1. Viết chương trình đếm số dòng của một file văn bản.
2. Viết chương trình đọc in từng kí tự của file văn bản ra màn hình, mỗi màn hình 20 dòng.
3. Viết chương trình tìm xâu dài nhất trong một file văn bản.
4. Viết chương trình ghép một file văn bản thứ hai vào file văn bản thứ nhất, trong đó tất cả chữ cái của file văn bản thứ nhất phải đổi thành chữ in hoa.
5. Viết chương trình in nội dung file ra màn hình và cho biết tổng số chữ cái, tổng số chữ số đã xuất hiện trong file.
6. Cho 2 file số thực (đã được sắp tăng dần). In ra màn hình dãy số xếp tăng dần của cả 2 file.
7. Viết hàm nhập 10 số thực từ bàn phím vào file INPUT.DAT. Viết hàm đọc các số thực từ file trên và in tổng bình phương của chúng ra màn hình.

8. Viết hàm nhập 10 số nguyên từ bàn phím vào file văn bản tên INPUT.DAT. Viết hàm đọc các số nguyên từ file trên và ghi những số chẵn vào file EVEN.DAT còn các số lẻ vào file ODD.DAT.
9. Nhập bằng chương trình 2 ma trận số nguyên vào 2 file văn bản. Hãy tạo file văn bản thứ 3 chứa nội dung của ma trận tích của 2 ma trận trên.
10. Tổ chức quản lý file học sinh (Họ tên, ngày sinh, giới tính, ĐTB) với các chức năng: Nhập, xem, xóa, sửa, xếp loại theo quy chế của BGD&ĐT.
11. Thông tin về một nhân viên trong cơ quan bao gồm : họ và tên, nghề nghiệp, số điện thoại, địa chỉ nhà riêng. Viết hàm nhập từ bàn phím thông tin của 7 nhân viên và ghi vào file INPUT.DAT. Viết hàm tìm trong file INPUT.DAT và in ra thông tin của 1 nhân viên theo số điện thoại được nhập từ bàn phím.

# CHƯƠNG VII. THƯ VIỆN STL CỦA C++

Trong các ngôn ngữ lập trình hiện nay, C++ được đánh giá là ngôn ngữ mạnh vì tính mềm dẻo, gần gũi với ngôn ngữ máy. Ngoài ra, với khả năng lập trình theo mẫu (template), C++ đã khiến ngôn ngữ lập trình trở thành khái quát, người sử dụng (NSD) không cần phải quan tâm nhiều đến các vấn đề cụ thể và chi tiết như quản lý bộ nhớ, sự tương thích kiểu dữ liệu... Sức mạnh của C++ đến từ STL, viết tắt của Standard Template Library - một thư viện mẫu cho C++ với những cấu trúc dữ liệu cũng như giải thuật được xây dựng tổng quát mà vẫn tận dụng được hiệu năng và tốc độ của C truyền thống. Với khả năng lập trình theo mẫu, những người lập trình đã đề ra khái niệm lập trình khái lược (generic programming).

Thư viện STL thực hiện toàn bộ các công việc vào ra dữ liệu (iostream, ...), quản lý mảng (vector, ...), thực hiện hầu hết các tính năng của các cấu trúc dữ liệu cơ bản (stack, queue, map, set...). Ngoài ra, STL còn chứa các thuật toán cơ bản: tìm min/max, tính tổng, sắp xếp (với nhiều thuật toán khác nhau), thay thế các phần tử, tìm kiếm (tìm kiếm tuyến tính, tìm kiếm nhị phân), trộn. Toàn bộ các tính năng nêu trên đều được cung cấp dưới dạng mẫu nên việc lập trình luôn thể hiện tính khái quát hóa cao. Nhờ có STL mà chương trình viết bằng ngôn ngữ C++ trở nên dễ đọc, dễ viết hơn nhiều.

Để sử dụng STL, bạn cần khai báo từ khóa *"using namespace std;"* sau các khai báo thư viện ("#include", hay "#define", ...)

*Ví dụ:*

```
#include <iostream>
#include <stack>        //khai báo sử dụng container stack
#define n 100
using namespace std;    //khai báo sử dụng STL
int main()
{
    ....
}
```

Việc sử dụng các hàm trong STL tương tự như việc sử dụng các hàm khác của C++.

## VII.1. ĐỐI TƯỢNG ITERATOR (BIẾN LẶP)

Trong C++, *iterator* là một đối tượng bất kì, trỏ tới một phần tử nào đó trong một dãy các phần tử (như mảng hoặc container), nó có khả năng duyệt qua các phần tử trong phạm vi bằng cách sử dụng một tập các toán tử như tự tăng/giảm (++/--) hoặc tham chiếu *.

Dạng rõ ràng nhất của iterator là một con trỏ: Một con trỏ có thể trỏ tới các phần tử trong mảng, và có thể lặp qua các phần tử của mảng bằng cách sử dụng toán tử tự tăng giảm (++, --). Tuy nhiên, còn có các dạng khác của iterator. Ví dụ: mỗi loại *container* (chẳng hạn như vector) có một loại iterator được thiết kế riêng để lặp với các phần tử của nó một cách hiệu quả.

- So sánh: == , != giữa 2 iterator.
- Gán: = giữa 2 iterator.
- Cộng, trừ: +, - với hằng số và tự tăng giảm ++ , — .

- Lấy giá trị: *

## VII.2. THƯ VIỆN CONTAINERS (LỚP THÙNG CHỨA)

Một đối tượng *container* là đối tượng có khả năng lưu trữ một tập các đối tượng khác (coi như các phần tử của nó). Nó được thực hiện như các lớp mẫu (class templates).

Container quản lý không gian lưu trữ cho các phần tử của nó và cung cấp các hàm thành viên (member function) để truy xuất, thao tác với các phần tử theo cách hoặc trực tiếp hoặc thông qua các biến lặp iterator.

Container gồm các cấu trúc thường sử dụng trong lập trình như: mảng động - dynamic arrays (vector), hàng đợi – queues (queue), hàng đợi ưu tiên – heaps (priority queue), danh sách kiên kết – linked list (list), cây – trees (set), mảng ánh xạ - associative arrays (map),...

Nhiều container chứa một số hàm thành viên giống nhau. Sử dụng loại container nào cho nhu cầu cụ thể nói chung không chỉ phụ thuộc vào các hàm được cung cấp mà còn phải dựa vào hiệu quả của các hàm thành viên của nó (độ phức tạp - bigO của các hàm). Điều này đặc biệt đúng với container dạng dãy (sequence containers), mà trong đó có sự khác nhau về độ phức tạp đối với các thao tác chèn/xóa phần tử hay truy cập vào phần tử.

### VII.2.1. Iterator

Tất cả các container loại Sequence container và Associative container đều hỗ trợ các iterator như sau (ví dụ với vector, những loại khác có chức năng cũng vậy).

```
vector <int> :: iterator it;  //khai báo iterator "it" it=vector.begin();
                              // trỏ đến vị trí phần tử đầu tiên của vector
it=vector.end();    // trỏ đến v.trí kết thúc (không phải p.tử cuối) của vector)
// khai báo iterator ngược "rit"
vector <int> :: reverse_iterator rit; rit = vector.rbegin();
// trỏ đến v.trí kết thúc của vector theo chiều ngược (không phải p.tử đầu tiên)
rit = vector.rend();
```

Tất cả các hàm iterator này đều là O(1).

### VII.2.2. Vector (Mảng động)

#### *a. Khai báo vector*

```
#include <vector>
```

*Ví dụ*:

```
/***** Vector 1 chiều *****/
vector <int> first;            //tạo vector rỗng kiểu dữ liệu int
vector <int> second (4,100);   //tạo vector với 4 phần tử là 100
// khởi tạo bằng cách lấy các giá trị từ đầu đến cuối vector second
vector <int> third (second.begin(),second.end())
// khởi tạo bằng cách lấy các giá trị từ vector third
vector <int> four (third)
/***** Vector 2 chiều *****/
vector < vector <int> > v;     // Tạo vector 2 chiều rỗng
vector < vector <int> > v(5, 10); //khai báo vector 5×10
```

```
vector < vector <int> > v(5) ;      // khai báo 5 vector 1 chiều rỗng
// khai báo vector 5*10 với các phần tử khởi trị là 1
vector < vector <int> > v (5, vector <int> (10,1) ) ;
```

Các bạn chú ý 2 dấu ngoặc ở dòng trên > > không được viết liền nhau.

Ví dụ như sau là **sai**:

```
// Khai báo vector 2 chiều SAI
vector <vector <int>> v;
```

***b. Các hàm thành viên***

| Nhóm | Tên hàm | Ý nghĩa | Độ phức tạp |
|---|---|---|---|
| Kích thước | **size** | trả về số lượng phần tử của vector. | O(1) |
| | **empty** | trả về true(1) nếu vector rỗng, ngược lại là false(0) | O(1) |
| Truy cập phần tử | **v[i]** | trả về giá trị phần tử thứ i của vector v[], không kiểm tra phạm vi | O(1) |
| | **at(i)** | trả về giá trị phần tử thứ i của vector v[], có kiểm tra phạm vi | O(1) |
| | **front** | trả về giá trị phần tử đầu tiên | O(1) |
| | **back** | trả về giá trị phần tử cuối cùng | O(1) |
| Thêm bớt phần tử | **push_back(x)** | thêm x vào cuối vector | O(1) |
| | **pop_back** | xóa bỏ phần tử ở cuối vector | O(1) |
| | **insert(iterator,x)** | chèn *x* vào trước vị trí *iterator* ( x có thể là phần tử hay iterator của 1 đoạn phần tử…) | O(số phần tử chèn thêm) |
| | **erase(iterator)** | xóa phần tử ở vị trí *iterator* | O(số phần tử bị xóa đi) |
| | **erase(it1,it2)** | xóa các phần tử từ vị trí iterator *it1* đến vị trí *it2* | O(số phần tử bị xóa đi) |
| | **swap** | đổi 2 vector cho nhau. ví dụ: *first.swap(second)* | O(n) |
| | **clear** | xóa vector | O(n) |

***c. Nhận xét***

- Nên sử dụng vector khi:
  - Truy cập cần truy cập ngẫu nhiên các phần tử. O(1)
  - Thêm hay xóa phần tử ở vị trí cuối cùng. O(1)
- Vector làm việc giống như một mảng động.

*Ví dụ 7.1*: Làm quen cách sử dụng các hàm.

```
#include <iostream>
```

```
#include <vector>
using namespace std;
vector <int> v;         //Khai báo vector
vector <int>::iterator it; //Khai báo iterator
vector <int>::reverse_iterator rit; //Khai báo iterator ngược
int i;
main() {
    for (i=1;i<=5;i++) v.push_back(i); // v={1,2,3,4,5}
    cout << v.front() << endl; // In ra 1
    cout << v.back() << endl;  // In ra 5
    cout << v.size() << endl; // In ra 5
    v.push_back(9);                     // v={1,2,3,4,5,9}
    cout << v.size() << endl;           // In ra 6
    v.clear();                          // v={}
    cout << v.empty() << endl;          // In ra 1 (vector rỗng)
    for (i=1;i<=5;i++) v.push_back(i); // v={1,2,3,4,5}
    v.pop_back();                       // v={1,2,3,4}
    cout << v.size() << endl;           // In ra 4
    v.erase(v.begin()+1);               // Xóa ptử thứ 1 v={1,3,4}
    v.erase(v.begin(),v.begin()+2);     // v={4}
    v.insert(v.begin(),100);            // v={100,4}
    v.insert(v.end(),5);                // v={100,4,5}
    /*Duyệt theo chỉ số phần tử*/
    for (i=0;i < v.size();i++) cout << v[i] << " "; // 100 4 5
    cout << endl;
    /*Chú ý: Không nên viết
    for (i=0;i<=v.size()-1;i++) ...
    Vì nếu vector v rỗng thì sẽ dẫn đến sai khi duyệt !!!
    */
    /*Duyệt theo iterator*/
    for (it=v.begin();it!=v.end();it++)
    cout << *it << " " ;
    //In ra giá trị ma iterator đang trỏ tới "100 4 5"
    cout << endl;
    /*Duyệt iterator ngược*/
    for (rit=v.rbegin();rit!=v.rend();rit++)
    cout << *rit << " "; // 5 4 100
    cout << endl;
}
```

*Ví dụ 7.2*: Cho đồ thị vô hướng G có n đỉnh (các đỉnh đánh số từ 1 đến n) và m cạnh và không có khuyên (đường đi từ 1 đỉnh tới chính đỉnh đó). Cài đặt đồ thị bằng danh sách kề và in ra các cạnh kề đối với mỗi cạnh của đồ thị.

| INPUT | OUTPUT |
|---|---|
| 6 7 | 2 3 5 6 |
| 1 2 | 1 3 6 |
| 1 3 | 1 2 5 |
| 1 5 | |
| 2 3 | 1 3 |
| 2 6 | 1 2 |
| 3 5 | |
| 6 1 | |

Input:

- Dòng đầu chứa n và m cách nhau bởi dấu cách

- M dòng sau, mỗi dòng chứa u và v cho biết có đường đi từ u tới v. Không có hai cặp đỉnh u, v nào chỉ cùng 1 đường đi.

Output: Gồm M dòng, dòng thứ i chứa các đỉnh kề cạnh i theo thứ tự tăng dần.

Giới hạn: $1 \le n, m \le 10000$

*Chương trình mẫu:*

```
#include <iostream>
#include <vector>
using namespace std;
vector < vector <int> > a (10001);
//Khai báo vector 2 chiều với 10001 vector 1 chiều rỗng chứa danh sách kề
int m,n,i,j,u,v;
main() {
    /*Input data*/
    cin >> n >> m;
    for (i=1;i<=m;i++) {
        cin >> u >> v;
        a[u].push_back(v);
        a[v].push_back(u);
    }
    /*Sort cạnh kề*/
    for (i = 1; i <=m ; i++)
        sort(a[i].begin(),a[i].end());
    /*Print Result*/
    for (i=1;i <= m;i++) {
        for (j=0;j < a[i].size();j++) cout << a[i][j] << " ";
        cout << endl;
    }
}
```

## II.3. Deque (Hàng đợi hai đầu)

Deque (thường được phát âm giống như "deck") là từ viết tắt của double-ended queue (hàng đợi hai đầu).

Deque có các ưu điểm như:

- Các phần tử có thể truy cập thông cua chỉ số vị trí của nó. O(1)
- Chèn hoặc xóa phần tử ở cuối hoặc đầu của dãy. O(1)

### II.3.1. Khai báo

```
#include <deque>
```

### II.3.2. Các hàm thành viên

| Nhóm | Tên hàm | Ý nghĩa | Độ phức tạp |
|---|---|---|---|
| Kích thước | **size** | trả về số lượng phần tử của deque. | O(1) |
| | **empty** | trả về true(1) nếu deque rỗng, ngược lại là false(0) | O(1) |
| Truy | **v[i]** | trả về giá trị phần tử thứ i của deque v[], không kiểm | O(1) |

| | | | |
|---|---|---|---|
| cập phần tử | | tra phạm vi | |
| | **at(i)** | trả về giá trị phần tử thứ i của deque v[], có kiểm tra phạm vi | O(1) |
| | **front** | trả về giá trị phần tử đầu tiên | O(1) |
| | **back** | trả về giá trị phần tử cuối cùng | O(1) |
| Thêm bớt phần tử | **push_back(x)** | thêm x vào cuối deque | O(1) |
| | **pop_back** | xóa bỏ phần tử ở cuối deque | O(1) |
| | **push_font(x)** | thêm x vào đầu deque | O(1) |
| | **pop_front** | xóa bỏ phần tử ở đầu deque | O(1) |
| | **insert(iterator,x)** | chèn *x* vào trước vị trí *iterator* ( x có thể là phần tử hay iterator của 1 đoạn phần tử…) | O(số phần tử chèn thêm) |
| | **erase(iterator)** | xóa phần tử ở vị trí *iterator* | O(số phần tử bị xóa đi) |
| | **erase(it1,it2)** | xóa các phần tử từ vị trí iterator *it1* đến vị trí *it2* | O(số phần tử bị xóa đi) |
| | **swap** | đổi 2 deque cho nhau. ví dụ: *first.swap(second)* | O(n) |
| | **clear** | xóa deque | O(n) |

*Ví dụ 7.3:* Minh họa việc sử dụng deque

```
#include <iostream>
#include <deque>
using namespace std;
main ()
{
    unsigned int i;
    deque<int> first;                // khai báo deque fisrt kiểu int
    deque<int> second (4,100);       // khai báo và khởi trị 4 p.tử là 100
         // khai báo third và khởi trị bằng các giá trị của second
    deque<int> third (second.begin(),second.end());
         // khai báo deque fourth và khởi trị từ third
    deque<int> fourth (third);
    int myints[] = {16,2,77,29};
         // khai báo và khởi trị từ mảng myints[]
    deque<int> fifth (myints, myints + sizeof(myints)/sizeof(int));
    cout << "Cac phan tu cua fifth gom:";
    deque<int>::iterator i;
    for (i = fifth.begin(); i!=fifth.end(); ++i) cout <<' '<< *i;
}
```

## II.4. List (Danh sách liên kết)

list được thực hiện như danh sách liên kết kép (doubly-linked list). Mỗi phần tử trong danh sách liên kết kép chứa con trỏ liên kết đến một phần tử trước đó và một phần tử sau nó.

Do đó, list có các ưu, nhược điểm sau:

- Chèn và loại bỏ phần tử ở bất cứ vị trí nào trong container. O(1).
- Điểm yếu của list là khả năng truy cập tới phần tử thông qua vị trí. O(n).

### III.4.1. Khai báo

```
#include <list>
```

### III.4.2. Các hàm thành viên

| Nhóm | Tên hàm | Ý nghĩa | Độ phức tạp |
|---|---|---|---|
| Kích thước | **size** | trả về số lượng phần tử của list. | O(1) |
| | **empty** | trả về true(1) nếu list rỗng, ngược lại là false(0) | O(1) |
| Truy cập phần tử | **front** | trả về giá trị phần tử đầu tiên | O(1) |
| | **back** | trả về giá trị phần tử cuối cùng | O(1) |
| Thêm bớt phần tử | **push_back(x)** | thêm x vào cuối list | O(1) |
| | **pop_back** | xóa bỏ phần tử ở cuối list | O(1) |
| | **push_front** | thêm x vào đầu list | O(1) |
| | **pop_front** | xóa bỏ phần tử ở đầu list | O(1) |
| | **insert(iterator,x)** | chèn *x* vào trước vị trí *iterator* ( x có thể là phần tử hay iterator của 1 đoạn phần tử…) | O(số phần tử chèn thêm) |
| | **erase(iterator)** | xóa phần tử ở vị trí *iterator* | O(số phần tử bị xóa đi) |
| | **erase(it1,it2)** | xóa các phần tử từ vị trí iterator *it1* đến vị trí *it2* | O(số phần tử bị xóa đi) |
| | **swap** | đổi 2 list cho nhau. ví dụ: *first.swap(second)* | O(n) |
| | **clear** | xóa list | O(n) |
| Thao tác với các phần tử | **splice** | di chuyển phần tử từ list này sang list khác | O(n) |
| | **remove(x)** | loại bỏ tất cả phần tử bằng x trong list | O(n) |
| | **remove_if (f)** | loại bỏ tất các phần tử trong list nếu hàm f ==true | O(n) |
| | **unique([f])** | loại bỏ các phần tử bị trùng lặp hoặc thỏa mãn hàm f==true. Lưu ý: Trước đó các phần tử trong list đã được sắp xếp | O(n) |
| | **sort** | sắp xếp các phần tử của list | O(NlogN) |
| | **reverse** | đảo ngược lại các phần tử của list | O(n) |

*Ví dụ 7.4*: Minh họa việc sử dụng list

```
#include <iostream>
#include <list>
using namespace std;
main ()
{
    list<int> first;           // khai baos list fisrt kiểu int rỗng
    list<int> second (4,100); // khai báo và khởi trị four bằng 100
        // khai báo và khởi trị third từ giá trị của second
    list<int> third (second.begin(),second.end());
    list<int> fourth (third); // khai báo và khởi trị fourth từ third
    int myints[] = {16,2,77,29};
        // khai báo list fifth và khởi trị từ mảng myints
    list<int> fifth(myints, myints + sizeof(myints)/sizeof(int));
    cout << "Cac phan tu cua fifth gom: ";
    std::list<int>::iterator i;
    for (i = fifth.begin(); i != fifth.end(); i++) cout <<*i <<' ';
}
```

## II.5. Stack (Ngăn xếp)

Stack là một loại container adaptor, được thiết kế để hoạt động theo kiểu LIFO (Last - in first - out) (vào sau ra trước), tức là một kiểu danh sách mà việc bổ sung và loại bỏ một phần tử được thực hiển ở cuối danh sách. Vị trí cuối cùng của stack gọi là đỉnh (top) của ngăn xếp.

### II.5.1. Khai báo

```
#include <stack>
```

### II.5.2. Các hàm thành viên

| Nhóm | Tên hàm | Ý nghĩa | Độ phức tạp |
|---|---|---|---|
| Kích thước | **size** | trả về số lượng phần tử của stack. | O(1) |
| | **empty** | trả về true(1) nếu stack rỗng, ngược lại là false(0) | O(1) |
| Truy cập phần tử | **top** | trả về giá trị phần tử ở đỉnh stack | O(1) |
| Thêm bớt phần tử | **push** | thêm x vào đỉnh stack | O(1) |
| | **pop** | xóa bỏ phần tử ở đỉnh stack | O(1) |

*Ví dụ 7.5:* Minh họa việc dùng stack

```
#include <iostream>
#include <stack>
using namespace std;
stack <int> s;
int i;
main() {
    for (i=1;i<=5;i++) s.push(i);  // s={1,2,3,4,5}
```

```
    s.push(100);                    // s={1,2,3,4,5,100}
    cout << s.top() << endl;        // In ra 100
    s.pop();                        // s={1,2,3,4,5}
    cout << s.empty() << endl;      // In ra 0
    cout << s.size() << endl;       // In ra 5
}
```

## II.6. Queue (Hàng đợi)

Queue là một loại container adaptor, được thiết kế để hoạt động theo kiểu FIFO (First - in first - out) (vào trước ra trước), tức là một kiểu danh sách mà việc bổ sung được thực hiển ở cuối danh sách và loại bỏ ở đầu danh sách.

Trong queue, có hai vị trí quan trọng là vị trí đầu danh sách (front), nơi phần tử được lấy ra, và vị trí cuối danh sách (back), nơi phần tử cuối cùng được thêm vào.

### II.6.1. Khai báo

```
#include <queue>
```

### II.6.2. Các hàm thành viên

| Nhóm | Tên hàm | Ý nghĩa | Độ phức tạp |
|---|---|---|---|
| Kích thước | **size** | trả về số lượng phần tử của queue. | O(1) |
| | **empty** | trả về true(1) nếu queue rỗng, ngược lại là false(0) | O(1) |
| Truy cập phần tử | **front** | trả về giá trị phần tử ở đầu queue | O(1) |
| | **back** | trả về giá trị phần tử ở cuối queue | O(1) |
| Thêm bớt phần tử | **push(x)** | thêm x vào cuối queue | O(1) |
| | **pop** | xóa bỏ phần tử ở đầu queue | O(1) |

*Ví dụ 7.5:* Minh họa việc dùng queue

```
#include <iostream>
#include <queue>
using namespace std;
queue <int> q;
int i;
main() {
    for (i=1;i<=5;i++) q.push(i);   // q={1,2,3,4,5}
    q.push(100);                    // q={1,2,3,4,5,100}
    cout << q.front() << endl;      // In ra 1
    q.pop();                        // q={2,3,4,5,100}
    cout << q.back() << endl;       // In ra 100
    cout << q.empty() << endl;      // In ra 0
    cout << q.size() << endl;       // In ra 5
}
```

## II.7. Priority Queue (Hàng đợi ưu tiên)

Priority queue là một loại container adaptor, được thiết kế đặc biệt để phần tử ở đầu luôn luôn lớn nhất (theo một quy ước về độ ưu tiên nào đó) so với các phần tử khác.

Nó giống như một heap, mà ở đây là heap max, tức là phần tử có độ ưu tiên lớn nhất có thể được lấy ra và các phần tử khác được chèn vào bất kì.

Phép toán so sánh mặc định khi sử dụng priority queue là phép toán less (Xem thêm ở thư viện functional). Để sử dụng priority queue một cách hiệu quả, bạn nên học cách viết hàm so sánh để sử dụng cho linh hoạt cho từng bài toán.

### II.7.1. Khai báo

```
#include <queue>

/*Dạng 1 (sử dụng phép toán mặc định là less)*/

priority_queue <int> pq;

/* Dạng 2 (sử dụng phép toán khác) */

priority_queue <int,vector<int>,greater<int> > q; //phép toán greater
```

Phép toán khác cũng có thể do người dùng tự định nghĩa. Ví dụ:

Cách khai báo ở dạng 1 tương đương với:

```
/* Dạng sử dụng class so sánh tự định nghĩa */

struct cmp{
    bool operator() (int a,int b) {return a<b;}
};
main() {
    …
    priority_queue <int,vector<int>,cmp > q;
}
```

### II.7.2. Các hàm thành viên

| Nhóm | Tên hàm | Ý nghĩa | Độ phức tạp |
|---|---|---|---|
| Kích thước | **size** | trả về số lượng phần tử của priority queue. | O(1) |
| | **empty** | trả về true(1) nếu priority queue rỗng, ngược lại là false(0) | O(1) |
| Truy cập phần tử | **top** | trả về giá trị phần tử ở đỉnh priority queue | O(1) |
| Thêm bớt phần tử | **push** | thêm x vào priority queue | O(1) |
| | **pop** | xóa bỏ phần tử ở đỉnh priority queue | O(1) |

*+ Ví dụ 7.6:*

```
#include <iostream>
#include <queue>
#include <vector>
```

```
using namespace std;
main() {
    priority_queue <int> p;    // p={}
    p.push(1);                 // p={1}
    p.push(5);                 // p={1,5}
    cout << p.top() << endl; // In ra 5
    p.pop();                   // p={1}
    cout << p.top() << endl; // In ra 1
    p.push(9);                 // p={1,9}
    cout << p.top() << endl; // In ra 9
}
```

+ *Ví dụ 7.7:*

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
main() {
    priority_queue < int , vector <int> , greater <int> > p; // p={}
    p.push(1);                 // p={1}
    p.push(5);                 // p={1,5}
    cout << p.top() << endl; // In ra 1
    p.pop();                   // p={5}
    cout << p.top() << endl; // In ra 5
    p.push(9);                 // p={5,9}
    cout << p.top() << endl; // In ra 5
}
```

+ *Ví dụ 7.8:*

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
struct cmp{
    bool operator() (int a,int b) {return a<b;}
};
main() {
    priority_queue < int , vector <int> , cmp > p; // p={}
    p.push(1);                 // p={1}
    p.push(5);                 // p={1,5}
    cout << p.top() << endl; // In ra 1
    p.pop();                   // p={5}
    cout << p.top() << endl; // In ra 5
    p.push(9);                 // p={5,9}
    cout << p.top() << endl; // In ra 5
}
```

*Ví dụ 7.9*: Thuật toán Dijkstra tìm đường đi ngắn nhất từ đỉnh s đến các đỉnh còn lại trong đơn đồ thị vô hướng G. Trong chương trình dưới đây G được biểu diễn dưới dạng danh sách kề và

có sử dụng hàng đợi ưu tiên để giảm chi phí thời gian của thuật toán từ $O(n^2)$ xuống O(nlogn). Dữ liệu của bài toán được biểu diễn như sau:

1. Khai báo danh sách kề: *vertor <list nut> dske*;

Trong đó dske[i] là danh sách các đỉnh kề với đỉnh i, mỗi đỉnh kề có kèm theo trọng số cạnh tương ứng, vì vậy kiểu *nut* là một kiểu cấu trúc có 2 thành phần: v - số hiệu đỉnh, w - trọng số cạnh.

2. Khai báo heap là một hàng đợi ưu tiên phục vụ cho thuật toán Dijkstra. Mỗi thành phần của heap chứa hai giá trị: v - chỉ số đỉnh, w - độ dài đường đi từ s đến v. Vì vậy heap có kiểu là hàng đợi ưu tiên với các phần tử *nut*. Bên cạnh đó, các phần tử của heap là kiểu cấu trúc, ta cần tự viết hàm so sánh cho hàng đợi ưu tiên kiểu như ví dụ 7.8. Việc xuất/nhập dữ liệu sử dụng dạng file văn bản graph.inp, graph.out.

Input:

- Dòng đầu lần lượt chứa số đỉnh n, số cạnh m và đỉnh xuất phát s ($n \leq 1000$)

- m dòng sau, mỗi dòng chứa 3 số nguyên dương u, v, w với ý nghĩa là cạnh (u,v) có trọng số w.

Output: Ghi độ dài đường đi từ s đến mỗi đỉnh và mô tả đường đi. Xem ví dụ:

| graph.inp | graph.out |
|---|---|
| 5 7 1<br>1 2 10<br>1 3 2<br>1 5 100<br>2 4 3<br>3 2 5<br>4 3 15<br>4 5 5 | do dai duong di tu 1 den 2 = 7<br>1 -> 3 -> 2<br>do dai duong di tu 1 den 3 = 2<br>1 -> 3<br>do dai duong di tu 1 den 4 = 10<br>1 -> 3 -> 2 -> 4<br>do dai duong di tu 1 den 5 = 15<br>1 -> 3 -> 2 -> 4 -> 5 |

Sau đây là chương trình chi tiết:

```
#include <fstream>
#include <climits>
#include <queue>
#include <list>
#include <vector>
using namespace std;
const int MAXN = 10000;
struct nut { int v, w; };
struct khoangcach{ int dodai, truoc;};

struct cmp{ bool operator()(nut x, nut y){return x.w > y.w;} };
vector< list<nut> > dske;
vector<khoangcach> d;
priority_queue<nut, vector<nut>, cmp > Q ;
int n, m, s;
nut start;
ofstream fou("graph.out");
ifstream fin("graph.inp");
void get_data() {
    fin >> n >> m >> s;
```

```
        dske.resize(n+1);
        int u, v, w;
        nut nu;
        for (int i = 1; i <= m; ++i) {
            fin >> u >> v >> w;
            nu.v = v; nu.w = w;
            dske[u].push_back(nu);
            nu.v = u; nu.w = w;
            dske[v].push_back(nu);
        }
    }
    void printPath(int i){
        if (d[i].truoc) { printPath(d[i].truoc); fou << " -> " <<i; }
        else fou<<i;
    }
    void printPaths(){
        for(int i = 1; i<=n ; i++)
        if(i!=s) {
            fou << "do dai duong di tu "<< s << " den "<<i<<" = " <<
    d[i].dodai<<endl;
            printPath(i); fou<<endl;
        }
    }

    void Dijkstra(nut s) {
        khoangcach x;
        for (int i = 1; i <= n; i++) {
            x.dodai = INT_MAX; x.truoc = s.v;
            d.push_back(x);
        }
        d[s.v].dodai = 0; d[s.v].truoc = 0;
        Q.push(s);
        nut nu;
        while (!Q.empty()) {
            nu = Q.top(); Q.pop();
            if (nu.w <= d[nu.v].dodai){


                for (list<nut>::iterator i = dske[nu.v].begin(); i !=
    dske[nu.v].end(); ++i)
                    if (d[i->v].dodai > d[nu.v].dodai + i->w) {
                        d[i->v].dodai = d[nu.v].dodai + i->w;
                        d[i->v].truoc = nu.v;
                        nut no;
                        no.v = i->v; no.w = d[i->v].dodai;
                        Q.push(no);
                    }
            }
```

```
            }
        }

        int main() {
            get_data();
            start.v = s; start.w = 0; d.resize(1);
            Dijkstra(start);
            printPaths();
            return 0;
        }
```

## II.8. Set (Tập hợp)

Set là một loại associative containers để lưu trữ các phần tử không bị trùng lặp (unique elements), và các phần tử này chính là các khóa (keys).

Khi duyệt set bằng iterator từ begin đến end, các phần tử của set sẽ tăng dần theo phép toán so sánh.

Mặc định của set là sử dụng phép toán less, bạn cũng có thể viết lại hàm so sánh theo ý mình.

Set được thực hiện giống như cây tìm kiếm nhị phân (Binary search tree).

### II.8.1. Khai báo

```
#include <set>

set <int> s;
set <int, greater<int> > s;
```

Hoặc viết class so sánh theo ý mình:

```
struct cmp{
    bool operator() (int a,int b) {return a<b;}
};
set <int,cmp > myset ;
```

### II.8.2. Các hàm thành viên

| Nhóm | Tên hàm | Ý nghĩa | Độ phức tạp |
|---|---|---|---|
| Kích thước | **size** | trả về số lượng phần tử của set. | O(1) |
| | **empty** | trả về true(1) nếu set rỗng, ngược lại là false(0) | O(1) |
| Thêm bớt phần tử | **insert(x)** | chèn *x* vào set | O(logN) |
| | **erase(iterator)** | xóa phần tử ở vị trí *iterator* | O(logN) |
| | **erase(it1,it2)** | xóa các phần tử từ vị trí iterator *it1* đến vị trí *it2* | O(logN) |
| | **erase(key)** | xóa phần tử có giá trị key trong set | O(số phần tử bị xóa đi) |
| | **swap** | đổi 2 set cho nhau. ví dụ: *first.swap(second)* | O(n) |
| | **clear** | xóa set | O(n) |

| | | | |
|---|---|---|---|
| Thao tác với các phần tử | **find** | trả về itarator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy itarator trỏ về *end()* của set | O(LogN) |
| | **lower_bound(key)** | trả về iterator đến vị trí phần tử nhỏ nhất ≥ key (theo phép so sánh), nếu không tìm thấy trả về vị trí *end()* của set. | O(LogN) |
| | **upper_bound(key)** | trả về iterator đến vị trí phần tử nhỏ nhất > key, nếu không tìm thấy trả về vị trí *end()* của set. | O(LogN) |
| | **count(key)** | Trả về 1 nếu key có trong set, và 0 nếu không có | O(LogN) |

*Chương trình Ví dụ 7.10:*

```
#include <iostream>
#include <set>
using namespace std;
main() {
    set <int> s;
    set <int> ::iterator it;
    s.insert(9);                    // s={9}
    s.insert(5);                    // s={5,9}
    cout << *s.begin() << endl;     //In ra 5
    s.insert(1);                    // s={1,5,9}
    cout << *s.begin() << endl;     // In ra 1
    it = s.find(5);
    if (it==s.end()) cout << "Khong co " << endl;
    else cout << "Co trong set" << endl;
    s.erase(it);                    // s={1,9}
    s.erase(1);                     // s={9}
    s.insert(3);                    // s={3,9}
    s.insert(4);                    // s={3,4,9}
    it=s.lower_bound(4);
    if (it==s.end()) cout << "Trong set ko co ptu >= 4" << endl;
    else cout << "Ptu nho nhat >= 4 la "<< *it <<endl; //In ra 4
    it=s.lower_bound(10);
    if (it==s.end()) cout << "Trong set ko co ptu >=10" << endl;
    else cout << "Ptu nho nhat >= 10 la " << *it << endl; //Khong co
    it=s.upper_bound(4);
    if (it==s.end()) cout << "Trong set k0 co ptu > 4" << endl;
    else cout << "Phan tu nho nhat > 4 la " << *it << endl; //In ra 9
    // Duyet set: In ra 3 4 9 */
    for (it=s.begin();it!=s.end();it++) cout << *it << " ";
    cout << endl;
}
```

*Lưu ý*: Nếu bạn muốn sử dụng hàm *lower_bound* hay *upper_bound* để tìm phần tử lớn nhất "bé hơn hoặc bằng" hoặc "bé hơn" bạn có thể thay đổi cách so sánh của set để tìm kiếm. Chương trình sau sẽ min họa rõ hơn:

*Ví dụ 7.11:*

```
#include <iostream>
#include <set>
using namespace std;
main() {
    set <int, greater <int> > s;
    set <int, greater <int> > :: iterator it; //so sánh là greater
    s.insert(1);                // s={1}
    s.insert(2);                // s={2,1}
    s.insert(4);                // s={4,2,1}
    s.insert(9);                // s={9,4,2,1}
    /* Tim phần tử lớn nhất bé hơn hoặc bằng 5 */
    it=s.lower_bound(5);
    cout << *it << endl;        // In ra 4
    /* Tim phần tử lớn nhất bé hơn 4 */
    it=s.upper_bound(4);
    cout << *it << endl;        // In ra 2
}
```

## II.9. Mutiset (Tập hợp)

Multiset giống như set nhưng có thể chứa các khóa có giá trị giống nhau.

### II.9.1. Khai báo

```
#include <set>
```

### II.9.2. Các hàm thành viên

Các hàm thành viên trong multiset giống như trong set. Tuy nhiên multiset có thể chứa nhiều key giống nhau, nên có vài điểm lưu ý:

- Hàm **`find`** : Dù trong multiset có nhiều phần tử bằng khóa thì nó cũng chỉ iterator đến một phần tử.
- Hàm **`count`** : trả về số lần xuất hiện của khóa trong multiset.

*Ví dụ 7.12:*

```
#include <iostream>
#include <set>
using namespace std;
main() {
    multiset <int> s;
    multiset <int> :: iterator it;
    int i;
    for (i=1;i<=5;i++) s.insert(i*10);        // s={10,20,30,40,50}
    s.insert(30);                             // s={10,20,30,30,40,50}
    cout << s.count(30) << endl;              // In ra 2
    cout << s.count(20) << endl;              // In ra 1
    s.erase(30);                              // s={10,20,40,50}
    //Duyet set: In ra 10 20 40 50
```

```
        for (it=s.begin();it!=s.end();it++) cout << *it << " ";
        cout << endl;
    }
```

## II.10. Map (Ánh xạ)

Map là một loại associative container. Mỗi phần tử của map là sự kết hợp của khóa (key value) và ánh xạ của nó (mapped value). Cũng giống như set, trong map không chứa các khóa mang giá trị giống nhau.

Trong map, các khóa được sử dụng để xác định giá trị các phần tử. Kiểu của khóa và ánh xạ có thể khác nhau.

Và cũng giống như set, các phần tử trong map được sắp xếp theo một trình tự nào đó theo cách so sánh.

Map được cài đặt bằng red-black tree (cây đỏ đen) – một loại cây tìm kiếm nhị phân tự cân bằng. Mỗi phần tử của map lại được cài đặt theo kiểu pair (xem thêm ở thư viện utility).

### II.10. Khai báo

```
#include <map>

...
map <kiểu_dữ_liệu_1,kiểu_dữ_liệu_2>
// kiểu dữ liệu 1 là khóa, kiểu dữ liệu 2 là giá trị của khóa.
```

Sử dụng class so sánh:

```
struct cmp{
    bool operator() (char a,char b) {return a<b;}
};
.....
map <char,int,cmp> m;
```

Truy cập đến giá trị của các phần tử trong map khi sử dụng iterator:

Ví dụ ta đang có một iterator là *it* khai báo cho *map* thì:

```
(*it).first;    // Lấy giá trị của khóa, kiểu_dữ_liệu_1
(*it).second;   // Lấy giá trị của giá trị của khóa, kiểu_dữ_liệu_2
(*it)           // Lấy giá trị của phần tử mà iterator đang trỏ đến, kiểu pair
it->first;      // giống như (*it).first
it->second;     // giống như (*it).second
```

### II.10.2. Các hàm thành viên

| Nhóm | Tên hàm | Ý nghĩa | Độ phức tạp |
|---|---|---|---|
| Kích thước | **size** | trả về số lượng phần tử của map. | O(1) |
| | **empty** | trả về true(1) nếu map rỗng, ngược lại là false(0) | O(1) |
| Truy cập phần tử | **m[key]** | Nếu key có trong map m, thì hàm này sẽ trả về giá trị mà khóa ánh xạ đến. Ngược lại, nếu key chưa có trong m, thì khi gọi m[key] nó sẽ thêm vào m khóa đó. | O(logN) |

| | | | |
|---|---|---|---|
| Thêm bớt phần tử | **insert(p)** | Chèn phần tử p vào map. Chú ý: phần tử p chèn vào phải ở kiểu *pair* | O(logN) |
| | **erase(iterator)** | xóa phần tử ở vị trí *iterator* | O(logN) |
| | **erase(key)** | xóa phần tử có giá trị key trong map | O(số phần tử bị xóa đi) |
| | **swap** | đổi 2 map cho nhau. ví dụ: *first.swap(second)* | O(n) |
| | **clear** | xóa map | O(n) |
| Thao tác với các phần tử | **find** | trả về itarator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy itarator trỏ về *end()* của map | O(LogN) |
| | **lower_bound(key)** | trả về iterator đến vị trí phần tử nhỏ nhất ≥ key (theo phép so sánh), nếu không tìm thấy trả về vị trí *end()* của map. | O(LogN) |
| | **upper_bound(key)** | trả về iterator đến vị trí phần tử nhỏ nhất > key, nếu không tìm thấy trả về vị trí *end()* của map. | O(LogN) |
| | **count(key)** | Trả về số lần xuất hiện key trong map. | O(LogN) |
| | | | |

*Chương trình demo:*

```
#include <iostream>
#include <map>
using namespace std;
main() {
    map <char,int> m;
    map <char,int> :: iterator it;
    m['a']=1;                              // m={{'a',1}}
    m.insert(make_pair('b',2));            // m={{'a',1};{'b',2}}
    m.insert(pair<char,int>('c',3) );      // m={{'a',1};{'b',2};{'c',3}}
    cout << m['b'] << endl;                // In ra 2
    m['b']++;                              // m={{'a',1};{'b',3};{'c',3}}
    it=m.find('c');                        // it point to key 'c'
    cout << it->first << endl;             // In ra 'c'
    cout << it->second << endl;     // In ra 3
    m['e']=100;                     //m={{'a',1};{'b',3};{'c',3};{'e',100}}
    it=m.lower_bound('d');          // it point to 'e'
    cout << it->first << endl;      // In ra 'e'
    cout << it->second << endl;     // In ra 100
}
```

## II.11. Multi Map (Ánh xạ)

Giống như map nhưng có thể chứa các phần tử có khóa giống nhau, do đó nó khác map ở chỗ không có opearator[].

# III. THƯ VIỆN ALGORITHMS (LỚP THUẬT TOÁN)

Khai báo sử dụng: **`#include <algorithm>`**

Các hàm trong STL Algorithm khá nhiều, tài liệu này chỉ giới thiệu về một số hàm hay sử dụng trong lập trình giải toán.

Có một lưu ý nhỏ là khi sử dụng các hàm mà thực hiện trong một đoạn phần tử liên tiếp nào đó thì các hàm trong c++ thường có tác dụng trên nửa đoạn [..).

Ví dụ: muốn hàm f có tác dụng trong đoạn từ 1→n thì phải gọi hàm trong đoạn từ 1→n+1.

## III.1. Hàm min, max

### III.1.1. Hàm min

Trả về giá trị bé hơn theo phép so sánh (mặc định là phép toán less):

Ví dụ: `min('a','b')` trả về `'a';` `min(3,1)` trả về `1;`

### III.1.2. Hàm max

Hàm *max* ngược lại với hàm *min*:

Ví dụ: `max('a','b')` trả về `'b';` `min(3,1)` trả về `3;`

### III.1.3. Hàm next_permutation

Hàm này sinh ra hoán vị tiếp theo của dãy tham đối. Hàm trả về 1 nếu có hoán vị tiếp theo, 0 nếu không có hoán vị tiếp theo.

Ví dụ:

```
/* next_permutation */
#include <iostream>
#include <algorithm>
using namespace std;
int main () {
    int myints[] = {1,2,3};
    do {
         cout << myints[0]<<" "<< myints[1] <<" "<<myints[2]<<", ";
    } while ( next_permutation (myints,myints+3) );
    return 0;
}
```

Chương trình trên sẽ in ra kết quả gồm 3! hoán vị của tập {1, 2, 3}

```
1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 4 2 1
```

### III.1.4. Hàm prev_permution

Ngược lại với *next_permutation*, hàm *prev_permution* sinh ra hoán vị liền trước của hoán vị do tham đối cung cấp. Hàm trả về 1 nếu có hoán vị liền trước, 0 nếu không có.

## III.2. Hàm sort

Hàm *sort* sắp xếp đoạn phần tử theo một trình tự nào đó. Mặc định của sort là sử dụng phép toán so sánh <.

Có thể sử dụng hàm so sánh, hay class so sánh tự định nghĩa để sắp xếp cho linh hoạt.

*Chương trình mẫu:*

```
/* sort algorithm example */
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool myfunc (int i,int j) { return (i<j); }
struct myclass {
    bool operator() (int i,int j) { return (i>j);}
} myobject;

int main () {
    int myints[] = {32,71,12,45,26,80,53,33};
    // dùng phép toán so sánh mặc định là <
    sort (myints, myints+4);  //(12 32 45 71)26 80 53 33
    // dùng hàm so sánh tự định nghĩa
    sort (myints+4, myints+8, myfunc); // 12 32 45 71(26 33 53 80)
    for (int i=0;i<8;i++) cout << myints[i] << " ";
    cout << endl;
    // Dùng đối tượng so sánh tự định nghĩa
    sort (myints, myints+8, myobject);  //(80 71 53 45 33 32 26 12)
    for (int i=0;i<8;i++) cout << myints[i] << " ";
    cout << endl;
    return 0;
}
```

## III.3. Các hàm tìm kiếm nhị phân (áp dụng với dãy đã sắp xếp)

### III.3.1. Hàm binary_search

Hàm binary_search tìm khóa có trong dãy cần tìm không. Lưu ý: đoạn tìm kiếm phải được sắp xếp theo một trật tự nhất định. Nếu tìm được sẽ trả về true, ngược lại trả về false.

- Dạng 1: **`binary_search(pbegin, pend, key);`**

- Dạng 2: **`binary_search(pbegin, pend, key, cmp_function);`**

```
// binary_search example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool myfunction (int i,int j) { return (i<j); }
int main () {
    int myints[] = {1,2,3,4,5,4,3,2,1};
    vector<int> v(myints,myints+9);  // 1 2 3 4 5 4 3 2 1
    // Sử dụng toán tử so sánh mặc định
```

```
    sort (v.begin(), v.end());
    cout << "looking for a 3... ";
    if (binary_search (v.begin(), v.end(), 3))
    cout << "found!\n"; else cout << "not found.\n";
    // sử dụng hàm so sánh tự định nghĩa:
    sort (v.begin(), v.end(), myfunction);
    cout << "looking for a 6... ";
    if (binary_search (v.begin(), v.end(), 6, myfunction))
        cout << "found!\n";
    else
        cout << "not found.\n";
    return 0;
}
```

### III.3.2. Hàm lower_bound

Hàm *lower_bound* và *upper_bound* tương tự như ở trong *set* hay *map*.

Trả về iterator đến phần tử đầu tiên trong nửa đoạn *[first, last]* mà có giá trị không bé hơn khóa tìm kiếm.

Dạng 1: **`lower_bound(pbegin , pend, key);`**

Dạng 2: **`lower_bound(pbegin , pend, key, cmp_function);`**

### III.3.3. Hàm upper_bound

Hàm *upper_bound* trả về iterator đến phần tử đầu tiên trong nửa đoạn *[first,last]* mà có giá trị lớn hơn khóa tìm kiếm.

Dạng 1: **`upper_bound(pbegin , pend, key);`**

Dạng 2: **`upper_bound(pbegin , pend, key, cmp_function);`**

*// lower_bound/upper_bound example*

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main () {
    int myints[] = {10,20,30,30,20,10,10,20};
    vector<int> v(myints,myints+8);// 10 20 30 30 20 10 10 20
    vector<int>::iterator low,up;
    sort (v.begin(), v.end());      // 10 10 10 20 20 20 30 30
    low=lower_bound (v.begin(), v.end(), 20);
    up= upper_bound (v.begin(), v.end(), 20);
    cout << "lower_bound at position ";
    cout << int(low- v.begin()) << endl;
    cout << "upper_bound at position ";
    cout << int(up - v.begin()) << endl;
    return 0;
}
```

Kết quả in ra màn hình:

```
lower_bound at position 3
upper_bound at position 6
```

# IV. THƯ VIỆN STRING C++

*string* là một kiểu đặc biệt của container, thiết kế đặc để hoạt động với các chuỗi kí tự.

Khai báo: **`#include <string>`**

### *a. Phép duyệt string*

Sử dụng con trỏ iterator như với các container khác, string cũng hỗ trợ các iterator như *begin, end, rbegin, rend* với ý nghĩa như đã nói trong các container trước.

### *b. Nhập, xuất string*

- Sử dụng toán tử >> : Nhập đến khi gặp dấu cách.
- Sử dụng `getline`: Nhập cả một dòng kí tự cho string (nhận cả dấu cách nếu có).
- Xuất string: sử dụng toán tử *cout* << như bình thường.

### *c. Các phép toán*

Trong string bạn có thể sử dụng:

- Toán tử gán "=" để gán giá trị cho string,
- toán tử "+" để nối hai string với nhau,
- toán tử các so sánh "==", "!=" để so sánh hai string. Chức năng này khá giống với xâu ở trong NNLT pascal.

### *d. Các hàm thành viên*

| Nhóm hàm | Tên hàm | Ý nghĩa |
|---|---|---|
| Kích thước | **`size`** | trả về số lượng phần tử của string. |
| | **`empty`** | trả về true(1) nếu string rỗng, ngược lại là false(0) |
| | **`length`** | trả về số lượng phần tử của string (~ hàm **`size`**) |
| | **`resize(n,c)`** | đổi lại kích thước string thành n ký tự. Nếu n < length thì string bị xén đi, ngược lại sẽ thêm các ký tự c vào cuối string cho đủ n ký tự. Nếu không có tham số c thì string chỉ được thêm ký tự rỗng (coi như không thêm). Khi sử dụng cần chú ý kích thước tối đa của string. |
| | **`clear`** | xóa string |
| | **`max_size`** | trả về độ dài tối đa của string |
| Truy cập phần | **`begin`** | trả về con trỏ interator trỏ đến phần tử đầu tiên của string |
| | **`end`** | trả về con trỏ interator trỏ đến phần tử cuối của string |

| | | |
|---|---|---|
| tử | **rbegin** | trả về con trỏ reverse_interator trỏ đến phần tử cuối của string |
| | **rbegin** | trả về con trỏ reverse_interator trỏ đến phần tử đầu của string |
| | **s[i]** | trả về phần tử thứ i của biến string s. Không kiểm tra phạm vi |
| | **at(i)** | trả về phần tử thứ i string s. Có kiểm tra phạm vi |
| Thêm bớt phần tử | **toán tử +/+=** | nối hai string. Ví dụ: *s1 = s1+ s2* hoặc *s1 += s2;* |
| | **push_back(x)** | thêm ký tự x vào cuối string |
| | **insert(i,x)** | chèn *x* vào trước vị trí *i* của string (x có thể là kiểu string/C string) |
| | **insert(i,n,x)** | chèn ký tự *x* vào trước vị trí *i* của string và lặp n lần |
| | **erase(i,n)** | xóa n phần tử tính từ vị trí i của string |
| | **replace(i,n,s)** | Thay thế n phần tử tính từ vị trí i của string bằng s (s có thể là kiểu string/C string/char) |
| | **swap** | đổi 2 string cho nhau. ví dụ: *first.swap(second)* |
| Thao tác khác với string | **c_str** | trả về chuỗi giá trị của string ở dạng C string (đổi từ str → C string) |
| | **copy(s,n,i)** | trả về một chuỗi s dạng C string là một phần của string, gồm n ký tự, tính từ vị trí i (mặc định i = 0). Không tự thêm '\0' vào cuối s. |
| | **find(s,i)** | trả về vị trí đầu tiên xuất hiện chuỗi s trong string, tính từ vị trí i (mặc định i=0, s có thể là kiểu string/C string/char) |
| | **rfind(s,i)** | tương tự hàm find nhưng tìm từ cuối string trở lại |
| | **find_fisrt_of (s,i)** | trả về vị trí đầu tiên trong string, ký tự có xuất hiện trong chuỗi s, tính từ vị trí i (mặc định i=0, s có thể là kiểu string/C string/char) |
| | **rfind_fisrt_of (s,i)** | tương tự hàm find_fisrt_of, nhưng tìm từ cuối string trở lại |
| | **find_fisrt_not_ of (s,i)** | trả về vị trí đầu tiên trong string, ký tự không xuất hiện trong chuỗi s, tính từ vị trí i (mặc định i=0, s có thể là kiểu string/C string/char) |
| | **rfind_fisrt_not _of (s,i)** | tương tự hàm find_fisrt_not_of, nhưng tìm từ cuối string trở lại |
| hằng | **npos** | Cho biết kết thúc chuỗi. Thường có giá trị -1, thường để chỉ kết quả tìm kiếm thất bại (không tìm thấy) |

# V. THƯ VIỆN UTILITY

Trong thư viện này, ta chỉ tìm hiểu về đối tượng *pair*. Đây là đối tượng liên quan đến việc sử dụng các thư viện đã nói ở trên