l

C,C++ Programming

Linh La

Table of Contents

[I. C/C++ Cơ bản: 2](#_Toc462944494)

[1. Trình biên dịch 2](#_Toc462944495)

[2. Cú pháp cơ bản C,C++ 16](#_Toc462944496)

[3. Tổ chức bộ nhớ: 22](#_Toc462944497)

[4. Liên kết (Link): 31](#_Toc462944498)

[5. Qui tắc soạn thảo mã nguồn: 43](#_Toc462944499)

[6. Cơ Bản Debug Trong Visual Studio: 50](#_Toc462944500)

[7. Bản chất của biến: 54](#_Toc462944501)

[8. Phát Hiện Memory Leak Với VLD 58](#_Toc462944502)

[9. Biểu diễn số âm trong máy tính 62](#_Toc462944503)

[10. Biến và kiểu dữ liệu: 67](#_Toc462944504)

[11. Toán tử, độ ưu tiên và trật tự kết hợp: 77](#_Toc462944505)

[12. Cơ bản về chuyển đổi kiểu 83](#_Toc462944506)

[13. Định dạng chuỗi dữ liệu : 85](#_Toc462944507)

[14. Phạm vi biến: 91](#_Toc462944508)

[15. Đặc tính lưu trữ: 94](#_Toc462944509)

[16. Nhập xuất cơ bản trong C, C++ 100](#_Toc462944510)

[17. Binary và Hexa decimal 114](#_Toc462944511)

[18. Chuỗi: 137](#_Toc462944512)

[19.  Con trỏ trong C,C++ 145](#_Toc462944513)

[20. Cấp phát tĩnh và cấp phát động: 160](#_Toc462944514)

[21. Mảng: 164](#_Toc462944515)

[22. Tham chiếu C++ 175](#_Toc462944516)

[23. Thao Tác Với Constant Trong C,C++ 180](#_Toc462944517)

[24. Cấu trúc điều khiển: 185](#_Toc462944518)

[25. Typedef – trừu tượng hóa kiểu dữ liệu 200](#_Toc462944519)

[26. Kiểu enum 202](#_Toc462944520)

[27. Enum class 207](#_Toc462944521)

[28. Cấu trúc (Struct): 209](#_Toc462944522)

[29. Cấu trúc union: 217](#_Toc462944523)

[30. Cơ bản về hàm 222](#_Toc462944524)

[31. Random Số Trong C,C++ 229](#_Toc462944525)

[II. C/C++ Nâng cao 231](#_Toc462944526)

[1. Chuyển đổi kiểu chuỗi và số 231](#_Toc462944527)

[2. Kỹ thuật bit flags và bit mask 233](#_Toc462944528)

[3. Preprocessor và #include Directive 239](#_Toc462944529)

[4. Hàm có tham số tùy ý trong C 250](#_Toc462944530)

[5. Exception Handling C, C++ 252](#_Toc462944531)

[6. Ép kiểu trong C,C++ 260](#_Toc462944532)

[7. Date và Time trong C,C++ 264](#_Toc462944533)

[8. File I/O 268](#_Toc462944534)

[9. Đối số dòng lệnh trong C, C++ 302](#_Toc462944535)

[III. C,C++ Library and Dictionary: 305](#_Toc462944536)

[1. Thư viện hàm chuẩn: 305](#_Toc462944537)

[2. Thư viện dành cho lớp đối tượng: 330](#_Toc462944538)

[3. Chuỗi định dạng I/O 331](#_Toc462944539)

[4. Bảng mã ASCII 334](#_Toc462944540)

## C/C++ Cơ bản:

### Trình biên dịch

##### GNU là gì:

GNU được khởi đầu vào năm 1984, mục đích nhằm phát triển một hệ điều hành dạng Unix đầy đủ, đó là phần mềm tự do: hệ thống GNU. GNU được cấu tạo từ các chữ đầu của cụm từ "GNU's Not Unix". Nhiều biến thể của hệ thống GNU sử dụng nhân (kernel) Linux đang được phổ biến rộng rãi; mặc dù các hệ thống này thường được gọi là "Linux," tên chính xác cho các hệ thống này là Hệ điều hành GNU/Linux.

##### Giới thiệu GNU Complier:

Trình biên dịch làm nhiệm vụ chuyển mã nguồn sang mã máy để CPU có thế thực thi chương trình của bạn theo từng lệnh.

Trình biên dịch sẽ chuyển từng file nguồn riêng biệt \*.c đối với C compiler, \*.cpp đối với C++ compiler thành file \*.obj

Ngoài Visual Studio của Microsoft, GNU C/C++ là một trong những trình biên dịch khá phổ biến.

GNU gcc/g++ đều "hỗ trợ" C/C++ "chuẩn". Vì thế khi bạn biên dịch một chương trình với gcc/g++ bạn sẽ không phải lo việc chương trình của bạn viết đúng theo chuẩn C/C++ mà không biên dịch được.

Một điều các bạn ít để ý là đa số các IDE phổ biến để viết C/C++ bây giờ đều sử dụng các phiên bản khác nhau của GNU gcc/g++. Khi chạy chương trình thì các IDE này lại gọi tới các trình dịch gcc/g++.

1. GCC trên Window

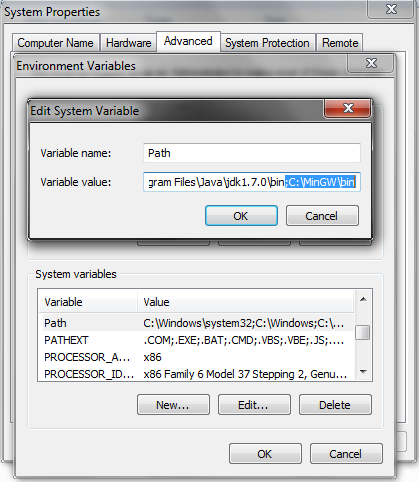
Khác với Linux thường được "cài" sẵn bộ trình dịch này, Windows là một hệ điều hành nguồn đóng nên họ chỉ muốn cho ta dùng những thứ của họ (Visual chẳng hạn).

Vì thế ta cần cài đặt thêm bộ trình dịch này vào Windows. Trong đó MinGW là phiên bản GNU compiler phổ biến nhất dành cho Windows.

Vào trang [**www.mingw.org**](http://www.mingw.org/), download phiên bản phù hợp với OS của bạn.

Trong khi setup MinWG bạn sẽ có gcc-core, gcc-g++, binutils, và MinGW runtime.

Chèn địa chỉ thư mục “\...\MinGW\bin” tới **Path environment variable**, để bạn có thể chạy tools trong command line.



Khi hoàn tất cài đặt bạn có thể chạy gcc, g++, ar, ranlib, dlltool, và vài công cụ GNU từ Windows command line.

1. Chạy chương trình C với GNU

Mở trình soạn thảo văn bản notepad, gõ đoạn code sau vào và lưu thành file hello.c

#include <stdio.h>

int main(){

/\* my first program in C \*/

printf("Hello, World! \n");

return 0;

}

Vào chế độ console, vào thư mục chứa file hello.c rồi đánh lệnh sau để biên dịch:

> gcc hello.c

Trình biên dịch sẽ biên dịch và liên kết hello.c thành file chạy mặc định là a.exe

Để chạy chương trình, ta đánh tên file chạy:

> a

$ ./a

1. Biên dịch và chạy chương trình C++

Mở trình soan thảo văn bản notepad, gõ đoạn code bên trên vào

Lưu thành file *hello.cpp*

Mở command prompt và đi tới đường dẫn lưu file.

Bạn cần g++ để biên dịch chương trình C++, chúng ta dùng tùy chọn –o để xác định tên file xuất ra.

> g++ -o hello.exe hello.cpp

> hello

$ ./hello

Hello World

##### Tạo Chương Trình C++ Đầu Tiên Với Visual Studio

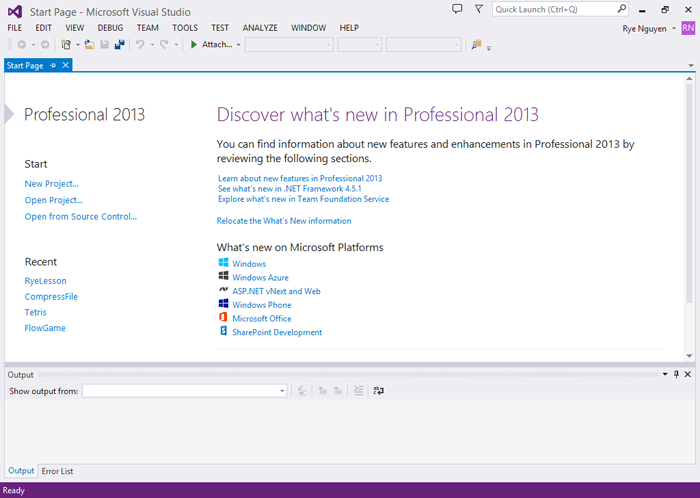
Visual Studio là một bộ công cụ hỗ trợ lập trình được Microsoft phát triển. Nhờ có Visual Studio, các nhà phát triển phần mềm có thể xây dựng các ứng dụng Windows, ứng dụng Web … một cách hiệu quả và dễ dàng hơn. Do đó việc sử dụng thành thạo Visual Studio là một điều cần thiết và quan trọng đối với lập trình viên. Bài viết này sẽ hướng dẫn các bạn cách tạo ra một project C++ trong Visual Studio và viết một chương trình đơn giản, qua đó giúp các bạn làm quen với bộ công cụ rất mạnh này.

1. Các chương trình cần sử dụng

Phiên bản tôi sử dụng trong bài hướng dẫn là Visual Studio 2013 Professional. Tuy nhiên các bạn có thể chọn bất kì phiên bản nào của Visual Studio để học và làm việc. Các phiên bản khác có thể sẽ có một số điểm khác biệt nhỏ.

1. Giao diện của Visual Studio

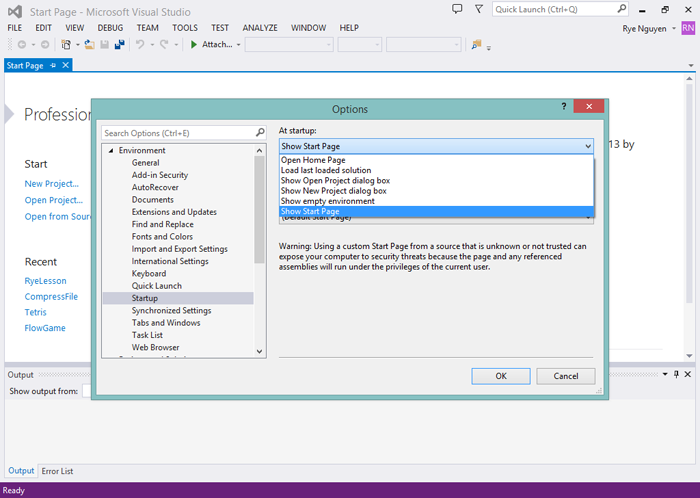
Sau khi khởi động, cửa sổ làm việc của chương trình sẽ có dạng như sau:



Như các bạn thấy thì giao diện chương trình tương đối đơn giản và trực quan. Mục Start bao gồm một số thao tác với project, mục Recent hiển thị những project mà bạn đã làm việc trước đó. Cả hai mục trên đều nằm trong tab Start Page của Visual Studio. Lưu ý là ở mỗi phiên bản Visual Studio sẽ có đôi chút khác biệt ở Start Page.

Nếu không hiện Start Page, các bạn thực hiện như sau:

* Vào menu Tool ➙ Option.
* Chọn tab Environment ➙ Startup.
* Click chuột vào hộp thoại At startup và chọn Show Start Page.

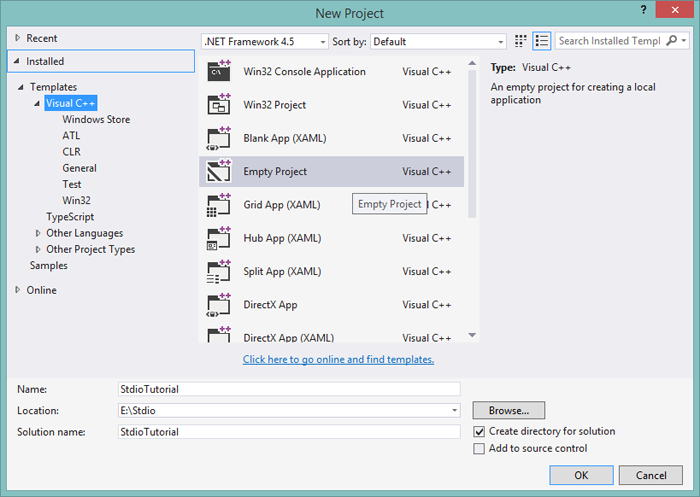


1. Tạo Project C++ đầu tiên với Visual Studio

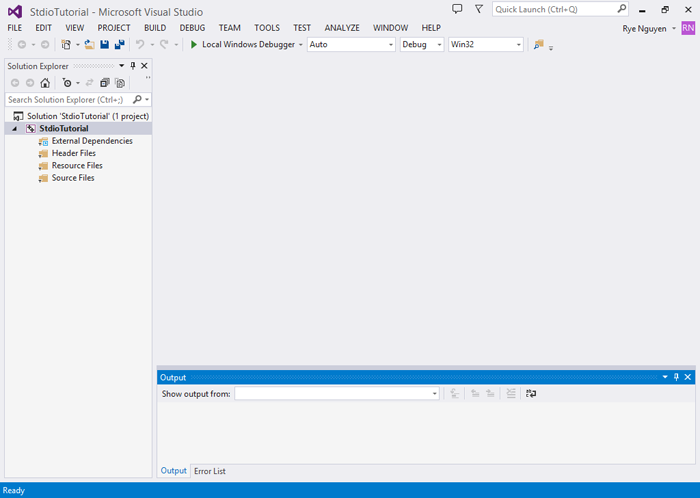
Để tạo một project mới, các bạn có thể làm theo một trong 4 cách sau:

* Vào menu File ➙ New ➙ Project.
* Nhấn tổ hợp phím Ctrl + Shift + N.
* Nhấn vào biểu tượng New trên thanh công cụ
* Nhấn vào dòng New Project ở mục Start trong Start Page.

Một cửa sổ sẽ hiện ra như sau:



Các bạn chọn mục Empty Project, đặt lại tên và đường dẫn cho project, sau đó bấm OK. Chương trình sẽ bắt đầu quá trình khởi tạo project. Sau khi khởi tạo xong, cửa sổ làm việc sẽ có dạng như sau:



Lưu ý là do chúng ta chọn Empty Project, chúng ta sẽ phải tự thực hiện các thao tác như viết hàm main, khai báo một số chỉ thị include, …

Ngoài ra Visual Studio cũng cho phép chúng ta tạo project có dạng Win32 Console Application. Cá nhân tôi không khuyến khích các bạn tạo project theo kiểu này nên sẽ không hướng dẫn chi tiết trong bài viết. Các bạn có thể tự tìm hiểu và rút ra kinh nghiệm cho bản thân.

1. Viết chương trình đầu tiên
2. Tạo file .cpp và viết chương trình

Sau khi tạo xong project, việc các bạn cần làm là tạo một file .cpp mới và bắt đầu viết chương trình.

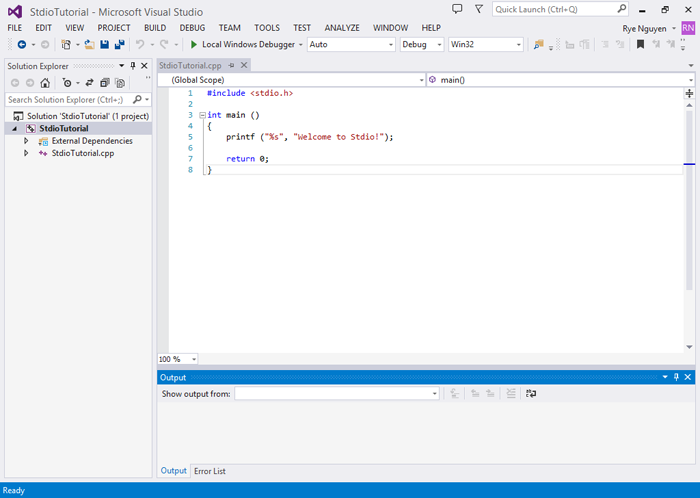
Chúng ta có thể thực hiện theo một trong 2 cách sau:

* Nhấn chuột phải vào tên Project/Filter, chọn Add ➙ New Item.
* Chọn tên Project/Filter thích hợp và nhấn tổ hợp phím Ctrl + Shift + A.

Sau đó chọn C++ File (.cpp), đặt lại tên và nhấn nút Add.

**Lưu ý: New Item sẽ được tạo bên trong Project/Filter được chọn. Do đó bạn nên chọn đúng vị trí cho nó trước khi thêm một file mới. Tuy nhiên bạn cũng có thể sắp xếp lại các file trong project bằng thao tác kéo – thả.**

Tới đây các bạn đã có thể bắt đầu viết chương trình đầu tiên của mình trên Visual Studio.

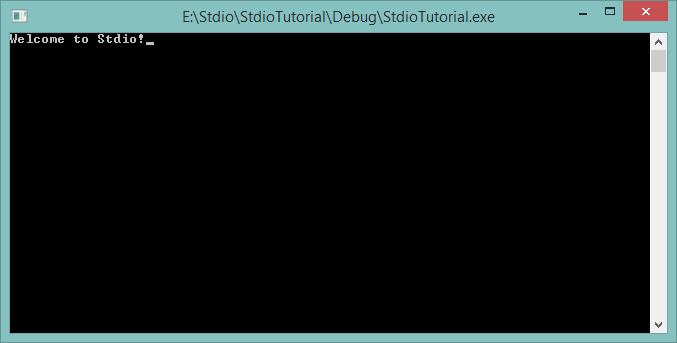


Hoặc các bạn có thể copy đoạn codes bên dưới

1. #include <stdio.h>
3. int main()
4. {
5. printf("%s", "Welcome to Stdio!");
7. return 0;
8. }
9. Build và chạy chương trình

Sau khi viết xong chương trình, các bạn save lại bằng các cách thông thường và tiến hành build chương trình. Các bạn chọn menu Build ➙ Build Solution hoặc nhấn phím F7. Chương trình sẽ kiểm tra lỗi và thông báo ở cửa sổ Output (nếu không hiển thị cửa sổ Output, các bạn chọn trong menu View ➙ Output). Nếu có lỗi xảy ra, sẽ có thông báo tại cửa sổ Output và lỗi sẽ được hiển thị trong Error List (View ➙ Error List).

Khi đã fix xong các lỗi, các bạn bấm tổ hợp phím Ctrl + F5 để chạy chương trình.



##### Từ khóa Name Space

Chắc hẳn trước đây các bạn ít nhiều đã có sử dụng dòng mã using namespace std; khi viết các chương trình Console Application với thư viện iostream. Vậy, ý nghĩa của dòng mã đó là gì, chúng ta sẽ cùng tìm hiểu thông qua bài viết này.

Các ví dụ trong bài viết được kiểm chứng trên môi trường:

* Visual Studio 2012
* Windows
* C++11

1. Namespace là gì?

Namespace là một cơ chế trong C++, cho phép ta nhóm các thực thể (class, object, function…) có liên quan thành từng nhóm khác nhau theo tên, mà theo đó tên của mọi thực thể trong mỗi namespace đều được gắn thêm tên của namespace đó như tiền tố.

1. Namespace dùng để làm gì?

Trong C++, không sử dụng namespace (hay có thể hiểu là dùng chung một namespace) có thể gây ra một số rắc rối. Chẳng hạn trong ví dụ:

1. // lib1.h
2. class String
3. {
4. // some helpful functions
5. };
7. // lib2.h
8. class String
9. {
10. // some others helpful functions
11. };

Giả sử ta cần cả 2 lớp để xử lý một vấn đề liên quan đến chuỗi. Lúc này ta dễ dàng thấy được là không thể sử dụng cả 2 thư viện lib1.h và lib2.h cùng lúc. Do trình biên dịch sẽ báo là lớp String được khai báo lại, hoặc giả trình biên dịch không báo lỗi thì ta cũng không thể phân biệt được 2 lớp String này.

Lúc này, sử dụng namespace sẽ giải quyết được vấn đề trên:

1. // lib1.h
2. namespace lib1{
3. class String {
4. // ...
5. };
6. }
8. // lib2.h
9. namespace lib2{
10. class String {
11. // some others helpful functions
12. };
13. }

Sau khi định nghĩa như vậy, ta có thể gọi các lớp trên bằng toán tử quan hệ (::) như sau:

1. #include <lib1.h>
2. #include <lib2.h>
3. int main(){
4. // do something
5. lib1::String a; // the String class in lib1.h
6. lib2::String b; // the String class in lib2.h
7. // do something
8. return 0;
9. }

Không chỉ giới hạn ở 1 namespace và các class, ta cũng có thể tổ chức các namespace theo dạng cây cho các đối tượng hay hàm, như trong ví dụ sau:

1. namespace stdio
2. {
3. // News
4. string whatsNew;
6. namespace founders {
7. namespace kevin {
8. void submitTopic() {
9. // do something
10. }
11. }
12. // ...
13. }
15. namespace authors {
16. namespace amy {
17. }
19. namespace rye {
20. class BasicGuide {
21. // something
22. };
23. }
25. namespace frederick {
26. }
27. }
28. // ...
29. }

Lúc này ta có thể gọi đến chuỗi News, lớp BasicGuide, hàm submitTopic như sau:

1. stdio::whatsNew;
2. stdio::authors::rye::BasicGuide PointerInCpp;
3. stdio::founders::kevin::submitTopic();

**Lưu ý**

**Một namespace có thể chứa một hoặc nhiều thực thể hay namespace khác, nhưng một thực thể không thể chứa trong nó bất kỳ namespace nào.**

1. Từ khóa Using

Từ khóa using có 2 cách sử dụng, và được gọi với 2 tên khác nhau:

* Using Declaration
* Using Directive

1. Using declaration

Cách gọi:

1. using <tên namespace>::<tên thực thể>;

Công dụng:

Khai báo này cho phép lập trình viên truy cập đến thực thể được khai báo thông qua tên namespace chứa khai báo.

Ví dụ:

Xét ví dụ tại mục 1, không chỉ có [Kevin La](http://www.stdio.vn/users/index/1) có nhu cầu xin đăng bài mà [Rye Nguyễn](http://www.stdio.vn/users/index/74), [Amy Lê](http://www.stdio.vn/users/index/70) hay [Frederick Trần](http://www.stdio.vn/users/index/95) cũng vậy nhưng khi gọi **stdio::founders::kevin::submitTopic();** để đăng một bài của **Amy Lê**thì không tường minh, khi đó, ta có thể chia sẻ quyền xin đăng bài (hàm submitTopic) của **Kevin La**thông qua từ khóa **using** như sau:

1. namespace stdio
2. {
3. // News
4. string whatsNew;
6. namespace founders
7. {
8. namespace kevin
9. {
10. void submitTopic()
11. {
12. // do something
13. }
14. }
15. // ...
16. }
18. namespace authors
19. {
20. namespace amy
21. {
22. // be able to submit topics
23. using stdio::founders::kevin::submitTopic;
24. }
26. namespace rye
27. {
28. // be able to submit topics
29. using stdio::founders::kevin::submitTopic;
30. class BasicGuide
31. {
32. // something
33. };
34. }
36. namespace frederick
37. {
38. // be able to submit topics
39. using stdio::founders::kevin::submitTopic;
40. }
41. }
42. // ...
43. }

Và như vậy ta có thể gọi hàm submitTopic như sau:

1. stdio::authors::amy::submitTopic();
2. stdio::authors::rye::submitTopic();
3. stdio::authors::frederick::submitTopic();
4. Using directive

Cách gọi:

1. using namespace <tên namespace>;

Công dụng:

* **using directive** cho phép người viết mã nguồn gọi các thực thể trong khối namespace được khai báo mà không cần gọi tên namespace đó.
* **using directive** có tác dụng kể từ dòng nó được gọi.
* **using directive** có thể được gọi nhiều lần tại nhiều vị trí khác nhau trong cùng một tập tin mã nguồn, và chúng có hiệu lực song song nhau.

Ví dụ:

Xét ví dụ tại mục 2.1, ta thấy rõ ràng phải gọi rất nhiều tên namespace khi các author gọi hàm submitTopic. để rút ngắn, ta có thể viết như sau:  
using namespace stdio::authors;

1. amy::submitTopic();
2. rye::submitTopic();
3. frederick::submitTopic();

**\*Kevin không nằm trong namespace authors, vậy làm sao gọi hàm submitTopic của Kevin?**

**using directive** là một cơ chế thông minh, nó sẽ không ảnh hưởng đến các dòng mã nguồn “không cần đến nó”. Ta hoàn toàn có thể viết như sau:

1. using namespace stdio::authors;
3. amy::submitTopic();
4. rye::submitTopic();
5. frederick::submitTopic();
6. stdio::founders::kevin::submitTopic();
8. // or
10. using namespace stdio::authors;
11. using namespace stdio::founders;
13. amy::submitTopic();
14. rye::submitTopic();
15. frederick::submitTopic();
17. kevin::submitTopic();
18. using namespace std

std (standard) là một gói các thực thể thường dùng được định nghĩa sẵn trong thư viện iostream thuộc các phiên bản C++ standard để tiện cho lập trình viện khi sử dụng và tránh các lỗi khai báo trùng. Một số thực thể trong std: cout, cin, string, abs, sin, cos, tan, malloc…

1. Lời kết

Vậy là chúng ta đã cùng nhau đi qua một nội dung nữa. Hy vọng rằng với bài viết này, tôi có thể phần nào giúp bạn hiểu hơn về dòng mã using namespace std;, hơn thế, tôi còn hy vọng bạn có thể hiểu và sử dụng namespace, using declaration, using directive thuần thục để nhờ nó, mã nguồn của bạn trông sẽ chuyên nghiệp hơn, và giải quyết được một số vấn đề về khai báo.

### Cú pháp cơ bản C,C++

#### C - Basic Syntax

1. Cấu trú chương trình

\\... Khai báo thư viện

#include <stdio.h>

\\... Khai báo biến toàn cục, hàm

void global\_Variable;

void function(void Parameter);

\\... Hàm chính

int main()

{

/\* my first program in C \*/

printf("Hello, World! \n");

return 0;

}

\\... Định nghĩa hàm

void function(void Parameter);

{ \\... }

1. Phần top của chương trình để khai báo các thư viện dùng trong chương trình. *#include <stdio.h>* sẽ báo cho trình biên dịch biết thư viện sẽ được dùng trước khi biên dịch.
2. Tiếp theo là phần khai báo biến toàn cục, hàm. Phần nay chỉ khai báo chứ không định nghĩa
3. *int main()* là hàm chính, nơi chương trình bắt đầu. Thân hàm nằm trong dấu ngoặc nhọn {}
4. *Cuối cùng là phần định nghĩa các hàm đã khai báo phía trên*
5. Nhận dạng:

C nhận dạng tên biến, hàm, dữ liệu người dùng định nghĩa bắt đầu bằng kí tự từ A tới Z hoặc a tới z hoặc bắt dầu bằng dấu gạch dưới “ \_” và có thể theo sau nhiều kí tự khác, “\_” và số (0 to 9).

C không cho phép dùng các kí tự đặc biệt như @,$ hay %.

C có phân biệt chữ hoa và chữ thường. Ví dụ:

mohd zara abc move\_name a\_123

myname50 \_temp j a23b9 retVal

1. Từ khóa

Được dành riêng cho C, tránh đặt tên biến trùng với từ khóa sau

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

1. Kí tự Whitespace trong C

Nếu 1 dòng chỉ chứa 1 whitespace được xem là dòng trống và trình biên dịch C bỏ qua nó.

Whitespace trong C bao gồm khoảng trắng, tabs, enter và comment.

Tuy nhiên trong vài trường hợp cần có khoảng trắng để trình biên dịch phân biệt các thành phần trong câu lệnh. Ví dụ:

int age;

Ví dụ khác:

fruit = apples + oranges; // get the total fruit

Không nhất thiết phải có khoảng trắng dòng lệnh trên, tuy nhiên cho mục đích dễ đọc hơn.

#### C++ Basic Syntax

1. Cấu trúc chương trình C++

Ví dụ về trình hiển thị hello word

\\... Khai báo thư viện, namespace

#include <iostream>

using namespace std;

\\... Khai báo biến toàn cục, hàm

void global\_Variable;

void function(void Parameter);

{\\... Định nghĩa hàm

}

int main()

{

cout << "Hello World"; // prints Hello World

return 0;

}

* Trên top là phần khai báo các header file cần cho chương trình, như là header **<iostream>** . Dòng **using namespace std;** báo cho trình biên dịch biết chương trình có sử dụng std namespace. Namespaces này được dùng cho C++.
* Tiếp theo là phần khai báo biến toàn cục, hàm nếu có. Trong C++ , chúng ta vừa khai báo vừa định nghĩa hàm ngay. Trong nội hàm, biến có thể khai báo tại đâu cũng được.
* **main() là hàm chính, nơi chương trình bắt đầu.** Thân hàm được khai báo trong dấu ngoặc nhọn {}

1. Nhận dạng trong C++

C nhận dạng tên biến, hàm, dữ liệu người dùng định nghĩa bắt đầu bằng kí tự từ A tới Z hoặc a tới z hoặc bắt dầu bằng dấu gạch dưới “ \_” và có thể theo sau nhiều kí tự khác, “\_” và số (0 to 9).

C không cho phép dùng các kí tự đặc biệt như @,$ hay %.

C có phân biệt chữ hoa và chữ thường. Ví dụ:

mohd zara abc move\_name a\_123

myname50 \_temp j a23b9 retVal

1. Từ khóa trong C++

Được dành riêng cho C++, tránh đặt tên biến trùng với từ khóa sau

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |  |

1. Chuỗi 3 kí tự đại diện Trigraphs:

Một vài kí tự sẽ có một đại diện thay thế , được gọi là một chuỗi trigraph . Một trigraph là một chuỗi gồm 3 kí tự kế tiếp đại diện cho một ký tự đơn và luôn luôn bắt đầu với hai dấu hỏi.

Chuỗi Trigraphs được mở rộng bất cứ nơi nào nó xuất hiện , bao gồm cả xâu chuỗi và xâu kí tự , phần ghi chú ​​, và chỉ thị tiền xử lý .

Sau đây là vài trình tự trigraph hay dùng:

Following are most frequently used trigraph sequences:

|  |  |
| --- | --- |
| **Trigraph** | **Replacement** |
| ??= | # |
| ??/ | \ |
| ??' | ^ |
| ??( | [ |
| ??) | ] |
| ??! | | |
| ??< | { |
| ??> | } |
| ??- | ~ |

Tất cả trình biên dịch mặc định sẽ không hỗ trợ trigraph vì dễ gây rắc rối cho chương trình.

1. Whitespace trong C++:

Nếu 1 dòng chỉ chứa 1 whitespace được xem là dòng trống và trình biên dịch C bỏ qua nó.

Whitespace trong C bao gồm khoảng trắng, tabs, enter và comment.

Tuy nhiên trong vài trường hợp cần có khoảng trắng để trình biên dịch phân biệt các thành phần trong câu lệnh. Ví dụ:

int age;

Ví dụ khác:

fruit = apples + oranges; // Get the total fruit

Không nhất thiết phải có khoảng trắng dòng lệnh trên, tuy nhiên cho mục đích dễ đọc hơn.

### Tổ chức bộ nhớ:

##### Little Endian Và Big Endian:

1. Giới thiệu

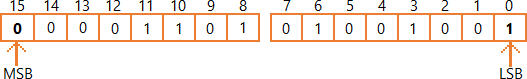
Cũng như con người, máy tính cũng "nói" các ngôn ngữ khác nhau. Một số “đọc” và “ghi” từ trái sang phải, trong khi một số khác thì ngược lại. Một bộ máy có thể đọc và xử lý dữ liệu được tạo ra bởi nó một cách bình thường. Vấn đề xảy ra khi một bộ máy “khác loại” cố gắng đọc dữ liệu đó. Thuật ngữ “big endian” và “little endian” diễn tả sự khác nhau về cách đọc và ghi dữ liệu giữa các nền tảng máy tính. Bài viết sẽ cung cấp cho các bạn một số kiến thức về big và little endian.

Endian là cách tổ chức dữ liệu trên một nền tảng máy tính. Bộ nhớ máy tính có thể được xem như một mảng có kích thước lớn, chia làm nhiều ô, mỗi ô có kích thước 1 byte. Nếu dữ liệu của bạn có thể được gói gọn trong 1 byte, nghĩa là mỗi lần bạn chỉ xử lý 1 byte thì không có gì khác biệt.

Vấn đề xảy ra khi dữ liệu của bạn vượt quá 1 byte, ví dụ như bạn cần lưu trữ một số nguyên. Khi đó 4 byte của biến số nguyên đó sẽ được lưu vào bộ nhớ máy tính theo thứ tự nào?

1. Big Endian

Đối với các nền tảng dùng big endian, Least Significant bit (LSB) luôn được lưu ở ô nhớ có địa chỉ lớn nhất còn Most Significant Bit (MSB) được lưu ở ô nhớ có địa chỉ nhỏ nhất trong vùng lưu trữ của biến. Trong đó, LSB là bit có trọng số nhỏ nhất, nằm ở ngoài cùng bên phải; MSB là bit có trọng số lớn nhất, nằm ở ngoài cùng bên trái của một biến.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_2/10/ss_1.png)

Lưu trữ số nguyên 7411 vào bộ nhớ máy trên nền tảng dùng big endian. Ta cùng xét ví dụ sau:

Số 7411 được viết dưới dạng nhị phân như sau:

1 1100 1111 0011

Minh họa việc lưu số 7411 vào bộ nhớ máy tính như sau:

[http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_2/10/ss_2.png](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_2/10/ss_2.png)

**Các nền tảng sử dụng Big Endian**

IBM zSeries, iSeries, AIX, NonStop Kernel, Motorola 68K…

1. Little Endian

Ngược lại với các nền tảng sử dụng big endian, ở các nền tảng sử dụng little endian, LSB luôn được lưu ở ô nhớ có địa chỉ nhỏ nhất còn MSB được lưu ở ô nhớ có địa chỉ lớn nhất trong vùng lưu trữ của biến.

Xét lại ví dụ trên, việc lưu trữ số nguyên 7411 vào bộ nhớ máy tính được minh họa như sau:

[http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_2/10/ss_3.png](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_2/10/ss_3.png)

**Các nền tảng sử dụng Little Endian**

Hầu hết các hệ thống sử dụng vi xử lý của Intel.

1. Ưu điểm của Big Endian và Little Endian

Big endian và little endian đều do con người sáng tạo nên và đều không gây ảnh hưởng đến tốc độ xử lý của CPU. Do đó sẽ không có câu trả lời hợp lý cho câu hỏi: cái nào lợi thế hơn cái nào?

Với big endian, do dữ liệu được ghi từ trái qua phải, bạn sẽ dễ dàng kiểm tra một số là số dương hay số âm do bit dấu được lưu trữ ở bit đầu tiên của biến. Ta không cần biết số đó chiếm bao nhiêu byte, cũng như không cần nhảy qua byte nào để xác định byte lưu trữ dấu của số đó. Các số cũng được lưu trữ đúng như trình tự mà nó được hiển thị, do đó sẽ hiệu quả trong phép toán đổi từ nhị phân sang thập phân.

Với little endian, một biến có thể được đọc với độ dài bất kì mà không cần phải thay đổi memory address. Do đó sẽ thuận tiện hơn trong các phép toán ép kiểu, chuyển kiểu, dễ dàng viết các phép toán phức tạp …

1. Chuyển đổi giữa Big Edian và Little Endian

Qua ví dụ ở trên, các bạn đã thấy được sự khác nhau giữa big endian và little endian. Điều này rất quan trọng trong lập trình đa nền tảng, vì các chương trình được viết ở nền tảng sử dụng big endian thì không thể sử dụng được các nền tảng khác sử dụng little endian. Muốn sử dụng được trước hết ta phải đồng bộ hai nền tảng bằng cách xem xét và đảo lại vị trí các byte cho phù hợp với hệ thống mới.

Codes C++ chuyển đổi qua lại giữa Little Endian và Big Endian.

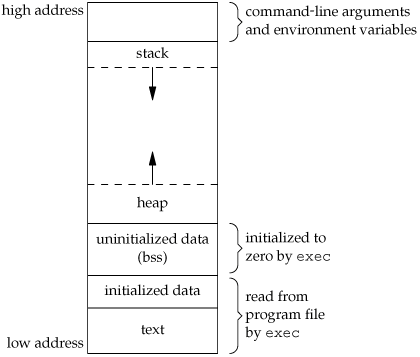
1. // 2 bytes
2. inline void EndianSwap (unsigned short& x)
3. {
4. x = (x >> 8) | (x << 8);
5. }
7. // 4 bytes
8. inline void EndianSwap (unsigned int& x)
9. {
10. x = (x >> 24) |
11. ((x << 8) & 0x00FF0000) |
12. ((x >> 8) & 0x0000FF00) |
13. (x << 24);
14. }
16. // 8 bytes
17. inline void EndianSwap (unsigned \_\_int64& x)
18. {
19. x = (x >> 56) |
20. ((x << 40) & 0x00FF000000000000) |
21. ((x << 24) & 0x0000FF0000000000) |
22. ((x << 8) & 0x000000FF00000000) |
23. ((x >> 8) & 0x00000000FF000000) |
24. ((x >> 24) & 0x0000000000FF0000) |
25. ((x >> 40) & 0x000000000000FF00) |
26. (x << 56);
27. }

##### Các vùng nhớ

1. Kiến thức mở đầu

Khi ta khai báo một biến trong C,C++, thì biến này gồm 2 phần là địa chỉ và giá trị của nó. Không chỉ có biến, mà các đoạn mã nguồn cũng có địa chỉ và giá trị. Có bao giờ bạn thắc mắc rằng biến hay mã nguồn được lưu trữ như thế nào khi chương trình thực thi?

Khi chương trình được khởi chạy, hệ điều hành sẽ chọn ra 4 vùng nhớ trống trong RAM có kích thước tương ứng với kích thước của 5 vùng nhớ mà chương trình cần sử dụng, mà ta thường gọi là Memory Segment. Biến và mã nguồn sẽ được lưu trữ trong các Segment này, địa chỉ của chúng chính là địa chỉ trong RAM.



1. Text Segment

Mã nguồn của chúng ta sau khi hoàn tất quá trình Build sẽ chuyển thành các đoạn mã máy 0 và 1. Khi chương trình khởi chạy, các đoạn mã máy này sẽ được nạp vào Text Segment. Cách duy nhất để truy xuất vào vùng nhớ này là sử dụng Con trỏ hàm.

Text Segment có kích thước cố định, đặt dưới heap hay stack để tránh heap hay stack tràn bộ đệm có thể ghi đè lên nó.

Text Segment có thể chia sẻ, thường là dạng read-only để tránh chương trình vô ý tùy chỉnh code của nó.

1. Initialized Data Segment

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char\* string = “hello world” makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment.

1. Uninitialized Data Segment

Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.

For instance a global variable declared int j; would be contained in the BSS segment.

1. Stack Segment

Khi một hàm được gọi, hàm đó và biến khai báo cục bộ của hàm sẽ được đưa vào vùng nhớ Stack, Khi hàm kết thúc, toàn bộ các biến trong hàm cùng với bản thân hàm sẽ được tự động giải phóng để các hàm sau sử dụng.

Trường hợp ngoại lệ như vùng nhớ cấp phát động hay biến tĩnh (static) không bị hủy.

 **Stack Segment có kích thước cố định. Khi chúng ta khai báo quá nhiều biến hoặc một mảng có số lượng phần tử quá lớn, một hàm đệ quy vô hạn… bộ nhớ Stack sẽ bị đầy, dẫn đến chương trình bị dừng. Chúng ta cần lưu ý chuyện này.**

1. Cấu trúc dữ liệu stack

Stack vận hành một cấu trúc dữ liệu

Cấu trúc dữ liệu là một cơ chế lập trình cho dữ liệu có tổ chức. Cấu trúc dữ liệu cung cấp khả năng lưu trữ và truy xuất hiệu quả. Có rất nhiều kiểu cấu trúc trong lập trình, một vài trong số đó được khai báo trong thư viện chuẩn, và stack nằm trong số này.

Xem xét một stack các đĩa đựng đồ ăn trên kệ. Mỗi chiếc đĩa được xếp chồng lên nhau, bạn thực sự chỉ làm được một trong các thao tác sau:

* Nhìn bề mặt chiếc đĩa trên cùng
* Lấy chiếc đĩa trên cùng ra
* Đặt chĩa đĩa khác lên trên cùng

Trong lập trình máy tính, stack là một cấu trúc dữ liệu lưu trữ các biến. Tuy nhiên, không như mảng, vốn cho phép bạn truy xuất và thay đổi giá trị bất kì phần tử nào bạn muốn (truy xuất ngẫu nhiên), stack bị giới hạn khả năng này.

Các thao tác trên stack tương ứng với 3 thứ đã nói ở trên:

* Truy xuất dữ liệu ở trên cùng (thường thực hiện qua hàm gọi **top**(), thỉnh thoảng dùng **peek**() )
* Rút dữ liệu trên cùng ra (hàm **pop**() )
* Đặt dữ liệu mới lên trên (hàm **push**() )

Stack là cấu trúc LIFO (last-in, first-out). Dữ liệu xếp vào cuối cùng sẽ được lấy ra trước. Nếu bạn đặt cái đĩa mới lên đỉnh stack, stack trở nên lớn hơn – khi bạn lấy ra, stack nhỏ đi.

Thử hình dung các thao tác push() và pop() trên stack hoạt động ra sao:

Stack: trống

Push 1

Stack: 1

Push 2

Stack: 1 2

Push 3

Stack: 1 2 3

Pop

Stack: 1 2

Pop

Stack: 1

Ví dụ thứ 2 là về các thư xếp chồng lên nhau. Mỗi thư neo cố định vào thư bên dưới nó, số lượng thư trong hộp thư là cố định, không được thay đổi.

Đầu tiên, chúng ta dùng miếng note dán vào thư ở đáy hòm. Vào lúc bắt đầu, các thư đều trống rỗng. Ta đặt nội dung vào thư đang được đánh dấu và dán miếng note lên chiếc thư bên trên. Khi ta lấy nội dung thư ra, ta cũng dán miếng note xuống thư và lấy nội dung ra.

Mọi thứ nằm dưới miếng note được xem là “đang trên stack”. Và tất cả những gì từ miếng note và ở phía trên gọi là nằm ngoài stack.

1. Vùng nhớ stack

Vùng nhớ stack quản lý vùng nhớ dùng cho stack. Khi ứng dụng bắt đầu, hàm main() được đẩy vào stack bởi hệ điều hành. Sau đó chương trình bắt đầu chạy.

Khi hàm gọi được gọi, hàm được đẩy vào stack. Khi hàm hiện hành kết thúc, nó được rút khỏi stack. Do đó, thông qua hàm vừa đẩy vào stack, chúng ta có thể xem tất cả hàm được gọi để biết chương trình đang thực thi tới đâu.

Stack là một chuỗi các địa chỉ vùng nhớ cố định. Hộp thư bên trên chính là địa chỉ vùng nhớ, các nội dung thư được gọi là các khung stack.

Một khung stack quản lý dữ liệu tương ứng với một hàm gọi.

Miếng note đánh dấu là con trỏ stack, có kiểu thanh ghi (register), vùng nhớ của CPU. Con trỏ stack cho biết đỉnh của stack đang ở đâu.

Khi đẩy một dữ liệu ra khỏi stack, chúng ta không phải xóa vùng nhớ, cứ để dữ liệu tiếp theo ghi đè lên nó.

1. Thao tác trên stack

Trước tiên ta nói thêm về cách hoạt động của stack khi hàm gọi được gọi:

1. Chương trình gọi hàm.
2. Một khung stack được tạo ra và có cấu trúc như sau:

* Địa chỉ của chỉ thị trước lệnh gọi hàm (gọi là địa chỉ trả về). Để khi hàm gọi trả về, CPU biết nó đang thực thi tới đâu
* Tất cả đối số hàm được đẩy vào stack
* Tất cả biến cục bộ hàm được đẩy vào stack
* Lưu trữ bản sao trạng thái của các thanh ghi CPU bị làm thay đổi bởi hàm, để khôi phục lại khi hàm trả về.

1. CPU nhảy tới điểm bắt đầu của hàm
2. Thực thi chỉ thị xử lí bên trong hàm

Khi hàm kết thúc, các bước sau sẽ xảy ra:

1. Các thanh ghi khôi phục lại trạng thái từ stack
2. Khung stack bị đẩy ra khỏi stack. Hủy tất cả biến cục bộ và đối số hàm.
3. Xử lí giá trị trả về
4. CPU tiếp tục thực thi chương trình tại địa chỉ trả về.

Giá trị trả về được xử lí theo nhiều cách khác nhau, tùy thuộc vào kiến trúc máy tính. Một số kiến trúc sẽ gộp giá trị trả về như một phần của khung stack. Một số thì dùng thanh ghi CPU.

Việc hiểu rõ stack hoạt động ra sao không quan trọng lắm. Tuy nhiên, việc hiểu rõ hàm gọi được đẩy vào stack khi chúng được gọi và đẩy ra khi kết thúc cho bạn nắm lý thuyết cơ bản về đệ quy, cũng như khi debug.

1. Ví dụ nhanh về việc gọi stack

int foo(int x){

// b

return x;

} // foo is popped off the call stack here

int main(){

// a

foo(5); // foo is pushed on the call stack here

// c

return 0;

}

Gọi stack diễn tiến như sau:

a:

main()

b:

foo() (bao gồm tham số x)

main()

c:

main()

1. Stack overflow

Stack có kích thước giới hạn, nên lưu trữ được ít thông tin. Trên Window OS, stack mặc định chỉ có 1MB. Trên vài máy Unix, nó có thể lớn hơn 8MB.

Nếu chương trình đẩy quá nhiều thông tin lên stack, stack sẽ bị tràn vùng nhớ. Trong trường hợp này, sự cấp phát bắt đầu tràn sang vùng nhớ của thành phần khác.

Tràn stack nói kỹ hơn là kết quả của việc cấp phát cho quá nhiều biến, hay quá nhiều hàm gọi chồng chéo lên nhau. Khiến chương trình bị sụp.

Ví dụ:

int main(){

int stack[100000000]; // tương đương 190 MB

return 0;

}

Chương trình đang cố cấp phát một mảng khổng lồ, gây tràn stack và sụp chương trình.

Một ví dụ về lý do khác khiến tràn stack:

void foo(){

foo();

}

int main(){

foo();

return 0;

}

Khung stack được tạo ra liên tiếp và đẩy vào stack, vì hàm **foo**() gọi vô hạn cho tới khi đầy vùng nhớ stack.

Những thuận lợi và bất lợi của stack:

* Cấp phát vùng nhớ cực nhanh
* Vùng nhớ cấp phát trên stack tồn tại tới khi nào ra ngoài phạm vi của hàm gọi, nó sẽ được tự hủy vùng nhớ.
* Tất cả vùng nhớ cấp phát trên stack được dự báo tại thời điểm biên dịch. Do đó vùng nhớ này có thể được truy xuất trực tiếp qua biến.
* Vì stack khá nhỏ, nên không phải nơi lý tưởng để lưu trữ cấu trúc dữ liệu lớn. Bạn nên cấp phát động sang vùng nhớ Heap thay thế.

1. Heap Segment

Vùng nhớ này chúng ta phải hoàn toàn kiểm soát nó. Khi chúng ta sử dụng con trỏ và cấp phát động một vùng nhớ cho con trỏ quản lý, vùng nhớ này sẽ nằm trong Heap Segment. Những vùng nhớ được cấp phát động sẽ không tự động thu hồi khi khối lệnh kết thúc, lập trình viên phải chủ động thu hồi chúng khi không còn nhu cầu sử dụng.

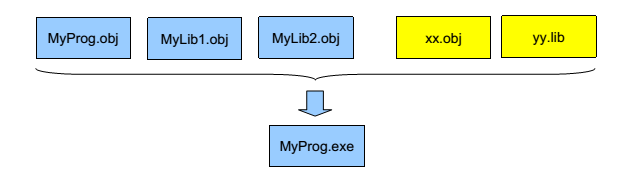
Heap Segment có kích thước không cố định, nên nó còn được gọi là vùng nhớ động, kích thước của nó có thể tăng hoặc giảm tùy vào sự cấp phát động.

**Lưu ý là Heap có thể mở rộng cho đến khi RAM đầy, nên chúng ta cần kiểm soát thật tốt, thu hồi các vùng nhớ được cấp phát động ngay khi không còn nhu cầu sử dụng để tránh việc lãng phí bộ nhớ.**

Những thuận lợi và bất lợi của bộ nhớ heap:

* Cấp phát vùng nhớ trên heap khá chậm
* Vùng nhớ cấp phát tồn tại tới khi bị thu hồi hay khi chương trình kết thúc, hệ điều hành sẽ tự dọn dẹp.
* Cấp phát động phải truy xuất qua con trỏ. Truy xuất nội dung con trỏ chậm hơn là trực tiếp qua biến
* Vì bộ nhớ heap rất lớn, rất tốt để cấu trúc, lớp, mảng lưu trú tại đây.

### Liên kết (Link):



Liên kết là quá trình ghép nhiều file ₫ích (\*.obj, \*.lib) ₫ể tạo ra chương trình chạy cuối cùng \*.exe

Lỗi liên kết có thể là do:

* Sử dụng hàm nhưng không có ₫ịnhnghĩa hàm
* Biến hoặc hàm ₫ược ₫ịnh nghĩa nhiều lần
* ….

##### Chia chương trình thành nhiều file

1. Header là gì

Khi chương trình trở nên lớn hơn, bạn muốn các thủ tục khai báo và định nghĩa hàm, lớp.. được sử dụng trong nhiều file. Cách tốt nhất là đặt hết chúng trong một file.

File “.h” được gọi là header file, thỉnh thoảng ta thấy “.hpp” file hay file không có phần mở rộng. Mục đích của header là giữ khai báo cho các file khác sử dụng.

Không bao giờ khai báo biến và định nghĩa hàm trong file header trừ khi chúng ta hằng. Header chỉ chứa thủ tục khai báo mà thôi

Mỗi file header nên là một chức năng riêng, độc lập. Tên file header và file source (cpp) phải giống nhau.

Cố gắng tối thiểu #include thư viện trong file source bằng cách khai báo trong header file. Tuyệt đối không #include .cpp file

1. Khai báo thư viện hàm chuẩn

Xem chương trình sau:

#include <iostream>

int main(){

using namespace std;

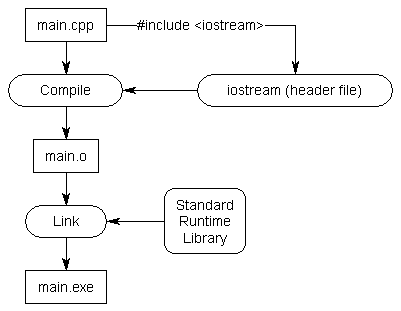
cout << "Hello, world!" << endl;

return 0;

}

Chương trình xuất câu chào "Hello, world!" có dùng cout. Cout được khai báo trong file header <iostream>. Khi ta gõ câu lệnh #include <iostream>, là đã yêu cầu nạp nội dung từ file header tên là “iostream” vào file hiện tại. Cho phép các khai báo được dùng trong đoạn code này.

Lưu ý là header file chỉ giữ khai báo, còn định nghĩa được giữ trong thư viện runtime C++, vốn được tự động liên kết với chương trình trong quá trình **LINK.**



1. Tạo header file

Chúng ta tạo hai file là add.cpp và main.cpp:

add.cpp:

int add(int x, int y){ // định nghĩa hàm

return x + y;

}

main.cpp:

#include <iostream>

int add(int x, int y); // thủ tục khai báo

int main(){

using namespace std;

cout << "The sum of 3 and 4 is " << add(3, 4) << endl;

return 0;

}

Nhớ thêm add.cpp vào project nhé. Chúng ta phải khai báo hàm add() để trình biên dịch biết khi nó biên dịch main.cpp.

Khi bạn có hàng tá hàm với dòng lệnh dài lê thê, hãy kéo chúng sang một file .cpp để định nghĩa riêng biệt. Đặt các khai báo hàm trong file header, chỉ cần khai báo một lần thôi.

Header file gồm hai phần:

* Header guard: đảm bảo header file chỉ #include một lần trong cùng một file
* Nội dung header: gồm các thủ tục khai báo và thư viện liên quan.

add.h:

// This is start of the header guard. ADD\_H can be any unique name. By convention, we use the name of the header file.

#ifndef ADD\_H

#define ADD\_H

// This is the content of the .h file, which is where the declarations go

int add(int x, int y); // thủ tục hàm có dấu phẩy cuối dòng!

// This is the end of the header guard

#endif

Vậy làm sao trình biên dịch liên kết chúng lại? Cách làm như sau:

#include header file trong main.cpp

include <iostream>

#include "add.h"

int main(){

using namespace std;

cout << "The sum of 3 and 4 is " << add(3, 4) << endl;

return 0;

}

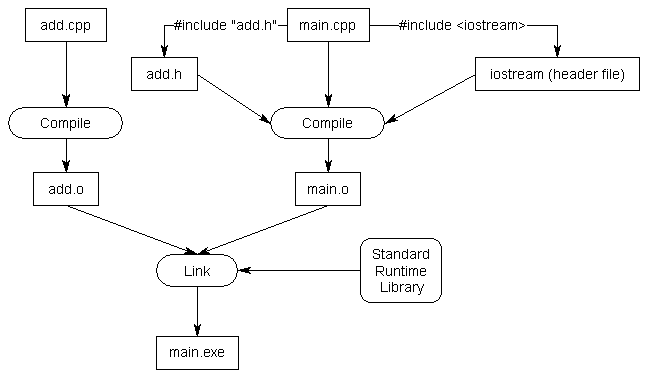
add.cpp như cũ:

int add(int x, int y){

return x + y;

}

Sơ đồ biên dịch:



Add.cpp và main.cpp được biên dịch riêng, sau đó nhờ quá trình LINK mà trình biên dịch tìm được định nghĩa add() trong add.o



Lưu ý: khi bạn #include một file, toàn bộ nội dung sẽ được chèn vào vị trí #include. Nhớ gõ chính xác tên header file và file định nghĩa .cpp phải nằm trong project.

1. <header> và “header.h”

You’re probably curious why we use angled brackets for iostream, and double quotes for add.h. The answer is that angled brackets are used to tell the compiler that we are including a header file that was included with the compiler, so it should look for that header file in the system directories. The double-quotes tell the compiler that this is a header file we are supplying, so it should look for that header file in the current directory containing our source code first. If it doesn’t find the header file there, it will check any other include paths that you’ve specified as part of your compiler/IDE settings. That failing, it will fall back to checking the system directories.

Rule: Use angled brackets to include header files that come with the compiler. Use double quotes to include any other header files.

Why doesn’t iostream have a .h extension?

Another commonly asked question is “why doesn’t iostream (or any of the other standard library header files) have a .h extension?”. The answer is, because iostream.h is a different header file than iostream is! To explain requires a very short history lesson.

When C++ was first created, all of the files in the standard runtime library ended in .h. Life was consistent, and it was good. The original version of cout and cin lived in iostream.h. When the language was standardized by the ANSI committee, they decided to move all of the functions in the runtime library into the std namespace (which is generally a good idea). However, this presented a problem: if they moved all the functions into the std namespace, none of the old programs would work any more!

To try to get around this issue and provide backwards compatibility for older programs, a new set of header files was introduced that use the same names but lack the .h extension. These new header files have all their functionality inside the std namespace. This way, older programs that include #include <iostream.h> do not need to be rewritten, and newer programs can #include <iostream>.

When you include a header file from the standard library, make sure you use the non .h version if it exists. Otherwise you will be using a deprecated version of the header that is no longer supported.

As a side note, many headers in the standard library do not have a non .h version, only a .h version. For these files, it is fine to include the .h version. Many of these libraries are backwards compatible with standard C programming, and C does not support namespaces. Consequently, the functionality of these libraries will not be accessed through the std namespace. Also, when you write your own header files, you should give them all a .h extension, since you will not be putting your code in the std namespace.

Rule: use the non .h version of a library if it exists, and access the functionality through the std namespace. If the non .h version does not exist, or you are creating your own headers, use the .h version

Including header files from other directories

Another common question involves how to include header files from other directories.

One (bad) way to do this is to include a relative path to the header file you want to include as part of the #include line. For example:

#include "headers/myHeader.h"

#include "../moreHeaders/myOtherHeader.h"

The downside of this approach is that it requires you to reflect your directory structure in your code. If you ever update your directory structure, your code won’t work any more.

A better method is to tell your compiler or IDE that you have a bunch of header files in some other location, so that it will look there when it can’t find them in the current directory. This can generally be done by setting an “include path” or “search directory” in your IDE project settings.

In Visual Studio, you can right click on your project in the Solution Explorer, and choose “Properties”, then the “VC++ Directories” tab. From here, you will see a line called “Include Directories”. Add your include directories there.

In Code::Blocks, go to the Project menu and select “Build Options”, then the “Search directories” tab. Add your include directories there.

Using g++, you can use the -I option to specify an alternate include directory.

g++ -o main -I /source/includes main.cpp

The nice thing about this approach is that if you ever change your directory structure, you only have to change a single compiler or IDE setting instead of every code file.

Can I put function definitions in a header file?

C++ won’t complain if you do, but generally speaking, you shouldn’t.

As noted above, when you #include a file, the entire content of the included file is inserted at the point of inclusion. This means that any definitions you put in your header get copied into every file that includes that header.

For small projects, this isn’t likely to be much of an issue. But for larger projects, this can make things take much longer to compile (as the same code gets recompiled each time it is encountered) and could significantly bloat the size of your executable. If you make a change to a definition in a code file, only that .cpp file needs to be recompiled. If you make a change to a definition in a header file, every code file that includes the header needs to be recompiled. One small change can cause you to have to recompile your entire project!

Sometimes exceptions are made for trivial functions that are unlikely to change (e.g. where the function definition is one line).

Header file best practices

* Here are a few best practices for creating your own header files
* Always include header guards.
* Do not define variables in header files unless they are constants. Header files should generally only be used for declarations.
* Do not define functions in header files.
* Each header file should have a specific job, and be as independent as possible. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h. That way if you only care about A later, you can just include A.h and not get any of the stuff related to B.
* Give your header files the same name as the source files they’re associated with (e.g. grades.h goes with grades.cpp).
* Try to minimize the number of other header files you #include in your header files. Only #include what is necessary.

Do not #include .cpp files..

##### Static Link Library Và Dynamic Link Library

Giai đoạn liên kết Linking trong quá trình thực thi một chương trình C/C++ là giai đoạn mà Linker sẽ nhúng, kết hợp các file mã nhị phân file .obj được biên dịch và các thư viện bên ngoài như thư viện liên kết tĩnh (.lib), thư viện liên kết động (.lib, .dll). Bài viết này sẽ phân tích về những thư viện đó cũng như cách tạo ra các thư viện đó trong C++ như thế nào

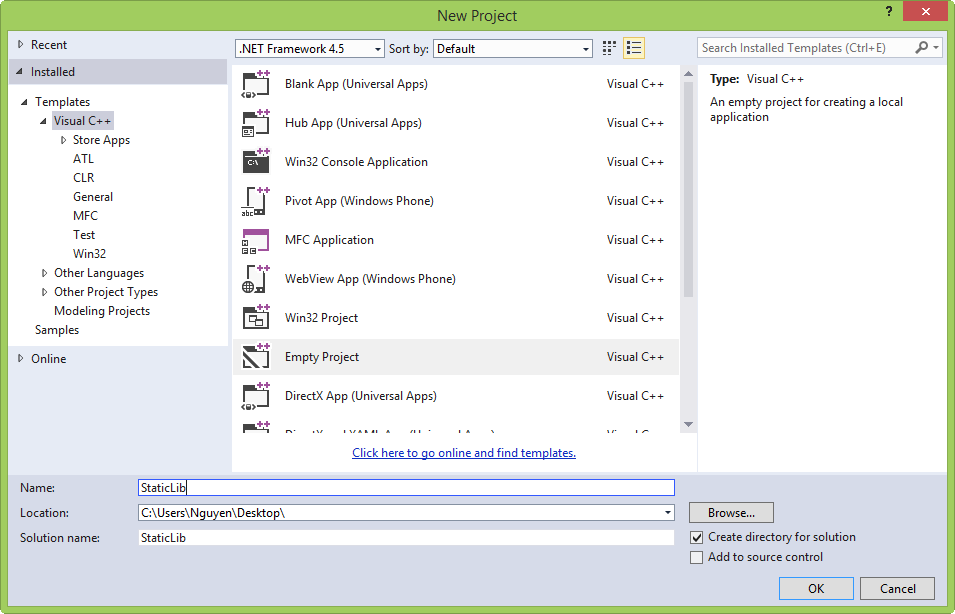
Với nhiều project, tôi phải sử dụng nhiều đoạn mã nguồn giống nhau trong những project khác nhau. Giải pháp đặt ra là có cách giải quyết nào hay không thay vì copy và paste mã nguồn. Thì thư viện liên kết động hay thư viện liên kết tĩnh chính là giải pháp tốt nhất. Nó còn giúp tôi giải quyết được vấn đề thời gian biên dịch quá lâu do thường các project tôi làm rất nhiều file.

1. Thư viện liên kết tĩnh

Thư viện liên kết tĩnh có phần mở rộng là .lib. File này chứa mã nhị phân của mã nguồn sau khi được build.

1. Tạo thư viện liên kết tĩnh

Mở Visual Studio 2013 để tạo một **Empty Project** bằng cách vào **File > New > Project**, sau đó chọn Empty Project, đặt tên và nhấn OK để tạo Project.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/279/ss_1.png)

Tạo 1 thư viện toán học gồm 2 file. StdioMath.h chứa prototype gồm các hàm toán học cơ bản Add,Sub, Mul, Div và StdioMath.cpp chứa mã nguồn định nghĩa.

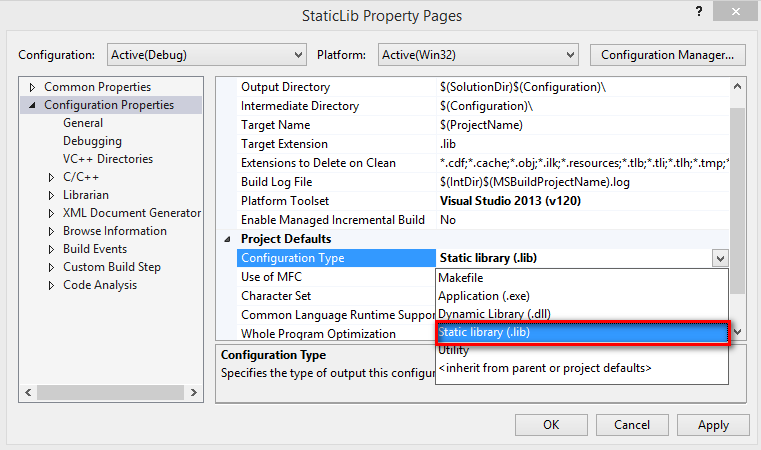
File StidoMath.h

1. int Add(int, int);
2. int Sub(int, int);
3. int Mul(int, int);
4. int Div(int, int);

Và file StdioMath.cpp

1. #include "StdioMath.h"
2. int Add(int a, int b)
3. {
4. return a + b;
5. }
7. int Sub(int a, int b)
8. {
9. return a - b;
10. }
12. int Mul(int a, int b)
13. {
14. return a \* b;
15. }
17. int Div(int a, int b)
18. {
19. return a / b;
20. }

Mở cửa sổ **Properties** bằng cách chuột phải vào tên project và chọn **Properties**. Mục **Configuration Properties/General** chọn **Configuration Type**, sau đó chọn kiểu ứng dụng mà chúng ta muốn build. Ở đây ta sẽ chọn là **Static library (.lib)**.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/279/ss_2.png)

Build Project để thấy kết quả.

Vào thư mục Debug chúng ta sẽ thấy file StaticLib.lib được tạo ra sau khi build project.

1. Thư viện liên kết động

Thư viện liên kết động gồm có 2 file:

* File .lib chứa các thông tin cần thiết để hệ điều hành nạp thư viện dll và xác đinh các hàm export trong dll.
* File .dll chứa chứa mã nhị phân của mã nguồn.

1. Tạo thư viện liên kết động

Tương tự như tạo thư viện liên kết tĩnh tôi tạo một project khác có tên là DynamicLib. Project này cũng có hai file là StdioMath.h và StdioMath.cpp được viết như sau:

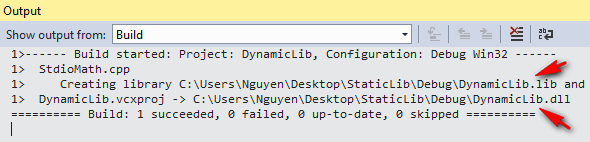
File StdioMath.h

1. #define Dllexport \_declspec(dllexport)
2. extern "C"
3. {
4. Dllexport int Add(int, int);
5. Dllexport int Sub(int, int);
6. Dllexport int Mul(int, int);
7. Dllexport int Div(int, int);
8. }

File StdioMath.cpp

1. #include "StdioMath.h"
3. int Add(int a, int b)
4. {
5. return a + b;
6. }
8. int Sub(int a, int b)
9. {
10. return a - b;
11. }
13. int Mul(int a, int b)
14. {
15. return a \* b;
16. }
18. int Div(int a, int b)
19. {
20. return a / b;
21. }

**Configuration Type**chọn Dynamic Library (.dll). Sau đó build project để thấy kết quả.

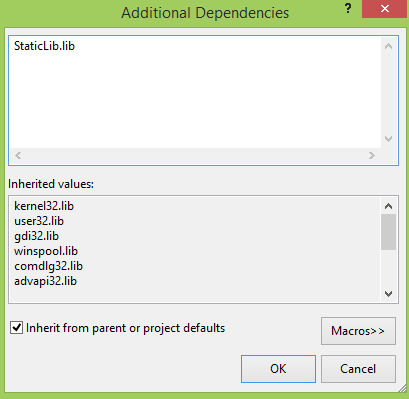
[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/279/ss_5.png)

Mở thư mục Debug sẽ thấy hai file StaticLib.lib và DynamicLib.dll

1. Sử dụng thư viện liên kết
2. Sử dụng thư viện liên kết tĩnh

Tạo một project khác với tên là TestStaticLib. Copy 2 File StdioMath.h và thư viện StaticLib.lib được build ở project StaticLib vào project TestStaticLib.

Chỉ thị cho trình biên dịch là chúng ta cần link tới file StaticLib.lib  bằng cách mở hộp thoại Properties.  Mục **Configuration Properties/Linker/Input**chọn **Additional Dependences** và gõ vào như sau:

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/279/ss_6.png)

Hoặc có thể sử dụng chỉ thị ở đầu file chương trình.

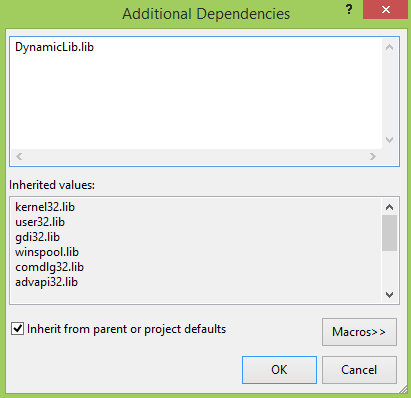
1. #pragma comment(lib, "StaticLib.lib")

Sử dụng thư viện bằng cách include StidoMath.h và dùng nhưng những hàm thông thường.

1. #include "StdioMath.h"
3. int main()
4. {
5. int a = Add(5, 7);
6. int b = Sub(7, 8);
7. int c = Mul(7, 8);
8. int d = Div(7, 5);
9. return 0;
10. }
11. Sử dụng thư viện liên kết động

Tạo một project khác với tên là TestDynamicLib. Copy 2 File StdioMath.h và thư viện StaticLib.lib được build ở project  DynamicLib vào project TestDynamicLib.

Chỉ thị cho trình biên dịch là chúng ta cần link tới file DynamicLib.lib  bằng cách mở hộp thoại Properties.  Mục **Configuration Properties/Input**chọn **Additional Dependences** và gõ vào như sau:

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/279/ss_7.png)

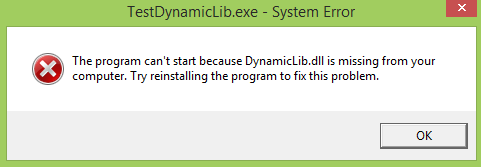
Hoặc có thể sử dụng chỉ thị ở đầu file chương trình.

1. #pragma comment(lib, "StaticLib.lib")

Sử dụng thư viện tương tự đối với thư viện liên kết tĩnh.

1. #include "StdioMath.h"
3. int main()
4. {
5. int a = Add(5, 7);
6. int b = Sub(7, 8);
7. int c = Mul(7, 8);
8. int d = Div(7, 5);
9. return 0;
10. }

Build và chạy chương trình. Quá trình chạy xảy ra lỗi thiếu dll

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/279/ss_8.png)

Thì copy file DynamicLib.dll build từ project DynamicLib sang thư mục debug của project TestDynamicLib.

### Qui tắc soạn thảo mã nguồn:

Coding Convention là một chuẩn mực và quy tắc để khi code ta sẽ phải tuân theo. Việc áp dụng các quy tắc khi code tối thiểu sẽ giúp cho chương trình chúng ta dễ đọc, có quy tắc hơn, đẹp hơn (nó cũng sẽ tương tự như quy tắc soạn văn bản cần có quy tắc canh lề hoặc quy tắc chữ đậm, chữ nhạt).

Xa hơn nữa, với các dự án lớn, việc đòi hỏi đồng bộ giữa các thành viên với nhau là cần thiết, các chuẩn mực sẽ giúp cho các lập trình viên cùng có cách hiểu, cùng nhìn về một hướng, tối ưu thời gian đọc code.

Trong bài viết này, tôi đưa ra một chuẩn mà Stdio đã áp dụng cho thời kỳ đầu tiên trong dự án [Sins](http://sins.stdio.vn/). Tôi không đưa ra bản Coding Convention với mức độ cao nhất vì như vậy sẽ làm các bạn bối rối hơn khi mới tiếp cận.

Mọi kiến thức đều sinh ra do quan điểm, mục tiêu và cách nhìn nhận vấn đề của mỗi người, mỗi tổ chức, Coding Convention của Stdio cũng vậy, do đó, tùy quan điểm, mục tiêu mà bạn có thể sinh ra một chuẩn riêng cho bản thân hoặc áp dụng thử phương pháp này.

Stdio Coding Convention dành cho một số ngôn ngữ quen thuộc như: C, C++, C#, Java.

1. Cách đặt tên biến phải mang nghĩa nhất định

Ta xét ví dụ

1. ConvergedSins\* ConvergedSins::createConvergedSins(Color a, int b)
2. {
3. ConvergedSins\* ret = new ConvergedSins();
5. if(ret && ret->init(a, b))
6. {
7. ret->autorelease();
8. return ret;
9. }
11. SAFE\_DEL(ret);
12. return NULL;
13. }

2 biến a và b được truyền vào hàm createConvergedSins không mang ý nghĩa. Sẽ tốt hơn nếu thay a là color và b trở thành level do trong ngữ cảnh này a và b mang 2 ý nghĩa đó, thay xong ta sẽ được code như bên dưới.

1. ConvergedSins\* ConvergedSins::createConvergedSins(Color color, int level)
2. {
3. ConvergedSins\* ret = new ConvergedSins();
5. if(ret && ret->init(color, level))
6. {
7. ret->autorelease();
8. return ret;
9. }
11. SAFE\_DEL(ret);
12. return NULL;
13. }

Sau khi thay đổi được đoạn code như trên lúc xem xét hoặc bảo trì chúng ta sẽ tránh mất thời gian khi phải tra lại ý nghĩa của một biến.

Tuy nhiên, cũng không nhất thiết phải tuân theo quy tắc 100% vì đôi lúc hàm ta triển khai khá đơn giản, ta khảo sát hàm cộng 2 số sau

1. int Add(int a, int b)
2. {
3. return a + b;
4. }
5. Bảng liệt kê một số loại biến và cách đặt tên

|  |  |  |
| --- | --- | --- |
| **LOẠI BIẾN** | **CÁCH ĐẶT BIẾN** | **VÍ DỤ** |
| Biến cục bộ (local) | Chữ thường toàn bộ cho các ký tự, ngăn cách mỗi từ trong biến bởi dấu gạch dưới. | int age\_of\_person  char\* name |
| Biến toàn cục (global) | Chữ thường toàn bộ cho các ký tự, tiền tố g\_, ngăn cách mỗi từ trong biến bởi gạch dưới. | int g\_age\_of\_person  char\* g\_name |
| Biến dạng static | Chữ thường toàn bộ cho các ký tự, tiền tố s\_, ngăn cách mỗi từ trong biến bởi gạch dưới. | int s\_age\_of\_person  char\* s\_name |
| Biến dạng hằng (constant) | Viết hoa toàn bộ các ký tự, tiền tố C\_, ngăn cách mỗi từ trong biến bởi gạch dưới. | const int C\_AGE\_OF\_PERSON  const char\* C\_NAME |
| Các trường của struct và class | Chữ thường toàn bộ cho các ký tự, tiền tố m\_, ngăn cách mỗi từ trong trường này bởi gạch dưới. | int m\_age\_of\_person  char\* m\_name |
| Các trường static của struct và class | Chữ thường toàn bộ cho các ký tự, tiền tố s\_, ngăn cách các từ trong trường bởi gạch dưới. | int s\_age\_of\_person  char\* s\_name |

1. Các quy tắc khác

Biến truyền vào hàm sẽ viết thường toàn bộ các ký tự, ngăn cách các từ trong biến bởi dấu gạch dưới.

Cách khai báo một struct cần viết hoa các ký tự đầu của mỗi từ tên struct và bắt đầu với tiền tố S, ví dụstruct STesla.

Cách khai báo một class cần viết hoa các ký tự đầu của mỗi từ trong tên class và bắt đầu với tiền tố C, ví dụ struct CHumanBeing.

1. Các vấn đề khi khai báo hàm

\*Lưu ý: tôi gọi phương thức của class hay hàm vắn tắt là hàm cho ngắn gọn mặc dù chúng ta cần phân biệt chúng.

Với Java, có nhiều bạn tiếp xúc với Java Coding Convention nhận thấy đa phần các lập trình viên bắt đầu tên phương thức bằng từ viết thường, nếu phương thức được tạo thành từ nhiều từ, thì từ thứ 2 trở đi sẽ viết hoa chữ cái đầu tiên, ta có ví dụ: createBasicSinsWithData(), playEffectDestroySins().

Với Stdio Coding Convention ở mức 1 thì ta có thể chọn 1 trong 2 cách sau, viết theo cách của Java Coding Convention hoặc viết hoa chữ cái đầu tiên của các từ trong tên hàm, ta có ví dụ cho cách thứ 2 này: CreateBasicSinsWithData(), PlayEffectDestroySins().

1. Cách đặt tên hàm phải mang nghĩa nhất định

Việc đặt tên hàm mang ý nghĩa cũng như đặt tên biến có ý nghĩa như đã trình bày ở trên, ở đây ta sẽ nói sâu hơn về các cấp độ của ý nghĩa. Đặt tên hàm hay biến càng mang đầy đủ ý nghĩa sẽ càng tốt.

Mức độ 1: ta xét ví dụ với hàm có ý nghĩa thấp sau, hàm create(...)

1. LightSins\* LightSins::create(eBasicSinsColor color)
2. {
3. LightSins\* ret = new LightSins();
5. if(ret && ret->init(color))
6. {
7. ret->autorelease();
8. return ret;
9. }
11. SAFE\_DEL(ret);
12. return NULL;
13. }

Mức độ 2: gia tăng ý nghĩa hơn cho hàm create(...), ta đặt lại tên hàm với nhiều từ hơn, sửa create(...)thành createLightSins(...)

1. LightSins\* LightSins::createLightSins(eBasicSinsColor color)
2. {
3. LightSins\* ret = new LightSins();
5. if(ret && ret->init(color))
6. {
7. ret->autorelease();
8. return ret;
9. }
11. SAFE\_DEL(ret);
12. return NULL;
13. }

Mức độ 3: làm rõ nghĩa hơn nữa, ta điều chỉnh thành createLightSinsWithColor(...)

1. LightSins\* LightSins::createLightSinsWithColor(eBasicSinsColor color)
2. {
3. LightSins\* ret = new LightSins();
5. if(ret && ret->init(color))
6. {
7. ret->autorelease();
8. return ret;
9. }
11. SAFE\_DEL(ret);
12. return NULL;
13. }

Khi sử dụng hàm trên, ta sẽ tránh mất thời gian hơn trong việc xét xem cần truyền tham số thuộc loại nào vào hàm, ngoài ra ta sẽ thấy rõ ràng hơn ý nghĩa của hàm, đôi lúc không cần xem phần hiện thực cũng có thể đoán được ý nghĩa của hàm.

1. Các vấn đề khác về soạn thảo văn bản

Bên dưới là phong cách soạn thảo code trong game Sins của Stdio. Với việc trình bày “văn bản” một cách trong sáng hơn và có "ý nghĩa" trong việc viết code sẽ làm cho chương trình dễ đọc và dễ bảo trì. Tuy nhiên, đó là quan điểm phù hợp với dự án [Sins](http://sins.stdio.vn/), tùy vào mỗi cá nhân mà nó sẽ có các biến thể. Các bạn có thể xem tham khảo.

1. Level của 1 khối code

Việc đặt level đúng của 1 khối code là rất quan trọng, nó giúp ta xác định nhanh hơn "tầm vực" của 1 dòng code và giúp ta xác định vấn đề nhanh hơn.

Xét ví dụ sau, code không được trình bày đúng level

1. if(m\_fAdd1ScoreTimer == 0.0f)
2. {
3. if(m\_iScorePool > 0)
4. {
5. int addScore = (int)(m\_iScorePool \* 0.2f);
6. if(addScore < 1)
7. addScore = 1;
9. m\_iScorePool -= addScore;
10. m\_iCurrentScore += addScore;
11. m\_fAdd1ScoreTimer = DELAY\_ADD\_1\_SCORE;
13. if(m\_iScorePool == 0)
14. {
15. m\_pScoreLabel->runAction(
16. CCSequence::create(
17. CCScaleTo::create(0.05f, 2.0f\*m\_ScaleFactorScore),
18. CCScaleTo::create(0.1f, 1.0f\*m\_ScaleFactorScore), NULL
19. ));
20. }
21. }
22. }

Với code như trên, ta có thể hiểu lầm rằng m\_iCurrentScore += addScore; (dòng 9) nằm cùng cấp với dòng if(m\_fAdd1ScoreTimer == 0.0f) (dòng 1), tức là nó không nằm trong if.

Thay vì vậy, ta nên đưa nó vào đúng level của nó, sẽ dễ nhận dạng hơn khi bảo trì.

1. if(m\_fAdd1ScoreTimer == 0.0f)
2. {
3. if(m\_iScorePool > 0)
4. {
5. int addScore = (int)(m\_iScorePool \* 0.2f);
6. if(addScore < 1)
7. addScore = 1;
9. m\_iScorePool -= addScore;
10. m\_iCurrentScore += addScore;
11. m\_fAdd1ScoreTimer = DELAY\_ADD\_1\_SCORE;
13. if(m\_iScorePool == 0)
14. {
15. m\_pScoreLabel->runAction(CCSequence::create(
16. CCScaleTo::create(0.05f, 2.0f\*m\_scaleFactorPts),
17. CCScaleTo::create(0.1f, 1.0f\*m\_scaleFactorPts),
18. NULL
19. ));
20. }
21. }
22. }

Khi làm việc với [Taco Nguyen](http://www.stdio.vn/users/index/2), anh ta sử dụng Python khá nhiều cho công việc, tôi mới biết thêm về  ngôn ngữ này, anh ta cho tôi biết rằng chỉ cần dư hoặc thiếu 1 khoảng trắng hoặc tab, thì chương trình sẽ chạy sai mục đích ngay, vì cách định ra 1 level của 1 dòng code trong Python là dựa vào số lượng tab hoặc khoảng trắng, khác với nhiều ngôn ngữ khác là “đống khối” 1 khối code dựa vào cặp ngoặc nhọn { và }.

1. Đặt dấu (chấm, phẩy, hai chấm, chấm phẩy, hỏi, chấm than)

Theo phương pháp đặt dấu trong văn bản thì "dấu câu" trong code cũng có vài nét tương đồng. Nguyên tắc là dấu sẽ nằm ngay phía sau ký tự cuối và sau đó là đến khoảng trắng rồi mới đến ký tự kế tiếp.

Ví dụ đặt dấu sai, các bạn khảo sát tại dấu phẩy (,) và dấu hỏi (?)

Tôi sẽ code tốt hơn sau khi đọc bài viết này ,còn bạn thì sao? (sai chỗ dấu phẩy, đúng chỗ dấu hỏi)  
Tôi sẽ code tốt hơn sau khi đọc bài viết này,còn bạn thì sao ? (sai chỗ dấu phẩy, sai chỗ dấu hỏi)  
Tôi sẽ code tốt hơn sau khi đọc bài viết này, còn bạn thì sao ? (đúng chỗ dấu phẩy, sai chỗ dấu hỏi)

Cách đặt dấu đúng dựa theo nguyên tắc đã đề cập phía trên:

**Tôi sẽ code tốt hơn sau khi đọc bài viết này, còn bạn thì sao?**

Từ cách trình bày văn bản như trên, ta có thể tận dụng vào việc trình bày code, xét ví dụ code về việc đặt dấu sau

1. playActionFollowDelete(1,this,callfunc\_selector(SinsDestroy::playDone)) ;

Các dấu phẩy hay chấm phẩy như trên đặt không tốt, ta có thể trình bày lại như sau

1. playActionFollowDelete(1, this, callfunc\_selector(SinsDestroy::playDone));
2. Cách dòng trong code

Khảo sát đoạn codes sau, đây là đoạn codes đã cách dòng tốt

1. void CLoadingData::releaseAllResource()
2. {
3. m\_iNumBackgroundAudios = 0;
5. // RELEASE SPRITE
6. for(u8 i = 0; i < m\_tSpritePacks.size(); i++)
7. {
8. GetSpriteManager()->unloadSprite(m\_tSpritePacks.at(i));
9. }
10. m\_tSpritePacks.clear();
12. // RELEASE AFX
13. for(u8 i = 0; i < m\_tAudioPacks.size(); i++)
14. {
15. unloadEffect(m\_tAudioPacks.at(i));
16. }
17. m\_tAudioPacks.clear();
18. }

Xem xét đoạn từ comment // RELEASE SPRITE (dòng 5) và tới đoạn // RELEASE AFX (dòng 12), giữa 2 đoạn đó cách nhau bởi 1 dòng trống (blank) (dòng 11). Ý nghĩa của nó nhằm ngăn cách 2 khối lệnh có 2 chức năng khác nhau. Như vậy, khi bảo trì ta có thể nhanh chóng xác định được đoạn codes nào làm việc gì thay vì để liên tục từ trên xuống như đoạn codes bên dưới.

1. void CLoadingData::releaseAllResource()
2. {
3. m\_iNumBackgroundAudios = 0;
4. // RELEASE SPRITE
5. for(u8 i = 0; i < m\_tSpritePacks.size(); i++)
6. {
7. GetSpriteManager()->unloadSprite(m\_tSpritePacks.at(i));
8. }
9. m\_tSpritePacks.clear();
10. // RELEASE AFX
11. for(u8 i = 0; i < m\_tAudioPacks.size(); i++)
12. {
13. unloadEffect(m\_tAudioPacks.at(i));
14. }
15. m\_tAudioPacks.clear();
16. }
17. Hướng phát triển

Chuẩn mực trên đã từng được sử dụng bởi Stdio trong dự án Sins, để ngày càng mạnh và chặt chẽ hơn, các chuẩn mực này đã tiến hóa hơn rất nhiều, do đó tôi đã cắt gọt lại và giữ bản Stdio Coding Convention đầu tiên cho các bạn dễ học, dễ nhớ.

Ngoài ra, từ đây các bạn có thể tự thiết kế nên chuẩn mực chung của nhóm làm việc, hoặc học cách tuân theo chuẩn mực cho các dự án của các bạn sau này.

1. Lời kết

Như một hành trình của máy bay đã được định tuyến bay thẳng từ Thành Phố Hồ Chí Minh đến Hà Nội, vì các yếu tố vật lý như gió, mây thì máy bay vẫn lệch khỏi đường bay đã vạch ra sẵn, nhưng nó sẽ mau chóng trở lại với định hướng ban đầu, đơn giản là vì nó đã có chuẩn mực về đường bay.

*Nguyên tắc do con người đặt ra và nguyên tắc là chết, con người là sống và ta cần linh động, nhưng việc linh động đó chưa phải là cái lúc mà bạn chưa nắm được định hướng chính vì sẽ dễ đi lạc.*

### Cơ Bản Debug Trong Visual Studio:

Một chương trình được viết ra thì thường không hoàn hảo, vẫn tồn tại những lỗ hổng mà các hacker có thể khai thác và lợi dụng, thậm chí phá hỏng chương trình hay đánh sập cả  một hệ thống. Ở mức độ học tập, một chương trình có thể tồn tại những lỗi về logic mà ta khó nhận ra, đòi hỏi kĩ năng tư duy logic cao để phát hiện và sửa lỗi.

Hiện nay, hầu hết các IDE đều hỗ trợ khả năng gỡ rối (debug) chương trình bằng cách chạy từng dòng lệnh (step by step) và xem xét sự thay đổi giá trị của các biến. Bài viết này sẽ giúp các bạn làm quen với công cụ Debug trong Visual Studio.

##### Debug là gì

Debug là quá trình tìm kiếm và fix bug (sửa lỗi) của chương trình. Quá trình debug thường mất nhiều thời gian hơn việc viết chương trình, vì những lỗi liên quan đến logic thường rất khó phát hiện. Visual Studio cung cấp cho chúng ta khá nhiều công cụ trực quan để đơn giản hoá quá trình debug, giúp ta tiến hành tìm và sửa lỗi chương trình một cách dễ dàng hơn.

##### Một số khái niệm liên quan

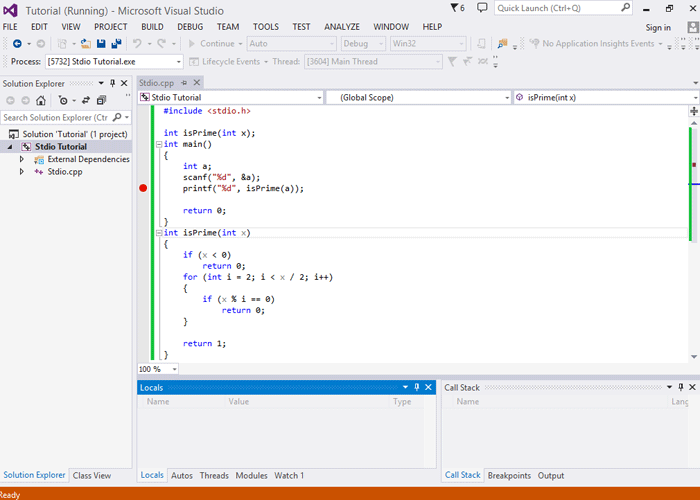
1. Chạy chương trình

Một chương trình sẽ lần lượt duyệt qua các dòng code có trong hàm main. Trong đó, có những đoạn “thẳng”, có những đoạn “rẽ nhánh” (if/switch), có những đoạn “vòng” (for/while/…), có những đoạn đường nhỏ hơn (các function). Để sửa lỗi chương trình, cần tìm ra được những “đoạn rối” của nó, tức là xác định xem đoạn code ở vị trí nào có khả năng phát sinh lỗi.

1. Breakpoints

**Breakpoints** là vị trí mà chương trình sẽ dừng lại để lập trình viên xem xét sự thay đổi của các biến qua từng dòng lệnh, từ đó phát hiện ra vị trí dòng code bị lỗi.

**Breakpoint** được kí hiệu bằng chấm tròn màu đỏ ở đầu dòng code. Để tạo ra một **breakpoint**, cách đơn giản nhất là click chuột vào đầu dòng code (như trong hình). Để huỷ breakpoint, chỉ cần click chuột vào **breakpoint** đó một lần nữa. Ngoài ra các bạn cũng có thể tạo/huỷ breakpoint bằng phím **F9**.

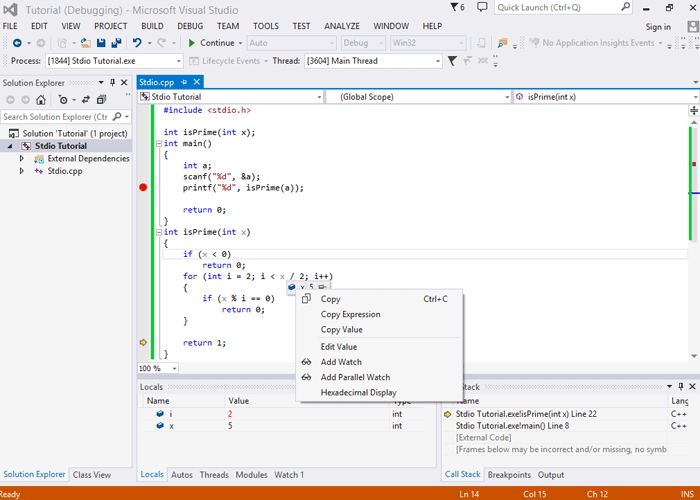


1. Watch Windows

Trong quá trình Debug, sau khi chương trình dừng lại ở một **breakpoint** nào đó, điều cần thiết của chúng ta là kiểm tra sự thay đổi giá trị của các biến, hàm, … qua từng dòng code để nhìn ra sai sót trong thuật toán. Watch windows là tập hợp các công cụ giúp giúp lập trình viên quan sát được giá trị hiện tại của biến. Các cửa sổ này có thể được tìm thấy trong menu **Debug → Windows** (chỉ xuất hiện khi đang trong quá trình debug).

1. Data Tip

Khi di chuyển con trỏ chuột đến tên biến ở bất kì vị trí nào trong phạm vi cặp dấu { } (scope) hiện tại, giá trị của biến sẽ được hiển thị trên màn hình. Khi đó, các bạn có thể “ghim” biến đó lên màn hình để tiện quan sát, hoặc add vào cửa sổ Watch, copy giá trị, thay đổi giá trị của biến, …



1. Locals

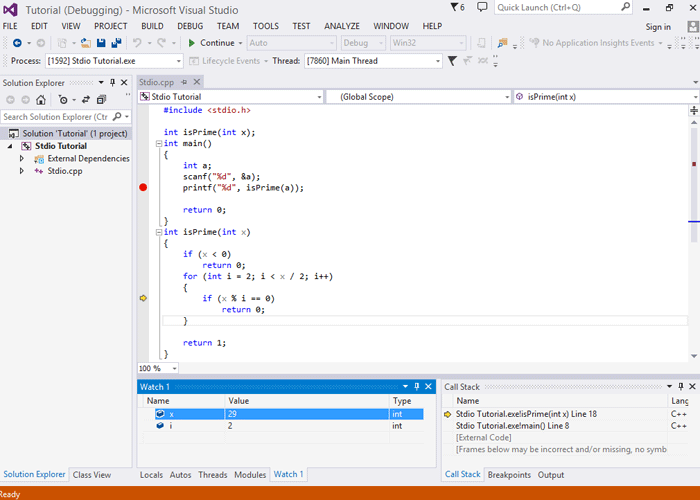
Cửa sổ Locals sẽ hiển thị tất cả các biến có liên quan đến dòng code hiện tại một cách tự động. Các biến hiển thị ở đây sẽ được thay đổi qua từng dòng code. Ngoài ra, màu sắc của các biến giúp ta phân biệt được những biến nào vừa thay đổi giá trị.

1. Autos

Tương tự như Locals, cửa sổ Autos hiển thị các biến vừa được sử dụng trong các dòng code trước. Visual Studio sẽ tự động nhận diện biến nào không còn cần thiết và loại bỏ ra khỏi cửa sổ Autos.

1. Watch

Visual Studio không thể nhận diện được tất cả những gì lập trình viên cần. Trong một số trường hợp, chúng ta cần theo dõi cụ thể một giá trị nào đó, chẳng hạn như phần tử thứ 10 trong một mảng số nguyên hay kí tự thứ 5 của chuỗi “Stdio Tutorial”, … Chúng ta sẽ cần sử dụng đến cửa sổ Watch. Nó cho phép lập trình viên nhập vào tên biến, hàm, … cụ thể trong scope hiện tại. Giá trị của các biến, hàm sẽ được hiển thị bên cạnh tên biến.



1. Debug chương trình

Trước khi debug, các bạn cần tạo ra các breakpoints cần thiết để tìm và sửa lỗi chương trình. Để bắt đầu tiến hành debug một chương trình, các bạn vào menu **Debug → Start Debugging** hoặc nhấn phím **F5** trên bàn phím. Visual Studio sẽ tiến hành Build chương trình. Sau khi Build xong và không có lỗi biên dịch (compile error), cửa sổ Debug sẽ xuất hiện. Nếu không có các lệnh dừng màn hình chờ nhập dữ liệu (như scanf, gets, …), chương trình sẽ dừng lại tại **breakpoint** đầu tiên.

Một thanh công cụ Debug sẽ xuất hiện. Trong đó, có các nút công cụ quan trọng như sau:

* http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_3/34/ss_4.png**Step Over**: Chạy step by step, lướt qua hàm (chỉ nhận giá trị return của hàm).
* http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_3/34/ss_5.png**Step Into**: Chạy step by step, đi vào nội dung của các hàm con.
* http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_3/34/ss_6.png**Step Out**: “Nhảy” đến breakpoint kế tiếp. Nếu không còn **breakpoint** nào thì sẽ kết thúc debug. Ngoài ra nó còn có chức năng chạy lướt qua hàm con hiện tại.

1. Một số phím tắt hữu ích

* **F5**: Bắt đầu quá trình Debug.
* **Shift + F5:** Thoát Debug.
* **Ctrl + F5**: Chạy chương trình không dùng công cụ Debug
* **F9**: Tạo/huỷ một breakpoint.
* **F10**: Step Over.
* **F11**: Step Into.
* **Shift + F11**: Step Out.

### Bản chất của biến:

1. Vấn đề size của từng biến:

* Các Biến được lưu trữ trong bộ nhớ CT dưới dạng Binary code (mã nhị phân)
* Do có nhiều kiểu dữ liệu nên trình biên dịch phải chuyển giá trị biến thành một Binary code tương ứng trước khi lưu vào bộ nhớ.
* Mỗi kiểu dữ liệu trên mỗi hệ điều hành sẽ được cấp phát 1 vùng nhớ khác nhau

Ví dụ:

|  |
| --- |
| Biến Char có dung lượng 1byte tất cả HDH.  Biến **Int** có dung lượng  2 byte ở HDH 16bit  4 byte ở HDH 32bit  8 byte ở HDH 64bit |
| **Lưu ý hệ điều hành tương thích khi tạo project C,C++**  **Đây là lý do tại sao app 32 bit chạy trên hệ 64 bit được. Nhưng ngược lại thì không được.** |

1. Rắc rối size of variable:

Khi chúng ta cố gắng nhét 1 giá trị (Binary code) có dung lượng lớn hơn vùng nhớ cấp phát của biến, hiện tượng *“overflow”* – tràn bộ nhớ sẽ xảy ra.

Tùy mỗi trường hợp sẽ có cách giải quyết cho việc tràn này, có loại sẽ tự động cắt bớt một số bit bị tràn ra bên ngoài để số bit còn lại vừa vặn với phân khúc được cấp phát, số khác thì sẽ không chấp nhận điều đó mà sẽ cho ra một thông báo lỗi (Error) hoặc sẽ đưa ra một ngoại lệ (Exception).

1. Hướng giải quyết cho overflow

Trong C/C++ hỗ trợ chúng ta giải quyết nhưng với mỗi trường hợp thì một cách khác nhau.

Nếu trường hợp xảy ra tràn do việc khai báo, gán, làm các phép toán số học dẫn đến tràn thì *overflow* sẽ được khắc phục bằng cách cắt bớt chuỗi bit bị tràn ra khỏi size của bộ nhớ biến.

Trường hợp khác (ví dụ như cấp phát vùng nhớ, ép kiểu trong thao tác..) thì C/C++ cho phép sự tràn xảy ra trong chương trình, các *binary code* được lưu một cách bất chấp (tràn sang ô nhớ bên cạnh) và thường thì lập trình viên phải chịu trách nhiệm trước lỗi này bằng cách tự khắc phục nó bằng bất kỳ cách nào.

1. TRƯỜNG HỢP 1:

Xét ví dụ bên dưới:

1. #include <stdio.h>
2. int main()
3. {
4. char i;
5. for(i = 0; i < 128; i++)
6. {}
7. return 0;
8. }

Thoạt nhìn có lẽ đa số trong các bạn sẽ nghĩ dòng for này hoàn toàn không có gì đặc biệt. Nhưng thực chất trong trường hợp này nó là một vòng lặp vô hạn. Để chứng thực cho việc này xin mời bạn viết đoạn code này vào trong Visual Studio và nhấn F5.

Để giải thích cho điều này không khó. Ta có một biến char i; suy ra biến i của chúng ta sẽ được cấp một vùng nhớ 1 byte = 8 bit trong bộ nhớ. Khi vào dòng for thì biến i đã vô tình được chuyển thành kiểu int có kích thước 1 byte, và giới hạn của biến int này từ -128 đến 127. Theo chuẩn số bù 2 thì 8 bit lưu trữ biến i sẽ có 1 bit cao nhất quyết định dấu của số đó ( bit 0 nếu là số dương và 1 nếu âm). Ta chạy từ i = 0 đến i = 127 mọi việc vẫn tốt đẹp bình thường, khi đến cuối dòng for của giá trị i = 127 thì i tự động tăng lên 1 theo nguyên lý dòng for.

Vì i tăng lên 1 nên binary code của i lúc này là 10000000, theo chuẩn số bù 2 thì giá trị này cho thấy i = -128 ta thấy -128 nhỏ hơn 128 nên vòng for tiếp tục. Biến i vẫn tăng lên đến lúc i = -1 thì binary code của i lúc này là 11111111, sau đó i tiếp tục tăng thêm 1 đơn vị và binary code trở thành100000000, vì ô nhớ chưa biến i chỉ có 8 bit nên nó tự động loại bỏ bit ở vị trí cao nhất (bit 1) và khiến cho binary code của i lúc này là 00000000 biểu diễn số 0. Vòng lặp mới bắt đầu tại đây.

Đến đây các bạn hẳn đã nắm rõ nguyên lý làm việc của một biến khi bị *overflow*. Khi ấy chuỗi bit sẽ được cắt bớt ở phía những bit cao nhất đến khi nào số bit còn lại vừa vặn với phân khúc vùng nhớ cho phép thì thôi. Tôi muốn nhấn mạnh ở đây rằng: "Trong một chương trình, không nhất thiết cứ tăng *n* lên 1 thì giá trị sau khi tăng là *n + 1,*điều tương tự với việc giảm *n* đi 1 đơn vị".

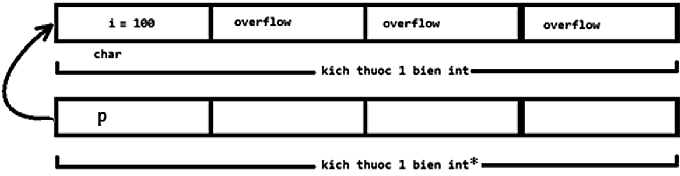
1. TRƯỜNG HỢP 2:

Trong một trường hợp khác thì C/C++ cho ta một cách xử lý khác. Sự tràn này được bỏ qua và đưa ra một lỗi buộc lập trình viên phải tự khắc phục.

Ta cùng xét 2 ví dụ bên dưới:

Ví dụ 1:

1. #include <stdio.h>
2. int main()
3. {
4. char i;
5. int\* p = (int\*) &i;
6. i = 100;
7. printf("%d %d", \*p, i);
8. p = NULL; // biến i đang nằm trên stack nên không thể ra lệnh hủy p;
9. return 0;
10. }

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_1/5/ss_2.png)

Trong trường hợp này khi chạy chương trình không có lỗi. Màn hình in ra 2 giá trị -858993564 và 100. Không biết có bạn nào thắc mắc hay không? Nhưng lúc test đoạn code này tôi đã thắc mắc rất lớn! Tại sao lại là -858993564 và 100? Giải thích là vì khi ta thao tác với biến i = 100 thì chỉ có mỗi 1 vùng nhớ biến i được thay đổi thành giá trị 100, còn lại 3 byte kia không hề thay đổi gì cả. Điều này khiến cho 2 giá trị \*p và i khác nhau. Nhưng kết thúc chương trình lại không có lỗi nào được xảy ra cả. Nguyên nhân ở đây theo mình hiểu là vì vùng nhớ 3 byte nằm ngoài biến i của con trỏ p không bị corrupted khi thao tác trong chương trình này.

Ví dụ 2:

1. #include <stdio.h>
2. int main()
3. {
4. char i;
5. int\* p = (int\*) &i;
6. \*p = 100;
7. printf("%d %d", \*p, i);
8. p = NULL; // biến i đang nằm trên stack nên không thể ra lệnh hủy p;
9. return 0;
10. }

Trường hợp này sẽ có lỗi ở cuối chương trình. Lúc mà vùng nhớ được giải phóng. Và sản phẩm được in ra màn hình là 100 và 100. Khác hẳn với đoạn code ở*ví dụ 1*.

Để giải thích vì sao ta phải lướt qua kiến thức về [*little endian*](#_Little_Endian_Và). Ta thấy rằng trong 4 byte của biến int, không phải tất cả 4 byte của biến int được lưu lần lượt vào trong ô nhớ đã được cấp phát mà là byte nằm trái cùng (cao nhất) của biến i được lưu vào byte phải cùng (thấp nhất) được cấp phát cho biến đó trong bộ nhớ. Và lần lượt mỗi lần lưu 8 bit vào trong 1 byte.

Cụ thể là thế này:

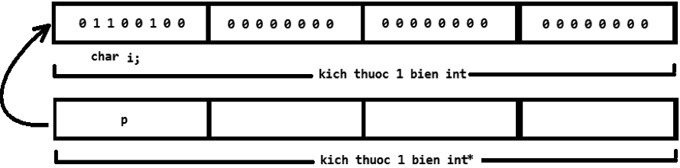
Lấy 8 bit của byte cao nhất đã được mã hóa đưa vào byte thấp nhất của vùng nhớ cấp phát cho biến, và cứ thế cho đến khi hết byte. Phần còn lại bị tràn ra thì được cắt bỏ. Nếu vậy thì giả sử biến int i = 69 được dịch từ thập phân ra binary code sẽ đượcđược:

**00000000 00000000 00000000 01000101**

Nhưng đưa vào trong ô nhớ thì sẽ lưu như sau:

**01000101 00000000 00000000 00000000**

Từ đó ta hiểu ở đoạn code trên, các ô nhớ được mô phỏng như sau:

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_1/5/ss_3.png)

Khi chương trình đọc giá trị của biến, nó sẽ đọc từ byte thấp đến byte cao để lấy được giá trị biến đó. Theo nhìn nhận chủ quan của tôi thì cách thao tác này rất phù hợp cho việc ép kiểu trong quá trình viết code của chúng ta. Vì tổ chức ô nhớ theo nguyên lý trên tôi đã trình bày. Nên 2 giá trị \*p và i đều bằng 100. Nhưng khi kết thúc chương trình lại xảy ra lỗi.

Vấn đề ở đây là lỗi gì và tại sao? Ta có câu trả lời như bên dưới:

Lỗi *corrupted,*lỗi này thường gặp khi chúng ta thao tác với vùng nhớ mà chúng ta không có quyền chạm đến. Cụ thể ở đây là ta có biến int\* p chỉ có quyền hạn điều khiển được biến i có dung lượng 1 byte, nhưng trong nguyên tắc cấp phát biến int thì nó được quyền quản lý đến 4 byte, vậy đã có 3 byte bị chương trình hiểu nhầm là có quyền truy cập trong trường hợp này. 3 byte này không được phép sử dụng nhưng vẫn bị sử dụng ở dòng lệnh \*p = 100 làm thay đổi nội dung 3 byte này. Nên cuối chương trình lỗi đã xảy ra. Và trường hợp này, không còn cách nào khác ngoại trừ việc lập trình viên phải chọn cho mình một cách thức khác để thực hiện vấn đề này.

1. Tổng kết

Trong việc lập trình, chúng ta hoàn toàn có thể kiểm soát size của biến ta cần dùng, và chức năng của từng biến do chính lập trình viên qui định. Để giảm thiểu những lỗi không rõ ràng về memory, thì điều tốt nhất đó chính là tập khả năng kiểm soát biến ngay từ bây giờ. Luôn nhớ rằng khi new bất kỳ một cái gì đó thì phải delete nó khi không còn sử dụng nữa. Và đừng bao giờ cho phép mình quên sizeof của từng kiểu biến, nếu có quên thì hãy dùng từ khóa sizeof(type).

### Phát Hiện Memory Leak Với VLD

Memory Leak – rò rỉ bộ nhớ xảy ra khi ta cấp phát một vùng nhớ và không thu hồi lại chúng sau khi sử dụng. Để tránh việc này xảy ra, theo lý thuyết là bất kể khi nào ta gọi toán tử new thì ngay sau đó ta cần gọi toán tử delete (tương ứng với cặp đôi malloc(...) - free(...)). Lý thuyết trên hoàn toàn đúng khi ta muốn quản lý và sử dụng bộ nhớ một cách có hiệu quả. Tuy nhiên, trong thực tế khi làm việc, tôi đã tiếp xúc với những dự án có đến hàng trăm, ngàn file khác nhau, và với mỗi file lại vài trăm cho tới vài ngàn dòng code. Đó là chưa kể những dự án này có những phần không phải do tôi viết ra mà kế thừa từ những đồng nghiệp, và trong đó ta có các lớp này chồng lớp khác, có các struct và các hàm cấp phát vùng nhớ …

Với lượng dữ liệu đồ sộ như vậy, việc xét xem ta đã hủy các vùng nhớ sau khi sử dụng mà không dùng bất kì một công cụ hỗ trợ nào, gần như là một việc tốn rất nhiều thời gian mà có khi lại không hiệu quả do sơ sót trong việc làm thủ công.

Chính vì vậy, với kinh nghiệm của mình, tôi muốn giới thiệu đến các bạn một thư viện hỗ trợ cho việc phát hiện Memory Leak – đó là Visual Leak Detector.

Trong bài viết này tôi sử dụng **Visual Studio 2013 Ultimate** để minh họa, tùy phiên bản mà bạn đọc có thể có những khác biệt nhỏ.

##### Visual Leak Detector là gì?

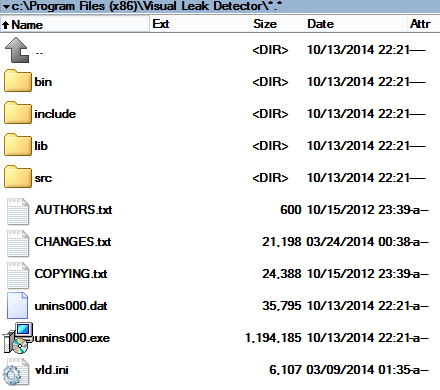
Visual Leak Detector (VLD) là một thư viện mã nguồn mở hoàn toàn miễn phí giúp cho ta có thể xác định được memory leak ở trong chương trình. VLD chỉ tương thích với Visual C++.

Để có thể sử dụng VLD, tùy theo phiên bản Visual Studio sử dụng mà ta tải version thích hợp về. Trong trường hợp của tôi vì tôi sử dụng VS 2013 Ultimate do đó version tôi sử dụng là [VLD v2.4rc2](https://vld.codeplex.com/releases/view/120547).

##### Cài đặt VLD

Như đã đề cập bên trên, sau khi việc tải xuống hoàn tất ta tiến hành cài đặt VLD vào máy tính. Để tiện lợi trong quá trình sử dụng các bạn có thể giữ nguyên phần cấu hình cài đặt mặc định.

Nếu quá trình cài đặt thành công, trong folder Program Files sẽ có thêm folder Visual Leak Detector với cấu trúc như sau:



##### Sử dụng VLD

Muốn sử dụng VLD, ta đơn giản chỉ cần thêm include sau vào tất cả các file .cpp

1. #include <vld.h>

Để minh họa cho việc sử dụng VLD, đầu tiên ta cần giả lập môi trường ứng dụng đang có memory leak bằng việc cấp phát một vùng nhớ ngẫu nhiên và “cố tình” không gọi lệnh hủy chúng.

1. #include <string.h>
3. void main()
4. {
5. char\* p = new char[1024];
6. char\* str = strdup("Hello Stdio !");
7. }

Nếu ta không khai báo sử dụng VLD, đoạn chương trình sau sẽ kết thúc mà không có bất cứ thông báo gì đặc biệt.

Tiếp theo ta thử nghiệm bằng việc khai báo sử dụng thư viện VLD như trong đoạn mã dưới đây:

1. #include <vld.h>
2. #include <string.h>
4. void main()
5. {
6. char\* p = new char[1024];
7. char\* str = strdup("Hello Stdio !");
8. }

Khi đó, chương trình của chúng ta kết thúc, VLD sẽ giúp ta lần lượt rà soát lại tất cả các vùng nhớ ta đã cấp phát mà chưa gọi lệnh hủy. Và điểm đặc biệt nhất là VLD thông báo cho ta một cách rất chi tiết về vùng nhớ bị leak: kích thước, dữ liệu, call stack … như trong ví dụ dưới đây:

1. Visual Leak Detector Version 2.4RC2 installed.
2. WARNING: Visual Leak Detector detected memory leaks!
3. ---------- Block 1 at 0x0044DC58: 1024 bytes ----------
4. Leak Hash: 0xABCDB444, Count: 1, Total 1024 bytes
5. Call Stack (TID 18000):
6. 0x0F87C260 (File and line number not available): MSVCR120D.dll!operator new
7. c:\users\huy.vuquang\documents\visual studio 2013\projects\stdio\_vld\_demo\main.cpp (6): STDIO\_VLD\_Demo.exe!main + 0xA bytes
8. f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c (626): STDIO\_VLD\_Demo.exe!\_\_tmainCRTStartup + 0x19 bytes
9. f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c (466): STDIO\_VLD\_Demo.exe!mainCRTStartup
10. 0x7648919F (File and line number not available): KERNEL32.DLL!BaseThreadInitThunk + 0xE bytes
11. 0x77BD0BBB (File and line number not available): ntdll.dll!RtlInitializeExceptionChain + 0x84 bytes
12. 0x77BD0B91 (File and line number not available): ntdll.dll!RtlInitializeExceptionChain + 0x5A bytes
13. Data:
14. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
15. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
16. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
17. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
18. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
19. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
20. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
21. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
22. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
23. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
24. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
25. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
26. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
27. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
28. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........
29. CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD ........ ........

32. ---------- Block 2 at 0x0044E088: 14 bytes ----------
33. Leak Hash: 0x9E7F558B, Count: 1, Total 14 bytes
34. Call Stack (TID 18000):
35. 0x0F7D80F0 (File and line number not available): MSVCR120D.dll!strdup
36. c:\users\huy.vuquang\documents\visual studio 2013\projects\stdio\_vld\_demo\main.cpp (7): STDIO\_VLD\_Demo.exe!main + 0xD bytes
37. f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c (626): STDIO\_VLD\_Demo.exe!\_\_tmainCRTStartup + 0x19 bytes
38. f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c (466): STDIO\_VLD\_Demo.exe!mainCRTStartup
39. 0x7648919F (File and line number not available): KERNEL32.DLL!BaseThreadInitThunk + 0xE bytes
40. 0x77BD0BBB (File and line number not available): ntdll.dll!RtlInitializeExceptionChain + 0x84 bytes
41. 0x77BD0B91 (File and line number not available): ntdll.dll!RtlInitializeExceptionChain + 0x5A bytes
42. Data:
43. 48 65 6C 6C 6F 20 53 74 64 69 6F 20 21 00 Hello.St dio.!...

46. Visual Leak Detector detected 2 memory leaks (1110 bytes).
47. Largest number used: 1110 bytes.
48. Total allocations: 1110 bytes.
49. Visual Leak Detector is now exiting.

Lúc này, công việc của ta đơn giản là đến dòng code, xem xét vùng nhớ được cấp phát như thế nào và tìm thời điểm thích hợp để hủy nó.

1. #include <vld.h>
2. #include <string.h>
4. void main()
5. {
6. char\* p = new char[1024];
7. char\* str = \_strdup("Hello Stdio !");
9. delete[] p;
10. delete[] str;
11. }

Nếu VLD thông báo như dưới đây, thì ta có thể chắc chắn là ứng dụng không còn memory leak nữa.

1. Visual Leak Detector Version 2.4RC2 installed.
2. No memory leaks detected.
3. Visual Leak Detector is now exiting.

##### Lời kết

VLD là một trong những thư viện rất mạnh giúp ta có thể dễ dàng biết được hiệu quả của việc quản lý bộ nhớ trong ứng dụng, tuy nhiên, phương pháp nào cũng có tính tương đối. Qua bài viết này, hy vọng với những kiến thức và kinh nghiệm tôi đề cập ở trên, các bạn sẽ không còn gặp khó khăn trong việc phát hiện và xử lý memory leak nữa.

### Biểu diễn số âm trong máy tính

1. Giới thiệu

Kiến thức này sẽ giúp lập trình viên nắm được cách máy tính biểu diễn số âm dạng nhị phân trong memory.

Trong toán học, các số âm (bất kể thuộc hệ cơ số nào) đều được biểu diễn bằng cách thông thường là đặt trước số dương tương ứng một dấu "−" (trừ).

Ví dụ: với hệ thập phân, số nguyên âm năm được biểu diễn là −5. Tuy nhiên, trong máy tính, khi mọi ký hiệu, con số,... đều được biểu diễn dưới hệ nhị phân thông qua hai chữ số 0 và 1 thì mọi chuyện lại trở nên phức tạp hơn.

Có nhiều cách được sử dụng để biểu diễn số âm trong máy tính. Bài này chỉ giới thiệu bốn phương pháp chủ yếu nhất, đó là: phương pháp dấu lượng (sign-and-magnitude), bù 1, bù 2 và số quá N (excess-N).

Các máy tính hiện nay hầu hết đều sử dụng phương pháp biểu diễn số bù 2. Tuy nhiên, trong vài tình huống, các phương pháp khác vẫn có thể được sử dụng.

1. Dấu lượng

Phương pháp dấu lượng dùng bit cực trái làm bit dấu (sign bit) – tức đại diện cho dấu của số – theo quy ước: nếu bit dấu là 1 thì số là số âm (1 tương đương với dấu "-"), ngược lại, nếu nó là 0 thì số là số dương (0 tương đương với dấu "+"). Các bit còn lại được dùng để biểu diễn độ lớn của số (hay giá trị tuyệt đối – absolute value – của số).

Để biểu diễn một số âm về dạng nhị phân có dấu với mẩu K bit là lấy số cần biểu diễn cộng thêm 2K-1 sau đó biểu diễn chúng ở hệ nhị phân

Theo phương pháp này, một byte 8 bit sẽ có 7 bit (trừ đi bit dấu) được dùng để biểu diễn cho các số có giá trị từ 0000000 (010) đến 1111111 (12710). Khi sử dụng bit dấu, ý nghĩa của 7 bit trên sẽ thay đổi, và ta có thể biểu diễn các số từ −12710 đến +12710. Trong phương pháp dấu lượng, số 0 có thể được biểu diễn ở hai dạng, đó là 00000000 (+0) và 10000000 (−0).

Ví dụ: giả sử mẫu 8 bit, khi sử dụng phương pháp dấu lượng, số 510 được biểu diễn sang hệ nhị phân là: 0000 0101, còn số −5 là 01111011 (vì -5+27=123, biểu diễn nhị phân là 0111 1011)

So sánh với cách biểu diễn số âm mà ta thường sử dụng, ta thấy phương pháp dấu lượng có nhiều điểm tương đồng. Trong hệ thập phân, khi muốn biểu diễn số có dấu, ta đặt dấu cần biểu diễn ngay trước giá trị tuyệt đối của số. Phương pháp dấu lượng cũng đặt dấu ngay trước giá trị tuyệt đối của số, chỉ có khác ở chỗ thay dấu "+" bằng "0" và "−" bằng "1".

Có lẽ vì sự tương đồng này, một vài máy tính thế hệ đầu tiên (như IBM 7090) đã sử dụng phương pháp dấu lượng khi b Bên cạnh phương pháp bù 2, bù 1 cũng thường được sử dụng để biểu diễn số âm trong máy tính. Theo phương pháp này, bit cực trái (là bit nằm bên trái cùng của byte) được sử dụng làm bit dấu (sign bit - là bit tượng trưng cho dấu của số) với quy ước: nếu bit dấu là 0 thì số là số dương, còn nếu nó là 1 thì số là số âm. Ngoài bit dấu này ra, các bit còn lại được dùng để biểu diễn độ lớn của số.

Ví dụ: số −5 được biểu diễn trong máy tính theo phương pháp bù 1 như sau (với mẫu 8 bit): đầu tiên, xác định số 5 được biểu diễn trong máy tính: 0000 0101. Tiếp theo, đảo tất cả các bit có trong số 5: kết quả sau khi đảo: 1111 1010. Vì là biểu diễn số âm nên bit bên trái cùng luôn giữ là 1.

Vậy với phương pháp bù 1, số −5 được biểu diễn trong máy tính như sau: 1111 1010.

1. Bù 1

Bù 1 (tiếng Anh: one's complement) là một số trong hệ nhị phân mà nó chính là bù cơ số trừ 1 (radix-minus-1 complement) của một số khác. Một số bù 1 có thể có được do đảo tất cả các bit có trong số nhị phân (đổi 1 thành 0 và ngược lại).

Bên cạnh phương pháp bù 2, bù 1 cũng thường được sử dụng để biểu diễn số âm trong máy tính. Theo phương pháp này, bit cực trái (là bit nằm bên trái cùng của byte) được sử dụng làm bit dấu (sign bit - là bit tượng trưng cho dấu của số) với quy ước: nếu bit dấu là 0 thì số là số dương, còn nếu nó là 1 thì số là số âm. Ngoài bit dấu này ra, các bit còn lại được dùng để biểu diễn độ lớn của số.

Ví dụ:

số −5 được biểu diễn trong máy tính theo phương pháp bù 1 như sau (với mẫu 8 bit): đầu tiên, xác định số 5 được biểu diễn trong máy tính: 0000 0101. Tiếp theo, đảo tất cả các bit có trong số 5: kết quả sau khi đảo: 1111 1010. Vì là biểu diễn số âm nên bit bên trái cùng luôn giữ là 1.

Vậy với phương pháp bù 1, số −5 được biểu diễn trong máy tính như sau: 1111 1010.

1. Thực hiện phép cộng với số bù 1

Khi thực hiện phép tính cộng với số âm biểu diễn theo phương pháp bù 1, ta thực hiện như phép cộng nhị phân bình thường. Trong trường hợp khi đã thực hiện phép cộng đến bit cực trái mà vẫn phát sinh bit nhớ thì ta cộng tiếp bit nhớ này vào kết quả vừa nhận được.

Ví dụ:

1. Cộng hai số thập phân −5 với 2 (mẫu 8 bit):

1111 1010 (số bù 1 của −5)

+ 0000 0010 (số 2 ở hệ nhị phân)

===========

1111 1100(số bù 1 của −3)

2. Cộng hai số thập phân −5 với −7 (mẫu 8 bit):

1111 1010 (số bù 1 của −5)

+ 1111 1000 (số bù 1 của −7)

===========

1111 0010 (còn nhớ 1)

+ 1 (cộng tiếp với bit nhớ)

===========

1111 0011 (số bù 1 của −12)

Ta thấy: khi cộng hai bit cực trái của hai số 1111 1010 và 1111 1000, ta được kết quả là 1111 0010 và còn nhớ 1, do đó, ta tiếp tục cộng bit nhớ vào kết quả vừa nhận được để ra kết quả cuối cùng.

1. Tràn số

Xét trường hợp ta đang có hai số âm −5 và −6 ở hệ thập phân. Biểu diễn nhị phân bằng phương pháp bù 1 với mẫu 4 bit của hai số trên lần lượt là 1010 và 1001. Giả sử, bây giờ, ta cần cộng hai số này.

Ta thực hiện phép cộng:

1010 (số bù 1 của −5)

+ 1001 (số bù 1 của −6)

======

0011 (còn nhớ 1)

+ 1

======

0100

Ta thấy, kết quả nhận được là 0100. Nếu đổi ra hệ thập phân, đây là số nguyên dương 4 chứ không phải −11 như mong đợi.

1. Bù 2

Bù 2 (tiếng Anh: two's complement) là một số trong hệ nhị phân là bù đúng (true complement) của một số khác. Một số bù 2 có được do đảo tất cả các bit có trong số nhị phân (đổi 1 thành 0 và ngược lại) rồi thêm 1 vào kết quả vừa đạt được.

Thực chất, số biểu diễn ở dạng bù 2 là số biểu diễn ở bù 1 rồi sau đó cộng thêm 1. Trong quá trình tính toán bằng tay cho nhanh người ta thường sử dụng cách sau: từ phải qua trái giữ 1 đầu tiên và các số còn lại bên trái số 1 lấy đảo lại (chỉ áp dụng cho số có bit cực phải là 1).

Phương pháp bù 2 thường được sử dụng để biểu diễn số âm trong máy tính. Theo phương pháp này, bit cực trái (là bit nằm bên trái cùng của byte) được sử dụng làm bit dấu (sign bit - là bit tượng trưng cho dấu của số) với quy ước: nếu bit dấu là 0 thì số là số dương, còn nếu nó là 1 thì số là số âm. Ngoài bit dấu này ra, các bit còn lại được dùng để diểu diễn độ lớn của số.

Ví dụ: số nguyên −5 ở hệ thập phân được biểu diễn trong máy tính theo phương pháp bù 2 như sau (với mẫu 8 bit):

* Bước 1: xác định số nguyên 5 ở hệ thập phân được biểu diễn trong máy tính là: 0000 0101.
* Bước 2: đảo tất cả các bit nhận được ở bước 1. Kết quả sau khi đảo là: 1111 1010.
* Bước 3: cộng thêm 1 vào kết quả thu được ở bước 2: kết quả sau khi cộng: 1111 1011.
* Bước 4: vì là biểu diễn số âm nên bit bên trái cùng luôn giữ là 1.

Vậy với phương pháp bù 2, số −5 ở hệ thập phân được biểu diễn trong máy tính như sau: 1111 1011.

Ngoài cách làm theo định nghĩa như trên ra, ta còn có thể áp dụng phương pháp bù 2 theo quy tắc sau: với biểu diễn nhị phân của một số dương cho trước, để biểu diễn số âm tương ứng, ta bắt đầu tìm từ phải sang trái cho đến khi gặp bit đầu tiên có giá trị 1. Khi gặp được bit này, ta đảo tất cả các bit từ ngay kề trước nó (tức trước bit có giá trị 1 vừa nói tới) cho đến bit cực trái, và luôn nhớ: bit cực trái là 1.

Ví dụ: ta cũng biểu diễn lại số nguyên −5 ở hệ thập phân sang hệ nhị phân theo quy tắc mới này (giả sử với mẫu 8 bit):

* Bước 1: xác định số nguyên 5 ở hệ thập phân được biểu diễn trong máy tính là: 0000 0101.
* Bước 2: bắt đầu tìm (từ phải qua trái) bit đầu tiên có giá trị 1, ta thấy, đó là bit thứ nhất (tính từ phải qua).
* Bước 3: đảo tất cả các bit nằm trước bit thu được ở bước 2. Kết quả nhận được: 1111 1011
* Bước 4: vì là biểu diễn số âm nên bit bên trái cùng luôn giữ là 1.

Vậy số −5 ở hệ thập phân được biểu diễn trong máy tính theo phương pháp bù 2 là: 1111 1011 (hoàn toàn giống như kết quả trong ví dụ trên).

Thực hiện phép cộng với số bù 2[sửa | sửa mã nguồn]Khi thực hiện phép tính cộng với số âm biểu diễn theo phương pháp bù 2, ta thực hiện như phép cộng nhị phân bình thường, tuy nhiên, trong trường hợp khi đã thực hiện phép cộng đến bit cực trái mà vẫn phát sinh bit nhớ thì ta bỏ bit nhớ này đi.

Ví dụ:

1. Cộng hai số thập phân −5 với 2 (mẫu 8 bit):

1111 1011 (số bù 2 của −5)

+ 0000 0010 (số 2 ở hệ nhị phân)

===========

1111 1101 (số bù 2 của −3)

2. Cộng hai số thập phân −5 với −7 (mẫu 8 bit):

1111 1011 (số bù 2 của −5)

+ 1111 1001 (số bù 2 của −7)

===========

1111 0100 (số bù 2 của −12)

Ta thấy: khi cộng hai bit cực trái của hai số 1111 1011 và 1111 1001, ta vẫn còn nhớ 1, tuy nhiên, trong kết quả và ta bỏ bit này đi.

Tràn số[sửa | sửa mã nguồn]Xét trường hợp ta đang có hai số âm −6 và −4 ở hệ thập phân. Biểu diễn nhị phân bằng phương pháp bù 2 với mẫu 4 bit của hai số trên lần lượt là 1010 và 1100. Giả sử, bây giờ, ta cần cộng hai số này.

Ta thực hiện phép cộng:

1010 (số bù 2 của −6)

+ 1100 (số bù 2 của −4)

======

0110

Ta thấy, kết quả nhận được là 0110. Nếu đổi ra hệ thập phân, đây là số nguyên dương 6 chứ không phải −10 như mong đợi.

1. Số quá N

Phương pháp biểu diễn số quá N – còn được gọi là biểu diễn số dịch (biased representation) – sử dụng một số nguyên N cho trước làm giá trị dịch ("dịch" hiểu nôm na theo nghĩa "sự dịch chuyển" hay "sự thiên lệch").

Theo phương pháp này, một giá trị thập phân (tức giá trị cần biểu diễn) sẽ được biểu diễn bằng dạng nhị phân của một số dương nào đó sao cho, giá trị của số dương này lớn hơn giá trị cần biểu diễn N đơn vị.

Ví dụ: giả sử cần biểu diễn giá trị 210 theo số quá 5 (mẫu 8 bit):

Bước 1: ta có:

* Giá trị cần biểu diễn: 2.
* N = 5.

Bước 2: xác định số dương lớn hơn 210 năm đơn vị, đó là số 7.

Vậy 210 sẽ được biểu diễn bằng dạng nhị phân của 7: 00000111.

Theo ví dụ trên, ta sẽ có bảng sau:

|  |  |  |
| --- | --- | --- |
| **Số thập** | **Giá trị thập phân của số quá 5** | **biểu diễn nhị phân** |
| −5 | 0 | 00000000 |
| −4 | 1 | 00000001 |
| −3 | 2 | 00000010 |
| −2 | 3 | 00000011 |
| −1 | 4 | 00000100 |
| 0 | 5 | 00000101 |
| 1 | 6 | 00000110 |
| 2 | 7 | 00000111 |
| 3 | 8 | 00001000 |
| 4 | 9 | 00001001 |
| 5 | 10 | 00001010 |
| 6 | 11 | 00001011 |
| 7 | 12 | 00001100 |
| 8 | 13 | 00001101 |
| 9 | 14 | 00001110 |
| 10 | 15 | 00001111 |

Ta thấy, 0 được biểu diễn bằng nhị phân của 5, và −5 được biểu diễn bằng nhị phân của 0. Tổng quát, 0 được biểu diễn bằng nhị phân của N, còn −N được biểu diễn bằng mẫu có tất cả các bit đều là 0.

Phương pháp này ngày nay còn được sử dụng rộng rãi để biểu diễn các số chấm động (floating point number), tiêu biểu là chuẩn số chấm động IEEE. Theo chuẩn này, các số chấm động có độ chính xác đơn (single-precision) 32 bit (như kiểu float của Java) có phần mũ (chính là số lượng ký số của phần nằm sau dấu chấm thập phân) được biểu diễn bằng số quá 127 với mẫu 8 bit, và các số chấm động có độ chính xác đôi (double-precision) 64 bit (như kiểu double của Java) có phần mũ biểu diễn bằng số quá 1023 với mẫu 11 bit.

### Biến và kiểu dữ liệu:

##### Các kiểu dữ liệu căn bản:

**LƯU Ý**

* **C++ phân biệt in hoa và in thường. Ví dụ: Var, var, VAR sẽ được tính như ba biến khác nhau.**
* **Tên biến phải mang một ý nghĩa nhất định, để thao tác hiệu quả và tránh mất thời gian kiểm tra lại ý nghĩa các biến  
  Tham khảo thêm cách đặt tên biến tại** [**Stdio Coding Convention - Level 1.**](#_Qui_tắc_soạn)

Khi nhắc tới một biến, ngoài tên biến ra, chúng ta cần quan tâm tới kiểu dữ liệu và giá trị. Kiểu dữ liệu mô tả loại dữ liệu chứa trong biến. Dữ liệu, hay giá trị của một biến thực ra cũng chỉ là một dãy số nhị phân được lưu trên vùng nhớ mà biến đó được cấp. Nhưng tùy theo kiểu dữ liệu mà những giá trị này sẽ được hiểu theo những cách khác nhau.

Ví dụ khi đọc một số nhị phân; cùng là một dãy số 1111, nhưng nếu đọc theo kiểu số nhị phân không dấu sẽ được giá trị 15, còn đọc theo kiểu số nhị phân có dấu thì sẽ được giá trị -1. Tham khảo thêm cách một giá trị  được lưu vào bộ nhớ tại [Little Endian Và Big Endian.](#_Little_Endian_Và)

Bên cạnh đó, biến của một kiểu dữ liệu nhất định cũng sẽ được cấp một số Bytes vùng nhớ nhất định, tùy theo hệ điều hành.

1. Kiểu dữ liệu C

Kiểu dữ liệu trong C phân thành 4 loại chính

|  |  |
| --- | --- |
| **S.N.** | **Types and Description** |
| 1 | **Basic Types:**  Là kiểu số học cơ bản gồm số nguyên integer và số hữu tỉ float-point. |
| 2 | **Enumerated types:**  Không phải kiểu số học và chúng được dùng để nghĩa biến chỉ có thể được gán một cách rời rạc trong chương trình |
| 3 | **Kiểu void:**  Là kiểu không xác định |
| 4 | **Derived types:**  Kiểu dẫn xuất gồm: con trỏ, mảng, cấu trúc, union và kiểu hàm. |

1. Dữ liệu kiểu số trong C:

|  |  |  |
| --- | --- | --- |
| **Kiểu** | **Dung lượng** | **Khoang giá trị** |
| char | 1 byte | -128 to 127  0 to 255 (mặc định) |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| int | 2 or  4 bytes | -32,768 to 32,767 or  -2,147,483,648 to 2,147,483,647 |
| unsigned int | 2 or  4 bytes | 0 to 65,535 or  0 to 4,294,967,295 |
| short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| long | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned long | 4 bytes | 0 to 4,294,967,295 |

1. Dữ liệu số có dấu chấm động

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Kiểu** | **Dung lượng** | **Value range** | **Độ chính xác** | **Fraction bit** |
| float | 4 byte | 1.2E-38 to 3.4E+38 | 6 số | 8-bits |
| double | 8 byte | 2.3E-308 to 1.7E+308 | 15 số | 52-bits |
| long double | 10 byte | 3.4E-4932 to 1.1E+4932 | 19 số |  |

header file <float.h> định nghĩa các macros kiểu số thực

Ví dụ sau sẽ hiển thị độ lớn dữ liệu kiểu float và độ dài

#include <stdio.h>

#include <float.h>

int main(){

printf("Storage size for float : %d \n", sizeof(float));

printf("Minimum float positive value: %E\n", FLT\_MIN );

printf("Maximum float positive value: %E\n", FLT\_MAX );

printf("Precision value: %d\n", FLT\_DIG );

return 0;

}

Kết quả

Storage size for float : 4

Minimum float positive value: 1.175494E-38

Maximum float positive value: 3.402823E+38

Precision value: 6

1. Dữ liệu kiểu Void

Dữ liệu kiểu Void không có giá trị xác định . Nó được dùng trong 3 tình huống sau

|  |  |
| --- | --- |
| **S.N.** | **Types and Description** |
| 1 | **Hàm trả về kiểu void**  Hàm này không trả về giá trị gì cả. |
| 2 | **Đối số hàm kiểu void**  Ám chỉ, hàm chấp nhận mọi kiểu dữ liệu truyền vào |
| 3 | **Con trỏ kiểu void**  Con trỏ void chỉ đại diện cho địa chỉ của đối tượng nó trỏ tới, không phải kiểu dữ liệu.  Con trỏ void dễ dàng được ép kiểu sang các dữ liệu khác. |

Ví dụ:

#include <stdio.h>

#include <stdlib.h>

void noReturn(void arg);

void noValue(void arg);

int main(){

int z = 5;

int \*pointer = &z;

noReturn(pointer);

noValue(&z);

return 0;

}

void noReturn(void\* arg){ arg = NULL;}

void noValue(void\* arg){ arg = 0;}

Đối số hàm **noReturn**() chấp nhận mọi kiểu con trỏ.

Còn đối số hàm **noValue**() chấp nhận mọi kiểu dữ liệu.

1. Kiểu dữ liệu C++
2. Các kiểu dữ liệu chính

C++ tích hợp một bộ các kiểu dữ liệu đa dạng cũng như kiểu người dùng tự định nghịa. Bảng sau liệt kê 7 kiểu cơ bản:

|  |  |
| --- | --- |
| **Kiểu** | **Từ khóa** |
| Boolean – kiểu chân trị | bool |
| Character – kiểu kí tự 1 byte | char |
| Integer – kiểu số nguyên | int |
| Floating point – kiểu dấu chấm động | float |
| Double floating point – kiếu dấu chấm động | double |
| Valueless – Kiểu void | void |
| Wide character – Kiểu kí tự 2/4 byte | wchar\_t |

Các từ khóa bổ nghĩa đứng trước kiểu dữ liệu:

* Signed : số nguyên có dấu (dương hay âm), áp dụng cho kiểu int, char.
* Unsigned: số nguyên không dấu (dương), áp dụng cho kiểu int, char
* Short : kích thước ít hơn , áp dụng cho kiểu int
* Long : kích thước tăng thêm, áp dụng cho kiểu int

Bảng sau liệt kê kiểu dữ liệu, kích thước bộ nhớ và khoảng giá trị mỗi kiểu.

|  |  |  |
| --- | --- | --- |
| **Kiểu** | **Kích thước** | **Khoảng giá trị** |
| char | 1byte | -127 to 127 or 0 to 255 |
| unsigned char | 1byte | 0 to 255 |
| signed char | 1byte | -127 to 127 |
| int | 4bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned int | 4bytes | 0 to 4,294,967,295 |
| signed int | 4bytes | -2,147,483,648 to 2,147,483,647 |
| short int | 2bytes | -32,768 to 32,767 |
| unsigned short int | Range | 0 to 65,535 |
| signed short int | Range | -32768 to 32767 |
| long int | 4bytes | -2,147,483,647 to 2,147,483,647 |
| signed long int | 4bytes | same as long int |
| unsigned long int | 4bytes | 0 to 4,294,967,295 |
| float | 4bytes | +/- 3.4e +/- 38 (~7 digits) |
| double | 8bytes | +/- 1.7e +/- 308 (~15 digits) |
| long double | 8bytes | +/- 1.7e +/- 308 (~15 digits) |
| wchar\_t | 2 or 4 bytes | 1 wide character |

Kích thước kiểu dữ liệu có thể sẽ khác tùy theo trình biên dịch và hệ máy bạn đang dùng.

Ví dụ sau sẽ chỉ rõ kích thước từng kiểu dữ liệu trên máy của bạn.

#include <iostream>

using namespace std;

int main(){

cout << "Size of char : " << sizeof(char) << endl;

cout << "Size of int : " << sizeof(int) << endl;

cout << "Size of short int : " << sizeof(short int) << endl;

cout << "Size of long int : " << sizeof(long int) << endl;

cout << "Size of float : " << sizeof(float) << endl;

cout << "Size of double : " << sizeof(double) << endl;

cout << "Size of wchar\_t : " << sizeof(wchar\_t) << endl;

return 0;

}

Từ khóa **endl**, sau toán tử **<<** sẽ ngắt dòng. Dùng toán tử **sizeof()** điể lấy kích thước.

Kết quả:

Size of char : 1

Size of int : 4

Size of short int : 2

Size of long int : 4

Size of float : 4

Size of double : 8

Size of wchar\_t : 4

1. Định nghĩa kiểu dữ liệu

Bạn có thể dùng **typedef** để khai báo kiểu dữ liệu mới.

Cú pháp:

typedef type newname;

Ví dụ sau khai báo với trình biên dịch rằng **feet** là kiểu dữ liệu mới dạng **int.**

typedef int feet;

Đoạn khai báo sau sẽ là hợp lệ với biến **distance** kiểu **feet**:

feet distance;

##### Các kiểu dữ liệu thêm vào trong C99

1. Chuẩn số nguyên fixed-size

Chuẩn C-99 đưa ra giải pháp là định nghĩa thêm các kiểu dữ liệu số nguyên mới : **int8\_t, int16\_t, int32\_t** và **int64\_t**

Cùng với kiểu không dấu tương ứng : **uint8\_t, uint16\_t, uint32\_t** và **uint64\_t,** kích thước kiểu tương đương là : **8**, **16**, **32** và **64**-**bits**

|  |  |  |  |
| --- | --- | --- | --- |
| **Kiểu dữ liệu** | **Kích thước** | **Độ rộng** | **Chuỗi định dạng** |
| Int8\_t | 1-bytes |  | %d |
| Int16\_t | 2-bytes |  | %d |
| Int32\_t | 4-bytes |  | %d |
| Int64\_t | 8-bytes |  | %d |
| Uint8\_t | 1-bytes |  | %d |
| Uint16\_t | 2-bytes |  | %d |
| Uint32\_t | 4-bytes |  | %d |
| Uint64\_t | 8-bytes |  | %d |

1. Kiểu long long

Sẽ là kiểu dữ liệu gốc trong C++11.

|  |  |  |  |
| --- | --- | --- | --- |
| **Kiểu dữ liệu** | **Kích thước** | **Độ rộng** | **Chuỗi định dạng** |
| long long int | 8-bytes | -263 tới +263 -1 | %lld |
| Unsigned long long int | 8-bytes | 0 tới 264 -1 | %llu |

1. Kiểu wchar\_t

Chuẩn mở rộng C99 định nghĩa kiểu **wchar**\_**t** (“wide character”) để chứa kí tự từ các bộ kí tự dùng nhiều hơn 1-byte để mã hóa. Thêm vào đó, Các thư viện hàm được định nghĩa để bổ sung thêm chức năng chuyển đổi giữa các chuỗi byte và mảng kiểu **wchar**\_**t,** theo cả hai chiều. (Trên thực tế, có tồn tại phiên bản thư viện hỗ trợ **wchar**\_**t** mà có thể làm việc cùng chuỗi kí tự).

|  |  |  |  |
| --- | --- | --- | --- |
| **Kiểu dữ liệu** | **Kích thước** | **Độ rộng** | **Chuỗi định dạng** |
| **Wchar\_t** | 2-bytes | 0 tới 65535 | **%c** |

1. Kiểu Bool

Chuẩn C99 định nghĩa kiểu **Bool** trong thư viễn <stdbool.h>**,** thay cho kiểu **\_Bool**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Kiểu dữ liệu** | **Kích thước** | **Độ rộng** | **Chuỗi định dạng** |
| **bool** | 1-bytes | 1 là true  0 là false | **%d** |

##### Khai báo và khởi tạo biến

C++ yêu cầu mọi biến phải được khai báo trước khi sử dụng.

Khai báo một biến tức là thông báo với trình biên dịch rằng đã có một biến với kiểu dữ liệu và tên biến đã được sử dụng cũng như vùng nhớ cần cấp tương ứng với kiểu dữ liệu cho biến đó. Để khai báo biến trong C++, ta dùng cú pháp đơn giản sau:

1. kiểu\_dữ\_liệu tên\_biến;

Kiểu dữ liệu ở đây có thể là kiểu dữ liệu có sẵn trong C++ hoặc kiểu dữ liệu do người dùng tự định nghĩa. Có thể khai báo một biến hoặc nhiều biến một lúc với tên biến cách nhau bởi dấu phẩy:

1. int a;
2. char b, c;
3. float d, e, f;
4. Khởi tạo tường minh

Khi một biến đã được khai báo, giá trị của nó vẫn là chưa xác định, ta có thể khởi tạo giá trị ban đầu cho biến ngay khi khai báo bằng những cách như sau:

1. int number = 5;
2. char letter('b');

Hoặc cũng có thể khởi tạo giá trị sau khi khai báo như sau:

1. int number1, number2;
2. char letter;
4. number1 = 5;
5. number2 = 8;
6. letter = 'b';
8. // Tinh toan co ban tren bien
9. int sum1 = number1 + 6;
10. int sum2 = number1 - number2;

Cách gán giá trị cho biến khi vừa khởi tạo như trên gọi là khởi tạo tường minh

1. Khởi tạo ngầm

Trong lập trình hướng đối tượng, bạn sẽ gặp trường hợp khai báo như sau:

Int value(5);

Khởi tạo bên trên được gọi là khởi tạo ngầm (không tường minh), biến ở đây là đối tượng kiểu số nguyên, nó khai báo giống như là gọi hàm .

Trình biên dịch sẽ tìm hàm dựng thông qua tên biến khớp với nhau.

Khởi tạo ngầm trong hướng đối tượng có vài ưu điểm, nhưng chúng ta sẽ xem xét khi đi vào **class**.

1. Khai báo biến giữa C và C++

C yêu cầu tất cả biến phải khai báo ngay đầu hàm. Nhưng trình biên dịch C++ cho phép bạn khai báo vị trí bất kì trong hàm, miễn là trước khi dùng

Cách khai báo tùy hứng trong C++ có vài ưu điểm:

* Biến chỉ được định nghĩa khi cần trong phạm vi nào đó, và hiển nhiên là nó không ảnh hưởng tới các mã lệnh bên trên.
* Thứ hai là hạn chế trường hợp khai báo biến mà không dùng tới.

Chúng ta khuyến khích định nghĩa biến ngay trước khi dùng tới hơn là đầu hàm.

##### Tương thích giữa các kiểu dữ liệu

Khi làm việc với biến, chúng ta còn gặp phải nhiều vấn đề, trong đó có vấn đề về ép kiểu (type casting), các bạn có thể tham khảo thêm tại [Cơ Bản Về Chuyển Kiểu Trong C.](#_Cơ_bản_về)

##### Demote và promote kiểu dữ liệu

Trong trường hợp kiểu dữ liệu bên trái hay bên phải phép gán không giống, giá trị biểu thức phải được demoted hay promoted cho phù hợp kiểu biến bên trái.

Ví dụ 1:

float a; a = 5; // 5 là kiểu int, không thể chứa trong float, nên được promoted sang kiểu float thành 5.000000

int b; b = 10.25;//10.25 là kiểu float, do đó bị demoted xuống kiểu int thành 10.

Ví dụ 2:

Float x, z; int a, y;

a = z \* y \* x/50 + 60/2 – 5\*1.1;

Vế phải biểu thức vẫn tuân theo luật: tất cả số int được promote lên kiểu float trước, kết quả sẽ là kiểu float nên bị demote xuống kiểu int.

### Toán tử, độ ưu tiên và trật tự kết hợp:

Toán tử - Thành phần không thể thiếu trong bất cứ ngôn ngữ lập trình nào, hầu hết các hàm đều có ít nhất một câu lệnh sử dụng tới toán tử. Nắm được các dạng toán tử cùng độ ưu tiên và trật tự kết hợp sẽ giúp cho lập trình viên giải quyết bài toán của mình nhanh chóng hơn.

##### Các loại toán tử

Toán tử được chia thành các nhóm chính:

* Toán tử số học
* Toán tử so sánh
* Toán tử logic
* Toán tử bitwise
* Toán tử gán
* Toán tử hỗn hợp

1. Toán tử số học
2. Các toán tử quen thuộc

| **TOÁN TỬ** | **MÔ TẢ** |
| --- | --- |
| + | Cộng |
| - | Trừ |
| \* | Nhân |
| / | Chia |
| % | Chia lấy số dư (chỉ dành cho số nguyên) |

1. Toán tử ++ và --

Đây là hai toán tử một ngôi, hai toán tử này tùy vào vị trí (tiền tố hoặc hậu tố) mà sẽ đưa ra các kết quả khác nhau:

* Tiền tố (đứng ngay trước tên biến): tăng (nếu là ++) hoặc giảm (nếu là --) giá trị của biến lên 1 rồi mới thực hiện tính toán trong biểu thức.
* Hậu tố (đứng ngay sau tên biến): thực hiện tính toán biểu thức rồi mới tăng (nếu là ++) hoặc giảm (nếu là --)  giá trị của biến lên một.

Ta lấy ví dụ sau:

1. int main()
2. {
3. int x = 10;
4. int y = 10;
6. int a, b;
7. a = ++x; // Tiền tố
8. b = y++; // Hậu tố
10. cout << "a = " << a << endl;
11. cout << "x = " << x << endl;
12. cout << "b = " << b << endl;
13. cout << "y = " << y << endl;
15. return 0;
16. }

Ta có kết quả xuất ra màn hình như sau:

1. a = 11
2. x = 11
3. b = 10
4. y = 11
5. Toán tử so sánh

Là các toán tử thường dùng trong các biểu thức đi kèm với các câu lệnh điều kiện.

Biểu thức chứa phép toán so sánh sẽ trả về kết quả logic là true (đúng) hoặc false (sai).

Bao gồm các toán tử sau: ==, !=, >, <, >=, <=.

Giả sử ta có A và B là hai số cần so sánh:

| **BIỂU THỨC** | **KẾT QUẢ** |
| --- | --- |
| A == B | true nếu A bằng B |
| A != B | true nếu A khác B |
| A > B | true nếu A lớn hơn B |
| A < B | true nếu A bé hơn B |
| A >= B | true nếu A lớn hơn hoặc bằng B |
| A <= B | true nếu A bé hơn hoặc bằng B |

1. Toán tử logic

Gồm các phép toán &&, ||, !

Đây là các biểu thức dùng để kết hợp kết quả của các biểu thức so sánh (các biểu thức sử dụng toán tử so sánh) để trả về các kết quả logic là true hoặc false.

Giả sử ta có X và Y là hai biểu thức so sánh:

| **BIỂU THỨC** | **KẾT QUẢ** |
| --- | --- |
| X && Y | true nếu X true và Y true |
| X || Y | true nếu X true hoặc Y true |
| !Y | true nếu X false |

**Lưu ý: trong trường hợp X hoặc Y là 2 số, biểu thức vẫn thực hiện được, với quy ước: 0 tương đương với false, các số khác 0 tương đương với true.**

1. Toán tử bitwise

Là các toán tử xử lý tính toán liên quan đến các dãy bit nhị phân.

1. Toán tử &, |, ^

Đây là các toán tử 2 ngôi với 2 số hạng là 2 dãy bit

* &: đọc là AND, kết quả của biểu thức là 1 nếu cả 2 bit số hạng là 1, ngược lại kết quả là 0.
* | : đọc là OR, kết quả của biểu thức là 1 nếu một trong 2 bit số hạng là 1, ngược lại kết quả là 0.
* ^: đọc là XOR, kết quả của biểu thức là 1 nếu 2 bit số hạng khác nhau, ngược lại kết quả là 0.

Ví dụ ta có p và q là 2 bit cần đưa vào tính toán:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| p | q | p & q | p | q | p ^ q |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

1. Toán tử ~

~ (đọc là NOT) là toán tử một ngôi, tiền tố, kết quả thu được là dãy bit bị đổi 0 thành 1 và 1 thành 0 so với dãy bit ban đầu

Ví dụ tổng quát cho các toán tử bitwise

Ta có khai báo:

1. char A = 71;
2. char B = 19;
3. char Result;

Chuyển A và B thành các dãy bit:

A = 00100111

B = 00010011

Nếu:

1. Result = A & B // Result = 00000011
2. Result = A | B // Result = 00110111
3. Result = A ^ B // Result = 00110100
4. Result = A << 3 // Result = 00111000
5. Result = A >> 3 // Result = 00000100
6. Result = ~A // Result = 11011000
7. Toán tử dịch bit

Gồm 2 toán tử là << và >> (khác với << trong của cout và >> của cin)

* A << n: kết quả thu được là dãy bit A đã dịch sang trái n bit, dễ hiểu hơn, kết quả là dãy bit A bỏ đi n bit bên trái cùng và thêm n bit 0 vào bên phải cùng.
* A >> n: kết quả thu được là dãy bit A đã dịch sang phải n bit, dễ hiểu hơn, kết quả là dãy bit A bỏ đi n bit bên phải cùng và thêm n bit 0 vào bên trái cùng.

Dịch trái 1-bit tương đương nhân với 2. Dịch 2-bits tương đương nhân với 4. Quy tắc :

* Dịch n bit sang trái tương ứng nhân 2n
* Dịch n bit sang phải tương ứng chia 2n

Thỉnh thảng người ta dùng toán tử dịch bit vì nó nhanh hơn so với phép nhân, chia thông thường

i = j << 3; // nhân jvới 8 (2^3)

nhanh hơn

i = j \* 8;

Hay trình biên dịch không đủ thông minh để hiểu là “nhân số mũ 2” thành “dịch trái”. Tuy nhiên chúng ta không nên làm vậy để tăng tốc độ, trình biên dịch ngày nay đủ thông minh để tự tối ưu hóa chương trình.

1. Dịch phải và bit dấu

Có một điều cần lưu ý khi dùng toán tử (>>) để dịch phải bit với số nguyên có dấu. Mặc định số nguyên không dấu sẽ được chèn bit zero ở bên trái cùng.

Đối với số nguyên có dấu: trình biên dịch sẽ chèn bit dấu vào bên trái cùng.

1. Toán tử gán

Là các toán tử thực hiện phép gán và có thể thực hiện biểu thức đi kèm với phép gán đó, bao gồm =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=

Trong đó:

* = là toán tử gán thông thường, gán biến bên trái bằng với giá trị của biểu thức bên phải.
* Các toán tử còn lại: giả sử ta có biểu thức A += B, thì biểu thức này tương đương với A = A + B. Các toán tử còn lại hoàn toàn tương tự.

1. Toán tử hỗn hợp

Là các toán tử được C++ hỗ trợ thêm như sizeof, chuyển kiểu, ->, new, delete… Các toán tử dạng này sẽ được đề cập rõ hơn ở các bài viết sau.

##### Độ ưu tiên và trật tự kết hợp

Trong một biểu thức sử dụng nhiều toán tử, thì mỗi loại toán tử sẽ có độ ưu tiên khác nhau, toán tử có độ ưu tiên lớn hơn sẽ được tính toán trước, khi các toán tử có độ ưu tiên ngang hàng, thì sẽ tính toán theo thứ tự từ trái sang phải hoặc từ phải sang trái, tùy thuộc vô nhóm toán tử.

Ta có bảng chi tiết sau, độ ưu tiên sẽ giảm dần theo thứ tự từ trên xuống dưới:

| **ĐỘ ƯU TIÊN** | **NHÓM TOÁN TỬ** | **TOÁN TỬ** | **THỨ TỰ THỰC HIỆN** |
| --- | --- | --- | --- |
| 1 | Phạm vi | :: | Trái sang phải |
| 2 | Hậu tố (một ngôi) | ++ -- | Trái sang phải |
| ( ) |
| [ ] |
| . -> |
| 3 | Tiền tố (một ngôi) | ++ -- | Phải sang trái |
| ~ ! |
| + - |
| & \* |
| new detete |
| sizeof |
| (type) |
| 4 | Trỏ tới thành viên | .\* ->\* | Trái sang phải |
| 5 | Số học: tỉ lệ | \* / % | Trái sang phải |
| 6 | Số học: tăng giảm | + - | Trái sang phải |
| 7 | Dịch bit | << >> | Trái sang phải |
| 8 | So sánh hơn | < > <= >= | Trái sang phải |
| 9 | So sánh bằng | == != | Trái sang phải |
| 10 | Bitwise AND | & | Trái sang phải |
| 11 | Bitwise XOR | ^ | Trái sang phải |
| 12 | Bitwise NOT | | | Trái sang phải |
| 13 | Logical AND | && | Trái sang phải |
| 14 | Logical OR | || | Trái sang phải |
| 15 | Gán | = \*= /= %= += -= &= |= ^= >>= <<= | Phải sang trái |
| ?: |
| 16 | Sắp xếp | , | Trái sang phải |

### Binary và Hexa decimal

1. Giới thiệu:

Nắm rõ hệ nhị phân và thập lục phân là điều cốt lõi của lập trình mức độ hệ thống. Máy tính chỉ làm việc với hệ nhị phân – gồm số 1 và 0.

Hệ thập lục phân thuận tiện vì chúng cho phép chúng ta xử lí số nhị phân dễ dàng hơn.

Hệ thống số chúng ta tiếp xúc thường xuyên là thập phân, gồm các số từ 0 tới 9. Vì thế, số thập phân thường được xem là hệ số mười cơ bản.

Hệ số cơ bản tiếp theo là hệ bát phân gồm các số từ 0 tới 7, hay còn gọi là octal.

Với hệ bát phân, chúng ta có thể làm các phép toán như cộng, trừ ,nhân và chia, giống như thập phân. Chúng ta có thể nói rằng 2 + 5 = 7, hay 4 – 1 = 3.

Vậy nếu có số > 7 thì sao. Chúng ta đếm trong hệ bát phân thế nào?

Hãy xem cách đếm trong hệ thập phân trước, bỏ qua số 0 khi ta đếm: 1, 2, 3, 4, 5, 6, 7, 8, 9. Số tiếp theo nằm ngoài phạm vi bảng chữ số thập phân. Nên ta đặt số 1 vào hàng chục, số 0 vào hàng đơn vị, ta có số 10. Sau đó đếm tiếp: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19. Lại tiếp tục, ta tang hàng chục lên 1 đơn vị thành 2, ta đặt lại hàng đơn vị về 0. Tiếp tục đếm: 20, 21, 22, 23, 24, 25, … và cứ thế. Khi tới 99, ta cạn số cho cả hàng chục và đơn vị, ta lại đặt số 1 vào hàng trăm và số 0 vào hàng chục , hàng đơn vị. Ta có số 100.

Một cách nghĩ khác là hình dung tất cả các vị trí để trống có thể có trong số đều là 0. Sau khi chúng ta hết số ở hàng đơn vị, ta cộng thêm 1 vào vị trí bên trái, và thay đổi các vị trí còn lại về 0. Khi vị trí hàng chục hết số, ta cộng thêm 1 vào vị trí bên trái và tiếp tục. Vậy điểm chính là khi đếm, nếu một vị trí hết số, đặt vị trí đó về 0 và tăng một ở vị trí bên trái.

Chúng ta áp dụng luật trên với hệ bát phân. Khi đếm qua 7, ta tăng hàng chục lên 1, đặt hàng đơn vị về 0. Ta có số 10 hệ bát phân (tương đương số 8 hệ thập phân). Tiếp tục đếm qua số 17, ta lại tăng 1 lên hàng chục, đặt hàng đơn vị về 0. Ta có số 20.

Như ta thấy, rất khó theo dõi giá trị tương ứng giữa hệ bát phân và thập phân, phương pháp chuyển hệ số sẽ được đề cập sau.

Để phân biệt chữ số thuộc hệ nào, ta thêm phần đuôi để phân biệt. Ví dụ: “oct” hay “OCT” cho hệ bát phân, “dec” hay “DEC” cho hệ thập phân.

Thỉnh thoảng bạn sẽ thấy “q”,”Q” sau số bát phân. Hay số “8” phía trước hay sau số bát phân.

Ta xem một bảng chứa 24 giá trị đầu của hệ bát và thập phân.

Dec: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Oct: 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 20 21 22 23 24 25 26 27 30

Ta thấy 12 oct + 16 oct = 30 oct (bạn tham chiếu lên số dec cho dễ tính)

1. Hệ nhị phân

Hệ bát phân được sử dụng không nhiều trong điện tử và lập trình, nên ta sẽ không đề cập tới nhiều. Phần này sẽ giúp bạn làm quen với hệ nhị phân.

Hệ nhị phân là hệ số gồm 2 chữ số 0 và 1. Quy tắc đếm cũng tuân theo luật đếm ở phần trước.

Đây là bảng liệt kê các chữ số hệ thập phân, bát phân và nhị phân:

**Decimal Octal Binary**

------------------------------------------------

0 0 0

1 1 1

2 2 10

3 3 11

4 4 100

5 5 101

6 6 110

7 7 111

8 10 1000

9 11 1001

10 12 1010

11 13 1011

12 14 1100

13 15 1101

14 16 1110

15 17 1111

16 20 10000

Quy tắc viết số nhị phân là thêm "bin" hay "BIN" phía sau chữ số. Bạn cũng có thể thêm “b” hay “B” phía sau cùng với số 0 phía trước

Ví dụ: Ta có số 1010: “01010b” hay “01010B”

Khi thực hiện phép toán trên hệ nhị phân, bạn cần dùng mẹo. Ta quy mọi phép toán về phép cộng. Phép trừ đơn giản là cộng với số âm (dùng phương pháp bù 2 để biểu diễn số âm trong máy tính). Phép nhân thực hiện bằng cách lặp phép cộng, phép chia cũng tương tự là lặp phép trừ.

Phép cộng cũng áp dụng theo luật trong hệ thập phân, ví dụ:

|  |  |  |  |
| --- | --- | --- | --- |
| Hạng tử 1 |  | 0 | 1 |
| Hạng tử 2 | 0 | 0 | 1 |
| Kế t quả | 1 | 1 | 10(\*) |

(\*) là "0, và nhớ 1"

Cộng 01100100bin với 01110101bin:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hạng tử 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| Hạng tử 2 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| Kế t quả | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

1. Hệ thập lục phân

Thỉnh thoảng bạn sẽ thấy tẻ nhạt, mệt mỏi khi phải viết và nhớ chuỗi nhị phân dài dòng, khó nhớ. Nhưng trong lập trình, việc tiếp xúc với số nhị phân (các số nguyên 8, 16, 32, 64-bits) là bình thường.

Nhưng hệ thập lục phân thuận tiện hơn nhiều so với nhị phân. Đây là một hệ số căn bản khác của hệ thống, bảng alphabet gồm 16 chữ số từ 0 tới 9 và A, B, C, D, E, F.

Hệ thập lục cũng áp dụng luật đếm như phần trước.

Quy cách viết là thêm “hex” hay “HEX” phía sau. Hoặc “h” hay “H” phía sau như : 3F9B hex" , "03F9Bh”.

Đây là bảng liệt kê các hệ số

**Decimal Octal Binary Hexadecimal**

0 0 0 0

1 1 1 1

2 2 10 2

3 3 11 3

4 4 100 4

5 5 101 5

6 6 110 6

7 7 111 7

8 10 1000 8

9 11 1001 9

10 12 1010 A

11 13 1011 B

12 14 1100 C

13 15 1101 D

14 16 1110 E

15 17 1111 F

16 20 10000 10

17 21 10001 11

18 22 10010 12

19 23 10011 13

20 dec 24 oct 10100 bin 14 hex

Như ta thấy, hệ này khá phức tạp hơn là hệ thập phân. Nhưng nếu bạn so sánh kỹ những số hex 2 chữ số cùng với số nhị phân bên cạnh, 0hex và Fhex tương ứng với 0000bin và 1111bin, 10hex và 1Fhex tương ứng với 1 0000bin và 1 1111bin.

Thực tế khi ta mở rộng bảng, quy trình tiếp tục lặp lại sau mỗi 16 số thập lục phân. Cứ mỗi 4 bit “0000” ở cuối là ta có “0” ở vị trí tương ứng, “0001” tương ứng với “1”, “0002” tương ứng với “2”… ,”1111” tương ứng với “F”

Vậy là ta đã khám phá ra mối liên hệ giữa số thập lục phân và nhị phân. 1 bộ 4 bit nhị phân tương ứng với 1 đơn vị số hex, ta có phương pháp để chuyển từ hệ nhị phân sang hệ thập lục là cứ 4 đơn vị nhị phân làm thành 1 đơn vị hex từ phải qua trái.

Ví dụ:

0 1 0 0 1 1 0 1 0 0 1 1 0 1 0 1 : 4D 35hex

1100 1111 0101 1001 : CF 59hex

Chuyển từ hệ 16 sang hệ nhị cũng tượng tự như vậy

Để dễ dàng khi làm các phép toán + - \* / trên hệ 16, bạn chuyển về hệ 10, tính toán rồi chuyển lại hệ 16.

1. Chuyển đổi số từ nhiều hệ khác nhau

Phần này chúng ta sẽ thảo luận các phương pháp chuyển số giữa hệ 10 và hệ 2, một trong những thao tác phổ biến và cần thiết. Qua đó ta rút ra cách chuyển đổi tổng quát cho các hệ khác.

We have discussed converting numbers to and from binary and hexadecimal, but those tricks don't usually work with other number bases. Here I will describe methods for converting numbers to and from decimal and binary, which are probably amongst the most common converting tasks. I will also briefly show how these methods can be generalized for use with other number bases.

Imagine that we want to know what 110101 bin represents in decimal. We could extend our conversion table given above until we reached this particular binary number, but of course that's impractical. But let's consider how we could re-write a decimal number first. Say we have the number 576 dec. We can re-write this as 500 + 70 + 6, or:

2 1 0

5 \* 10 + 7 \* 10 + 6 \* 10

Let's consider the exponents (the superscripts). We can refer to each place in the number by its exponent, so the units place is place 0, the tens place is place 1, the hundreds place is place 2, and so on. Now notice that the bases in that expression are 10's. Recall that decimal is a base-ten system. Given this hint, we can try expressing numbers in other bases in a similar form, say, in binary. Let's take our 110101 bin example:

5 4 3 2 1 0

110101 bin = 1 \* 2 + 1 \* 2 + 0 \* 2 + 1 \* 2 + 0 \* 2 + 1 \* 2 dec

A 2 is used instead of 10 because binary is a base-two system. Now we can evaluate the expression we have constructed. (Remember that a number to the power of zero equates to one. And, of course, multiplying anything by zero equates that term to zero.) So, evaluating this expression, we get 32 + 16 + 0 + 4 + 0 + 1, or 53 dec.

Basically, we can use method this to convert a number in any base to base ten -- just change the 2 to the number of the base. For example, to change a hexadecimal number to decimal, use 16 instead of 2, because hex is a base-sixteen number system. Here's an example: we'll convert 3DA hex to decimal:

2 1 0

3DA hex = 3 \* 16 + 13 \* 16 + 10 \* 16 = 242 dec

Notice that I converted D hex to 13 dec, and A hex to 10 dec. (If you wish, you can use the conversion table.) Then, it becomes a simple matter to evaluate the expression.

Another method which I like to use is to draw a quick diagram. Here is the number 01100011 bin, surrounded in boxes. Above each box is the base of the number, with each base having an exponent equal to the number of the place which it represents, that is:

7 6 5 4 3 2 1 0

2 2 2 2 2 2 2 2

+---+---+---+---+---+---+---+---+

| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

+---+---+---+---+---+---+---+---+

Now it is easy to look at the digits inside the boxes, and for each one that is not a zero, you can add to a total the product of that digit and the expression above it. So for the above example, we get 1 \* 2^6, plus 1 \* 2^5, plus 1 \* 2^1, plus 1 \* 2^0, and the sum of these products is 99 dec.

To save time, you can write the numbers 1, 2, 4, 8, 16, etc. above the boxes instead of the base and exponents:

2^0 = 1

2^1 = 2

2^2 = 4

2^3 = 8

2^4 = 16

2^5 = 32

2^6 = 64

2^7 = 128, etc.

We'll be seeing more diagrams like these in the further chapters.

Finally, we often want to convert decimal numbers to binary (or some other base). To do this, we can repeatedly divide the decimal number by the base we to which we are converting (so for binary, we keep dividing the decimal number by 2). We can use the remainders (also called modulos) of each division to construct our new number. As this is difficult to explain, I'll show some examples. Say we want to convert 235 dec to binary. We start by dividing 235 dec by the base we wish to convert to, which is 2. Then we divide that result by 2, and then we divide that result by 2, and so on. We stop when we get a zero as a quotient. We must keep track of the remainders from each step:

235 / 2 = 117, remainder 1

117 / 2 = 58, remainder 1

58 / 2 = 29, remainder 0

29 / 2 = 14, remainder 1

14 / 2 = 7, remainder 0

7 / 2 = 3, remainder 1

3 / 2 = 1, remainder 1

1 / 2 = 0, remainder 1.

Now, if we read the remainders backwards (from the bottom to the top), we get 11101011 bin. This is how you would represent 235 dec in binary.

Let's try converting 49 dec to octal:

49 / 8 = 6, remainder 1

6 / 8 = 0, remainder 6.

Going from bottom to top, we read off a 6 and then a 1. We get 61 oct, so 49 dec = 61 oct.

An alternate method, which can be faster in some cases and slower in others, is to draw the box diagram and then try plugging in combinations of numbers in the boxes until the sum of the products of the numbers in the boxes and the expression above them equals your decimal number. For small numbers, this is much quicker: in a decimal to binary conversion, you can tell very quickly that, say, 9 dec can be created by adding an 8 and a 1; you can then put ones in the places associated with the 8 and 1, and put zeroes everywhere else. We'll become more familiar with these diagrams in the later chapters, so don't worry if this is confusing right now.

### Cơ bản về chuyển đổi kiểu

1. Giới thiệu

Qua [Bản Chất Của Biến Trong C/C++](#_Bản_chất_của) , chúng ta hiểu được rằng giá trị của biến được lưu trên các dãy bit, và kiểu dữ liệu của một biến cho trình biên dịch biết phải chuyển các dãy bit đó thành giá trị phù hợp.

Tuy nhiên có một số trường hợp cần chuyển giá trị thành một kiểu dữ liệu khác để phù hợp trong quá xử lý, quá trình đó gọi là chuyển kiểu (hay ép kiểu, lưu ý vấn đề overflow khi ép kiểu)

1. Chuyển kiểu ngầm định

Trong quá trình biên dịch, nếu phát hiện một lệnh gán giá trị khác loại cho một biến, trình biên dịch sẽ tự động chuyển kiểu, gọi là cơ chế chuyển kiểu ngầm định.

Ví dụ:

1. double d = 13; // tự động chuyển giá trị của d thành 13.0
2. int n = 16.95; // tự động chuyển giá trị của n thành 16

Trong quá trình khai báo biến, hệ thống sẽ ngầm chuyển dữ liệu nhập thành kiểu biến đã khai báo trước.

Ví dụ:

1. string d ;
2. double n ;
3. cin>>d; // nếu bạn nhập kiểu số tự động chuyển giá trị thành chuỗi
4. cin>>n; // nếu bạn nhập 10 (số nguyên) tự động chuyển giá trị 10.0

Trình biên dịch sẽ hiện cảnh báo nếu xuất hiện sự chuyển kiểu không an toàn, gây mất mát giá trị, ví dụ như chuyển một dữ liệu có kích thước lớn, mang giá trị lớn về kiểu dữ liệu có kích thước nhỏ, hay dữ liệu số có dấu sang kiểu dữ liệu không dấu đều là chuyển kiểu không an toàn.

Khi tính toán một biểu thức, ví dụ như các phép toán số học, các hạng tử cần phải có cùng một kiểu thì việc tính toán mới diễn ra, nếu các hạng tử không cùng kiểu, sẽ xảy ra chuyển kiểu ngầm định, kiểu nhỏ hơn sẽ chuyển sang kiểu lớn hơn. Ta có dãy cấp độ ưu tiên như sau:

* long double // (highest)
* double
* float
* unsigned long int
* long int
* unsigned int
* int // (lowest)

Tại sao int lại là kiểu dữ liệu nhỏ nhất trong dãy, còn short và char ở đâu? Khi tính toán, kiểu dữ liệu short và char sẽ tự động được chuyển sang int.

Ta xét biểu thức tính toán đơn giản sau:

1. int n = 10 + 16.95;

Ta thấy biểu thức bên phải là phép toán cộng giữa kiểu int (10) và kiểu double (16.95), khi đó 10 sẽ được chuyển kiểu thành 10.0 và phép toán trở thành 10.0 + 16.95, kết quả là 26.95, được gán cho n là một biến int nên được chuyển kiểu và n = 26.

1. Chuyển kiểu tường minh

Là việc chuyển một dữ liệu sang kiểu dữ liệu khác theo ý muốn của lập trình viên. Cú pháp chuyển kiểu như sau:

**(<kiểu dữ liệu>)<dữ liệu cần chuyển>**

Ví dụ:

1. int n = 10;
2. float f = (float)n; // chuyển kiểu tường minh

Khi bắt đầu lập trình, tôi bị mắc một lỗi mà nhớ mãi, đó là nhầm lẫn giữa phép chia các số nguyên trong toán học và phép chia các số nguyên trong lập trình, dẫn đến kết quả lập trình không được như mong muốn.

Ví dụ:

1. float f = 10 / 4;

Tôi mong muốn kết quả f = 2.5 nhưng quy ước trong C là khi hai số nguyên chia nhau thì kết quả lấy phần nguyên do đó f = 2. Khắc phục như sau:

1. float f = ((float)10) / 4; // chuyển kiểu tường minh

Hoặc một cách khác:

1. float f = (1.0 \* 10) / 4; // chuyển kiểu ngầm định
2. Lời kết

Trên đây chỉ là những phép chuyển kiểu cơ bản trong C, trong ngôn ngữ C++, lập trình viên còn được hỗ trợ nhiều phép chuyển kiểu khác nhau như các phép casting, hay chuyển kiểu bằng constructor.

### Hằng kí tự :

Chuỗi hằng được lưu trữ trong bộ nhớ dữ liệu (Data segment)

Hằng (constant) là từ chỉ những thứ không thay đổi và lặp đi lặp lại. Trong lập trình cũng vậy, có những thứ mà lập trình viên sử dụng nhiều lần, nhưng lại không muốn chúng bị thay đổi giá trị, chẳng hạn như số PI, số phút trong một giờ hay vận tốc của ánh sáng.

Có hai cách để khai báo hằng là sử dụng từ khóa const hoặc sử dụng preprocessor definition.

1. Dùng từ khóa const

Cú pháp khai báo biến với từ khóa const như sau:

1. const <kiểu dữ liệu> <tên biến> = <giá trị>;

hoặc

1. <kiểu dữ liệu> const <tên biến> = <giá trị>;

Ta lấy ví dụ chương trình tính chu vi hình tròn, với PI là một hằng số:

1. #include <iostream>
2. using namespace std;
4. const double PI = 3.14159;
6. int main()
7. {
8. double R = 10; // Bán kính
9. double Circumference; // Chu vi hình tròn
11. Circumference = 2 \* PI \* R;
12. cout << Circumference;
14. return 0;
15. }

Khi sử dụng từ khóa const, bất kì hành vi nào làm thay đổi giá trị của biến const đều bị trình biên dịch báo lỗi.

1. Dùng chỉ thị tiền xử lí

Dùng từ khóa **#define**:

1. #define <tên hằng> <giá trị>

Ta viết lại chương trình trên theo cách này như sau:

1. #include <iostream>
2. using namespace std;
4. #define PI 3.14159 // PI không phải là 1 biến
6. int main()
7. {
8. double R = 10; // Bán kính
9. double Circumference; // Chu vi hình tròn
11. Circumference = 2 \* PI \* R;
12. cout << Circumference;
14. return 0;
15. }

**Lưu ý**

**Dòng #define PI 3.14159 không phải là một dòng code thông thường của C++, nên nó không có dấu chấm phẩy (;) ở cuối.**

**Về bản chất, PI không phải là một biến, nên nó không có vùng nhớ, do đó mọi thao tác liên quan như gán, tham chiếu, con trỏ tới nó đều không thực hiện được.**

1. Chuỗi số nguyên

Tất cả số nguyên được xuất ra màn hình ở dạng chuỗi kí tự số, gọi là hằng số nguyên. Chúng ta có thể biểu diễn số thập phân, số bát phân (thêm zero), số thập lục (thêm 0x hay 0X) cho biến hay số.

Hằng số nguyên có thể có hậu tố U hay u,L hay l tương ứng unsigned và long, chữ hoa hay chữ thường, đứng trước hay sau đều được.

|  |  |  |
| --- | --- | --- |
| **Kiểu** | **Hậu tố** | **Ý nghĩa** |
| int | u hay U, l hay L | unsigned int, long int |
| int | ul, uL, Ul, UL, lu, lU, Lu, or LU | unsigned long int |
| int | ll hay LL | long long int |
| int | ull, uLL, Ull, ULL, llu, llU, LLu, or LLU | unsigned long long int |
| double | f hay F | float |
| double | l hay L | long double |

Ví dụ:

212 /\* Legal \*/

215u /\* Legal \*/

0xFeeL /\* Legal \*/

078 /\* Illegal: 8 is not an octal digit \*/

032UU /\* Illegal: cannot repeat a suffix \*/

Ví dụ sau biểu diễn nhiều kiểu chuỗi số nguyên:

85 /\* decimal \*/

0213 /\* octal \*/

0x4b /\* hexadecimal \*/

30 /\* int \*/

30u /\* unsigned int \*/

30l /\* long \*/

30ul /\* unsigned long \*/

1. Chuỗi số có dấu chấm động

Hằng số có dấu chấm động gồm phần nguyên, dấu chấm thập phân, phần lẻ, và phần lũy thừa phía sau. Bạn có thể biểu diễn ở dạng thập phân hay dạng lũy thừa.

Ở dạng thập phân, phải có phần nguyên, dấu chấm thập phân và phần lẻ. Khi ở dạng lũy thừa, phải có phần nguyên, và phần lũy thừa. Dấu lũy thừa biểu diễn là e hay E

Ví dụ

3.14159 /\* Legal \*/

314159E-5L /\* Legal \*/

510E /\* Illegal: incomplete exponent \*/

210f /\* Illegal: no decimal or exponent \*/

.e55 /\* Illegal: missing integer or fraction \*/

1. Chuỗi kí tự đặc biệt

Một kí tự được bao đóng bởi dấu nháy đơn, ví dụ: ‘x’ có thể chứa trong một biến kiểu **char,** chúng được xem là kí tự hằng

Trong C++, nếu hằng bắt đầu bởi L (viết hoa), thì đây là kí tự mở rộng (ví dụ: L’x’) và phải dùng kiểu **wchar\_t**

Một hằng kí tự có thể là thuần kí tự ‘x’, một dãy thoát (‘\t’), hay một kí tự phổ quát (‘\u02C0’)

Một số kí tự trong C, C++ khi được đứng trước bởi dấu \ thì chúng sẽ mang một ý nghĩa đặc biệt như bắt đầu dòng mới '\n' hay tạo một tab '\t'. Chúng được biết như là escape sequence (dãy thoát). Bảng dưới đây thể hiện một số mã escape sequence phổ biến:

|  |  |
| --- | --- |
| **Escape sequence** | **Meaning** |
| \\ | Kí tự \ |
| \' | Kí tự ' |
| \" | Kí tự " |
| \? | Kí tự ? |
| \a | Phát tiếng Alert hay bell |
| \b | Backspace |
| \f | Form feed |
| \n | Ngắt dòng |
| \r | Carriage return |
| \t | Tab ngang |
| \v | Tab dọc |
| \ooo | Số hệ 8 lên tới 3 ký số |
| \xhh . . . | Số hệ 16 gồm nhiều kí số |

Ví dụ về dãy thoát

#include <stdio.h>

int main() {

printf("Hello\tWorld\n\n");

return 0;

}

Kết quả

Hello World

1. Chuỗi hằng

Chuỗi hằng nằm trong dấu nháy kép “”. Giống như hằng kí tự, chuỗi hằng có thể gồm thuần kí tự, dãy thoát, và kí tự phổ quát.

Bạn có thể ngắt một dòng rất dài thành nhiều đoạn dùng chuỗi hằng và tách chúng bằng kí tự trắng.

Ví dụ về chuỗi hằng

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

1. Chuỗi nhị phân C++ 14

Trong C++14, chúng ta có thể dùng định dạng chuỗi nhị phân bằng cách dùng tiền tố **0b.** Ví dụ:

#include <iostream>

int main(){

int bin(0);

bin = 0b1; // assign binary 0000 0001 to the variable

bin = 0b11; // assign binary 0000 0011 to the variable

bin = 0b1010; // assign binary 0000 1010 to the variable

bin = 0b11110000;// assign binary 1111 0000 to the variable

return 0;

}

Cho dễ đọc, bạn có thể đánh dấu phẩy đơn (‘) giữa các chuỗi bit.

#include <iostream>

int main(){

int bin = 0b1011'0010; // assign binary 1011 0010 to the variable

long value = 2'532'673'462; // much easier to read than 2532673462

return 0;

}

Nếu trình biên dịch không hỗ trợ C++14, IDE sẽ báo lỗi.

1. Lvalues và Rvalues:

Trong C,C++ định nghĩa 2 kiểu đối tượng:

* **lvalue :** Là những đối tượng được lưu trữ trong bộ nhớ. Xuất hiện ở bên trái hay phải của phép gán
* **rvalue :** Là những đối tượng tạm được lưu trữ tại nơi nào đó trong bộ nhớ. Nhưng không thể gán giá trị cho nó, nó chỉ xuất hiện bên phải phép gán.

Biến là kiểu **lvalue**.

Số , chữ…. Là kiểu **rvalue**.

int g = 20;

Câu lệnh sau là vô giá trị:

10 = 20;

**C: Toàn bộ biến phải khai báo ngay ₫ầu thân hàm chính**

**C++: Biến có thể khai báo tại chỗ nào cần, trước khi sử dụng**

### Phạm vi biến:

— **Biến toàn cục**: Khai báo ngoài hàm, lưu giữ trong vùng nhớ dữ liệu chương trình (data segment)

— **Biến cục bộ**: Khai báo trong thân hàm, lưu giữ trong ngăn xếp (stack segment)

— **Tham biến**: Khai báo trên danh sách tham số của hàm, lưu giữ trong ngăn xếp (stack segment)

Chúng ta sẽ xem xét biến cục bộ và toàn cục. Tham biến sẽ được giới thiệu trong chương sau cùng định nghĩa về hàm.

1. Biến cục bộ:

Được khai báo nội hàm và chỉ được truy xuất nội hàm đó, không được truy xuất bởi các hàm bên ngoài.

Ví dụ:

#include <iostream>

using namespace std;

int main (){

// Local variable declaration:

int a, b;

int c;

// actual initialization

a = 10;

b = 20;

c = a + b;

cout << c;

return 0;

}

1. Biến toàn cục:

Được khai báo ngoại hàm, nằm top của chương trình. Giá trị biến toàn cục được giữ suốt runtime của program.

Có thể truy xuất bởi bất kì hàm nào.

Ví dụ:

#include <iostream>

using namespace std;

// Global variable declaration:

int g;

int main (){

// Local variable declaration:

int a, b;

// actual initialization

a = 10;

b = 20;

g = a + b;

cout << g;

return 0;

}

Có thể có trùng tên biến cục bộ với biến toàn cục. Khi này giá trị biến cục bộ sẽ được ưu tiên trong nội hàm.

Ví dụ:

#include <iostream>

using namespace std;

// Global variable declaration:

int g = 20;

int main (){

// Local variable declaration:

int g = 10;

cout << g;

return 0;

}

Kết quả

10

1. Giá trị khởi tạo của biến cục bộ và toàn cục:

Hệ thống không khởi tạo giá trị cho biến cục bộ, lập trình viên phải tự khai báo. Biến toàn cục tự động khởi tạo nếu không được khai báo. Tham khảo bảng sau:

|  |  |
| --- | --- |
| **Data Type** | **Initializer** |
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

Tốt hơn hết là lập trình viên phải khởi tạo giá trị của biến để tránh program xuất ra kết quả không mong muốn.

### Đặc tính lưu trữ:

Tính lưu trữ định nghĩa pham vi và vòng đời của biến hay hàm trong C, C++ program. Các lớp lưu trữ sau được dùng trong C, C++

* auto
* register
* static
* extern
* mutable (C++ only)

1. Bộ nhớ đệm (cache)

Bộ xử lý máy tính rất nhanh và liên tục đọc thông tin từ bộ nhớ có nghĩa là nó phải chờ thông tin đến vì thời gian truy cập bộ nhớ chậm hơn tốc độ bộ xử lý. Một bộ nhớ đệm là bộ nhớ rất nhỏ, tạm thời và nhanh, cái mà bộ xử lý sử dụng để lấy thông tin mà nó cần trong một khơảng thời gian ngắn. Có rất nhiều ví dụ xung quanh chúng ta.

VD: Trong một ngôi nhà và những công cụ để trong gara và bạn cần sửa lại tầm hầm. Nó sẽ cần nhiều công cụ như: mũi khoan, bộ vặn đai ốc, cưa, ốc vít... Việc đầu tiên bạn nghĩ tới là đo và cắt các tấm gỗ, bạn chạy ra ngoài gara lấy cái thước đo từ kho lưu trữ, chạy xuống tầm hầm đo gỗ rồi quay trở lại nhà xe. Bỏ cái thước dây và lấy cái cưa sau đó trở lại tầng hầm với cái cưa và cắt gỗ. Bạn quyết định vặn ốc vít nối các mảnh gỗ lại. Bạn chạy lại gara để lấy khoan quay xuống tần hầm, khoan các lỗ để đặt ốc vít trở lại nhà để xe lấy cái vặn ốc, sau đó phát hiện ra là sai kích thước trở lại nhà để xe lấy cái khác và chạy lại. Bạn có làm theo cách này không? Nếu cần một cái vặn ốc có lẽ sẽ cần nhiều kích cỡ khác nhau tại sao không lấy toàn bộ những cái vặn ốc. Tiến thêm một bước tại sao không lấy một gói nhỏ bao gồm các công cụ cần thiết vào tầng hầm. Nó sẽ nhanh hơn rất nhiều. Bạn chỉ cần có bộ nhớ đệm. Một công cụ để dễ dàng truy cập và sử dụng nhanh chóng! Những công cụ mà bạn đang có ít có khả năng hơn để sử dụng vẫn còn được lưu giữ trong một địa điểm đó là đi xa hơn và đòi hỏi nhiều thời gian hơn để truy cập vào.

Đây là tất cả những gì mà bộ nhớ đệm phải làm, nó lưu trữ dữ liệu được truy cập và dữ liệu có thể được truy cập bởi CPU nhanh hơn và gần gũi hơn bộ nhớ. Bộ nhớ cache làm việc trên cùng một nguyên tắc cơ bản như là các ví dụ trước bằng cách sao chép dữ liệu thường xuyên được sử dụng vào bộ nhớ cache hơn là một đòi hỏi quyền truy cập vào bộ nhớ chính, để lấy các dữ liệu. Copy những dữ liệu cần sử dụng vào bộ đệm hơn là truy cập tới bộ nhớ chính để truy xuất dữ liệu. Bộ đệm có thể được tổ chức như cuốn sổ địa chỉ

Kích cỡ của bộ nhớ cache bộ nhớ có thể khác nhau về độ lớn. Một điển hình của máy tính cá nhân cấp hai (L2) là bộ nhớ cache 256K hoặc 512K. Cấp bậc một (L1) là bộ nhớ cache nhỏ hơn, điển hình là 8K hay 16K. Bộ nhớ cache L1 soáng chung về việc xử lý, trong khi L2 cache soáng chung giữa CPU và bộ nhớ chính. Bộ nhớ cache L1 được, do đó, nhanh hơn bộ nhớ cache L2. Các mối quan hệ giữa L1 và L2 cache có thể được minh họa bằng cách sử dụng của chúng tôi cửa hàng Grocery ví dụ: Nếu các cửa hàng chính là bộ nhớ, bạn có thể xem xét tủ lạnh các L2 cache, và trên thực tế các bàn ăn bộ nhớ cache L1.

Mục đích của vùng đệm là tăng tốc độ truy cập bộ nhớ bằng cách lưu trữ các dữ liệu cần sử dụng với CPU thay vì lưu trong bộ nhớ chính. Dù kích cỡ của bộ nhớ đệm không lớn như bộ nhớ chính nhưng nó nhanh hơn. Bộ nhớ chính được tạo bằng DRAM với thời gian truy cập là 60ns.Còn bộ đệm được tạo bằng SRAM truy cập nhanh hơn DRAM với chu kỳ ngắn hơn (một điển hình của thời gian truy cập bộ nhớ cache là 10ns).

Cache hiện không cần phải quá lớn để thực hiện tốt. Một quy tắc chung của bộ nhớ đệm nhở đủ để cho tổng chi phí trung bình cho mỗi bit, gần đó là của chính bộ nhớ, nhưng đủ lớn để mang lại lợi ích. Vì bộ nhớ nhanh này là hơi đắt tiền, nó không khả thi để sử dụng công nghệ tìm thấy trong bộ nhớ cache bộ nhớ để xây dựng các bộ nhớ chính.

Điều gì làm cho bộ nhớ cache "đặc biệt"? Cache không được truy cập theo địa chỉ mà được truy cập theo nội dung. Vì lý do này, bộ nhớ cache đôi khi được gọi là CAM. Theo hầu hết các chương trình lập bản đồ bộ nhớ cache, bộ nhớ cache mục phải được kiểm tra hoặc tìm kiếm nếu giá trị được yêu cầu được lưu trong bộ nhớ cache. Để đơn giản hóa việc này, quá trình tìm những dữ liệu, lập bản đồ bộ nhớ đệm sử dụng các thuật toán khác nhau.

Ánh xạ bộ nhớ đệm nhanh

Bộ nhớ đệm nhanh lưu trữ những dữ liệu có ích tuy nhiên nó sẽ trở thành vô ích nếu CPU không tìm thấy nó. Khi truy cập vào dữ liệu hoặc các tập lệnh, CPU lần đầu tiên tạo ra một địa chỉ bộ nhớ chính. Nếu dữ liệu đã được sao chép vào bộ nhớ cache, các địa chỉ của dữ liệu trong bộ nhớ cache không giống như là địa chỉ chính của bộ nhớ. Ví dụ: dữ liệu được cấp phát tại địa chỉ trong bộ nhớ chính là 2E3 có thể được cấp phát rất sớm trong bộ nhớ đệm. Làm thế nào CPU tìm ra dữ liệu khi nó đã được sao chép vào bộ nhớ đệm? CPU sử dụng một chương trình ánh xạ "chuyển đổi" các địa chỉ của bộ nhớ chính vào một vị trí bộ nhớ cache.

Việc chuyển đổi đia chỉ được thực hiện bằng cách vào trong các bit trong bộ nhớ địa chỉ chính. Chúng tôi chia các bit ra làm hai phần riêng biệt gọi là các trường. Tùy thuộc vào chương trình ánh xạ, chúng tôi có thể có hai hoặc ba trường. Làm thế nào chúng tôi sử dụng các trường này phụ thuộc vào chương trình ánh xạ đang được sử dụng. Các chương trình ánh xạ xác định nơi mà các dữ liệu được đặt ban đầu khi nó được sao chép vào bộ nhớ cache và cũng có thể cung cấp một phương pháp cho CPU để tìm kiếm trước đó đã sao chép dữ liệu khi tìm kiếm bộ nhớ đệm.

Làm thế nào để chúng ta sử dụng được các trường trong bộ nhớ địa chỉ chính. Một trường trong bộ nhớ địa chỉ chính chỉ ra cho ta thấy cái dữ liệu ở bên trong nếu nó ở trong bộ nhớ cache goi là cache hit nếu không gọi là cache miss. Các khối đệm tham chiếu được kiểm tra nếu nó hợp lệ. Kết thúc nó sẽ đưa ra các bít tương ứng với mỗi khối đệm. Giá trị Bit không nghĩa là khối đệm đó không hợp lệ, và phải truy cập bộ nhớ chính. Bit một có nghĩa là hợp lệ. Sau đó chúng ta so sánh thẻ trong khối đệm tới các thẻ của địa chỉ. Nếu các thẻ giống nhau ta có thể tìm ra khối bộ đệm mong muốn (chúng tôi có một bộ nhớ cache hit). Lúc này chúng ta cần xác định từ mong muốn của khối. Tất cả các chương trình ánh xạ đều yêu cầu trường từ **ba chương trình ánh xạ bộ nhớ chính:**

1. Cache ánh xạ trực tiếp

Ánh xạ trực tiếp sử dụng tiếp cận module. Bởi vì có nhiều khối bộ nhớ chính hơn bộ nhớ đệm. Cần rõ ràng rằng các khối bộ nhớ chính cấp phát cho bộ nhớ đệm. Ánh xạ trực tiếp khối X của bộ nhớ chính tới khối Y của bộ nhớ đệm, mod N với N là tổng số khối trong bộ nhớ đệm.

Cả hai khối '0', '10' đều trỏ tới khối '0' của khối '0' của bộ đệm làm thế nào để phân biệt. Cả hai khối đều được lưu trong bộ đệm và được nhận ra bằng các thẻ được khai báo trước.

Để thực hiện ánh xạ trực tiếp các bit đại chỉ bộ nhớ chính được chia làm ba trường. Độ dài của mỗi khối phụ thuộc vào đặc điểm vật lý của bộ nhớ chính và đệm. Trường word hay offset chứa các từ để nhận dạng các khối và chứa số bit tương ứng. Khối '0' chứa word trong bộ nhớ chính và được nhận ra = 00000000.Khi một khối của bộ nhớ chính được copy vào bộ nhớ cache, trường tag chứa cái khối và mã để nhận ra khối này. Tổng cộng có ba trường

1. Bộ nhớ đệm nhanh kết hợp hoàn toàn (Fully Associative Cache)

Bộ nhớ đệm nhanh ánh xạ trực tiếp không đắt như các bộ đệm khác vì kịch bản ánh xạ không đòi hỏi bất kỳ sự tìm kiếm. Khi địa chỉ bộ nhớ chính được chuyển sang bộ nhớ đệm, CPU biết chính xác vị trí để tìm trong bộ đệm bằng cách kiểm tra đơn giản số bit trong các trường. Gần giống cuốn sổ địa chỉ các trang thường có chỉ mục theo alphabetic.

Sử dụng ánh xạ tương đối địa chỉ bộ nhớ chính được chia ra làm hai phần: tag, word hình 6.8 trường word chứa ba bit nhưng trường tag là mười một bit. Khi bộ đệm được tìm kiếm, trường tag của bộ nhớ chính sẽ được so sánh với trường tag đúng của bộ đệm. Nếu không đúng ta sẽ có cache miss.

Tạo bộ nhớ tương ứng: Bộ đệm tương đối nhanh, phức tạp nhưng nó rất đắt. Mặc dù ánh xạ trức tiếp không đắt nhưng nó rất hạn chế giới hạn bộ đệm sử dụng hỗ trợ chạy 1 chương trình sử dụng khối '0', khối '16' sau đó thực thi các câu lệnh. Bộ nhớ đệm nhanh kết hợp hoàn toàn cho phép một khối từ bộ nhớ chính có thể ở bất cứ đâu. Tuy nhiên nó yêu cầu thẻ lớn được chứa khối thêm vào đó yêu cầu phần cứng để tìm tất cả các khối của tạm thời.

1. Tạo bộ nhớ đệm nhanh tương ứng

Bộ nhớ đệm nhanh tương đối nhanh, phức tạp nhưng nó rất đắt. Mặc dù ánh xạ trức tiếp không đắt nhưng nó rất hạn chế giới hạn bộ đệm sử dụng hỗ trợ chạy một chương trình sử dụng khối '0', khối '16' sau đó thực thi các câu lệnh. Bộ nhớ đệm nhanh kết hợp hoàn toàn cho phép một khối từ bộ nhớ chính có thể ở bất cứ đâu. Tuy nhiên nó yêu cầu thẻ lớn được chứa khối thêm vào đó yêu cầu phần cứng để tìm tất cả các khối của tạm thời

Ngoài ra còn chương trình ánh xạ thứ ba là N-way set associative cache map-ping kết hợp của hai phương pháp trên giống ánh xạ trực tiếp sử dụng địa chỉ để ánh xạ tới khối. Khác thay vì ánh xạ tới một khối đơn thì nó ánh xạ tới một tập các khối. Tất cả các khối phải có cùng kích cỡ

1. Lớp Auto

Là lớp lưu trữ mặc định cho tất cả biến cục bộ và hàm.

{

int mount;

auto int month;

}

Chỉ có thể khai báo với hàm và biến cục bộ. Lớp Auto được lưu trữ trong vùng stack.

1. Lớp register (Bộ nhớ đệm cache của CPU)

[Tham khảo kiến thức bộ nhớ đệm CPU](#_Bộ_nhớ_đệm)

Lớp **register** dùng để định nghĩa biến cục bộ, được lưu trữ trong thanh ghi bộ nhớ đệm thay vì RAM. Điều đó có nghĩa biến register có độ lớn tối đa là 1 word = 32 bit = 4 bytes và không thể dùng toán tử ‘&’ để truy xuất địa chỉ vì cách quản lý bộ nhớ đệm khác so với RAM.

{

register int miles;

}

**Register** chỉ dùng cho biến cần tốc độ xử lí nhanh ( như bộ đếm).

Tuy nhiên, tùy vào yêu cầu về phần cứng mà biến có thể được lưu trữ trong bộ đệm hay không.

1. Lớp Static

Lớp biến **static** được lưu trữ trong vùng nhớ data. Trình biên dịch compiler sẽ giữ biến **static** suốt vòng đời program.

Biến toàn cục **static** chỉ được phép truy xuất trong file mà thôi.

Trong C++ hướng đối tượng, thành viên dữ liệu **static** trong 1 class sẽ chỉ tạo ra 1 bản sao chia sẻ cho các đối tượng thuộc lớp này.

#include <iostream>

// Function declaration

void func(void);

static int count = 10; /\* Global variable \*/

Void main(){

while(count--) // stop at count = 0 , same as false {

func();

}

return 0;

}

// Function definition

void func( void ){

static int i = 5; // local static variable

i++;

std::cout << "i is " << i ;

std::cout << " and count is " << count << std::endl;

}

1. Lớp extern

Lớp **extern** chỉ dùng để tham chiếu tới biến toàn cục, giúp nó được truy xuất trên toàn file program. Biến **extern** sẽ không cần phải khởi tạo lại trong bộ nhớ vì nó được lưu trữ trong vùng data segment.

Bạn dùng từ khóa **extern** cho biến toàn cục hay hàm khi cần truy xuất trên file khác.

Ví dụ:

First File: main.cpp

#include <iostream>

int count ;

extern void write\_extern();

main(){

count = 5;

write\_extern();

}

Second File: support.cpp

#include <iostream>

extern int count;

void write\_extern(void){

std::cout << "Count is " << count << std::endl;

}

Biên dịch 2 file như sau:

$g++ main.cpp support.cpp -o write

Nó sẽ làm 1 chương trình thực thi “write” . Chay thử và kết quả:

$./write

5

1. Lớp Mutable (C++)

Từ khóa “mutable” chỉ được dùng cho dữ liệu thành viên của lớp, khiến nó có thể được thay đổi giá trị dù đối tượng đang có thuộc tính const.

Lưu ý không thể dùng ”mutable” chung với “static” hay “const” hay thành viên kiểu tham chiếu.

Ví dụ:

class A

{

public:

A() : x(4), y(5) { };

mutable int x;

int y;

};

int main()

{

const A var2;

var2.x = 345;

// var2.y = 2345;

}

Trình biên dịch không cho phép gán thành viên var2.y = 2345 vì var2 đang được khai báo là const, nhưng thành viên var2.x có thể thay đổi giá trị vì nó có thuộc tính mutable.

### Nhập xuất cơ bản trong C, C++

1. Trên C

Trong bất cứ một ngôn ngữ lập trình nào, việc tương tác trực tiếp với máy tính là rất quan trọng và cần thiết. Một chương trình được viết ra cần có khả năng nhận dữ liệu từ người dùng và hiển thị những kết quả thu được sau tính toán. Do đó việc thành thạo trong việc nhập/xuất dữ liệu là điều không thể thiếu đối với mỗi lập trình viên. Bài viết này sẽ giúp các bạn hiểu rõ hơn về thao tác nhập/xuất dữ liệu thông qua phương tiện nhập xuất chuẩn.

1. Tổng quan

Trong C hay bất cứ ngôn ngữ lập trình nào khác, việc nhập, xử lý và xuất dữ liệu có thể được thực hiện theo hai cách:

* Thông qua phương tiện nhập/xuất chuẩn (Standard Input/Output).
* Thông qua tập tin (File)

Bài viết này sẽ chỉ đề cập đến Standard Input/Output.

Thư viện nhập xuất chuẩn trong C là [**<stdio.h>,**](#_Stdlib.h) do đó các lập trình viên cần khai báo thư viện này trước khi thực hiện việc nhập xuất dữ liệu bằng chỉ thị **#include.**

<stdio.h> cung cấp cho chúng ta các hàm nhập và xuất dữ liệu có định dạng:

* printf(): Hàm xuất có định dạng.
* scanf(): Hàm nhập có định dạng.
* gets()
* puts()
* getchar()
* putchar()

1. The Standard Files

Ngôn ngữ C xem tất cả thiết bị I/O như là files, và sau đây là 3 kiểu FILE tự động mở khi chương trình được thực thi để cung cấp truy xuất các I/O device

|  |  |  |
| --- | --- | --- |
| **Standard File** | **Con trỏ FILE** | **Thiết bị** |
| Standard input | stdin | Keyboard |
| Standard output | stdout | Screen |
| Standard error | stderr | Your screen |

1. Hàm getchar() và putchar()

Hàm **int** **getchar(void)** dùng để đọc kí tự tiếp theo từ màn hình và return kí tự này về kiểu integer .

Hàm **int putchar(int c)** xuất kí tự lên màn hình và hàm return đúng kí tự đó.

Getchar() và putchar() này chỉ xuất nhập 1 kí tự cho 1 lần. Bạn dùng phương pháp lặp trường hợp cần xuất nhập nhiều hơn 1 kí tự lên màn hình.

Ưu điểm là khi dùng vòng lặp, bạn có thể nhập chuỗi nhiều dòng.

Ví dụ:

#include <stdio.h>

int main( )

{

int c;

printf( "Enter a value :");

c = getchar( );

printf( "\nYou entered: ");

putchar( c );

return 0;

}

Tôi nhập 1 chuỗi kí tự “this is test” và xem kết quả , 2 hàm này chỉ xuất nhập 1 kí tự mà thôi.

$./a.out

**Enter a value :** //this is test

**You entered:** t

1. Hàm gets() và puts()

Khi nhập xuất chuỗi, thay vì phải dùng vòng lặp như **putchar**() và **getchar**(). **Gets**() và **puts**() xuất nhập chuỗi tới khi gặp “enter” (1 dòng ).

Khuyết điểm là bạn không thể nhập nhiều dòng.

Hàm **char \*gets(char \*s)** để đọc 1 dòng từ **stdin** vào con trỏ bộ đệm đến khi ngắt dòng

Hàm **int puts(const char \*s)** viết một chuỗi **s** lên màn hình

#include <stdio.h>

int main( ){

char str[100];

printf( "Enter a value :");

gets( str );

printf( "\nYou entered: ");

puts( str );

return 0;

}

Chương trình đợi bạn nhập đoạn text đến khi “enter”, sau đó hiển thị lên màn hình.

$./a.out

**Enter a value :** this is test

**You entered:** This is test

Trong môi trường Visual Studio hàm **gets()** sẽ được thay thế bằng hàm **gets\_s().**

Hàm này có 2 kiểu overload là **gets\_s(char\* buffer, rsize\_t size)**

Hàm trả về chuỗi nạp vào bộ đệm nhập từ **stdin** tới khi gặp whitespace, hoặc đầy bộ đệm buffer.

Khi gặp whitespace, hàm sẽ thay thế kí tự whitespace bằng byte kết zero “\0” để báo kết thúc chuỗi.

Ví dụ:

// crt\_gets\_s.c

// This program retrieves a string from the stdin and

// prints the same string to the console.

#include <stdio.h>

int main(){

char line[21]; // room for 20 chars + '\0'

gets\_s( line, 20 );

printf( "The line entered was: %s\n", line );

}

Kết quả

**Hello there!**

The line entered was: Hello there!

1. Output với printf

Hàm printf trong C có nguyên mẫu của hàm (prototype) như sau:

1. int printf(“<chuỗi định dạng>”, <danh sách tham số>);

Trong đó, int là kiểu trả về của hàm, là giá trị đại diện cho hàm sau khi hết phạm vi của hàm. Chuỗi định dạng (Format string) có nhiệm vụ định dạng dữ liệu xuất ra màn hình. Danh sách tham số có thể bao gồm biến, hằng số, biểu thức và hàm (function) và được phân cách bằng dấu “,”.

1. Input với scanf

Prototype của hàm scanf trong C như sau:

1. int scanf(“<chuỗi định dạng>”, <danh sách tham số>);

Tương tự như hàm printf, danh sách tham số của scanf cũng được phân cách bằng dấu “,”. Tuy nhiên, tham số phải được truyền vào dưới dạng tham chiếu, tức truyền vào địa chỉ của biến.

Tham chiếu của các kiểu dữ liệu cơ bản (primitive data type) như int, float, char, … là & (address-of operator) cùng với tên biến.

Đối với các kiểu dữ liệu dẫn xuất thì tham chiếu đơn giản là tên biến.

Tương tự như hàm printf, chuỗi định dạng của scanf có cấu trúc như sau:

1. %[\*][width][length]specifier

Các bạn hãy hiện thực ví dụ sau để nắm được cách sử dụng hàm scanf cũng như các lệnh định dạng thông dụng:

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main ()
4. {
5. printf ("Welcome to Stdio!\n");
7. int a;
8. float b;
10. printf ("Enter an integer and a float: ");
11. scanf ("%d %f", &a, &b);
12. printf ("The integer you've entered is %d\nThe float you've entered is %f\n", a, b);

15. int dd, mm, yyyy;
16. printf ("Enter a date: "); //Format: dd/mm/yyyy
17. scanf ("%d/%d/%d", &dd, &mm, &yyyy);
18. printf ("The date you've entered is %2d/%2d/%4d\n", dd, mm, yyyy);
20. return 0;
21. }

**Lưu ý: Hàm scanf() hay scanf\_s() không thể dùng để input 1 chuỗi. Bạn phải dùng gets(), gets\_s() hay getchar() để input chuỗi.**

1. Một số vấn đề thường gặp

Qua kinh nghiệm học tập và nghiên cứu, tôi nhận thấy các lập trình viên MỚI BẮT ĐẦU thường gặp một số vấn đề trong việc nhập xuất dữ liệu. Do đó tôi tổng hợp lại vào bài viết này với mong muốn các bạn sẽ hạn chế được những sai sót này.

1. Không truyền tham chiếu vào hàm scanf

Đây là lỗi mà các bạn thường gặp nhất. Với những project nhỏ thì lỗi này có thể phát hiện được dễ dàng. Tuy nhiên, khi làm việc với những dự án lớn, việc phát hiện và khắc phục lỗi này là cực kỳ khó khăn, vì đây không phải là lỗi trong quá trình Build nên ta không xác định được vị trí dòng code bị lỗi. Do đó lỗi này là đặc biệt nghiêm trọng và cần được khắc phục trong quá trình học tập.

Cách khắc phục: thêm toán tử & vào trước tên biến (primitive data type).

1. Chưa khai báo hay khởi tạo giá trị cho biến

Khi làm việc với những người bạn, tôi thấy khá nhiều bạn sử dụng biến mà không khai báo hoặc quên khởi tạo giá trị ban đầu cho biến. Đây không phải là lỗi nghiêm trọng và rất dễ khắc phục, tuy nhiên nó cho thấy sự thiếu cẩn thận trong công việc của bạn. Do đó nếu các bạn còn mắc phải lỗi này, hãy cố gắng khắc phục sớm.

1. Trôi dòng lệnh khi xử lý chuỗi

Lỗi này cũng khá phổ biến và ít nhiều gây hoang mang cho các bạn. Thao tác với chuỗi ký tự, các bạn có thể tham khảo bài viết [Thao tác với chuỗi trong C.](#_Thao_tác_với)

Trong C, dòng (stream) vào tiêu chuẩn là stdin. Các hàm như scanf, gets, getchar, … đều nhận dữ liệu từ stdin. Khi trên stdin không còn dữ liệu, các hàm nhập dữ liệu sẽ yêu cầu người dùng nhập vào từ bàn phím.

Các hàm trên chỉ nhận đủ dữ liệu mà chúng yêu cầu (trong trường hợp này, scanf chỉ nhận chuỗi không có khoảng trắng), do đó một phần dữ liệu còn sót lại trên stdin, có thể là ký tự ‘\n’ hoặc phần dữ liệu sau khoảng trắng. Điều này ảnh hưởng đến các hàm nhập dữ liệu phía sau.

Ta có thể khắc phục bằng cách làm rỗng bộ đệm stdin trước mỗi hàm nhập để đảm bảo độ chính xác của chương trình. Câu lệnh như sau:

1. **fflush**(stdin);

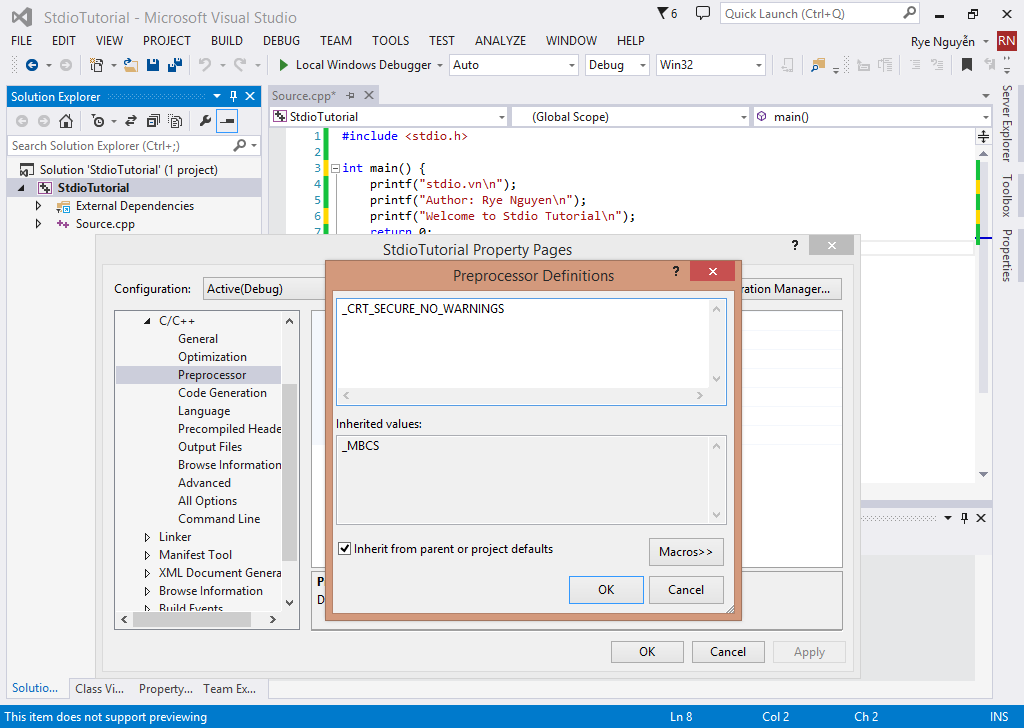
Hàm fflush() nằm trong thư viện <stdio.h> nên các bạn có thể dễ dàng sử dụng mà không phải include thêm thư viện khác.

1. Error C4996 trong Visual Studio 2012 về sau

Từ Visual Studio 2012 về sau, một số hàm đã được Microsoft loại bỏ hoặc thay thế bởi những hàm tương tự. Lý do là những hàm này không còn an toàn nữa.

Có khá nhiều cách khắc phục lỗi này, tôi xin hướng dẫn các bạn một vài cách như sau:

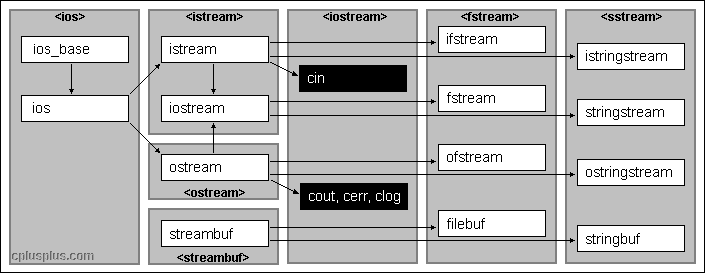
* Sử dụng các hàm được khuyến nghị bởi Microsoft. Các hàm này được hiển thị trong thông báo lỗi (scanf\_s thay cho scanf, printf\_s thay cho printf, …)
* Khai báo thêm một cờ (flag) nhằm bỏ qua các cảnh báo này ở đầu chương trình.   
  Khai báo như sau: #define \_CRT\_SECURE\_NO\_WARNINGS
* Một cách khác nữa là các bạn tắt cảnh báo trực tiếp trong project. Các bạn vào Project Properties > Configuration Properties > C/C++ > Preprocessor. Tại dòng Preprocessor Definitions, chọn Edit và thêm dòng \_CRT\_SECURE\_NO\_WARNINGS. Nhấn OK để thay đổi có hiệu lực.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_2/14/ss_1.png)

1. Trên C++
2. Thư viện chuẩn

Đối với mọi ngôn ngữ lập trình, thao tác nhập/xuất dữ liệu luôn là thao tác cơ bản và quan trọng mà mỗi lập trình viên cần nắm rõ. Bài viết này sẽ cung cấp kiến thức cơ bản về các hàm nhập xuất chuẩn (Standard input/output) trong C++.

Thư viện xuất nhập có cấu trúc như sau



Stream là một lớp trừu tượng đại diện cho thiết bị xuất nhập. Nói chung stream tương ứng với nguồn hay đích vật lý của dữ liệu, như file, bàn phím hay màn hình, nên dữ liệu được lấy hay viết tới/từ lớp trừu tượng này gọi là stream.

Ví dụ File stream trong C,C++ là đối tượng để xử lí file.

1. Tổ chức

Thư viện và các lớp thừa kế được chia thành nhiều files:

**<ios>, <istream>, <ostream>, <streambuf>** và **<iosfwd>:** Chúng là các lớp cơ sở và tự động include bởi các header file của thư viện lớp dẫn xuất

Thư viện <iostream> trong C++ cung cấp đầy đủ các tính năng cần thiết cho các thao tác nhập/xuất dữ liệu. Do đó để thực hiện các thao tác này, chúng ta phải include thư viện này bằng chỉ thị sau:

#include <iostream>

Ngoài ra, using namespace std; cũng cần thiết giúp chúng ta thực hiện các thao tác trên C++. Tìm hiểu thêm về [Từ Khóa namespace.](#_Từ_khóa_Name)

C++ cung cấp cho chúng ta các hàm nhập xuất chuẩn:

* cin – Hàm nhập dữ liệu từ bàn phím (Standard input device).
* cout – Hàm xuất dữ liệu ra màn hình (Standard output device).

Đây là các thư viện I/O quan trong trong C++:

|  |  |
| --- | --- |
| **Header File** | **Function and Description** |
| **<iostream>** | Định nghĩa đối tượng **cin, cout, cerr,** tương ứng với standard input stream, standard output stream, standard error stream không dùng bộ đệm và **clog** , standard log stream có bộ đệm. |
| **<iomanip>** | Khai báo các dịch vụ hữu ích cho việc thực thi các I/O đã chuẩn hóa với bộ vận hành stream được tham số hóa, như hàm **setw** và **setprecision** |
| **<fstream>** | Khai báo các dịch vụ xử lí file cho người dùng. Ta sẽ thảo luận sau trong phần File Stream |
| <**sstream**>: | định nghĩa các lớp xử lí stream là đối tượng chuỗi |

Các yếu tố thư viện chuẩn **<iostream>** :

* Các phần thừa kế từ lớp cơ sở
* Hai bản thể lớp :   
  Một để xử lí dữ liệu kiểu **char: <ios>, <istream>, <ostream>**Một để xử lí kiểu **wchar\_t: <wios>, <wistream>, <wostream>**
* Các đối tượng stream chuẩn
* Các kiểu dữ liệu
* Các hàm global, được thiết kế để dùng với toán tử chèn stream **<<** và khai stream **>>** nhằm xử lí các đối tượng stream iostream. Chúng tùy chỉnh thuộc tính và định dạng của streams.   
  Ví dụ: phương thức endl sẽ ngắt dòng và **flush** streams.

1. Cin

**Cin** là một đối tượng thuộc **istream** class. Đối tượng này được gán với bàn phím. **Cin** được dùng kết hợp với toán tử xuất stream “>>”. Ví dụ:

1. int variable;
2. cin >> variable;

Khi đó chương trình sẽ dừng lại và chờ đợi người sử dụng nhập dữ liệu vào từ bàn phím.

Nếu cần nhập liên tục dữ liệu cho nhiều biến, ta có thể sử dụng toán tử **>>** liên tục trên cùng một hàm cin:

1. cin >> variable\_1 >> variable\_2 >> … >> variable\_n;

Khi đó, các biến sẽ được phân cách bằng một ký tự trắng (**white space characters**) khi bạn nhập giá trị vào. White space character bao gồm: ký tự khoảng trắng (ASCII code = 32), ký tự tab **(‘\t’)** và ký tự xuống dòng **(‘\n’)**.

Toán tử “>>” đã được overload sẵn cho một số kiểu dữ liệu cơ bản của C++ như int, float, char, … Do đó chúng ta có thể nhập dữ liệu cho các biến có kiểu dữ liệu khác nhau trên cùng một hàm cin.

**Lưu ý: Dữ liệu nhập vào từ bàn phím sẽ được tự động casting (ép kiểu) sang kiểu dữ liệu của biến.**

**Việc chuyển kiểu này là ngầm định, do đó lập trình viên cần lưu ý khi khai báo biến để tránh tình trạng mất dữ liệu.**

1. Cout

Cout là đối tượng của lớp **ostream.** Đối tượng cout kết nối tới màn hình. Cout được dùng để kết hợp với toán tử chèn stream “<<”,

Để xuất một biểu thức ra màn hình (Standard output device), ta thực hiện như sau:

1. cout << expression;

Ta cũng có thể xuất một dãy các biểu thức bằng cách sử dụng nhiều toán tử “<<” trên cùng một hàm cout:

1.cout << expression\_1 << expression\_2 << … << expression\_n;

Cũng như toán tử “>>” được sử dụng trong cin, toán tử “”<<” cũng được overload sẵn cho các kiểu dữ liệu cơ bản. C++ sẽ tự động nhận dạng kiểu dữ liệu và xuất dữ liệu ra màn hình một cách phù hợp.

1. Chương trình minh hoạ

Chương trình dưới đây sẽ giúp các bạn hiểu rõ hơn về cin, cout và cách hoạt động của chúng.

1. #include <iostream>
2. #include <string>
3. using namespace std;
5. int main()
6. {
7. string studentName;
8. float midtermScore, practiceScore, finalScore;
10. cout << "Enter student name: ";
11. getline(cin, studentName);
13. cout << "Enter student scores: ";
14. cin >> midtermScore >> practiceScore >> finalScore;
16. cout << "Average score: "
17. << midtermScore \* 0.2 + practiceScore \* 0.3 + finalScore \* 0.5
18. << endl;
20. return 0;
21. }

Tôi sử dụng hàm **getline** để nhập chuỗi ký tự có khoảng trắng.

Thư viện chuẩn của C++ cung cấp khả năng bao quát để quản lý tiến trình nhập xuất. Phần này sẽ thảo luận về tiến trình I/O một cách căn bản

Tiến trình I/O trong C++ thể hiện thành streams, là chuỗi bytes. Luồng byte này từ các device như bàn phím, ổ đĩa, mạng, thiết bị lưu trữ…., được gọi là tiến trình input. Và nếu luồng byte này từ bộ nhớ chính đi tới các thiết bị như màn hình, máy in, ổ đĩa hay mạng… được gọi là tiến trình output.

1. Cerr

Cerr là một đối tượng thuộc ostream calss. Đối tượng này gắn liền với màn hình, để hiển thị lỗi. Nhưng đối tượng cerr không có bộ đệm và mỗi stream bit đi vào cerr sẽ xuất ra output ngay tức thì.

Cerr được dùng với toán tử chèn stream “<<”. Ví dụ:

#include <iostream>

using namespace std;

int main( )

{

char str[] = "Unable to read....";

cerr << "Error message : " << str << endl;

return 0;

}

Kết quả là:

Error message : Unable to read....

1. Clog

clog là một đối tượng thuộc ostream calss. Đối tượng này gắn liền với màn hình để hiển thị lỗi. Nhưng đối tượng clog có bộ đệm nên mỗi stream bit đi vào clog sẽ được giữ lại ở bộ đệm tới khi fully buffer hoặc bị flush.

clog được dùng với toán tử chèn stream “<<”. Ví dụ:

#include <iostream>

using namespace std;

int main( ){

char str[] = "Unable to read....";

clog << "Error message : " << str << endl;

}

Kết quả là:

Error message : Unable to read....

Bạn sẽ không thể thấy khác biệt giữa cout, cerr và clog với ví dụ nhỏ, nhưng trong chương trình lớn, sẽ rất tốt khi hiển thị lỗi với cerr stream và trong khi hiển thị tin nhắn log với clog

### Xử lí chuỗi:

Một trong những thứ vĩ đại mà con người chúng ta có đó là ngôn ngữ, nó giúp phân biệt chúng ta với các loài khác, nhờ ngôn ngữ mà chúng ta trao đổi thông tin dễ dàng hơn. Ngôn ngữ được diễn đạt bằng nhiều hình thức, trong đó có chữ viết. Trong lập trình cũng thế, nhu cầu sử dụng các dãy chữ có ý nghĩa rất lớn. Các dãy chữ đó được gọi là chuỗi.

Chuỗi được diễn tả bằng một dãy các kí tự đặt trong cặp ngoặc kép, ví dụ như “Hello World!”, “Stdio”, “Nguyen Minh Hieu”….

*Nhưng về bản chất, chuỗi là một mảng các kí tự liên tiếp nhau, và kết thúc ở kí tự ‘\0’.*

**Ví dụ:**

char s[] = "0123456789";

s[5] = '\0';

prinft("%s", s); /\* se ra 01234 \*/

**\*Note: Byte kết zero (\0) dùng để dánh dấu kết thúc chuỗi trong C**

* **Trong C/C++, chuỗi ký tự không phải là kiểu cơ bản, mà thực chất là một mảng kí tự**
* **Phân biệt chuỗi ký tự thường và chuỗi ký tự kết zero (/0)**
* **Đa số các hàm trong thư viện C làm việc với chuỗi ký tự kết zero**
* **Với C++, chuỗi ký tự ₫ược ₫ịnh nghĩa bằng lớp string trong thư viện chuẩn.**
* **C++ không sử dụng byte kết zero**

1. Chuỗi char
2. Chuỗi định dạng

Được sử dụng trong rất nhiều hàm xuất, nhập chuỗi như printf, sprintf, scanf, vsprintf…

**Cú pháp:**

**%[flags][width][.precision][length]specifier**

|  |  |
| --- | --- |
| **specifier** | **Mô tả** |
| %c | Ký tự char |
| %d hoặc %i | Kiểu int hay short int |
| %e | Ký hiệu khoa học (mantissa/exponent) sử dụng ký tự e |
| %E | Ký hiệu khoa học (mantissa/exponent) sử dụng ký tự E |
| %f | Số thực dấu chấm động hệ thập phân float |
| %g | Sử dụng rút gọn của %e hoặc %f |
| %G | Sử dụng rút gọn của %E hoặc %f |
| %o | Số bát phân có dấu octal |
| %s | Chuỗi ký tự string |
| %u | Kiểu unsigned int |
| %ul | Kiểu unsigned long int |
| %lf | Kiểu double |
| %ld | Kiểu long int |
| %x | Số nguyên hệ thập lục phân không dấu |
| %X | Số nguyên hệ thập lục phân không dấu (các chữ cái hoa) |
| %p | Địa chỉ con trỏ |
| %n | Không in cái gì |

|  |  |
| --- | --- |
| **flags** | **Mô tả** |
| - | Căn chỉnh trái vào bên trong độ rộng trường đã cho. Căn chỉnh phải là mặc định |
| + | Ép buộc đặt trước kết quả một dấu cộng hoặc trừ (+ hoặc -) ngay cả với các số dương. Theo mặc định, chỉ có các số âm được đặt trước bởi một dấu - |
| (space) | Nếu không có ký hiệu nào được ghi, thì một khoảng trống sẽ được chèn trước giá trị |
| # | Được sử dụng với các specifier là o, x hoặc X. Giá trị được đặt trước với 0, 0x hoặc 0X tương ứng cho các giá trị khác 0. Sử dụng với e, E và f, nó ép buộc output đã được ghi để thu nhận một con trỏ thập phân ngay cả nếu không có chữ số nào theo sau. Theo mặc định, nếu không có chữ số nào theo sau thì không có con trỏ thập phân được ghi. Sử dụng với g hoặc G, kết quả là giống với e hoặc E nhưng các số 0 ở cuối không bị gỡ bỏ |
| 0 | Đệm vào bên trái (left-pad) của số với các số 0 thay vì bởi các khoảng trống |

|  |  |
| --- | --- |
| **width** | **Mô tả** |
| (number) | Số ký tự tối thiểu để được in. Nếu giá trị để được in là ngắn hơn số này, thì kết quả được đệm thêm với các khoảng trống. Giá trị không bị cắt ngay cả khi kết quả là quá lớn |
| \* | Độ rộng không được xác định trong chuỗi định dạng format, nhưng như một tham số giá trị nguyên bổ sung đặt trước tham số đó mà phải được định dạng |

|  |  |
| --- | --- |
| **.precision** | **Mô tả** |
| .number | Với các Integer Specifier (d, i, o, u, x, X) − thì Precision xác định số chữ số nhỏ nhất được ghi. Nếu giá trị được ghi là ngắn hơn số này thì kết quả được đệm thêm với các 0 vào đầu. Giá trị không bị cắt ngay cả nếu kết quả là dài hơn. Một precision là 0 nghĩa là không có ký tự nào được ghi cho giá trị 0. Với e, E và f specifier: đây là số chữ số để được in sau điểm thập phân. Với g và G specifier: đây là số chữ số có nghĩa tối đa để được in. Với s specifier: đây là số ký tự tối thiểu để được in. Theo mặc định, tất cả ký tự được in tới khi bắt gặp ký tự null cuối cùng. Với kiểu c: nó không có ảnh hưởng. Khi không có precision nào được xác định, thì mặc định là 1. Nếu period được xác định mà không kèm với một giá trị precision rõ ràng, thì 0 được giả sử |
| .\* | Precision không được xác định trong chuỗi định dạng format, nhưng như một tham số giá trị nguyên bổ sung đặt trước tham số đó mà phải được định dạng |

|  |  |
| --- | --- |
| **length** | **Mô tả** |
| h | Tham số được thông dịch như một short int hoặc unsigned short int (chỉ được áp dụng cho integer specifiers: i, d, o, u, x và X) |
| l | Tham số được thông dịch như một long int hoặc unsigned long int cho integer specifier (i, d, o, u, x và X), và như một wide char hoặc wide char string cho các specifier là c và s |
| L | Tham số được thông dịch như một long double (chỉ được áp dụng cho các floating point specifier: e, E, f, g và G) |

1. Khai báo

Để khai báo chuỗi, chúng ta không cần include thêm bất cứ thư viện nào. Tuy nhiên, để thao tác với chuỗi, ta không thể xử lý từng phần tử trong mảng ký tự đã khai báo, vì sẽ rất mất thời gian và nhiều khả năng gây ra sai sót. Do đó, ngôn ngữ C cung cấp cho chúng ta các function cần thiết để thao tác với chuỗi. Hầu hết các hàm này nằm trong thư viện <string..h>. Xem thêm [Nhập xuất cơ bản trong C.](#_Nhập_Xuất_Cơ) để biết về nhập xuất chuỗi.

Chúng ta có thể khai báo chuỗi bằng nhiều cách:

// Cách thông thường

char str1[20] = "Stdio";

char str2[20] = { 'S', 't', 'd', 'i', 'o', '\0' };// Cách khai báo từng phần tử, ít được sử dụng

string str3 = "Stdio";// Sử dụng thư viện std::string của C++

1. Hàm snprintf

Cú pháp:

int snprintf(char \*str, size\_t size, const char \*format, ...)

Tham số:

* Str: Con trỏ bộ đệm chuỗi nhận các kí tự định dạng phía sau.
* Size: kích thước bộ đệm str
* format – Chuỗi định dạng gồm: kí tự trắng, kí tự thường và kí tự định dạng

Chuỗi định dạng có cú pháp : [=%[\*][width][modifiers]type=]

Ví dụ Chèn chuỗi định dạng vào con trỏ chuỗi

#include <stdio.h>

#include <math.h>

int main(){

char str[80];

snprintf(str, sizeof(str),"Value of Pi = %f", M\_PI);

puts(str);

return(0);

}

Kết quả:

Value of Pi = 3.141593

1. Hàm sscanf

Cú pháp

int sscanf(const char \*str, const char \*format, ...)

Tham số

* Str: Bộ đệm chuỗi.
* format : Chuỗi định dạng
* … : các tham số nhận giá trị từ bộ đệm theo định dạng format

Chuỗi định dạng có cú pháp : [=%[\*][width][modifiers]type=]

Ví dụ:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main(){

int day, year;

char weekday[20], month[20], dtm[100];

strcpy( dtm, "Saturday March 25 1989" );

sscanf( dtm, "%s %s %d %d", weekday, month, &day, &year );

printf("%s %d, %d = %s\n", month, day, year, weekday );

return(0);

}

Kết quả

March 25, 1989 = Saturday

1. Hàm vsprintf

Cú pháp:

int vsprintf(char \*str, const char \*format, va\_list arg)

Dùng để áp chuỗi kí tự định dạng **format** của danh sách tham số **arg** vào con trỏ chuỗi **str**

Ví dụ:

#include <stdio.h>

#include <stdarg.h>

char buffer[80];

int vspfunc(char \*format, ...){

va\_list aptr;

int ret;

va\_start(aptr, format);

ret = vsprintf(buffer, format, aptr);

va\_end(aptr);

return(ret);

}

int main(){

int i = 5;

float f = 27.0;

char str[50] = "tutoriaspoint.com";

vspfunc("%d %f %s", i, f, str);

printf("%s\n", buffer);

return(0);

}

Kết quả:

5 27.000000 tutoriaspoint.com

1. Thư viện string.h
2. Một vài hàm cơ bản

Một số hàm thao tác với chuỗi thường dùng được liệt kê trong bảng sau:

|  |  |
| --- | --- |
| **TÊN HÀM** | **CHỨC NĂNG** |
| **Strlen(s1)** | Trả về độ dài chuỗi s1 , không bao gồm kí tự ‘\0’ |
| **Strcpy(s1, s2)** | Copy chuỗi s2 sang chuỗi s1  Lưu ý, chuỗi s1 phải được cấp phát trước.  Hàm sao chép tới khi gặp byte kết zero (s1 hoặc s2) |
| **Strcmp(s1, s2)** | So sánh nội dung 2 chuỗi.  Trả về 0 nếu 2 chuỗi giống nhau.  > 0 nếu 2 chuỗi khác nhau và s1 dài hơn s2.  < 0 nếu 2 chuỗi khác nhau và s1 ngắn hơn s2 |
| **Strncmp(s1, s2 , n)** | So sánh nội dung n kí tự đầu tiên hai chuỗi  Trả về 0 nếu 2 chuỗi giống nhau.  > 0 nếu 2 chuỗi khác nhau và s1 dài hơn s2.  < 0 nếu 2 chuỗi khác nhau và s1 ngắn hơn s2 |

Ngoài ra còn nhiều hàm khác không được liệt kê trong bảng này. Tuỳ theo nhu cầu sử dụng, các bạn có thể tìm hiểu và vận dụng vào việc học tập của mình. [Xem thêm tại đây](#_String.h)

Các bạn xem xét đoạn code sau để hiểu được cách thao tác với chuỗi:

1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
5. int main()
6. {
7. char a[10];
8. **gets**(a);
9. printf("%d", **strlen**(a));
11. char \*b = (char\*)**malloc**(**strlen**(a)\*sizeof(char)) ;
12. **strcpy**(b, a);
13. **puts**(b);
14. **free**(b); //giải phóng con trỏ chuổi b trong HEAP
16. return 0;
17. }
18. Tìm kí tự: strchr và strrchr

Cú pháp:

char \*strchr(const char \*str, int c)

Tham số

* str – Chuỗi tìm.
* c – Kí tự cần tìm.

Hàm **strchr** tìm và trả về con trỏ tới vị trí đầu tiên của kí tự cần tìm trong chuỗi hoặc NULL nếu không tìm thấy.

Ngược lại ta có hàm **strtchr** trả về con trỏ tới vị trí cuối cùng kí tự cần tìm

char \*strrchr(const char \*str, int c)

Ví dụ

#include <stdio.h>

#include <string.h>

int main (){

const char str[] = "http://www.tutorialspoint.com";

const char ch = '.';

char \*ret;

ret = strchr(str, ch);

printf("String after |%c| is - |%s|\n", ch, ret);

return(0);

}

Kết quả

String after |.| is - |.tutorialspoint.com|

1. Cắt chuỗi String token

Cú pháp

char \*strtok(char \*str, const char \*delim)

Tham số

* **str** – Chuỗi kí tự cần cắt (tokens).
* **delim** – Kí tự phân cách chuỗi .

Sau khi cắt, các chuỗi con sẽ được phân bổ trên các vùng nhớ khác nhau. Trong lần gọi đầu tiên, hàm trả về con trỏ tới chuỗi con đầu tiên.

Để lấy tiếp con trỏ tới các chuỗi còn lại, chúng ta gọi tuần tự: strtok(NULL, delim)

Hàm trả về NULL khi không còn chuỗi con nào nữa.

Ví dụ

#include <string.h>

#include <stdio.h>

int main(){

char str[80] = "This is - www.tutorialspoint.com - website";

const char s[2] = "-";

char \*token;

/\* lấy token đầu tiên \*/

token = strtok(str, s);

/\* Duyệt tokens \*/

while( token != NULL ) {

printf( " %s\n", token );

token = strtok(NULL, s);

}

return(0);

}

Kết quả

This is

www.tutorialspoint.com

website

1. Nối chuỗi strcat() và strncat()

Hàm **strcat** nối chuỗi **src** vào cuối chuỗi **des.** Hàm trả về con trỏ tới chuỗi **dest,** phải đảm bảo **des** đủ vùng nhớ để chứa **src**.

char \*strcat(char \*dest, const char \*src)

Khi gặp lỗi hàm trả về giá trị kiểu **errno\_t** tương ứng với lỗi và return 0 nếu thành công.

Trường hợp chuỗi **src** nhập từ chuẩn **stdin** , C sẽ tự set “\0” cuối chuỗi.

Có nhược điểm là:

* Nếu trường hợp dữ liệu chuỗi có được do bạn tự gán giá trị và bạn không set byte kết zero “/0” cuối chuỗi, hàm **strcat** sẽ tiếp tục copy dữ liệu vùng địa chỉ sau **src** tới khi đầy buffer hoặc tới vùng địa chỉ **NULL('\0’)** thì dừng.

Để an toàn, nên dùng **strncat** có qui định số kí tự tối đa **n** được sao chép

char \*strncat(char \*dest, const char \*src, size\_t n)

Ví dụ:

#include <stdio.h>

#include <string.h>

int main (){

char src[50], dest[50];

strcpy(src, "This is source");

strcpy(dest, "This is destination");

strncat(dest, src, 15);

printf("Final destination string : |%s|", dest);

return(0);

}

Final destination string : |This is destinationThis is source|

1. Tìm chuỗi con strstr

Hàm **strstr** tìm và trả về con trỏ tới vị trí tồn tại chuỗi con **needle** trong chuỗi **haystack**

char \*strstr(const char \*haystack, const char \*needle)

Nếu không tìm thấy, hàm trả về **NULL**.

#include <stdio.h>

#include <string.h>

int main(){

const char haystack[20] = "TutorialsPoint";

const char needle[10] = "Point";

char \*ret;

ret = strstr(haystack, needle);

printf("The substring is: %s\n", ret);

return(0);

}

The substring is: Point

1. Thư viện ctype.h

Cung cấp các hàm để kiểm tra và ánh xạ kí tự. Tất cả các hàm đều nhận tham số kí tự nhập vào là kiểu integer (EOF hay unsigned char).

Giá trị trả về khác zero (true) nếu kí tự thỏa mãn yêu cầu của hàm, và zero (false) ngược lại.

1. Kiểm tra white-space: isspace

Định nghĩa white-space gồm:

' ' (0x20) space (SPC)

'\t' (0x09) horizontal tab (TAB)

'\n' (0x0a) newline (LF)

'\v' (0x0b) vertical tab (VT)

'\f' (0x0c) feed (FF)

'\r' (0x0d) carriage return (CR)

Cú pháp:

int isspace(int c);

1. Kiểm tra số isdigit

int isdigit(int c);

1. Kiểm tra alphabet isalpha

Hàm kiểm tra kí tự có phải alphabet

int isalpha(int c);

1. Chuyển kí tự hoa – thường: tolower & toupper

Chuyển sang kí tự thường:

int tolower(int c);

Chuyển sang kí tự hoa

int toupper(int c);

Nếu kí tự không nằm trong khoảng a-z, A-Z thì giá trị trả về vẫn giữ nguyên.

Ví dụ:

#include <stdio.h>

#include <ctype.h>

int main(){

int i = 0;

char c;

char str[] = "Tutorials Point";

while(str[i]) {

putchar (toupper(str[i++]));

}

return(0);

}

TUTORIALS POINT

1. Những vấn đề thường gặp
2. Nhập chuỗi có khoảng trắng

Hàm **scanf** trong C nhận diện khoảng trắng giống như ký tự xuống dòng ‘\n’, như một dấu hiệu để kết thúc chuỗi thu được từ bộ đệm **stdin** (khác với ký tự kết thúc chuỗi ‘\0’). Do đó, khi nhập dữ liệu có khoảng trắng, chẳng hạn như nhập họ tên, phần dữ liệu còn lại sau khoảng trắng sẽ bị giữ lại ở **stdin** và hàm **scanf** sẽ không nhận đủ lượng dữ liệu cần thiết.

Để nhập chuỗi có khoảng trắng, ta sử dụng hàm **gets**. Cú pháp của hàm như sau:

1. char\* **gets**(char\* str);

Hàm **gets** sẽ nhận khoảng trắng như một ký tự bình thường. Quá trình nhập sẽ kết thúc khi bạn nhấn phím Enter.

1. Trôi dòng lệnh

Tôi đã từng đề cập đến vấn đề này trong bài viết [Nhập xuất cơ bản trong C](#_Nhập_Xuất_Cơ). Khi trên **stdin** còn dữ liệu phù hợp với yêu cầu (thông thường là ký tự '\n' của lần nhập dữ liệu trước), nó sẽ bỏ qua các hàm nhập dữ liệu và truyền dữ liệu còn lại đó cho các hàm.

Để không bị trôi các hàm nhập, cần xoá sạch bộ đệm **stdin** trước mỗi hàm nhập bằng dòng code **fflush**(stdin).

1. Quên huỷ vùng nhớ

"Có new là phải có delete". Tôi nhớ mãi câu nói này của người thầy của tôi. Việc không huỷ vùng nhớ sẽ không để lại "dấu vết" gì cho các bạn thấy được để khắc phục, nhưng nó sẽ gây ra lỗi memory leak (rò rỉ vùng nhớ).

Vùng nhớ này sẽ không bị huỷ (vì không có một biến nào được phép quản lý vùng nhớ này nữa) cho đến khi các bạn ngắt nguồn điện. Khi làm việc với những project lớn, việc reset máy thường xuyên là điều khó chấp nhận được, vì nó sẽ làm chậm tiến độ công việc của bạn. Do đó các bạn hãy cẩn thận huỷ tất cả vùng nhớ được tạo ra bằng cấp phát động trước khi kết thúc chương trình.

1. C++ - std::string

Làm việc với chuỗi là vấn đề mà bất cứ lập trình viên nào cũng trải qua. Tuy nhiên, khi mới bắt đầu tiếp xúc với ngôn ngữ C chắc hẳn không ít bạn gặp vấn đề khi khai báo và thao tác trên chuỗi thông qua mảng tĩnh hoặc con trỏ. Với mức độ trừu tượng hóa cao hơn, C++ đã hỗ trợ thêm cho chúng ta std::string dành riêng cho việc xử lý chuỗi.

1. Khái niệm

Trong C++ lớp std::string là sự thể hiện tiêu biểu cho chuỗi văn bản; cho phép thao tác với chuỗi một cách an toàn. Nói cách khác, string giảm thiểu tối đa các vấn đề phát sinh về bộ nhớ khi bạn thao tác với chuỗi trong C.

1. Khác biệt giữa chuỗi trong C++ so với  C

Nói chung, về mặt ý nghĩa, std::string trong C++ vẫn giữ tư tưởng thao tác xử lý chuỗi giống trong C. Tuy nhiên, nó cũng hỗ trợ thêm một số tính năng khác nữa.

Trong C++, vẫn hỗ trợ việc thao tác với chuỗi trên mảng tĩnh như trong C, tuy nhiên, để dễ thao tác cho người sử dụng, C++ hỗ trợ thêm lớp std::string. Nói cách khác, trong C++, chúng ta có thể thao tác với chuỗi trên bằng mảng tĩnh hoặc đối tượng thuộc lớp string.

1. Khai báo và khởi tạo

Cú pháp:

1. string <tên chuỗi>;

Ví dụ:

1. string str1;
2. str1 = "Welcome to ";
3. string str2 = "Stdio";
4. string str3;
5. Độ dài chuỗi

Cú pháp:

1. <tên chuỗi>.length();

Ví dụ:

1. cout << "Getting the length of a C++ string object: " << str2.length() << endl;
2. Truy cập một phần tử bất kỳ trong chuỗi

Cú pháp:

<tên chuỗi>.at(<vị trí cần truy cập>);

Ví dụ:

1. cout << "Accessing a single character: " << str1.at(0) << endl;
2. Nối chuỗi

Cú pháp:

<tên chuỗi thứ 1> + <tên chuỗi thứ 2>;

Ví dụ:

1. str3 = str1 + str2;
2. So sánh độ dài hai chuỗi

Sử dụng các toán tử so sánh  **> == <** để thực hiện so sánh 2 chuỗi.

Cú pháp:

1. <tên chuỗi thứ 1> <toán tử so sánh> <tên chuỗi thứ 2>

Ví dụ:

1. if (str1 > str2)
2. cout << "str1 is bigger than str2!" << endl;
4. if (str1 == str2)
5. cout << "Equal strings!" << endl;
7. if (str1 < str2)
8. cout << "str1 is smaller than str2!" << endl;
9. Tìm chuỗi con

Cú pháp:

<tên chuỗi ban đầu>.find(<tên chuỗi cần tìm kiếm>);

Ý nghĩa: trả về vị trí đầu tiên chuỗi cần tìm xuất hiện trong chuỗi ban đầu.

Ví dụ:

1. string str1 = "Welcome to Stdio";
2. string str2 = "Stdio";
3. size\_t index = str1.find(str2);

Chú ý: Trong C++, khi thao tác với chuỗi, chỉ số vị trí sẽ có kiểu **size\_t** thay vì kiểu integer như thông thường.

1. Sự khác nhau giữa thao tác với chuỗi bằng std::string và với bằng mảng tĩnh

Các bạn hãy theo dõi đoạn mã nguồn đơn giản sau:

#include <iostream>

#include <string>

using namespace std;

int main()

{

/\*string in C++\*/

string str1 = "Welcome to ";

string str2 = "Stdio";

string str\_temp1;

cout << "Accessing a single character: " << str1.at(0) << endl;

cout << "Getting the length of a C++ string object: " << str2.length() << endl;

cout << "Concatenating 2 C++ string objects: " << (str1 + str2) << endl;

cout << "Copying a C++ string object: ";

str\_temp1 = str2;

cout << str\_temp1 << endl;

cout << "Comparing 2 C++ string objects: ";

if (str1 > str2) cout << "str1 is bigger than str2!" << endl;

if (str1 == str2) cout << "Equal strings!" << endl;

if (str1 < str2) cout << "str1 is smaller than str2!" << endl;

/\*string in C\*/

char str3[20] = "Welcome to ";

char str4[20] = "Stdio";

char str\_temp2[20];

printf("Accessing a single character: %c \n", str3[3]);

printf("Getting the length of a C string variable: %d \n", strlen((char\*)str4));

printf("Concatenating 2 C string variables: %s \n", strcat((char\*)str3, (char\*)str4));

printf("Copying a C string variable: %s \n", strcpy((char\*)str\_temp2, (char\*)str4));

printf("Comparing 2 C string variables: ");

if (strcmp((char\*)str3, (char\*)str4) > 0)

printf("str3 is bigger than str4! \n");

if (strcmp((char\*)str3, (char\*)str4) == 0)

printf("Equal strings! \n");

if (strcmp((char\*)str3, (char\*)str4) < 0)

printf("str3 is smaller than str4! \n");

return 0;

}

Như các bạn thấy đó, rõ ràng, string không chỉ mở rộng chức năng mà còn hỗ trợ chúng ta ngay cả việc ghi nhớ. Một trường hợp trong ví dụ trên, thay vì phải nhớ hàm **strcat** để cộng 2 chuỗi như trong C thì khi thao tác với C++ sử dụng string, chỉ cần sử dụng **toán tử ‘+’** là có thể cộng ngay được 2 chuỗi lại một cách dễ dàng.

1. Chuỗi wide-character

Bạn phải khai báo thư viện **wchar.h,** chuỗi định dạng cho chuỗi wchar\_t\* sẽ là “**%ls**” hoặc **“%S”**

1. Khai báo chuỗi wchar\_t

Tương tự như chuỗi char, nhưng phải dùng Macro (L) phía trước:

wchar\_t myString[1024] = L"My Test Unicode String!";

const wchar\_t\* myString = L"Another text";

1. Xuất nhập từ Console
2. Hàm getwchar()

Cú pháp:

wint\_t getwchar (void);

Lấy một kí tự **wchar\_t** từ **stdin**

Hàm trả về giá trị số kiểu **wint\_t** tương ứng với kí tự từ (stdin) khi thành công, hoặc trả về **WEOF,** đồng thời set giá trị errno = **EILSEQ** nếu không thể chuyển đổi về **wchar\_t**

Ví dụ:

/\* getwchar example \*/

#include <stdio.h>

#include <wchar.h>

int main ()

{

wint\_t wc;

fputws (L"Enter text. Include a dot (.) in a sentence to exit:\n",stdout);

do {

wc=getwchar();

putwchar (wc);

} while (wc != L'.');

return 0;

}

1. Hàm wprintf()

Cú pháp

int wprintf (const wchar\_t\* format, ...);

Xuất chuỗi kí tự **format** trực tiếp ra stdout. Chuỗi dịnh dạng cũng giống như trong hàm printf().

Hàm này sẽ nhờ **wcrtomb**() chuyển đổi kí tự **wchar\_t** sang chuỗi kí tự nhiều byte rồi xuất ra stdout..

Hàm trả về số kí tự xuất ra nếu thành công, hoặc set giá trị lỗi cho **ferror**() và trả về số <0 nếu bị lỗi

Ví dụ:

/\* wprintf example \*/

#include <wchar.h>

int main()

{

wprintf (L"Characters: %lc %lc \n", L'a', 65);

wprintf (L"Decimals: %d %ld\n", 1977, 650000L);

wprintf (L"Preceding with blanks: %10d \n", 1977);

wprintf (L"Preceding with zeros: %010d \n", 1977);

wprintf (L"Some different radixes: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);

wprintf (L"floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);

wprintf (L"Width trick: %\*d \n", 5, 10);

wprintf (L"%ls \n", L"A wide string");

return 0;

}

Characters: a A

Decimals: 1977 650000

Preceding with blanks: 1977

Preceding with zeros: 0000001977

Some different radixes: 100 64 144 0x64 0144

floats: 3.14 +3e+000 3.141600E+000

Width trick: 10

A wide string

1. Hàm wscanf()

Cú pháp

int wscanf (const wchar\_t\* format, ...);

Đọc chuỗi định dạng **format** vào stream stdin

Hàm này sẽ nhờ **mbrtowc**() để chuyển đổi từ dạng **wchar**\_t sang chuỗi nhiều byte

Hàm trả về số kí tự được nạp khi thành công , hoặc trả về số < 0 khi bị lỗi.

Ví dụ:

/\* wscanf example \*/

#include <wchar.h>

int main ()

{

wchar\_t str [80];

int i;

wprintf (L"Enter your family name: ");

wscanf (L"%ls",str);

wprintf (L"Enter your age: ");

wscanf (L"%d",&i);

wprintf (L"Mr. %ls, %d years old.\n",str,i);

wprintf (L"Enter a hexadecimal number: ");

wscanf (L"%x",&i);

wprintf (L"You have entered %#x (%d).\n",i,i);

return 0;

}

Enter your family name: Soulié

Enter your age: 29

Mr. Soulié, 29 years old.

Enter a hexadecimal number: ff

You have entered 0xff (255).

1. Xuất nhập từ File stream
2. Hàm fputwc() và putwc()

Cú pháp

Wint\_t fputwc (wchar\_t wc, FILE \* stream)

Wint\_t putwc (wchar\_t wc, FILE \* stream)

Cả 2 hàm đều ghi một kí tự **wchar**\_t vào đối tượng stream và đẩy chỉ số vị trí trong stream lên một.

Hai hàm này sẽ nhờ **wcrtomb**() chuyển đổi kí tự **wchar**\_t sang chuỗi kí tự nhiều byte rồi ghi lên stream.

hàm **fputwc**() và **putwc**() tương đương với nhau.

Nếu kí tự wchar không thể biểu diễn thành kí tự nhiều byte hàm trả về lỗi **WEOF** và set giá trị cho **errno** là **EILSEQ**  
Nếu xảy ra lỗi khi ghi file, hàm trả về lỗi **WEOF** và set giá trị cho **ferrno**.  
Khi thành công, kí tự **wchar**\_t được trả về ở dạng số **wint\_t**

Ví dụ:

/\* example \*/

#include <stdio.h>

int main ()

{

FILE \* pFile;

wchar\_t wc;

pFile = fopen ("example.txt","w");

if (pFile!=NULL) {

for (wc = L'A' ; wc <= L'Z' ; ++wc)

fputwc ( wc , pFile );

fclose (pFile);

}

return 0;

}

1. Hàm fputws()

Cú pháp

int fputws (const wchar\_t\* ws, FILE\* stream);

Hàm ghi một chuỗi kí tự wchar\_t **ws** vào stream từ địa chỉ con trỏ tới khi nó gặp kí tự **null wide** (NULL terminator sẽ không được ghi vào stream)

Dữ liệu trong file được biểu diễn bằng chuỗi kí tự nhiều bytes. Hàm **wcrtomb**() sẽ được gọi để chuyển đổi.

Hàm trả về số > 0 khi thành công.

Nếu xảy ra lỗi chuyển đổi, hàm trả về **EOF** và **errno** được set là **EILSEQ.**

Nếu xảy ra lỗi ghi, hàm trả về **EOF** và set giá trị cho **ferror**

Ví dụ:

/\* fputws example \*/

#include <stdio.h>

int main (){

FILE \* pFile;

wchar\_t sentence [256];

wprintf (L"Enter sentence to append: ");

fgetws (sentence,255,stdin);

pFile = fopen ("mylog.txt","a");

fputws (sentence,pFile);

fclose (pFile);

return 0;

}

1. Hàm fgetwc() và getwc()

Cú pháp

wint\_t getwc (FILE\* stream)

wint\_t fgetwc (FILE\* stream)

2 hàm trên dùng để lấy một kí tự **wchar**\_t từ stream và đẩy chỉ số vị trí trong file lên một

Khi gọi thành công, hàm trả về kí tự mở rộng **wchar**\_t ở dạng số **wint**\_t

Vì wchar\_t được biểu diễn thành nhiều byte ở trong file nên để đọc một kí tự **wchar**\_t , 2 hàm này sẽ nhờ hàm **mbrtowc**() để chuyển đổi từ chuỗi nhiều byte sang dạng **wchar**\_t

Hàm **getwc**() và **fgetwc**() là tương đương với nhau.

Nếu xảy ra lỗi chuyển đổi sang kí tự mở rộng wchar\_t, hàm trả về **WEOF** và set giá trị **errno** là **EILSEQ.**

Khi đọc tới cuối stream, hàm trả về **WEOF** và set chỉ số vị trí cuối file (**feof**())

Nếu xảy ra lỗi đọc file, hàm trả về **WEOF** và set giá trị lỗi cho stream **ferror**().

Ví dụ

/\* getwc example \*/

#include <stdio.h>

#include <wchar.h>

int main (){

FILE \* pFile;

wint\_t wc;

int n = 0;

pFile=fopen ("myfile.txt","r");

if (pFile!=NULL) {

do {

wc = getwc (pFile);

if (wc == L'$') n++;

} while (wc != WEOF);

fclose (pFile);

wprintf (L"The file contains %d dollar sign characters ($).\n",n);

}

return 0;

}

1. Hàm fgetws()

Cú pháp:

wchar\_t\* fgetws (wchar\_t\* ws, int num, FILE\* stream);

Hàm dùng để lấy một chuỗi kí tự **wchar**\_t \* từ stream vào bộ đệm **ws,** tới khi gặp kí tự ngắt dòng hoặc đọc được num-1 kí tự.

NULL terminator vẫn được nạp vào bộ đệm.

Hàm có sử dụng **mbrtowc**() để chuyển đổi kí tự nhiều byte từ stream thành wchar\_t rồi nạp vào bộ đệm.

Khi thành công, hàm trả về con trỏ tới chuỗi.

Bạn có thể dùng foef() để kiểm tra xem stream đã tới cuối file chưa. Nếu chỉ số vị trí nằm ở cuối file trước khi có bất kì kí tự nào được đọc, hàm trả về **NULL**

Nếu xảy ra lỗi khi đọc, giá trị lỗi được set cho **ferror**(), hàm trả về NULL. Nội dung trong bộ đệm có thể đã thay đổi

Ví dụ:

/\* fgetws example \*/

#include <stdio.h>

int main(){

FILE \* pFile;

wchar\_t mystring [100];

pFile = fopen ("myfile.txt" , "r");

if (pFile != NULL)

{

if ( fgetws (mystring , 100 , pFile) != NULL )

fputws ( mystring, stdout );

fclose (pFile);

}

return 0;

}

1. Đặc tính hướng của stream
2. Hàm fwide()

Cú pháp

int fwide (FILE\* stream, int mode);

Thiết lập đặc tính hướng wide cho stream tùy theo **mode** nếu stream là vô hướng.

Hàm sẽ không thể hủy đặc tính hướng của stream nếu stream đã có đặc tính hướng, trừ khi bạn gọi hàm **freopen**() trước rồi gọi **fwide**() sau

Khi mở (**fopen**()), stream là vô hướng. Nhưng khi có bất kì tiến trình IO nào xảy ra trên stream sẽ thiết lập đặc tính hướng tùy theo hàm.

Chúng ta có stream hướng bytes hay hướng wide tương ứng với hàm hướng bytes (**fread**, **fwrite**), hàm hướng wide (**fputwc**, **fgetwc**, **fputws**, **fgetws**)

Khi set **mode = 0,** hàm sẽ trả về thông tin hướng của stream hiện tại.

Giá trị **mode** có thể là :

* **mode =** zero : không thay đổi đặc tính hướng stream.
* **mode** > 0 : set đặc tính hướng stream thành *wide-oriented*.
* **mode** < 0 : set đặc tính hướng stream thành *byte-oriented*

Ví dụ:

/\* fwide example \*/

#include <stdio.h>

#include <wchar.h>

int main (){

FILE \* pFile;

int ret;

pFile = fopen ("myfile.txt","a");

if (pFile) {

fwide (pFile,1);//set wide-oriented stream

ret = fwide (pFile,0);

if (ret>0) puts ("The stream is wide-oriented");

else if (ret<0) puts ("The stream is byte-oriented");

else puts ("The stream is not oriented");

fclose (pFile);

}

return 0;

}

The stream is wide-oriented

1. Chuyển đổi qua lại giữa wchar\_t và char
2. Hàm wcstombs()

Thư viện C cung cấp hàm  **wcstombs()** để chuyển chuỗi kí tự mở rộng **wcs** thành kiểu chuỗi kí tự 1 bytes bắt đầu từ con trỏ chuỗi **des.** Tối đa **n** byte sẽ được ghi vào **des**

Cú pháp:

size\_t wcstombs(char \*des, const wchar\_t \*src, size\_t n)

* **des** – Con trỏ chuỗi có độ dài ít nhất là **n-bytes**
* **src** – Con trỏ chuỗi kiểu **wchar**\_**t** cần chuyển đổi.
* **n** – Số bytes tối đa được ghi vào **des.**

Hàm trả về số bytes được chuyển đổi và ghi vào **des** nếu thành công, ngoại trừ kí tự null. Nếu bị lỗi hàm trả về (-1).

Ví dụ

#include <stdio.h>

#include <stdlib.h>

#define BUFFER\_SIZE 50

int main(){

size\_t ret;

char \*MB = (char \*)malloc( BUFFER\_SIZE );

wchar\_t \*WC = L"http://www.tutorialspoint.com";

/\* converting wide-character string \*/

ret = wcstombs(MB, WC, BUFFER\_SIZE);

printf("Characters converted = %u\n", ret);

printf("Multibyte character = %s\n\n", MB);

return(0);

}

Characters converted = 29

Multibyte character = http://www.tutorialspoint.com

1. Hàm mbstowcs()

Thư viện C cung cấp hàm  **mbstowcs()** để chuyển chuỗi nhiều bytes thành kiểu chuỗi kí tự mở rộng **wchar\_t**

Cú pháp:

size\_t mbstowcs(wchar\_t \*des, const char \*src, size\_t n)

* **des** – Con trỏ kiểu **wchar\_t** , đủ dài để chứa chuỗi kí tự **src**.
* **src** – Chuỗi kí tự cần phiên dịch
* **n** –Quy định số kí tự wchar\_t tối đa được phiên dịch.

Hàm trả về số kí tự được phiên dịch, ngoại trừ kí tự null. Nếu bị lỗi hàm trả về (-1)

Ví dụ

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main(){

int len;

char \*pmbnull = NULL;

char \*pmb = (char \*)malloc( MB\_CUR\_MAX );

wchar\_t \*pwc = L"Hi";

wchar\_t \*pwcs = (wchar\_t \*)malloc( sizeof( wchar\_t ));

printf("Converting to multibyte string\n");

len = wcstombs( pmb, pwc, MB\_CUR\_MAX);

printf("Characters converted %d\n", len);

printf("Hex value of first multibyte character: %#.4x\n", pmb);

printf("Converting back to Wide-Character string\n");

len = mbstowcs( pwcs, pmb, MB\_CUR\_MAX);

printf("Characters converted %d\n", len);

printf("Hex value of first wide character %#.4x\n\n", pwcs);

return(0);

}

Converting to multibyte string

Characters converted 1

Hex value of first multibyte character: 0x19a60010

Converting back to Wide-Character string

Characters converted 1

Hex value of first wide character 0x19a60030

1. Xử lí wchar\_t
2. Độ dài chuỗi wcslen

Trả về độ dài chuỗi wchar\_t

size\_t wcslen (const wchar\_t\* wcs);

Tính từ wcs tới kí tự null wide đầu tiên.

1. Hàm swprintf

int swprintf (wchar\_t\* ws, size\_t len, const wchar\_t\* format, ...)

* ws: bộ đệm chuỗi nhận dữ liệu
* len: Kích thược bộ đệm tính theo số ký tự wchar\_t
* format: chuỗi định dạng tương tự như printf, ngoại trừ %lc thay cho %c, %ls thay cho %s
* ... (additional arguments): danh sách đối số

Hàm trả về số kí tự ghi vào bộ đệm nếu thành công, không tính kí tự null. Kí tự null tự động thêm vào cuối.

Hàm trả về số <0 nếu thất bại.

1. Hàm swscanf

int swscanf (const wchar\_t\* ws, const wchar\_t\* format, ...)

* ws: bộ đệm chuỗi lọc dữ liệu
* len: Kích thược bộ đệm tính theo số ký tự wchar\_t
* format: chuỗi định dạng tương tự như printf, ngoại trừ %lc thay cho %c, %ls thay cho %s
* ... (additional arguments): danh sách đối số

Hàm trả về số lượng đối số lấy dữ liệu thành công.

Trường hợp tham số không hợp lệ, hàm trả về EOF.

1. So sánh chuỗi wcsncmp

int wcsncmp (const wchar\_t\* wcs1, const wchar\_t\* wcs2, size\_t num)

* wcs1: chuỗi 1
* wcs2: chuỗi 2
* num: số kí tự tối đa so sánh

Hàm trả về:

* 0 : Nội dung 2 chuỗi giống nhau.
* > 0 : 2 chuỗi khác nhau và wcs1 dài hơn wcs2;
* < 0 : 2 chuỗi khác nhau và wcs1 ngắn hơn wcs2.

Hàm so sánh num kí tự hoặc cho tới khi gặp kí tự null, 2 kí tự khác nhau.

1. Sao chép wcsncpy

wchar\_t\* wcsncpy (wchar\_t\* destination, const wchar\_t\* source, size\_t num)

* destination: bộ đệm nhận
* source: bộ đệm sao chép sang
* num: Số kí tự wchar\_t cần sao chép

Hàm trả về con trỏ bộ đệm nhận

Hàm không gán kí tự null vào cuối destination nếu sao chép vượt quá bộ đệm destination.

1. Nối chuỗi wcsncat

wchar\_t\* wcsncat (wchar\_t\* destination, const wchar\_t\* source, size\_t num)

* destination: bộ đệm chuỗi được nối vào
* source: bộ đệm chuỗi nối qua
* num: Số kí tự tối đa được nối.

Hàm trả về địa chỉ destination. Kí tự null được chèn vào cuối.

1. Thao tác trên vùng nhớ
2. memset()

Cú pháp của hàm như sau:

1. void \* memset(void \* ptr, int value, size\_t num);

Tác dụng của hàm là set **num** byte giá trị của vùng nhớ tính từ vị trí ptr trỏ tới bằng giá trị **value**

Ví dụ:

1. #include <stdio.h>
2. #include <string.h>
4. int main()
5. {
6. char str[30] = "We are Stdio's Authors";
7. printf("%s\n", str);
9. memset(str + 3, '@', 10); // set 10 byte từ vị trí thứ 4 của str thành @
10. printf("%s\n", str);
12. return 0;
13. }

Kết quả in ra màn hình sẽ là:

1. We are Stdio's Authors
2. We @@@@@@@@@@s Authors

Tuy memset() có công dụng khá hay, nhưng ít khi lập trình viên sử dụng đến, trừ trường hợp muốn set toàn bộ vùng nhớ thành 1 giá trị gì đó, thường là giá trị 0.

1. memcpy()

Cú pháp của hàm như sau:

1. void \* memcpy(void \* destination, const void \* source, size\_t num);

Tác dụng của hàm là sao chép num byte từ vị trí mà source trỏ tới đến vị trí mà destination trỏ tới

Ta có ví dụ sau:

1. #include <stdio.h>
2. #include <string.h>
4. int main()
5. {
6. char str1[30] = "We are Stdio's Authors";
7. char str2[30] = "World";
9. memcpy(str1 + 2, str2, 5);
10. printf("%s\n", str1);
12. return 0;
13. }

Kết quả in ra màn hình như sau:

1. WeWorldStdio's Authors

Tác dụng của hàm này khác với strcpy(), strcpy() sao chép vùng nhớ từ vị trí mà source trỏ tới cho tới vị trí của kí tự ‘\0’ tới destination, do đó destination có giá trị chuỗi giống với source, còn vớimemcpy() thì việc sao chép như thế nào hoàn toàn do người dùng kiểm soát. Tùy trường hợp mà sử dụng 2 hàm này.

1. memcmp()

Cú pháp của hàm này như sau:

1. int memcmp(const void \* ptr1, const void \* ptr2, size\_t num);

Tác dụng: so sánh giá trị các vùng nhớ mà ptr1 và ptr2 trỏ tới theo từng byte, sẽ dừng lại khi so sánh đủ num byte. Khác với strcmp() luôn dừng lại khi gặp kí tự ‘\0’.

Giá trị trả về:

* -1: Khi giá trị của byte đầu tiên không trùng của ptr1 nhỏ hơn ptr2.
* 0: Khi giá trị của 2 vùng nhớ bằng nhau
* 1: Khi giá trị của byte đầu tiên không trùng của ptr1 lớn hơn ptr2.

Về độ lớn giá trị của các kí tự các bạn có thể tham khảo bài viết [Bảng Mã ASCII](#_Bảng_mã_ASCII)

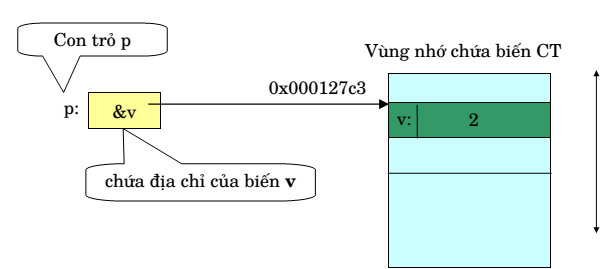
###  Con trỏ trong C,C++

Con trỏ (Pointer) là đặc trưng và là một trong những sức mạnh lớn nhất trong lập trình C,C++. Vì thế việc tìm hiểu và vận dụng con trỏ là điều không thể thiếu đối với một lập trình viên. Bài viết này sẽ giúp cho người đọc có cái nhìn sâu sắc hơn về con trỏ trong lập trình C,C++. Qua đó sẽ giúp các bạn sử dụng con trỏ một cách linh hoạt, tránh được những sai sót và làm việc một cách hiệu quả.

1. Con trỏ là gì?

Con trỏ là một biến lưu trữ địa chỉ bộ nhớ (memory address). Thông thường, một biến lưu trữ giá trị, ví dụ như 5, "stdio.vn", hoặc "rye nguyen", … và những giá trị đó được lưu trữ ở một vị trí cụ thể trong vùng nhớ máy tính.

Biến con trỏ khác biệt hoàn toàn, vì nó lưu trữ memory address như là giá trị của nó và có khả năng "trỏ" (point) đến giá trị lưu trữ trong địa chỉ bộ nhớ đó.



1. Con trỏ với biến

Giống như biến thông thường, con trỏ cũng có kiểu dữ liệu và cần được khai báo trước khi sử dụng. Kiểu dữ liệu có thể là:

* Kiểu dữ liệu được định nghĩa sẵn (built-in data type): int, float, char, void, …
* Kiểu dữ liệu là cấu trúc do người dùng định nghĩa (user-define data type): struct, union.
* Kiểu dữ liệu là lớp do người dùng định nghĩa (*Class*).

Chương trình sau minh họa việc khai báo con trỏ đối với một số kiểu dữ liệu thường dùng:

1. int main()
2. {
3. int \*ip; //Pointer to an integer
4. float \*fp; //Pointer to a float
5. char \*cp; //Pointer to a character
6. void \*vp; //Pointer to an unidentified data type
8. return 0;
9. }

Chú ý rằng một kí tự \* giữa kiểu dữ liệu và tên biến có nghĩa là biến mà bạn khai báo là một biến con trỏ. Trong ngữ cảnh này \* không phải là phép nhân. Ngoài ra, không có sự khác biệt gì nếu \* gần với kiểu dữ liệu hơn hay gần với tên biến hơn.

Tùy vào mỗi lập trình viên chọn cách khai báo nào mà họ cảm thấy phù hợp. Tuy nhiên, trong trường hợp khai báo nhiều hơn một biến trên cùng một dòng, chỉ những biến nào có kí tự \* phía trước tên biến (không kể khoảng trắng) mới là biến con trỏ.

Bởi vì con trỏ chỉ lưu giữ địa chỉ, nên khi bạn chỉ được gán giá trị tham chiếu cho một con trỏ.

Để lấy địa chỉ của một biến, chúng ta sử dụng toán tử địa chỉ address-of operator &.

Để gán giá trị cho 1 con trỏ, bạn phải ép kiểu dữ liệu con trỏ

1. int main()
2. {
3. int a = 5;
4. int \*des = (int\*)5;
5. int \*ip = &a;
7. return 0;
8. }

Khi đó biến ip sẽ lưu trữ địa chỉ của byte đầu tiên của biến a. Lưu ý toán tử & ở đây là *unary operator – toán tử một ngôi*, hoàn toàn khác với *toán tử* & *hai ngôi* (*bitwise*).

Một toán tử khác thường sử dụng với con trỏ là toán tử truy xuất nội dung (\*).Toán tử (\*) nằm trước con trỏ để lấy nội dung của địa chỉ mà biến đó trỏ vào.

1. #include <iostream>
3. using namespace std;
5. int main()
6. {
7. int a = 5;
9. int \*ip = &a;
10. cout << \*ip << endl;
12. void \*vp = &a;
13. cout << \*(int \*)vp << endl;// ép kiểu và truy xuất nội dung con trỏ vp (1)
15. return 0;
16. }

Trong chương trình trên, để lấy giá trị được lưu trữ trong biến a có hai cách: gọi trực tiếp biến a hoặc gọi gián tiếp qua con trỏ ip. Khi đó, ta có thể thay đổi giá trị của biến a thông qua con trỏ ip như sau:

\*ip = 11;

**Một lưu ý là con trỏ kiểu void có thể lưu địa chỉ của biến bất kì, nhưng không thể truy xuất nội dung (toán tử \*) trực tiếp, vì con trỏ void không biết kiểu dữ liệu của đối tượng mà nó trỏ đến.**

**Do đó, lập trình viên cần định kiểu hoặc gán nó cho một con trỏ khác có kiểu dữ liệu cụ thể trước khi sử dụng.**

Ngoài ra ta có thể gán trực tiếp một con trỏ cho một con trỏ khác. Tuy nhiên, khi đó hai con trỏ sẽ cùng lưu trữ memory address giống nhau, dễ gây ra lỗi memory leak và dangling pointer nếu lập trình viên không quản lý chương trình tốt. Do đó, lập trình viên cần kiểm soát con trỏ trong suốt quá trình sử dụng để đạt hiệu quả tốt nhất.

1. Con trỏ với hàm
2. Hàm trả về con trỏ

Ví dụ sau dùng hàm **getRandom()** trả về địa chỉ vùng nhớ kiểu **int** của một mảng các số ngẫu nhiên được khai báo với đặc tính **static**.

Khi này biến mảng được lưu trữ trong vùng nhớ data, không bị mất giá trị khi hàm gọi kết thúc. Nên ta có thể gán hàm cho một con trỏ.

#include <stdio.h>

#include <time.h>

/\* function to generate and retrun random numbers. \*/

int \* getRandom( )

{

static int r[10];

int i;

/\* set the seed \*/

srand( (unsigned)time( NULL ) );

for ( i = 0; i < 10; ++i)

{

r[i] = rand();

printf("%d\n", r[i] );

}

return r;

}

/\* main function to call above defined function \*/

int main ()

{

/\* a pointer to an int \*/

int \*p;

int i;

p = getRandom();

for ( i = 0; i < 10; i++ )

{

printf("\*(p + [%d]) : %d\n", i, \*(p + i) );

}

return 0;

}

1. Con trỏ là tham số hàm

Khi khai báo con trỏ là tham số của hàm. Bạn có thể gọi hàm với tham số con trỏ hoặc tham chiếu của biến, hoặc để tham số là *Null ,* tức không truyền con trỏ nào cả

Ví dụ:

#include <stdio.h>

#include <time.h>

void getSeconds(unsigned long \*par);

int main ()

{

unsigned long sec, \*p;

getSeconds( p); // Truyền với đối số hàm là kiểu con trỏ

getSeconds( &sec ); // Truyền với tham chiếu của biến

/\* print the actual value \*/

printf("Number of seconds: %ld\n", \*p );

printf("Number of seconds: %ld\n", sec );

return 0;

}

void getSeconds(unsigned long \*par)

{

/\* get the current number of seconds \*/

\*par = time( NULL ); // Gọi hàm với tham số con trỏ *Null*

return;

}

Kết quả

Number of seconds :1294450468

Number of seconds :1294450468

1. Con trỏ với mảng

Ví dụ:

char array[10];

char \*arrayptr = &array[O];

Trong ví dụ trên, \*arary\_ptr cũng giống như array[0], \*(array\_ptr + 1) tương tự array[1].

Mỗi phần tử trong mảng tĩnh được gán cho một địa chỉ cụ thể. Ví dụ array hay array[0] sẽ có địa chỉ là 0xFF000024. Array[1] sẽ là 0xFF000025…

C++ cho bạn phương pháp nhanh gọn để xử lí mảng. Thay vì viết

array\_ptr = &array[0];

Bạn có thể viết :

array\_ptr = array;

C++ xóa mờ ranh giới giữa con trỏ và mảng, xem cả hai là như nhau. Bạn có thể sử dụng biến mảng như con trỏ

Ví dụ duyệt mảng bằng con trỏ và dừng lại khi gặp phần tử bằng zero:

Rewriting this program to use pointers gives us Example 15-4.

#include <iostream.h>

int array[10] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};

int \*array\_ptr;

main(){

array\_ptr = array;

while ((\*array\_ptr) != 0)

++array\_ptr;

cout << "Number of elements before zero << (array\_ptr - array) << '\n';

return (0);

}

Khi bạn dùng toán tử [] để truy xuất ngẫu nhiên phần tử mảng (array[index]!=0), trình biên dịch phải qua bước tính toán index, chậm hơn so với tham chiếu bằng con trỏ (\*array\_ptr)!=0.

Khi truyền mảng vào hàm, C++ đơn thuần là truyền con trỏ tới mảng vào. Nếu bạn đặt (&) trước tham số mảng, C++ sẽ báo lỗi

1. Kích thước của con trỏ

Như các bạn đã biết, những biến có kiểu dữ liệu khác nhau sẽ được cấp phát vùng nhớ có kích thước khác nhau. Còn với biến con trỏ thì như thế nào?

Để trả lời cho câu hỏi trên, chúng ta cần trả lời câu hỏi này: biến con trỏ dùng để làm gì? Như tôi có đề cập ở trên, biến con trỏ dùng để lưu trữ memory address của một biến khác. Dù cho biến con trỏ được khai báo như thế nào đi nữa thì giá trị được lưu trữ trong nó cũng là một memory address.

Đến đây các bạn đã thấy được rằng đối với biến con trỏ, kích thước của nó là một hằng số, bất kể kiểu dữ liệu mà nó được khai báo. Hằng số này phụ thuộc vào OS, IDE, Compiler, … Trên hệ điều hành 32-bit, hằng số này là 4 byte.

Vậy nếu kích thước của con trỏ là một hằng số, tại sao lại có nhiều kiểu biến con trỏ như vậy? Chúng ta cùng xét ví dụ sau:

1. #include <iostream>
3. using namespace std;
5. int main ()
6. {
7. int a = 257;
8. char \*cp = (char \*)&a;
10. cout << \*cp;
12. return 0;
13. }

Kết quả in ra màn hình là ký tự tương ứng với số nhị phân 1 trong bảng ascii . Để lý giải cho trường hợp này, tôi xin minh họa bằng cách cụ thể hóa vùng nhớ của máy tính như sau:

0001 0000 0001 0000

Do kiểu char có kích thước 1 byte, khi ép kiểu (char \*), biến cp chỉ nhận được dữ liệu trong byte đầu tiên (byte trái nhất) của biến a, dẫn đến việc đọc thiếu một phần dữ liệu có trong biến a. Kiểu dữ liệu sẽ quy định cho biến con trỏ đọc số byte tương ứng kể từ ô nhớ đầu tiên đó.

1. Các phép toán trên con trỏ

C++ cho phép thực hiện các phép toán cộng, trừ trên con trỏ thông qua các toán tử +, -, ++, --. Khi đó, giá trị địa chỉ lưu trữ trong con trỏ sẽ được cộng/trừ số byte tương ứng với kiểu dữ liệu của con trỏ. Thao tác tương tự như với dữ liệu kiểu số, mời các bạn xem ví dụ sau:

1. int main()
2. {
3. int a = 5;
4. int \*ip = &a;
6. cout << ip + 1;
7. cout << ip++; //cout << ip; ip = ip + 1
8. cout << ++ip; //ip = ip + 1; cout << ip
10. return 0;
11. }

Ở dòng thứ 6, giá trị in ra màn hình là địa chỉ của biến a + **sizeof(int),** con trỏ ip vẫn mang giá trị là địa chỉ của biến a. Dòng 7 và 8 thao tác tương tự như với dữ liệu kiểu số, sau khi thực hiện, giá trị chứa trong con trỏ ip đã bị thay đổi.

Ngoài các phép toán giữa con trỏ với số, ta còn có thể thực hiện phép trừ hai con trỏ cùng kiểu**.** Khi đó, giá trị trả về là một số nguyên độ chênh lệch địa chỉ giữa hai con trỏ đó với đơn vị là kiểu dữ liệu của con trỏ.

1. Con trỏ tới con trỏ C,C++

Bình thường, 1 con trỏ chứa địa chỉ của biến.

Con trỏ tới con trỏ là dạng phức tạp, một chuỗi con trỏ. Con trỏ đầu sẽ chứa địa chỉ con trỏ thứ hai, con trỏ thứ hai sẽ chứa địa chỉ thực của biến như biểu đồ dưới



Cách khai báo con trỏ tới con trỏ kiểu int

int \*\*var;

Để truy xuất giá trị gián tiếp từ con trỏ tới con trỏ yêu cầu double toán tử \*. Ví dụ

#include <stdio.h>

int main (){

int var;

int \*ptr;

int \*\*pptr;

var = 3000;

/\* take the address of var \*/

ptr = &var;

/\* take the address of ptr using address of operator & \*/

pptr = &ptr;

/\* take the value using pptr \*/

printf("Value of var = %d\n", var );

printf("Value available at \*ptr = %d\n", \*ptr );

printf("Value available at \*\*pptr = %d\n", \*\*pptr);

return 0;

}

Kết quả

Value of var = 3000

Value available at \*ptr = 3000

Value available at \*\*pptr = 3000

1. Hủy con trỏ C,C++

Gán giá trị *Null* cho con trỏ khi bạn không sử dụng nữa, con trỏ null không chứa địa chỉ nào cả và có giá trị là 0.

Ví dụ

#include <stdio.h>

int main ()

{

int \*ptr = NULL;

printf("The value of ptr is : %x\n", ptr );

return 0;

}

Kết quả

The value of ptr is 0

Hầu hết các hệ điều hành, chương trình không được phép truy xuất bô nhớ địa chỉ là 0, vì đây là vùng dành riêng cho hệ thống.

Ví dụ để kiểm tra con trỏ *Null*

if(ptr) /\* succeeds if p is not null \*/

if(!ptr) /\* succeeds if p is null \*/

1. Ưu điểm của con trỏ

Khả năng truy cập bộ nhớ máy tính thông qua con trỏ là một thế mạnh của ngôn ngữ C, C++ so với một số ngôn ngữ lập trình khác. Nó giúp việc viết code trở nên hiệu quả và linh hoạt hơn. Tuy nhiên, việc vận dụng con trỏ trong lập trình sẽ làm tăng độ phức tạp của chương trình, đặc biệt là những chương trình lớn. Do đó đòi hỏi lập trình viên cần đầu tư tìm hiểu chuyên sâu về nó.

Một số ứng dụng của con trỏ:

* Sử dụng vùng nhớ heap (heap segment) thông qua cấp phát động.
* Thực thi tính Polymorphism (Đa hình) trong lập trình hướng đối tượng.

1. Truy xuất thành viên với con trỏ và tham chiếu

Truy xuất thành viên là thao tác phổ biến với cấu trúc và lớp thông qua toán tử thành viên (.).

It is common to have either a pointer or a reference to a struct (or class). As you learned previously, you can select the member of a struct using the member selection operator (.):

Xem cấu trúc **Line** như sau:

struct Line{

int length;

};

Cú pháp chung cho biến và tham chiếu:

Line line;

Line &refLine = line;

line.length = 5;

refLine.length = 5;

Tuy nhiên với con trỏ, bạn phải truy xuất nội dung để sử dụng toán tử thành viên (.)

Line line;

Line \*ptr = &line;

(\*ptr).length = 5;

Chú ý: Tham chiếu lại tới biến của con trỏ phải nằm trong đấu ngặc đơn, vì toán tử (.) có độ ưu tiên cao hơn toán tử (\*)

C++ cung cấp thêm toán tử thành viên khác riêng cho con trỏ là (->). Hai dòng lệnh sau là tương đương nhau:

Line line;

Line \*ptr = &line;

(\*ptr).length = 5;

ptr->length = 5;

1. Con trỏ void
2. Tổng quát:

The void pointer, also known as the generic pointer, is a special type of pointer that can be pointed at objects of any data type! A void pointer is declared like a normal pointer, using the void keyword as the pointer’s type:

void \*ptr; // ptr is a void pointer

A void pointer can point to objects of any data type:

int nValue;

float fValue;

struct Something

{

int n;

float f;

};

Something sValue;

void \*ptr;

ptr = &nValue; // valid

ptr = &fValue; // valid

ptr = &sValue; // valid

However, because the void pointer does not know what type of object it is pointing to, it cannot be dereferenced directly! Rather, the void pointer must first be explicitly cast to another pointer type before it is dereferenced.

int value = 5;

void \*voidPtr = &value;

//cout << \*voidPtr << endl; // illegal: cannot dereference a void pointer

int \*intPtr = static\_cast<int\*>(voidPtr); // however, if we cast our void pointer to an int pointer...

cout << \*intPtr << endl; // then we can dereference it like normal

Kết quả:

5

The next obvious questions is: If a void pointer doesn’t know what it’s pointing to, how do we know what to cast it to? Ultimately, that is up to you to keep track of.

Here’s an example of a void pointer in use:

#include <iostream>

enum Type

{

INT,

FLOAT,

CSTRING,

};

void printValue(void \*ptr, Type type)

{

switch (type)

{

case INT:

std::cout << \*static\_cast<int\*>(ptr) << '\n'; // cast to int pointer and dereference

break;

case FLOAT:

std::cout << \*static\_cast<float\*>(ptr) << '\n'; // cast to float pointer and dereference

break;

case CSTRING:

std::cout << static\_cast<char\*>(ptr) << '\n'; // cast to char pointer and dereference

break;

}

}

int main()

{

int nValue = 5;

float fValue = 7.5;

char szValue[] = "Mollie";

printValue(&nValue, INT);

printValue(&fValue, FLOAT);

printValue(szValue, CSTRING);

return 0;

}

Kết quả:

5

7.5

Mollie

1. Con trỏ void hỗn hợp

Void pointers can be set to a null value:

void \*ptr = 0; // ptr is a void pointer that is currently a null pointer

Although some compilers allow deleting a void pointer that points to dynamically allocated memory, doing so should be avoided, as it can result in undefined behavior.

It is not possible to do pointer arithmetic on a void pointer. This is because pointer arithmetic requires the pointer to know what size object it is pointing to, so it can increment or decrement the pointer appropriately.

Note that there is no such thing as a void reference. This is because a void reference would be of type void &, and would not know what type of value it referenced.

1. Kết luận

In general, it is a good idea to avoid using void pointers unless absolutely necessary, as they effectively allow you to avoid type checking. This allows you to inadvertently do things that make no sense, and the compiler won’t complain about it. For example, the following would be valid:

int nValue = 5;

printValue(&nValue, CSTRING);

But who knows what the result would actually be!

Although the above function seems like a neat way to make a single function handle multiple data types, C++ actually offers a much better way to do the same thing (via function overloading) that retains type checking to help prevent misuse. Many other places where void pointers would once be used to handle multiple data types are now better done using templates, which also offer strong type checking.

However, very occasionally, you may still find a reasonable use for the void pointer. Just make sure there isn’t a better (safer) way to do the same thing using other language mechanisms first!

1. Từ khóa Const với con trỏ:

Như chúng ta đã biết, con trỏ là một trong những thành phần quan trọng nhất của C++, nó là một biến, do đó nó cũng có thể sử dụng từ khóa **const**, tùy vào trường hợp mà có thể gọi là con trỏ hằng hoặc hằng con trỏ.

1. Con trỏ hằng

Con trỏ hằng là loại con trỏ có thể thay đổi được địa chỉ vùng nhớ mà nó lưu trữ, nhưng không thể thay đổi được giá trị của vùng nhớ thông qua con trỏ đó.

Cú pháp khai báo con trỏ hằng:

1. **const <kiểu dữ liệu con trỏ> <tên con trỏ> = <địa chỉ>;**

Ta có ví dụ sau:

1. int main()
2. {
3. int x = 7;
4. int y = 13;
6. **const** int \*p = &x; // Con trỏ hằng
8. \*p = 100; // Báo lỗi
9. p = &y; // Không báo lỗi
11. return 0;
12. }
13. Hằng con trỏ

Ngược lại với con trỏ hằng, hằng con trỏ là loại con trỏ không thể thay đổi được địa chỉ vùng nhớ mà nó lưu trữ, nhưng vẫn có thể thay đổi được giá trị của vùng nhớ đó thông qua con trỏ đó.

Cú pháp khai báo hằng con trỏ

1. <kiểu dữ liệu con trỏ> **const** <tên con trỏ> = <địa chỉ>;

Ta cũng có ví dụ như sau để minh họa:

1. int main()
2. {
3. int x = 7;
4. int y = 13;
6. int \***const** p = &x; // Hằng con trỏ
8. \*p = 100; // Không báo lỗi
9. p = &y; // Báo lỗi
11. return 0;
12. }
13. Mở rộng

Trong trường hợp ta muốn sử dụng một con trỏ vừa không thay đổi được địa chỉ vùng nhớ mà nó lưu trữ, vừa không thay đổi được giá trị của vùng nhớ đó, ta sử dụng hai từ khóa **const** như sau:

1. **const** <kiểu dữ liệu con trỏ> **const** <tên con trỏ> = <địa chỉ>;
2. Con trỏ mảng và mảng con trỏ:

Con trỏ mảng con trỏ quản lý một mảng, còn mảng con trỏ là mảng trong đó các phần tử là các con trỏ. Phân biệt và sử dụng hai khái niệm này sẽ giúp cho lập trình viên hiểu được vấn đề tốt hơn, từ đó giải quyết bài toán của mình dễ dàng hơn.

1. Con trỏ mảng

Đây là con trỏ quản lý vùng nhớ của mảng, cả mảng tĩnh và mảng động đều có con trỏ mảng.

1. int Static\_Array[10]; // C,C++
2. int \*Dynamic\_Array\_Cpp = new int[10]; // cấp phát động trong C++
3. int \*Dynamic\_Array\_C = (int\*)calloc(10, sizeof(int)) // cấp phát động trong C

Như cách khai báo ở trên, thì Static\_Array là mảng tĩnh, cònDynamic\_Array là mảng động. Static\_Array và Dynamic\_Array cũng chính là tên của con trỏ quản lý vùng nhớ của các mảng. Địa chỉ mà hai con trỏ này lưu trữ chính là địa chỉ của phần tử đầu tiên trong mảng.

Khi muốn sử dụng vùng nhớ của các phần tử khác trong mảng, ví dụ như phần tử thứ 6 ta có 2 cách sau:

Cách thông thường:

1. Static\_Array[5] = 15;
2. Dynamic\_Array\_Cpp[5] = 15; //C++
3. \*Dynamic\_Array\_C[5] = 15; //C

Hoặc:

1. \*(Static\_Array + 5) = 15;
2. \*(Dynamic\_Array\_Cpp + 5) = 15;
3. \*(Dynamic\_Array\_C + 5) = 15;
4. Mảng các con trỏ

Con trỏ cũng là một biến, do đó nó có thể được lưu trữ trong các mảng. Mảng này được gọi là mảng con trỏ.

Khai báo cấp phát động như sau:

1. int \*\*Pointer\_Array = new int\*[10];

Chúng ta có thể sử dụng mảng này như một mảng hai chiều, để làm được điều đó, chúng ta làm như sau, giả sử chúng ta cần tạo một mảng 2 chiều có kích thước **m x n**:

1. int m = 10, n = 5;
2. int \*\*Pointer\_Array = new int\*[m];
4. for (int i = 0; i < m; i++)
5. {
6. Pointer\_Array[i] = new int[n];
7. }

Chúng ta có thể sử dụng các phần tử của mảng như một mảng 2 chiều tĩnh thông thường:

1. Pointer\_Array[0][0] = 0;
2. Pointer\_Array[0][1] = 2;
3. Pointer\_Array[0][2] = 7;
4. ……

Khi không có nhu cầu sử dụng mảng này nữa, chúng ta phải giải phóng vùng nhớ cho nó:

1. for (int i = 0; i < m; i++)
2. {
3. delete[] Pointer\_Array[i]; // hủy từng phần tử trước
4. }
5. delete[] Pointer\_Array; // Cuối cùng là hủy mảng

### Cấp phát tĩnh và cấp phát động:

##### Giới thiệu

Khi chúng ta khai báo một biến hoặc một mảng, chúng ta cần cung cấp vùng nhớ cho chúng quản lý. Có hai cách cung cấp là cấp phát tĩnh và cấp phát động, chúng ta cần phân biệt và hiểu rõ chúng để sử dụng các vùng nhớ một cách hiệu quả hơn.

##### Cấp phát tĩnh

Đây là kiểu cấp phát thông dụng trong ngôn ngữ C++, là kiểu cấp phát mà chúng ta đã biết trong các bài học trước.

1. int a = 5;
2. int Arr[10];

Ta gọi Arr là mảng tĩnh. Cấp phát tĩnh sẽ lưu trữ toàn bộ giá trị của biến a và mảng Arr vào bộ nhớ Stack.

**Lưu ý:** khi khai báo mảng tĩnh, **tham số ở trong cặp ngoặc vuông [ ] phải là số nguyên dương hoặc một hằng số.**

##### Cấp phát động trong C

**A.** Stack là một phần bộ nhớ đặc biệt được quản lý một các tự động bởi CPU theo cơ chế FILO (First In, Last Out) dùng để chứa các biến tạm trong quá trình thực hiện các funtion. Mỗi khi ta khai báo một biến trong hàm thì biến đó được đẩy vào stack và khi hàm return thì tất các biến tạm của hàm đó bị xóa vĩnh viễn. Vùng nhớ này tùy thuộc vào loại CPU và OS và nó ngày càng lớn với các thế hệ CPU hiện đại.

**B.** Heap là một vùng nhớ mà không được CPU quản lý một cách tự động mà lập trình viên phải làm việc này (cấp phát, giải phóng).

Cấp phát động trong C bằng các hàm như **calloc(**) và **malloc()**.

Nếu heap được cấp phát mà không giải phóng thì phần bộ nhớ đó sẽ không sử dụng được và có thể dẫn đến memory leak( thiếu bộ nhớ). Heap có thể được truy cập bởi bất cứ tiến trình nào.

Tốc độ truy cập Heap thấp hơn Stack vì nó còn có cơ chế con trỏ tới ô nhớ đầu tiên và đọc theo kiểu serial.

**C.** Buffer cũng giống Heap ở chỗ lập trình viên phải tự quản lý nhưng nó chung chung hơn.

Nó có kiểu dữ liệu, có thể là một string, mảng byte hay bất cứ gì lập trình viên muốn lưu lại để xử lý sau.

**D.** Ngôn ngữ C cung cấp cho chúng ta 4 hàm dùng để cấp phát và giải phóng bộ nhớ động, nhằm giải quyết các vấn đề liên quan đến bộ nhớ. Đó là **malloc**, **calloc**, **realloc** và **free**.

Các hàm này đều nằm trong thư viện <stdlib.h>

Để biết cách sử dụng các hàm trên ta phải xem nguyên mẫu hàm (prototype) của nó:

1. Hàm malloc:

void \* malloc ( size\_t size );

*// Hàm này dùng để cấp phát bộ nhớ động cho một biến với kích thước size (Đơn vị là byte).*

// size\_t là kiểu unsigned int, số nguyên dương

***Ví dụ:***

int \*bobby;

bobby = (int \*) malloc (5 \* sizeof(int));

Cái không hay của malloc là các phần tử mới được tạo ra không được khởi gán, mang giá trị bất định!

1. Hàm calloc:

void \* calloc ( size\_t num, size\_t size );  
*//Hàm này dùng để cấp phát bộ nhớ động cho một mảng với size\_t num là số phần tử của mảng.*

Cái hay của calloc còn là ở chỗ các phần tử mới tạo ra sẽ được set to zeros.

***Ví dụ:***

int \* bobby;

bobby = (int \*)[calloc](http://cppreference.com/stdmem/calloc.html) (5, sizeof(int));

1. Hàm realloc:

void \*realloc ( void \*ptr, size\_t size );   
//Dùng để thay đổi kích thước bộ nhớ đã cấp phát với ptr là địa chỉ của bộ nhớ đã cấp phát và size là kích thước muốn cấp phát lại **(cập nhập)** cho vùng nhớ đó.

***Ví dụ****:*

int \*pi = malloc(12);  // cấp phát 12 bytes cho pi  
pi = realloc(pi,14);  // tăng bộ nhớ của pi lên 2 bytes

1. **Hàm free:**

void free (void \*ptr ); //*Dùng để giải phóng vùng nhớ đã cấp phát.*

Chương trình ví dụ sử dụng cấp phát và giải phóng bộ nhớ động.

1. /\* calloc example \*/
2. #include <stdio.h>
3. #include <stdlib.h>
4. int main () {
5. int i, n;
6. int \*pData;
7. printf ("Amount of numbers to be entered: ");
8. scanf ("%d",&i);
9. pData = (int\*) calloc(i, sizeof(int));
10. if (pData == *NULL*) exit (1);
11. for (n=0; n<i; n++){
12. printf ("Enter number #%d: ", n);
13. scanf ("%d", &pData[n]);
14. }
15. printf ("You have entered: ");
16. for (n=0; n<i; n++) printf ("%d ”, pData[n]);
17. free (pData);  // sau khi sử dụng phải giải phóng bộ nhớ
18. **return** 0;
19. }

##### Cấp phát động trong C++

1. Toán tử new

Khi sử dụng cấp phát động, chúng ta buộc phải sử dụng đến biến con trỏ và toán tử new

1. int \*a = new int;
2. int n = 10;
3. int \*Arr = new int[n];

Biến a là một biến con trỏ, chúng ta chỉ cho phép nó quản lý một vùng giá trị có kích thước như biến int thông thường. Khi muốn truy xuất tới giá trị vùng nhớ đó ta sử dụng tới toán tử \*

1. \*a = 5;
2. cout << \*a;

Ta gọi Arr là mảng động. Cấp phát động sẽ lưu trữ toàn bộ vùng nhớ mà a và Arr tham chiếu tới vào bộ nhớ Heap.

Có thể sử dụng biến làm tham số trong cặp ngoặc vuông [ ], ví dụ trên chúng ta sử dụng biến n. Do đó, chúng ta hoàn toàn có thể tính toán trước kích thước của mảng động trước khi cấp phát, tránh tình trạng lãng phí bộ nhớ. Hơn nữa chúng ta có thể truy xuất đến từng phần tử trong mảng bằng [ ] giống như trong cấp phát tĩnh

1. Arr[0] = 5;
2. Arr[1] = 10;
3. ….

Cấp phát động lưu trữ vùng nhớ trong Heap, mà ta đã biết rằng, Heap là vùng nhớ động, nó có thể mở rộng cho đến hết RAM, nên chúng ta không cần lo lắng về sự thiếu bộ nhớ. Trong khi đối với Stack, kích thước của nó là cố định, dễ bị thiếu vùng nhớ hơn so với Heap, tuy nhiên cũng không cần quá lo lắng khi cấp phát tĩnh, trình biên dịch cung cấp cho chúng ta một vùng nhớ Stack với kích thước đủ sử dụng. Tôi đã kiểm tra thử bằng Visual Stdio 2013 thì thấy chúng ta có thể cấp phát tĩnh cho một mảng kiểu long với kích thước hơn 250000 phần tử.

Tuy nhiên cấp phát động cũng có một số nhược điểm.

1. Thu hồi vùng nhớ

Thứ nhất là khai báo phức tạp hơn cấp phát tĩnh. Thứ hai là bạn phải hoàn toàn kiểm soát được vùng nhớ động mà mình đang sử dụng, vì trình biên dịch không tự động thu hồi vùng nhớ như khi bạn sử dụng cấp phát tĩnh. Để làm được điều này chúng ta phải sử dụng đến toán tử delete

1. delete a;
2. delete[] Arr;

Toán tử delete cho phép ta thu hồi vùng nhớ đã được cấp phát, khi đó vùng nhớ đó sẽ trở thành vùng nhớ tự do, có thể dùng để cấp phát tiếp hoặc cho các process khác sử dụng. Bạn phải lưu ý sử dụng delete cho đúng cách, **delete đối với vùng nhớ đơn và delete[] đối với mảng.**

**Bất cứ khi nào không có nhu cầu sử dụng đến vùng nhớ đã được cấp phát động, bạn nên phải giải phóng nó bằng toán tử delete để tránh sự lãng phí vùng nhớ đáng tiếc.**

### Mảng:

1. Mảng tĩnh
2. Cấu trúc:

* Số lượng các phần tử cố ₫ịnh
* Các phần tử có cùng kiểu
* Các phần tử ₫ược sắp xếp kế tiếp trong bộ nhớ
* Có thể truy nhập từng phần tử một cách tự do theo chỉ số hoặc theo ₫ịa chỉ.

1. Khai báo mảng:

Tương tự như khai báo một biến bình thường, một mảng có thể dễ dàng được khai báo như sau:

1. kiểu\_dữ\_liệu tên\_mảng [số\_phần\_tử];

* Kiểu dữ liệu ở đây có thể là các kiểu dữ liệu có sẵn hay do người dùng tự định nghĩa.
* Tên mảng được đặt cùng quy tắc với tên biến trong C++
* C chỉ chấp nhận Số phần tử của mảng là số nguyên dương hay hằng số lớn hơn 0.
* C++ chấp nhận số phần tử là biến hay biến const

**Ví dụ**

Ta khai báo mảng int có tên number gồm 5 phần tử như sau:

1. int number [5];

**Lưu ý**

Vì đây là khai báo tĩnh, số vùng nhớ cấp phát cho mảng được xác định từ giai đoạn biên dịch, trước khi chương trình chạy. Cho nên, số\_phần\_tử ở đây phải mang giá trị hằng số.

1. #include <iostream>
2. using namespace std;
4. int main()
5. {
6. const int a = 5;
7. int b = 5;
8. int c[a]; // Khong bao loi, vi a là hang so.
9. int d[5]; // Khong bao loi, vi 5 là hang so.
10. int e[b]; // Bao loi, vi b khong la hang so.
11. }
12. Khởi tạo mảng

Sau khi khai báo, nếu chúng ta không khởi tạo giá trị ban đầu cho các phần tử, thì các phần tử sẽ mang giá trị rác. Tuy nhiên, nếu là mảng static, thì các phần tử sẽ được khởi tạo giá trị 0. Ví dụ khi khai báo:

1. static int a[5];

ta sẽ được mảng a chứa 5 giá trị 0 mà không cần khởi tạo.

1. Khởi tạo ngay khi khai báo

Chúng ta khởi tạo giá trị cho các phần tử ngay khi khai báo bằng danh sách khởi tạo.

1. kiểu\_dữ\_liệu tên\_mảng [số\_phần\_tử] = {giá\_trị\_1, giá\_trị\_2, giá\_trị\_3...};

**Ví dụ**

khi khai báo

1. int a[5] = { 1, 2, 3, 4, 5 };

ta được mảng a với các giá trị a[0], a[1], a[2], a[3], a[4] có giá trị lần lượt là 1, 2, 3, 4, 5.

**Lưu ý**

* Nếu số lượng giá trị trong dấu {...} lớn hơn **số\_phần\_tử**của mảng, trình biên dịch sẽ báo lỗi.
* Nếu số lượng giá trị trong dấu {...} nhỏ hơn **số\_phần\_tử**của mảng, những phần tử còn lại sẽ được gán giá trị mặc định là 0.

**Ví dụ**

* Khai báo int a[5] = { 1, 2, 3}; ta được mảng a với các giá trị a[0], a[1], a[2], a[3], a[4] có giá trị lần lượt là 1, 2, 3, 0, 0.
* Khai báo int a[5] = { }; ta được mảng a với tất cả các phần tử đều có giá trị 0.

Ngoài ra, chúng ta cũng có thể khai báo một mảng và gán giá trị cho nó mà không biết trước số lượng phần tử bằng cách bỏ trống phần**số\_phần\_tử**.

* Khai báo int a[] = { 1, 2, 3}; ta được mảng a - 3 phần tử với các giá trị a[0], a[1], a[2] có giá trị lần lượt là 1, 2, 3.
* Giống như việc khai báo mảng với **số\_phần\_tử**= 0, khai báo int a[] = {}; cũng sẽ báo lỗi tương tự.

1. Khởi tạo sau khi khai báo

Đôi khi chúng ta không cần thiết phải khởi tạo giá trị cho mảng ngay lập tức, chúng ta có thể làm việc này bằng cách gán thủ công từng phần tử, hoặc sử dụng vòng lặp:

1. #include <iostream>
2. using namespace std;
4. int main()
5. {
6. int a[3];
7. a[0] = 0;
8. a[1] = 2;
9. a[2] = 3;
10. }
11. Mảng cấp phát động
12. Khởi tạo

Không giống mảng tĩnh, mảng động có thể thay đổi kích thước tại thời điểm chạy.

Để cấp phát, chúng ta dùng new,

int size;

std::cin >> size;

int \*array = new int[size]; // use array new. Note that size does not need to be constant!

Nếu bạn muốn cấp phát mảng về zero, cú pháp đơn giản là:

int \*array = new int[size]();

Không có cách nào để gán mảng động về số khác zero (danh sách khởi tạo chỉ làm việc với mảng tĩnh). Ta phải dùng vòng lặp để gán từng giá trị cho phần tử.

Tuy nhiên, với C++11, bạn đã có thể dùng danh sách khởi tạo như sau:

int fixedArray[5] = { 9, 7, 5, 3, 1 }; // initialize a fixed array in C++03

int \*array = new int[5] { 9, 7, 5, 3, 1 }; // chú ý không có toán tử gán trước danh sách khởi tạo

Danh sách khởi tạo cho mảng tĩnh trong C++11 có thể bỏ qua toán tử gán:

int fixedArray[5]{ 9, 7, 5, 3, 1 }; // initialize a fixed array in C++11

1. Hủy mảng động

Chúng ta dùng lệnh delete[] để xóa mảng động:

delete[] array; // use array delete to deallocate array

1. Mảng động và mảng tĩnh

Phần tử đầu tiên trong mảng tĩnh sẽ nắm địa chỉ bắt đầu của mảng trong vùng nhớ. Mảng tĩnh có thể tham chiếu bởi con trỏ, trỏ tới địa chỉ đầu tiên của mảng. Ở dạng này, kích thước của mảng tĩnh không thể nào đoán được.

Mảng động khởi tạo từ một con trỏ, trỏ tới địa chỉ phần tử đầu tiên của mảng. Theo đó, nó không biết kích thước mảng mà nó trỏ tới. Nguyên lý về con trỏ cũng rất giống với mảng tĩnh.

1. Thay đổi kích thước

Mảng động trong C++ cho phép bạn cấp phát ngay thời gian chạy, nhưng không có cách nào để thay đổi kích thước.

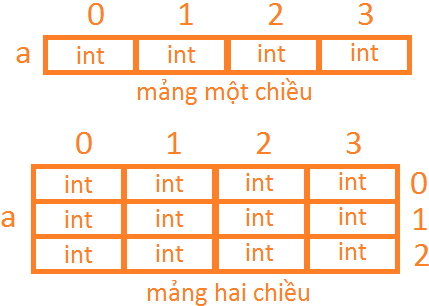
Chỉ có cách tự khai báo một mảng động mới rồi sao chép các phần tử mảng cũ sang , sau đó xóa mảng cũ. Tuy nhiên, phương pháp này không được khuyến khích, rất dễ xảy ra lỗi khi phần tử là đối tượng .

C++ cung cấp cho bạn mảng có thể co giãn kích thước qua thư viện chuẩn std::vector, bạn tham khảo [tại đây](#_Std::vector).

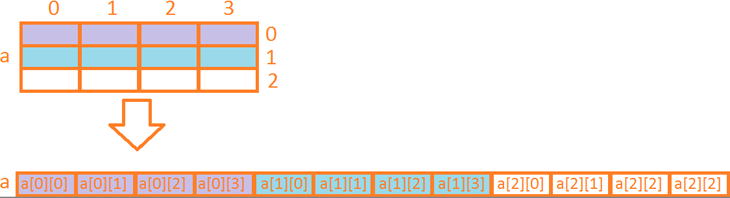
1. Mảng Nhiều Chiều
2. Mảng nhiều chiều là gì?

Như chúng ta đã biết, mảng một chiều chứa tập hợp nhiều phần tử có cùng kiểu dữ liệu với nhau, tương tự như thế, mảng nhiều chiều cũng chứa nhiều phần tử, mỗi phần tử lại là một mảng một chiều, các mảng một chiều này có kích thước bằng nhau và cũng được truy xuất thông qua một chỉ số - số thứ tự của mảng một chiều đó trong mảng nhiều chiều.

Ví dụ: Mảng 2 chiều là mảng chứa n phần tử, mỗi phần tử là một mảng một chiều. Mảng 3 chiều sẽ chứa n phần tử, mỗi phần tử là một mảng 2 chiều chứa m mảng một chiều.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_1/150/ss_1.png)

Khi lưu trữ, mảng nhiều chiều được “chuyển đổi” thành mảng một chiều và lưu trên một vùng nhớ liên tục tương tự như mảng một chiều.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_1/150/ss_2.png)

1. Khai báo mảng nhiều chiều

Trong C++, mảng nhiều chiều được khai báo như sau:

1. kiểu\_dữ\_liệu tên\_mảng[số\_phần\_tử\_chiều\_1][số\_phần\_tử\_chiều\_2]...[ số\_phần\_tử\_chiều\_n];

kiểu\_dữ\_liệu  tên\_mảng[số\_phần\_tử\_chiều\_y][ số\_phần\_tử\_chiều\_x]

Có quy tắc sử dụng tương tự như mảng một chiều là phải là hằng số.

Ví dụ:

1. // Khai báo mảng 2 chiều 3 x 4 kiểu int
2. int a[3][4];
4. // Khai báo mảng 3 chiều 3 x 4 x 5 kiểu float
5. float a[3][4][5];
6. Khởi tạo giá trị phần tử của mảng nhiều chiều

Giống như mảng một chiều, ta có thể khởi tạo giá trị cho các phần tử của mảng nhiều chiều ngay khi khai báo. Ví dụ khai báo và khởi tạo mảng hai chiều a như sau:

1. int a[3][2] = {6, 4, 5, 9, 0, 7};

Hoặc khởi tạo theo từng chiều để dễ quản lý như sau:

1. int a[3][5] =
2. {
3. {1, 2, 3, 4, 5},
4. {6, 7, 8, 9, 0},
5. {4, 9, 3, 2, 5}
6. };

Ngoài ra cũng có thể lược bỏ số\_phần\_tử đầu tiên như sau:

1. int a[][5] =
2. {
3. {1, 2, 3, 4, 5},
4. {6, 7, 8, 9, 0},
5. {4, 9, 3, 2, 5}
6. };

**LƯU Ý**

Chỉ được bỏ trống số\_phần\_tử đầu tiên, nếu không trình biên dịch sẽ không thể nào xác định được kích thước của mảng. Ví dụ khai báo mảng a như sau là không hợp lệ:

1. int a[][] =
2. {
3. {1, 2, 3, 4, 5},
4. {6, 7, 8, 9, 0},
5. {4, 9, 3, 2, 5}
6. };

Nếu khai báo như trên, mảng sẽ có tổng cộng 8 phần tử int, nhưng trình biên dịch không thể biết được mảng a là 2 x 4, 4 x 2, 1 x 8 hay 8 x 1.

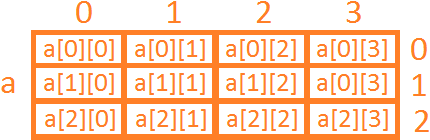
Chúng ta cũng có thể khởi tạo giá trị mảng nhiều chiều sau khi khai báo bằng cách sử dụng vòng lặp, mảng bao nhiêu chiều thì sẽ có bấy nhiêu vòng lặp lồng vào nhau:

1. int a[5][5];
3. for (i = 0; i < 5; i++)
4. for (j = 0; j < 5; j++)
5. {
6. Cout<<”Enter value #[“<<i<<”][“<<j<<”]”;
7. Cin>>a[i][j];
8. }
9. Mảng nhiều chiều

Các bạn xem ví dụ sau để nắm được cách khai báo và sử dụng mảng 2 chiều.

1. #include <stdio.h>
2. #include <string.h>
4. int main()
5. {
6. int a[3][3];
7. for(int i = 0; i < 3; i++)
8. {
9. for(int j = 0; j < 3; j++)
10. {
11. scanf("%d", &a[i][j]);
12. }
13. }
14. for(int i = 0; i < 3; i++)
15. {
16. for(int j = 0; j < 3; j++)
17. {
18. printf("%d ", a[i][j]);
19. }
20. printf("\n");
21. }
23. int b[3][3] = {{1, 3}, {2, 4, 6}};
24. for(int i = 0; i < 3; i++)
25. {
26. for(int j = 0; j < 3; j++)
27. {
28. printf("%d ", b[i][j]);
29. }
30. printf("\n");
31. }
33. return 0;
34. }
35. Truy xuất phần tử

Để truy xuất phần tử trong mảng nhiều chiều, chúng ta sử dụng lần lượt chỉ số của từng “chiều”. Ví dụ: để truy xuất đến phần tử thuộc hàng 2 cột 3 của mảng hai chiều a, ta sử dụng a[2][3] , Nên nhớ các chỉ số bắt đầu từ 0.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_1/150/ss_3.png)

1. Mảng và hàm
2. Truyền tham số mảng trong C, C++:
3. Tham số hàm kiểu con trỏ

void myFunction(int \*param)

{…

}

Call function: myFunction(arrayName);

1. Tham số hàm kiểu mảng có khai báo số phần tử

void myFunction(int param[10])

{…

}

Call function: myFunction(arrayName);

Cú pháp trên có khai báo số phần tử cho tham số, và truyền mảng thực sự vào hàm. Bất kì tác động nào lên biến tham số đều ảnh hương tới mảng

1. Tham số hàm kiểu mảng không khai báo số phần tử

void myFunction(int param[])

{…

}

Call function: myFunction(arrayName);

Ba cách khai báo trên chỉ truyền con trỏ hay địa chỉ vùng nhớ đầu tiên của mảng vào hàm.

Trên thực tế, chúng ta luôn phải khai báo tham số thứ hai là số phần tử của mảng giúp hàm biết kích thước của mảng.

Ví dụ ta có hàm **getAverage**() tính giá trị trung bình cho mảng như sau:

double getAverage(int arr[], int size){

int i, sum = 0;

double avg;

for (i = 0; i < size; ++i)

sum += arr[i];

avg = double(sum) / size;

return avg;

}

Truy xuất từ hàm main

#include <iostream>

using namespace std;

// function declaration:

double getAverage(int arr[], int size);

int main (){

// an int array with 5 elements.

int balance[5] = {1000, 2, 3, 17, 50};

double avg;

// pass pointer to the array as an argument.

avg = getAverage( balance, 5 ) ;

// output the returned value

cout << "Average value is: " << avg << endl;

return 0;

}

Kết quả

Average value is: 214.4

Độ dài của mảng không thành vấn đề đối với hàm vì C,C++ không kiểm tra ranh giới của tham số chính.

**Lưu ý:  Một số trường hợp không muốn thay đổi giá trị của mảng, ta sử dụng từ khoá const, các bạn có thể đọc thêm**[**Thao Tác Với Constant Trong C++**](http://www.stdio.vn/articles/read/47-thao-tac-voi-constant-trong-c)**.**

**Trường hợp cấp phát động, ta chỉ cần bỏ dấu “&” trước tên mảng.**

1. Hàm trả về mảng

Khai báo

int \* myFunction()

{…

}

**C,C++ sẽ hủy tất cả biến cục bộ cấp phát tĩnh trong nội hàm ra bên ngoài trong bộ nhớ Stack. Vì chủ ý người lập trình muốn trả về, bạn nên khai báo Static cho biến cục bộ. Lúc này biến sẽ được giữ lại trong vùng nhớ data. Xem thêm về** [**đặc tính lưu trữ của biến**](#_Đặc_tính_lưu)

Ví dụ:

#include <stdio.h>

/\* function to generate and return random numbers \*/

int \* getRandom( )

{

static int r[10];

int i;

/\* set the seed \*/

srand( (unsigned)time( NULL ) );

for ( i = 0; i < 10; ++i)

{

r[i] = rand();

printf( "r[%d] = %d\n", i, r[i]);

}

return r;

}

/\* main function to call above defined function \*/

int main ()

{

/\* a pointer to an int \*/

int \*p;

int i;

p = getRandom();

for ( i = 0; i < 10; i++ )

printf( "\*(p + %d) : %d\n", i, \*(p + i));

return 0;

}

1. Duyệt mảng với con trỏ

Ví dụ

double \*p;

double balance[10];

p = balance;

Tên mảng được xem hợp lệ là con trỏ hằng. Do đó, \*(**balance** + 4) sẽ truy xuất tới **balance**[4].

Khi bạn gán địa chỉ mảng **balance** cho con trỏ p, bạn có thể truy xuất phần tử thông qua \*p, \*(p+1), \*(p+2)….

Ví dụ:

#include <stdio.h>

int main (){

/\* an array with 5 elements \*/

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

double \*p;

int i;

p = balance;

/\* output each array element's value \*/

printf( "Array values using pointer\n");

for ( i = 0; i < 5; i++ )

printf("\*(p + %d) : %f\n", i, \*(p + i) );

printf( "Array values using balance as address\n");

for ( i = 0; i < 5; i++ )

printf("\*(balance + %d) : %f\n", i, \*(balance + i) );

return 0;

}

Kết quả

Array values using pointer

\*(p + 0) : 1000.000000

\*(p + 1) : 2.000000

\*(p + 2) : 3.400000

\*(p + 3) : 17.000000

\*(p + 4) : 50.000000

Array values using balance as address

\*(balance + 0) : 1000.000000

\*(balance + 1) : 2.000000

\*(balance + 2) : 3.400000

\*(balance + 3) : 17.000000

\*(balance + 4) : 50.000000

**Lưu ý:**

**Phương thức truy xuất mảng hay chuỗi kí tự đặc biệt hiệu quả khi duyệt với con trỏ. Dùng các phép toán trên con trỏ để di chuyển tới vị trí phần tử xác định. Chúng ta nên tích cực tận dụng ưu điểm này trong lập trình C,C++.**

### Tham chiếu

Một biến tham chiếu là một biến ₫ại diện trực tiếp cho một biến khác (thay cho con trỏ)

Ý nghĩa sử dụng chủ yếu về sau trong truyền tham số cho hàm

Tham chiếu trong C,C++ là một khái niệm không mới xong không phải bất cứ lập trình viên nào cũng có nhìn nhận đầy đủ về nó. Vì vậy, bài viết này nhằm giới thiệu cho các bạn về biến tham chiếu trong C,C++.

##### So sánh tham chiếu và con trỏ

Mặc dù, giữa con trỏ và tham chiếu có điểm tương đồng, tuy nhiên, giữa chúng cũng có một số điểm khác biệt cơ bản về mặt ý nghĩa như sau:

* Một con trỏ có thể nhận giá trị null trong khi một tham chiếu thì không thể.
* Một con trỏ có thể khởi tạo bất cứ lúc nào, trong khi một tham chiếu **phải được khởi tạo ngay** khi nó được khai báo. Nói cách khác, câu lệnh sau trình biên dịch sẽ báo lỗi:

int &k;    // error

##### Định nghĩa và cú pháp khai báo

Tham chiếu nói một cách đơn giản là một bí danh của một biến. Nói cách khác, tham chiếu là một tên gọi khác của một biến đã có sẵn. Điều đó có nghĩa là, có thể thao tác trên biến đó thông qua tên biến hoặc tham chiếu.

Một biến tham chiếu đại diện cho duy nhất một biến dữ liệu. .Ta không thể có thể gán tham chiếu sang một biến khác, nhưng được phép thực hiện mọi phép toán trên biến tham chiếu.

Cú pháp:

**<Tên kiểu tham chiếu>& <Tên biến>**

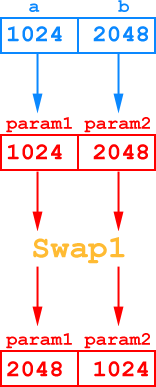
1. void main()
2. {
3. double d = 2.0;
4. double& r = d; // biến tham chiếu r đại diện cho d.
5. double \*p1 = &d, \*p2 = &r; // p1 và p2 đều mang địa chỉ của d.
6. r = 1.0; // OK, d = 1.0
7. double d2 = 0;
8. r = d2; // r = 0, d=0
9. r = 1.0; // r = d = 1, d2 =0
10. }

##### Tham chiếu là tham số hàm C++

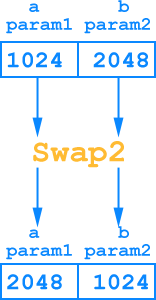
Để hiểu rõ lý do của việc sử dụng tham chiếu, các bạn hãy xem một ví dụ đơn giản sau:

1. #include <iostream>
2. using namespace std;
4. void Swap1(int param1, int param2)
5. {
6. int temp = param1;
7. param1 = param2;
8. param2 = temp;
9. }
11. void Swap2(int& param1, int& param2)
12. {
13. int temp = param1;
14. param1 = param2;
15. param2 = temp;
16. }
18. int main()
19. {
20. int a = 1024;
21. int b = 2048;
23. cout << "a = " << a << ", b = " << b << endl;
25. cout << "Swap1: ";
26. Swap1(a, b);
27. cout << "a = " << a << ", b = " << b << endl;
29. cout << "Swap2: ";
30. Swap2(a, b);
31. cout << "a = " << a << ", b = " << b << endl;
33. return 0;
34. }
36. // Difference between value parameter and reference parameter
37. // a = 1024, b = 2048
38. // Swap1: a = 1024, b = 2048
39. // Swap2: a = 2048, b = 1024

Bạn có thắc mắc câu chuyện đằng sau 2 hàm trên không? Tôi sẽ giải thích bằng hình ảnh sau đây:



Truyền tham trị: param1 và param2 chỉ là 2 bản sao chép của a và b

  
Truyền tham chiếu: param1 và param2 cũng chính là a và b

Hàm Swap1 tôi sử dụng cách truyền tham trị. Trong khi, hàm Swap2 tôi sử dụng cách truyền tham chiếu. Và kết quả, như các bạn thấy ở trên. Lý do tại sao vậy? Thông thường, chúng ta hay sử dụng tham trị nhiều và “bị” quen thuộc với nó. Tuy nhiên, như bạn thấy đó, rõ ràng mong muốn của chúng ta khi viết hàm trên là có thể đổi chỗ được 2 số 1024 và 2048. Nhưng với cách truyền tham trị, mong muốn đó không thực hiện được, trong khi, sử dụng tham chiếu hoàn toàn có thể làm được điều này.

Rõ ràng, việc truyền tham trị trong trường hợp này, không những không đáp ứng được mong muốn ban đầu, mà còn gây thêm chi phí cho việc tạo ra 1 bản copy của tham số và thao tác trên đó mà không thu được kết quả gì. Vì thế, trong những trường hợp cần thay đổi giá trị của biến, bạn hãy nghĩ tới việc sử dụng tham chiếu.

##### Hàm trả về tham chiếu

Chương trình sẽ dễ đọc và bảo trì hơn khi dùng tham chiếu thay vì con trỏ. Hàm trong C++ có thể dùng cả tham chiếu và con trỏ cho đối số cũng như giá trị return.

Khi khai báo hàm tham chiếu, nó ngầm trả về con trỏ. Hàm tham chiếu là kiểu Lvalue, nên nó được phép nằm bên trái phép gán. Bạn được quyền thay đổi giá trị cho nó

Ví dụ:

#include <iostream>

using namespace std;

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0}; // Biến toàn cục

double& setValues( int i ) {

return vals[i]; // trả về tham chiếu phần tử thứ i

}

// main function to call above defined function.

int main (){

cout << "Value before change" << endl;

for ( int i = 0; i < 5; i++ ) {

cout << "vals[" << i << "] = ";

cout << vals[i] << endl;

}

setValues(1) = 20.23; // change 2nd element

setValues(3) = 70.8; // change 4th element

cout << "Value after change" << endl;

for ( int i = 0; i < 5; i++ ) {

cout << "vals[" << i << "] = ";

cout << vals[i] << endl;

}

return 0;

}

Khi trả về 1 tham chiếu, phải chắc chắn đối tượng tham chiếu tới không nằm ngoài phạm vi truy xuất. Trả về tham chiếu của một biến cục bộ là không hợp lệ. Giải pháp là trả về tham chiếu biến **static** hay **global**. Vì biến static luôn được giữ trong vùng nhớ data segment.

int& func() {

int q;

//!If return q, Compile time error

static int x;

return x; // Safe, x lives outside this scope

}

##### Lời kết

Hi vọng qua bài viết này, sẽ cung cấp thêm cho các bạn mảng kiến thức về tham chiếu. Từ đó, có thể vận dụng linh hoạt và hiệu quả tham chiếu trong công việc. Do giữa tham chiếu và con trỏ có mối liên hệ mật thiết với nhau và trong phạm vi bài viết tôi không đề cập hết được mọi vấn đề một cách chi tiết, vì thế, các bạn hãy tham khảo các bài viết khác về con trỏ để hiểu sâu hơn về hai loại biến này.

### Constant Trong C,C++

**Const** là một từ khóa quen thuộc trong nhiều ngôn ngữ lập trình, nhưng để hiểu và sử dụng hiệu quả nó thì không phải ai cũng có thể.

##### Mở đầu

const (Constant – Hằng) là từ khóa để chỉ định một biến hay đối tượng nào đó là hằng, tức là không thay đổi được giá trị, mọi hành động thay đổi giá trị của hằng đều được compiler báo lỗi.

Sử dụng biến hằng khi bạn muốn giá trị của biến, đối tượng đó không bị thay đổi trong mọi trường hợp, ví dụ như số pi luôn bằng 3.14 chẳng hạn.

##### Cơ bản

Biến được khai báo với **const** không được phép đổi giá trị:

1. int main()
2. {
3. const int x = 10;
4. int const y = 50;
6. x = 5; // Báo lỗi
7. y = 7; // Báo lỗi
9. return 0;
10. }

Ta thấy là vị trí đặt **const** dù trước hay sau int thì đều có chung một chức năng là chỉ định x, y là hằng.

1. Const với biến tham chiếu trong C++

Biến tham chiếu là biến có thể thay đổi giá trị của vùng nhớ mà nó tham chiếu tới một cách trực tiếp. Trong trường hợp biến tham chiếu là hằng thì sao? Ta xét ví dụ sau:

1. int main()
2. {
3. int x = 10;
4. const int &A = x; // Khai báo A là hằng tham chiếu tới vùng nhớ của x
6. A = 6; // Báo lỗi do A là biến tham chiếu hằng
7. x = 6; // Không báo lỗi do x là một biến bình thường
9. return 0;
10. }

Vậy là ta đã thấy hằng tham chiếu A sẽ không thể thay đổi giá trị của **x** được nữa. Ta cũng có thể khai báo **int const &A = x;**

1. Const với biến con trỏ

Biến con trỏ là biến lưu địa chỉ vùng nhớ mà nó trỏ tới. Vậy thì biến con trỏ mà có thêm từ khóa const thì chuyện gì sẽ xảy ra? Ta xét ví dụ sau:

1. int main()
2. {
3. int x = 7;
4. int y = 13;
6. const int \*p = &x; // Đọc là Non-const pointer to a const value
8. \*p = 100; // Báo lỗi
9. p = &y; // Không báo lỗi
11. return 0;
12. }

Trong trường hợp trên ta dễ thấy được rằng không thể thay đổi giá trị vùng nhớ mà p chỉ tới thông qua p,  nhưng vẫn có thể thay đổi được địa chỉ vùng nhớ mà p trỏ tới.

Ta xét ví dụ tiếp theo:

1. int main()
2. {
3. int x = 7;
4. int y = 13;
6. int \*const p = &x; // Đọc là Const pointer to a non-const value
8. \*p = 100; // Không báo lỗi
9. p = &y; // Báo lỗi
11. return 0;
12. }

Lần này ta nhận thấy có sự ngược lại với ví dụ trước đó, p đã có thể thay đổi giá trị vùng nhớ mà nó trỏ tới, nhưng không thể thay đổi địa chỉ mà nó trỏ tới được nữa.

Nếu muốn con trỏ p vừa không thể thay đổi địa chỉ của vùng nhớ, vừa không thể thay đổi giá trị vùng nhớ đó thì sao? Rất đơn giản, ta chỉ cần khai báo:

**const int \*const p = &x;    // Đọc là Constant Pointer to a const value**

1. Tham số hàm là const (C++)

Khi muốn sử dụng tham chiếu để vừa tiết kiệm chi phí, vừa ngăn được tình trạng thay đổi giá trị thì ta sẽ dùng tới tham số const reference. Cách dùng như sau:

1. int func(const int &number)
2. {
3. //Your Codes
4. }

Tượng tự với tham số hàm con trỏ

1. int func(const int\* number)
2. {
3. //Your Codes
4. }
5. Hàm hằng (C++)

Khái niệm hàm hằng chỉ áp dụng trong hướng đối tượng

##### Hàm hằng trong hướng đối tượng

Ta sử dụng lớp sau làm tiền đề để giải thích:

1. class MyClass
2. {
3. private:
4. int m\_Data;
6. public:
7. MyClass(int Data) : m\_Data(Data) {}
9. // Message() Function for non-const object
10. void Message()
11. {
12. cout << "This is non-const object" << endl;
13. }
15. // Message() Function for const object
16. void Message() const
17. {
18. cout << "This is const object" << endl;
19. }
21. // Get value of m\_Data to use or modify
22. int &getData()
23. {
24. return m\_Data;
25. }
26. };

void Message() const

Phương thức **Message**() thứ 2 có khai báo **const** ở cuối. Điều này có nghĩa **Message**() đó là một phương thức hằng thành viên.

Khi một phương thức là phương thức hằng, thì tất cả các thuộc tính (biến thành viên) của lớp đó sẽ không được thay đổi giá trị trong quá trình thực thi hàm, mọi hành động thay đổi giá trị của thuộc tính trong hàm compiler đều báo lỗi, trừ khi thuộc tính đó được khai báo với từ khóa **mutable** đằng trước.

Các đối tượng hằng chỉ được sử dụng được các phương thức hằng mà thôi.

Ta có hàm main như sau:

1. int main()
2. {
3. MyClass obj1(13);
4. const MyClass obj2(100);
6. obj1.getData() = 86;
7. //obj2.getData() = 1; //báo lỗi do getData() không phải phương thức hằng
9. obj1.Message();
10. obj2.Message();
12. return 0;
13. }

Khi chương trình được thực thi, kết quả in ra màn hình như sau:

This is non-const object  
This is const object

Như vậy đối tượng **obj1** đã gọi phương thức **Message**() thứ nhất, còn **obj2**  gọi phương thức **Message**() thứ 2. Lý do đơn giản: **obj1** là đối tượng bình thường, còn **obj2** là đối tượng hằng nên chỉ có thể gọi hàm **Message**() thứ 2.

Lưu ý: Phương thức hằng có thể luôn được gọi. Tức là **obj1** vẫn có thể gọi được phương thức **Message**() thứ 2, khi phương thức **Message**() thứ 1 bị xóa đi, thì kết quả in ra màn hình sẽ là:

This is const object  
This is const object

Một vấn đề nữa, ta thấy **obj2** không thể dùng phương thức **getData**().

Vậy nếu muốn dùng thì sao? Theo như nãy giờ đã nói thì chỉ cần thêm **const** vào sau **getData**() là xong.

Nhưng do **getData**() là hàm tham chiếu, thuộc loại **Lvalue** , nên nếu chỉ thêm **const** vào sau thì nhất định sẽ có lỗi.

Ta hãy overload lại 1 phương thức **getData**() như sau:

1. const int &getData() const
2. {
3. return m\_Data;
4. }

Như vậy ta có thể dùng obj2 để gọi **getData**() được, nhưng do trả về hằng tham chiếu nên không thể dùng getData() để thay đổi giá trị của m\_Data giống như đối với obj1 được.

##### Ép kiểu hằng với const\_cast

Hàm **func**() sẽ tăng giá trị của đối số thêm **100**, đồng thời trả về giá trị đó luôn:

1. int func(int &number)
2. {
3. number += 100;
4. return number;
5. }

Nhưng trong trường hợp ta muốn truyền biến **const**, do biến đó là **const** nên khi truyền vào compiler sẽ báo lỗi, vậy ta phải làm thế nào mà không cần phải overload lại hàm đó?

Lúc đó ta sẽ dùng tới toán tử **const**\_**cast**. Cách dùng như sau:

1. int main()
2. {
3. const int x = 13;
4. int y;
6. //y = func(x); // Báo lỗi
8. y = func(const\_cast<int &>(x)); // Dùng const\_cast
10. printf("y = %d\n", y);
11. printf("x = %d\n", x);
13. return 0;
14. }

Kết quả xuất ra màn hình như sau:

y = 113;  
x = 13;

Thật tuyệt vời! Như vậy là ta vẫn có giá trị của y như mong muốn và giá trị của **x** vẫn giữ nguyên, ta đã có thể sử dụng hàm **func**() mà không lo compiler báo lỗi nữa.

##### Sự khác biệt Const giữa C và C++

Khai báo const ở C: const int i = 2;

Khai báo ở module khác extern const int i;

Khai báo const ở C++: extern const int i = 2;

Khai báo const ở 1 file C tới 1 file C++: extern "C" const int x=10; để tránh hư hại khi biên dịch

##### Tham chiếu và hằng

1. Tham chiếu tới giá trị hằng

Chúng ta có thể khai báo con trỏ tới giá trị hằng, biến tham chiếu cũng có thể làm được khi dùng với từ khóa **const**.

const int value = 5;

const int &ref = value; // ref là tham chiếu tới giá trị hằng

Tham chiếu tới hằng thường được gọi là tham chiếu hằng.

1. Khởi tạo tham chiếu hằng

Khác với tham chiếu tới giá trị non-const, vốn chỉ có thể khởi tạo với giá trị non-const Lvalue, tham chiếu hằng có thể khởi tạo với

* non-const Lvalue
* const Lvalue
* Rvalue

int x = 5;

const int &ref1 = x; // ok, x là non-const l-value

const int y = 7;

const int &ref2 = y; // ok, y là const l-value

const int &ref3 = 6; // ok, 6 là r-value

Giống như con trỏ trỏ tới giá trị hằng, tham chiếu hằng có thể tham chiếu tới biến non-const. Khi truy xuất giá trị qua tham chiếu hằng, giá trị được xem là hằng dù giá trị gốc không phải hằng

int value = 5;

const int &ref = value; // tạo tham chiếu hằng tới biến non-const value

value = 6; // ok, value là non-const

ref = 7; // không hợp lệ -- ref là const

1. Tham chiếu tới r-value mở rộng vòng đời cho biến được tham chiếu

Bình thường r-value chỉ tồn tại trong phạm vi sử dụng.

std::cout << 2 + 3; // 2 + 3 là r-value có giá trị bằng 5, sẽ bị hủy cuối dòng lệnh này

Tuy nhiên, khi tham chiếu hằng được khởi tạo với r-value, vòng đời của r-value được mở rộng tương ứng với vòng đời của tham chiếu

#include <iostream>

int somefcn()

{

const int &ref = 2 + 3; // giá trị r-value liên kết với tham chiếu hằng

std::cout << ref; // vì thế có thể dùng cho câu lệnh này

} // vòng đời r-value được mở rộng cho tới khi tham chiếu ra khỏi phạm vi hàm và bị hủy

### Cấu trúc điều khiển:

##### Nhóm lệnh phân nhánh:

1. Cấu trúc Switch … case

Switch so sánh một biểu thức nguyên với một danh sách giá trị các số nguyên, các hằng kí tự hoặc biểu thức hằng. Mỗi giá trị trong danh sách chính là một case label (nhãn trường hợp) trong khối codes của switch. Ngoài ra, trong khối codes switch còn có thể có một default label (nhãn mặc định) có thể có hoặc không. Mặt khác, trong mỗi label còn chứa các khối codes chờ được thực thi.

1. Cú pháp của cấu trúc switch…case

Muốn sử dụng cấu trúc switch…case, bạn dùng cú pháp như sau:

1. switch (expression)
2. {
3. case constant\_1:
4. {
5. Statements;
6. break;
7. }
8. case constant\_2:
9. {
10. Statements;
11. break;
12. }
13. .
14. .
15. .
16. case constant\_n:
17. {
18. Statements;
19. break;
20. }
21. default:
22. {
23. Statements;
24. }
25. }

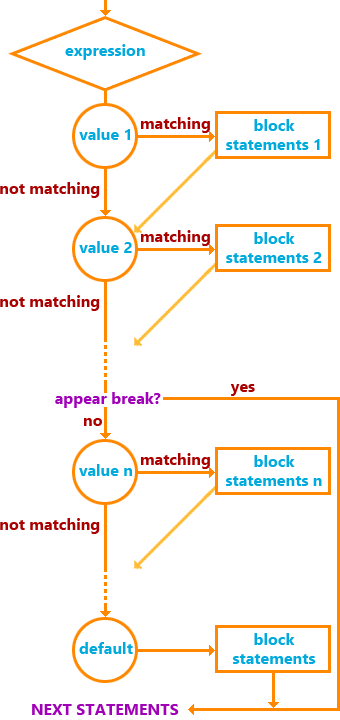
Ví dụ:

1. #include <iostream.h>
2. using namespace std;
4. int main()
5. {
6. int month;
7. cout << "Month: " << endl;
8. cin >> month;
10. switch (month)
11. {
12. case 1:
13. {
14. cout << "January" << endl;
15. break;
16. }
17. case 2:
18. {
19. cout << "February" << endl;
20. break;
21. }
22. case 3:
23. {
24. cout << "March" << endl;
25. break;
26. }
27. case 4:
28. {
29. cout << "April" << endl;
30. break;
31. }
32. case 5:
33. {
34. cout << "May" << endl;
35. break;
36. }
37. case 6:
38. {
39. cout << "June" << endl;
40. break;
41. }
42. case 7:
43. {
44. cout << "July" <<endl;
45. break;
46. }
47. case 8:
48. {
49. cout << "August" << endl;
50. break;
51. }
52. case 9:
53. {
54. cout << "September" << endl;
55. break;
56. }
57. case 10:
58. {
59. cout << "October" << endl;
60. break;
61. }
62. case 11:
63. {
64. cout << "November" << endl;
65. break;
66. }
67. case 12:
68. {
69. cout << "December" << endl;
70. break;
71. }
72. default:
73. {
74. cout << "Input is false" << endl;
75. }
76. }
78. return 0;
79. }
80. Nguyên tắc hoạt động cấu trúc switch…case

Biểu thức nguyên trong switch được tính toán và kiểm tra lần lượt với giá trị của từng case label. Đầu tiên, nó sẽ được so sánh với giá trị của case label đầu tiên. Nếu bằng nhau thì sẽ thực hiện các statement trong case label này cho đến khi nó gặp được từ khoá break. Khi đó, cấu trúc switch…case kết thúc.

Chương trình sẽ thực hiện tiếp những dòng code sau cấu trúc switch…case. Ngược lại, nếu như giá trị biểu thức nguyên không bằng giá trị case label đầu tiên thì nó sẽ tiếp tục so sánh đến giá trị của case label thứ hai và tiếp tục thực hiện như những bước trên. Giả sử, đến cuối cùng vẫn không tìm được giá trị bằng nó thì các khối codes trong default label sẽ được thực hiện nếu như có tồn tại default label.

Ta có sơ đồ hoạt động của cấu trúc switch…case như sau:



1. Một số lưu ý khi dùng cấu trúc switch…case

* Các giá trị của mỗi case label phải cùng kiểu dữ liệu với giá trị của biểu thức đem đi so sánh.
* Số lượng các case label là không giới hạn nhưng chỉ có thể có duy nhất một default label.
* Giá trị của các case label là một hằng số và các giá trị của các case label phải khác nhau.
* Từ khóa break có thể sử dụng hoặc không. Nếu không được sử dụng thì chương trình sẽ không kết thúc cấu trúc switch…case khi đã thực hiện hết khối codes của case label có giá trị bằng với biểu thức nguyên. Thay vào đó, nó sẽ thực hiện tiếp các khối codes tiếp theo cho đến khi gặp từ khoá break hoặc dấu “ } “ cuối cùng của cấu trúc switch…case. Vì vậy, các bạn có thể sử dụng một khối codes cho nhiều trường hợp khác nhau.

Ví dụ:

1. #include <iostream.h>
2. using namespace std;
4. int main()
5. {
6. int month, day;
7. cout << "Month: " << endl;
8. cin >> month;
10. switch (month)
11. {
12. case 1:
13. case 3:
14. case 5:
15. case 7:
16. case 8:
17. case 10:
18. case 12:
19. {
20. day = 30;
21. break;
22. }
23. case 4:
24. case 6:
25. case 9:
26. case 11:
27. {
28. day = 31;
29. break;
30. }
31. default:
32. {
33. day = 28;
34. }
35. }
37. return 0;
38. }
39. Cấu trúc if .. else

Như chính tên gọi của nó. Cấu trúc này có hai thành phần. Thành phần đầu tiên là từ khoá if chứa một biểu thức, tiếp đó là khối lệnh đang chờ được thực thi của nó.

Thành phần thứ hai là từ khoá else cũng kèm theo khối lệnh đang chờ thực thi. Vì cấu trúc rất rõ ràng mạch lạc như đã nêu nên bên cạnh cấu trúc if…else còn có các cấu trúc khác như if, if lồng if, else if,….

Tất cả các cấu trúc này đều phục vụ với mục đích chung đó là giúp bạn kiểm soát được chương trình của chính mình. Vì những dòng mã được thực thi hay không là dựa vào điều kiện đúng hay sai mà kết quả của biểu thức trong if trả về. Do đó, giúp cho lập trình viên dễ dàng quản lí chương trình của mình hơn.

1. Cú pháp cấu trúc if…else

Muốn sử dụng cấu trúc if…else, bạn chỉ cần nhớ cú pháp sau:

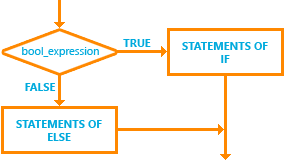
1. if (boolean\_expression)
2. {
3. statements;
4. }
5. else
6. {
7. statements;
8. }

Ví dụ:

1. void foo()
2. {
3. int a, b, max;
4. a = 5;
5. b = 10;
7. if ( a > b) max = a;
8. else max = b;
10. }
11. Nguyên tắc hoạt động của cấu trúc if…else

Biểu thức nguyên trong từ khoá if sẽ được tính toán sau đó trả về giá trị boolean. Nếu giá trị trả về bằng true thì những statement thuộc khối lệnh của từ khoá if sẽ được thực thi. Ngược lại, nếu biểu thức trả về giá trị là false thì những statement thuộc khối lệnh của từ khoá else sẽ được thực thi.

Ta có thể biểu diễn quá trình này bằng sơ đồ sau:



1. Lời kết

Kiến thức trên tuy là đơn giản nhưng là nền tảng để các bạn có thể giải quyết nhiều vấn đề trong lập trình. Vì thế, mong rằng bài viết này giúp bạn hiểu rõ hơn về cấu trúc if…else. Từ đó, vận dụng chúng một cách dễ dàng và linh hoạt hơn.

1. Từ khóa Goto
2. Cú pháp

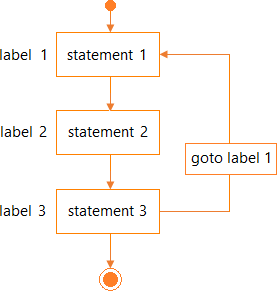
Thao tác với goto khá đơn giản. Cú pháp như sau:

1. goto label\_name;
2. // Several code line
3. label\_name: statement;

Khi gặp từ khoá goto, chương trình sẽ nhảy đến biểu thức gắn nhãn được khai báo. Biểu thức này và dòng code goto bắt buộc phải nằm trong cùng một scope (phạm vi), nếu không chương trình sẽ báo lỗi (error C0294).

Biểu thức được gắn nhãn có thể là bất cứ biểu thức nào, theo sau một định danh và dấu hai chấm.

1. Sơ đồ hoạt động



1. Một số lưu ý khi sử dụng goto

Do đặc tính nhảy vô điều kiện đến bất kỳ vị trí nào trong hàm nên dễ dẫn đến lỗi chương trình như chưa khai báo hoặc khởi tạo biến. Do đó lập trình viên cần hết sức thận trọng khi quyết định sử dụng goto.

Ngoài ra, sử dụng goto cũng có thể dẫn đến trường hợp lặp vô hạn. Ví dụ sau đây sẽ minh hoạ rõ hơn:

1. int main()
2. {
3. label1: // Some statement
4. goto label2;
6. // Some statement
8. label2: // Some statement
9. goto label1;
11. return 0;
12. }

Với những chương trình nhỏ, lỗi này dường như có thể được phát hiện dễ dàng. Nhưng các bạn hãy tưởng tượng một dự án game cỡ vừa, có hàng chục file mã nguồn, cùng với hàng trăm hàm, biến hỗ trợ. Việc phát hiện ra lỗi này sẽ không hề đơn giản nếu như chúng ta không cẩn thận ngay từ đầu.

Đa số các trường hợp sử dụng goto đều có các giải pháp thay thế an toàn hơn. Do đó nếu không bắt buộc, các bạn nên chọn những giải pháp khác để tránh tối đa rủi ro khi sử dụng.

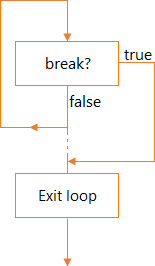
Tuy nhiên, nếu goto được sử dụng hợp lý, chương trình của các bạn sẽ rất trực quan và gọn gàng. Dưới đây là một ví dụ về việc sử dụng hợp lý goto:

1. // Neccessary files and function
2. void main()
3. {
4. if(!openDataFile())
5. goto quit;
7. if(!getDataFromFile())
8. goto closeFileAndQuit;
10. if(!allocateSomeResources())
11. goto releaseResourcesAndQuit;
13. // Do more work here....
15. releaseResourcesAndQuit:
16. // Release resources
17. closeFileAndQuit :
18. // Close file
19. quit :
20. // Quit!
21. }
22. Từ khóa break và continue
23. Từ khóa break

Từ khoá break thường được sử dụng để dừng vòng lặp (for, while, do … while, …) mà không quan tâm đến biểu thức điều kiện trong vòng lặp. Khi gặp break, chương trình sẽ thoát khỏi vòng lặp trong cùng (nếu có nhiều vòng lặp lồng nhau) và chuyển đến dòng code kế tiếp sau vòng lặp đó.

Thông thường, break được sử dụng với điều kiện trong if … else. Ngoài ra từ khoá break còn được dùng trong cấu trúc switch … case (Đọc thêm bài viết [Cấu Trúc Switch ... Case Trong C++](#_Cấu_trúc_Switch)).

1. Sơ đồ hoạt động của break

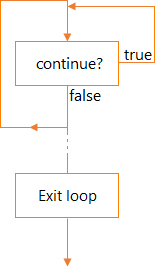


1. Từ khóa continue

Từ khoá continue có tác dụng bỏ qua phần statement còn lại của lần lặp hiện tại và chuyển sang lần lặp kế tiếp của vòng lặp.

Giống như break, continue thường được sử dụng kèm với điều kiện trong biểu thức if … else.

1. Sơ đồ hoạt động của continue



1. Chương trình minh hoạ

Chương trình sau sẽ giúp các bạn hiểu rõ hơn về cách thức hoạt động và tính ứng dụng của break và continue, trong lập trình C++ nói riêng và trong lập trình nói chung.

1. #include <iostream>
2. using namespace std;
4. int main()
5. {
6. int numberArray[10] = {5, 3, 8, 1, 12, 6, 7, 15, 2, 4};
8. int number;
9. cout << "Enter a number: ";
10. cin >> number;
12. for(int i = 0;; i++)
13. {
14. //Stop condition
15. if(i == 10)
16. {
17. cout << "Number not found" << endl;
18. break;
19. }
21. if(number != numberArray[i])
22. continue;
24. //Print the position and break when found number in array
25. cout << "Found at position " << i << endl;
26. break;
27. }
29. fflush(stdin);
30. cin.get();
31. return 0;
32. }

Trong ví dụ trên có sử dụng mảng một chiều số nguyên.

1. Nhóm lệnh vòng lặp:

Vòng lặp có thể thực hiện với **if..else + goto,** song không bao giờ nên như vậy

 Ứng dụng vòng lặp chủ yếu khi làm việc với mảng và các cấu trúc dữ liệu tổng quát khác => truy nhập qua biến mảng + chỉ số, qua con trỏ hoặc qua **iterator** (sẽ ₫ề cập sau này)

##### Nhóm lệnh vòng lặp

1. Cấu trúc while

Vòng lặp while có cấu trúc như sau:

1. while (condition)
2. {
3. // Code while condition is true
4. }

Các biểu thức Initialization statement và Update statement không xuất hiện trong cấu trúc while. Do đó ta cần hiện thực chúng trước và trong vòng lặp để chương trình không bị lặp vô hạn.

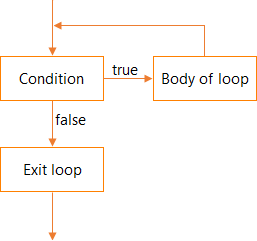
Đoạn code trong thân vòng lặp (body of loop) có thể chứa các biểu thức tính toán logic, các cấu trúc điều kiện (if … else, switch … case) hoặc cả các cấu trúc lặp khác. Nhờ đó ta có thể xây dựng được các chu trình lặp lồng nhau (sử dụng để thao tác với mảng nhiều chiều, …).

1. Nguyên tắc hoạt động

Giống như cấu trúc for, quá trình lặp trong while sẽ diễn ra liên tục cho đến khi biểu thức Condition trả về giá trị false.

Một điều cần lưu ý là biểu thức Condition tồn tại trước khi quá trình lặp bắt đầu. Do đó vòng lặp có thể sẽ không được thực hiện lần nào, tuỳ thuộc vào việc khởi tạo giá trị cho các biến có liên quan.

Sơ đồ hoạt động của cấu trúc while như sau:



Vòng lặp while thường được sử dụng trong các trường hợp không biết rõ số lần lặp. Ví dụ sau sẽ giúp các bạn dễ dàng hình dung cách làm việc của vòng lặp while:

1. #include <iostream>
2. using namespace std;
4. int main()
5. {
6. int s = 0;
7. int i = 0;
9. while(s < 100)
10. {
11. i++;
12. s += i;
13. }
15. s = 0;
16. i = 0;
18. while(i++, s < 100)
19. s += i;
21. s = 0;
22. i = 0;
24. while(1)
25. {
26. i++;
27. s += i;
29. if(s >= 100)
30. break;
31. }
33. return 0;
34. }

Ba đoạn code trên đây thực hiện chính xác một công việc giống nhau (chỉ quan tâm đến kết quả, không quan tâm đến giá trị của các biến trong suốt quá trình). Bạn có thể lồng ghép nhiều biểu thức vào Condition, phân cách với nhau bởi dấu “,” như ở đoạn code thứ hai. Khi đó biểu thức sau dấu “,” cuối cùng chính là điều kiện chính để dừng vòng lặp.

1. Cấu trúc do…while

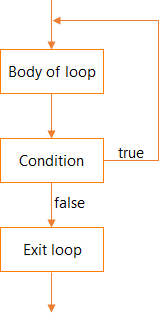
Vòng lặp do … while có cấu trúc như sau:

1. do
2. {
3. // Code
4. } while (Condition);

do…while có đầy đủ các tính chất mà một vòng lặp cần phải có. Tuy nhiên, khác với cấu trúc while, biểu thức điều kiện của do … while được đặt phía sau đoạn code, do đó vòng lặp sẽ được thực hiện ít nhất một lần, bất kể các giá trị đầu vào có thoả biểu thức điều kiện hay không.

1. Nguyên tắc hoạt động

Sơ đồ hoạt động của cấu trúc do…while như sau:



Cấu trúc do … while thường được sử dụng để kiểm tra các giá trị đầu vào (thường là các giá trị được nhập từ bàn phím).

Chương trình sau kiểm tra dữ liệu nhập vào có thể là một tháng trong năm hay không:

1. int main()
2. {
3. int month;
4. do
5. {
6. cin >> month;
7. } while (month < 1 || month > 12);
9. return 0;
10. }

Ngoài ra ta có thể tận dụng do…while để làm một số công việc "thú vị", chẳng hạn như sau:

1. #include <iostream>
2. using namespace std;
4. int main()
5. {
6. int n;
8. do
9. {
10. cin >> n;
11. } while(n <= 0);
13. int i;
14. int s = 0;
16. do
17. {
18. cin >> i;
19. } while(s += i, --n);
21. return 0;
22. }

Chương trình trên thực hiện việc tính tổng n số nguyên nhập vào từ bàn phím. Ta không cần tạo mảng để lưu trữ các số mà thực hiện cộng trực tiếp sau từng lần nhập.

1. Một số lưu ý

Ta có break và continue cũng có thể được sử dụng trong hai cấu trúc lặp này, break sẽ thoát khỏi vòng lặp ở mức thấp nhất (vòng lặp trong cùng nếu có nhiều vòng lặp lồng nhau).

Tương tự như trong cấu trúc for, sau dòng lệnh while không có dấu ";".

1. Cấu trúc for
2. Cấu trúc for trong C/C++ có prototype như sau:
3. for(Initialization statement; Condition; Update statement)
4. {
5. //Code while condition is true
6. }

Trong đó:

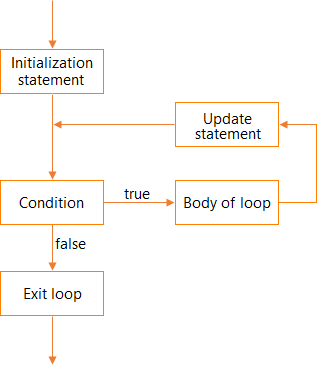
* Initialization statement là biểu thức khởi tạo các giá trị ban đầu cho các tham số. Các tham số này dùng để kiểm tra điều kiện trong biểu thức Condition.
* Condition là biểu thức điều kiện của vòng lặp. Vòng lặp sẽ kết thúc khi biểu thức này trả về giá trị false.
* Update statement là biểu thức cập nhật giá trị cho các tham số.

Ngoài ra, nếu phần code trong vòng lặp for chỉ có 1 dòng, ta có thể bỏ cặp dấu { }.

1. Nguyên tắc hoạt động

Bắt đầu với việc khởi tạo giá trị ban đầu cho các biến, quá trình lặp sẽ diễn ra liên tục, cho tới khi biểu thức Condition không còn đúng nữa. Các biến sẽ được cập nhật trong phần Update statement hoặc cả ở trong vòng lặp (nếu có).

Sơ đồ hoạt động của cấu trúc for như sau:



1. Một số lưu ý

Vòng lặp sẽ kết thúc khi và chỉ khi biểu thức điều kiện trả về giá trị false, do đó các bạn cần lưu ý biểu thức Update statement để thu được kết quả mong muốn.

Sau dòng lệnh for không có dấu “;”, nếu có chương trình sẽ báo lỗi hoặc cho kết quả không mong muốn.

Các biến nếu được khai báo trong dòng lệnh for sẽ chỉ có phạm vi sử dụng trong vòng lặp. Cũng giống như các biến được khai báo trong một scope thì chỉ có phạm vi sử dụng trong scope đó.  
Các biểu thức Initialization statement, Condition và Update statement có thể có hoặc không, nhưng bắt buộc phải có đủ hai dấu “;”. Khi vắng mặt các biểu thức đó, các bạn phải tự định nghĩa chúng để tránh chương trình bị lặp vô hạn. Tuy nhiên, ngoại trừ các trường hợp đặc biệt, các bạn nên định nghĩa các biểu thức đó trong dòng lệnh for để tạo tính nhất quán và chuẩn mực chung của các lập trình viên.

Để dừng vòng lặp một cách ép buộc (force exit), ta sử dụng lệnh break. Lệnh này cần được đặt bên trong đoạn code của vòng lặp. Ngoài ra còn có lệnh continue để bỏ qua lần lặp hiện tại và thực hiện lần lặp kế tiếp.

Tôi đã viết một chương trình minh hoạ giúp các bạn dễ làm việc với cấu trúc for.

1. #include <stdio.h>
3. int main()
4. {
5. int s1 = 0;
6. for (int i = 0; i < 10; i++) //Full type
7. s1 += i;
8. printf("%d\n", s1);
10. int s2 = 0;
11. for (int j = 0;;j++)
12. {
13. if (j % 2 == 0)
14. {
15. s2 += j;
16. }
18. if (j > 20) //Condition
19. {
20. break;
21. }
22. }
23. printf("%d\n", s2);
25. int s3 = 0;
26. int k = 0; //Initialization
27. for (;;)
28. {
29. if (k % 3 == 0)
30. {
31. continue;
32. }
34. if (s3 > 100) //Condition
35. {
36. break;
37. }
39. s3 += k;
40. k++; //Update
41. }
42. printf("%d\n", s3);
44. return 0;
45. }
46. Cấu trúc for each

C++11 giới thiệu kiêu vòng lặp for-each cung cấp phương thức lặp đơn giản và an toàn hơn cho mảng, nhưng phổ biến đối với các cấu trúc dữ liệu danh sách như vector hay list

Lưu ý rằng trình biên dịch cần hỗ trợ C++11 để có thể sử dụng for-each

1. For-each loops

Cú pháp

for (element\_declaration : array)

statement;

element\_declaration: khai báo biến có cùng kiểu dữ liệu với các phần tử trong mảng array, dùng để lấy giá trị cho từng phần tử trong mảng.

Nếu element\_declaration khác kiểu thì sẽ được chuyển kiểu ngầm

Ví dụ dùng for-each trên mảng fibonacci:

#include <iostream>

int main()

{

int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };

for (int number : fibonacci) // duyệt mảng fibonacci

std::cout << number << ' '; // truy xuất phần tử qua biến number

return 0;

}

0 1 1 2 3 5 8 13 21 34 55 89

1. For each dùng từ khóa auto

Vì element\_declaration nên có cùng kiểu dữ liệu với phần tử trong mảng, tuy nhiên để cho thuận tiện khi dữ liệu có tên kiểu quá dài, C++ 11 cho phép dùng từ khóa auto để tự động luận kiểu cho phần tử mảng

#include <iostream>

int main()

{

int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };

for (auto number : fibonacci) // type sẽ có cùng kiểu với phần tử fibonacci array

std::cout << number << ' ';

return 0;

}

1. For-each loops dùng tham chiếu

Các ví dụ trên của chúng ta đang dùng copy-semantic :

int array[5] = { 9, 7, 5, 3, 1 };

for (auto element: array) // element được sao chép từng giá trị phần tử mảng

std::cout << element << ' ';

Việc sao chép các phần tử sẽ tốn chi phí, chúng ta nên dùng kiểu tham chiếu:

int array[5] = { 9, 7, 5, 3, 1 };

for (auto &element: array) // element đang tham chiếu tới từng phần tử mảng

std::cout << element << ' ';

Hiển nhiên, khi dùng kiểu tham chiếu thì bất kì thay đổi nào trên biến tham chiếu cũng sẽ làm thay đổi giá trị của phần tử mảng.

Trường hợp chúng ta chỉ cần truy xuất giá trị mà không muốn bị thay đổi giá trị thì cách hay nhất là dùng tham chiếu hằng

int array[5] = { 9, 7, 5, 3, 1 };

for (const auto &element: array) // element là tham chiếu hằng tới từng phần tử mảng

std::cout << element << ' ';

Luật: Nên tận dụng tham chiếu hay tham chiếu hằng trong for-each để tối ưu hiệu suất.

1. For-each loops với stl container

For-each không chỉ làm việc với mảng, mà con với các container trong thư viện STL của C++ như vector, list, set, map...

#include <vector>

#include <iostream>

int main()

{

std::vector<int> fibonacci = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };

for (const auto &number : fibonacci)

std::cout << number << ' ';

return 0;

}

1. For-each không làm việc với con trỏ tới mảng

Cơ chế duyệt của for-each cần biết chính xác kích cỡ của mảng, khi khởi tạo mảng bằng con trỏ, for-each sẽ không nhận biết được kích thước thật sự nên sẽ không làm việc được:

#include <iostream>

int sumArray(int array[]) // mảng là con trỏ

{

int sum = 0;

for (const auto &number : array) // biên dịch lỗi do kích thước mảng chưa xác định

sum += number;

return sum;

}

int main()

{

int array[5] = { 9, 7, 5, 3, 1 };

std::cout << sumArray(array); // truyền tham số mảng theo kiểu con trỏ

return 0;

}

Tương tự, mảng động cũng sẽ không làm việc với for-each.

1. Có thể lấy được index hiện tại của phần tử?

For-each không cung cấp cách trực tiếp để lấy được index phần tử, lý do là nhiều cấu trúc dữ liệu dùng trong for-each không thực sự dùng ý niệm về index trong triển khai.

### Typedef – trừu tượng hóa kiểu dữ liệu

Từ khóa **Typedef** tạo ra một tên mới cho một kiểu có sẵn, không ₫ịnh nghĩa một kiểu mới

Ý nghĩa: ₫ưa tên mới dễ nhớ, phù hợp với ứng dụng cụ thể, dễ thay ₫ổi về sau

1. Chức năng

Từ khóa typedef giúp tạo thêm một tên (định danh) cho kiểu dữ liệu. Ngoài ra, từ khóa typedef còn dùng để định nghĩa tên cho một hàm, cấu trúc, lớp. Tuy nhiên, trong phạm vi bài viết này chỉ tập trung vào việc sử dụng typedef cho kiểu dữ liệu.

1. Cú pháp khai báo
2. typedef <tên kiểu> <tên mới>;
3. Các ví dụ
4. typedef short smallNumber;
6. int main(){
7. smallNumber var\_s = 10;
9. var\_s = var\_s + 10;
10. var\_s++;
12. return 0;
13. }

* typedef là từ khóa.
* short là tên của kiểu dữ liệu.
* smallNumber là tên mới muốn đặt cho kiểu dữ liệu.

However, typedefs can be used to define more complex objects which are beyond the

scope of a simple #define statement, such as:

Tuy nhiên typedef còn có thể dùng định nghĩa một kiểu dữ liệu phức tạp hơn :

1. typedef int point2D[2];
3. int main(){
4. point2D position;
5. position[0] = 0;
6. position[1] = 1;
8. return 0;
9. }

Thay vì sử dụng mảng tĩnh 2 phân tử để khai báo một điểm như sau: int position[2], thì ta có thể sử dụng typedef để làm cho tên kiểu dữ liệu dễ hiểu hơn: point2Dposition.

1. Phạm vi khai báo

* Khai báo typedef ở trong hàm thì phạm vi sử dụng ở trong hàm đó.
* Khai báo typedef ở ngoài hàm (trong file .cpp) thì phạm vi sử dụng trong file cpp đó.
* Bạn có thể tìm hiểu việc khai báo trong file header. Trong phạm vi bài viết xin không đi sâu về vấn đề này.

### Kiểu enum

Trong lập trình, đôi khi những kiểu dữ liệu không mang lại ý nghĩa phù hợp với mục đích người lập trình, hoặc tệ hơn có thể làm trở ngại quá trình lập trình. Lấy ví dụ khi ta lưu trữ các ngày trong tuần bằng một biến integer. Ta buộc phải nhớ chính xác bảy giá trị tương tứng với 7 ngày trong tuần. Điều này gây khó khăn cho việc ghi nhớ và phát triển lâu dài. Chúng ta cần một kiểu dữ liệu mới mang lại ý nghĩa rõ ràng hơn.

1. Khái niệm

C++ cho phép lập trình viên tự tạo ra kiểu dữ liệu, cách đơn giản nhất là kiểu liệt kê. Một liệt kê là kiểu dữ liệu mà các giá trị được định nghĩa là biểu tượng hằng.(kiểu liệt kê).

Kiểu liệt kê được khai báo qua từ khóa **enum.** Xem ví dụ

// Declare a new enumeration named Color

enum Color{

// Here are the enumerators

// These define all the possible values this type can hold

// Each enumerator is separated by a comma, not a semicolon

COLOR\_BLACK,

COLOR\_RED,

COLOR\_BLUE,

COLOR\_GREEN,

COLOR\_WHITE,

COLOR\_CYAN,

COLOR\_YELLOW,

COLOR\_MAGENTA // for maximum compatibility, the last enumerator should not have a comma

}; // however the enum itself must end with a semicolon

Khai báo một kiểu liệt kê thì không tốn bộ nhớ. Vùng nhớ được cấp phát chỉ khi có biến kiểu liệt kê được định nghĩa

// Define a few variables of enumerated type Color

Color paint = COLOR\_WHITE;

Color house(COLOR\_BLUE);

1. Cú pháp khai báo
2. enum <tên enum>
3. {
4. <tên hằng 1>,
5. <tên hằng 2>,
6. ...
7. };

* Sau từ khóa enum là tên kiểu liệt kê.
* Bên trong là các hằng số, mỗi một hằng số được cách nhau bằng dấu ','. Riêng hằng số cuối cùng không có dấu ','.
* Tên hằng không được có khoảng trắng.
* Kết thúc enum phải có dấu ';'.
* Có thể tạo giá trị cho hằng số. Nếu hằng số đầu tiên không được gán giá trị, thì giá trị mặc định là 0.

1. Đặt tên enums

Tên kiểu Enum thường bắt đầu bằng kí tự hoa, và các liệt kê thường dùng chữ hoa hết. Vì các liệt kê được đặt trong cùng namspace với khai báo enum, tên một liệt kê không thể dùng trong đi dùng lại trong nhiều Enum cùng một namespace.

enum Color{

RED,

BLUE, // BLUE đang ở namespace toàn cục

GREEN

};

enum Feeling{

HAPPY,

TIRED,

BLUE // lỗi, BLUE đã có khai báo trong enum Color ở namespace toàn cục.

};

Do đó, thông thường người ta thêm tiền tố tên **enum** phía trước các liệt kê bên trong như ANIMAL\_ hay COLOR\_, dễ nhận biết và tránh xung đột tên.

1. Giá trị các liệt kê

Mỗi liệt kê được tự động gán một giá trị số nguyên dựa trên vị trí của nó trong enum. Mặc định, liệt kê đầu tiên có giá trị là zero, và những giá trị của liệt kê sau đó cứ tăng dần lên 1.

#include <iostream>

using namespace std;

enum Color{

COLOR\_BLACK, // assigned 0

COLOR\_RED, // assigned 1

COLOR\_BLUE, // assigned 2

COLOR\_GREEN, // assigned 3

COLOR\_WHITE, // assigned 4

COLOR\_CYAN, // assigned 5

COLOR\_YELLOW, // assigned 6

COLOR\_MAGENTA // assigned 7

};

int main(){

Color paint(COLOR\_WHITE);

cout << paint;

return 0;

}

Kết quả đoạn code trên in ra giá trị “4”,

Chúng ta có thể khai báo giá trị cho liệt kê, có thể âm hay dương. Bất kì liệt kê không định nghĩa sẽ có giá trị bằng giá trị liệt kê trước nó cộng thêm 1.

// define a new enum named Animal

enum Animal{

ANIMAL\_CAT = -3,

ANIMAL\_DOG, // assigned -2

ANIMAL\_PIG, // assigned -1

ANIMAL\_HORSE = 5,

ANIMAL\_GIRAFFE = 5, // cùng giá trị với ANIMAL\_HORSE

ANIMAL\_CHICKEN // assigned 6

};

Trong trường hợp này, ANIMAL\_HORSE và ANIMAL\_GIRAFFE có cùng giá trị, nên có thể hoán đổi cho nhau. Mặc dù C++ cho phép, nhưng bạn nên tránh việc gán cùng một giá trị cho 2 liệt kê trong 1 enum.

Bạn hãy tập cách không bao giờ gán một giá trị cụ thể cho liệt kê.

1. Định giá trị kiểu enum và xuất nhập

Vì các liệt kê được định lượng các giá trị số, chúng có thể được gán cho số nguyên. Nghĩa là chúng cũng có thể xuất ra màn hình như kiểu số vậy.

int mypet = ANIMAL\_PIG;

std::cout << ANIMAL\_HORSE; // được hiểu là kiểu số nguyên trước khi đẩy vào std::cout

Kết quả:

5

Trình biên dịch không hoàn toàn chuyển dạng số sang kiểu liệt kê. Đoạn code sau bị lỗi:

Animal animal = 5; // Lỗi trình biên dịch

Tuy nhiên, bạn có thể ép kiểu với static\_cast:

Color color = static\_cast<Color>(5); // ugly

Trình biên dịch cũng không cho bạn nhập giá trị cho liệt kê dùng std::cin:

enum Color{

COLOR\_BLACK, // assigned 0

COLOR\_RED, // assigned 1

COLOR\_BLUE, // assigned 2

COLOR\_GREEN, // assigned 3

COLOR\_WHITE, // assigned 4

COLOR\_CYAN, // assigned 5

COLOR\_YELLOW, // assigned 6

COLOR\_MAGENTA // assigned 7

};

Color color;

std::cin >> color; // lỗi biên dịch

Một ví dụ nữa dùng static\_cast để ép trình biên dịch gán giá trị số nguyên sang kiểu liệt kê:

int inputColor;

std::cin >> inputColor;

Color color = static\_cast<Color>(inputColor);

Mỗi kiểu enum được coi như một kiểu dữ liệu riêng biệt, Do đó, việc cố gắng gán liệt kê từ một kiểu enum khác gây lỗi biên dịch.

Animal animal = COLOR\_BLUE; // will cause compiler error

Vì là biến hằng, kiểu enum được hiển thị ở trình debug, khai báo enum sẽ tốt hơn là dùng chỉ thị #defined

1. In kiểu liệt kê

Như bạn thấy ở trên, std::out xem kiểu enum như số nguyên. Vậy có cách nào in chính tên kiểu liệt kê ra chuỗi không?

Có một cách là viết hàm dùng cấu trúc điều kiện if:

#include <iostream>

using namespace std;

enum Color{

COLOR\_BLACK, // assigned 0

COLOR\_RED, // assigned 1

COLOR\_BLUE, // assigned 2

COLOR\_GREEN, // assigned 3

COLOR\_WHITE, // assigned 4

COLOR\_CYAN, // assigned 5

COLOR\_YELLOW, // assigned 6

COLOR\_MAGENTA // assigned 7

};

void printColor(Color color){

if (color == COLOR\_BLACK) cout << "Black";

else if (color == COLOR\_RED) cout << "Red";

else if (color == COLOR\_BLUE) cout << "Blue";

else if (color == COLOR\_GREEN) cout << "Green";

else if (color == COLOR\_WHITE) cout << "White";

else if (color == COLOR\_CYAN) cout << "Cyan";

else if (color == COLOR\_YELLOW) cout << "Yellow";

else if (color == COLOR\_MAGENTA) cout << "Magenta";

else

cout << "Who knows!";

}

Bạn dùng switch để cấu trúc dòng lệnh dễ đọc hơn.

1. Phạm vi khai báo

* Khai báo kiểu liệt kê bên trong hàm thì có thể sử dụng kiểu liệt kê đó bên trong hàm.
* Khai báo kiểu liệt kê bên trong một file cpp và ngoài hàm, thì kiểu liệt kê đó có thể được sử dụng trong toàn file cpp.

Bạn có thể tìm hiểu việc khai báo trong file header. Trong phạm vi bài viết xin không đi sâu về vấn đề này.

1. Cấp phát vùng nhớ và khai báo

Kiểu Enum được xem như một phần gia đình của kiểu số, tùy thuộc vào trình biên dịch mà vùng nhớ cấp phát cho biến enum sẽ khác.

C++ chuẩn qui định kích cỡ enum cần đủ lớn để đại diện cho tất cả giá trị liệt kê. Hầu như trình biên dịch sẽ cho kiểu liệt kê có cùng kích thước với kiểu **int**

Vì trình biên dịch cần phải biết bao nhiêu vùng nhớ để cấp phát cho liệt kê, bạn không thể chuyển tiếp khai báo kiểu liệt kê. Tuy nhiên, có một cách để làm. Vì định nghĩa một liệt kê sẽ chưa cấp phát vùng nhớ, nếu một liệt kê cần cho nhiều file, sẽ tốt hơn khi định nghĩa liệt kê trong file header, và #include header file đó tại bất cứ chỗ nào cần.

1. Kiểu enum dùng để làm gì?

Enum làm việc đọc code dễ hơn, khi bạn cần biểu đạt một đặc tính nào đó, hay các trạng thái.

Ví dụ, hàm thường trả về kiểu integer, mỗi giá trị đại diện cho một lỗi cụ thể, thường là số âm.

int readFileContents(){

if (!openFile()) return -1;

if (!readFile()) return -2;

if (!parseFile()) return -3;

return 0; // success

}

Tuy nhiên, các con số không có tính mô tả, hàm sau sử dụng kiểu liệt kê để trả về:

enum ParseResult{

SUCCESS = 0,

ERROR\_OPENING\_FILE = -1,

ERROR\_READING\_FILE = -2,

ERROR\_PARSING\_FILE = -3

};

ParseResult readFileContents(){

if (!openFile()) return ERROR\_OPENING\_FILE;

if (!readFile()) return ERROR\_READING\_FILE;

if (!parsefile()) return ERROR\_PARSING\_FILE;

return SUCCESS;

}

Cấu trúc lệnh dễ đọc và nơi gọi hàm dễ dàng kiểm tra giá trị trả về với giá trị liệt kệ để bug lỗi.

if (readFileContents() == SUCCESS) {

// do something

}

else {

// print error message

}

Tốt nhất là dùng kiểu liệt kê để định nghĩa một bộ nhận dạng. Ví dụ trong một game, khi người chơi đem nhiều đồ, để phân loại đồ ta định nghĩa kiểu enum như sau:

#include <iostream>

#include <string>

using namespace std;

enum ItemType{

ITEMTYPE\_SWORD,

ITEMTYPE\_TORCH,

ITEMTYPE\_POTION

};

string getItemName(ItemType itemType){

if (itemType == ITEMTYPE\_SWORD) return string("Sword");

if (itemType == ITEMTYPE\_TORCH) return string("Torch");

if (itemType == ITEMTYPE\_POTION) return string("Potion");

}

int main(){

// ItemType là kiểu enum đã định nghĩa ở trên.

// itemType (lower case i) là tên biến kiểu enum.

// ITEMTYPE\_TORCH là giá trị liệt kê khởi tạo cho biến itemType.

ItemType itemType(ITEMTYPE\_TORCH);

cout << "You are carrying a " << getItemName(itemType) << "\n";

return 0;

}

Ví dụ khác về hàm sắp xếp:

enum SortType{

SORTTYPE\_FORWARD,

SORTTYPE\_BACKWARDS

};

void sortData(SortType type){

if (type == SORTTYPE\_FORWARD)

// sort data in forward order

else if (type == SORTTYPE\_BACKWARDS)

// sort data in backwards order

}

Nhiều ngôn ngữ lập trình dùng kiểu enum để định nghĩa chân trị. Một chân trị cơ bản gồm hai liệt kê là false và true. Tuy nhiên, trong C++, true và false được định nghĩa là từ khóa thay vì enum.

### Enum class

1. Giới hạn của Enum chuẩn

Ta định nghĩa hai enum Color và Fruit:

#include <iostream>

using namespace std;

int main(){

enum Color {

RED, // RED có cùng phạm vi với enum Color

BLUE

};

enum Fruit {

BANANA, // BANANA có cùng phạm vi với enum Fruit

APPLE

};

Color color = RED; // Color và RED truy xuất trong cùng phạm vi (không cần tiền tố ::)

Fruit fruit = BANANA; // Fruit và BANANA truy xuất trong cùng phạm vi (không cần tiền tố ::)

if (color == fruit) // Trình biên dịch so sánh như 2 số integer

cout << "color and fruit are equal\n"; //Bằng nhau, cả 2 đều = 0.

else

cout << "color and fruit are not equal\n";

return 0;

}

Như vậy ta thấy không có sự khác biệt giữa các Enum khai báo khác nhau:

* Trình biên dịch hiểu chúng như kiểu integer.
* Các liệt kê bên trong có cùng phạm vi truy xuất vói enum

1. Enum class – strong type

C++11 bổ sung một lý thuyết mới về lớp enum (hay gọi là kiểu liệt kê có phạm vi), tốt hơn về kiểu dữ liệu và phạm vi.

Cú pháp:

**enum class <tên\_enum> {danh sách liệt kê};**

Xem ví dụ sau

#include <iostream>

using namespace std;

int main(){

enum class Color {

RED, // RED nằm trong phạm vi của Color

BLUE

};

enum class Fruit {

BANANA, // BANANA nằm trong phạm vi của Fruit

APPLE

};

Color color = Color::RED; // phải truy xuất qua toán tử phạm vi::RED

Fruit fruit = Fruit::BANANA; // phải truy xuất qua toán tử phạm vi Fruit::BANANA

if (color == fruit) // Lỗi biên dịch, 2 kiểu Color và Fruit khác nhau

cout << "color and fruit are equal\n";

else

cout << "color and fruit are not equal\n";

return 0;

}

Luật về phạm vi enum không cho phép truy xuất trực tiếp, hai enum khác nhau cũng không thể so sánh. Bạn cũng không cần lo lắng về việc trùng tên giữa các liệt kê khai báo bên trong các enum khác nhau.

Có thể nói các liệt kê là một thành viên của enum class, không cần cách thêm tiền tố tên\_enum trước liệt kê, nhưng phải dùng toán tử phạm vi để truy xuất.

Với enum class, trình biên dịch không hiểu ngầm sang kiểu integer. Trong vài tình huống, việc chuyển kiểu enum class sang integer lại hữu ích, mẹo là dùng **static\_cast**

#include <iostream>

using namespace std;

int main(){

enum class Color {

RED,

BLUE

};

Color color = Color::BLUE;

cout << color; // error

cout << static\_cast<int>(color); // sẽ in ra 1

return 0;

}

Lưu ý với từ khóa **class**, cùng với từ khóa **static**, là một trong những từ khóa quá tải của C++, và có nhiều ý nghĩa khác nhau trong từng hoàn cảnh.

Mặc dù enum class dùng từ khóa **class** để khai báo, chúng không phải là lớp

1. Khai báo kiểu dữ liệu cho enum

Khai báo kiểu dữ liệu cho enum class như sau:

// we only have three colors, so no need for ints!

enum class Colors : char { RED = 1, GREEN = 2, BLUE = 3 };

Nhờ có kiểu dữ liệu rõ ràng các giá trị trong enum class có kích thước rõ ràng, C++11 có thể set kích thước chính xác enum class qua thư viện <cstdint>

#include <cstdint>

enum class Colors : std::int8\_t { RED = 1, GREEN = 2, BLUE = 3 };

1. Chuyển tiếp khai báo

Enum class có thuận lợi khác so với enum cũ là có thể fowrard khai báo:

enum class Mood;

void assessMood (Mood m);

// later on:

enum class Mood { EXCITED, MOODY, BLUE };

Chuyển tiếp khai báo cho phép bạn khai báo kiểu enum trong header file trong khi thiết lập giá trị cụ thể trong file .cpp.

Nhờ đó, bạn có thể thay đổi danh sách các giá trị enum thường xuyên mà không bắt tất cả các file liên quan phải biên dịch lại.

1. Tổng kết

Nếu bạn đang dùng trình biên dịch C++11, không có lý do gì để không dùng enum class.

Hãy nhớ những đặc trưng của enum class:

* Các liệt kê phạm vi bên trong enum class
* Phải dùng toán tử phạm vi để truy xuất liệt kê
* Trình biên dịch không còn hiểu các liệt kê là integer, nên liệt kê các enum class khác nhau sẽ không so sánh được.

### Kiểu cấu trúc (Struct):

1. Định nghĩa

Struct là một tập hợp các phần tử dữ liệu cùng kiểu hoặc khác kiểu được gộp chung thành một nhóm. Các phần tử này được gọi là thành viên của struct.

Thông dụng nhất trong C và ₫ược mở rộng trong C++

1. Cú pháp
2. struct <Tên struct>
3. {
4. <Kiểu dữ liệu thành viên số 1> <Tên thành viên số 1>;
5. <Kiểu dữ liệu thành viên số 2> <Tên thành viên số 2>;
6. …
7. };

**Ví dụ**

1. struct Subject{
2. string m\_name\_subject;
3. string m\_pre\_subject;
4. int m\_hour\_per\_week;
5. string m\_trainer;
6. };
7. Struct Time{
8. int hour,minute,second;
9. };
10. Struct Date {
11. int day, month, year;
12. };
13. Struct Student {
14. char name[32];
15. Struct Date birthday;
16. int id\_number;
17. };
18. Struct vô danh

Khai báo cấu trúc có thể bỏ qua phần tên. Ví dụ về thùng chứa dây cáp

struct {

char name[30]; // tên thùng

int quantity; // số lượng cáp trong thùng

int cost; // giá tiền một mét

} printer\_cable\_box; // khai báo biến

Biến printer\_cable\_box được định nghĩa nhưng không có kiểu dữ liệu định nghĩa. Kiểu dữ liệu trên gọi là cấu trúc vô danh.

Một trường hợp khác là tên struct, và phần khai báo biến bị bỏ qua, là một đoạn code vô nghĩa.

1. Danh sách khởi tạo cho struct

Chúng ta khởi tạo struct qua danh sách khởi tạo giá trị cho biến thành viên đặt trong cặp ngoặc nhọn {}

/\*Printer cables\*/

struct bin {

char name[30];

int quantity;

int cost;

};

struct bin printer\_cable\_box = {

"Printer Cables", // Tên thùng

0, // Bắt dầu từ thùng rỗng

1295 // Giá $12.95

};

Định nghĩa cấu trúc và khởi tạo biến có thể gộp lại 1 bước:

struct bin {char name[30]; int quantity; int cost; } printer\_cable\_box = {"Printer Cables", 0, 1295 };

1. Sử dụng struct

* Để truy cập vào các thành viên của struct ta sử dụng toán tử dấu chấm (.) đặt giữa tên biến kiểu struct và tên thành viên của struct cần truy cập.
* Một biến thuộc kiểu struct cũng có thể trở thành tham số của một hàm.
* Ta dùng toán tử (\*) và (->) để truy cập thành phần của struct thông qua biến con trỏ.

Để rõ hơn, các bạn có thể theo dõi đoạn mã nguồn dưới đây:

1. #include <iostream>
2. #include <string>
3. using namespace std;
5. /\*Declare a struct called Subject\*/
6. struct Subject
7. {
8. string m\_name\_subject; //delare member types
9. string m\_pre\_subject;
10. int m\_hour\_per\_week;
11. string m\_trainer;
12. };
14. void PrintSubjectInformation(Subject object)
15. {
16. cout << object.m\_name\_subject<< endl;
17. cout << object.m\_trainer << endl;
18. cout << object.m\_hour\_per\_week<< endl;
19. cout << object.m\_pre\_subject<< endl;
20. }
22. int main()
23. {
24. Subject subject;
26. // Access the member of struct Subject through subject
27. subject.m\_name\_subject = "Game Programming";
28. subject.m\_pre\_subject = "Basic Programming";
29. subject.m\_hour\_per\_week = 4;
30. subject.m\_trainer = "Brian Vu";
32. PrintSubjectInformation(subject);
34. return 0;
35. }
36. Bit fields

Để giúp tối ưu bộ nhớ, struct sử dụng Bit Field để qui định số bit cho một thành viên

Cú pháp:

struct {

<kieu\_du\_lieu> <ten\_bien\_1> : n1;

<kieu\_du\_lieu> <ten\_bien\_2> : n2;

...

<kieu\_du\_lieu> <ten\_bien\_k> : nk;

};

• <kieu\_du\_lieu> : là kiểu dữ liệu số nguyên (int, signed int, unsigned int,..)

• <ten\_bien\_1>, <ten\_bien\_2>, … <ten\_bien\_k>: gọi các biến bit field.

• n1, n2, n3, …, nk : kích thước (theo bit) của các biến bit field, số bit phải nhỏ hơn hoặc bằng số bit của kiểu dữ liệu.

***TẠI SAO PHẢI DÙNG BIT FIELD?***

Xét ví dụ về biểu diễn thông tin ngày, tháng, năm

struct st\_day {

unsigned int date; //ngay

unsigned int month; //thang

unsigned int year; //nam

};

Trên hệ máy 32-bits, sizeof st\_day trả về 12 (bytes). Mỗi thành viên dữ liệu có kích thước 4-bytes, nhưng sẽ không bao giờ dùng hết vì:

• Trường date có giá trị từ 1 -> 31. Chỉ cần dùng 5 bit (2^5 – 1 = 31) để biểu diễn

• Trường month có giá trị từ 1 -> 12. Chỉ cần dùng 4 bit (2^4 – 1 = 15) để biểu diễn

• Trường year có giá trị từ 0 -> 4000. Chỉ cần dùng 12 bit(2^12 – 1 = 4095) để biểu diễn.

Nếu ta dùng kiểu usigned int (kích thước 4 byte) để biểu diễn giá trị của các trường date, month, year thì gây lãng phí bộ nhớ

Ví dụ 3: So sánh kích thước của việc khai báo struct bằng Bit Field và việc khai báo không dùng Bit Field.

#include "stdio.h"

#include "conio.h"

struct st\_day {

unsigned int date; //ngay

unsigned int month; //thang

unsigned int year; //nam

};

struct bf\_day {

unsigned int date : 5; //1->31

unsigned int month : 4; //1->12

unsigned int year : 12; //0->4000

};

void main() {

printf("\nSize of no bit field struct: %d",sizeof st\_day);

printf("\nSize of bit field struct: %d",sizeof bf\_day);

getch();

}

Kết quả:

Size of no bit field struct: 12

Size of bit field struct: 4

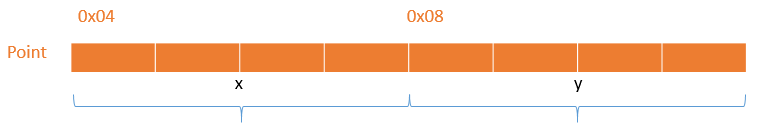
Tại sao là 4-byte? Nếu dùng 3-bytes thì đã qua đủ cho 21-bits rồi. Phần tiếp theo sẽ giải thích cách C,C++ quy định vùng nhớ cho struct.

1. Tổ chức bộ nhớ

Tôi tạo một struct Point thể hiện tọa độ của điểm trong không gian hai chiều:

1. struct Point{
2. int x;
3. int y;
4. };

Tôi lấy kích thước của struct bằng cách sử dụng toán tử sizeof được cung cấp trong C++, và tôi thấy được kích thước của nó là 8 bytes ( int là 4 byte trong HDH 32 bit, [xem lại bản chất của biến](#_Bản_chất_của)) đúng bằng các kích thước các trường dữ liệu thành viên công lại. Và dễ hình dùng hơn struct Point này được tổ chức trên bộ nhớ như sau:



Ví dụ struct Student thể hiện thông tin của một sinh viên gồm id, age, gpa là điểm trung bình học kì:

1. struct Student{
2. char id;
3. int age;
4. double gpa;
5. };

Lấy sizeof của struct Student:

1. int size = sizeof(Student);

Theo tính toán kích thước struct Student = 13 bytes ( kích thước của id (1bytes) + age (4 bytes) + gpa (8 bytes)).

Nhưng khi debug, kích thước struct Student = 16 bytes , không phải là 13 bytes. Vậy struct được cấp phát và tổ chức như thế nào ở trên bộ nhớ?.

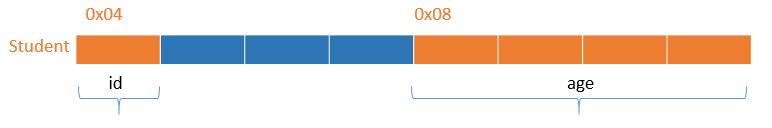
Đầu tiên trình biên dịch sẽ lấy kích thước của trường dữ liệu thành viên có kích thước lớn nhất .Trong trường hợp này là 8 bytes theo kích thước của gpa.

Sau đó cấp phát thêm 1 block gồm đúng bằng kích thước này là 8 bytes.

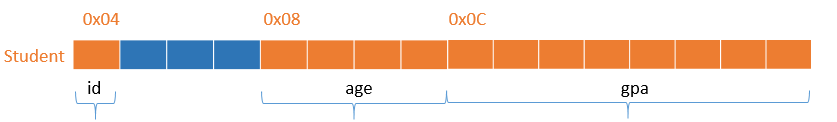
Id và age tổng cộng là 5-bytes, chưa đủ cấp số của 8 (bits) nên cần thêm 3-bytes padding. id sẽ được nạp vào trước:



Trình biên dịch sẽ padding 3-bytes ngay sau id ,  rồi "đẩy" age kiểu int có kích thước 4 bytes vào.



Đã hết chổ trống, tiếp tục cấp thêm block 8 byte nữa và "đẩy" gpa kiểu double có kích thước 8byte vào:



Vậy kích thước của struct này là 16 bytes.

Vùng nhớ cấp phát cho struct luôn luôn là liên tục trong bộ nhớ như ở trên ta thấy là địa chỉ của id, age, gpa lần lượt là 0x04, 0x08, 0x0C.

Tôi khảo sát thêm một struct nữa có tên là Demo

1. struct Demo{
2. int a, b;
3. double c, d;
4. };

Kích thước struct này là: 24 bytes;

Tôi sắp xếp lại các trường dữ liệu như sau:

1. struct Demo
2. {
3. int a;
4. double c, d;
5. int b;
6. };

Kiểm tra thấy kích thức của struct này là 32 bytes

Như vậy struct có cùng các trường dữ liệu như nhau, nhưng nếu thay đổi vị trí của chúng trong struct thì có thể thay đổi kích thước của struct.

Dựa vào cách sắp xếp bộ nhớ trong struct mà tôi đã trình bày ở trên giúp các bạn có thể tránh được việc lãng phí vùng nhớ.

Lưu ý: Cách tổ chức bộ nhớ trong class cũng tượng tự như struct.

1. Struct, class không có dữ liệu thành viên

Chúng ta đã khảo sát struct có các trường dữ liệu, nếu struct đó không có dữ liệu thành viên thì sao, việc lưu trữ như thế nào?. Một ví dụ điển hình là class xử lý toán học Math trong C#.

1. struct Person {};

Tôi có 1 struct Person không có thành viên nào cả. Và tôi lấy sizeof của nó:

1. int size = sizeof(Person);

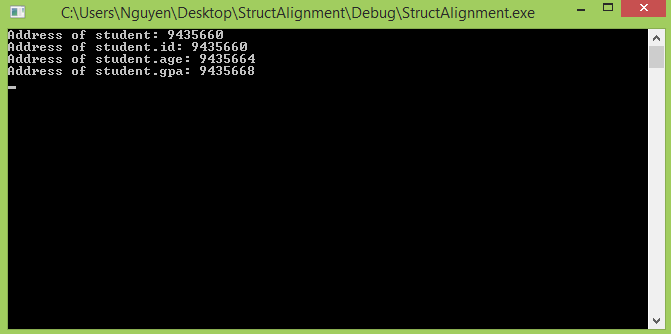
Debug thấy size có giá trị 1bytes. Thì đối với những struct, hay class như thế này thì trình biên dịch sẽ cấp phát 1 bytes để có thể lưu trữ nó dưới bộ nhớ.

1. Địa chỉ của struct, class

Cũng giống như mảng, địa chỉ của mảng chính là đại chỉ của phần tử đầu tiên, thì đối với class, struct cũng vậy. Địa chỉ của nó chính là địa chỉ của thành viên đầu tiên trong struct, class đó.

1. struct Student{
2. char id;
3. int age;
4. double gpa;
5. };
7. int main(){
8. Student student;
9. printf("Address of student: %d\n", &student);
10. printf("Address of student.id: %d\n", &student.id);
11. printf("Address of student.age: %d\n", &student.age);
12. printf("Address of student.gpa: %d\n", &student.gpa);
13. return 0;
14. }

Và kết quả cho thấy rằng địa chỉ của student cũng chính là địa chỉ của  id trong struct đó:



### Kiểu Union:

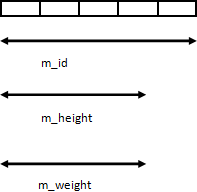
Cũng giống như struct, union là một tập hợp các phần tử dữ liệu cùng kiểu hoặc khác kiểu được gộp chung thành một nhóm. Các phần tử này được gọi là thành viên của union.

Nhưng khác với struct, union không có đủ vùng nhớ cho tất cả dữ liệu, mà dùng chung vùng nhớ.

Ít thông dụng trong cả C và C++.

1. Cú pháp
2. union <Tên cấu trúc> {
3. <Kiểu dữ liệu > <Tên thành viên>;
4. <Kiểu dữ liệu > <Tên thành viên>;
5. …
6. };
7. So sánh struct và union

Về mặt ý nghĩa, struct và union cơ bản giống nhau. Tuy nhiên, về mặt lưu trữ trong bộ nhớ, chúng có sự khác biệt rõ rệt như sau:

* Dữ liệu của các thành viên của struct được lưu trữ ở những vùng nhớ khác nhau. Do đó kích thước của một struct chắc chắn lớn hơn kích thước của các thành viên cộng lại.
* [](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_1/166/ss_3.png)với union, các thành viên sẽ dùng chung một vùng nhớ. Kích thước của union được tính là kích thước lớn nhất của kiểu dữ liệu trong union. Việc thay đổi nội dung của một thành viên sẽ dẫn đến thay đổi nội dung của các thành viên khác.

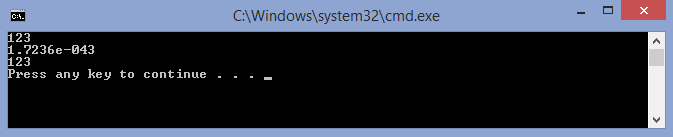
1. Sử dụng union

* Để truy cập vào các thành viên của union ta sử dụng toán tử dấu chấm (.)đặt giữa tên biến kiểu union và tên thành viên của union cần truy cập.
* Một biến thuộc kiểu union cũng có thể trở thành tham số của một hàm.
* Có thể truy cập thành viên của union thông qua biến con trỏ.

Để rõ hơn, các bạn có thể theo dõi đoạn mã nguồn dưới đây:

1. #include <iostream>
3. using namespace std;
5. /\* Declare \*/
6. union Post {
7. int m\_iValue;
8. float m\_fValue; //delare member types
9. long m\_lValue;
10. };
12. void PrintSubjectInformation(Post post) {
13. cout << post.m\_iValue << endl;
14. cout << post.m\_fValue << endl;
15. cout << post.m\_lValue << endl;
16. }
18. int main() {
19. Post post;
20. post.m\_iValue = 169; // Access the member of union
21. post.m\_fValue = 39.01;
22. post.m\_lValue = 123;
24. PrintSubjectInformation(post);
26. return 0;
27. }

**Output**

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_1/166/ss_1.png)

Khi thay đổi giá trị post.m\_lValue = 123;. Ta đã làm ảnh hướng đến giá trị của các thành viên còn lại. Điều này sẽ được giải thích ở phần sau.

1. Union vô danh

Bạn có thể khai báo 1 union mà không có tên. Thành viên dữ liệu được truy xuất trực tiếp. Tuy nhiên có vài giới hạn:

* Chúng phải được khai báo là static nếu khai báo trong phạm vi file.
* Nếu khai báo phạm vi local, chúng phải khai báo static hoặc automatic.
* Thành viên dữ liệu đều là public trong union vô danh
* Không có thành viên hàm trong union vô danh.

Không chỉ bỏ qua không khai báo tên, để union thành vô danh, bạn không được khai báo đối tượng nào phía sau.

Ví dụ:

*// anonymous\_unions.cpp*

#include <iostream>

**using** **namespace** std;

int main() {

**union** {

int d;

char f;

};

d = 4;

cout << d << endl;

f = 'i';

cout << f << endl;

}

Union vô danh được hỗ trợ trong C++11 và non-standard GCC

1. Các trường hợp dùng Union

Trong 3 kiến thức cơ bản về struct - union - enum, có thể nói union khó hiểu không phải về cách sử dụng mà sử dụng để làm gì?

Với các nhà phát triển có kinh nghiệm thì đã gặp và giải quyết các vấn đề của họ bằng union, và để cho các bạn mặc dù tốt về kỹ thuật lập trình nhưng đôi lúc chưa có trải nghiệm qua nhiều vấn đề mà giải quyết bằng union có thể biết được tính hữu dụng của nó, tôi chia sẻ vài trường hợp.

Khi tìm hiểu về union, bạn sẽ làm quen với khái niệm DÙNG CHUNG VÙNG NHỚ và sẽ cảm thấy rất khó hiểu và không biết vì sao, lúc nào cần DÙNG CHUNG VÙNG NHỚ vì nó có vẻ như KHÔNG CẦN THIẾT, DÙNG STRUCT ĐỦ RỒI.

Ví dụ về cú pháp khai báo và sử dụng

1. // Khai báo 1 union
2. union Hardware
3. {
4. float \_cpu;
5. short \_ram;
6. int \_hdd;
7. };
9. // Sử dụng union
10. int main()
11. {
12. Hardware h;
14. h.\_cpu = 3.2f;
15. h.\_ram = 256;
16. h.\_hdd = 1024;
18. return 0;
19. }

union Hardware trên sẽ có kích thước của thuộc tính lớn nhất, trong trường hợp này float hoặc int là 4 bytes nên union này có kích thước là 4 bytes.

Vậy với như trên cách lưu trữ này, cả 3 thuộc tính trên sẽ bị ghi đè lên nhau khi 1 trong 3 thuộc tính được gán giá trị. Nếu bạn chưa nghĩ ra ý tưởng sử dụng nó thì union thật tai hại, tuy nhiên nó tồn tại có cái lý của nó. Ta bắt đầu thử vài trường hợp để thấy tính hiệu quả của union.

1. Cần tiết kiệm vùng nhớ

Đặt vấn đề, trong 1 phạm vi code, 1 biến được cấp phát tĩnh sẽ được khởi tạo khi bắt đầu vào hàm và chỉ hủy khi kết thúc hàm. Giả sử như ta có 1 hàm có 16 dòng codes như giả định bên dưới

void stdio\_foo()

// (1) Đoạn code cần dùng biến ram

int ram = 2048; // MB

if (ram > 1){

printf("STDIO program needs only 1 byte to function.\n");

printf("STDIO is good.");

}

// (2) Đoạn code không cần dùng biến ram, nhưng cần dùng biến hdd

int hdd = 1024; // GB

if (hdd > 1){

printf("STDIO needs less than 100MB to store system.");

}

}

Với đoạn codes trên, ta cần 8 bytes dành cho 2 biến là ram và hdd, mặc dù ta biết ram chỉ sử dụng 1 nửa hàm, còn nửa còn lại không cần nhưng ta vẫn không thể tận dụng được 4 bytes của ram, bởi vì nó liên quan tới tính bảo trì, dù biến tên là ram để sử dụng như hdd thì sẽ gây hiểu lầm cho tương lai, do đó ta phải tốn thêm 4 bytes cho hdd.

Để giải quyết được vấn đề này sao cho toàn vẹn? Ta sẽ sử dụng năng lực của trình biên dịch và sự hỗ trợ của cú pháp C/C++, cụ thể là ràng buộc bằng union. Điều mong đợi là giữ được codes rõ ràng và chỉ dùng 4 bytes vùng nhớ. Ta cải tiến codes trên như sau

1. // Khai báo 1 union
2. union Hardware{
3. short \_ram;
4. int \_hdd;
5. };
7. void stdio\_foo(){
8. Hardware hw;
10. // (1) Đoạn code cần dùng ram
11. hw.ram = 2048; // MB
12. if (hw.ram > 1){
13. printf("STDIO needs only 1 byte to run.\n");
14. printf("STDIO is good.");
15. }
16. // (2) Đoạn code không cần dùng ram, nhưng cần dùng hdd
17. hw.hdd = 1024; // GB
19. if (hw.hdd > 1){
20. printf("STDIO needs less than 100MB to store system.");
21. }
22. }

Mặc dù đa phần các chương trình đào tạo hiện nay đã lướt qua hoặc bỏ việc đề cập union. Lý do có thể là do bộ nhớ của máy tính ngày càng lớn, và điều này trở nên lạc hậu "tại thời điểm đó". Nhưng những năm gần đây, xu hướng IoT mang chúng ta quay trở lại với hạ tầng điện tử, tôi thấy đó là cơ hội rất lớn để ta quay lại cái gốc của mình. Ví dụ như Arduino Uno R3 dùng ATmega328P, dung lượng bộ nhớ (RAM) chỉ khoảng 2KB (2048 bytes) là rất nhỏ với tư duy phần mềm trước giờ.

Để có được cơ hội mới, ta phải biết tận dụng nhiều kỹ thuật và sử dụng phần cứng hiệu quả hơn. Xem Thông Số Kĩ Thuật Arduino Uno R3 - Các Biến Thể Và Lưu Ý để biết được sự khó khăn này và cách mà union tạo ra cơ hội.

1. Cần tổ chức chương trình

Xem xét đoạn codes bên dưới

1. union Vector4{
2. struct {
3. float r, g, b, a;
4. }
5. struct {
6. float x, y, z, w;
7. }
8. struct {
9. float s, t, p, q;
10. }
11. float color[4];
12. float position[4];
13. float textcoord[4];
14. };
16. void rgba\_vn(){
17. Vector4 logo\_color;
18. logo\_color.r = 255.0f;
19. logo\_color.color[0] = 240.0f;
20. Vector4 monster\_position;
21. monster\_position.x = 100.0f;
22. // ...
23. }

Khi tôi thiết kế các tính năng cho trang web rgba.vn của STDIO, tôi đã tổ chức lại kiểu dữ liệu Vector4 dùng với nhiều mục đích khác nhau, với 1 kiểu dữ liệu như trên, tôi đã dùng linh hoạt nó như màu sắc, hay tọa độ, hoặc có thể dùng với dạng mảng như mong muốn.

### Cơ bản về hàm

Hàm là một nhóm câu lệnh cùng thực hiện một tác vụ. Mỗi chương trình C,C++ có ít nhất 1 hàm là hàm **main**() và hầu hết các chương trình bình thường khác đều khai báo thêm các hàm bổ sung.

Bạn có thể chia code thành nhiều hàm tùy ý, nhưng theo logic thì mỗi hàm sẽ có 1 tác vụ chuyên biệt.

Khai báo hàm giúp trình biên dịch biết tên hàm, kiểu return và tham số parameters.

Trong mỗi thư viện C,C++ đều cung cấp rất nhiều hàm.

1. Định nghĩa hàm:

Cú pháp:

return\_type function\_name( parameter list ){

body of the function

}

1 hàm gồm 3 thành phần sau

* **Return Type**: là kiểu dữ liệu mà hàm sẽ trả về. Những hàm không có ý định trả về giá trị nào sẽ dùng từ khóa Void
* **Function Name:**  Tên hàm
* **Parameters:** Đối số hàm, nơi bạn truyền giá trị vào hàm. Danh sách đối số định nghĩa kiểu, thứ tự và số lượng. Các tham số là tùy biến, có thể không có.
* **Function Body:** Thân hàm chứa các tập lệnh.

**Ví dụ:**

// function returning the max between two numbers

int max(int num1, int num2)

{

// local variable declaration

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

1. Khai báo hàm:

Hàm sẽ được khai báo ở top trước khi được định nghĩa

int max(int num1, int num2);

Void main(){

// ...

int max(int num1, int num2) {

// ...

}

1. Gọi hàm:

Khi gọi tên hàm, ta truyền tham số kèm theo tên hàm. Hàm sẽ trả về giá trị hoặc không. Ví dụ:

#include <iostream>

using namespace std;

// function declaration

int max(int num1, int num2);

int main (){

// local variable declaration:

int a = 100;

int b = 200;

// calling a function and print max value to screen

cout << "Max value is : " << max(a, b)<< endl;

return 0;

}

// function returning the max between two numbers

int max(int num1, int num2) {

Return (num1>num2) ? num1:num2;

}

Kết quả:

Max value is : 200

1. Đối số hàm:

Nếu hàm có sử dụng tham số, ta phải khai báo danh sách tham số hàm trong dấu ngoặc “()” sau tên hàm khi khai báo và định nghĩa hàm.

Các tham số này được gọi là tham số chính thức của hàm. Chúng có đặc tính như biến cục bộ của hàm, nên sẽ bị hủy khi kết thúc hàm.

Khi gọi hàm, các biến được truyền vào danh sách tham số được gọi là đối số của hàm.

Có ba cách để truyền giá trị cho đối số của hàm:

|  |  |
| --- | --- |
| **Kiểu truyền** | **Ý nghĩa** |
| [**Truyền tham trị**](http://www.tutorialspoint.com/cplusplus/cpp_function_call_by_value.htm) | Phương thức này chỉ sao chép giá trị của đối số vào tham số hàm. Không tác động tới đối số. |
| [**Truyền con trỏ**](#_Con_trỏ_với) | Phương thức này sao chép địa chỉ của đối số và truyền vào tham số hàm. Địa chỉ này sẽ được dùng để truy xuất giá trị đối số trong hàm.  Phương thức này có thể thay đổi giá trị của đối số.  Nếu không muốn tác động vào đối số hàm, ta có thể khai báo kiểu con trỏ hằng như sau:  Type function\_name(const type \*pointer) |
| [**Truyền tham chiếu**](#_Tại_sao_phải) **(C++)** | Phương thức này định nghĩa tham số hàm như một đại diện của đối số. Bất kỳ tác động nào lên tham số cũng thay đổi giá trị đối số. |

1. Giá trị trả về
2. Trả về giá tri

Là phương thức đơn giản nhất và an toàn. Thân hàm tạo ra bản sao của giá trị, trả ngược về nơi gọi hàm. Với cách truyền giá trị vào hàm, bạn có thể trả về chuỗi hằng, biến, hay biểu thức.

Thuận lợi khác là khi trả về, biến phạm vi hàm sẽ tự hủy. Lưu ý là khi trả về giá trị kiểu cấu trúc hay lớp sẽ chậm hơn khá nhiều.

Chúng ta trả về giá trị khi

* Giá trị của biến được khai báo trong thân hàm
* Trả về đối số hàm

Không nên dùng khi

* Trả về mảng truyền thống hay con trỏ (trả về địa chỉ)
* Trả về cấu trúc và lớp.

1. Trả về địa chỉ

Giống như truyền đối số địa chỉ, hàm tham chiếu luôn trả về địa chỉ của biến, không phải của chuỗi hằng hay biểu thức.

Trả về địa chỉ nghĩa là sao chép địa chỉ từ hàm về nơi gọi, trả về địa chỉ có tốc độ nhanh.

Tuy nhiên, có một điểm bất lợi là bạn không thể return địa chỉ của biến sinh ra trong hàm. Ví dụ:

int\* doubleValue(int x)

{

int value = x \* 2;

return &value; // Trả về địa chỉ của value

} // value sẽ bị hủy khi thoát khỏi hàm

Biến value bị hủy sau khi địa chỉ của nó trả về nơi gọi. Kết quả là nơi gọi nhận được một địa chỉ vùng nhớ chưa được cấp phát, sẽ phát sinh lỗi nếu bạn truy xuất.

Đây là lỗi khá phổ biến với lập trình viên mới. Vì vùng nhớ của hàm nằm trong stack, sau khi hàm kết thúc, vùng nhớ được giải phóng khởi stack, trừ khi bạn trả về địa chỉ vùng nhớ data hay heap.

Ví dụ sau minh họa một mẹo cấp phát vùng cho con trỏ thông qua hàm

int\* allocateArray(int size){

return new int[size]; // toán tử new trả về địa chỉ đầu tiên của vùng nhớ cấp phát

}

int main(){

int \*array = allocateArray(25);

delete[] array; // hủy vùng nhớ

return 0;

}

Chúng ta trả về địa chỉ khi:

* Trả về vùng nhớ cấp phát động
* Trả về đối số truyền kiểu địa chỉ

Không nên dùng khi

* Trả về địa chỉ biến cục bộ hàm
* Trả về cấu trúc hay lớp.

1. Trà về tham chiếu

Tương tự như truyền tham chiếu, giá trị tham chiếu trả về phải là của biến. Hàm sẽ trả về nơi gọi tham chiếu của biến, nơi gọi dùng tham chiếu này để thao tác lên biến.

Kiểu trả về này khá nhanh, đặc biệt dùng với cấu trúc hay lớp.

Tuy nhiên hàm không thể trả về tham chiếu biến cục bộ. Hàm tham chiếu thường dùng để trả về tham chiếu đối số truyền vào. Ví dụ sau minh họa phần tử của khuôn mẫu array truyền tham chiếu vào hàm

#include <array>

#include <iostream>

using namespace std;

// hàm trả về tham chiếu phần tử của mảng

int& getElement(array<int, 25> &array, int index){

// array[index] không bị hủy khi thoát khỏi hàm

// nên trả về được

return array[index];

}

int main(){

array<int, 25> array;

// gán 5 vào phần tử 10

getElement(array, 10) = 5;

cout << array[10] << '\n';

return 0;

}

Khi nào bạn dùng hàm tham chiếu

* Trả về tham chiếu đối số truyền vào
* Trả về cấu trúc hay lớp có phạm vi nằm ngoài hàm

Không nên dùng:

* Khi trả về biến cục bộ nằm trong hàm
* Khi trả về giá trị của mảng truyền thống hay con trỏ (dùng kiểu trả về địa chỉ)

1. Trả về hỗn hợp tham chiếu và giá trị

Xem đoạn code sau:

int returnByValue(){

return 5;

}

int& returnByReference(){

static int x = 5; // biến static nằm trong vùng nhớ data

return x; // x không bi hủy khi ra khỏi hàm

}

int main(){

int value = returnByReference(); // Hàm trả về giá trị cho value.

int &ref = returnByValue(); // Trường hợp B - error,

const int &cref = returnByValue(); // Trường hợpc C – giá trị trả về tồn tại song song với &cref

}

Trong trường hợp B, xảy ra lỗi khi khởi tạo tham chiếu **ref** với bản sao trả về của hàm **returnByValue(),** vì giá trị trả về không có địa chỉ. (kiểu Rvalue).

Trường hợp C, khởi tạo tham chiếu hằng **cref** với bản sao trả về bởi **returnByValue()**, khá ngạc nhiên là trình biên dịch không báo lỗi. Hằng tham chiếu có thể gán tới Rvalues.

1. Kết luận

Trong mọi trường hợp, trả về giá trị đáp ứng nhu cầu tính toán của bạn, nó mềm dẻo và an toàn. Nhưng trả về địa chỉ và tham chiếu đặc biệt hiệu quả hơn khi dùng với lớp hay cấu trúc cấp phát động.

Khi trả về địa chỉ hay tham chiếu, bạn phải chắc đối tượng trả về không ra ngoài phạm vi khi hàm kết thúc.

1. Giá trị mặc định:

Chúng ta khởi tạo giá trị mặc định cho tham số bằng phép gán giá trị khi khai báo hàm. Nếu không truyền tham số khi gọi hàm, giá trị mặc định sẽ được gán. Ví dụ:

#include <iostream>

using namespace std;

int sum(int a, int b = 20){

return (a+b);

}

int main (){

// local variable declaration:

int a = 100;

int b = 200;

// calling a function to add the values.

cout << "Total value is :" << sum(a, b)<< endl;

// calling a function again as follows.

cout << "Total value is :" << sum(a)<< endl;

return 0;

}

Kết quả là:

Total value is :300

Total value is :120

### Random Số Trong C,C++

Trong quá trình test chương trình, vấn đề nhập số liệu có thể làm bạn cảm thấy vô cùng nhàm chán, tốn thời gian và thậm chí là khiến bạn căng thẳng. Đặc biệt, điều này càng dễ dàng xảy ra hơn nếu bạn gặp phải lỗi phải sửa đi sửa lại nhiều lần hoặc số lượng số liệu cần nhập quá nhiều. Hiểu được điều đó C,C++ đã cung cấp cho bạn hàm **rand()** nhằm giúp bạn tạo ngẫu nhiên số một cách dễ dàng hơn.

Để dùng hàm rand() trong C bạn khai báo thư viện <stdlib.h> . Tương ứng trong C++ là thư viện <cstdlib.h>.

Ngoài ra trong bài cũng cần thư viện <time.h> và <ctime>

1. Cơ chế sinh số

Số ngẫu nhiên được sinh ra bởi công thức toán học nổi tiếng Linear Congruential Generator (LCG), sử dụng quan hệ đệ qui

**xn+1 = (axn + b) mod m với n = 0, 1, 2, 3, …**

Ví dụ cho a=5, b=3, m=16, và x0=3, Ta được chuỗi số:

x0 = 3, x1 = 2, x2 = 13, x3 = 4, …

Giá trị x zero được xem là giá trị mầm. LCG sẽ tự đệ qui, các hằng số a,b và m sẽ quyết định một bộ sinh số tốt hay dở.

Theo Knuth, một LCG tốt phải là :

**( 3141592653xn + 2718281829 ) mod 235  x0 = 0**

Dựa vào công thức ta luôn đoán trước số tiếp theo là gì, tuy nhiên công thức đã phân phối giá trị đều giữa 0 và **RAND\_MAX**.

Như John Von Neumann đã tuyên bố năm 1951:

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.”

**LCG phát hiện bởi Dick Lehmer năm 1949. Lehmer là nhà toán học lỗi lạc trong thế kỷ 20. Ông ấy góp phần xây dựng ENIAC, chiếc máy tính đầu tiên của Mỹ, và cũng là giám đốc của học viện tiêu chuẩn quốc gia Hoa Kỳ trong phân tích số học.**

1. Rand() trong C

Hàm **int rand(void)** trả về số ngẫu nhiên từ 0 tới giá trị **RAND\_MAX** .

RAND\_MAX là marco nguyên dương, giá trị của nó thay đổi giữa những lần sử dụng hàm, nhưng tối đa là 32767

Cú pháp

int rand(void)

Giá trị return nằm giữa 0 tới RAND\_MAX.

Để tạo các giá trị khác nhau giữa nhiều lần chạy chương trình ta dùng hàm **srand(),** cũng nằm trong thư viện <stdlib.h>

Bạn cũng cần include thêm thư viện <time.h>.

Ta sẽ dùng hàm **time(time\_t \*time)** return về số giây tính từ 01-jan-1900 tới thời điểm hiện tại cho tham số **seed** của hàm **srand().** Như vậy số seed sẽ liên tục thay đổi mỗi lần chạy.

Cú pháp

void srand(unsigned int seed)

Tham số **seed** được dùng cho giải thuật sinh số ngẫu nhiên trong hàm **rand()**

Ví dụ

#include <stdio.h>

#include <stdlib.h>

int main(){

int i, n;

n = 5;

/\* Intializes random number generator \*/

srand((unsigned)time(Null));

/\* Print 5 random numbers from 0 to 49 \*/

for( i = 0 ; i < n ; i++ ) {

printf("%d\n", rand() % 50);

}

return(0);

}

38

45

29

29

47

1. Rand() trong C++

Bạn cần khai báo thư viện <cstdlib> , <ctime> trước khi sử dụng hàm **srand()** và **rand().**

Về nguyên tắc vẫn giống như trong C

Ví dụ:

#include <cstdlib>

#include <ctime>

int main(){

srand(time(NULL));

int a = rand();

return 0;

}

**Lưu ý:** Giá trị RAND\_MAX trên từng trình biên dịch là không giống nhau. Chẳng hạn như với trình biên dịch của **Visual Studio 2013 cho C++**, giá trị RAND\_MAX là 32767. Bạn có thể kiểm tra giá trị này một cách dễ dàng bằng đoạn code ngắn gọn sau.

Ví dụ:

#include <cstdlib>

#include <iostream>

using namespace std;

int main(){

cout << RAND\_MAX;

return 0;

}

1. Tạo số ngẫu nhiên với một khoảng xác định

Hai cách sử dụng trên đều cho bạn kết quả giá trị ngẫu nhiên là những giá trị trong khoảng từ 0 đến RAND\_MAX. Vậy nếu bạn muốn tạo ra một giá trị ngẫu nhiên trong khoảng xác định mà mình muốn thì sao?

Rất đơn giản, bạn có thể sử dụng theo công thức sau:

**rand() % (b – a + 1) + a.**

Khi ấy, bạn sẽ nhận được kết quả trả về là một giá trị nằm trong khoảng từ a đến b.

Ví dụ: random giá trị trong khoảng [1, 50]

#include <cstdlib>

#include <ctime>

int main(){

srand(time(NULL));

int a = rand() % 50 + 1;

return 0;

}

## C/C++ Nâng cao

### Chuyển tiếp khai báo

1. Tình huống

Xem đoạn code sau:

#include <iostream>

int main()

{

std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl;

return 0;

}

int add(int x, int y)

{

return x + y;

}

Trình biên dịch sẽ báo lỗi (“identifier not found”) ngay lập tức vì nó chưa hiểu hàm add hay chưa nhìn thấy định nghĩa của add.

Điều này có thể giải quyết bằng cách chuyển định nghĩa hàm add trước hàm main.

Tuy nhiên, khi xét trường hợp 2 hàm gọi qua lại lẫn nhau thì không có cách nào để xử lí cả.

1. Thủ tục hàm và chuyển tiếp khai báo

Chuyển tiếp khai báo sẽ giúp trình biên dịch biết về sự tồn tại của đối tượng biên dịch trước khi nhìn thấy định nghĩa của đối tượng đó.

Đối với hàm gọi, trình dịch hiểu rằng đang có lệnh gọi hàm và có thể kiểm tra để đảm bảo hàm được được thực hiện đúng, dù nó chưa thấy định nghĩa của hàm gọi.

Để chuyển tiếp khai báo, đơn giản chỉ cần viết thủ tục hàm lên trước gồm kiểu trả về, tên hàm, tham số và dấu chấm phẩy ở cuối.

#include <iostream>

int add(int x, int y); // forward declaration of add()

int main()

{

std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl; // this works because we forward declared add() above

return 0;

}

int add(int x, int y) { return x + y;}

1. Bỏ quên định nghĩa

Nếu chuyển tiếp khai báo, nhưng quên không định nghĩa thì tùy thuộc vào ngữ cảnh.

Nếu hàm không bao giờ được gọi, hoạt động biên dịch diễn ra bình thường, ứng dụng vẫn chạy. Tuy nhiên, nếu hàm được gọi, ứng dụng vẫn biên dịch thành công nhưng trình liên kết sẽ cảnh báo rằng nó không thể xử lí hàm gọi này.

add.obj : error LNK2001: unresolved external symbol "int \_\_cdecl add(int,int)" (?add@@YAHHH@Z)

add.exe : fatal error LNK1120: 1 unresolved externals

1. Giảm thời gian build

Bạn có thể đặt chuyển tiếp khai báo của hàm trong file .cpp hay file .h, bằng cách #include header có khai báo thủ tục hàm.

Tuy nhiên, điều nay làm chậm tốc độ biên dịch, đặc biệt khi #include header vào một file header khác thay vì file .cpp.

Mọi thứ trong file header sẽ được chèn vào nội dung file header đang #include vào, trình biên dịch sẽ phải biên dịch tất cả mọi thứ trong 2 file dù bạn chỉ cần dùng một phần trong đó thôi.

Để tránh điều này, chúng ta dùng chuyển tiếp khai báo đưa lên đầu file, trình biên dịch sẽ chạy nhanh hơn so với cách #include header.

Dự án càng lớn thì thời gian biên dịch tiết kiệm được càng nhiều.

### Kiểm soát Heap memory C,C++

1. Tình huống

Chúng ta đang bàn về các phương pháp sử dụng **malloc, calloc, realloc, new**

Xem xét vài tình huống. Trên C:

char\* a = (char\*)malloc(256);

char\* b = (char\*)malloc(256);

a = b; // Memory leak!

Vùng nhớ con trỏ a bị rò rỉ, cách giải quyết:

char\* a = (char\*)malloc(256);

char\* b = (char\*)malloc(256);

char\* c = (char\*)malloc(256);

memcpy (c, a, 256); // then

free(a);

a = b;

Và trên C++:

ABigClass\* a = new ABigClass();

ABigClass\* b = new ABigClass();

a = b; // Memory leak!

Thêm trường hợp nữa liên quan đến việc sao chép chuỗi:

char\* srcStr = "Hello there";

char\* textString = (char\*)malloc(strlen(srcStr));

strcpy(textString, srcStr);

// không chừa chỗ cho kí tự kết chuỗi ‘\0’

Strcpy() không an toàn vì nó sao chép vùng nhớ cho tới khi gặp byte kết chuỗi. Trường hợp quên không khai báo byte kết chuỗi, khả năng strcpy() sẽ sao chép một lượng lớn memory vượt ra ngoài vùng nhớ của con trỏ.

Khuyến cáo nên dùng strncpy(), vì nó có tham số khai báo số kí tự cần sao chép:

char\* srcStr = "Hello there";

char\* textString = NULL;

textString = malloc(strlen(srcStr) + 1);

if (textString != NULL) {

memset(textString, '\0', strlen(srcStr) + 1);

strncpy(textString, srcStr, strlen(srcStr));

// …

}

if (textString != NULL) {

free(textString);

textString = NULL;

}

1. Hậu quả

Nếu việc rò rỉ diễn ra thường xuyên, chương trình càng lớn, tần suất xử lí càng nhiều thì vùng nhớ càng hao hụt nhanh chóng (vùng nhớ vật lý và sau đó là vùng nhớ ảo).

1. Set Null cho con trỏ

Tưởng tượng bạn đang quản lý vùng nhớ của hàng loạt con trỏ, nếu không gán NULL, bạn sẽ không biết con trỏ nào đã cấp phát hay đã thu hồi vùng nhớ. Để tránh việc thu hồi vùng nhớ cho con trỏ chưa cấp phát, thì cách tốt nhất là gán NULL cho con trỏ chưa cấp phát/ đã thu hồi vùng nhớ

Trường hợp nữa là khi chương trình đang thực thi thì xảy ra ngoại lệ, nếu không khai báo NULL, làm sao biết con trỏ nào cần phải thu hồi vùng nhớ khi xử lí exception.

1. Giải pháp

* Chỉ thu hồi vùng nhớ (**free**) khi cấp phát thành công

char\* textString = NULL;

textString = malloc(256);

if (textString == NULL) {

cout<< "Failed malloc" << endl;

return NULL;

}

// Dùng free() chỉ khi if malloc() thành công

if (textString != NULL) {

free(textString);

textString = NULL;

}

Nếu dùng free trên con trỏ chưa cấp phát có thể gây ra lỗi không ngờ tới.

* Thứ 2: khi khai báo con trỏ, hay sau khi thu hồi vùng nhớ, con trỏ phải gán **NULL**
* Thứ 3: Phải nhận ra trường hợp nào các hàm trong library trả về vùng nhớ cấp phát.

Nhưng giải pháp ở trên là chỉ là tạm thời , vì có nhiều trường hợp nữa chưa kể tới.

### Debug và tối ưu code

##### Debug

1. Vấn đề:

Phần khó nhất của một chương trình không phải là thiết kế và viết, nhưng giai đoạn gỡ lỗi. Bạn phải tìm hiểu cách chương trình của bạn làm việc .

Để sửa lỗi, bạn cần hai điều:

* Tìm lỗi và thông tin từ các chương trình cho phép bạn xác định vị trí
* Sửa chữa vấn đề.

Trong một số trường hợp, việc tìm kiếm các lỗi là dễ dàng. Bạn phát hiện ra các lỗi chính mình, các phòng lab

tạo ra một bài kiểm tra rõ ràng và dễ dàng hiển thị các lỗi, hoặc đầu ra luôn luôn đi ra xấu.

Trong một số trường hợp, đặc biệt là với các chương trình tương tác, lỗi phát sinh chiếm tới 90 %

vấn đề. Điều này đặc biệt đúng khi đối mặt các báo cáo từ người dùng trong lĩnh vực này.

Chương trình sắp xếp database ngắn, nó yêu cầu người dùng nhập dữ liệu là tên và kiểm tra lại tên trong database . Dữ liệu khá lớn và phức tạp

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Database – Minh họa bằng một danh sách cố định \*

\* Chương trình sẽ hỏi bạn tên. \*

\* Nhập tên – nó sẽ nói tên bạn có trong danh sách hay không\*

\* Tên trống sẽ tắt chương trình. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const int STRING\_LENGTH = 80; /\* Độ dài mặc định cho tên \*/

#include <iostream>

#include <string>

main()

{

char name[STRING\_LENGTH]; // tên nhập vào

int lookup(char \*); // tìm

while (1) {

std::cout << "Enter name: ";

std::cin.getline(name, sizeof(name));

// Kiểm tra tên trống

if (strlen(name) <= 0)

break;

if (lookup(name))

std::cout << name <<" is in the list\n";

else

std::cout << name <<" is not in the list\n";

}

return (0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Lookup – tìm tên trong danh sách \*

\* Tham số name

\* Return \*

\* 1 – có trong list \*

\* 0 – không có trong list \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int lookup(char \*name){

// Note: NULL ám chỉ kết thúc danh sách

static char \*list[] = {

"John",

"Jim",

"Jane",

"Clyde",

NULL

};

int index; // Index trong list

for (index = 0; list[index] != NULL; ++index) {

if (strcmp(list[index], name) == 0)

return (1);

}

return (0);

}

Kết quả:

Enter name: Sam

Sam is not in the list

Enter name: John

John is in the list

Enter name:

##### Assert và static\_assert

Sử dụng các lệnh điều kiện để kiểm tra cùng với xuất lỗi và ngắt ứng dụng là một trong những phương tiện phổ biến trong C++, gọi là xác thực (assert)

Một dòng lệnh xác thực là một Macro để đánh giá biểu thức điều kiện trong runtime. Nếu biểu thức đúng, thì dòng lệnh assert không làm gì cả, ngược lại, nếu biểu thức sai, một dòng lỗi in trên màn hình và ứng dụng sẽ ngắt.

Dòng tin nhắn lỗi chứa thông tin về biểu thức điều kiện thất bại, cùng với tên file, hàm, dòng lệnh bao nhiêu, giúp truy tìm nguyên nhân dễ dàng.

Các tính năng xác thực đều nằm trong header <cassert>, thường được dùng để kiểm tra tham số truyền vào hàm là hợp lệ, và cả giá trị trả về của hàm là hợp lệ hay không.

1. Std::assert

Phương thức **Std::assert,** nằm trong header <assert> C và <cassert> C++

Cú pháp:

**Std::assert(condition)**

Nếu macro **NDEBUG** được định nghĩa thì assert chẳng làm gì cả, ngược lại **NDEBUG** không định nghĩa thì trình biên dịch sẽ kiểm tra tham số **condition**, so sánh với zero.

Nếu biểu thức **condition =** 0: assert sẽ xuất ra nội dung Marco \_\_FILE\_\_, \_\_LINE\_\_, và \_\_func\_\_ trên error và gọi **std::abort**

Ví dụ: xác thực index trong mảng

#include <array>

#include <cassert> // for assert()

int getArrayValue(const std::array<int, 10> &array, int index)

{

// xác thực index nằm giữa 0 và 9

assert(index >= 0 && index <= 9);

return array[index];

}

Nếu gọi GetArrayValue(-3), ứng dụng sẽ xuất tin nhắn:

Assertion failed: index >= 0 && index <=9, file C:\\VCProjects\\Test.cpp, line 6

Chúng ta nên dùng lệnh assert tại bất kì vị trí nào cần thiết.

1. Tăng tính mô tả của assert

Một số biểu thức assert không đủ ý nghĩa khi xác thực thông tin:

assert(found);

Nếu assert này là một trigger, nó sẽ in ra:

Assertion failed: found, file C:\\VCProjects\\Test.cpp, line 34

Không có ý nghĩa gì cả, chúng ta cần thông tin chi tiết hơn thế.

Có một mẹo nhỏ là chèn thêm vào chuỗi C-style với phép AND trong assert:

assert(found && "Car could not be found in database");

Đây là kết quả:

Assertion failed: found && "Car could not be found in database", file C:\\VCProjects\\Test.cpp, line 34

1. NDEBUG

Hàm assert() tốn ít chi phí mỗi khi được kiểm tra. Ngài ra, assert không bao giờ nên đặt trong ngữ cảnh code đã thành phẩm.

Do đó, nhiều lâp trình viên chỉ kích hoạt nó trong chế độ build debug mà thôi. Phương pháp truyền thống khi debug trong C++ là định nghĩa macro NDEGUG.

#define NDEBUG

// tất cả hàm gọi assert() sẽ bị bỏ qua hết

Một số IDE thiết lập NDEBUG mặc định trong cấu hình project release. Ví dụ như Visual Studio, các macro sau được định nghĩa tại level project: WIN32; \_CONSOLE.

Nếu bạn đang dùng Visual Studio và muốn assert trigger trong khi build release thì phải xóa NDEBUG trong cấu hình.

Lưu ý là hàm exit() và assert() (nếu có trigger) ngắt ứng dụng ngay lập tức, mà không thực hiện bất kỳ hành vi cleanup nào (như ngắt kết nối, đóng file…). Do đó chúng nên dùng một cách thận trọng

1. Static\_assert

C++11 bổ sung thêm một kiểu xác thực khác gọi là static\_assert()

Khác với assert, vốn hoạt động khi runtime, static\_assert được thiết kế hoạt động trong thời điểm biên dịch, nếu điều kiện không thỏa thì trình biên dịch báo lỗi với dòng tin nhắn được in ra.

Ví dụ dùng static\_assert khi xác thực kích thước kiểu:

static\_assert(sizeof(long) == 8, "long must be 8 bytes");

static\_assert(sizeof(int) == 4, "int must be 4 bytes");

int main()

{

return 0;

}

Tin nhắn lỗi:

1>c:\consoleapplication1\main.cpp(19): error C2338: long must be 8 bytes

Vì static\_assert được tính toán bởi trình biên dịch, phần điều kiện của static\_assert phải có khả năng xác định được trong thời điểm biên dịch.

Cũng bởi vậy nên static\_assert có thể đặt ở bất kì vị trí nào trong code file, thậm chí là global.

Trên C++11, tin nhắn lỗi phải đặt trong tham số thứ 2, trên C++17 thì tham số tin nhắn trở thành tham số phụ

### Chuyển đổi kiểu chuỗi và số

1. String -> const char\*

string str;

const char \* c = str.c\_str();

1. String -> char\*

Khai báo hàm chuyển string -> char\*, dùng iterator cho lớp chứa string

#include <string>

using namespace std;

char\* strToChar(const string &src){

char \*des = new char[src.size() + 1];

copy(src.begin(), src.end(), des);

des[src.size()] = '\0'; //byte zero

return des;

}

1. Const char -> string

const char \*s = "Hello, World!";

std::string str(s);

Phải đảm bảo biến char không phải NULL.

1. Number ->string

Thư viện C++ strings stream <sstream>rất mạnh mẽ, bằng cách đa năng hóa toán tử (<<,>>) cho phép định dạng mọi kiểu dữ liệu I/O thành dạng chuỗi. 2 bước chuyển số sang chuỗi:

* Xuất giá trị số vào string stream (<<)
* Dùng phương thức .**str**() Lấy chuỗi ra từ string stream

#include <sstream>

int main() {

int Number = 123;

string Result;

ostringstream out;

out << Number;

Result = out.str(); //trả về chuỗi

}

Cách thứ hai là bạn dùng hàm (C++11)

**std::to\_String**(<kiểu\_số>)

để chuyển số sang chuỗi.

1. Number ->char

Dùng hàm **sprint**() cho cả C,C++. Hàm này nằm trong thư viện chuẩn <stdio.h>

int Number = 123; // number to convert

char Result[16]; // string which will contain the number

sprintf ( Result, "%d", Number ); // %d định dạng kiểu thập phân

1. C – stdlib : kiểu số sang char\* và string

Thư viện <stdlib>, <cstdlib> của C,C++ chứa nhiều hàm hỗ trợ chuyển đổi giữa số và chuỗi

* **itoa** : integer sang char
* **atoi** : char sang integer
* **atoll** : char sang long
* **atof** : char sang float
* **stol** : string sang long
* **stoul** : string sang unsigned long
* **stod** : string sang double
* **stof** : string sang float
* **stoi :** string sang int

### Kỹ thuật bit flags và bit mask

##### Bit flags

1. Khái niệm

Chúng ta cùng xét một biến dữ liệu có kích thước 1 byte gồm 8 bit sẽ đại diện cho 8 chân trị riêng lẻ (0 hoặc 1).

Các bit chân trị này gọi là bit flag. Các bit này không thể truy xuất trực tiếp, chúng ta phải dùng toán tử bitwise để bật, tắt hay truy vấn.

1. Định nghĩa bit

Theo ký hiệu hexa

#include <stdint.h>

// Định nghĩa 8 flags (these can represent whatever you want)

// Note: in C++11, better to use "uint8\_t"

const uint8\_t option1 = 0x01; // hex for 0000 0001

const uint8\_t option2 = 0x02; // hex for 0000 0010

const uint8\_t option3 = 0x04; // hex for 0000 0100

const uint8\_t option4 = 0x08; // hex for 0000 1000

const uint8\_t option5 = 0x10; // hex for 0001 0000

const uint8\_t option6 = 0x20; // hex for 0010 0000

const uint8\_t option7 = 0x40; // hex for 0100 0000

const uint8\_t option8 = 0x80; // hex for 1000 0000

// byte-size value to hold some combination of the above 8 options

uint8\_t myOption = 0; // hex for 0000 0000 các bit đều tắt

Hay để dễ hiểu hơn, ta dùng dịch bit trái để định nghĩa flag

const uint8\_t option1 = (1<<0); //for 0000 0001

const uint8\_t option2 = (1<<1); //for 0000 0010

const uint8\_t option3 = (1<<2); //for 0000 0100

const uint8\_t option4 = (1<<3); //for 0000 1000

const uint8\_t option5 = (1<<4); //for 0001 0000

const uint8\_t option6 = (1<<5); //for 0010 0000

const uint8\_t option7 = (1<<6); //for 0100 0000

const uint8\_t option8 = (1<<7); //for 1000 0000

Số bit dịch tương ứng với thứ tự bit được set.

1. Set bit

Để bật bit, ta dùng toán tử OR |:

myOption |= option4; // bật opt 4 on.

myOption |= option4 | option5; // bật opts 4 và 5 on.

1. Clear bit

Để tắt bit, ta dùng toán tử and & với toán tử đảo bit (~):

myOption &= ~option4; // bật opt 4 off

myOption &= ~option4 & ~option5; // bật opt 4 và 5 off.

1. Test bit

Để truy vấn trạng thái bit, ta dùng toán tử and &:

if (myOption & option4) {

// if option4 is set, do something

}

1. Toggle bit

Để bật/tắt một tùy chọn nào đó, ta dùng toán tử XOR:

myOption ^= option4; // đảo opt 4

myOption ^= option4 | option5; // đảo opt 4 và 5

Một ví dụ thực tế nữa là khi dùng bộ thư viện 3D OpenGL, vài hàm sẽ dùng một hay nhiều bit flags như là tham số:

glClear(**GL\_COLOR\_BUFFER\_BIT** | **GL\_DEPTH\_BUFFER\_BIT**); // clear the color and the depth buffer

**GL\_COLOR\_BUFFER\_BIT** và **GL\_DEPTH\_BUFFER\_BIT** được định nghĩa như sau (trong thư viện gl2.h):

#define GL\_DEPTH\_BUFFER\_BIT 0x00000100

#define GL\_STENCIL\_BUFFER\_BIT 0x00000400

#define GL\_COLOR\_BUFFER\_BIT 0x00004000

1. ****Khả năng tiết kiệm bộ nhớ****

Ví dụ trên phải dùng 9-bytes (8-bytes định nghĩa tùy chọn, 1-bytes cho bit flags). Nhưng khi bạn phải định nghĩa nhiều thuộc tính, bit flag giúp bạn tiết kiệm bộ nhớ rất nhiều.

Tưởng tượng một trò chơi có rất nhiều loại quái vật. Khi một con được tạo ra, nó có sức đề kháng trước vài kiểu đánh bao gồm: poison, lightning, fire, cold, theft, acid, paralysis và blindness.

Để biết kiểu đánh mà quái vật có sức đề kháng với, ta dùng một chân trị cho một đề kháng . Vậy là mỗi quái vậy có 8 chân trị.

#include <iostream>

#include <stdint.h>

using namespace std;

class monster{

private:

//khai báo sức đề kháng

const uint8\_t resistsPoison = 0x01;

const uint8\_t resistsLightning = 0x02;

const uint8\_t resistsFire = 0x04;

const uint8\_t resistsCold = 0x08;

const uint8\_t resistsTheft = 0x10;

const uint8\_t resistsAcid = 0x20;

const uint8\_t resistsParalysis = 0x40;

const uint8\_t resistsBlindness = 0x80;

//member function ...

};

Với 100 lớp, phải khai báo 800 biến chân trị tương đương 800-bytes. Nhưng nếu dùng bit flags:

#include <iostream>

#include <stdint.h>

using namespace std;

const uint8\_t resistsPoison = 0x01;

const uint8\_t resistsLightning = 0x02;

const uint8\_t resistsFire = 0x04;

const uint8\_t resistsCold = 0x08;

const uint8\_t resistsTheft = 0x10;

const uint8\_t resistsAcid = 0x20;

const uint8\_t resistsParalysis = 0x40;

const uint8\_t resistsBlindness = 0x80;

class snake\_monster{

private:

//khai báo sức đề kháng

const uint8\_t resistsAttribute = resistsPoison | resistsLightning | resistsCold;

//member function ...

};

Chúng ta chỉ cần 1-byte qui định sức đề kháng cho mỗi quái vật, 8-bytes định nghĩa thuộc tính và 100-bytes qui định sức đề kháng cho 100 lớp quái vật.

Trong hầu hết chương trình, lượng bộ nhớ tiết kiệm với bit flags sẽ không tăng độ phức tạp. Nhưng với chương trình có hàng ngàn hay hàng triệu đối tượng tương tự nhau, dùng bit flags giúp tối ưu hóa chi phí rất nhiều.

1. Tối ưu cho tham số và dễ đọc

Giả sử chúng ta định nghĩa một hàm với tham số có thể kết hợp 32 tùy chọn khác nhau:

void someFunction(bool option1, bool option2, …, bool option32)

Tham số hàm quá dài, bất tiện khi phải gọi hàm và khó đọc. Thay vào đó, bạn định nghĩa hàm dùng bit flags thế này:

void someFunction(unsigned int options);

Sau đó chọn các option cần kích hoạt:

someFunction(option10 | option32);

Đoạn mã trở nên dễ đọc hơn rất nhiều vì chỉ cần khai báo hai option, về sau có cần định nghĩa lại option thì cứ thêm vào cùng toán tử OR (|), không phải định nghĩa lại thủ tục hàm mới.

Cũng là lý do tại sao thư viện OpenGL chọn bit flags thay vì nhiều biến chân trị liên tiếp.

1. Giới thiệu std::bitset

Thư viện chuẩn C++ cung cấp lớp khuôn mẫu (class teamplate) std::bitset để quản lý bit flags.

Trước hết bạn phải khai báo thư viện **<bitset>,** sau đó định nghĩa biến std::bitset có bao nhiêu bit cần dùng. Số lượng bits phải có thời gian biên dịch cố định.

#include <bitset>

std::bitset<8> bits; // dùng 8 bits

Nếu muốn, bitset có thể khởi tạo với giá trị khai báo ban đầu:

#include <bitset>

std::bitset<8> bits(option1 | option2) ; // khởi tạo cho opt 1 và 2 bật

std::bitset<8> morebits(0x4) ; // Khởi tạo với dãy bit 0000 0100

std::bitset cung cấp 4 hàm chính:

* **test**() cho phép truy vấn bit là 0 hay 1
* **set**() cho phép bật bit (hoặc không làm gì nếu đã bật)
* **reset**() cho phép tắt bit (hoặc không làm gì nếu đã tắt)
* **flip**() cho phép đảo bit

Mỗi hàm trên đều có tham số chỉ vị trí bit cần thao tác. Vị trí bit cùng phải (bit trọng số thấp nhất LSB) là 0, càng về trái (bit trọng số cao hơn) thì tăng dần lên.

Mẹo ở đây là bạn nên đặt vị trí bit theo tên thuộc tính cho dễ nhớ.

#include <bitset>

#include <iostream>

// các hàm của std::bitset sẽ dùng số thứ tự bit

const int option\_1 = 0;

const int option\_2 = 1;

const int option\_3 = 2;

const int option\_4 = 3;

const int option\_5 = 4;

const int option\_6 = 5;

const int option\_7 = 6;

const int option\_8 = 7;

int main(){

std::bitset<8> bits(0x2); // we need 8 bits, start with bit pattern 0000 0010

bits.set(option\_5); // set bit 4 to 1 (now we have 0001 0010)

bits.flip(option\_6); // flip bit 5 (now we have 0011 0010)

bits.reset(option\_6); // set bit 5 back to 0 (now we have 0001 0010)

std::cout << "Bit 4 has value: " << bits.test(option\_5) << '\n';

std::cout << "Bit 5 has value: " << bits.test(option\_6) << '\n';

std::cout << "All the bits: " << bits << '\n';

return 0;

}

Kết quả:

Bit 4 has value: 1

Bit 5 has value: 0

All the bits: 00010010

Chú ý là gửi biến bitset vào std::cout sẽ in ra giá trị tất cả bits trong bitset.

Lớp mẫu std::bitset khá thuận tiên khi cần thao tác với bit flags, bạn nên dùng thường xuyên nhưng đừng lệ thuộc quá nhiều.

##### Mặt nạ bit

Khi các bit chân trị trong một biến đại diện cho một thuộc tính nào đó, thì được gọi là mặt nạ bit (bit mask).

Chương trình sau minh họa người dùng sẽ nhập một số, ta sẽ dùng mặt nạ bit giữ lại 4-bits thấp và in ra.

#include <stdint.h>

#include <iostream>

using namespace std;

int main()

{

const uint16\_t lowMask = 0xF; // bit mask to keep low 4 bits (hex for 0000 0000 0000 1111)

cout << "Enter an integer: ";

int num;

cin >> num;

num &= lowMask; // remove the high bits to leave only the low bits

cout << "The 4 low bits have value: " << num<< '\n';

return 0;

}

Đoạn code trên không thực tiễn lắm, nhưng điều quan trọng là chúng ta đã tùy chỉnh nhiều bit cùng lúc được.

Bây giờ chúng ta sẽ xem một minh họa phức tạp hơn về 3 kênh màu RGB của Ti vi. Mỗi kênh R, G, B có kích thước 8-bits (0-> 255), kênh có giá trị càng lớn thì cường độ màu càng cao trên điểm ảnh.

Ngoài 3 kênh, chúng ta còn kênh Alpha, qui định độ mờ cho màu, giá trị kênh A càng cao thì màu sắc càng mờ. Kênh A có kích thước 8-bits.

Thông thường chúng ta dùng 1 biến số nguyên 32-bits (mỗi kênh 8-bits) để qui định 4 kênh :

|  |  |  |  |
| --- | --- | --- | --- |
| 32-bit RGBA value | | | |
| bits 31-24 | bits 23-16 | bits 15-8 | bits 7-0 |
| red | green | blue | alpha |

Coding:

#include <cstdlib>

#include <iostream>

#include <stdint.h>

using namespace std;

//chúng ta khai báo cấu trúc màu RGB cho pixel

typedef struct rgba {

uint8\_t red;

uint8\_t green;

uint8\_t blue;

uint8\_t alpha;

} RGBA;

//khai báo các bit mask cho 4 kênh.

const uint32\_t redMask = 0x000000FF;

const uint32\_t greenMask = 0x0000FF00;

const uint32\_t blueMask = 0x00FF0000;

const uint32\_t alphaMask = 0xFF000000;

int main(int argc, char\*\* argv) {

RGBA pixel;

uint32\_t n, A, R, G, B;

cout << "Nhap so n bat ki" << endl;

cin >> hex>>n; // >>hex cho phép người dùng nhập kí tự hex

//dùng kỹ thuật con trỏ để gán giá trị từng field

uint8\_t \*p = (uint8\_t\*) & n;

pixel.red = \*p;

pixel.green = \*(p + 1);

pixel.blue = \*(p + 2);

pixel.alpha = \*(p + 3);

//dùng bit mask để xuất giá trị từng kênh, không dùng toán tử thành viên

R = redMask & \*(uint32\_t\*) p;

G = greenMask & \*(uint32\_t\*) p;

B = blueMask & \*(uint32\_t\*) p;

A = alphaMask & \*(uint32\_t\*) p;

//dich bit để lấy giá trị kênh cần xuất.

cout << "Kenh R: " << R << endl;

cout << "Kenh G: " << (G >> 8) << endl;

cout << "Kenh B: " << (B >> 16) << endl;

cout << "Kenh A: " << (A >> 24) << endl;

return 0;

}

Kết quả:

Nhap so n bat ki

FFFFFFFF

Kenh R: 255

Kenh G: 255

Kenh B: 255

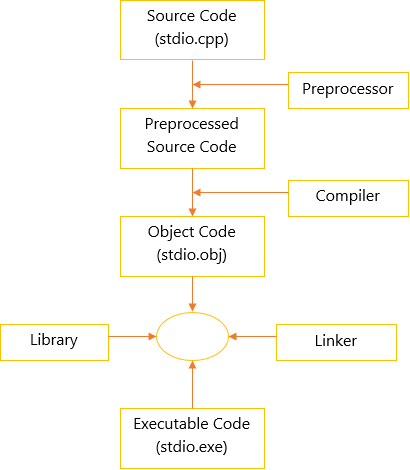
Kenh A: 255

### Chỉ thị tiền xử lí

Chỉ thị include (include directive) là thuật ngữ không hề xa lạ đối với những bạn bước đầu tìm hiểu lập trình và những lập trình viên trong quá trình sử dụng ngôn ngữ C/C++. Bài viết này mong đợi các bạn sẽ hiểu rõ hơn, sâu hơn về khái niệm cũng như cách sử dụng chỉ thị này. Qua đó, sẽ giúp các bạn sử dụng một cách linh hoạt, đồng thời tránh những sai lầm khi sử dụng.

##### Preprocessor là gì?

Bất cứ một file CPP hoặc C nào khi được biên dịch cũng sẽ trải qua các bước sau:

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_3/24/ss_1.png)

Preprocessor không khác gì các đoạn văn bản được định dạng đặc biệt. Cú pháp của chúng khác hẳn với C,C++.

Preprocessor tỏ ra có ích và được tích hợp vào trình biên dịch của C, C++. Trên hệ Unix, nó vẫn là một chương trình riêng lẻ, tự động thực thi bởi trình biên dịch đóng gói cc.

Ngày nay, đa số trình biên dịch đều được tích hợp sẵn bên trong.

Các preprocessor khác nhau sẽ thực hiện các nhiệm vụ khác nhau. Chúng ta có thể phân loại preprocessor như sau:

* Inclusion directives
* Macro Definition Directives
* Conditional Compilation Directives

**Mỗi lệnh preprocessor bắt đầu với #. Kí tự phía sau đó không được là khoảng trắng. Trước # có thể tồn tại các khoảng trắng và preprocessor chỉ kéo dài trên một dòng code.**

**Ngay sau khi một kí tự xuống dòng được phát hiện, các preprocessor kết thúc. Tuy nhiên, một directive có thể được tiếp tục bằng cách đặt dấu (\) cuối dòng.**

**Các preprocessor không phải là lệnh của ngôn ngữ C/C++ nên không kết thúc bằng dấu chấm phẩy.**

Ta có 1 số directive như sau:

|  |  |
| --- | --- |
| **Directive** | **Description** |
| #define | Khai báo macro thuộc bộ tiền xử lí |
| #include | Chèn nội dung header file tại vị trí khai báo |
| #undef | Hủy định nghĩa macro |
| #ifdef | Return True nếu macro đã được định nghĩa |
| #ifndef | Return True nếu macro chưa được định nghĩa |
| #if | Bắt đầu điều khiển #if của bộ tiền xử lí |
| #else | Rẽ nhánh thay thế cho #if |
| #elif | #else và #if rẽ nhánh điều kiện thay thế |
| #endif | Kết thúc điều khiển #if của bộ tiền xử lí |
| #error | Truyền tin nhắn lỗi vào **stderr** |
| #pragma | Dùng phương thức chuẩn để truyền 1 lệnh đặc biệt cho trình biên dịch. |

1. Giới thiệu một số Directive:
2. #define

#define MAX\_ARRAY\_LENGTH 20

Directive này khai báo cho CPP macro **MAX\_ARRAY\_LENGTH** = 20 . Trình biên dịch sẽ thay thế 20 tại mọi sự hiện diện của MAX\_ARRAY\_LENGTH trong đoạn code.

Dùng *#define* cho khai báo marco giúp mã nguồn dễ đọc hơn.

#undef FILE\_SIZE

#define FILE\_SIZE 42

Dòng 1 báo CPP to hủy marco FILE\_SIZE.

Dòng thứ 2 định nghĩa FILE\_SIZE là 42.

Ngoài ra bạn còn có thể định nghĩa như sau:

#define FOR\_ALL for (i = 0; i < ARRAY\_SIZE; ++i)

Và dùng để xóa mảng:

/\*Clear the array\*/

FOR\_ALL {

data[i] = 0;

}

Nhưng cách diễn đạt trên khá tối nghĩa, làm cấu trúc chương trình không rõ ràng. Người đọc phải dò lại định nghĩa macro để hiểu FOR\_ALL để làm gì.

1. #define và const

Từ khóa const được ưu dùng trong C,C++ hơn vì:

* Đầu tiên là C,C++ sẽ kiểm tra cú pháp của const ngay lập tức. #define không được kiểm tra cho tới khi Macro được dùng.
* Thứ hai, biến const sẽ tuân thủ quy tắc phạm vi của C,C++. Trong khi directive #define ở đâu cũng hiểu.

1. #include

#include <stdio.h>

#include "myheader.h"

Directives báo CPP lấy stdio.h từ **bộ thư viện hệ thống** và chèn nội dung header vào vị trí hiện tại. Dòng kế báo CPP lấy **myheader.h** từ thư mục hiện hành

1. #ifndef … #endif và #ifdef … #endif

Đoạn code báo CPP định nghĩa Macro **MESSAGE** nếu **MESSAGE** chưa được định nghĩa bao giờ.

#ifndef MESSAGE //if not define … then define

#define MESSAGE "You wish!"

#endif

Ngoài ra chỉ thị này còn dùng để ngăn các file header không bị #**include** trùng lặp trong project nhiều file

Nguyên tắc như sau:

#ifndef SOME\_HEADER\_HERE

#define SOME\_HEADER\_HERE

// phần khởi tạo nằm đây

#endif

Nếu header file được #**include** nhiều lần, trình biên dịch khi thấy Macro SOME\_HEADER\_HERE đã định nghĩa thì không nạp thêm code vào chương trình nữa.

1. #pragma once

Rất nhiều trình biên dịch hỗ trợ tiền xử lí **#pragma** , đơn giản hơn và thay thế cho **#ifndef …#define … #endif.**

**#pragma** đảm bảo các thư viện khai báo đằng sau nó chỉ được include duy nhất một lần

Ví dụ file stdafx.h của Visual studio nằm trong project có chứa chỉ thị này.

#pragma once

// your code here

#pragma comment(lib,"Ws2\_32.lib")

#include "targetver.h"

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <stdint.h>

#include <winsock2.h>

#include <ws2tcpip.h>

**#pragma** **once** làm việc giống như một chỉ thị quản lý header, ưu điểm ngắn gọn và ít lỗi. Dầu vậy, **#pragma one** không phải chỉ thị chính thức của C++, nên một số trình biên dịch không hỗ trợ. Và nếu có, bạn nên gắn nó vào đầu header.

1. Điều kiện biên dịch

Một vấn đề khi viết code là chương trình phải chạy đươc trên nhiều hệ máy như Unix, MsDos và Windows.

Preprocessor cho phép bạn linh động trong mã lệnh tạo ra thông qua điều kiện biên dịch.

Giả sử bạn cần đặt một đoạn code debugging khi đang phát triển và sẽ xóa khi phát hành:

Ở đầu chương trình, bạn khai báo directive

#define DEBUG /\*đoạn code debug\*/

Hủy định nghĩa:

#undef DEBUG

Phương pháp tốt nhất là dùng #ifdef. Báo CPP xử lí câu lệnh trong #**if** nếu hằng **DEBUG** đã được định nghĩa.

#ifdef DEBUG

/\* Your debugging statements here \*/

cout << “value need to debugging” << value;

#endif

Việc này khá hữu ích khi bạn dùng hằng **DEBUG** như 1 flag báo cho trình biên dịch khi bạn cần debug chương trình (bật flag) hay không (tắt flag).

1. Macros mặc định

ANSI C định nghĩa sẵn 1 lượng macros. Các macro này đều có thể sử dụng cho chương trình, những macros định nghĩa trước không cho phép tác động thay đổi giá trị

|  |  |
| --- | --- |
| **Macro** | **Description** |
| \_\_DATE\_\_ | ngày hiện tại theo dạng chuỗi "MMM DD YYYY" |
| \_\_TIME\_\_ | giờ hiện tại dạng chuỗi "HH:MM:SS" |
| \_\_FILE\_\_ | tên file hiện hành dạng chuỗi. |
| \_\_LINE\_\_ | số dòng hiện hành dạng hằng số thập phân |
| \_\_STDC\_\_ | = 1 khi biên dịch theo chuẩn ANSI |

Ví dụ: file test.c

#include <stdio.h>

main(){

printf("File :%s\n", \_\_FILE\_\_ );

printf("Date :%s\n", \_\_DATE\_\_ );

printf("Time :%s\n", \_\_TIME\_\_ );

printf("Line :%d\n", \_\_LINE\_\_ );

printf("ANSI :%d\n", \_\_STDC\_\_ );

}

Kết quả

File :test.c

Date :Jun 2 2012

Time :03:36:24

Line :8

ANSI :1

1. Macros có tham số

Một chức năng mạnh của CPP là khả năng mô phỏng hàm có dùng tham số macro

Macros có đối số phải khai báo bằng directive **#define**. Danh sách đối số nằm trong dấu ngoặc đơn và theo sau tên maco. Không đượ có khoảng trắng giữa tên macro và dấu ngoặc. Ví dụ:

#include <stdio.h>

#define MAX(x,y) (x > y ? x : y)

int main(void){

printf("Max between 20 and 10 is %d\n", MAX(10, 20));

return 0;

}

Max between 20 and 10 is 20

Ví dụ hàm tính bình phương.

int square(int x) {

return x \* x;

}

Khai báo lại bằng macro như sau:

#define square(x) ((x)\* (x))

Tốt nhất, tuân theo luật là luôn đặt tham số trong dấu ngoặc đơn.

Trên các hệ thống đơn giản, nên hạn chế sử dụng toán tử tăng (++), giảm (--), xem ví dụ sau:

#define SQR(x) ((x) \* (x))

main (){

int counter; /\* Counter for loop \*/

counter = 0;

while (counter < 5)

std::cout << "x: " << counter + 1

<< "\nx squared: " << SQR(++counter) << std::endl;

return (0);

}

Kết quả xuất ra không như mong đợi, SQR(++counter) sẽ được khai thành : ((++counter)\* (++counter)). Counter bị nhảy giá trị lên 2.

1. Toán tử #

Toán tử ('#') được dùng bên trong marco có tham số, giúp convert tham số của macro thành 1 chuỗi. Chỉ được dùng khi macro có danh sách tham số.

Ví dụ:

#include <stdio.h>

#define message\_for(a, b) printf(#a " and " #b ": We love you!\n")

int main(void){

message\_for(Carole, Debra);

return 0;

}

Kết quả:

Carole and Debra: We love you!

1. Toán tử \

Thông thường khai báo macro chỉ nằm trên 1 line. Trường hợp khai báo quá dài, bạn có thể dùng toán tử nối “\”.

Ví dụ:

#define message\_for(a, b) \

printf(#a " and " #b ": We love you!\n")

1. Toán tử ##

Toán tử (##) trong khai báo macro để kết hợp hai đối số. Nó cho phép hai tokens riêng biệt chập lại thành 1 token đơn lẻ.

Ví dụ:

#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void){

int token34 = 40;

tokenpaster(34);

return 0;

}

Kết quả:

token34 = 40

Kết quả từ ví dụ được xuất ra từ bộ tiền xử lí như sau:

printf ("token34 = %d", token34);

Đoạn code trên biểu diễn cách nối 2 token: token và n thành token34 , vừa dùng stringize và tokenpasting.

1. Toán tử defined()

Toán tử **defined** được dùng để xác định macros đã được định nghĩa hay chưa. Return true (non-zero) hay false (zero)

Ví dụ:

#include <stdio.h>

#if !defined (MESSAGE)

#define MESSAGE "You wish!"

#endif

int main(void){

printf("Here is the message: %s\n", MESSAGE);

return 0;

}

Kết quả:

Here is the message: You wish!

1. Macro tham số và hàm inline:

Trong hầu hết trường hợp, chúng ta sẽ dùng hàm inline thay vì marco tham số hóa, để tránh các bẫy gây ra bởi loại marco này.

Ví dụ về marco SQR và inline function cùng tính bình phương một số:

#define SQR(x) ((x) \* (x)) // sẽ làm việc nhưng nguy hiểm

// Inline function cùng chức năng

inline int sqr(const int x) {

return (x \* x);

}

##### #include directive là gì?

Một trong những ứng dụng phổ biến nhất của preprocessor đó là Inclusion directive. Thể loại này chỉ có một chỉ thị, đó là #include.

Trước khi vào giai đoạn Compile, chỉ thị #include chúng ta sử dụng sẽ chỉ thị cho bộ tiền xử lí tìm kiếm tập tin theo tên được ghi trong lệnh #include.

Nếu tìm thấy, bộ tiền xử lí sẽ nhận nội dung của tập tin đó và đặt chèn vào tập tin nguồn đang xét ngay đúng vị trí của #include xuất hiện.

Ngược lại, nếu không tìm thấy tập tin, chương trình sẽ thông báo lỗi:

**Error C1083: Cannot open include file**

1. Cú pháp Include

#include <file>

#include "file"

Bạn có thể thêm vào trước các thư mục vào danh sách này với tùy chọn -I trong khi biên dịch mã nguồn của bạn .

1. Nguyên lý hoạt động

Khi #include được viết với dấu ngoặc nhọn có nghĩa chúng ta đã cung cấp hướng dẫn cho bộ tiền xử lí tìm kiếm tập tin theo một đường dẫn được xác định trong các  thư mục include được trình biên dịch thiết lập, chẳng hạn như các thư viện chuẩn của C++( iostream, string…).

Còn khi #include được viết với dấu ngoặc kép, con đường tìm kiếm của bộ tiền xử lí được mở rộng, bao gồm cả các thư mục hiện hành.

Trình biên dịch C/C++ và môi trường lập trình đều có một cơ sở cho phép lập trình viên xác định nơi bao gồm các tập tin có thể được tìm thấy.

Trong trường hợp, tập tin không được tìm thấy ở các thư mục hiện hành thì các thư mục include của trình biên dich sẽ được kiểm tra, hay các bạn có thể xem như dấu ngoặc kép được thay thế bởi các dấu ngoặc nhọn <...>.

Ví dụ: ta có file header.h

char \*test (void);

program.c khai báo include như sau:

int x;

#include "header.h"

int main (void){

puts (test ());

}

Trình biên dịch sẽ hiểu như sau:

int x;

char \*test (void);

int main (void){

puts (test ());

}

1. Header bị include lặp?

Nếu header được include 2 lần, trình biên dịch sẽ báo lỗi. Cách thức tránh tình trạng này là rào toàn bộ nội dung header file trong một mệnh đề điều kiện như sau

#ifndef HEADER\_FILE

#define HEADER\_FILE

the entire header file file

#endif

Ý tưởng là rào nội dung trong bộ chứa #ifndef . Khi header bị include đúp, mệnh đề điều kiện trả về false vì HEADER\_FILE đã định nghĩa rồi.

Bộ tiền xử lí sẽ bỏ qua toàn nội dung header file.

1. Includes header nào?

Thỉnh thoảng bạn phải dắn đo include header nào vào chương trình cho phù hợp với hệ điều hành (Window hay UNIX)

C cung cấp cho bạn phương thức định nghĩa tham số cấu hình cho phù hợp hệ điều hành. Ví dụ:

#if SYSTEM\_1

# include "system\_1.h"

#elif SYSTEM\_2

# include "system\_2.h"

#elif SYSTEM\_3

...

#endif

Cho đỡ tẻ nhạt, bạn vẫn có thể khai báo macros với #include directive như sau:

#define SYSTEM\_H "system\_1.h"

...

#include SYSTEM\_H

SYSTEM\_H sẽ được triển khai chi tiết ra, bộ tiền xử lí sẽ tìm kiếm header system\_1.h.

SYSTEM\_H nên được định nghĩa bằng Makefile với tùy chọn -D .

##### Variadic Macros

Một macro có thể khai báo nhận số lượng tham số không giới hạn tương tự như hàm. Ví dụ:

#define eprintf(…) fprintf (stderr, \_\_VA\_ARGS\_\_)

Khi marco được gọi, tất cả tokens trong danh sách đối số, sau tham số cuối cùng (trường hợp này thì không có), gọi là tham số tùy ý.

Chuỗi tokens được thay thế bởi từ khóa \_\_VA\_ARGS\_\_ , chúng ta có phiển bản mở rộng:

eprintf ("%s:%d: ", input\_file, lineno)

→ fprintf (stderr, "%s:%d: ", input\_file, lineno)

Tham số tùy ý được mở rộng trước khi được chèn vào marco ở hình thái mở rộng. Chúng ta có thể dùng toán tử # và ## để stringize tham số tùy ý hay dán nó trước hay sau token khác.

Nhưng nếu cần dùng một macro phức tạp hơn, bạn muốn một cái tên có nghĩa hơn cho tham số tùy ý hơn là \_\_VA\_ARGS\_\_, phần mở rộng C/C++ cho phép bạn viết tên đại diện trước từ khóa “…”. Ví dụ:

#define eprintf(args…) fprintf (stderr, args)

Ngoài ra có thể khai báo thêm tham số, định nghĩa eprintf như sau:

#define eprintf(format, …) fprintf (stderr, format, \_\_VA\_ARGS\_\_)

Triển khai trên chưa linh hoạt, vì phải cung cấp ít nhất một tham số trước tham số tùy ý. Trên standard C, có thể bỏ qua dấu phẩy phân cách, ngoài ra nếu bạn để trống tham số tùy, nhưng phải có dấu phẩy để trình biên dịch không báo lỗi

eprintf("success!\n", );

→ fprintf(stderr, "success!\n", );

GNU C++ có 2 phiên bản mở rộng giúp giải quyết vấn đề.

Thứ nhất, bạn được phép bỏ qua tham số tùy ý hoàn toàn:

eprintf ("success!\n")

→ fprintf(stderr, "success!\n", );

Thứ hai, toán tử dán token ## có tác dụng đặc biệt khi đặt giữa dấy phẩy và tham số tùy ý. Nếu viết

#define eprintf(format, …) fprintf (stderr, format, ##\_\_VA\_ARGS\_\_)

Tham số tùy ý sẽ bị bỏ qua khi dùng macro eprintf, dấy phẩy trước ## sẽ bị xóa.

Điều này sẽ không xảy ra nếu truyền tham số trống hay token trước ## không phải dấu phẩy

eprintf ("success!\n")

→ fprintf(stderr, "success!\n");

Cách diễn giải trên khá mơ hồ nếu trường hợp marco chỉ có tham số tùy ý, vì nó vô nghĩa khi phân biệt xem không có tham số nào trống hay bị bỏ qua.

C++ giữ lại dấu phẩy cho phù hợp với cú pháp C.

Standard C chỉ chấp nhận từ khóa \_\_VA\_ARGS\_\_ cho macro tham số tùy ý. Chúng ta không dùng từ khóa này để đặt tên cho macro, tên tham số trong macro hay cho các mục đích, ngữ cảnh khác nhau.

Macro tham số tùy ý được đưa vào C99, GNU CPP và Visual C++ hỗ trợ chúng với cách đặt tên (‘arg…’, not’…’ và \_\_VA\_ARGS\_\_).

### Hàm có tham số tùy ý trong C,C++

Nếu để ý, bạn sẽ thấy hàm printf, scanf và một số hàm khác trong ngôn ngữ C có tập danh sách tham số khá kì lạ. Số lượng tham số của các hàm này là không giới hạn, cả về kiểu dữ liệu lẫn số lượng tham số. Trong một số trường hợp, các hàm có dạng như thế tỏ ra khá hiệu quả trong việc xử lý dữ liệu. Mặc dù chúng ta hoàn toàn có thể dùng một số cách thay thế, nhưng code sẽ không đẹp và trông rối mắt hơn. Bài viết này sẽ giới thiệu đến các bạn cách để hiện thực những hàm tương tự như thế trong ngôn ngữ C.

1. Yêu cầu

Đối với hàm có danh sách tham số tuỳ ý, các yêu cầu sau phải được thoả mãn:

* Kiểu dữ liệu của một hoặc một vài tham số đầu tiên phải rõ ràng. Kiểu dữ liệu có thể là các kiểu dữ liệu cơ bản hoặc do người dùng tự định nghĩa trước khi xây dựng hàm.
* Các tham số tiếp theo được thể hiện bằng dấu ba chấm **“…”.** Số lượng tham số là tuỳ ý, do đó phụ thuộc vào tham số đầu tiên mà chúng ta sẽ xử lý các tham số còn lại.

Cách làm trên là phổ biến, dễ hiện thực và có tính tái sử dụng cao. Tuy nhiên trong một số trường hợp, ta có thể không sử dụng tham số đầu tiên mà dùng một phương pháp nào đó, miễn là có thể đánh dấu và nhận biết thời điểm kết thúc danh sách tham số.

1. Xây dựng hàm

Trước tiên, để xây dựng hàm có danh sách tham số tuỳ ý, ta cần khai báo thư viện [**<stdarg.h>**](#_Stdarg.h_(_C)trong C, và <cstdarg> trong C++

Nguyên tắc xây dựng hàm như sau:

1. Định nghĩa một hàm với tham số cuối cùng là dấu ba chấm **“…”.** Một hoặc các tham số trước đó phải có dấu hiệu cho biết việc kết thúc của chuỗi tham số.
2. Khởi tạo biến **va\_list**. Biến này được định nghĩa sẵn trong thư viện <**stdarg.h**>, dùng để lưu trữ danh sách tham số đầu vào.
3. Sử dụng macro **va\_start** nhận tham số là tên biến nằm trong danh sách tham số. Khai báo này để nhận biết bắt đầu từ biến nào sẽ nằm trong **va\_list**
4. Sử dụng macro **va\_arg** nhận tham số **va\_list** và kiểu dữ liệu để trích xuất từng tham số.
5. Sử dụng macro **va\_end** để giải phóng bộ nhớ đã cấp phát cho **va\_list.**
6. Trường hợp C

Minh hoạ cho hàm tìm số lớn nhất trong các số nguyên. Đối số đầu tiên truyền vào hàm là số lượng các số nguyên sẽ truyền vào.

#include <stdarg.h>

#include <stdio.h>

int FindMax(int \_count, ...){

va\_list ap; //khởi tạo đối tượng va\_list

va\_start(ap, \_count); //Khởi tạo ap gồm \_count phần tử

int greater = va\_arg(ap, int);

for(int i = 1; i < \_count; i++) {

int val = va\_arg(ap, int);

greater = (greater > val) ? greater : val;

}

va\_end(ap); //Hủy danh sách

return greater;

}

int main(){

int m = FindMax(7, 489, 2258, 5622, 7411, 892, 1045, 6081);

printf("The greatest one is : %d\n", m);

return 0;

}

Minh hoạ việc in các chuỗi kí tự ra màn hình. Điều kiện nhận biết là kí tự NULL ở tham số cuối cùng.

void PrintLines(char\* \_firstStr, ...){

char\* str;

va\_list ap;

va\_start(ap, \_firstStr);

str = \_firstStr;

while(str != NULL) {

printf("%s\n", str);

str = va\_arg(ap, char\*);

}

va\_end(ap);

}

int main(){

PrintLines("STDIO", "Rye", "Welcome", NULL);

return 0;

}

1. Trường hợp C++

#include <cstdarg>

#include <iostream>

// Hàm tính trung bình cho num số

double average ( int num, ... ){

va\_list arguments; // khai bao doi tuong chua tham so

double sum = 0;

va\_start(arguments, num); // Khoi tao doi tuong cho num tham so

for ( int x = 0; x < num; x++ ) // Loop tinh sum num tham so trong danh sach

sum += va\_arg ( arguments, double );

va\_end (arguments); // Huy danh sach

return sum / num;

}

### Xử lí ngoại lệ C, C++

##### C - Error Handling

**errno** là biến toàn cục và giúp chỉ ra lỗi nào trong quá trình gọi hàm. Tham khảo danh sách lỗi code định nghĩa trong [<errno.h>](#_Errno.h_(_C)

C không cung cấp trực tiếp công cụ kiểm soát lỗi, nó vẫn cho bạn truy xuất ở level thấp dưới dạng giá trị trả về của **errno**.

Ngầm định các hàm của C đều trả về “-1” hoặc “*Null”* trong trường hợp xảy ra lỗi khi gọi hàm.

Khi này macros **errno** được gán giá trị tương ứng với lỗi**. Errno =** 0 là không có lỗi gì

Như vậy lập trình viên có thể kiểm tra giá trị trả về và có những hành động phù hợp sau đó.

Lập trình viên nên set **errno** = 0 ở đầu chương trình

1. errno, perror() và strerror()

C cung cấp 2 hàm **perror()** <stdio.h> and **strerror()**  <string.h> để hiển thị tin nhắn tương ứng với **errno**.

* **perror()** hiển thị chuỗi kí tự bạn truyền tới nó, theo sau bởi dấu hai chấm, khoảng trắng và sau đó là tham chiếu tới giá trị **errno** hiện tại .
* **strerror()** trả về con trỏ tới giá trị của **errno** hiện tại

Xem ví dụ mở 1 file không tồn tại trong hệ thống. Ví dụ sẽ biểu diễn cách sử dụng cả 2 hàm trên.

Điểm quan trong thứ 2 là khi bạn thao tác với **FILE** nên dùng **stderr** file stream để xuất lỗi ra màn hình.

#include <stdio.h>

#include <errno.h>

#include <string.h>

extern int errno ;

int main ()

{

FILE \* pf;

int errnum;

pf = fopen ("unexist.txt", "rb"); // Mở file ở chế độ binary read-only

if (pf == NULL)

{

errnum = errno; // Lấy giá trị errno

fprintf(stderr, "Value of errno: %d\n", errno);

perror("Error printed by perror");

fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));

}

else

{

fclose (pf);

}

return 0;

}

Kết quả:

Value of errno: 2

Error printed by perror: No such file or directory

Error opening file: No such file or directory

1. Lỗi chia zêro

Một lỗi rất phổ biến là lập trình viên không check số chia có bằng zero hay không, kết cục là tạo ra một runtime error.

Đoạn code sau giúp kiểm tra số bị chia :

#include <stdio.h>

#include <stdlib.h>

main()

{

int dividend = 20;

int divisor = 0;

int quotient;

if( divisor == 0){

fprintf(stderr, "Division by zero! Exiting...\n");

exit(-1);

}

quotient = dividend / divisor;

fprintf(stderr, "Value of quotient : %d\n", quotient );

exit(0);

}

Kết quả:

Division by zero! Exiting...

1. Tham số trạng thái hàm Exit()

Tham số **EXIT\_SUCCESS** được dùng cho exit() sau khi chương trình chạy thành công. **EXIT\_SUCCESS** có giá trị cố định là 0

Nếu chương trình gặp lỗi và bạn cần thoát ra, bạn nên dùng hàm **exit**() với tham số status “**EXIT\_FAILURE**” (Giá trị cố định là -1). Ví dụ:

#include <stdio.h>

#include <stdlib.h>

Void main()

{

int dividend = 20;

int divisor = 5;

int quotient;

if( divisor == 0){

fprintf(stderr, "Division by zero! Exiting...\n");

exit(EXIT\_FAILURE);

}

quotient = dividend / divisor;

fprintf(stderr, "Value of quotient : %d\n", quotient );

exit(EXIT\_SUCCESS);

}

Kết quả

Value of quotient : 4

##### C++ - Error Handling

Mọi đoạn chương trình được viết ra đều tiềm ẩn khả năng sinh lỗi. Có thể là lỗi chủ quan do lập trình sai hoặc có thể là lỗi khách quan do dữ liệu hay trạng thái của hệ thống (thiếu memory, mạng bị corrupt …).

Thật khó để kiểm soát tất cả trường hợp của chương trình chúng ta viết ra, đặc biệt các lỗi liên quan đến số học, bộ nhớ, … Vì thế, việc giải quyết các ngoại lệ để ứng dụng chúng ta tránh được những rủi ro là 1 điều không kém phần quan trọng.

1. Tổng quan

Exception – Ngoại lệ là cơ chế thông báo và xử lý những vấn đề tiềm ẩn phát sinh (giống như lỗi runtime) trong chương trình bằng cách chuyển quyền điều khiển đến những hàm đặc biệt nhằm tách phần xử lý lỗi ra khỏi thuật toán chính.

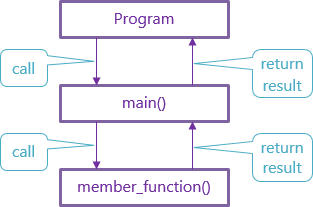
Một ngoại lệ là một đối tượng chứa thông tin về lỗi và truyền thông tin của lỗi cho bộ phận xử lý để có những hướng giải quyết phù hợp.

Ngoại lệ có thể thuộc bất kì kiểu dữ liệu bất kỳ của C++:

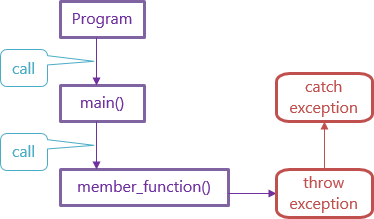
* Có sẵn như int, char, int\* , char\* ,…
* Kiểu người dùng tự định nghĩa
* Các lớp ngoại lệ trong thư viện <exception>

1. Cơ chế của ngoại lệ
2. Cơ chế

Quy trình gọi hàm và trả về trong trường hợp bình thường:



Quy trình ném và bắt ngoại lệ tại member\_function():



Cơ chế xử lý ngoại lệ của C++ có 3 tính năng chính:

* Khả năng tạo và ném ngoại lệ (sử dụng từ khóa throw).
* Khả năng bắt và giải quyết ngoại lệ (sử dụng từ khóa catch).
* Khả năng tách phần xử lý ngoại lệ ra khỏi phần có thể sinh lỗi (sử dụng từ khóa try).

1. Quăng ngoại lệ

Khi có lỗi xảy ra khiến tác vụ không thể hoàn thành, bạn dùng **throw** để thông báo lại

throw -1; // quăng ra giá trị integer

throw ENUM\_INVALID\_INDEX; // quăng enum

throw "Can not take square root of negative number"; // quăng chuỗi

throw dX; // thậm chí là biến

throw MyException("Fatal Error"); // Hay đối tượng exception

1. Bắt ngoại lệ

Muốn bắt exception tại vị trí nào của chương trình thì ta phải đặt tại đó 1 đoạn code nhằm kiểm tra ngoại lệ. Đoạn chương trình này được đóng gói trong 1 khối try và catch.

Ví dụ đoạn chương trình sau:

// Exceptions

#include <iostream>

#include <string>

using namespace std;

int main(){

Try{

If (no\_error) do\_job

else throw string("String!");

}

catch (string ex) {

// exception handler }

catch(...)

{ // exception handler

}

return 0;

}

Nếu xảy ra lỗi hay 1 ngoại lệ (throw an exception) được ném ra – bằng cách sử dụng từ khóa throw, chương trình sẽ nhảy tới cuối khối try.

Song, nếu không xảy ra bất cứ ngoại lệ nào thì chương trình được build bình thường và các bộ xử lý đều được bỏ qua.

1. Cơ chế xử lí ngoại lệ

Khi có lỗi trong quá trình thực thi, ngoại lệ bị bắt (catch the exception) hay được ném (throw),..

Khi tìm thấy biểu thức catch() có kiểu tham số phù hợp với đối tượng exception, nó được truyền đến phần xử lý ngoại lệ (exception handler). Nếu chưa tìm thấy nó sẽ ngảy tiếp tới khối catch phía sau nếu có.

Nếu không tìm thấy biểu thức catch() nào phù hợp , chương trình sẽ báo lỗi exception error và kết thúc.

1. Ngoại lệ trong hàm

Xét ví dụ:

#include "math.h" // for sqrt() function

using namespace std;

double MySqrt(double dX){

if (dX < 0.0) throw "Can not take sqrt of negative number";

return sqrt(dX);

}

int main(){

double dX = -5.0;

try {

//...

}

catch (const char\* strException) {// catch exceptions kieu const char\*

cerr << "Error: " << strException << endl;

}

}

Kết quả:

Error: Can not take sqrt of negative number

Dù ngoại lệ không được xử lí hàm MySqrt(), nhưng hàm main vẫn bắt được ngoại lệ.

Theo cơ chế xử lí ngoại lệ, chương trình sẽ tìm tới cuối khối try, nó nhảy ra khỏi hàm MySqrt(), quay lại hàm main, nó kiểm tra và tìm thấy khối try catch.

Quy trình này gọi là **unwinding the stack.**

1. Bắt tất cả ngoại lệ

Do không thể nắm hết được các kiểu ngoại lệ có thể xảy ra, C++ cung cấp cú pháp

catch(…){

//exception handler

}

Tránh để lọt ngoại lệ, chúng ta thường đặt vào cuối chuỗi biểu thức catch

try{

throw 5; // throw an int exception

}

catch (double dX){

cout << "We caught an exception of type double: " << dX << endl;

}

catch (...) // catch-all handler{

cout << "We caught an exception of an undetermined type" << endl;

}

Và tốt nhất là đặt trong hàm main để kiểm soát tốt chương trình:

int main(){

try {

RunGame();

}

catch(...){

cerr << "Abnormal termination" << endl;

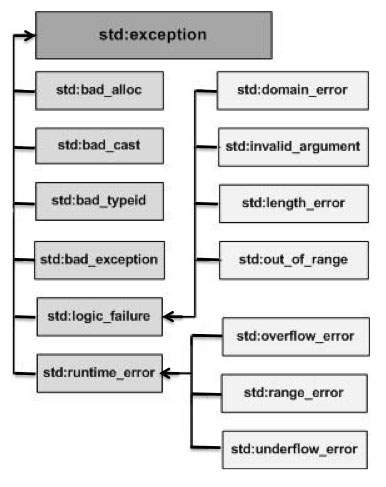
}

return 1;

}

1. C++ Standard Exceptions

Các đối tượng ngoại lệ chuẩn C++ được định nghĩa trong thư viện <exception>, chúng được thiết kế theo cây phân cấp như hình dưới.



|  |  |
| --- | --- |
| **Ngoại lệ** | **Mô tả** |
| std::exception | An exception and parent class of all the standard C++ exceptions. |
| std::bad\_alloc | Quăng ra bởi hành vi cấp phát động |
| std::bad\_cast | Quăng ra bởi chuyển kiểu dynamic\_cast. |
| std::bad\_exception | Một kiểu ngoại lệ để xử lí các ngoại lệ không mong muốn trong chương trình |
| std::bad\_typeid | Quăng ra bởi typeid. |
| std::logic\_error | An exception that theoretically can be detected by reading the code. |
| std::domain\_error | This is an exception thrown when a mathematically invalid domain is used |
| std::invalid\_argument | Quăng ra khi đối số truyền không hợp lệ |
| std::length\_error | Quăng ra khi chuỗi std::string quá lớn được tạo |
| std::out\_of\_range | Quăng ra bởi phương thức truy xuất phần tử ngẫu nhiên std::vector and std::bitset<>::operator[](). |
| std::runtime\_error | An exception that theoretically can not be detected by reading the code. |
| std::overflow\_error | Lỗi tràn bộ đệm. |
| std::range\_error | Lỗi khi truyền giá trị cho phần tử. |
| std::underflow\_error | Nếu xảy ra trường hợp tính toán không xác định |

1. Thiết kế lớp ngoại lệ

Chúng ta định nghĩa mới bằng cách kế thừa và override các lớp ngoại lệ đã có. Ví dụ sau dùng lớp std::exeption để định nghĩa ngoại lệ mới

#include <iostream>

#include <exception>

using namespace std;

struct MyException : public exception{

const char \* what () const {

return "C++ Exception";

}

};

int main(){

try {

throw MyException();

} catch(MyException& e) {

std::cout << "MyException caught" << std::endl;

std::cout << e.what() << std::endl;

}

catch(std::exception& e) {

//Other errors

}

}

MyException caught

C++ Exception

Ở đây, what() là phương thức virtual public phải được override bởi các lớp dẫn xuất từ exception, nó trả về mô tả của ngoại lệ.

##### Hàm noexcept

Một số hàm không ném ra ngoại lệ, 1 số thì không nên: và ta gán từ khóa **noexcept** cho hàm đó

double compute(double) noexcept; // không nên ném exception

throw() đã bị loại bỏ từ C++11, thay vào đó là noexcept.

Khai báo noexcept, nó mang ý nghĩa rằng lập trình viên không cần bận tâm khối try..catch trong hàm và để trình biên dịch tối ưu hóa, không lo kiểm soát path code từ ngoại lệ. Chỉ nếu thực sự nó sẽ không xảy ra ngoại lệ nào.

Hàm noexcept hoàn toàn không bị kiểm tra bởi complier và linker, nhưng nếu lập trình viên đang lừa nó khi vẫn có thể xảy ra ngoại lệ trong hàm trước khi trả về thì sao? Xem ví dụ:

double compute(double x) noexcept{

string s = "Courtney and Anya";

vector<double> tmp(10);

// ...

}

Nếu vector thất bại trong việc cấp phát, nó sẽ quang ra ngoại lệ **std::bad\_alloc**, và gọi tới **std::terminal()** khiến chương trình bị dừng đột ngột. Nó không gọi hàm hủy trong hàm.

Trường hợp này, chúng ta đơn thuần chỉ là đánh dấu hàm đó chưa sẵn sàng để xử lí bất kì ngoại lệ nào ném ra.

##### Toán tử noexcept

Chúng ta có thể định nghĩa hàm noexcept nhưng có điều kiện kèm theo

template<typename T>

void my\_fct(T& x) noexcept(Is\_pod<T>());//my\_fct có thể không ném exception nếu Is\_pod<T> = true

my\_fct vẫn có thể ném exception nếu Is\_pod = false. Mục đích là muốn kiểm tra xem **x** có phải kiểu **pod** thì my\_fct sẽ không ném, ngược lại nếu là kiểu **string hay int..** sẽ có ngoại lệ.

Tham số trong noexcept() bắt buộc là biểu thức hằng. Mặc định noexcept nghĩa là noexcept(true).

Thư viện chuẩn cho phép biểu thức tham số tùy biến, đa năng. Ví dụ ta muốn biết việc gọi hàm f(x) bên trong có ném ra ngoại lệ?

void call\_f(vector<T>& v) noexcept(noexcept(f(v[0])){

for (auto x : v)

f(x);

}

Noexcept sẽ nhận chính nó và tham số f cho phần tử zero của vector v làm biểu thức, trả về true nếu trình biên dịch biết nó không thể xảy ra ngoại lệ trong hàm f() và ngược lại là false.

Tuy nhiên, chúng ta không được dùng noexcept(exp) để kiểm tra xem exp có ném ra ngoại lệ hay không. Trình biên dịch đơn giản chỉ nhìn vào mọi toán tử noexcept trong exp có giá trị true thì sẽ trả về true. Nó không đi sâu vào phần định nghĩa của hàm.

Tùy hoàn cảnh mà noexcept và toán tử noexcept() quan trọng trong các thao tác trên container:

template<class T, size\_t N>

void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(∗a, ∗b)))

### Date và Time trong C,C++

1. Kiểu dữ liệu và phương thức

Thư viện chuẩn của C++ không có kiểu date thích hợp nên phải lấy từ C. Ta khai báo thư viện **<ctime>** đầu chương trình hoặc trong header file.

Có 4 kiểu thời gian là :

* **clock\_t:**
* **time\_t:**
* **size\_t**:

Kiểu **clock\_t, size\_t và time\_t** có khả năng trừu tượng hóa thời gian hệ thống và ngày tháng, có dạng là một số kiểu số nguyên dương (unsigned int)

* **tm.** Kiểu cấu trúc **tm** định nghĩa ngày giờ trong C có các thành viên dữ liệu như sau:

struct tm {

int tm\_sec; // seconds of minutes from 0 to 59

int tm\_min; // minutes of hour from 0 to 59

int tm\_hour; // hours of day from 0 to 24

int tm\_mday; // day of month from 1 to 31

int tm\_mon; // month of year from 0 to 11

int tm\_year; // year since 1900

int tm\_wday; // days since sunday

int tm\_yday; // days since January 1st

int tm\_isdst; // hours of daylight savings time

}

Các hàm quan trọng trong library để làm việc với cấu trúc ngày tháng cho C, C++

|  |  |
| --- | --- |
| **SN** | **Function & Purpose** |
| 1 | **time\_t time(time\_t \*time);**  Trả về lịch hiện tại của hệ thống **theo giây** bắt đầu từ ngày 1-jan-1970.  Nếu hệ thống không có thời gian, sẽ trả về “1” |
| 2 | **Char \*ctime(const time\_t \*time);**  Trả về con trỏ kiểu chuỗi dạng: *day month year hours:minutes:seconds year\n\0*. |
| 3 | **Struct tm \*localtime(const time\_t \*time);**  Trả về ngày giờ của địa phương theo kiểu cấu trúc **tm** |
| 4 | **clock\_t clock(void);**  Trả về giá trị gần đúng thời gian chương trình chạy. Trả về “1” nếu hệ thống không có thời gian |
| 5 | **char \* asctime ( const struct tm \* time );**  Trả về con trỏ kiểu chuỗi là thông tin chứa trong con trỏ **tm** theo dạng: *day month date hours:minutes:seconds year\n\0* |
| 6 | **struct tm \*gmtime(const time\_t \*time);**  Trả về con trỏ kiểu cấu trúc **tm.** Thời gian được biểu diễn ở dạng UTC , bản chất vẫn là GMT  Coordinated Universal Time (UTC) Greenwich Mean Time (GMT). |
| 7 | **time\_t mktime(struct tm \*time);**  Trả về calendar-time tương đương với thời gian tìm thấy từ con trỏ cấu trúc |
| 8 | **double difftime ( time\_t time2, time\_t time1 );**  Trả về sai biệt theo giây giữa time1 và time2. |
| 9 | **size\_t strftime();**  Hàm được dùng để dịnh dạng ngày tháng theo nhiều kiểu |

1. Ngày giờ với time\_t:

Xem xét việc bạn cần tìm ngày giờ hệ thống hiện tại, theo cả giờ địa phương hay UTC.

Mặc định tham số có giá trị “0” của hàm **time(time\_t \*time)**  sẽ trả về giây hiện tại của hệ thống. Ví dụ:

#include <iostream>

#include <ctime>

using namespace std;

int main( ){

// current date/time based on current system

time\_t now = time(0);

// convert now to string form

char\* dt = ctime(&now);

cout << "The local date and time is: " << dt << endl;

// convert now to tm struct for UTC

tm \*gmtm = gmtime(&now);

//convert tm to string form in format *day month date hours:minutes:seconds year\n\0*

dt = asctime(gmtm);

cout << "The UTC date and time is:"<< dt << endl;

}

Kết quả:

The local date and time is: Sat Jan 8 20:07:41 2011

The UTC date and time is:Sun Jan 9 03:07:41 2011

1. Date format với tm:

Cấu trúc **tm** rất quan trọng khi làm việc với ngày giờ trong C,C++. Hầu hết các hàm liên quan tới thời gian đều dùng cấu trúc **tm**.

Ví dụ sau hiển thị nhiều dạng ngày giờ liên quan tới hàm và cấu trúc **tm** :

#include <iostream>

#include <ctime>

using namespace std;

int main( ){

// current date/time based on current system

time\_t now = time(0);

cout << "Number of sec since January 1,1970:" << now << endl;

struct tm \*ltm = localtime(&now);

// print various components of tm structure.

cout << "Year: "<< 1900 + ltm->tm\_year << endl;

cout << "Month: "<< 1 + ltm->tm\_mon<< endl;

cout << "Day: "<< ltm->tm\_mday << endl;

cout << "Time: " << 1 + ltm->tm\_hour << ":"

<< 1 + ltm->tm\_min << ":"

<< 1 + ltm->tm\_sec << endl;

}

Kết quả:

Number of sec since January 1, 1970:1294548238

Year: 2011

Month: 1

Day: 8

Time: 22: 44:59

1. Làm việc với clock\_t

Thông qua thư viện <time.h>, chúng ta khai báo:

clock\_t clock();

Hầu hết chương trình tính toán đều cần tính chính xác cao theo milliseconds, microseconds, hay thậm chí nanoseconds.

Hàm clock() trả về khoảng khắc ticks của đồng hồ hệ thống kể từ lúc process khởi động. Số ticks/giây được định nghĩa trong macro CLOCKS\_PER\_SEC . Giá trị marco thay đổi tùy theo platform phần cứng.

Win32 API cung cấp phương thức GetTickCount() trả về số milli giây trôi qua kể từ lúc OS khởi động. Nếu muốn portable code, ta có thể dùng clock().

#include<stdio.h>

#include<time.h>

void main(){

clock\_t ticks1,ticks2;

ticks1 = clock();

/\*do some work\*/

ticks2 = clock();

printf("Application running time <ms>: %lu\n",(ticks2-ticks1)/CLOCKS\_PER\_SEC);

return;

}

### C File I/O

**Lưu ý: Ở level thấp khi máy tính sẽ tự hiểu 1 kí tự (char) như 1 số nhị phân và xem kí tự đó như là dạng số nguyên không dấu *(int)*. Sau đó máy tính sẽ biểu diễn kí tự tương ứng với số nguyên này theo** [**bảng mã ASCII**](#_Bảng_mã_ASCII)

**Đó là lý do tại sao hàm đọc/ghi kí tự lại định nghĩa kiểu integer.**

Tất cả các hàm liên quan tới File I/O đều nằm trong thư viện [**<stdio.h>**](#_Stdio.h_(_C)

Kiểu FILE : chứa thông tin liên quan tới output steam, có thể là thực thể stream như **stderr, stdin,** and **stdout.**

Ngoài ra thư viện còn định nghĩa các con trỏ kiểu FILE như **stderr, stdin,** and **stdout** để kiểm soát error, input stream và output stream trong quá trình thao tác với File.

##### Mở Files

Bạn có thể dùng hàm **fopen()** để tạo mới hay mở file, hàm sẽ khởi tạo đối tượng **FILE,** để điều khiển stream.

Hàm trả về con trỏ kiểu File hoặc *Null* nếu gặp lỗi.

Cú pháp:

FILE \*fopen( const char \* filename, const char \* mode );

* **Filename** : Đường dẫn tới file.
* **Mode**: chế độ mở file.

Hàm trả về con trỏ FILE. Hoặc NULL nếu không thành công, biến toàn cục errno được set.

|  |  |
| --- | --- |
| **Mode** | **Description** |
| r | Mở file đã tồn tại nhưng chỉ đọc. Không tạo mới nếu chưa có. |
| w | Mở file text để ghi, nếu chưa có file thì tạo mới. Chương trình sẽ bắt đầu ghi vào vị trí đầu file. |
| a | Mỡ file text để ghi ở chế độ ghi chèn nội dung vào cuối file. Tạo mới file nếu chưa có. |
| r+ | Mở file text vừa để đọc, vừa để ghi. |
| w+ | Mở file text vừa để đọc, vừa để ghi.  Nó sẽ ghi đè nếu file đã có nội dung.Tạo mới nếu chưa có. |
| a+ | Mở file text vừa để đọc, vừa để ghi.  Nó chỉ được phép ghi chèn thêm nội dung vào cuối file. Tạo mới nếu chưa có. |

Nếu bạn dự định thao tác với file nhị phân, bạn nên để ý các chế độ truy xuất bên dưới đây

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

Trong môi trường visual studio, hàm fopen sẽ được thay thế bởi hàm **fopen\_s(FILE\*\* stream, char\* filename, char\* mod)**

Hàm return về giá trị kiểu **errno\_t** , 0 nếu thành công, khác 0 nếu gặp lỗi.

Ví dụ:

FILE \*stream; // declair as global so function can access this

int \_tmain(int argc, \_TCHAR\* argv[])

{

errno\_t err;

err = fopen\_s(&stream, fileName, mod);

//Check file is exist?

if(err != 0)

{

printf("Error in opening file");

}

fclose(stream);

return 0;

}

##### Đóng File

Dùng hàm **fclose**( ) để đóng file.

Cú pháp:

int fclose( FILE \*fp );

Hàm **fclose()** trả về “0” nếu thành công, hoặc **EOF** (*<0*) nếu có lỗi khi đóng file.

Thực ra, hàm này sẽ flush bất kì dữ liệu nào đang chờ trong bộ đệm để ghi vào file, sau đó đóng file và giải phóng bộ nhớ.

**EOF** (End of File) là một biến hằng, định nghĩa kiểu số thực âm (*<0*) trong **stdio.h**

##### Kiểm tra feof

Cú pháp:

int feof(FILE \*f );

Trả về false nếu chưa hết file, true nếu đã đọc hết file

##### Ghi File

1. Hàm fputc

Để ghi giá trị kí tự **c** vào output stream **fb.**

Cú pháp:

int fputc( int c, FILE \*fp );

Hàm này trả về kí tự đã ghi thành công (<>*Null*) hoặc **EOF** nếu gặp lỗi. *(<0)*

1. Hàm fputs

Để ghi chuỗi không giới hạn kí tự vào stream. Hàm sẽ ghi tới khi gặp kí tự kết chuỗi *‘\0’*

Cú pháp:

int fputs( const char \*s, FILE \*fp );

Hàm này cũng trả về giá trị dương *(>0)* nếu thành công, hoặc **EOF** nếubịlỗi *(<0)*

1. Hàm fprintf

Ghi chuỗi kí tự vào file theo cú pháp chuỗi định dạng **format**. Hàm này có tham số tùy biến như hàm **printf**()

int fprintf (FILE \*fp , const char \*format,…);

Ví dụ này tôi dùng đường dẫn thư mục **/tmp** , bạn nhớ tạo trước khi sử dụng

#include <stdio.h>

Void main(){

FILE \*fp;

fp = fopen("/tmp/test.txt", "w+");

fprintf(fp, "This is testing for fprintf...\n");

fputs("This is testing for fputs...\n", fp);

fclose(fp);

}

Đoạn code sẽ tạo file **test.txt** trong thư mục /tmp và ghi hai dòng vào file .

1. Hàm vfprintf

Cú pháp:

int vfprintf(FILE \*stream, const char \*format, va\_list arg)

Dùng để ghi chuỗi kí tự định dạng **format** của danh sách tham số **arg** vào **stream**

Ví dụ:

#include <stdio.h>

#include <stdarg.h>

void WriteFrmtd(FILE \*stream, char \*format, ...){

va\_list args;

va\_start(args, format);

vfprintf(stream, format, args);

va\_end(args);

}

int main (){

FILE \*fp;

fp = fopen("file.txt","w");

WriteFrmtd(fp, "This is just one argument %d \n", 10);

fclose(fp);

return(0);

}

##### Đọc File

1. Hàm fgetc()

Đọc một kí tự từ con trỏ FILE **fb**

int fgetc( FILE \* fp );

Trả về kí tự đọc được nếu thành công, hoặc **EOF** nếu bị lỗi

1. Hàm fgets()

Đọc 1 chuỗi dài n-1 kí tự từ stream **fb** vào bộ đệm bufvà chèn thêm byte kết zero “/0” để báo kết thúc chuỗi

char \*fgets( char \*buf, int n, FILE \*fp );

Hàm trả về con trỏ chuỗi nếu thành công, hoặc **EOF** khi đọc hết file.

Trong quá trình ghi vào bộ đệm, nếu gặp kí tự xuống dòng ‘\n” hàm sẽ trả về chuỗi tại vị trí “\n”. Hàm cũng sẽ trả về chuỗi khi đã đọc tới cuối file dù bộ đệm chưa đầy

Lưu ý khi sử dụng fgets để đọc file:

while(fgets(buff, max, fp)) {// Không dùng feof(), kiểm fgets trả về NULL

// process a line

}

1. Hàm fscanf()

Để đọc chuỗi kí tự từ file vào theo cú pháp chuỗi định dạng **format** nhưng nó dừng đọc sau khi gặp **whitespace**. Hàm này có tham số tùy biến như hàm **scanf**()

int \*fscanf(FILE \*fp, const char \*format,…);

Ví dụ:

#include <stdio.h>

Void main(){

FILE \*fp;

char buff[255]; // khai báo bộ đệm 255 bytes

fp = fopen("/tmp/test.txt", "r");

fscanf(fp, "%s", buff);

printf("1 : %s\n", buff );

fgets(buff, 255, (FILE\*)fp);

printf("2: %s\n", buff );

fgets(buff, 255, (FILE\*)fp);

printf("3: %s\n", buff );

fclose(fp);

}

Kết quả:

1 : This

2: is testing for fprintf...

3: This is testing for fputs...

Trong môi trường Visual Studio, hàm **fscanf()** sẽ được thay thế bằng hàm **fscanf\_s(FILE \*stream, char\* stringFormat, char\* buffer, size\_t size)**

Ví dụ:

char buffer[255]; // Must declair a buffer for reading text

FILE \*stream; // declair as global so function can access this

int \_tmain(int argc, \_TCHAR\* argv[])

{

if (openFile("text.txt", "r") < 0) break;

readByFscanf();

return 0;

}

void readByFscanf()

{

//read a string untill meet whitespace (tab, enter), this fucntion fscanf() not safe in case the line overhead the buffer.

while(fscanf\_s(stream,"%s", buffer, \_countof(buffer)) != EOF) printf("\n%s", buffer);

clearBuff();

closeFile();

}

##### Hàm Binary I/O

1. Hàm fread():

size\_t fread(void \*ptr, size\_t size,size\_t size\_of\_elements, FILE \*stream);

**ptr –** Con trỏ trỏ tới bộ đệm**.**

**size\_of\_elements –** Số phần tử**.**

**size–** Kích thước một phần tử theo byte.

**stream –** Con trỏ trỏ tới FILE input object**.**

Hàm trả về số phần tử đã đọc nếu thành công, nếu không bằng **size\_of\_elements** một là gặp lỗi, errno được set. Hai là đã đọc hết file, dùng **feof**(FILE \*stream) để kiểm tra.

1. Hàm fwrite():

size\_t fwrite(const void \*ptr, size\_t size, size\_t size\_of\_elements, FILE \*stream);

**ptr –**Con trỏ trỏ tới bộ đệm**.**

**size\_of\_elements–** Số phần tử**.**

**size–** Kích thước một phần tử theo byte**.**

**stream –** Con trỏ trỏ tới FILE output object**.**

Hàm trả về số phần tử đã ghi thành công, nếu không bằng **size\_of\_elements**, là bị lỗi, errno được set.

Ví dụ:

#include <stdio.h>

#include <string.h>

int main()

{

FILE \*fp;

char c[] = "this is tutorialspoint";

char buffer[100];

/\* Open file for both reading and writing \*/

fp = fopen("file.txt", "w+");

/\* Write data to the file \*/

fwrite(c, strlen(c) + 1, 1, fp);

/\* Seek to the beginning of the file \*/

fseek(fp, SEEK\_SET, 0);

/\* Read and display data \*/

fread(buffer, strlen(c)+1, 1, fp);

printf("%s\n", buffer);

fclose(fp);

return(0);

}

##### Thao tác vị trí trong File

1. Hàm fseek()

Thiết lập vị trí trong stream tại tham số khoảng cách **offset**

Cú pháp:

int fseek(FILE \*stream, long int offset, int whence)

* **stream** – Con trỏ File.
* **offset** – quy định số byte tính từ tham số vị trí whence.
* **whence** – Chỉ số vị trí sẽ được cộng bởi tham số offset. Gồm các hằng số sau:

|  |  |
| --- | --- |
| **Constant** | **Description** |
| SEEK\_SET | Vị trí đầu file |
| SEEK\_CUR | Vi trí hiện tại |
| SEEK\_END | Vị trí cuối file |

Hàm trả về 0 nếu thành công, != 0 nếu bị lỗi.

Ví dụ

#include <stdio.h>

int main ()

{

FILE \*fp;

fp = fopen("file.txt","w+");

fputs("This is tutorialspoint.com", fp);

fseek( fp, 7, SEEK\_SET );

fputs(" C Programming Language", fp);

fclose(fp);

return(0);

}

Input tiếp chuỗi sau " C Programming Language" vào vị trí stream cách 7 byte tính từ đầu file.

Nội dung file sẽ là:

This is C Programming Language

1. Hàm fsetpos() và fgetpos():

C định nghĩa con trỏ vị trí file kiểu **\*fpos\_t** .

Hàm **fsetpos()** sẽ thiết lập vị trí của **stream** với con trỏ vị trí **pos** truyền vào, tương tự chúng ta có hàm **fgetpos()** để lấy vị trí hiện tại của **stream** vào con trỏ **\*pos** truyền vào

Cú pháp

int fgetpos(FILE \*stream, fpos\_t \*pos)

* **stream** – Con trỏ File
* **pos** – Con trỏ vị trí thuộc đối tượng **fpos\_t object**.

int fsetpos(FILE \*stream, const fpos\_t \*pos)

Cả 2 hàm trả về 0 nếu thành công, != 0 nếu bị lỗi và đồng thời set chỉ số **errno** về giá trị lỗi tương ứng.

Ví dụ

#include <stdio.h>

int main ()

{

FILE \*fp;

fpos\_t position;

fp = fopen("file.txt","w+");

fgetpos(fp, &position); //trả về vị trí đầu file

fputs("Hello, World!", fp);

fsetpos(fp, &position); //đặt lại vị trí đầu file

fputs("This is going to override previous content", fp);

fclose(fp);

return(0);

}

Sau lần nhập chuỗi, chúng ta set lại vị trí đầu file với con trỏ **position** và ghi đè chuỗi thứ 2 vào.

Chúng ta sẽ có nội dung file như sau:

This is going to override previous content

1. Hàm ftell()

Trả về vị trí byte tính từ đầu file tới vị trí hiện tại của stream.

Cú pháp

long int ftell(FILE \*stream)

* **stream** – Con trỏ File.

Hàm trả về số byte. Trả về -1L nếu bị lỗi, đồng thời set giá trị biến global **errno.**

Ví dụ về lấy tổng byte của file

#include <stdio.h>

int main ()

{

FILE \*fp;

int len;

fp = fopen("file.txt", "r");

if( fp == NULL )

{

perror ("Error opening file");

return(-1);

}

fseek(fp, 0, SEEK\_END); //Đặt con trỏ vi trí về cuối file

len = ftell(fp); //Lấy số byte từ đầu file tới vi tri hiện tại

fclose(fp);

printf("Total size of file.txt = %d bytes\n", len);

return(0);

}

Kết quả

Total size of file.txt = 27 bytes

1. Hàm ungetc

Để đẩy 1 byte hay một kí tự vào stream ta dùng hàm **ungetc()**

Cú pháp:

int ungetc(int char, FILE \*stream)

* **char** – Kí tự cần trả về stream.
* **stream** – Con trỏ kiểu FILE, định danh input stream.

Hàm trả về kí tự đẩy vào nếu thành công, hoặc **EOF** nếu bị lỗi.

Ví dụ:

#include <stdio.h>

int main ()

{

FILE \*fp;

int c;

char buffer [256];

fp = fopen("file.txt", "r");

if( fp == NULL ) {

perror("Error in opening file");

return(-1);

}

while(!feof(fp)) {

c = getc (fp);

/\* replace ! with + \*/

if( c == '!' ) {ungetc ('+', fp);}

else {ungetc(c, fp);}

fgets(buffer, 255, fp);

fputs(buffer, stdout);

}

return(0);

}

Trong vòng lặp while, chúng ta sẽ lấy từng kí tự tới khi tới cuối FILE. Đồng thời thay thế các kí tự ‘!’ ở đầu line thành ‘+’ trong file.

Chúng ta có text file **file.txt** như sau:

this is tutorials point

!c standard library

!library functions and macros

Biên dịch và chạy chương trình, ta có kết quả là:

this is tutorials point

+c standard library

+library functions and macros

##### Bộ đệm stream

Khi stream không được cấp bộ đệm (unbuffered), byte sẽ chuyển từ source tới destination ngay lập tức; ngược lại nếu stream được cấp bộ đệm đủ lớn (fully buffered), byte tích tụ rồi chuyển theo một khối lớn sau khi nạp vào bộ đệm.

Khi stream cấp bộ đệm dòng (line buffered), byte được tích tụ tới khi gặp kí tự ngắt dòng **‘\n’.**

Để thiết lập đặc tính bộ đệm stream, ta dùng hàm **setbuf**() và **setvbuf**().

1. Hàm setbuf()

Hàm này nên được gọi trước khi gọi các tiến trình xuất nhập

Cú pháp

void setbuf(FILE \*stream, char \*buffer)

* **stream** – Con trỏ tới mã số nhận diện FILE tương ứng với stream được mở .
* **buffer** – Bộ đệm đã cấp phát. Nên có kích thước >= BUFSIZ

Ví dụ:

#include <stdio.h>

int main(){

char buf[BUFSIZ];

setbuf(stdout, buf);

puts("This is tutorialspoint");

fflush(stdout);

return(0);

}

Kết quả:

This is tutorialspoint

1. Hàm setvbuf()

Hàm này nên được gọi trước khi gọi các tiến trình xuất nhập

Cú pháp

int setvbuf(FILE \*stream, char \*buffer, int mode, size\_t size)

* **stream** – Con trỏ tới mã số nhận diện FILE tương ứng với stream được mở .
* **buffer** – Bộ đệm đã được cấp pháp. Nếu = NULL , Hàm tự động thiết lập bộ đệm có kích thước mặc định
* **size** – Khai báo kích thước cho phép dùng của bộ đệm.
* **mode** – Khai báo chế độ cho bộ đệm file:

|  |  |
| --- | --- |
| **Chế độ** | **Mô tả** |
| \_IOFBF | **Full buffering** – Ở đầu ra, dữ liệu được ghi tới khi đầy bộ đệm. Ở đầu vào, bộ đệm chỉ được nạp tới khi có yêu cầu từ tiến trình input và bộ đệm hoàn toàn rỗng. |
| \_IOLBF | **Line buffering** – Ở đầu ra, dữ liệu được ghi vào bộ đệm tới khi gặp kí tự ngắt dòng hoặc bộ đệm đầy. Ở đầu ra, bộ đệm chỉ được nạp khi có yêu cầu từ tiến trình input và bộ đệm hoàn toàn rỗng, bộ đệm nạp tới khi gặp kí tự ngắt dòng. |
| \_IONBF | **No buffering** – Hàm sẽ bỏ qua tham số **buffer** và **size.** Byte sẽ chuyển từ nguồn sang đích ngay lập tức |

Hàm trả về zero nếu thành công, hoặc khác 0 nếu bị lỗi.

Ví dụ

#include <stdio.h>

int main(){

char buff[1024];

memset( buff, '\0', sizeof( buff ));

fprintf(stdout, "Going to set full buffering on\n");

setvbuf(stdout, buff, \_IOFBF, 1024);

fprintf(stdout, "This is tutorialspoint.com\n");

fprintf(stdout, "This output will go into buff\n");

fflush( stdout );

fprintf(stdout, "and this will appear when programm\n");

fprintf(stdout, "will come after sleeping 5 seconds\n");

sleep(5);

return(0);

}

Here program keeps buffering the output into **buff** until it faces first call to fflush(), after which it again starts buffering the output and finally sleeps for 5 seconds. It sends remaining output to the STDOUT before the program comes out.

Going to set full buffering on

This is tutorialspoint.com

This output will go into buff

and this will appear when programm

will come after sleeping 5 seconds

##### Rắc rối EOF và feof()

1. Định nghĩa EOF

* Không phải Kiểu char
* Một giá trị nằm ở cuối file
* Hoặc cũng có thể nàm ở giữa file

EOF là marco kiểu số nguyên int (2-byte) có giá trị là: 0xFFFF (-1)

Đoạn code mà chúng ta thường thấy:

int c;

while ((c = fgetc(fp)) != EOF){

putchar (c);

}

int ch;

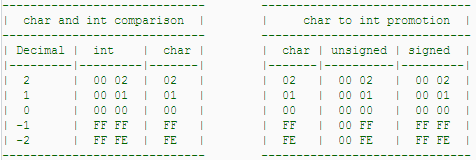
while ((ch = cin.get()) != EOF){

cout <<(char)ch;

}

1. Từ char cho tới int

Trong ví dụ trên, chuyện gì xảy ra nếu bạn dùng kiểu char thay vì kiểu int? Để hiểu rõ vấn đề, hãy xem cơ chế promote từ char sang int trong bảng bên phải.



Khi promote, bit 0 kiểu không dấu, bit dấu (1 hoặc 0) của kiểu có dấu sẽ được chèn. Xem output của ví dụ sau để hiểu rõ:

#include <stdio.h>

int main(void){

int i = -1;

signed char sc = 0xff;

unsigned char usc = 0xff;

printf ("Comparing %x with %x\n", i, sc);

if (i == sc) puts("i == sc");

else puts("i != sc");

putchar ('\n');

printf ("Comparing %x with %x\n", i, usc);

if (i == usc) puts("i == usc");

else puts("i != usc");

return 0;

}

/\*

\* Output

Comparing ffff with ffff <--- Notice this has been promoted

i == sc

Comparing ffff with 00ff

i != usc

\*

\*/

Nếu dùng kiểu char để bắt EOF

char c;

while ((c = fgetc(fp)) != EOF){

putchar (c);

}

Khi đọc tới cuối file:

- EOF so sánh với c: 0xffff và 0x00ff.

- Kết quả luôn là FALSE , dẫn tới lặp vô hạn.

1. Cẩn thận với feof

Trong phần định nghĩa của C có nói rõ rằng:

* Hàm feof kiểm tra end-of-file indicator của con trỏ stream
* Nó trả về: số khác zero nếu và chỉ nếu end-of-file indicator được set.

Vấn đề ở đây là sẽ có hàm đọc không set end-of-file indicator. Hầu hết các hàm khi đọc hết dữ liệu sẽ set EOF, sau đó chỉ trả về EOF mà thôi.

Trong ví dụ bên dưới:

while (!feof(fp)) {

fgets(buf, sizeof(buf), fp);

printf ("Line %4d: %s", i, buf);

i++;

}

Fget() sẽ không set EOF, nó chỉ trả về NULL khi đọc hết dữ liệu, dẫn tới vòng lặp vô hạn.

Đoạn code tốt hơn là kiểm tra giá trị trả về của fget

while (fgets(buf, sizeof(buf), fp) != NULL) {

printf ("Line %4d: %s", i, buf);

i++;

}

##### Binary và ASCII

1. Kí tự ngắt dòng

Từ thời xuất hiện máy in, có vấn đề là khi đầu in muốn xuống dòng, nó phải chạy từ phải qua trái về vị trí cũ. Nó mất 2/10 giây để trở lại. Trường hợp nếu có kí tự nhảy tới trong lúc đang trở về, thì kí tự xem như bị mất.

Thế là người ta phải chèn thêm kí tự <carriage return> để định vị lại đầu in về bên trái, và kí tự <line feed> để chuyển giấy in lên 1 dòng.

Khi máy tính xuất hiện, người ta nhận ra rằng lưu trữ 2 kí tự này quá lãng phí (memory thời đó là hiếm). Một số thì chọn < carriage return>, số khác thì chọn <line feed>.

Như ta thấy, hệ thống Unix chọn <line feed> để ngắt dòng, đại diện bởi kí tự ‘\n’ hex code: 0xA (LF). MS-DOS/Window chọn cả 2: < carriage return>,<line feed>.

Các nhà thiết kế trình biên dịch gặp vấn đề với các chương trình C cũ, khi nó nghĩ ngắt dòng chỉ là <line feed>. Thế là kí tự < carriage return> bị xóa khỏi ASCII, <line feed> đổi thành <carriage return><line feed> cho đầu ra.

1. Vấn đề hàm put

Hàm **std::ostream.put** có thể ghi file ASCII hay Binary. Đoạn code sau ghi 128 kí tự vào test.out:

#include <iostream>

#include <fstream>

#include <cstdlib>

main(){

int cur\_char; // Current character to write

std::ofstream out\_file; // Output file

out\_file.open("test.out", fstream::out);

if (out\_file.bad()) {

std::cerr << "Cannot open output file\n";

exit (8);

}

for (cur\_char = 0; cur\_char < 128; ++cur\_char)

out\_file << cur\_char;

return;

}

Trên hệ thống Unix ghi đủ 128 byte nhưng trên MS\_DOS/Window lại là 129 byte.Khi ghi tới kí tự 10 (LF), thì cả kí tự <carriage return> được thêm vào.

Để giải quyết bạn phải ghi dạng binary.

1. Vấn đề về bộ đệm IO

IO dùng bộ đệm sẽ không ghi ngay ra file. Thay vào đó dữ liệu giữ trong bộ đệm cho tới khi đủ lớn hoặc bị “flushed”.

std::cout << "Starting";

do\_step\_l ( );

std::cout << "Step 1 complete";

do\_step\_2();

std::cout << "Step 2 complete";

do\_step\_3();

std::cout << "Step 3 complete\n";

Thay vì in tin nhắn từng bước, **cout** nạp vào bộ đệm. Chỉ khi chương trình hoàn thành thì bộ đệm mới “flushed” ra màn hinh một lúc.

Bạn có thể dùng I/O manip **flush** để ép stream flush bộ đệm

std::cout << "Starting"<< std::flush;

do\_step\_l ( );

std::cout << "Step 1 complete"<< std::flush;

do\_step\_2();

std::cout << "Step 2 complete"<< std::flush;

do\_step\_3();

std::cout << "Step 3 complete\n"<< std::flush;

1. IO không dùng bộ đệm

Đối với kiểu I/O không dùng bộ đệm, dữ liệu ghi tức thì lên file. I/O có bộ đệm được dùng phổ biến hơn, vì I/O không có bộ đệm mỗi lần ghi hay đọc sẽ dùng system call, vốn rất tốn chi phí.

I/O không bộ đệm chỉ nên dùng khi thao tác với dữ liệu Binary lớn hay kiểm soát trực tiếp thiết bị và file.

Ví dụ như khi bạn gom giấy, bạn có thể nhét đầy tay trái rồi cho vào thùng, nhưng cầm một trái banh thì phải cho vào ngay.

Các hàm gọi hệ thống trên Unix sẽ khác với Window, chúng ta cần dùng macro khai báo thư viện phù hợp với OS:

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#ifdef \_MSDOS\_ // If we are MS-DOS

#include <io.h> // Get the MS-DOS include file for raw I/O

#else /\* \_MSDOS\_ \*/

#include <unistd.h> // Get the UNIX include file for raw I/O

#endif /\* \_MSDOS\_ \*/

int file\_descriptor = open("file\_path", O\_RDONLY); // Existing file

//file\_descriptor = open(name, flags, mode) :New file

Ý nghĩa:

* file\_descriptor: là mã số định danh cho file, cần thiết cho thao tác đọc, ghi , đóng stream. Giá trị này <0 xem như bị lỗi.
* Tham số thứ 2 là flag

Bảng flag

|  |  |
| --- | --- |
| **Marco** | **Mô tả** |
| O\_RDONLY | Mở để chỉ đọc |
| O\_WRONLY | Mở để chỉ ghi |
| O\_RDWR | Mở vừa đọc vừa ghi |
| O\_APPEND | Ghi tiếp cuối file |
| O\_CREAT | Tạo file (tham số **mode** phải khai báo cho flag này) |
| O\_TRUNC | Nếu file tồn tại thì cho nó về zero |
| O\_EXCL | Thất bại nếu file tồn tại |
| O\_BINARY | Mở dạng nhị phân |

* Mode : chế độ bảo vệ cho file. Bình thường là 0666 cho mọi file.

Ví dụ: khi mở file đã tồn tại data.txt theo chế độ đọc kí tự

data\_fd = open("data.txt", O\_RDONLY);

Tạo file output.dat chỉ để ghi:

out\_fd = open("output.dat", O\_CREAT|O\_WRONLY, 0666);

Nhớ dùng toán tử OR (|) để kết hợp nhiều flag.

Khi chương trình được mở, mặc định ta có 3 file descriptor

* 0 Standard in
* 1 Standard out
* 2 Standard error

Cú pháp hàm đọc:

read\_size = read(filedescriptor, buffer, size);

Hàm trả về số byte đã đọc. Trả về zero tức đã đọc hết file , <0 tức xảy ra error.

**Size** qui định số byte phải đọc trong 1 lần gọi. Thường size = sizeof(buffer)

Cú pháp hàm ghi:

write\_size = write(file\_descriptor, buffer, size);

Hàm trả về số byte đã ghi, nếu số byte <0 tức xảy ra lỗi.

**Size**: qui định số byte phải ghi trong 1 lần gọi. Thường ta để size = sizeof(buffer)

Đóng file:

Int flag = close(file\_descriptor)

flag: là đóng thành công, <0 là xảy ra error.

Ví dụ sau minh họa việc sao chép dữ liệu trên I/O không bộ đệm. Chúng ta khai báo kích thước bộ đệm là 16KB

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Usage: copy <from> <to> \*

\* <from> -- file nguồn \*

\* <to> -- file đích \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const int BUFFER\_SIZE = (16 \* 1024); // Use 16K buffers

main(int argc, char \*argv[]) {

char buffer[BUFFER\_SIZE]; // Buffer for data

int in\_file; // Input file descriptor

int out\_file; // Output file descriptor

int read\_size; // Number of bytes on last read

if (argc != 2) {

std::cerr << "Error: Wrong number of arguments\n";

cerr << "Usage is: copy <from> <to>\n";

exit(8);

}

in\_file = open(argv[1], O\_RDONLY);

if (in\_file < 0) {

std::cerr << "Error: Unable to open " << argv[1] << '\n';

exit(8);

}

out\_file = open(argv[2], O\_WRONLY | O\_TRUNC | O\_CREAT, 0666);

if (out\_file < 0) {

std::cerr << "Error: Unable to open " << argv[2] << '\n';

exit(8);

}

while (1){

read\_size = read(in\_file, buffer, sizeof(buffer));

if (read\_size == 0){

break; // End of file

if (read\_size < 0)

std::cerr << "Error: Read error\n";

exit(8);

}

write(out\_file, buffer, (unsigned int) read\_size);

}

close(in\_file);

close(outfile);

return (0);

}

Chương trình có xuất ra lỗi cảnh báo để hướng dẫn sử dụng đúng cách. Khi ghi dữ liệu, có thể lần gọi cuối cùng sẽ không ghi đúng số byte của bộ đệm, nên ta phải khai báo read\_size cho số byte đọc được, truyền vào tham số của ghi file.

1. Thiết kế định dạng file

Giả sử rằng bạn đang định dạng file config, gồm thông số và mô tả. Như sau:

Plot Configuration File V1.0

log Logarithmic or normal plot

10.0 height (in inches)

7.0 width (in inches)

0 x lower limit

100 x upper limit

30 y lower limit

300 y upper limit

0.5 x-scale

2.0 y-scale

Dòng tiều đề bên trên để nhằm tránh bạn chọn nhầm file config cho chương trình, và chương trình có thể in ra lỗi

Đồng thời có thể thêm phiên bản phía sau để nhận biết.

Trong binary file, có thực tế khá phổ biến là đặt 4-bytes đầu tiên, gọi là số ma thuật, để nhận biết file. Người ta thường lấy 4 kí tự đầu tên file chuyển sang dạng hex rồi cộng thêm gì đó (vd: 0x80808080), sau đó chèn vào 4-bytes đầu.

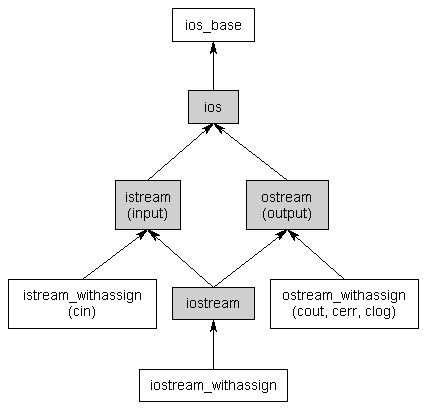
Cách tạo số ma thuật khá độc đáo. Bit cao nhất được set 1 cho mọi byte để tránh nhầm với ASCII file.

Khi đọc hay ghi binary các file có cấu trúc khác nhau, rất dễ bị fail. Lập trình viên nên đặt số ma thuật đầu file

### C++ Stream

##### Thư viện iostream

Khi include <iostream>, bạn đã truy xuất tới toàn bộ cây phân cấp I/O (trong có có 1 class tên là iostream). Cây thừa kế cho lớp non-file IO có dạng sau:

Chúng ta có thể thấy cây phân cấp này dùng đa thừa kế, đa thừa kế tiềm ẩn nhiều rủi ro nhưng thư viện iostream được test rất cẩn thận nên an toàn để sử dụng.

##### Streams

Thuật ngữ stream được dùng phổ biến, cơ bản IO trong C++ được hiểu là stream. Hiểu theo nghĩa trừu tượng, stream là chuỗi các kí tự nối tiếp nhau, theo thời gian, stream sẽ mở rộng ra thành lượng lớn dữ liệu.

Thông thường chúng ta sẽ gặp 2 loại stream, streams Input để chứa dữ liệu từ bàn phím, file, network… Ví dụ khi bạn nhấn phím trong khi ứng dụng không trong trạng thái đợi input, thay vì bỏ qua, dữ liệu sẽ được nạp vào input stream, sau đó ứng dụng sẽ xử lí chúng sau.

Ngược lại, output stream được dùng để giữ dữ liệu xuất ra tới màn hình, file, máy in, network… Khi ghi dữ liệu vào thiết bị output, nếu thiết bị không trong trạng thái nhận dữ liệu – ví dụ máy in đang làm nóng thì có thể đưa dữ liệu vào output stream của nó, sau đó được xử lí khi máy in sẵn sàng.

Một số thiết bị, file và network, vừa có thể là input và output.

Lập trình viên chỉ cần tương tác với stream để đọc hay ghi dữ liệu cho bất kì thiết bị nào, stream rất tiện lợi.

##### Input/output stream in C++

Mặc dù lớp ios thừa kế từ ios\_base, nhưng sẽ là lớp bạn sẽ làm việc trực tiếp. Định lớp ios gồm một mớ hỗn độn các input và output stream.

* Lớp istream là class chính được dùng khi làm việc với input stream: toán tử xuất stream (>>) dùng để xóa giá trị trong stream. Có nghĩa: khi người dùng nhấn phím, mã phím được đặt vào input stream, ứng dụng sẽ tách giá trị từ stream để dùng.
* Lớp ostream là class chình dùng khi làm việc với output stream: toán tử chèn stream (<<) dùng để chèn giá trị vào stream. Có nghĩa: bạn chèn một giá trị vào stream, và thiết bị sẽ dùng nó.
* Lớp iostream có thể xử lí cả input và output, cho phép giao tiếp IO 2 chiều

Cuối cùng, có một loạt các class có hậu tố “\_withassign”: các lớp stream này được thừa kế từ istream, ostream và iostream, được định nghĩa toán tử gán, cho phép gán stream sang stream khác. Nhưng hầu như bạn ít khi phải đụng tới chúng.

##### Output với ostream và ios flag

1. Toán tử chèn stream <<

C++ đã định nghĩa trước toán tử chèn stream cho tất cả các kiểu dữ liệu sẵn có, bạn cũng có thể quá tải cho class.

Istream và ostream đều thừa kế từ ios, một trong nhiệm vụ của ios (và ios\_base) là kiểm soát tùy chọn định dạng cho output stream

1. Định dạng stream output

Toán tử (<<) dùng để chuyển đổi mọi loại dữ liệu sang kí tự và đẩy vào ostream hay ofstream. Tính năng này rất linh hoạt, có thể chuyển kiểu số (hệ 10, 8 hay 16) thành chuỗi cố định hay co giãn, canh trái hay canh phải.

Có 2 cách để định dạng stream:

* flags
* manip

1. Dùng biến flag

Để xuất ra đúng chuỗi định dạng, bạn cần dùng flags chuyển đổi để kiểm soát.

cout.setf(<ios::biến\_flag>); // Set flags

cout.unsetf(<ios::biến\_flag>); // Clear flags

Bảng flag:

|  |  |
| --- | --- |
| **Flag** | **Mô tả** |
| ios:: skipws | Bỏ qua các kí tự white space khi xuất |
| ios:: left | Canh trái khi xuất |
| ios:: right | Canh phải khi xuất |
| ios::internal | Chuỗi số xuất ra sẽ được chèn thêm kí tự khoảng cách với dấu hay kí hiệu hệ số |
| ios::dec | Chuỗi số định dạng hệ thập phân |
| ios::oct | Chuỗi số định dạng hệ bát phân |
| ios::hex | Chuỗi số định dạng hệ thập lục phân |
| ios::showbase | In kí hiệu hệ số kèm theo mỗi chuỗi số |
| ios::showpoint | In dấu chấm động cho chuỗi số dù có cần hay không |
| ios::uppercase | Định dạng chữ hoa cho số thập lục phân (A,B,C,D,E,F) |
| ios: :showpos | Đấu + trước mỗi số dương |
| ios::scientific | Định dạng các số có dấu chấm động thành tích số với lũy thừa cơ số 10 |
| ios::fixed | Định dạng số có dấu chấm động sang số nguyên |
| ios::unitbuf | Buffer output. (More on this later). |
| ios: :stdio | Flush bộ đệm stream |

Ví dụ:

int number = 0x3FF; //gán giá trị kiểu hex

cout << "Dec: " << number << '\n';

cout.setf(ios::hex, ios::basefield);

cout << "Hex: " << number << '\n';

cout.setf(ios::dec);

Kết quả:

Dec: 1023

Hex: 3ff

Bạn nhớ trả lại trạng thái cho cout sau khi xuất stream..

Khi định dạng số sang chuỗi số, ta dùng các hàm thành viên sau:

cout.width(int size); //size : số kí tự cần biểu diễn chuỗi số

cout.precision(int digits); //digits: số kí tự biểu diễn phần thập phân

cout.fill(char pad); //pad: kí tự thêm vào để lấp đầy đầu chuỗi số nếu chưa đủ size.

Nếu chuỗi số có số kí tự < size,mặc định khoảng trắng sẽ được thêm vào trước.

**Lưu ý:**

**Vài flag và tham số sẽ tự động trở về trạng thái ban đầu sau khi xuất, một số thì không. Tốt nhất là unsetf() để an toàn, do sự không tương thích giữa các trình biên dịch khác nhau**

1. Dùng manipulator

Cách dùng flag khá phức tạp và phiền phức khi bạn phải set và clear flag.

Để đơn giản hơn, C++ cung cấp cho bạn các I/O manip là các hàm đặc biệt được dùng trong các câu lệnh I/O để định dạng chuỗi xuất.

Các manip I/O này được xem là quá trình trung gian làm thay đổi trạng thái của file stream. Để sử dụng, bạn phải khai báo thư viện <iomanip>

Ví dụ về manip hex, chuyển output sang dạng số thập lục phân:

#include <iostream>

#include <iomanip>

number = Ox3FF;

cout << "Number is " << hex << number << '\n';//in ra 3ff

Danh sách các manip

|  |  |
| --- | --- |
| **Manip** | **Mô tả** |
| **setiosflags**(long flags) | Thiết lập flag . |
| **resetiosflags**(long flags) | Xóa thiết lập flag |
| **dec** | Định dạng chuỗi số hệ 10 |
| **hex** | Định dạng chuỗi số hệ 16 |
| **oct** | Định dạng chuỗi số hệ 8 |
| **setbase**(int base) dec, hex, oct. | Thiết lập hệ số cơ sở cho định dạng là 8, 10 , hay 16 |
| **setw**(int width) | Cấu hình độ rộng chuỗi xuất |
| **setprecision**(int precision) | Cấu hình số kí tự phần thập phân |
| **setfill** (char ch) | Cấu hình kí tự lấp đầy |
| **ws** | Bỏ qua white space khi xuất |
| **endl** | Xuất ra end-of-line |
| **ends** | Xuất ra end-of-string ('\0'). |
| **flush** | Flush bộ đệm output. |

Ví dụ

#include <iostream>

#include <iomanip>

int main(){

int number = 12; // A number to output

float real = 12.34; // A real number

cout << setw(5) << number << "\n";

cout << setw(5) << setfill('\*') << number << "\n";

cout << setiosflags(ios::showpos|ios::left) << setw(5) <<number << "\n";

cout << real << "\n";

cout << setprecision(1) << setiosflags(ios::fixed) << real << "\n";

cout << setiosflags(ios::scientific) << real << "\n";

return (0);

}

Kết quả

\*\*\*12

+12\*\*

12.34

12.3

le+01

##### Input với istream

1. Toán tử tách stream >>

C++ đã định nghĩa trước toán tử tách stream cho tất cả kiểu dữ liệu sẵn có, bạn có thể quá tải cho định nghĩa lớp

Khi đọc chuỗi, một lỗi phổ biến với toán tử tách stream là ngăn không cho tràn bộ đệm input, ví dụ:

char buf[10];

std::cin >> buf;

Điều gì xảy ra nếu nhập 18 kí tự? bộ đệm sẽ tràn. Nhìn chung đây là ý tưởng tồi khi giả định số kí tự người dùng sẽ nhập.

Một cách để xử lí vấn đề này là dùng manip “**setw**” để giới hạn số kí tự được đọc từ stream.

#include <iostream>

#include <iomanip.h>

char buf[10];

std::cin >> setw(10) >> buf;

Ứng dụng sẽ chỉ đọc 9 kí tự ra khỏi stream, chừa chỗ cho kí tự kết chuỗi, các kí tự còn lại nếu còn sẽ vẫn nằm trong bộ đệm stream chờ cho lượt tách stream sau đó.

1. Tách stream và kí tự trắng

Một điều cần chú ý khi dùng toán tử tách stream với dữ liệu đã được định dạng, đó là nó sẽ bỏ qua các kí tự trắng (blanks, tabs, and newlines).

Xem đoạn code nhỏ sau:

#include <iostream>

int main()

{

char ch;

while (cin >> ch)

cout << ch;

return 0;

}

Nếu người dùng nhập chuỗi : “Hello my name is Alex”, thì toán tử tách stream sẽ bỏ qua khoảng trắng và ngắt dòng. Do đó, tin nhắn xuất ra sẽ chỉ là “HellomynameisAlex”

Để chấp nhận kí tự trắng, cách thông thường nhất là dùng phương thức thành viên **istream**::**get**()

int main()

{

char ch;

while (cin.get(ch))

cout << ch;

return 0;

}

Kết quả:” Hello my name is Alex”

Phương thức get() chấp nhận cả tham số chuỗi, có thể nhận số kí tự tối đa để đọc:

int main()

{

char strBuf[11];

cin.get(strBuf, 11);

cout << strBuf << endl;

return 0;

}

Kết quả: Hello my n

Vì chúng ta chỉ đọc 10 kí tự, chừa 1 chỗ cho kí tự kết chuỗi. Các kí tự còn lại vẫn nằm trong bộ đệm stream

Một điều quan trọng về get() là nó không đọc kí tự ngắt dòng:

int main()

{

char strBuf[11];

// Có thể đọc 10 kí tự

cin.get(strBuf, 11);

cout << strBuf << endl;

// Có thể đọc 10 kí tự hoặc hơn

cin.get(strBuf, 11);

cout << strBuf << endl;

return 0;

}

Nếu nhập: Hello! và ngắt dòng

Ứng dụng sẽ in: Hello! và thoát

Tại sao nó không yêu cầu input 10 kí tự của hàm get() phía sau? Vì hàm get() đầu tiên đọc tới kí tự ngắt dòng thì dừng lại, tuy nhiên kí tự ngắt dòng vẫn nằm trong stream.

Hàm get() thứ hai đọc tiếp, nhưng nó lại đụng phải kí tự ngắt dòng, nên ngắt ngay lập tức.

Do đó, khi cần đọc theo dòng thì nên dùng hàm thành viên **istream**::**getline**()

int main()

{

char strBuf[11];

// Read up to 10 characters

cin.getline(strBuf, 11);

cout << strBuf << endl;

// Read up to 10 more characters

cin.getline(strBuf, 11);

cout << strBuf << endl;

return 0;

}

Nếu bạn cần biết có bao nhiêu kí tự đã được tách bởi lần gọi getline() cuối cùng, dùng **istream**::**gcount**()

int main()

{

char strBuf[100];

cin.getline(strBuf, 100);

cout << strBuf << endl;

cout << cin.gcount() << " characters were read" << endl;

return 0;

}

1. Phiên bản getline() cho std::string

Có một phiên bản đặc biệt **getline**() nằm ngoài class istream và ostream, được dùng để đọc tham số đầu vào kiểu **std::string**

Phương thức độc lập này nằm trong header string. Xem cách dùng qua đoạn code sau:

#include <string>

#include <iostream>

int main()

{

using namespace std;

string strBuf;

getline(cin, strBuf);

cout << strBuf << endl;

return 0;

}

1. Một vài hàm thành viên của istream

* **ignore**():loại bỏ kí tự đầu tiên trong stream
* **ignore**(int **nCount**) : loại nCount kí tự đầu tiên trong stream
* **peek**() : cho phép đọc kí tự từ stream mà không phải tách stream
* **unget**() : chèn kí tự cuối cùng đã tách stream vào lại stream
* **putback**(char **ch**) : cho phép chèn kí tự ch vào lại stream

##### Stream classes cho chuỗi

C++ cung cấp một số stream class để xử lí chuỗi có dùng toán tử nhập stream (<<) và tách stream (>>)

Giống như istream và ostream, stream chuỗi cung cấp bộ đệm chứa dữ liệu. Tuy nhiên, khác với cin và cout, những stream class này không kết nối tới các kênh IO (màn hình, bàn phím, máy in…).

Mục đích chính của stream chuỗi là bộ đệm output sau này, hay để xử lí chuỗi input theo từng dòng một.

Có 3 stream class cho chuỗi:

* istringstream (thừa kế từ istream), ostringstream (thừa kế từ ostream), stringstream (thừa kế từ iostream) được dùng để đọc và ghi chuỗi kí tự.
* wistringstream, wostringstream, và wstringstream được dùng để đọc và ghi chuỗi kí tự nhiều bytes

1. stringstream

Để dùng stringstreams, bạn cần #include <sstream>

Có 2 cách để set dữ liệu cho stringstream:

* Dùng toán tử chèn stream (<<):

#include <sstream>

std::stringstream os;

os << "en garde!" << endl; // chèn "en garde!" vào stringstream

* Dùng hàm thành viên **stringstream**::**str**(**std**::**string**) để nạp nội dung chuỗi **std**::**string** vào bộ đệm stream:

std::stringstream os;

os.str("en garde!"); // set bộ đệm stringstream nhận "en garde!"

Tương tự có 2 cách lấy dữ liệu từ stringstream:

* Dùng toán tử tách stream (>>):

#include <sstream>

using namespace std;

stringstream os;

os << "12345 67.89"; // chèn chuỗi số vào stream

string strValue;

os >> strValue;

string strValue2;

os >> strValue2;

// xuất chuỗi số

cout << strValue << " - " << strValue2 << endl;

Kết quả:

12345 - 67.89

Lưu ý toán tử tách strream “>>” duyệt qua chuỗi – mỗi lần dùng thành công “>>” sẽ trả về giá trị có thể tách tiếp theo trong strream.

Ngược lại, hàm thành viên **str**() trả về toàn bộ giá trị chuỗi, dù toán tử tách stream “>>” đã dùng trên stream.

* Dùng hàm thành viên **stringstream**::**str**() để lấy chuỗi kí tự từ bộ đệm:

#include <sstream>

std::stringstream os;

os << "12345 67.89" << endl;

std::cout << os.str();

Kết quả:

12345 - 67.89

1. Chuyển đổi số và chữ

Toán tử tách và chèn stream được hỗ trợ xử lí tất cả các kiểudữ liệu cơ bản của C++, chúng ta có thể tận dụng stream để chuyển đổi chuỗi kí tự thành số và ngược lại.

Ví dụ về chuyển đổi số thành chuỗi kí tự:

#include <sstream>

#include <iostream>

using namespace std;

template<class NumbericType>

std::string ToString(NumbericType &n){

stringstream os;

std::string str;

os << n ;

os >> str;

return str;

}

int main(){

double dValue = 67.89;

cout << ToString<double>(dValue) << endl;

return 0;

}

Kết quả:

67.89

Ví dụ đổi chuỗi kí tự số thành số:

#include <sstream>

#include <iostream>

using namespace std;

template<class NumbericType>

NumbericType ToNumberic(std::string num\_str){

stringstream os;

NumbericType nValue;

os << num\_str; // chèn chuỗi kí tự số stream

os >> nValue;

return nValue;

}

int main(){

cout << ToNumberic<double>("67.89") << endl;

return 0;

}

Kết quả:

67.89

1. CleanUp stringstream để tái sử dụng

Có nhiều cách để làm trống bộ đệm stream, có thể dùng chuỗi rỗng C-style hoặc đối tượng rỗng std::string() vào tham số hàm thành viên stringstream::str().

Khi dọn dẹp bộ đệm stringstream, nhớ gọi hàm thành viên stringstream::clear() để reset error flags để trở về trạng thái tốt.

##### Trạng thái stream

Ios\_base class chứa nhiều flag được dùng để nhận biết trạng hiện tại của stream:

|  |  |
| --- | --- |
| **Flag** | **Meaning** |
| **goodbit** | Trạng thái tốt |
| **badbit** | Một số lỗi đã xảy ra ( vd: ứng dụng đang cố đọc qua điểm kết thúc của file) |
| **eofbit** | Stream đã đi tới điểm kết file |
| **failbit** | Lỗi xảy ra (vd: người dùng nhập kí tự khi ứng dụng đang chờ chuỗi số) |

Mặc dù những flag này nằm trong ios\_base, nhưng class ios thừa kế từ ios\_base và ngắn gọn hơn nên người ta lại hay dùng class ios để truy xuất các flag này (ví dụ std::ios::failbit).

Ios class cũng cung cấp nhiều hàm thành viên để check flag tiện hơn:

|  |  |
| --- | --- |
| **Member function** | **Meaning** |
| **good**() | Returns true nếu goodbit được set |
| **bad**() | Returns true nếu badbit được set |
| **eof**() | Returns true nếu eofbit được set |
| **fail**() | Returns true nếu failbit được set |
| **clear**() | Clears tất cả flags và đưa stream về trạng thái goodbit |
| **clear**(state) | Clears tất cả flags và sets trạng thái stream theo tham số flag **state** |
| **rdstate**() | Return về flag hiện đang set của stream |
| **setstate**(state) | Sets trạng thái stream theo tham số flag **state** |

Hầu như chúng ta sẽ làm việc với failbit, được set khi người dùng input giá trị không hợp lệ. Ví dụ:

cout << "Enter your age: ";

int nAge;

cin >> nAge; //failbit được set khi chúng ta input chuỗi kí tự khác số

Khi xảy ra lỗi và stream được set bất kì flag nào khác goodbit, thì các thực thi tiếp theo trên stream đều bị bỏ qua.

Vấn đề có thể giải quyết bằng cách gọi phương thức **clear()** để reset trạng thái cho stream.

##### Kiểm tra input

Để stream hoạt động hiệu quả, và luôn ở trạng thái tốt (goodbit), chúng ta cần quá trình kiểm tra input cho stream.

Input có thể chia làm 2 dạng chính chuỗi kí tự và số.

Khi kiểm tra chuỗi, chúng ta chấp nhận tất cả input là chuỗi hết, sau đó tùy theo chính sách về định dạng mà có thể từ chối dữ liệu input.

Trong hầu hết ngôn ngữ (đặc biệt là script language như Pearl và PHP), việc kiểm tra có thể dùng regular expressions, C++ đã hỗ trợ tính năng này.

Trước đó, việc kiểm tra được thực hiện qua từng kí tự của chuỗi.

Khi kiểm tra chuỗi số, chúng ta đảm bảo chỉ có kí tự số được nhập trong khoảng nào đó, vd 0 tới 20, tuy nhiên khác với chuỗi kí tự, có khả năng người dùng input kí tự không phải số.

C++ cung cấp nhiều hàm giúp kiểm tra các kí tự là số hay chữ, trong header <cctype>:

|  |  |
| --- | --- |
| **Function** | **Meaning** |
| **isalnum**(int) | Returns non-zero nếu tham số là kí tự alphabet hay số |
| **isalpha**(int) | Returns non-zero nếu tham số là kí tự alphabet |
| **iscntrl**(int) | Returns non-zero nếu tham số là kí tự điều khiển |
| **isdigit**(int) | Returns non-zero nếu tham số là kí tự số |
| **isgraph**(int) | Returns non-zero nếu tham số là kí tự có thể in ra, không phải nhóm kí tự trắng |
| **isprint**(int) | Returns non-zero nếu tham số là kí tự có thể in ra (bao gồm nhóm kí tự trắng) |
| **ispunct**(int) | Returns non-zero nếu tham số không phải nhóm kí tự trắng, cũng không phải kí tự số hay alphabet |
| **isspace**(int) | Returns non-zero nếu tham số là kí tự trắng |
| **isxdigit**(int) | Returns non-zero nếu tham số là kí tự biểu diễn số hexa (0-9, a-f, A-F) |

1. Kiểm tra chuỗi kí tự

Ví dụ về kiểm tra việc nhập tên người dùng, chỉ gồm kí tự alphabet và khoảng trắng:

#include <cctype>

#include <string>

#include <iostream>

using namespace std;

int main(){

while (1)

{

// nhập user's name

cout << "Enter your name: ";

string strName;

getline(cin, strName); // nhập 1 dòng ,kể cả kí tự trắng

bool bRejected = false; // has strName been rejected?

// Duyệt từng kí tự chuỗi cho tới khi kết chuỗi hay gặp kí tự không hợp lệ

for (unsigned int nIndex=0; nIndex < strName.length() && !bRejected; nIndex++)

{

// bỏ qua kí tự alphabet

if (isalpha(strName[nIndex]))

continue;

// khoảng trắng cũng ok

if (strName[nIndex]==' ')

continue;

// các kí tự khác đều không hợp lệ, phải reject

bRejected = true;

}

// Nếu input hợp lệ thì break while nếu không thì lặp tiếp

if (!bRejected)

break;

}

}

Đoạn code trên chưa hoàn chỉnh lắm khi xét về mặt ý nghĩa của chuỗi kí tự, người dùng có thể nhập các từ vô nghĩa, chúng ta có thể mở rộng phạm vi kiểm tra ra như chỉ chấp nhận chuỗi có chứa ít nhất một kí tự và nhiều nhất 1 khoảng trắng.

Ví dụ tiếp theo yêu cầu nhập số điện thoại, số kí tự sẽ giới hạn, vị trí các con số sẽ phải tuân theo qui ước. Do đó, chúng ta phải có chiến lược khác, tinh vi bằng cách phát triển một template như sau:

|  |  |
| --- | --- |
| **Kí tự đặc biệt** | **Ý nghĩa** |
| # | Đại diện cho kí tự số. |
| @ | Đại diện cho kí tự alphabet. |
| \_ | Đại diện cho kí tự trắng. |
| ? | Đại diện cho kí tự bất kì. |

Luật là kí tự mà người dùng input và template phải khớp với nhau

Ví dụ: Nếu chúng ta yêu cầu template “(###) ###-####”, tức là người dùng phải nhập theo dạng : (222) 846-4445

Xem triển khai của hàm InputMatches:

bool InputMatches(string strUserInput, string strTemplate)

{

if (strTemplate.length() != strUserInput.length())

return false;

// Duyệt từng kí tự input

for (unsigned int nIndex=0; nIndex < strTemplate.length(); nIndex++)

{

switch (strTemplate[nIndex])

{

case '#': // kí tự số

if (!isdigit(strUserInput[nIndex]))

return false;

break;

case '\_': // Kí tự trắng

if (!isspace(strUserInput[nIndex]))

return false;

break;

case '@': // kí tự alphabet

if (!isalpha(strUserInput[nIndex]))

return false;

break;

case '?': // Kí tự bất kì

break;

default: // khớp kí tự nào đó

if (strUserInput[nIndex] != strTemplate[nIndex])

return false;

}

}

return true;

}

int main()

{

string strValue;

while (1)

{

cout << "Enter a phone number (###) ###-####: ";

getline(cin, strValue); // nhập 1 dòng có gồm khoảng trắng

if (InputMatches(strValue, "(###) ###-####"))

break;

}

cout << "You entered: " << strValue << endl;

}

Phương thức có 1 điểm yếu là nếu người dùng nhập kí tự đặc biệt thì không hoạt động.

Khác với regular expression, hàm không có symbol cho phép nhập một chuỗi số có độ dài tùy biến. Một template không thể dùng để đảm bảo người dùng nhập 2 từ cách nhau bởi kí tư trắng, nó không thể xử lí một từ có độ dài tùy biến.

Với vấn đề trên, giải pháp non-template là tối ưu hơn.

1. Kiểm tra số

Các nhanh nhất là dùng toán tử tách stream ra kiểu số, sau đó kiểm tra failbit để biết người dùng có nhập đúng hay không.

int main()

{

int nAge;

while (1)

{

cout << "Enter your age: ";

cin >> nAge;

if (cin.fail()) // no extraction took place

{

cin.clear(); // reset trạng thái stream về goodbit để có thể dùng ignore()

cin.ignore(1000, '\n'); // clear input trong stream tới 1000 kí tự

continue; // thử lại input

}

if (nAge <= 0) // đảm bảo input là số dương

continue;

break;

}

cout << "You entered: " << nAge << endl;

}

Tuy nhiên, sẽ có trường hợp người dùng bắt đầu nhập số và kết thúc cũng là số, nhưng ở giữa là kí tự (“32bbcd345”), khi đó “32” sẽ được tách vào nAge, chuỗi còn lại vẫn nằm trong stream (“bbcd345”), failbit vẫn không bị set.

Có 2 vấn đề:

* Chập nhận input như vậy thì phải dọn stream
* Nếu không muốn xác thực input, nó sẽ không bị reject (bạn phải dọn stream cho những thứ còn lại)

Vấn đề đầu tiên có thể xử lí dễ dàng như sau:

int main()

{

int nAge;

while (1) {

cout << "Enter your age: ";

cin >> nAge;

if (cin.fail()) // no extraction took place

{

cin.clear();// reset trạng thái stream về goodbit để có thể dùng ignore()

cin.ignore(1000, '\n'); // dọn input không hợp lệ trong stream

continue; // thử lại

}

cin.ignore(1000, '\n'); // cleanUp những thứ còn lại trong stream

if (nAge <= 0) // đảm bảo nAge là dương

continue;

break;

}

cout << "You entered: " << nAge << endl;

}

Nếu không muốn chấp nhận kiểu input như vậy (“32bbcd345”), chúng ta có thể dùng **gcount**() để xác định bao nhiêu kí tự bị bỏ qua khi dọn stream với **ignore**().

Nếu input hợp lệ, **gcount**() sẽ return 1. (cho kí tự ngắt dòng).

Nếu **gcount**() return khác 1, tức người dùng đang input thứ gì đó không thể tách, chúng ta sẽ đề nghị input mới.

int main()

{

int nAge;

while (1)

{

cout << "Enter your age: ";

cin >> nAge;

if (cin.fail()) // kiểm tra failbit

{

cin.clear(); // reset trạng thái stream về goodbit để có thể dùng ignore()

cin.ignore(1000, '\n'); // dọn input không hợp lệ trong stream

continue; // thử lại

}

cin.ignore(1000, '\n'); // cleanUp những thứ còn lại trong stream

if (cin.gcount() > 1) // Nếu số kí tự bỏ đi > 1

continue; // thì xem như input chưa hợp lệ

if (nAge <= 0) // đảm bảo nAge là dương

continue;

break;

}

cout << "You entered: " << nAge << endl;

}

1. Kiểm tra chuỗi số

Có một cách khác để xác thực chuỗi số input, là xử lí nó như chuỗi kí tự thông thường, nếu nó vượt qua cuộc kiểm tra thì convert sang kiểu số.

Đoạn code sau sẽ dùng stringstream:

int main()

{

int nAge;

while (1)

{

cout << "Enter your age: ";

string strAge;

cin >> strAge;

// Kiểm tra từng kí tự là kiểu số

bool bValid = true;

for (unsigned int nIndex=0; nIndex < strAge.length(); nIndex++)

if (!isdigit(strAge[nIndex]))

{

bValid = false;

break;

}

if (!bValid)

continue;

// lúc này chuỗi input là hợp lệ

// dùng stringstream để chuyển đổi chuỗi sang số

stringstream strStream;

strStream << strAge;

strStream >> nAge;

if (nAge <= 0) // đảm bảo nAge dương

continue;

break;

}

cout << "You entered: " << nAge << endl;

}

Phương pháp này nhanh gọn hơn tách stream rồi kiểm tra tính hợp lệ.

### C++ File I/O

Để thao tác I/O trên file chúng ta phải dùng đối tượng stream trong thư viện <fstream.h>. . Gồm

* **std::ifstream:** Đọc dữ liệu. **std::ifstream** được dẫn xuất từ **std::istream**.
* **std::ofstream:** Ghi dữ liệu **. std::ofstream** được dẫn xuất từ **std::ostream**
* **std::fstream:** được dẫn xuất từ **std::iostream**

##### Mở/Tạo file

Tạo đối tượng đọc file:

std::ifstream data\_file;

Dùng **std::ifstream.open** để mở đường dẫn file:

data\_file.open("numbers.dat" , fstream::in);

C++ cho phép mở trực tiếp qua hàm dựng của stream:

std::ifstream data\_file("numbers.dat")

Hàm hủy sẽ tự động đóng stream cho bạn.

Tạo đối tượng để ghi file

std::ofstream out\_file("out.dat", fstream::out);

Thực ra hàm dựng có thể nhận 3 tham số: đường dẫn file , chế độ mở và chế độ bảo vệ

std::ofstream(const char \*name, int mode=ios::out, int prot = filebuf::openprot)

Ý nghĩa tham số :

* filePath có kiểu dữ liệu là const char\*, là đường dẫn đến file cần mở.
* prot **:** Chế độ bảo vệ (có thể để trống). Trên Unix mặc định là 0644 (read/write owner,group read, other read), MS DOS/Window mặc định là 0
* mod là chế độ mở file. Chúng ta có một số chế độ như sau:

| **Mod** | **CHỨC NĂNG** |
| --- | --- |
| fstream::in | Mở file để đọc. |
| fstream::out | Mở file có sẵn để ghi |
| fstream::binary | Mở file ở chế độ nhị phân. |
| fstream:ate | Mở file và đặt con trỏ file vào cuối file. |
| fstream::app | Mở file và ghi dữ liệu vào cuối file. Nếu file không tồn tại thì tạo file mới. |
| fstream::trunc | Chế độ mở file, xóa bỏ hoàn toàn nội dung trong file được mở. |

File I/O in C++ works very similarly to normal I/O (with a few minor added complexities). There are 3 basic file I/O classes in C++: ifstream (derived from istream), ofstream (derived from ostream), and fstream (derived from iostream). These classes do file input, output, and input/output respectively. To use the file I/O classes, you will need to include the fstream header.

Unlike the cout, cin, cerr, and clog streams, which are already ready for use, file streams have to be explicitly set up by the programmer. However, this is extremely simple: to open a file for reading and/or writing, simply instantiate an object of the appropriate file I/O class, with the name of the file as a parameter. Then use the insertion (<<) or extraction (>>) operator to read/write to the file. Once you are done, there are several ways to close a file: explicitly call the close() function, or just let the file I/O variable go out of scope (the file I/O class destructor will close the file for you).

##### File output

Để ghi file, chúng ta sẽ dùng class ofstream:

#include <fstream>

#include <iostream>

#include <cstdlib> // for exit()

int main()

{

using namespace std;

// ofstream được dùng cho ghi file chúng ta sẽ mở file Sample.dat

ofstream outf("Sample.dat");

// trường hợp không thể mở file

if (!outf)

{

// Xuất lỗi và thoát

cerr << "Uh oh, Sample.dat could not be opened for writing!" << endl;

exit(1);

}

// Ghi 2 dòng vào file

outf << "This is line 1" << endl;

outf << "This is line 2" << endl;

return 0;

// Khi ra khỏi phạm vi, hàm hủy ofstream sẽ tự close file

}

Ngoài ra có thể dùng phương thức ofstream::put() để ghi từng kí tự vào file.

##### File input

Để đọc dữ liệu từ file, chúng ta dùng class ifstream.

Lưu ý là ifstream sẽ trả về 0 khi đã kết file (EOF):

#include <fstream>

#include <iostream>

#include <string>

#include <cstdlib> // for exit()

int main()

{

using namespace std;

// ifstream dùng để đọc file, chúng ta sẽ mở Sample.dat

ifstream inf("Sample.dat");

// trường hợp không thể mở file

if (!inf)

{

// xuất lỗi và exit

cerr << "Uh oh, Sample.dat could not be opened for reading!" << endl;

exit(1);

}

// dùng while để check EOF

while (inf)

{

// dùng std::string để nạp data và print

std::string strInput;

inf >> strInput;

cout << strInput << endl;

}

return 0;

// khi đi ra khỏi phạm vi, ifstream sẽ tự đóng file

}

Kết quả:

This

is

line

This

is

line

Nhớ rằng toán tử tách stream vẫn có định dạng output và tự ngắt khi gặp kí tự trắng. Để đọc toàn bộ 1 dòng, phải dùng std::getline()

#include <fstream>

#include <iostream>

#include <string>

#include <cstdlib> // for exit()

int main()

{

using namespace std;

ifstream inf("Sample.dat");

// trường hợp không mở được file

if (!inf)

{

// xuất lỗi và exit

cerr << "Uh oh, Sample.dat could not be opened for reading!" << endl;

exit(1);

}

// Dùng while check EOF

while (inf)

{

std::string strInput;

getline(inf, strInput);

cout << strInput << endl;

}

return 0;

}

This produces the result:

This is line 1

This is line 2

##### Bộ đệm output

Quá trình ghi dữ liệu có thể thông qua bộ đệm trước khi thực sự ghi lên đĩa, lý do là để tối ưu hiệu suất. Khi bộ đệm ghi lên đĩa, quá trình này gọi là flush bộ đệm.

Khi đóng file thì tự động bộ đệm tự động được flush.

Quá trình đệm dữ liệu không phải là vấn đề, nhưng sự cố có thể xảy ra do sự bất cẩn. Nhất là trường hợp là khi dữ liệu nằm trong bộ đệm, ứng dụng bị ngắt đột ngột (do crash hay exit()), khi đó hàm hủy chưa thực thi, tức những file này vẫn đang trong tình trạng được mở (file đang bị khóa), bộ đệm vẫn chưa được flush.

Do đó, trước khi exit() phải đóng tất cả file lại hoặc chúng ta có thể thủ công flush dùng std::flush cho output stream.

Có một mẹo nhỏ nữa là khi dùng manip std::endl cũng gây flush output stream, đây cũng là nguyên nhân việc lạm dụng std::endl làm giảm hiệu suất khi dùng với bộ đệm I/O để ghi file.

Vì lý do này, các lập trình viên thường dùng kí tự trắng ‘\n’ để ngắt dòng thay vì std::endl.

##### Chế độ mở file

Hàm dựng của file stream nhận tham số thứ hai cho phép bạn cung cấp thông tin về cách file được mở, các thông tin này được trừu tượng hóa bởi các flag sau:

|  |  |
| --- | --- |
| **ios file mode** | **Meaning** |
| **ios::app** | Mở file và đặt con trỏ file vào cuối để ghi tiếp, không thể chuyển con trỏ file khỏi vị trí cuối file |
| **ios::ate** | Con trỏ file đặt ở kết file trước khi đọc/ghi, nhưng cho phép chuyển con trỏ tùy ý trong file |
| **ios::binary** | Mở theo chế độ nhị phân |
| **ios::in** | Mở file ở chế độ đọc ( mặc định của ifstream) |
| **ios::out** | Mở file ở chế độ ghi (mặc định cho ofstream) |
| **ios::trunc** | Xóa file nếu đã tồn tại |

* Chúng ta có quyền kết hợp nhiều flag này cùng nhau dùng toán tử OR (|).
* Lưu ý rằng ios::in và ios::out là chế độ mở mặc định.
* Nếu dùng fstream (vừa input và output), thì phải tường minh truyền tham số flag ios::in hay ios::out tùy theo mục đích sử dụng.

Bên dưới là ứng dụng nhỏ ghi 2 dòng liên tiếp vào file Sample.dat

#include <cstdlib> // for exit()

#include <iostream>

#include <fstream>

int main()

{

using namespace std;

// truyền flag ios:app để ofstream ghi tiếp vào file thay vì ghi đè, ios::out là mặc định

ofstream outf("Sample.dat", ios::app);

// trường hợp không thể mở file

if (!outf)

{

// xuất lỗi và exit

cerr << "Uh oh, Sample.dat could not be opened for writing!" << endl;

exit(1);

}

outf << "This is line 3" << endl;

outf << "This is line 4" << endl;

return 0;

// ofstream tự động close file khi ra khỏi phạm vi

}

Nội dụng file Sample.dat:

This is line 1

This is line 2

This is line 3

This is line 4

##### Mở file tường minh dùng open()

Cũng giống như việc đóng file tường minh dùng close(), open() cũng hoạt động giống với hàm dựng của file stream.

Ví dụ:

#include <cstdlib> // for exit()

#include <iostream>

#include <fstream>

using namespace std;

ofstream outf("Sample.dat");

outf << "This is line 1" << endl;

outf << "This is line 2" << endl;

outf.close(); //tường minh đóng file

// Oops, we forgot something

outf.open("Sample.dat", ios::app);

outf << "This is line 3" << endl;

outf.close();

##### I/O nhị phân

Khi mở file mà không ghi rõ chế độ mở file, trình biên dịch sẽ chuyển về chế độ mở file text. Vì vậy, khi muốn mở ở chế độ binary, ta cần phải ghi rõ ràng mod làios::binary**.** Việc mở file dưới chế độ binary có nghĩa là chúng ta sẽ đọc thông tin theo dạng binary file. Chúng ta sẽ sử dụng hàm write và read để đọc và ghi file dưới dạng binary.

1. Std::istream::read()

read(const char\* buffer, size\_t n);

Đọc n byte từ stream vào bộ đệm buffer

1. Std::ostream::write()

write(char\* buffer, size\_t n);

Ghi n byte từ bộ đệm buffer vào stream.Thường n là sizeof(buffer)

Ví dụ ta có đoạn mã sau:

1. #include <fstream>
3. using namespace std;
5. int main()
6. {
7. char data[100];
9. fstream f;
10. f.open("text.txt", fstream::in | fstream::binary);
11. if (!f)
12. return 1;
14. f.read(data, sizeof(data));
15. f.close();
17. return 0;
18. }
19. Std::istream::get()

Đọc 1 byte/kí tự từ stream vào tham số **c.**

Cú pháp:

istream &get(char &c)

Ví dụ

// typewriter

#include <iostream> // std::cin, std::cout

#include <fstream> // std::ofstream

int main () {

std::ofstream outfile ("test.txt");

char ch;

std::cout << "Type some text (type a dot to finish):\n";

do {

ch = std::cin.get();

outfile.put(ch);

} while (ch!='.');

return 0;

}

1. Std::ostream::put()

Ghi 1 byte/kí tự vào stream

Cú pháp:

ostream &put(char c)

Ví dụ:

// istream::get example

#include <iostream> // std::cin, std::cout

#include <fstream> // std::ifstream

int main () {

char str[256];

std::cout << "Enter the name of an existing text file: ";

std::cin.get (str,256); // get c-string

std::ifstream is(str); // open file

char c;

while (is.get(c)) // loop getting single characters

std::cout << c;

is.close(); // close file

return 0;

}

##### Truy xuất file ngẫu nhiên

1. Con trỏ file

Mỗi class stream chứa con trỏ file được dùng để giám sát vị trí đọc/ghi trong file, các thao tác đọc/ ghi sẽ bắt đầu từ vị trí của con trỏ file. Mặc định, khi mở file, con trỏ file sẽ ở đầu file, tuy nhiên nếu file được mở ở chế độ ios::app thì con trỏ sẽ được chuyển xuống cuối file.

1. Truy xuất ngẫu nhiên với seekg() và seekp()

Khi nội dụng file của bạn là một tập hợp nhiều recoed và muốn lấy đúng một record nào đó, bạn phải dùng con trỏ file nhảy tới đúng vị trí bắt đầu của record đó thay vì đọc từ đầu.

Đây là cách truy xuất file ngẫu nhiên, để làm được điều này, chúng ta dùng

* **seedkg**(Offset, ios::seek\_flag) : cho input , **g** viết tắt của **get**
* **seekp**(Offset, ios::seek\_flag) : cho output, **p** viết tắt của **put**

Cả 2 hàm đều nhận 2 tham số: tham số đầu tiên là số byte hay offset để di chuyển con trỏ file, tham số thứ hai là ios flag qui định vị trí khởi điểm trước khi di chuyển con trỏ file

|  |  |
| --- | --- |
| **ios seek flag** | **Meaning** |
| **ios::beg** | Di chuyển từ đầu file(mặc định) |
| **ios::cur** | Di chuyển tại vị trí hiện tại của con trỏ file |
| **ios::end** | Di chuyển từ cuối file |

* Offset > 0 : di chuyển con trỏ về cuối file
* Offset <0 : di chuyển con trỏ về đầu file

Ví dụ:

#include <cstdlib> // for exit()

#include <iostream>

#include <fstream>

using namespace std;

ofstream inf("Sample.dat");

inf.seekg(14, ios::cur); // nhảy về cuối file 14 bytes

inf.seekg(-18, ios::cur); // nhảy về đầu file 18 bytes

inf.seekg(22, ios::beg); // nhảy về byte thứ 22 tính đầu file

inf.seekg(24); // nhảy về cuối 24 byte tính đầu file

inf.seekg(-28, ios::end); // nhảy về đầu file 28 byte tính từ cuối file

Khi muốn di chuyển con trỏ về đầu hay cuối thì chỉ cần set Offset = 0 và chọn tham số flag ios phù hợp:

inf.seekg(0, ios::beg); // nhảy về đầu file

inf.seekg(0, ios::end); // nhảy về cuối file

Xét nội dung file Sample.dat:

This is line 1

This is line 2

This is line 3

This is line 4

Ví dụ:

int main()

{

using namespace std;

ifstream inf("Sample.dat");

// trường hợp không thể mở file

if (!inf)

{

// xuất lỗi và exit

cerr << "Uh oh, Sample.dat could not be opened for reading!" << endl;

exit(1);

}

string strData;

inf.seekg(5); // move to 5th character

// Get the rest of the line and print it

getline(inf, strData);

cout << strData << endl;

inf.seekg(8, ios::cur); // move 8 more bytes into file

// Get rest of the line and print it

getline(inf, strData);

cout << strData << endl;

inf.seekg(-15, ios::end); // move 15 bytes before end of file

// Get rest of the line and print it

getline(inf, strData);

cout << strData << endl;

return 0;

}

Kết quả:

is line 1

line 2

his is line 4

Lưu ý: một số trình biên dịch có lỗi khi triển khai seekg() và tellg() khi dùng kết hợp với file text (vì lý do bộ đệm). Vậy thì thử mở file theo chế độ binary

ifstream inf("Sample.dat", ifstream::binary);

Hai hàm tellg() và tellp(), sẽ trả về vị trí của con trỏ file, chúng ta có thể dùng để xác định kích thước file:

ifstream inf("Sample.dat");

inf.seekg(0, ios::end); // di chuyển tới cuối file

cout << inf.tellg();// xuất ra 64-bytes kích thước của Sample.dat

1. Kết hợp đọc và ghi file cùng lúc dùng fstream

Lớp fstream có khả năng vừa đọc, vừa ghi file cùng lúc, nhưng chúng ta không được đọc và ghi một cách tùy tiện được.

Khi đang thao tác đọc hay ghi, chỉ có một cách chuyển thao tác là di chuyển con trỏ file

* **seedkg**(Offset, ios::seek\_flag) : cho input , khi muốn chuyển sang chế độ đọc
* **seekp**(Offset, ios::seek\_flag) : cho output, khi muốn chuyển sang chế độ ghi

Nếu bạn không thực sự muốn di chuyển con trỏ file thì chỉ cần seek luôn vị trí hiện tại.

#include <iostream>

#include <fstream>

using namespace std;

fstream iofile("Sample.dat");

iofile.seekg(iofile.tellg(), ios::beg); // seek vị trí hiện tại của con trỏ file

Còn một bẫy khác: khác với ifstream, không thể dùng vòng lặp while(ifstream) để check EOF đối với fstream. Thay vào đó phải dùng fstream::get() , trả về kí tự hoặc EOF.

Ví dụ về file I/O dùng fstream, mở file, đọc dữ liệu và thay đổi bất kì nguyên âm nào với biểu tượng ‘#’

#include <cstdlib> // for exit()

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

using namespace std;

// chúng ta sẽ dùng cả flag ios::in và ios::out vì chúng ta đang dùng fstream

fstream iofile("Sample.dat", ios::in | ios::out);

// Xử lí khi không mở được file

if (!iofile)

{

cerr << "Uh oh, Sample.dat could not be opened!" << endl;

exit(1);

}

char chChar; // chúng ta sẽ duyệt từng kí tự

// Duyệt khi vẫn còn kí tự

while (iofile.get(chChar))

{

switch (chChar)

{

// khi tìm thấy nguyên âm

case 'a':

case 'e':

case 'i':

case 'o':

case 'u':

case 'A':

case 'E':

case 'I':

case 'O':

case 'U':

// Lùi 1 cho kí tự nguyên âm

iofile.seekg(-1, ios::cur);

// Vì chúng ta đã seek nên có thể write

iofile << '#'; // ghi đè # lên nguyên âm

// Bây giờ cần quay trở lại chế độ đọc. Chúng ta sẽ lại seekg() tới vị trí hiện tại

iofile.seekg(iofile.tellg(), ios::beg);

break;

}

}

return 0;

}

1. 1 số hàm hữu ích

Để xóa file chỉ cần dùng phương thức remove().

is\_open() để kiểm tra xem stream đang trong trạng thái mở file hay không.

1. Cảnh báo khi ghi con trỏ vào đĩa cứng

Khi stream các biến vào file khá dễ thì lại phức tạp hơn cho con trỏ. Nhớ rằng các con trỏ đơn giản là giữa địa chỉ của biến mà nó trỏ tới.

Mặc dù có thể đọc và ghi địa chỉ vào đĩa, nhưng nguy hiểm vì địa chỉ của biến có thể thay đổi trong quá trình thực thi.

Luật: không được ghi địa chỉ con trỏ vào file. Các biến nằm trong các địa chỉ có thể được đánh địa chỉ khác nhau khi đọc ngược lại từ file, do đó chúng không còn hợp lệ.

### Đối số dòng lệnh trong C, C++

C cho phép truyền tham số từ command line vào chương trình. Tham số dòng lệnh command line khá quan trọng cho chương trình nếu bạn cần kiểm soát chương trình từ bên ngoài thay vì từ mã nguồn bên trong.

Đối số dòng lệnh truyền vào qua hàm **int main( int argc, char \*argv[] )** , **argc** là số lượng đối số được truyền và mảng con trỏ **argv[]** trỏ tới mỗi đối số truyền vào.

Ví dụ:

#include <stdio.h>

int main( int argc, char \*argv[] ) {

if( argc == 2 )

{

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 )

{

printf("Too many arguments supplied.\n");

}

else

{

printf("One argument expected.\n");

}

}

Kết quả:

Trường hợp truyền 2 đối số:

$./a.out testing

The argument supplied is testing

Trường hợp 2 đối số truyền vào:

$./a.out testing1 testing2

Too many arguments supplied.

Trường hợp không truyền đố số:

$./a.out

One argument expected

**Argv[0]** có chứa tên của chương trình, có thể rỗng nếu không tồn tại tên chương trình và **argv[1]**  là con trỏ trỏ tới đối số đầu tiên command line cung cấp. Bạn có thể dùng mỗi phần tử \*argv[] như 1 chuỗi, hoặc như một mảng 2 chiều.

Nếu không có đối số truyền vào, argc sẽ bằng 1. Bạn truyền vào n đối số truyền vào, argc sẽ tăng thành n+1. Argv[argc] sẽ là con trỏ NULL

Khi nhập từ command line, mỗi đối số cách nhau bằng khoảng trắng, nếu đối số có chứa khoảng trắng, đặt đối số trong ngoặc kép.

Ví dụ

#include <stdio.h>

int main( int argc, char \*argv[] )

{

printf("Program name %s\n", argv[0]);

if( argc == 2 )

{

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 )

{

printf("Too many arguments supplied.\n");

}

else

{

printf("One argument expected.\n");

}

}

Kết quả:

$./a.out "testing1 testing2"

Progranm name ./a.out

The argument supplied is testing1 testing2

Xét một ví dụ khác.

Một trong những tác vụ phổ biến trong môi trường command line là xuất nội dung một file lên màn hình.

#include <stdio.h>

int main ( int argc, char \*argv[] )

{

if ( argc != 2 ) /\* argc should be 2 for correct execution \*/

{

/\* We print argv[0] assuming it is the program name \*/

printf( "usage: %s filename", argv[0] );

}

else

{

// We assume argv[1] is a filename to open

FILE \*file = fopen( argv[1], "r" );

/\* fopen returns 0, the NULL pointer, on failure \*/

if ( file == 0 )

{

printf( "Could not open file\n" );

}

else

{

int x;

/\* read one character at a time from file, stopping at EOF, which

indicates the end of the file. Note that the idiom of "assign

to a variable, check the value" used below works because

the assignment statement evaluates to the value assigned. \*/

while ( ( x = fgetc( file ) ) != EOF )

{

printf( "%c", x );

}

fclose( file );

}

}

}

## C,C++ Library and Dictionary:

Thư viện chuẩn của C++ sẽ không làm việc với thư viện của C do có sự khác biệt về định nghĩa kiểu dữ liệu.

### Thư viện hàm chuẩn:

Thư viện này được phát triển với nhiều mục đích, các hàm độc lập không phải thành viên bất kì lớp nào. Thư viện hàm được phân cấp từ C

Gồm các mục sau

* I/O
* String and character handling
* Mathematical
* Time, date, and localization
* Dynamic allocation
* Miscellaneous
* Wide-character functions

##### Stdio.h ( C )

1. Library Variables

Các kiểu được định nghĩa trong thư viện

|  |  |
| --- | --- |
| **S.N.** | **Variable & Description** |
| 1 | **size\_t**  Kiểu số, được dùng bởi **sizeof** . |
| 2 | **FILE**  Đối tượng chứa thông tin cho file stream |
| 3 | **fpos\_t**  Đối tượng chứa thông tin vị trí truy xuất hiện tại trong stream. |

1. Library Macros

Các Macro định nghĩa trong thư viện

|  |  |
| --- | --- |
| **S.N.** | **Macro & Description** |
| 1 | **NULL**  This macro is the value of a null pointer constant. |
| 2 | **\_IOFBF, \_IOLBF** and **\_IONBF**  These are the macros which expand to integral constant expressions with distinct values and suitable for the use as third argument to the **setvbuf** function. |
| 3 | **BUFSIZ**  This macro is an integer, which represents the size of the buffer used by the **setbuf** function. |
| 4 | **EOF**  This macro is a negative integer, which indicates that the end-of-file has been reached. |
| 5 | **FOPEN\_MAX**  This macro is an integer, which represents the maximum number of files that the system can guarantee to be opened simultaneously. |
| 6 | **FILENAME\_MAX**  This macro is an integer, which represents the longest length of a char array suitable for holding the longest possible filename. If the implementation imposes no limit, then this value should be the recommended maximum value. |
| 7 | **L\_tmpnam**  This macro is an integer, which represents the longest length of a char array suitable for holding the longest possible temporary filename created by the **tmpnam** function. |
| 8 | **SEEK\_CUR, SEEK\_END,** and **SEEK\_SET**  These macros are used in the **fseek** function to locate different positions in a file. |
| 9 | **TMP\_MAX**  This macro is the maximum number of unique filenames that the function**tmpnam** can generate. |
| 10 | **stderr, stdin,** and **stdout**  These macros are pointers to FILE types which correspond to the standard error, standard input, and standard output streams. |

1. Library Functions

Following are the functions defined in the header stdio.h −

Follow the same sequence of functions for better understanding and to make use of **Try it**(Online compiler) option, because file created in the first function will be used in subsequent functions.

|  |  |
| --- | --- |
| **S.N.** | **Function & Description** |
| 1 | [**int fclose(FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_fclose.htm)  Closes the stream. All buffers are flushed. |
| 2 | [**void clearerr(FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_clearerr.htm)  Clears the end-of-file and error indicators for the given stream. |
| 3 | [**int feof(FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_feof.htm)  Tests the end-of-file indicator for the given stream. |
| 4 | [**int ferror(FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_ferror.htm)  Tests the error indicator for the given stream. |
| 5 | [**int fflush(FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_fflush.htm)  Flushes the output buffer of a stream. |
| 6 | [**int fgetpos(FILE \*stream, fpos\_t \*pos)**](http://www.tutorialspoint.com/c_standard_library/c_function_fgetpos.htm)  Gets the current file position of the stream and writes it to pos. |
| 7 | [**FILE \*fopen(const char \*filename, const char \*mode)**](http://www.tutorialspoint.com/c_standard_library/c_function_fopen.htm)  Opens the filename pointed to by filename using the given mode. |
| 8 | [**size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_fread.htm)  Reads data from the given stream into the array pointed to by ptr. |
| 9 | [**FILE \*freopen(const char \*filename, const char \*mode, FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_freopen.htm)  Associates a new filename with the given open stream and same time closing the old file in stream. |
| 10 | [**int fseek(FILE \*stream, long int offset, int whence)**](http://www.tutorialspoint.com/c_standard_library/c_function_fseek.htm)  Sets the file position of the stream to the given offset. The argument *offset*signifies the number of bytes to seek from the given *whence* position. |
| 11 | [**int fsetpos(FILE \*stream, const fpos\_t \*pos)**](http://www.tutorialspoint.com/c_standard_library/c_function_fsetpos.htm)  Sets the file position of the given stream to the given position. The argument*pos* is a position given by the function fgetpos. |
| 12 | [**long int ftell(FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_ftell.htm)  Returns the current file position of the given stream. |
| 13 | [**size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_fwrite.htm)  Writes data from the array pointed to by ptr to the given stream. |
| 14 | [**int remove(const char \*filename)**](http://www.tutorialspoint.com/c_standard_library/c_function_remove.htm)  Deletes the given filename so that it is no longer accessible. |
| 15 | [**int rename(const char \*old\_filename, const char \*new\_filename)**](http://www.tutorialspoint.com/c_standard_library/c_function_rename.htm)  Causes the filename referred to, by old\_filename to be changed to new\_filename. |
| 16 | [**void rewind(FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_rewind.htm)  Sets the file position to the beginning of the file of the given stream. |
| 17 | [**void setbuf(FILE \*stream, char \*buffer)**](http://www.tutorialspoint.com/c_standard_library/c_function_setbuf.htm)  Defines how a stream should be buffered. |
| 18 | [**int setvbuf(FILE \*stream, char \*buffer, int mode, size\_t size)**](http://www.tutorialspoint.com/c_standard_library/c_function_setvbuf.htm)  Another function to define how a stream should be buffered. |
| 19 | [**FILE \*tmpfile(void)**](http://www.tutorialspoint.com/c_standard_library/c_function_tmpfile.htm)  Creates a temporary file in binary update mode (wb+). |
| 20 | [**char \*tmpnam(char \*str)**](http://www.tutorialspoint.com/c_standard_library/c_function_tmpnam.htm)  Generates and returns a valid temporary filename which does not exist. |
| 21 | [**int fprintf(FILE \*stream, const char \*format, ...)**](http://www.tutorialspoint.com/c_standard_library/c_function_fprintf.htm)  Sends formatted output to a stream. |
| 22 | [**int printf(const char \*format, ...)**](http://www.tutorialspoint.com/c_standard_library/c_function_printf.htm)  Sends formatted output to stdout. |
| 23 | [**int sprintf(char \*str, const char \*format, ...)**](http://www.tutorialspoint.com/c_standard_library/c_function_sprintf.htm)  Sends formatted output to a string. |
| 24 | [**int vfprintf(FILE \*stream, const char \*format, va\_list arg)**](http://www.tutorialspoint.com/c_standard_library/c_function_vfprintf.htm)  Sends formatted output to a stream using an argument list. |
| 25 | [**int vprintf(const char \*format, va\_list arg)**](http://www.tutorialspoint.com/c_standard_library/c_function_vprintf.htm)  Sends formatted output to stdout using an argument list. |
| 26 | [**int vsprintf(char \*str, const char \*format, va\_list arg)**](http://www.tutorialspoint.com/c_standard_library/c_function_vsprintf.htm)  Sends formatted output to a string using an argument list. |
| 27 | [**int fscanf(FILE \*stream, const char \*format, ...)**](http://www.tutorialspoint.com/c_standard_library/c_function_fscanf.htm)  Reads formatted input from a stream. |
| 28 | [**int scanf(const char \*format, ...)**](http://www.tutorialspoint.com/c_standard_library/c_function_scanf.htm)  Reads formatted input from stdin. |
| 29 | [**int sscanf(const char \*str, const char \*format, ...)**](http://www.tutorialspoint.com/c_standard_library/c_function_sscanf.htm)  Reads formatted input from a string. |
| 30 | [**int fgetc(FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_fgetc.htm)  Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream. |
| 31 | [**char \*fgets(char \*str, int n, FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_fgets.htm)  Reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. |
| 32 | [**int fputc(int char, FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_fputc.htm)  Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream. |
| 33 | [**int fputs(const char \*str, FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_fputs.htm)  Writes a string to the specified stream up to but not including the null character. |
| 34 | [**int getc(FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_getc.htm)  Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream. |
| 35 | [**int getchar(void)**](http://www.tutorialspoint.com/c_standard_library/c_function_getchar.htm)  Gets a character (an unsigned char) from stdin. |
| 36 | [**char \*gets(char \*str)**](http://www.tutorialspoint.com/c_standard_library/c_function_gets.htm)  Reads a line from stdin and stores it into the string pointed to by, str. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first. |
| 37 | [**int putc(int char, FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_putc.htm)  Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream. |
| 38 | [**int putchar(int char)**](http://www.tutorialspoint.com/c_standard_library/c_function_putchar.htm)  Writes a character (an unsigned char) specified by the argument char to stdout. |
| 39 | [**int puts(const char \*str)**](http://www.tutorialspoint.com/c_standard_library/c_function_puts.htm)  Writes a string to stdout up to but not including the null character. A newline character is appended to the output. |
| 40 | [**int ungetc(int char, FILE \*stream)**](http://www.tutorialspoint.com/c_standard_library/c_function_ungetc.htm)  Pushes the character char (an unsigned char) onto the specified stream so that the next character is read. |
| 41 | [**void perror(const char \*str)**](http://www.tutorialspoint.com/c_standard_library/c_function_perror.htm)  Prints a descriptive error message to stderr. First the string str is printed followed by a colon and then a space. |

##### Iostream.h (C++)

This header is part of the [Input/output](http://en.cppreference.com/w/cpp/io) library.

|  |
| --- |
|  |

|  |  |
| --- | --- |
| 1. Includes | |
| [<ios>](http://en.cppreference.com/w/cpp/header/ios) | (since C++11) |
| [<streambuf>](http://en.cppreference.com/w/cpp/header/streambuf) | (since C++11) |
| [<istream>](http://en.cppreference.com/w/cpp/header/istream) | (since C++11) |
| [<ostream>](http://en.cppreference.com/w/cpp/header/ostream) | (since C++11) |
| 1. Objects | |
| [std::cin](http://en.cppreference.com/w/cpp/io/cin) | standard input |
| [std::cout](http://en.cppreference.com/w/cpp/io/cout) | standard output |
| [std::cerr](http://en.cppreference.com/w/cpp/io/cerr) | standard error |
| [std::clog](http://en.cppreference.com/w/cpp/io/clog) | standard log |
| [std::wcin](http://en.cppreference.com/w/cpp/io/cin) | standard input |
| [std::wcout](http://en.cppreference.com/w/cpp/io/cout) | standard output |
| [std::wcerr](http://en.cppreference.com/w/cpp/io/cerr) | standard error |
| [std::wclog](http://en.cppreference.com/w/cpp/io/clog) | standard log |

1. [[edit](http://en.cppreference.com/mwiki/index.php?title=cpp/header/iostream&action=edit&section=1)] Synopsis

#include <ios>

#include <streambuf>

#include <istream>

#include <ostream>

namespace std {

extern istream cin;

extern ostream cout;

extern ostream cerr;

extern ostream clog;

extern wistream wcin;

extern wostream wcout;

extern wostream wcerr;

extern wostream wclog;

}

##### Stdlib.h ( C )

1. Library Variables

Following are the variable types defined in the header stdlib.h:

|  |  |
| --- | --- |
| **S.N.** | **Variable & Description** |
| 1 | **size\_t**  This is the unsigned integral type and is the result of the **sizeof** keyword. |
| 2 | **wchar\_t**  This is an integer type of the size of a **wide** character constant. |
| 3 | **div\_t**  This is the structure returned by the **div** function. |
| 4 | **ldiv\_t**  This is the structure returned by the **ldiv** function. |

1. Library Macros

Following are the macros defined in the header stdlib.h:

|  |  |
| --- | --- |
| **S.N.** | **Macro & Description** |
| 1 | **NULL**  This macro is the value of a null pointer constant. |
| 2 | **EXIT\_FAILURE**  Tham số cho hàm **exit(),** trường hợp gọi chương trình gặp lỗi. |
| 3 | **EXIT\_SUCCESS**  Tham số cho hàm **exit(),** trường hợp gọi chương trình không gặp lỗi gì. |
| 4 | **RAND\_MAX**  Giá trị giả về lớn nhất cho hàm **rand()**. |
| 5 | **MB\_CUR\_MAX**  This macro is the maximum number of bytes in a multi-byte character set which cannot be larger than MB\_LEN\_MAX. |

1. Library Functions

Following are the functions defined in the header stdio.h:

|  |  |
| --- | --- |
| **S.N.** | **Function & Description** |
| 1 | [**double atof(const char \*str)**](http://www.tutorialspoint.com/c_standard_library/c_function_atof.htm)  Converts the string pointed to, by the argument *str* to a floating-point number (type double). |
| 2 | [**int atoi(const char \*str)**](http://www.tutorialspoint.com/c_standard_library/c_function_atoi.htm)  Converts the string pointed to, by the argument *str* to an integer (type int). |
| 3 | [**long int atol(const char \*str)**](http://www.tutorialspoint.com/c_standard_library/c_function_atol.htm)  Converts the string pointed to, by the argument *str* to a long integer (type long int). |
| 4 | [**double strtod(const char \*str, char \*\*endptr)**](http://www.tutorialspoint.com/c_standard_library/c_function_strtod.htm)  Converts the string pointed to, by the argument *str* to a floating-point number (type double). |
| 5 | [**long int strtol(const char \*str, char \*\*endptr, int base)**](http://www.tutorialspoint.com/c_standard_library/c_function_strtol.htm)  Converts the string pointed to, by the argument *str* to a long integer (type long int). |
| 6 | [**unsigned long int strtoul(const char \*str, char \*\*endptr, int base)**](http://www.tutorialspoint.com/c_standard_library/c_function_strtoul.htm)  Converts the string pointed to, by the argument *str* to an unsigned long integer (type unsigned long int). |
| 7 | [**void \*calloc(size\_t nitems, size\_t size)**](http://www.tutorialspoint.com/c_standard_library/c_function_calloc.htm)  Allocates the requested memory and returns a pointer to it. |
| 8 | [**void free(void \*ptr**](http://www.tutorialspoint.com/c_standard_library/c_function_free.htm)  Deallocates the memory previously allocated by a call to *calloc, malloc,* or*realloc*. |
| 9 | [**void \*malloc(size\_t size)**](http://www.tutorialspoint.com/c_standard_library/c_function_malloc.htm)  Allocates the requested memory and returns a pointer to it. |
| 10 | [**void \*realloc(void \*ptr, size\_t size)**](http://www.tutorialspoint.com/c_standard_library/c_function_realloc.htm)  Attempts to resize the memory block pointed to by ptr that was previously allocated with a call to *malloc* or *calloc*. |
| 11 | [**void abort(void)**](http://www.tutorialspoint.com/c_standard_library/c_function_abort.htm)  Causes an abnormal program termination. |
| 12 | [**int atexit(void (\*func)(void))**](http://www.tutorialspoint.com/c_standard_library/c_function_atexit.htm)  Causes the specified function **func** to be called when the program terminates normally. |
| 13 | [**void exit(int status)**](http://www.tutorialspoint.com/c_standard_library/c_function_exit.htm)  Causes the program to terminate normally. |
| 14 | [**char \*getenv(const char \*name)**](http://www.tutorialspoint.com/c_standard_library/c_function_getenv.htm)  Searches for the environment string pointed to by name and returns the associated value to the string. |
| 15 | [**int system(const char \*string)**](http://www.tutorialspoint.com/c_standard_library/c_function_system.htm)  The command specified by string is passed to the host environment to be executed by the command processor. |
| 16 | [**void \*bsearch(const void \*key, const void \*base, size\_t nitems, size\_t size, int (\*compar)(const void \*, const void \*))**](http://www.tutorialspoint.com/c_standard_library/c_function_bsearch.htm)  Performs a binary search. |
| 17 | [**void qsort(void \*base, size\_t nitems, size\_t size, int (\*compar)(const void \*, const void\*))**](http://www.tutorialspoint.com/c_standard_library/c_function_qsort.htm)  Sorts an array. |
| 18 | [**int abs(int x)**](http://www.tutorialspoint.com/c_standard_library/c_function_abs.htm)  Returns the absolute value of x. |
| 19 | [**div\_t div(int numer, int denom)**](http://www.tutorialspoint.com/c_standard_library/c_function_div.htm)  Divides numer (numerator) by denom (denominator). |
| 20 | [**long int labs(long int x)**](http://www.tutorialspoint.com/c_standard_library/c_function_labs.htm)  Returns the absolute value of x. |
| 21 | [**ldiv\_t ldiv(long int numer, long int denom)**](http://www.tutorialspoint.com/c_standard_library/c_function_ldiv.htm)  Divides numer (numerator) by denom (denominator). |
| 22 | [**int rand(void)**](http://www.tutorialspoint.com/c_standard_library/c_function_rand.htm)  Return về số ngẫu nhiên từ 0 tới *RAND\_MAX*. |
| 23 | [**void srand(unsigned int seed)**](http://www.tutorialspoint.com/c_standard_library/c_function_srand.htm)  Hàm này khởi phát bộ sinh số ngẫu nhiên dùng bởi hàm **rand()** |
| 24 | [**int mblen(const char \*str, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_mblen.htm)  Returns the length of a multibyte character pointed to by the argument *str*. |
| 25 | [**size\_t mbstowcs(schar\_t \*pwcs, const char \*str, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_mbstowcs.htm)  Converts the string of multibyte characters pointed to by the argument *str* to the array pointed to by *pwcs*. |
| 26 | [**int mbtowc(whcar\_t \*pwc, const char \*str, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_mbtowc.htm)  Examines the multibyte character pointed to by the argument *str*. |
| 27 | [**size\_t wcstombs(char \*str, const wchar\_t \*pwcs, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_wcstombs.htm)  Converts the codes stored in the array *pwcs* to multibyte characters and stores them in the string *str*. |
| 28 | [**int wctomb(char \*str, wchar\_t wchar)**](http://www.tutorialspoint.com/c_standard_library/c_function_wctomb.htm)  Examines the code which corresponds to a multibyte character given by the argument *wchar*. |

##### time.h ( C )

1. Library Variables

|  |  |
| --- | --- |
| **S.N.** | **Variable & Description** |
| 1 | **size\_t**  This is the unsigned integral type and is the result of the **sizeof** keyword. |
| 2 | **clock\_t**  This is a type suitable for storing the processor time. |
| 3 | **time\_t**  This is a type suitable for storing the calendar time. |
| 4 | **struct tm**  This is a structure used to hold the time and date. |

1. Library Macros

|  |  |
| --- | --- |
| **S.N.** | **Macro & Description** |
| 1 | **NULL**  This macro is the value of a null pointer constant. |
| 2 | **CLOCKS\_PER\_SEC**  This macro represents the number of processor clocks per second. |

1. Library Functions

|  |  |
| --- | --- |
| **SN** | **Function & Purpose** |
| 1 | **time\_t time(time\_t \*time);**  Trả về số giây tính từ ngày 1-jan-1970 tới giờ hiện tại của hệ thống  Nếu hệ thống không có thời gian, sẽ trả về “1” |
| 2 | **char \*ctime(const time\_t \*time);**  Trả về con trỏ kiểu chuỗi dạng: *day month year hours:minutes:seconds year\n\0*. |
| 3 | **struct tm \*localtime(const time\_t \*time);**  Trả về ngày giờ của địa phương theo kiểu cấu trúc **tm** |
| 4 | **clock\_t clock(void);**  Trả về giá trị gần đúng thời gian chương trình chạy. Trả về “1” nếu hệ thống không có thời gian |
| 5 | **char \* asctime ( const struct tm \* time );**  Trả về con trỏ kiểu chuỗi là thông tin chứa trong cấu trúc **tm** được trỏ tới theo dạng: *day month date hours:minutes:seconds year\n\0* |
| 6 | **struct tm \*gmtime(const time\_t \*time);**  Trả về con trỏ kiểu cấu trúc **tm.** Thời gian được biểu diễn ở dạng UTC , bản chất vẫn là GMT  Coordinated Universal Time (UTC) Greenwich Mean Time (GMT). |
| 7 | **time\_t mktime(struct tm \*time);**  Trả về calendar-time tương đương với thời gian tìm thấy từ con trỏ cấu trúc |
| 8 | **double difftime ( time\_t time2, time\_t time1 );**  Trả về sai biệt theo giây giữa time1 và time2. |
| 9 | **size\_t strftime();**  Hàm được dùng để dịnh dạng ngày tháng theo nhiều kiểu |

##### Stdarg.h ( C )

The **stdarg.h** header defines a variable type **va\_list** and three macros which can be used to get the arguments in a function when the number of arguments are not known i.e. variable number of arguments.

A function of variable arguments is defined with the ellipsis (,...) at the end of the parameter list.

1. Library Variables

Following is the variable type defined in the header stdarg.h −

|  |  |
| --- | --- |
| **S.N.** | **Variable & Description** |
| 1 | **va\_list**  This is a type suitable for holding information needed by the three macros **va\_start(), va\_arg()** and **va\_end()**. |

1. Library Macros

Following are the macros defined in the header stdarg.h −

|  |  |
| --- | --- |
| **S.N.** | **Macro & Description** |
| 1 | [**void va\_start(va\_list ap, last\_arg)**](http://www.tutorialspoint.com/c_standard_library/c_macro_va_start.htm)  This macro initializes **ap** variable to be used with the **va\_arg** and **va\_end** macros. The **last\_arg** is the last known fixed argument being passed to the function i.e. the argument before the ellipsis. |
| 2 | [**type va\_arg(va\_list ap, type)**](http://www.tutorialspoint.com/c_standard_library/c_macro_va_arg.htm)  This macro retrieves the next argument in the parameter list of the function with type **type**. |
| 3 | [**void va\_end(va\_list ap)**](http://www.tutorialspoint.com/c_standard_library/c_macro_va_end.htm)  This macro allows a function with variable arguments which used the **va\_start** macro to return. If **va\_end** is not called before returning from the function, the result is undefined. |

##### String.h ( C++ )

Kiểu #typedef trong string.h:

|  |  |
| --- | --- |
| **S.N.** | **Variable & Description** |
| 1 | **size\_t**  This is the unsigned integral type and is the result of the **sizeof** keyword. |

1. Library Macros

Macros định nghĩa trong string.h:

|  |  |
| --- | --- |
| **S.N.** | **Macro & Description** |
| 1 | **NULL**  This macro is the value of a null pointer constant. |

1. Library Functions

Danh sách hàm:

|  |  |
| --- | --- |
| **S.N.** | **Function & Description** |
| 1 | [**void \*memchr(const void \*str, int c, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_memchr.htm)  Searches for the first occurrence of the character c (an unsigned char) in the first n bytes of the string pointed to, by the argument *str*. |
| 2 | [**int memcmp(const void \*str1, const void \*str2, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_memcmp.htm)  Compares the first n bytes of *str1* and *str2*. |
| 3 | [**void \*memcpy(void \*dest, const void \*src, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_memcpy.htm)  Copies n characters from src to *dest*. |
| 4 | [**void \*memmove(void \*dest, const void \*src, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_memmove.htm)  Another function to copy n characters from *str2* to *str1*. |
| 5 | [**void \*memset(void \*str, int c, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_memset.htm)  Copies the character c (an unsigned char) to the first n characters of the string pointed to, by the argument *str*. |
| 6 | [**char \*strcat(char \*dest, const char \*src)**](http://www.tutorialspoint.com/c_standard_library/c_function_strcat.htm)  Appends the string pointed to, by *src* to the end of the string pointed to by *dest*. |
| 7 | [**char \*strncat(char \*dest, const char \*src, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_strncat.htm)  Appends the string pointed to, by *src* to the end of the string pointed to, by *dest* up to n characters long. |
| 8 | [**char \*strchr(const char \*str, int c)**](http://www.tutorialspoint.com/c_standard_library/c_function_strchr.htm)  Searches for the first occurrence of the character c (an unsigned char) in the string pointed to, by the argument *str*. |
| 9 | [**int strcmp(const char \*str1, const char \*str2)**](http://www.tutorialspoint.com/c_standard_library/c_function_strcmp.htm)  Compares the string pointed to, by *str1* to the string pointed to by *str2*. |
| 10 | [**int strncmp(const char \*str1, const char \*str2, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_strncmp.htm)  Compares at most the first n bytes of *str1* and *str2*. |
| 11 | [**int strcoll(const char \*str1, const char \*str2)**](http://www.tutorialspoint.com/c_standard_library/c_function_strcoll.htm)  Compares string *str1* to *str2*. The result is dependent on the LC\_COLLATE setting of the location. |
| 12 | [**char \*strcpy(char \*dest, const char \*src)**](http://www.tutorialspoint.com/c_standard_library/c_function_strcpy.htm)  Copies the string pointed to, by *src* to *dest*. |
| 13 | [**char \*strncpy(char \*dest, const char \*src, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_strncpy.htm)  Copies up to n characters from the string pointed to, by *src* to *dest*. |
| 14 | [**size\_t strcspn(const char \*str1, const char \*str2)**](http://www.tutorialspoint.com/c_standard_library/c_function_strcspn.htm)  Calculates the length of the initial segment of str1 which consists entirely of characters not in str2. |
| 15 | [**char \*strerror(int errnum)**](http://www.tutorialspoint.com/c_standard_library/c_function_strerror.htm)  Searches an internal array for the error number errnum and returns a pointer to an error message string. |
| 16 | [**size\_t strlen(const char \*str)**](http://www.tutorialspoint.com/c_standard_library/c_function_strlen.htm)  Computes the length of the string str up to but not including the terminating null character. |
| 17 | [**char \*strpbrk(const char \*str1, const char \*str2)**](http://www.tutorialspoint.com/c_standard_library/c_function_strpbrk.htm)  Finds the first character in the string *str1* that matches any character specified in *str2*. |
| 18 | [**char \*strrchr(const char \*str, int c)**](http://www.tutorialspoint.com/c_standard_library/c_function_strrchr.htm)  Searches for the last occurrence of the character c (an unsigned char) in the string pointed to by the argument *str*. |
| 19 | [**size\_t strspn(const char \*str1, const char \*str2)**](http://www.tutorialspoint.com/c_standard_library/c_function_strspn.htm)  Calculates the length of the initial segment of *str1* which consists entirely of characters in *str2*. |
| 20 | [**char \*strstr(const char \*haystack, const char \*needle)**](http://www.tutorialspoint.com/c_standard_library/c_function_strstr.htm)  Finds the first occurrence of the entire string *needle* (not including the terminating null character) which appears in the string *haystack*. |
| 21 | [**char \*strtok(char \*str, const char \*delim)**](http://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm)  Breaks string *str* into a series of tokens separated by *delim*. |
| 22 | [**size\_t strxfrm(char \*dest, const char \*src, size\_t n)**](http://www.tutorialspoint.com/c_standard_library/c_function_strxfrm.htm)  Transforms the first **n** characters of the string **src** into corrent locale and places them in the string **dest**. |

##### Errno.h ( C )

**errno.h** header file định nghĩa biến số nguyên **errn.** Macro này được xếp vào loại Lvalue (do có kiểu integer), do đó **errn** có thể đọc hay thay đổi bới chương trình.

**Errno** được set = 0 ngay đầu chương trình. Và thông qua một hàm trong thư viện chuẩn của C sẽ giúp bạn đổi giá trị của **errno** để báo hiệu kiểu lỗi.

**Header file** errno.hcũng định nghĩa 1 danh sách các marcos chỉ ra các mã lỗi khác nhau. Các macros này được triển khai thành các hằng số nguyên integer.

1. Library Macros

|  |  |
| --- | --- |
| **S.N.** | **Macro & Description** |
| 1 | [**extern int errno**](http://www.tutorialspoint.com/c_standard_library/c_macro_errno.htm)  Là 1 Macro được thiết lập bởi hệ thống và 1 vài thư viện hàm sự kiện liên quan tới xử lí error.  Macro này giúp xác định error là do cái gì gây ra. |
| 2 | [**EDOM Domain Error**](http://www.tutorialspoint.com/c_standard_library/c_macro_edom.htm)  This macro represents a domain error, which occurs if an input argument is outside the domain, over which the mathematical function is defined and errno is set to EDOM. |
| 3 | [**ERANGE Range Error**](http://www.tutorialspoint.com/c_standard_library/c_macro_erange.htm)  This macro represents a range error, which occurs if an input argument is outside the range, over which the mathematical function is defined and errno is set to ERANGE. |

Danh sách định nghĩa macros:

|  |  |  |
| --- | --- | --- |
| #define EPERM | 1 | /\* Operation not permitted \*/ |
| #define ENOENT | 2 | /\* No such file or directory \*/ |
| #define ESRCH | 3 | /\* No such process \*/ |
| #define EINTR | 4 | /\* Interrupted system call \*/ |
| #define EIO | 5 | /\* Input/output error \*/ |
| #define ENXIO | 6 | /\* Device not configured \*/ |
| #define E2BIG | 7 | /\* Argument list too long \*/ |
| #define ENOEXEC | 8 | /\* Exec format error \*/ |
| #define EBADF | 9 | /\* Bad file descriptor \*/ |
| #define ECHILD | 10 | /\* No child processes \*/ |
| #define EDEADLK | 11 | /\* Resource deadlock avoided \*/ |
|  |  | /\* 11 was EAGAIN \*/ |
| #define ENOMEM | 12 | /\* Cannot allocate memory \*/ |
| #define EACCES | 13 | /\* Permission denied \*/ |
| #define EFAULT | 14 | /\* Bad address \*/ |
| **#ifndef \_POSIX\_SOURCE** | | |
| #define ENOTBLK | 15 | /\* Block device required \*/ |
| #define EBUSY | 16 | /\* Device busy \*/ |
| **#endif** | | |
| #define EEXIST | 17 | /\* File exists \*/ |
| #define EXDEV | 18 | /\* Cross-device link \*/ |
| #define ENODEV | 19 | /\* Operation not supported by device \*/ |
| #define ENOTDIR | 20 | /\* Not a directory \*/ |
| #define EISDIR | 21 | /\* Is a directory \*/ |
| #define EINVAL | 22 | /\* Invalid argument \*/ |
| #define ENFILE | 23 | /\* Too many open files in system \*/ |
| #define EMFILE | 24 | /\* Too many open files \*/ |
| #define ENOTTY | 25 | /\* Inappropriate ioctl for device \*/ |
| **#ifndef \_POSIX\_SOURCE** | | |
| #define ETXTBSY | 26 | /\* Text file busy \*/ |
| **#endif** | | |
| #define EFBIG | 27 | /\* File too large \*/ |
| #define ENOSPC | 28 | /\* No space left on device \*/ |
| #define ESPIPE | 29 | /\* Illegal seek \*/ |
| #define EROFS | 30 | /\* Read-only file system \*/ |
| #define EMLINK | 31 | /\* Too many links \*/ |
| #define EPIPE | 32 | /\* Broken pipe \*/ |
|  |  |  |
| /\* math software \*/ |  |  |
| #define EDOM | 33 | /\* Numerical argument out of domain \*/ |
| #define ERANGE | 34 | /\* Result too large \*/ |
|  |  |  |
| /\* non-blocking and interrupt i/o \*/ |  |  |
| #define EAGAIN | 35 | /\* Resource temporarily unavailable \*/ |
|  | | |
| **#ifndef \_POSIX\_SOURCE** | | |
| #define EWOULDBLOCKEAGAIN |  | /\* Operation would block \*/ |
| #define EINPROGRESS | 36 | /\* Operation now in progress \*/ |
| #define EALREADY | 37 | /\* Operation already in progress \*/ |
|  |  |  |
| /\* ipc/network software -- argument errors \*/ |  |  |
| #define ENOTSOCK | 38 | /\* Socket operation on non-socket \*/ |
| #define EDESTADDRREQ | 39 | /\* Destination address required \*/ |
| #define EMSGSIZE | 40 | /\* Message too long \*/ |
| #define EPROTOTYPE | 41 | /\* Protocol wrong type for socket \*/ |
| #define ENOPROTOOPT | 42 | /\* Protocol not available \*/ |
| #define EPROTONOSUPPORT | 43 | /\* Protocol not supported \*/ |
| #define ESOCKTNOSUPPORT | 44 | /\* Socket type not supported \*/ |
| #define EOPNOTSUPP | 45 | /\* Operation not supported on socket \*/ |
| #define EPFNOSUPPORT | 46 | /\* Protocol family not supported \*/ |
| #define EAFNOSUPPORT | 47 | /\* Address family not supported by protocol family \*/ |
| #define EADDRINUSE | 48 | /\* Address already in use \*/ |
| #define EADDRNOTAVAIL | 49 | /\* Can't assign requested address \*/ |
|  |  |  |
| /\* ipc/network software -- operational errors \*/ |  |  |
| #define ENETDOWN | 50 | /\* Network is down \*/ |
| #define ENETUNREACH | 51 | /\* Network is unreachable \*/ |
| #define ENETRESET | 52 | /\* Network dropped connection on reset \*/ |
| #define ECONNABORTED | 53 | /\* Software caused connection abort \*/ |
| #define ECONNRESET | 54 | /\* Connection reset by peer \*/ |
| #define ENOBUFS | 55 | /\* No buffer space available \*/ |
| #define EISCONN | 56 | /\* Socket is already connected \*/ |
| #define ENOTCONN | 57 | /\* Socket is not connected \*/ |
| #define ESHUTDOWN | 58 | /\* Can't send after socket shutdown \*/ |
| #define ETOOMANYREFS | 59 | /\* Too many references: can't splice \*/ |
| #define ETIMEDOUT | 60 | /\* Connection timed out \*/ |
| #define ECONNREFUSED | 61 | /\* Connection refused \*/ |
|  |  |  |
| #define ELOOP | 62 | /\* Too many levels of symbolic links \*/ |
| #endif /\* \_POSIX\_SOURCE \*/ |  |  |
| #define ENAMETOOLONG | 63 | /\* File name too long \*/ |
|  |  |  |
| /\* should be rearranged \*/ |  |  |
| #ifndef \_POSIX\_SOURCE |  |  |
| #define EHOSTDOWN | 64 | /\* Host is down \*/ |
| #define EHOSTUNREACH | 65 | /\* No route to host \*/ |
| **#endif /\* \_POSIX\_SOURCE \*/** | | |
| #define ENOTEMPTY | 66 | /\* Directory not empty \*/ |
|  |  |  |
| /\* quotas & mush \*/ |  |  |
| #ifndef \_POSIX\_SOURCE |  |  |
| #define EPROCLIM | 67 | /\* Too many processes \*/ |
| #define EUSERS | 68 | /\* Too many users \*/ |
| #define EDQUOT | 69 | /\* Disc quota exceeded \*/ |
|  |  |  |
| /\* Network File System \*/ |  |  |
| #define ESTALE | 70 | /\* Stale NFS file handle \*/ |
| #define EREMOTE | 71 | /\* Too many levels of remote in path \*/ |
| #define EBADRPC | 72 | /\* RPC struct is bad \*/ |
| #define ERPCMISMATCH | 73 | /\* RPC version wrong \*/ |
| #define EPROGUNAVAIL | 74 | /\* RPC prog. not avail \*/ |
| #define EPROGMISMATCH | 75 | /\* Program version wrong \*/ |
| #define EPROCUNAVAIL | 76 | /\* Bad procedure for program \*/ |
| #endif /\* \_POSIX\_SOURCE \*/ |  |  |
|  |  |  |
| #define ENOLCK | 77 | /\* No locks available \*/ |
| #define ENOSYS | 78 | /\* Function not implemented \*/ |
|  |  |  |
| #define EFTYPE | 79 | /\* Inappropriate file type or format \*/ |
|  |  |  |
| **#ifdef KERNEL** | | |
| /\* pseudo-errors returned inside kernel to modify return to process \*/ |  |  |
| #define ERESTART | -1 | /\* restart syscall \*/ |
| #define EJUSTRETURN | -2 | /\* don't modify regs, just return \*/ |
| **#endif** | | |

### Thư viện dành cho lớp đối tượng:

This is a collection of classes and associated functions.

Standard C++ Object Oriented Library defines an extensive set of classes that provide support for a number of common activities, including I/O, strings, and numeric processing. This library includes the following:

* The Standard C++ I/O Classes
* The String Class
* The Numeric Classes
* The STL Container Classes
* The STL Algorithms
* The STL Function Objects
* The STL Iterators
* The STL Allocators
* The Localization library
* Exception Handling Classes
* Miscellaneous Support Library

### Bảng mã ASCII

Ký tự là một hình vẽ, một biểu tượng, và nó dường như không hề có một quy luật cụ thể nào cả. Do đó, để giúp cho máy tính có thể hiểu và hiển thị được ký tự, con người đã nghĩ ra một công cụ. Đó chính là bảng mã ASCII.

1. ASCII là gì?

ASCII là viết tắt của cụm từ "American Standard Code for Information Interchange". Như các bạn đã biết, máy tính chỉ có thể hiểu được các con số, do đó, một mã ASCII là đại diện của một ký tự chẳng hạn như là ‘A’ hay là ‘@’ hoặc một số tín hiệu khác. Tuy nhiên, giờ đây các tín hiệu điều khiển đó đã ít được sử dụng so với thời điểm mới ra đời của ASCII. Ngoài ra, còn rất nhiều ký tự đặc biệt khác để hỗ trợ nhiều loại ngôn ngữ khác nhau, và đó chính là lý do ra đời của bảng mã ASCII mở rộng.

1. Bảng mã ASCII

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DEC** | **BIN** | **HEX** | **OCT** | **HTML** | **SYM** | **DEC** | **BIN** | **HEX** | **OCT** | **HTML** | **SYM** |
| **0** | 00000000 | 0x00 | 000 | &#0; | NUL | **64** | 01000000 | 0x40 | 100 | &#64; | @ |
| **1** | 00000001 | 0x01 | 001 | &#1; | SOH | **65** | 01000001 | 0x41 | 101 | &#65; | A |
| **2** | 00000010 | 0x02 | 002 | &#2; | STX | **66** | 01000010 | 0x42 | 102 | &#66; | B |
| **3** | 00000011 | 0x03 | 003 | &#3; | ETX | **67** | 01000011 | 0x43 | 103 | &#67; | C |
| **4** | 00000100 | 0x04 | 004 | &#4; | EOT | **68** | 01000100 | 0x44 | 104 | &#68; | D |
| **5** | 00000101 | 0x05 | 005 | &#5; | ENQ | **69** | 01000101 | 0x45 | 105 | &#69; | E |
| **6** | 00000110 | 0x06 | 006 | &#6; | ACK | **70** | 01000110 | 0x46 | 106 | &#70; | F |
| **7** | 00000111 | 0x07 | 007 | &#7; | BEL | **71** | 01000111 | 0x47 | 107 | &#71; | G |
| **8** | 00001000 | 0x08 | 010 | &#8; | BS | **72** | 01001000 | 0x48 | 110 | &#72; | H |
| **9** | 00001001 | 0x09 | 011 | &#9; | TAB | **73** | 01001001 | 0x49 | 111 | &#73; | I |
| **10** | 00001010 | 0x0A | 012 | &#10; | LF | **74** | 01001010 | 0x4A | 112 | &#74; | J |
| **11** | 00001011 | 0x0B | 013 | &#11; | VT | **75** | 01001011 | 0x4B | 113 | &#75; | K |
| **12** | 00001100 | 0x0C | 014 | &#12; | FF | **76** | 01001100 | 0x4C | 114 | &#76; | L |
| **13** | 00001101 | 0x0D | 015 | &#13; | CR | **77** | 01001101 | 0x4D | 115 | &#77; | M |
| **14** | 00001110 | 0x0E | 016 | &#14; | SO | **78** | 01001110 | 0x4E | 116 | &#78; | N |
| **15** | 00001111 | 0x0F | 017 | &#15; | SI | **79** | 01001111 | 0x4F | 117 | &#79; | O |
| **16** | 00010000 | 0x10 | 020 | &#16; | DLE | **80** | 01010000 | 0x50 | 120 | &#80; | P |
| **17** | 00010001 | 0x11 | 021 | &#17; | DC1 | **81** | 01010001 | 0x51 | 121 | &#81; | Q |
| **18** | 00010010 | 0x12 | 022 | &#18; | DC2 | **82** | 01010010 | 0x52 | 122 | &#82; | R |
| **19** | 00010011 | 0x13 | 023 | &#19; | DC3 | **83** | 01010011 | 0x53 | 123 | &#83; | S |
| **20** | 00010100 | 0x14 | 024 | &#20; | DC4 | **84** | 01010100 | 0x54 | 124 | &#84; | T |
| **21** | 00010101 | 0x15 | 025 | &#21; | NAK | **85** | 01010101 | 0x55 | 125 | &#85; | U |
| **22** | 00010110 | 0x16 | 026 | &#22; | SYN | **86** | 01010110 | 0x56 | 126 | &#86; | V |
| **23** | 00010111 | 0x17 | 027 | &#23; | ETB | **87** | 01010111 | 0x57 | 127 | &#87; | W |
| **24** | 00011000 | 0x18 | 030 | &#24; | CAN | **88** | 01011000 | 0x58 | 130 | &#88; | X |
| **25** | 00011001 | 0x19 | 031 | &#25; | EM | **89** | 01011001 | 0x59 | 131 | &#89; | Y |
| **26** | 00011010 | 0x1A | 032 | &#26; | SUB | **90** | 01011010 | 0x5A | 132 | &#90; | Z |
| **27** | 00011011 | 0x1B | 033 | &#27; | ESC | **91** | 01011011 | 0x5B | 133 | &#91; | [ |
| **28** | 00011100 | 0x1C | 034 | &#28; | FS | **92** | 01011100 | 0x5C | 134 | &#92; | \ |
| **29** | 00011101 | 0x1D | 035 | &#29; | GS | **93** | 01011101 | 0x5D | 135 | &#93; | ] |
| **30** | 00011110 | 0x1E | 036 | &#30; | RS | **94** | 01011110 | 0x5E | 136 | &#94; | ^ |
| **31** | 00011111 | 0x1F | 037 | &#31; | US | **95** | 01011111 | 0x5F | 137 | &#95; | \_ |
| **32** | 00100000 | 0x20 | 040 | &#32; | SPACE | **96** | 01100000 | 0x60 | 140 | &#96; | ` |
| **33** | 00100001 | 0x21 | 041 | &#33; | ! | **97** | 01100001 | 0x61 | 141 | &#97; | a |
| **34** | 00100010 | 0x22 | 042 | &#34; | " | **98** | 01100010 | 0x62 | 142 | &#98; | b |
| **35** | 00100011 | 0x23 | 043 | &#35; | # | **99** | 01100011 | 0x63 | 143 | &#99; | c |
| **36** | 00100100 | 0x24 | 044 | &#36; | $ | **100** | 01100100 | 0x64 | 144 | &#100; | d |
| **37** | 00100101 | 0x25 | 045 | &#37; | % | **101** | 01100101 | 0x65 | 145 | &#101; | e |
| **38** | 00100110 | 0x26 | 046 | &#38; | & | **102** | 01100110 | 0x66 | 146 | &#102; | f |
| **39** | 00100111 | 0x27 | 047 | &#39; | ' | **103** | 01100111 | 0x67 | 147 | &#103; | g |
| **40** | 00101000 | 0x28 | 050 | &#40; | ( | **104** | 01101000 | 0x68 | 150 | &#104; | h |
| **41** | 00101001 | 0x29 | 051 | &#41; | ) | **105** | 01101001 | 0x69 | 151 | &#105; | i |
| **42** | 00101010 | 0x2A | 052 | &#42; | \* | **106** | 01101010 | 0x6A | 152 | &#106; | j |
| **43** | 00101011 | 0x2B | 053 | &#43; | + | **107** | 01101011 | 0x6B | 153 | &#107; | k |
| **44** | 00101100 | 0x2C | 054 | &#44; | , | **108** | 01101100 | 0x6C | 154 | &#108; | l |
| **45** | 00101101 | 0x2D | 055 | &#45; | - | **109** | 01101101 | 0x6D | 155 | &#109; | m |
| **46** | 00101110 | 0x2E | 056 | &#46; | . | **110** | 01101110 | 0x6E | 156 | &#110; | n |
| **47** | 00101111 | 0x2F | 057 | &#47; | / | **111** | 01101111 | 0x6F | 157 | &#111; | o |
| **48** | 00110000 | 0x30 | 060 | &#48; | 0 | **112** | 01110000 | 0x70 | 160 | &#112; | p |
| **49** | 00110001 | 0x31 | 061 | &#49; | 1 | **113** | 01110001 | 0x71 | 161 | &#113; | q |
| **50** | 00110010 | 0x32 | 062 | &#50; | 2 | **114** | 01110010 | 0x72 | 162 | &#114; | r |
| **51** | 00110011 | 0x33 | 063 | &#51; | 3 | **115** | 01110011 | 0x73 | 163 | &#115; | s |
| **52** | 00110100 | 0x34 | 064 | &#52; | 4 | **116** | 01110100 | 0x74 | 164 | &#116; | t |
| **53** | 00110101 | 0x35 | 065 | &#53; | 5 | **117** | 01110101 | 0x75 | 165 | &#117; | u |
| **54** | 00110110 | 0x36 | 066 | &#54; | 6 | **118** | 01110110 | 0x76 | 166 | &#118; | v |
| **55** | 00110111 | 0x37 | 067 | &#55; | 7 | **119** | 01110111 | 0x77 | 167 | &#119; | w |
| **56** | 00111000 | 0x38 | 070 | &#56; | 8 | **120** | 01111000 | 0x78 | 170 | &#120; | x |
| **57** | 00111001 | 0x39 | 071 | &#57; | 9 | **121** | 01111001 | 0x79 | 171 | &#121; | y |
| **58** | 00111010 | 0x3A | 072 | &#58; | : | **122** | 01111010 | 0x7A | 172 | &#122; | z |
| **59** | 00111011 | 0x3B | 073 | &#59; | ; | **123** | 01111011 | 0x7B | 173 | &#123; | { |
| **60** | 00111100 | 0x3C | 074 | &#60; | < | **124** | 01111100 | 0x7C | 174 | &#124; | | |
| **61** | 00111101 | 0x3D | 075 | &#61; | = | **125** | 01111101 | 0x7D | 175 | &#125; | } |
| **62** | 00111110 | 0x3E | 076 | &#62; | > | **126** | 01111110 | 0x7E | 176 | &#126; | ~ |
| **63** | 00111111 | 0x3F | 077 | &#63; | ? | **127** | 01111111 | 0x7F | 177 | &#127; | DEL |

1. Bảng mã ASCII mở rộng

Trải qua thời gian kỹ thuật và công nghệ thông tin phát triển, chiếc máy tính đến tay nhiều quốc gia trên thế giới, và do các ngôn ngữ có khá nhiều ký tự lạ nên đã ra đời Bảng mã ASCII mở rộng. Không như bảng mã ASCII truyền thống dùng 7 bit để biểu thị ký tự, bảng mã ASCII mở rộng sử dụng cả 8 bit. Điều này giúp cho nhiều ngôn ngữ có thể xuất hiện trên chiếc máy tính và giúp cho thị trường máy tính được mở rộng hơn. Bảng mã ASCII mở rộng lúc mới được công bố đã sinh ra thêm khá nhiều biến thể. Tuy nhiên, bởi vì Internet sử dụng chuẩn ký tự là ISO 8859-1 và hệ điều hành phổ biến là Microsoft Windows cũng sử dụng chuẩn tương tự nên bảng mã ASCII mở rộng theo chuẩn này được ngầm định là chuẩn.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DEC** | **BIN** | **HEX** | **OCT** | **HTML** | **SYM** | **DEC** | **BIN** | **HEX** | **OCT** | **HTML** | **SYM** |
| **128** | 10000000 | 0x80 | 200 | &#128; | € | **192** | 11000000 | 0xC0 | 300 | &#192; | À |
| **129** | 10000001 | 0x81 | 201 | &#129; |  | **193** | 11000001 | 0xC1 | 301 | &#193; | Á |
| **130** | 10000010 | 0x82 | 202 | &#130; | ‚ | **194** | 11000010 | 0xC2 | 302 | &#194; | Â |
| **131** | 10000011 | 0x83 | 203 | &#131; | ƒ | **195** | 11000011 | 0xC3 | 303 | &#195; | Ã |
| **132** | 10000100 | 0x84 | 204 | &#132; | „ | **196** | 11000100 | 0xC4 | 304 | &#196; | Ä |
| **133** | 10000101 | 0x85 | 205 | &#133; | … | **197** | 11000101 | 0xC5 | 305 | &#197; | Å |
| **134** | 10000110 | 0x86 | 206 | &#134; | † | **198** | 11000110 | 0xC6 | 306 | &#198; | Æ |
| **135** | 10000111 | 0x87 | 207 | &#135; | ‡ | **199** | 11000111 | 0xC7 | 307 | &#199; | Ç |
| **136** | 10001000 | 0x88 | 210 | &#136; | ˆ | **200** | 11001000 | 0xC8 | 310 | &#200; | È |
| **137** | 10001001 | 0x89 | 211 | &#137; | ‰ | **201** | 11001001 | 0xC9 | 311 | &#201; | É |
| **138** | 10001010 | 0x8A | 212 | &#138; | Š | **202** | 11001010 | 0xCA | 312 | &#202; | Ê |
| **139** | 10001011 | 0x8B | 213 | &#139; | ‹ | **203** | 11001011 | 0xCB | 313 | &#203; | Ë |
| **140** | 10001100 | 0x8C | 214 | &#140; | Œ | **204** | 11001100 | 0xCC | 314 | &#204; | Ì |
| **141** | 10001101 | 0x8D | 215 | &#141; |  | **205** | 11001101 | 0xCD | 315 | &#205; | Í |
| **142** | 10001110 | 0x8E | 216 | &#142; | Ž | **206** | 11001110 | 0xCE | 316 | &#206; | Î |
| **143** | 10001111 | 0x8F | 217 | &#143; |  | **207** | 11001111 | 0xCF | 317 | &#207; | Ï |
| **144** | 10010000 | 0x90 | 220 | &#144; |  | **208** | 11010000 | 0xD0 | 320 | &#208; | Ð |
| **145** | 10010001 | 0x91 | 221 | &#145; | ‘ | **209** | 11010001 | 0xD1 | 321 | &#209; | Ñ |
| **146** | 10010010 | 0x92 | 222 | &#146; | ’ | **210** | 11010010 | 0xD2 | 322 | &#210; | Ò |
| **147** | 10010011 | 0x93 | 223 | &#147; | “ | **211** | 11010011 | 0xD3 | 323 | &#211; | Ó |
| **148** | 10010100 | 0x94 | 224 | &#148; | ” | **212** | 11010100 | 0xD4 | 324 | &#212; | Ô |
| **149** | 10010101 | 0x95 | 225 | &#149; | • | **213** | 11010101 | 0xD5 | 325 | &#213; | Õ |
| **150** | 10010110 | 0x96 | 226 | &#150; | – | **214** | 11010110 | 0xD6 | 326 | &#214; | Ö |
| **151** | 10010111 | 0x97 | 227 | &#151; | — | **215** | 11010111 | 0xD7 | 327 | &#215; | × |
| **152** | 10011000 | 0x98 | 230 | &#152; | ˜ | **216** | 11011000 | 0xD8 | 330 | &#216; | Ø |
| **153** | 10011001 | 0x99 | 231 | &#153; | ™ | **217** | 11011001 | 0xD9 | 331 | &#217; | Ù |
| **154** | 10011010 | 0x9A | 232 | &#154; | š | **218** | 11011010 | 0xDA | 332 | &#218; | Ú |
| **155** | 10011011 | 0x9B | 233 | &#155; | › | **219** | 11011011 | 0xDB | 333 | &#219; | Û |
| **156** | 10011100 | 0x9C | 234 | &#156; | œ | **220** | 11011100 | 0xDC | 334 | &#220; | Ü |
| **157** | 10011101 | 0x9D | 235 | &#157; |  | **221** | 11011101 | 0xDD | 335 | &#221; | Ý |
| **158** | 10011110 | 0x9E | 236 | &#158; | ž | **222** | 11011110 | 0xDE | 336 | &#222; | Þ |
| **159** | 10011111 | 0x9F | 237 | &#159; | Ÿ | **223** | 11011111 | 0xDF | 337 | &#223; | ß |
| **160** | 10100000 | 0xA0 | 240 | &#160; |  | **224** | 11100000 | 0xE0 | 340 | &#224; | à |
| **161** | 10100001 | 0xA1 | 241 | &#161; | ¡ | **225** | 11100001 | 0xE1 | 341 | &#225; | á |
| **162** | 10100010 | 0xA2 | 242 | &#162; | ¢ | **226** | 11100010 | 0xE2 | 342 | &#226; | â |
| **163** | 10100011 | 0xA3 | 243 | &#163; | £ | **227** | 11100011 | 0xE3 | 343 | &#227; | ã |
| **164** | 10100100 | 0xA4 | 244 | &#164; | ¤ | **228** | 11100100 | 0xE4 | 344 | &#228; | ä |
| **165** | 10100101 | 0xA5 | 245 | &#165; | ¥ | **229** | 11100101 | 0xE5 | 345 | &#229; | å |
| **166** | 10100110 | 0xA6 | 246 | &#166; | ¦ | **230** | 11100110 | 0xE6 | 346 | &#230; | æ |
| **167** | 10100111 | 0xA7 | 247 | &#167; | § | **231** | 11100111 | 0xE7 | 347 | &#231; | ç |
| **168** | 10101000 | 0xA8 | 250 | &#168; | ¨ | **232** | 11101000 | 0xE8 | 350 | &#232; | è |
| **169** | 10101001 | 0xA9 | 251 | &#169; | © | **233** | 11101001 | 0xE9 | 351 | &#233; | é |
| **170** | 10101010 | 0xAA | 252 | &#170; | ª | **234** | 11101010 | 0xEA | 352 | &#234; | ê |
| **171** | 10101011 | 0xAB | 253 | &#171; | « | **235** | 11101011 | 0xEB | 353 | &#235; | ë |
| **172** | 10101100 | 0xAC | 254 | &#172; | ¬ | **236** | 11101100 | 0xEC | 354 | &#236; | ì |
| **173** | 10101101 | 0xAD | 255 | &#173; |  | **237** | 11101101 | 0xED | 355 | &#237; | í |
| **174** | 10101110 | 0xAE | 256 | &#174; | ® | **238** | 11101110 | 0xEE | 356 | &#238; | î |
| **175** | 10101111 | 0xAF | 257 | &#175; | ¯ | **239** | 11101111 | 0xEF | 357 | &#239; | ï |
| **176** | 10110000 | 0xB0 | 260 | &#176; | ° | **240** | 11110000 | 0xF0 | 360 | &#240; | ð |
| **177** | 10110001 | 0xB1 | 261 | &#177; | ± | **241** | 11110001 | 0xF1 | 361 | &#241; | ñ |
| **178** | 10110010 | 0xB2 | 262 | &#178; | ² | **242** | 11110010 | 0xF2 | 362 | &#242; | ò |
| **179** | 10110011 | 0xB3 | 263 | &#179; | ³ | **243** | 11110011 | 0xF3 | 363 | &#243; | ó |
| **180** | 10110100 | 0xB4 | 264 | &#180; | ´ | **244** | 11110100 | 0xF4 | 364 | &#244; | ô |
| **181** | 10110101 | 0xB5 | 265 | &#181; | µ | **245** | 11110101 | 0xF5 | 365 | &#245; | õ |
| **182** | 10110110 | 0xB6 | 266 | &#182; | ¶ | **246** | 11110110 | 0xF6 | 366 | &#246; | ö |
| **183** | 10110111 | 0xB7 | 267 | &#183; | · | **247** | 11110111 | 0xF7 | 367 | &#247; | ÷ |
| **184** | 10111000 | 0xB8 | 270 | &#184; | ¸ | **248** | 11111000 | 0xF8 | 370 | &#248; | ø |
| **185** | 10111001 | 0xB9 | 271 | &#185; | ¹ | **249** | 11111001 | 0xF9 | 371 | &#249; | ù |
| **186** | 10111010 | 0xBA | 272 | &#186; | º | **250** | 11111010 | 0xFA | 372 | &#250; | ú |
| **187** | 10111011 | 0xBB | 273 | &#187; | » | **251** | 11111011 | 0xFB | 373 | &#251; | û |
| **188** | 10111100 | 0xBC | 274 | &#188; | ¼ | **252** | 11111100 | 0xFC | 374 | &#252; | ü |
| **189** | 10111101 | 0xBD | 275 | &#189; | ½ | **253** | 11111101 | 0xFD | 375 | &#253; | ý |
| **190** | 10111110 | 0xBE | 276 | &#190; | ¾ | **254** | 11111110 | 0xFE | 376 | &#254; | þ |
| **191** | 10111111 | 0xBF | 277 | &#191; | ¿ | **255** | 11111111 | 0xFF | 377 | &#255; | ÿ |

1. Bảng mã ASCII và C,C++

Như các bạn đã biết, bảng mã ASCII thể hiện ký tự dưới dạng số, vậy chuyện gì xảy ra nếu như tôi chạy dòng lệnh sau?

1. #include <stdio.h>
3. int main()
4. {
5. printf("%c\n", -2);
6. return 0;
7. }

-2 trong dạng nhị phân sẽ được biểu hiện bằng 8 bit ở dạng bù 2 như sau: 1 1 1 1 1 1 1 0 . Khi tôi cho in giá trị -2 ra màn hình dưới dạng kiểu char, máy tính sẽ chuyển 1 1 1 1 1 1 1 0 sang số nguyên không âm (thành 254) và in ký tự mà giá trị 254 biểu thị ra màn hình.