C/C++ Advance

Linh La

## List must reading

Take the opportunity offered by the new C++11 facilities to modernize your design and programming techniques:

* [10] Use templates to maintain static type safety (eliminate casts) and avoid unnecessary use of class hierarchies (§27.2).

## OOP trong C++

Mục đích chính của C++ là thêm hướng đối tượng vào ngôn ngữ C và khái niệm về lớp là tính năng chính của C++, nhằm hỗ trợ lập trình hướng đối tượng mà chúng ta thường gọi là kiểu người dùng định nghĩa

Class được dùng để xác định một định dạng đối tượng và kết hợp dữ liệu , phương thức để xử lí dữ liệu thành một khối trật tự. Dữ liệu và hàm trong class được gọi là thành viên của lớp

### Khái niệm mở đầu

1. Định nghĩa lớp:

Khi định nghĩa lớp, bạn phải lên kế hoạch cho các kiểu dữ liệu. Việc này khác với định nghĩa dữ liệu, mà là định nghĩa lớp, định nghĩa tất cả những gì thuộc về class và các hoạt động liên quan tới nó.

Chúng ta bắt đầu định nghĩa với từ khóa **class,** theo sau là tên lớp, phần thân lớp được bao đóng trong cặp ngoặc nhọn.

Sau mỗi định nghĩa dữ liệu thành viên phải có dấu chấm phẩy.

Ví dụ định nghĩa lớp Box như thế này:

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

Từ khóa **Public** quy định thuộc tính truy xuất của thanh viên dữ liệu theo sau nó. Thành viên public có thể được truy xuất cả bên trong và bên ngoài phạm vi lớp.

1. Định nghĩa đối tượng:

Một lớp xem như bản thiết kế cho đối tượng, cơ bản là các đối tượng được tạo từ lớp. Chúng ta khai báo đối tượng tương tự như khai báo biến. Ví dụ:

Box Box1; // Box1 là kiểu Box

Box Box2; // Box2 là kiểu Box

Cả hai đối tượng Box1 và Box2 đều có bản sao thành viên dữ liệu của lớp.

1. Truy xuất thành viên dữ liệu:

Thành viên puplic của đối tượng truy xuất qua toán tử thành viên “.”. Ví dụ:

#include <iostream>

using namespace std;

class Box{

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main( ){

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 specification

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// volume of box 1

volume = Box1.height \* Box1.length \* Box1.breadth;

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.height \* Box2.length \* Box2.breadth;

cout << "Volume of Box2 : " << volume <<endl;

return 0;

}

Kết quả, thể tích Box1 và Box2

Volume of Box1 : 210

Volume of Box2 : 1560

Chú ý là thành viên private và protected không thể truy xuất trực tiếp bằng toán tử “.”, chương sau sẽ đề cập tới vấn đề này.

1. Hàm thành viên

Hàm thành viên là hàm được khai báo bên trong lớp. Nó có thể tương tác với mọi thuộc tính, phương thức của lớp, và có quyền truy xuất tới tất cả thành viên của lớp.

Ví dụ sau dùng hàm thành viên truy xuất gián tiếp thành viên dữ liệu thay vì trực tiếp dùng toán tử thành viên:

class Box

{

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

double getVolume(void);// Returns box volume

};

Hàm thành viên có thể định nghĩa bên trong lớp. Nếu định nghĩa ngoài lớp phải dùng toán tử phạm vi “::”.

Cú pháp:

<tên kiểu> <tên lớp> :: <tên hàm> (…tham số hàm..) {định nghĩa hàm }

Định nghĩa hàm thành viên bên trong lớp:

class Box

{

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

double getVolume(void) {

return length \* breadth \* height;

}

};

Hoặc bên ngoài lớp qua toán tử phạm vi, lưu ý là phải có tên lớp trước toán tử “::”

double Box::getVolume(void){

return length \* breadth \* height;

}

Đối tượng sẽ gọi hàm thành viên qua toán tử thành viên “.” như thế này:

Box myBox; // Create an object

myBox.getVolume(); // Call member function for the object

Chúng ta sẽ xây dựng một lớp Box chi tiết hơn, đoạn code sau hướng dẫn bạn thủ tục khai báo và định nghĩa:

#include <iostream>

using namespace std;

class Box{

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

// Member functions declaration

double getVolume(void);

void setLength( double len );

void setBreadth( double bre );

void setHeight( double hei );

};

// Member functions definitions

double Box::getVolume(void){

return length \* breadth \* height;

}

void Box::setLength( double len ){

length = len;

}

void Box::setBreadth( double bre ){

breadth = bre;

}

void Box::setHeight( double hei ){

height = hei;

}

// Main function for the program

int main( ){

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

return 0;

}

Kết quả:

Volume of Box1 : 210

Volume of Box2 : 1560

### Các thuộc tính truy xuất

Che dấu dữ liệu là một trong tính nằng quan trong của OOP (kể từ giờ chúng ta viết tắt hướng đối tượng là OOP), ngăn không cho các hàm trong chướng trình truy xuất tùy tiện thành viên của lớp.

Từ khóa **public, private,** và **protected** bên trong thân lớp, quy định thuộc tính truy xuất cho hàm và dữ liệu thành viên.

Trong một lớp có thể có nhiều đoạn **public**, **protected**, hay **private.** Những thành viên sẽ có thuộc tính truy xuất nằm trước chúng.

Thuộc tính truy xuất mặc định là **private** nếu bạn để trống.

Ví dụ

class Base {

public:

// public members go here

protected:

// protected members go here

private:

// private members go here

};

1. Thành viên public:

Thành viên public được truy xuất từ trong hay bên ngoài lớp, nhưng trong phạm vi chương trình. Bạn có thể lấy hay gán giá trị cho biến public mà không cần thông qua hàm thành viên lớp, ví dụ:

#include <iostream>

using namespace std;

class Line{

public:

double length;

void setLength( double len );

double getLength( void );

};

// Member functions definitions

double Line::getLength(void){

return length ;

}

void Line::setLength( double len ){

length = len;

}

// Main function for the program

int main( ){

Line line;

// set line length

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

// set line length without member function

line.length = 10.0; // OK: because length is public

cout << "Length of line : " << line.length <<endl;

return 0;

}

Kết quả

Length of line : 6

Length of line : 10

1. Thành viên private:

Thành viên private không thể truy xuất từ bên ngoài lớp. Chỉ có hàm thành viên hay hàm **friend** mới truy xuất được.

Mặc định mọi thành viên có đặc tính private nếu không khai báo, ví dụ **width** là thành viên private của lớp

class Box {

double width;

public:

double length;

void setWidth( double wid );

double getWidth( void );

};

Chúng ta làm bài thực hành nhỏ, định nghĩa thành viên private và hàm thành viên puplic giúp chương trình truy xuất dữ liệu private qua việc gọi hàm.

#include <iostream>

using namespace std;

class Box {

public:

double length;

void setWidth( double wid );

double getWidth( void );

private:

double width;

};

// Member functions definitions

double Box::getWidth(void){

return width ;

}

void Box::setWidth( double wid ){

width = wid;

}

// Main function for the program

int main( ){

Box box;

// set box length without member function

box.length = 10.0; // OK: because length is public

cout << "Length of box : " << box.length <<endl;

// set box width without member function

// box.width = 10.0; // Error: because width is private

box.setWidth(10.0); // Use member function to set it.

cout << "Width of box : " << box.getWidth() <<endl;

return 0;

}

Kết quả

Length of box : 10

Width of box : 10

1. Thành viên protected:

Thành viên protected có nét khá giống với private, tức là thành viên protected không thể truy xuất từ bên ngoài lớp nhưng có điểm khác là chúng được thừa kế và có thể truy xuất từ lớp con (hay còn gọi là lớp dẫn xuất)

Lớp dẫn xuất và thừa kế được bàn trong chương sau, bây giờ ta xét ví dụ về lớp con **SmallBox.** Thành viên **width** có thể truy xuất bởi bất kì hàm thành viên nào của lớp dẫn xuất.

#include <iostream>

using namespace std;

class Box{

protected:

double width;

};

class SmallBox:Box // SmallBox là lớp dẫn xuất.

{

public:

void setSmallWidth( double wid );

double getSmallWidth( void );

};

// Hàm thành viên của lớp con

double SmallBox::getSmallWidth(void){

return width ;

}

void SmallBox::setSmallWidth( double wid ){

width = wid;

}

// Main function for the program

int main( ) {

SmallBox box;

// set box width using member function

box.setSmallWidth(5.0);

cout << "Width of box : "<< box.getSmallWidth() << endl;

return 0;

}

Kết quả

Width of box : 5

1. Hàm truy xuất

Hàm thành viên giúp chúng ta gián tiếp truy xuất, chỉnh sửa các dữ liệu thành viên **private** được gọi là hàm truy xuất.

Ví dụ: hàm thành viên **setLength** (), **setBreath**() và **setHeight** () truy xuất tới các thành viên dữ liệu của lớp **Box**:

#include <iostream>

using namespace std;

class Box {

private:

double length;

double height;

double breath;

public:

//Set data member

void setLength(double l) {}

void setHeight(double h) {}

void setBreath(double b) {}

};

void Box::setLength(double l) {

length = l;

}

void Box::setHeight(double h) {

height = h;

}

void Box::setBreath(double b) {

breath = b;

}

1. Định nghĩa lớp trong file header

Để dùng định nghĩa lớp trong nhiều file của project, bạn nên khai báo chúng trong file header.

Thủ tục khai báo nằm trong classbox.h

#ifndef BOX\_H

#define BOX\_H

/\*

\* define class Box

\*/

class Box {

private:

double length;

double height;

double breath;

public:

void display() {}

/\*

\* Set data member

\*/

void setLength(double l) {}

void setHeight(double h) {}

void setBreath(double b) {}

//Show the volume

double getVolume() {}

};

#endif

Và phần định nghĩa nằm trong classbox.cpp

#include <iostream>

#include "classbox.h"

using namespace std;

void Box::display() {

cout << "Length: " << length << "; Height: " << height << "; Breath: " << breath << endl;

cout << "Volume: " << getVolume() << endl;

}

void Box::setLength(double l) {

length = l;

}

void Box::setHeight(double h) {

height = h;

}

void Box::setBreath(double b) {

breath = b;

}

//Show the volume

double Box::getVolume() {

return height \* breath\*length;

}

### Hàm dựng và hàm hủy của lớp

1. Hàm dựng mặc định:

Hàm dựng của lớp là một hàm đặc biệt, được tự động thực thi bất kì lúc nào chúng ta tạo ra một đối tượng mới của lớp.

Hàm dựng sẽ có tên trùng với tên lớp và không trả về gì cả, thậm chí là **void.** Hàm dựng có thể rất hữu dụng khi dùng khởi tạo giá trị cho thành viên dữ liệu

Xem ví dụ sau

#include <iostream>

using namespace std;

class Line {

public:

void setLength( double len );

double getLength( void );

Line(); // Hàm dựng mặc định

private:

double length;

};

// Member functions definitions including constructor

Line::Line(void) {

Length = 10; // khai báo giá trị mặc định cho thành viên dữ liệu

cout << "Object is being created" << endl;

}

void Line::setLength( double len ) {

length = len;

}

double Line::getLength( void ) {

return length;

}

// Main function for the program

int main( ) {

Line line;

// set line length

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

return 0;

}

Kết quả

Object is being created

Length of line : 6

Hàm dựng không có đối số ở ví dụ trên gọi là hàm dựng mặc định, thông thường giá trị mặc định của biến thành viên sẽ được khởi tạo trong hàm dựng mặc định.

1. Hàm dựng có tham số:

Mặc định hàm dựng không có bất kì tham số nào, nhưng nếu bạn cần, chúng vẫn có thể có. Việc này giúp bạn gán giá trị khởi đầu cho đối tượng tạt thời điểm nó mới tạo.

Xem ví dụ

#include <iostream>

using namespace std;

class Line {

public:

void setLength( double len );

double getLength( void );

Line(double len); // Hàm dựng

private:

double length;

};

// Member functions definitions including constructor

Line::Line( double len) {

cout << "Object is being created, length = " << len << endl;

length = len;

}

void Line::setLength( double len ) {

length = len;

}

double Line::getLength( void ) {

return length;

}

// Main function for the program

int main( )

{

Line line(10.0);

// get initially set length.

cout << "Length of line : " << line.getLength() <<endl;

// set line length again

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

return 0;

}

Kết quả

Object is being created, length = 10

Length of line : 10

Length of line : 6

1. Danh sách khởi tạo hàm dựng:

Trong trường hợp hàm dựng có tham số, bạn có thể dùng cú pháp sau để khởi tạo giá trị cho các biến thành viên:

Line::Line( double len): length(len){

cout << "Object is being created, length = " << len << endl;

}

Cú pháp trên cũng tương tự như bên dưới đây

Line::Line( double len){

cout << "Object is being created, length = " << len << endl;

length = len;

}

Giả sử ta có lớp C, với nhiều trường X,Y,Z … cần khởi tạo, dùng cú pháp trên, mỗi trường cách nhau bởi dấu phẩy

C::C( double a, double b, double c): X(a), Y(b), Z(c){

....

}

1. Khởi tạo cho hằng và tham chiếu:

Như chúng ta đã biết, biến hằng và biến tham chiếu bắt buộc phải khởi tạo giá trị khi khai báo, nếu bạn khởi tạo bằng phép gán trong hàm dựng, chương trình sẽ bị lỗi:

class Line {

private:

const double length;

// Khai báo hàm dựng

Public:

Line(double l){

Length = l; //Lỗi, biến hằng không thể là lvalue

/\*đoạn code trên cũng tương tự như:  
\* const double length;  
\* length = l;  
\*/

}

};

Sử dụng danh sách khởi tạo chính là cách khởi tạo ngầm, tức khởi tạo giá trị cho thành viên dữ liệu mà không thực hiện toán tử gán trong hàm dựng:

Đoạn code sau là hợp lệ:

class Line {

private:

const double length;

// Khai báo hàm dựng

Public:

Line():length(0){}

}

};

Chúng ta khuyến khích dùng danh sách khởi tạo thay vì dùng phép gán trong hàm dựng.

1. Danh sách khởi tạo chuẩn trong C++11

Nếu bạn dùng trình biên dịch chuẩn C++ 11, bạn phải dùng cú pháp mới cho danh sách khởi tạo, đặc biệt khi kết hợp hay thừa kế lớp:

class Line {

private:

const double length;

// Khai báo hàm dựng

Public:

Line(double l):length{l} {} // dùng cặp ngoặc nhọn

}

};

1. Danh sách khởi tạo cho mảng (C++11)

Định nghĩa lớp **something**, có thành viên dữ liệu mảng **m\_array**[5]. set zero tất cả phần tử mảng bằng danh sách khởi tạo hàm dựng như sau :

class something {

private:

double m\_array[5];

// Khai báo hàm dựng

Public:

something():m\_array{} // zero all elements

{// khởi tạo giá trị phần tử ở đây nếu muốn.}

};

Liệt kê từng giá trị cách nhau bằng dấu phẩy, bao đóng trong cặp ngoặc nhọn để khởi tạo cho từng phần tử:

class something {

private:

double m\_array[5];

// Khai báo hàm dựng

Public:

something():m\_array{1,2,3,4,5} // Dùng cú pháp khởi tạo chuẩn C++ 11

{

}

};

Danh sách khởi tạo là phương tiện duy nhất để khởi tạo thành viên có kiểu dữ liệu hằng và tham chiếu.

1. Hàm dựng sao chép

Hàm dựng sao chép giúp tạo ra một đối tượng bằng cách sao chép đối tượng cùng lớp đã có trước. Mục đích hàm dựng sao chép thường là:

* Khởi tạo một đối tượng từ một đối tượng cùng kiểu
* Truyền kiểu giá trị một đối tượng vào đối số một hàm
* Sao chép một đối tượng rồi trả về trong một hàm

Nếu hàm dựng sao chép không được định nghĩa trong lớp, trình biên dịch tự định nghĩa một cái. Nếu lớp có biến con trỏ và được cấp phát tĩnh, thì lớp đó phải có hàm dựng sao chép.

Cú pháp hàm dựng sao chép là:

<tên\_lớp> (const <tên\_lớp> &<biến>) { // thân hàm }

Ở đây, <biến>là tham chiếu tới đối tượng được dùng cho việc khởi tạo đối tượng khác.

#include <iostream>

using namespace std;

class Line {

public:

Line( int len ); // hàm dựng tham số

Line( const Line &obj); // hàm dựng sao chép

~Line(); // hàm hủy mặc định

private:

int \*ptr;

};

// Member functions definitions including constructor

Line::Line(int len) {

cout << "Normal constructor allocating ptr" << endl;

// allocate memory for the pointer;

ptr = new int;

\*ptr = len;

}

Line::Line(const Line &obj) {

cout << "Copy constructor allocating ptr." << endl;

ptr = new int;

\*ptr = \*obj.ptr; // copy the value

}

void display(Line obj) {

cout << "Length of line : " << \*obj.ptr <<endl;

}

// Main function for the program

int main( ) {

Line line(10); //Hàm dựng tham số được gọi

display(line); //Hàm dựng sao chép được gọi

return 0;

}

Kết quả

Normal constructor allocating ptr

Copy constructor allocating ptr.

Length of line : 10

Đối tượng thuộc lớp Line khi được tạo sẽ gọi hàm dựng. Trình biên dịch sẽ gọi hàm dựng sao chép của line1 , tạo ra bản sao đối tượng rồi truyền vào hàm **display()**

Kết thúc hàm **display()**, hàm hủy đối tượng sao chép được gọi.

1. Toán tử gán mặc định

Khi bạn dùng phép gán hai đối tượng với nhau, nếu không quá tải phép gán thì C++ sẽ gọi quá tải mặc định của toán tử gán, tức sao chép mọi thành viên từ đối tượng này sang đối tượng kia.

Trường hợp đối tượng được gán ngay khi khai báo thì hàm dựng sao chép sẽ được gọi.

Xét ví dụ sau:

// Main function for the program

int main( ){

Line line1(10);

Line line2 = line1; // Chỉ hàm dựng sao chép của line2 sẽ được gọi

display(line1);

display(line2);

return 0;

}

Kết quả

Normal constructor allocating ptr //hàm dựng line 1 được gọi

Copy constructor allocating ptr. // hàm dựng sao chép line2 được gọi

Copy constructor allocating ptr. // hàm dựng sao chép line1 được gọi truyền tham số vào hàm

Length of line : 10

Freeing memory! // hàm hủy của đối tượng sao chép line1

Copy constructor allocating ptr. // hàm dựng sao chép line2 được gọi truyền tham số vào hàm

Length of line : 10

Freeing memory! // hàm hủy của đối tượng sao chép line2

Freeing memory! // hàm hủy line1

Freeing memory! // hàm hủy line2

1. Khởi tạo thành viên non-static (C++ 11)

Trước kia, trình biên dịch không cho phép khởi tạo giá trị mặc định cho dữ liệu thành viên non-static, mà phải khai báo qua hàm dựng.

Trường hợp lớp có quá nhiều hàm dựng, khi cần chỉnh giá trị mặc định một biến thì phải chỉnh tất cả hàm dựng.

C++11 cho phép thành viên dữ liệu non-static khởi tạo luôn giá trị mặc định khi khai báo:

#include <iostream>;

using namespace std;

class Square

{

private:

double m\_length = 1.0; // m\_length có giá trị mặc định = 1.0

double m\_width = 1.0; // m\_width có giá trị mặc định = 1.0

public:

Square() {

/\* Hàm dựng này sẽ dùng giá trị mặc định nếu không gán giá trị ở đây hoặc không cung cấp  
 \* danh sách khởi tạo

\*/

}

void print() {

cout << "length: " << m\_length << ", width: " << m\_width << '\n';

}

};

int main(){

Square x; // x.m\_length = 1.0, x.m\_width = 1.0

x.print();

return 0;

}

Kết quả:

length: 1.0, width: 1.0

Dù khởi tạo giá trị mặc định cho thành viên dữ liệu, nhưng hàm dựng vẫn quy định cách đối tượng được tạo ra. Đoạn code sau không cung cấp hàm dựng mặc định mà chỉ khai báo hàm dựng có danh sách khởi tạo.

Khi này, danh sách khởi tạo có quyền ưu tiên hơn khởi tạo mặc định:

#include <iostream>;

using namespace std;

class Square

{

private:

double m\_length = 1.0;

double m\_width = 1.0;

public:

Square(double l, double w): m\_length(l), m\_width(w) {

/\*m\_length và m\_width ưu tiên khởi tạo bởi hàm dựng  
 \* bỏ qua giá trị khởi tạo mặc định

\*/

}

void print() {

cout << "length: " << m\_length << ", width: " << m\_width << '\n';

}

};

int main(){

Square x(2.0, 3.0);

x.print();

return 0;

}

Kết quả:

length: 2.0, width: 3.0

Kết luận: bạn nên dùng khởi tạo mặc định hơn là dùng hàm dựng để gán giá trị ban đầu cho thành viên dữ liệu non-static.

1. Tối ưu hàm dựng

Một lớp có thể có nhiều hàm dựng, nhưng các hàm dựng có các chức năng chồng chéo lên nhau là không tốt. Hãy xem class **Square**

class Square{

private:

double m\_length = 1.0;

double m\_width = 1.0;

public:

Square(){

cout<<"constructor is called"<<endl;

}

Square(double l, double w){

cout<<"constructor is called"<<endl;

m\_length = l;

m\_width = w;

}

};

Lớp **Square** có hai hàm dựng: hàm dựng mặc định và hàm dựng có đối số. Cả hai đều có mã lệnh bị lặp **cout.**

Trong lập trình, việc lặp lại các dòng lệnh phải hạn chế hết mức. Vậy có giải pháp nào cho vấn đề trên?

**Dùng hàm riêng biệt**

Giải pháp tốt nhất là tạo một hàm thành viên riêng biệt, để mọi hàm dựng gọi tới:

class Square{

private:

double m\_length = 1.0;

double m\_width = 1.0;

public:

Square(){

init();

}

Square(double l, double w){

init();

m\_length = l;

m\_width = w;

}

void init(){

cout<<"constructor is called"<<endl;

}

};

Hàm **init()** lo phần xuất thông báo ra màn hình, các hàm dựng đều gọi tới nó.

Theo cách này , chúng ta tối ưu được mã lệnh, cấu trúc chương trình dễ đọc và bảo trì hơn.

**Lưu ý: với đối tượng là con trỏ, chỉ khi bạn cấp phát vùng nhớ động thì hàm dựng mới được gọi**

1. Hàm dựng tường minh (explicit)

Mặc định hàm dựng trong C++ được hiểu là implicit, tức trình biên dịch được phép chuyển kiểu ngầm định khi truyền tham số đối tượng vào hàm.

Xét ví dụ:

class Foo {

public:

// Ham dung mot tham so

// Co the duoc goi cho viec chuyen kieu khong tuong minh

Foo(int foo) : m\_foo(foo) {}

int GetFoo() {

return m\_foo;

}

private:

int m\_foo;

};

//ham co tham so kieu doi tuong Foo

void DoBar(Foo foo) {

int i = foo.GetFoo();

}

//ham main

int main() {

DoBar(42);

}

Tham số truyền trong hàm main() kiểu integer, không phải kiểu đối tượng Foo. Tuy nhiên, kiểm tra lớp Foo có tồn tại hàm dựng nhận tham số kiểu integer, trình biên dịch dùng hàm dựng này để ép kiểu tham số.

Trình biên dịch được phép làm vậy một lần cho mỗi tham số.

Khai báo từ khòa explicit trước hàm dựng sẽ ngăn trình biên dịch dùng hàm dựng để ép kiểu ngầm định như trên

explicit Foo(int foo) : m\_foo(foo) {}

Trình biên dịch sẽ báo lỗi khi gọi hàm DoBar (42). Bạn sẽ phải chuyển kiểu tường minh như sau:

DoBar(Foo(42))

Khai báo tường minh (explicit) giúp bạn tránh các lỗi tình cờ liên quan tới hàm dựng, trừ khi bạn có chủ ý trong việc thiết kế.

1. Hàm dựng ủy thác C++ 11

C++ 11 cho phép hàm dựng gọi hàm dựng, hàm dựng ủy thác là hàm dựng được gọi tới .

Cú pháp là đặt hàm dựng ủy thác cùng đối số trong danh sách khởi tạo hàm dựng.

<tên\_hàm dựng>(danh sách tham số): <tên\_hàm\_dựng\_ủy\_thác>(danh sách tham số) {định nghĩa hàm}

Ví dụ lớp **Employee**, gồm 4 hàm dựng để truy xuất 2 dữ liệu thành viên **id** và **name**.

class Employee

{

private:

int id;

string name;

public:

Employee(int id, string name):id(id), name(name) {}

// Cả 3 hàm dựng bên dưới dùng hàm dựng ủy thác để tối ưu code

Employee() : Employee(0, "") { }

Employee(int id) : Employee(id, "") { }

Employee(string name) : Employee(0, name) { }

};

1. Hàm hủy:

Hàm hủy là hàm thành viên đặc biệt của lớp, được tự động thực khi khi đối tượng của lớp ra ngoài phạm vi hay khi con trỏ tới đối tượng bị xóa.

Hàm hủy có tên trùng với tên lớp, nhưng có tiền tố (~) phía trước, nó không trả về và cũng không nhận bất kì tham số nào cả. Các thao tác giải phóng tài nguyên trước khi ra khỏi chương trình như là đóng files, giải phóng bộ nhớ… phải được khai báo trong hàm hủy.

Xem ví dụ sau

#include <iostream>;

using namespace std;

class Line {

public:

void setLength( double len );

double getLength( void );

Line(); // Khai báo hàm dựng

~Line(); // Khai báo hàm hủy

private:

double \*length;

};

// Member functions definitions including constructor

Line::Line(void) {

cout << "Object is being created" << endl;

length = new double(0);

}

Line::~Line(void) {

cout << "Object is being deleted" << endl;

delete(length);

}

void Line::setLength( double len ) {

\*length = len;

}

double Line::getLength( void ) {

return \*length;

}

// Main function for the program

int main( ) {

Line line;

// set line length

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

return 0;

}

Kết quả

Object is being created

Length of line : 6

Object is being deleted

Đối tượng **Line** gồm con trỏ dữ liệu thành viên **length,** hàm dựng có nhiệm vụ cấp pháp vùng nhớ động cho **length.**

Khi thoát chương trình, hàm hủy được gọi và có nhiệm vụ giải phóng vùng nhớ đã cấp phát.

### Hàm Inline

1. Giới thiệu

Hàm **inline** trong C++ được dùng phổ biến với lớp. Nếu một hàm là **inline,** trình biên dịch đặt bản sao đoạn code hàm đó tại mỗi vi trí mà hàm đó được gọi tại thời điểm biên dịch.

Bất kì thay đổi nào cho hàm inline sẽ yêu cầu mọi chương trình sử dụng đang hàm bị biên dịch lại vì trình biên dịch sẽ cần thay thế toàn bộ mã lệnh mới hơn, thay vì tiếp tục với hàm cũ.

Đặt từ khóa **inline** trước tên hàm để khai báo hàm inline và phải định nghĩa hàm trước bất kì lệnh gọi hàm nào. Trình biên dịch có quyền hủy đặc tính **inline** trong trường hợp hàm được định nghĩa hơn một dòng lệnh.

Cú pháp:

**Inline** <tên\_hàm>

Định nghĩa hàm thành viên trong lớp là một định nghĩa hàm inline, cho dù không có chỉ định **inline.**

Tiếp theo là ví dụ về cách dùng hàm inline để trả về giá trị lớn hơn giữa hai số:

inline int Max(int x, int y){

return (x > y)? x : y;

}

// Main function for the program

int main( ){

cout << "Max (20,10): " << Max(20,10) << endl;

return 0;

}

Kết quả:

Max (20,10): 20

1. Dùng hàm inline để làm gì

Nói tóm lại, hàm inline chỉ có mục đích duy nhất là:

* Giúp chương trình thực thi nhanh hơn: do không phải gọi hàm vào stack rồi mới thực thi, mà chèn toàn bộ thân hàm vào điểm gọi hàm.
* Tuy nhiên bạn phải thiết kế hàm inline thật nhỏ gọn
* Khi khai báo thành viên lớp, nếu thân hàm chỉ có vài dòng thì khai báo inline ngay trong thân lớp.

### Con trỏ this

Mọi đối tượng trong C++ đều có nhu cầu truy xuất tới chính địa chỉ của nó qua một con trỏ đặc biệt, gọi là con trỏ **this**.

Con trỏ **this** là một tham số ẩn cho mọi hàm thành viên. Do đó, bên trong hàm thành viên, con trỏ này như là cách gọi đối tượng

Hàm friend không con trỏ **this,** vì nó không phải là thành viên của lớp. Chỉ hàm thành viên mới có mà thôi.

Chúng ta hãy thử ví dụ sau:

#include <iostream>

using namespace std;

class Box{

public:

// Constructor definition

Box(double l=2.0, double b=2.0, double h=2.0) {

cout <<"Constructor called." << endl;

length = l;

breadth = b;

height = h;

}

double Volume() {

return length \* breadth \* height;

}

int compare(Box box) {

return this->Volume() > box.Volume();

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main(void){

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

If (Box1.compare(Box2)) cout << "Box2 is smaller than Box1" <<endl;

else

cout << "Box2 is equal to or larger than Box1" <<endl;

return 0;

}

Kết quả

Constructor called.

Constructor called.

Box2 is equal to or larger than Box1

### Con trỏ tới lớp

Con trỏ tới lớp trong C++ tương tự như con trỏ tới cấu trúc. Để truy xuất tới thành viên từ con trỏ lớp, bạn dùng toán tử truy xuất thành viên “**->**” , giống như cấu trúc vậy.

Tương tự như mọi con trỏ, con trỏ lớp cũng phải khởi tạo trước khi dùng.

Hãy thử xem ví dụ sau để hiểu khái niệm con trỏ lớp:

#include <iostream>

using namespace std;

class Box{

public:

// Constructor definition

Box(double l=2.0, double b=2.0, double h=2.0) {

cout <<"Constructor called." << endl;

length = l;

breadth = b;

height = h;

}

double Volume() {

return length \* breadth \* height;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main(void){

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

Box \*ptrBox; // Khai báo con trỏ tới lớp.

// Save the address of first object

ptrBox = &Box1;

// Truy xuất tới thành viên bằng toán tử truy xuất

cout << "Volume of Box1: " << ptrBox->Volume() << endl;

// Save the address of first object

ptrBox = &Box2;

// Truy xuất tới thành viên bằng toán tử truy xuất

cout << "Volume of Box2: " << ptrBox->Volume() << endl;

return 0;

}

Kết quả

Constructor called.

Constructor called.

Volume of Box1: 5.94

Volume of Box2: 102

### Thuộc tính Static

1. Thành viên Static của lớp

Chúng ta có thể định nghĩa thành viên lớp có đặc tính **static**, nó có nghĩa là bất kể có bao nhiêu đối tượng của lớp được tạo thì vẫn chỉ tồn tại duy nhất một thành viên **static.**

Thành viên **static** được chia sẻ bởi tất cả đối tượng của lớp. Nếu không được khởi tạo bằng tay, tất cả dữ liệu static được tự động thiết lập zero khi đối tượng đầu tiên được tạo.

Chúng ta có thể khởi tạo cho dữ liệu thành viên **static** trong phần định nghĩa, bất kể đặc tính truy xuất.

Xét lớp Line

#include <iostream>

using namespace std;

class Line{

public:

static int objectCount; //thành viên objectCount để đếm số đối tượng của lớp được tạo ra

// Constructor definition

Line(double l=2.0) {

cout <<"Constructor called." << endl;

length = l;

objectCount++;// Tăng mỗi lần có đối tượng khởi tạo

}

};

Vì dữ liệu thành viên static không sở hữu bởi đối tượng nào, nó được khởi tạo ngay khi chương trình bắt đầu, nên bạn phải khởi tạo bên ngoài lớp, trong phạm vi global.

Dùng toán tử phạm vi “::” để xác định lớp nó thuộc về:

// Khởi tạo thành viên static của lớp Box

int Line::objectCount = 0;

Nếu bạn chỉ khai báo mà không khởi tạo giá trị cho thành viên static, trình biên dịch sẽ dùng giá trị mặc định.

int main(void){

Line Line1(1.5); // Declare box1

Line Line2(2.0); // Declare box2

// Print total number of objects.

cout << "Total objects: " << Line::objectCount << endl;

return 0;

}

Kết quả

Constructor called.

Constructor called.

Total objects: 2

Vài điểm chú ý về thành viên dữ liệu **static**:

* Do không phải đối tượng để kiểm soát, nên dù được khai báo là **private** hay **protected** trong lớp, bạn vẫn định nghĩa ngoài phạm vi lớp được. Còn để truy xuất, gán giá trị, bắt buộc phải thông qua hàm **static**
* Nếu lớp được định nghĩa trong header file, thành viên static thường định nghĩa trong file cpp, đặt ngay sau bên dưới.  
  Chúng ta không bao giờ đặt định nghĩa thành viên static trong file header (nếu file header include nhiều lần, sẽ gây lỗi cho trình biên dịch)
* Trường hợp đặc biệt khi dữ liệu thành viên **static** là kiểu **const** **integer** hay **const** **enum**, bạn có thể khởi tạo trực tiếp ngay khi khai báo mà không cần khởi tạo bên ngoài hàm

class Box{

public:

static const int objectCount = 4; // biến hằng static có thể khởi tạo trực tiếp

Hãy hình dung chúng ta có một mảng các đối tượng, thông qua biến static trong lớp giúp chúng ta biết có bao nhiêu đối tượng (phần tử) trong mảng.

Dữ liệu thành viên static còn hữu dụng khi lớp cần sử dụng một bảng tra cứu nội bộ (vd: mảng thường dùng để chứa các giá trị tính toán). Qua việc tìm kiếm bảng static, chỉ có một bản sao tồn tại cho mọi đối tượng, giúp tiết kiệm bộ nhớ rất nhiều.

1. Hàm thành viên Static:

Nếu thành viên **static** có đặc tính **public**, thì có thể truy xuất qua toán tử phạm vi từ lớp hay toán tử thành viên từ đối tượng.

Nhưng nếu là hàm thành viên **private** **static** thì chúng ta phải dùng hàm thành viên **public** **static** để truy xuất:

#include <iostream>

using namespace std;

class Line{

private:

static int objectCount;

// Constructor definition

Box(double l=2.0) {

cout <<"Constructor called." << endl;

length = l;

objectCount++;// tăng dần khi đối tượng mới được tạo

}

static int getCount() {return objectCount;}

};

int Line::objectCount = 0; // Khởi tạo thành viên static

Chúng ta không thể truy xuất **objectCount** từ hàm **main**() vì nó là **private**. Cách xử lí thông thường là khai báo một hàm trung gian trong lớp để truy xuất tới, nhưng phải tạo ra đối tượng nào đó.

Thay vì vậy, chúng ta tạo ra một hàm **static getCount()**. Cũng giống như biến thành viên **static**, hàm static hoạt động độc lập với mọi đối tượng của lớp, nghĩa là có thể gọi dù không có đối tượng nào của lớp tồn tại.

Cú pháp là dùng tên lớp với toàn tử phạm vi “**::**”, tiếp theo là tên hàm.

(khuyến cáo không nên truy xuất với đối tượng bằng toán tử thành viên)

Thủ tục định nghĩa ngoài lớp:

int Line::getCount() {return objectCount;}

Sau đây là hàm main:

int main(void){

// In tong so doi tuong lop Line truoc khi tao

cout << "Inital Stage Count: " << Line::getCount() << endl;

Line Line1(1.5); // Declare line 1

Line Line2(2.0); // Declare line 2

// In tong so doi tuong lop Line da tao

cout << "Final Stage Count: " << Line::getCount() << endl;

return 0;

}

Kết quả

Inital Stage Count: 0

Constructor called.

Constructor called.

Final Stage Count: 2

**Hàm thành viên static không có con trỏ \*this và không thể truy xuất thành viên non-static**

* Đầu tiên, vì hàm thành viên static không thuộc về đối tượng nào nên chúng không có con trỏ **\*this.** (Con trỏ này luôn chỉ về đối tượng mà hàm thành viên thuộc về)
* Thứ hai, hàm thành viên static chỉ có thể truy xuất dữ liệu thành viên static và dữ liệu bên ngoài lớp. Chúng không thể truy xuất thành viên non-static. Nguyên nhân vì biến thành viên non-static phải thuộc về đối tượng cụ thể nào đó.

1. Khuyến cáo về lớp thuần Static

Khi bạn tạo một lớp toàn thành viên static, lưu ý một số nguy cơ sau:

Đầu tiên, vì tất cả thành viên static được khởi tạo một lần duy nhất, nên bạn không tạo được bản sao nào (đối tượng) khác của lớp.

Thứ hai, bạn biết rằng biến toàn cục là nguy hiểm vì bất kì mã lệnh nào cũng có thể thay đổi giá trị của nó và gây ra lỗi cho phần nào đó của chương trình.

Điều này cũng tương tự cho lớp static. Vì khai báo lớp thường nằm ở phạm vi toàn cục, lớp thuần static cũng giống như khai báo hàm và biến toàn cục ở vùng namespace có thể truy xuất được

1. C++ không hỗ trợ khởi tạo Static member qua hàm dựng

Nếu bạn có thể khởi tạo thành viên dữ liệu qua hàm dựng, vậy có khả năng nào chúng ta khởi tạo giá trị cho thành viên **static** qua hàm dựng **static** không?

Một vài ngôn ngữ hiện đại hỗ trợ điều đó, nhưng không may là C++ không hỗ trợ.

Nếu bạn có thể khởi tạo thành viên dữ liệu **static** tại thời điểm định nghĩa thì không cần hàm dựng **static**.

Trường hợp ngoại lệ, là thành viên **static** là đối tượng và có hàm dựng khởi tạo.

#include <vector>

class MyClass{

private:

static vector<char> s\_mychars;

public:

class \_init // we're defining a nested class named \_init {

public:

\_init() // the \_init constructor will initialize our static variable {

s\_mychars.push\_back('a');

s\_mychars.push\_back('e');

s\_mychars.push\_back('i');

s\_mychars.push\_back('o');

s\_mychars.push\_back('u');

}

} ;

private:

static \_init s\_initializer; // khai báo thành viên static

};

vector<char> MyClass::s\_mychars; // khởi tạo thành viên static

MyClass::\_init MyClass::s\_initializer; // Bộ khởi tạo sẽ khởi tạo giá trị cho vector s\_mychars ngay lập tức.

1. Tổng kết

Hàm thành viên static chỉ làm việc với thành viên static trong lớp. Gọi trực tiếp qua tên lớp mà không cần tạo đối tượng nào.

Lớp toàn thành viên static được coi như biến và hàm toàn cục trong namespace toàn cục, tránh sử dụng trừ khi có lý do.

### Thuộc tính hằng

1. Đối tượng hằng

Khi đối tượng được khai báo hằng hay truyền tham số kiểu hằng, bất kì nỗ lực nào làm thay đổi giá trị thành viên dữ liệu là không được phép.

Việc thay đổi này bao gồm truy xuất trực tiếp (nếu là thành viên public), hay gián tiếp thông qua hàm thành viên.

Xem tiếp ví dụ lớp **Line:**

#include <iostream>

using namespace std;

class Line{

public:

int length;

Line(): length(0) { }

void setLength(int l) { length = l; }

int getLength() { return length; }

};

int main(){

const Line line; // calls default constructor

line.length = 5; // compiler error: violates const

line.setLenght(5); // compiler error: violates const

return 0;

}

Xét đoạn mã sau trong hàm funtion():

void funtion(const Line& line){

cout<<line.getLength();

}

Mặc dù **getLength**() không làm thay đổi dữ liệu thành viên khi truy xuất nhưng trình biên dịch vẫn báo lỗi.

1. Hàm thành viên hằng

Nguyên tắc là chỉ có hàm thành viên hằng mới truy xuất được các thành viên của đối tượng hằng.

Hàm hằng có hai đặc tính:

* Được phép truy xuất được các thành viên trong đối tượng hằng, nhưng không được phép làm thay đổi dữ liệu thành viên (const hay non-const) của lớp.
* Truy xuất được các phương thức bên ngoài lớp

Cú pháp khai báo:

<kiểu\_dl> <tên hàm> () const {định nghĩa hàm}

Ta định nghĩa lại lớp Line như sau:

class Line{

public:

int length;

Line(): length(0) { }

void setLength(int l) const { length = l; }//error, hàm hằng không dc thay đổi biến thành viên

int getLength() const { return length; }

};

Dòng khai báo hàm hằng **setLength()** báo lỗi ngay lập tức, vì hàm hằng không được phép làm thay đổi dữ liệu thành viên.

Cách khai báo hàm hằng **getLength()** bên ngoài lớp:

int Line::getLength() const{

return length;

}

**Lưu ý là hàm dựng không nên là hằng, vì đối tượng hằng vẫn cần có khả năng khởi tạo giá trị cho thành viên dữ liệu. Một đối tượng không thể khởi tạo giá trị thành viên là không hợp lý .**

**Luật: luôn đánh dấu hàm thành viên không làm thay đổi trạng thái của đối tượng là const**

1. Dữ liệu thành viên hằng

Biến hằng bên ngoài lớp phải được khởi tạo ngay khi khai báo. Nhưng trong phạm vi lớp, đặc tính hằng hoàn toàn khác.

Bạn không thể khởi tạo thành viên dữ liệu hằng trực tiếp, mà phải thông qua danh sách khởi tạo hàm dựng.

Ta lấy lại ví dụ trong phần [**danh sách khởi tạo hàm dựng**](#_Khởi_tạo_cho):

class Line {

private:

const double length;

// Khai báo hàm dựng

Public:

Line():length(0){}

}

};

Nếu bạn muốn khởi tạo hằng đơn giản như sau trong lớp thì C++ không cho phép, ngoại trừ C++11 trở lên:

class foo {

public:

const int foo\_size = 100; // C++11

1. Tham chiếu hằng

Ngoài khai báo **const** để tạo ra đối tượng hằng, ta hay gặp trường hợp truyền đối tượng vào hàm theo kiểu tham chiếu hằng

Bạn có thể chỉ ra điểm sai trong đoạn code sau ?

#include <iostream>

using namespace std;

class Date{

private:

int m\_year;

int m\_month;

int m\_day;

public:

Date(int year, int month, int day) {

setDate(year, month, day);

}

void setDate(int year, int month, int day) {

m\_year = year;

m\_month = month;

m\_day = day;

}

int getYear() { return m\_year; }

int getMonth() { return m\_month; }

int getDay() { return m\_day; }

};

// note: Chúng ta truyền tham số date kiểu tham chiếu hằng để tránh tạo bản sao của date

void printDate(const Date &date){

cout << date.getYear() << "/" << date.getMonth() << "/" << date.getDay() << endl;

}

int main(){

Date date(2016, 10, 16);

printDate(date);

return 0;

}

Đó là hàm **printDate,** đối số **date** là đối tượng hằng. Các hàm getYear(), getMonth(), and getDay() không phải là hàm hằng nên không thể truy xuất tới thành viên dữ liệu.

Để sửa lại, đơn giản là đánh dấu chúng như là hàm thành viên hằng:

int getYear() const { return m\_year; }

int getMonth() const{ return m\_month; }

int getDay() const { return m\_day; }

1. Hàm const và non-const

Bạn xem ví dụ sau:

class Line{

private:

Line length;

public:

const string& getLength() const { return length; } //#1

string& getLength() { return length; } //#2

};

Hàm const tại #1 đảm bảo an toàn cho dữ liệu thành viên khi nó trả về kiểu tham chiếu hằng. Phiên bản #2 kém an toàn, nó cho phép nơi gọi hàm có thể thay đổi giá trị biến thành viên.

1. Kết luận

Việc truyền đối tượng hằng tham chiếu vào đối số hàm là rất thường xuyên, định nghĩa lớp luôn phải khai báo hằng cho những hàm cần thiết phải bảo vệ dữ liệu thành viên.

### Accessor và Mutator

1. Hàm thành viên Accessor

Accessor cho phép truy xuất tới thành viên dữ liệu non-public. Có 3 điều kiện Accesor phải thỏa mãn:

* Accessor trả về đúng kiểu dữ liệu thành viên.
* Không có tham số.
* Nguyên tắc đặt tên thường có tiền tố "Get".   
  Ví dụ:

class Toto{

private :

int age;

public :

int GetAge();//Accessor

};

int Toto::GetAge(){ return age;}

1. Hàm thành viên Mutator

Mutator cho phép can thiệp, thay đổi nội dung của dữ liệu thành viên non-public. Nó thỏa mãn 3 yêu cầu:

* Có tham số là giá trị gán cho dữ liệu thành viên và phải cùng kiểu với dữ liệu thành viên.
* Không cần giá trị trả về
* Nguyên tắc đặt tên thường có tiền tố “Set”.

Ví dụ

class Toto{

private :

int \_age;

public :

void SetAge(int);//Mutator

};

void Toto::SetAge(int age){

\_age = age;

}

### Thuộc tính Friend

Che dấu dữ liệu qua thuộc tính **private/protected** là cần thiết, tuy nhiên trường hợp bạn có hàm/đối tượng nằm bên ngoài lớp cần làm việc với dữ liệu **non-public** thì sao?

* Nếu khai báo hàm **public** xuất dữ liệu riêng nằm trong lớp chứa dữ liệu. Nhưng vô hình làm lộ ra các dữ liệu bên trong.
* Giải pháp là khai báo tính **Friend,** cho phép truy xuất tất cả dữ liệu và hàm **non-public** của lớp dữ liệu. Trong khi ngăn được sự dòm ngó từ bên ngoài.

1. Hàm Friend

Hàm friend có thủ tục khai báo nằm trong lớp, nhưng phần định nghĩa nằm bên ngoài phạm vi của lớp đó. Dù vậy nó không được xem là hàm thành viên của lớp, nhưng lại có quyền truy xuất tất cả thành viên private và protected.

Đặc tính truy xuất không ảnh hưởng tới hàm **friend.**

Cú pháp:

friend <tên\_hàm> (danh\_sach\_tham\_so);

Ví dụ

#include <iostream>

using namespace std;

class Box{

double width;

public:

friend void printWidth( Box box );

void setWidth( double wid );

};

// Member function definition

void Box::setWidth( double wid ){

width = wid;

}

Định nghĩa phải ngằm ngoài lớp, và không có toán tử phạm vi vì không được xem là thành viên.

// Note: printWidth() không phải hàm thành viên của bất kì lớp nào.

void printWidth( Box box ){

/\* Vì printWidth() là friend của Box, nên nó truy xuất được bất kì thành viên nào của lớp B**ox** \*/

cout << "Width of box : " << box.width <<endl;

}

Hàm main()

int main( ){

Box box;

// set box width without member function

box.setWidth(10.0);

// Hàm friend được gọi trực tiếp như sau.

printWidth( box );

return 0;

}

Kết quả

Width of box : 10

Tham số bắt buộc của hàm **friend** là đối tượng của lớp Box, vì **printWidth**() không phải là hàm thành viên nên không thể gọi từ một đối tượng cụ thể của lớp Box.

Do đó, hàm **friend** cũng không có con trỏ \***this.**

Xem ví dụ khác:

class Value{

private:

int m\_value;

public:

Value(int value) { m\_value = value; }

friend bool isEqual(const Value &value1, const Value &value2);

};

bool isEqual(const Value &value1, const Value &value2){

return (value1.m\_value == value2.m\_value);}

Hàm **friend** **isEqual**() để so sánh **m\_value** từ hai đối tượng Value khác nhau. Dù **m\_value** có đặc tính const nhưng hàm friend vẫn truy xuất được.

1. Multi-Friend

Một hàm có thể là bạn của nhiều lớp khác nhau. Ví dụ:

#include <iostream>

using namespace std;

class Humidity;//chuyen tiep khai bao – forward declaration

class Temperature{//nhiet do

private:

int m\_temp;

public:

Temperature(int temp=0) { m\_temp = temp; }

void setTemperature(int temp) { m\_temp = temp; }

friend void printWeather(const Temperature &temperature, const Humidity &humidity);

};

class Humidity{// do am

private:

int m\_humidity;

public:

Humidity(int humidity=0) { m\_humidity = humidity; }

void setHumidity(int humidity) { m\_humidity = humidity; }

friend void printWeather(const Temperature &temperature, const Humidity &humidity);

};

void printWeather(const Temperature &temperature, const Humidity &humidity){

cout << "The temperature is " << temperature.m\_temp <<

" and the humidity is " << humidity.m\_humidity << '\n';

}

int main(){

Humidity hum(10);

Temperature temp(12);

printWeather(temp, hum);

return 0;

}

Chú ý khai báo prototype class **Humidity** ở phía trên thì hàm **friend printWeather()** trong lớp **Storage** mới hiểu

1. Lớp Friend

Thuộc tính **Friend** có thể là lớp. Trong trường hợp là lớp thì toàn bộ lớp và thành viên của nó đều có tính **friend.**

* Các phương thức **public** của lớp **friend** được quyền truy xuất tất cả của lớp hiện tại (không lệ thuộc đặc tính truy xuất)
* Các phương thức **private** và **protected** của lớp **friend** chỉ được quyền truy xuất thành viên **public** của lớp hiện tại

Cú pháp**:**

friend class <ten\_class>;

Xem chương trình:

class Storage{

private:

int m\_nValue;

double m\_dValue;

public:

Storage(int nValue, double dValue) {

m\_nValue = nValue;

m\_dValue = dValue;

}

// khai báo Display là lớp bạn của Storage

friend class Display;

};

class Display{

public:

Display() {}

void displayItem(Storage &storage) {

cout << storage.m\_nValue << " " << storage.m\_dValue << '\n';

}

};

int main(){

Storage storage(5, 6.7);

Display \*display = new Display();

display->displayItem(storage);

return 0;

}

Lưu ý: Thứ nhất, do không phải thành viên nên hàm và lớp **friend** không truy xuất được con trỏ **\*this** của bất kì đối tượng nào.

Thứ hai, tính **friend** không có tính bắc cầu, lớp A là bạn của lớp B, B là bạn của lớp C. Thì không có nghĩa A là bạn của C.

Tính **friend** vi phạm khá nhiều nguyên tắc bao đóng của lớp. Cần hạn chế tối thiểu sử dụng

Chỉ nên khai báo **friend** cho hàm thành viên của lớp khác, chúng ta sẽ đề cập hàm friend thành viên ngay bây giờ.

1. Hàm thành viên Friend

Thay vì khai báo lớp friend, bạn có thể khai báo hàm thành viên lớp là friend của lớp khác. Ví dụ sau sẽ xảy ra lỗi nếu khai báo không logic:

class Display;// Chuyển tiếp khai báo lên trên để Storage hiểu lớp Display

class Storage{

private:

int m\_nValue;

double m\_dValue;

public:

Storage(int nValue, double dValue);

// Khai báo thành viên displayItem của Display là friend

friend void Display::displayItem(Storage& storage); //Lỗi: Storage không thấy khai báo đẩy đủ của Display.

};

class Display{

public:

Display() ;

void displayItem(Storage &storage);

};

Vì khai báo Display nằm phía sau Storage, nên thành viên của Storage sẽ không hiểu Display. Giải pháp là chuyển tiếp khai báo lớp ở trên cùng.

#include <iostream>

using namespace std;

class Storage; // Chuyển tiếp khai báo

Sau đó khai báo lớp Display nhưng chưa định nghĩa hàm displayItem().

class Display{

public:

Display() {}

void displayItem(const Storage &storage); // Nhờ chuyển tiếp nên hàm displayItem hiểu lớp storage

};

Tiếp theo là khai báo lớp Storage .

class Storage{

private:

int m\_nValue;

double m\_dValue;

public:

Storage(int nValue, double dValue) {

m\_nValue = nValue;

m\_dValue = dValue;

}

// Nhờ khai báo Display trước nên không báo lỗi

friend void Display::displayItem(const Storage& storage);

};

Rồi mới định nghĩa hàm displayItem() sau cùng.

// Hàm friend thấy được khai báo storage nên truy xuất được.

void Display::displayItem(const Storage &storage){

cout << storage.m\_nValue << " " << storage.m\_dValue << '\n';

}

Hàm main()

int main(){

Storage storage(5, 6.7);

Display \*display = new Display();

display->displayItem(storage);

return 0;

}

Giải pháp tốt nhất là đặt khai báo từng lớp vào từng header file riêng biệt, định nghĩa nằm trong file .cpp, nên không cần sắp xếp như trên.

1. Tổng kết

* **Friend** và **const** rất hữu dụng dể làm việc với dữ liệu lớp khác mà không làm mất tính bao đóng.
* **Friend** rất phổ biến khi quá tải toán tử cho lớp

### Tính thừa kế

Một trong đặc tính quan trọng của OOP là tính thừa kế. Thừa kế cho phép chúng ta định nghĩa một lớp dưới dạng một lớp khác, giúp dễ dàng tạo và duy trì trình ứng dụng.

* Cho phép khả năng tái sử dụng mã lệnh và triển khai nhanh chóng.
* Khi tạo lớp, thay vì viết lại toàn bộ thành viên lớp, lập trình viên có thể chỉ định một lớp mới thừa kế thành viên của một lớp có sẵn. Lớp có sẵn gọi là lớp cơ sở (hay lớp cha) và lớp mới gọi là lớp dẫn xuất (hay lớp con).
* Ý tưởng của tính thừa kế là để triển khai đặc tính quan hệ. Ví dụ: động vật có vú **là** động vật, chó **là** động vật có vú

1. Lớp cơ sở và dẫn xuất:

Một lớp có thể được dẫn xuất từ nhiều hơn một lớp, nghĩa là nó thừa kế dữ liệu và hàm từ nhiều lớp cơ sở.

Để định nghĩa lớp dẫn xuất, chúng ta dùng danh sách lớp dẫn xuất để xác định các lớp cơ sở. Một danh sách lớp dẫn xuất đánh tên một hay nhiều lớp cơ sở và có cú pháp

class <lớp dẫn xuất>: <đặc tính truy xuất> <lớp cơ sở>

Đặc tính truy xuất gồm public, protected hay private, nếu không khai báo thì mặc định là private.

Xem ví dụ về lớp cơ sở **Shape** và dẫn xuất **Rectangle** của nó

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

void setWidth(int w) { width = w; }

void setHeight(int h) { height = h; }

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape{

public:

int getArea() { return (width \* height); }

};

int main(void){

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

Kết quả

Total area: 35

1. Quản lý truy xuất và tính thừa kế:

Một lớp dẫn xuất có thể truy xuất mọi thành viên non-private của lớp cơ sở. Do đó, thành viên lớp cơ sở không nên truy xuất từ lớp dẫn xuất thì khai báo đặc tính private cho chúng.

Chúng ta tổng kết các đặc tính truy xuất khác nhau dựa theo quyền truy cập như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| **Truy xuất** | **public** | **protected** | **private** |
| Bên trong lớp | yes | yes | yes |
| Lớp dẫn xuất | yes | yes | no |
| Bên ngoài lớp | yes | no | no |

Lớp dẫn xuất thừa kết mọi phương thức lớp cơ sở ngoại trừ:

* Hàm dựng, hàm hủy và hàm dựng sao chép của lớp cở sở
* Toán tử quá tải của lớp cơ sở
* Hàm friend của lớp cơ sở

1. Các đặc tính thừa kế:

Khi tạo lớp dẫn xuất, lớp cơ sở có thể được thừa kế qua **public, protected** hay **private**. Các kiểu thừa kế được qui định bởi đặc tính truy xuất như đã nói ở trên.

Chúng ta hầu như không dùng thừa kế **protected** hay **private,** còn **public** thì phổ biến hơn. Nhưng các kiểu thừa kế phải tuân theo luật sau:

* **Thừa kế public:**  các thành viên **public** và **protected** lớp cơ sở giữ nguyên đặc tính truy xuất khi thừa kế qua lớp dẫn xuất. Thành viên private lớp cơ sở sẽ không thể truy xuất trực tiếp từ lớp dẫn xuất, nhưng có thể truy xuất thông qua thành viên public, protected của lớp cơ sở.
* **Thừa kế Protected:** thành viên **public** và **protected** lớp cơ sở trở thành thành viên **protected** của lớp dẫn xuất.
* **Thừa kế Private:** thành viên **public** và **protected** của lớp cơ sở trở thành thành viên **private** của lớp dẫn xuất. Là đặc tính mặc định khi bạn không khai báo khi thừa kế.

1. Thứ tự hàm dựng trong lớp dẫn xuất

Đầu tiên bạn xem ví dụ sau:

class Base{

public:

int m\_nValue;

Base(int nValue=0): m\_nValue(nValue){}

};

class Derived: public Base{

public:

double m\_dValue;

Derived(double dValue=0.0): m\_dValue(dValue){}

};

Khi bạn khai báo lớp Base thông thường:

int main(){

Base cBase;

return 0;

}

Base không phải lớp dẫn xuất, nó không thừa kế từ ai cả. C++ cấp phát vùng nhớ cho biến cBase, sau đó gọi hàm dựng của Base.

Trường hợp là lớp dẫn xuất:

int main(){

Derived cDerived;

return 0;

}

Lớp dẫn xuất gồm một phần từ lớp cơ sở và một phần của chính nó. Khi C++ xây dựng đối tượng dẫn xuất, nó bắt đầu với phần từ lớp cơ sở.

Khi đã xong, nó duyệt qua cây phân cấp và xây dựng lớp dẫn xuất.

Trong ví dụ lớp dẫn xuất, hàm dựng lớp cơ sở được gọi trước. Sau khi xong lớp cơ sở, hàm dựng lớp dẫn xuất được gọi.

Đoạn code sau sẽ hình dung tiến trình gọi:

#include <iostream>

using namespace std;

class Base{

public:

int m\_nValue;

Base(int nValue=0): m\_nValue(nValue) {

cout << "Base" << endl;

}

};

class Derived: public Base{

public:

double m\_dValue;

Derived(double dValue=0.0): m\_dValue(dValue){

cout << "Derived" << endl;

}

};

int main(){

cout << "Instantiating Base" << endl;

Base cBase;

cout << "Instantiating Derived" << endl;

Derived cDerived;

return 0;

}

Kết quả:

Instantiating Base

Base

Instantiating Derived

Base

Derived

Nói theo logic, lớp con không thể tồn tại mà thiếu lớp cha. Có thể nói rằng:

* Lớp con thừa kế và dùng được mọi dữ liệu, hàm thành viên từ lớp cha.
* Lớp cha không nhận biết được sự hiện diện của lớp con.
* Khởi tạo lớp cha trước đảm bảo mọi thành viên được khởi tạo trước khi lớp con được tạo và có thể dùng.

1. Hàm dựng và khởi tạo của lớp dẫn xuất

Trong bài trước chúng ta đã hiểu cách hàm dựng được gọi theo thứ tự từ lớp cơ sở rồi tới lớp dẫn xuất. Trong phần này, sẽ tìm hiểu kỹ hơn vai trò hàm dựng trong khởi tạo lớp dẫn xuất.

class Base{

public:

int m\_nValue;

Base(int nValue=0): m\_nValue(nValue){}

};

class Derived: public Base{

public:

double m\_dValue;

Derived(double dValue=0.0): m\_dValue(dValue){}

};

Với lớp không phải dẫn xuất, hàm dựng chỉ quan tâm tới thành viên của nó thôi:

int main(){

Base cBase(5); // use Base(int) constructor

return 0;

}

Đây là các bước cBase được khởi tạo:

1. Vùng nhớ cấp cho Base
2. Hàm dựng lớp Base được gọi
3. Khởi tạo giá trị cho biến thành viên qua danh sách khởi tạo
4. Thân hàm dựng được thực thi
5. Quyền kiểm soát trả về nơi gọi.

Với lớp dẫn xuất, các bước phức tạp hơn:

int main(){

Derived cDerived(1.3); // use Derived(double) constructor

return 0;

}

1. Vùng nhớ cấp cho cDerived (đủ cả cho Base và Derived)
2. Hàm dựng lớp Derived được gọi
3. Nhưng hàm dựng đối tượng Base được thực thi trước khi hàm dựng lớp Drived được thực thi
4. Quyền kiểm soát trả về nơi gọi.

Sự khác biệt là trước khi hàm dựng lớp dẫn xuất kịp làm gì thì hàm dựng lớp cơ sở sẽ gọi trước. Hàm dựng lớp cơ sở sẽ thiết lập phần của lớp Base, quyền kiểm soát sau đó trả về hàm dựng lớp dẫn xuất để nó làm nốt công việc.

1. Khởi tạo thành viên lớp cơ sở

Một vấn đề xảy ra là lớp dẫn xuất không có cách nào để khởi tạo thành viên lớp cơ sở.

GIả sử bạn muốn khởi tạo cho thành viên cả lớp cơ sở lẫn dẫn xuất? Người mới lập trình thường gặp lỗi thế này:

class Derived: public Base

{

public:

double m\_dValue;

Derived(double dValue=0.0, int nValue=0) : m\_dValue(dValue), m\_nValue(nValue){}// lỗi

};

Chúng ta cần một tham số khác cho hàm dựng, nếu không C++ sẽ có cách nào để biết giá trị nào cần khởi tạo cho m\_nValue

Tuy nhiên C++ không cho phép khởi tạo thành viên kế thừa trong hàm dựng lớp dẫn xuất.

Tại sao lại như vậy?

Tưởng tượng **m\_nValue** là biến hằng , biến hằng bắt buộc phải dùng danh sách khởi tạo trong hàm dựng ngay khi đối tượng tạo ra. Tuy nhiên phải đợi hàm dựng lớp cơ sở hoàn thành, thì hàm dựng lớp dẫn xuất mới thực thi.

Như vậy mỗi lớp dẫn xuất đều có cơ hội để khởi tạo biến, và khả nằng thay đổi giá trị m\_nValue !!!

Bằng cách hạn chế khả năng khởi tạo biến của hàm dựng cho thành viên lớp nó thuộc về, C++ đảm bảo tất cả biến chỉ khởi tạo một lần.

Xem đoạn code sau sẽ cố gắng thay đổi giá trị của m\_nValue trong thân hàm.

class Derived: public Base{

public:

double m\_dValue;

Derived(double dValue=0.0, int nValue=0): m\_dValue(dValue) {

m\_nValue = nValue;

}

};

Giải pháp trên chỉ chạy được nếu m\_nValue không phải hằng hay tham chiếu (Cần danh sách khởi tạo hàm dựng).

Nên cách trên cũng không hiệu quả.

C++ cho ta khả năng chọn lựa hàm dựng lớp cơ sở nào sẽ được gọi từ danh sách khởi tạo lớp dẫn xuất. Đơn giản là thêm câu gọi hàm dựng lớp cơ sở trong danh sách khởi tạo lớp dẫn xuất.

class Derived: public Base

{

public:

double m\_dValue;

Derived(double dValue=0.0, int nValue=0): Base(nValue), // Goi hàm dựng Base(int)

m\_dValue(dValue){}

};

Hàm main:

int main(){

Derived cDerived(1.3, 5); // dùng hàm dựng Derived(double)

return 0;

}

Tuần tự các bước xảy ra:

1. Vùng nhớ cho cDerived cấp phát
2. Hàm dựng Derived(doube, int) được gọi
3. Trình biên dịch kiểm tra xem chúng ta có yêu cầu gọi hàm dựng lớp Base không? Có! Nó gọi Base(int)
4. Hàm dựng Base(int) được gọi và nValue = 5
5. Thân hàm dựng thực thi
6. Hàm dựng lớp dẫn xuất được trả về, và set m\_dValue = 1.3
7. Thân hàm dựng lớp dẫn xuất thực thi
8. Thân hàm dựng lớp dẫn xuất trả về nơi gọi.

**Kết luận: chỉ có hàm dựng của lớp cơ sở mới khởi tạo được thành viên của lớp cơ sở.**

**Bạn phải khai báo hàm dựng nào của lớp cơ sở nào sẽ được gọi trong danh sách khởi tạo hàm dựng lớp dẫn xuất**

### Chuyên đề về thành viên thừa kế

Ta có lớp Derived kế thừa lớp Base như sau:

#include <iostream>

using namespace std;

class Base{

protected:

int m\_nValue;

public:

Base(int nValue): m\_nValue(nValue){}

void Identify() { cout << "I am a Base" << endl; }

};

class Derived: public Base{

public:

Derived(int nValue):Base(nValue){}

int GetValue() { return m\_nValue; }

};

1. Định nghĩa hàm truy xuất thành viên thừa kế

Xem xét tình huống bạn mua một thư viện code từ nhà phát triển thứ 3, nhưng cần thêm một vài tính năng mở rộng. Bạn có thể chèn thêm vào phần code gốc, nhưng đây không phải giải pháp tốt nhất.

Nếu nhà phát triển tung ra bản cập nhật mới? Bạn chỉ có cách chuyển các tính năng đó qua bằng tay.

Đây là một trong vấn đề phổ biến mà lập trình viên gặp phải khi tạo header file chứa các định nghĩa lớp, nhưng chỉ coding các phần code của mình để dùng, mà không làm ảnh hưởng tới phần khác.

Giải pháp là tạo ra lớp dẫn xuất và thêm các tính năng mới vào lớp này.

Một thiếu sót từ lớp Base không cung cấp khả năng truy xuất m\_nValue từ phạm vi bên ngoài. Thông thường chúng ta sẽ viết một hàm truy xuất trong Base – nhưng mục tiêu bài này là đặt **getValue**() trong lớp Derived như trên.

Do đó chỉ có đối tượng lớp Derived mới truy xuất được hàm này, còn Base thì không.

1. Ghi đè hàm thừa kế

Phần này chúng ta sẽ làm việc với hàm thành viên **Identify**(), xét đoạn chương trình:

int main(){

Base cBase(5);

cBase.Identify();

Derived cDerived(7);

cDerived.Identify();

return 0;

}

Kết quả

I am a Base

I am a Base

Khi cDerived.Identify() được gọi, trình biên dịch không thấy hàm này trong định nghĩa Derived, nó bắt đầu kiếm trong lớp thừa kế Base. Sau đó, Base::Identify() được gọi vì Derived::Identify() không có.

Nếu bạn định nghĩa Derived::Identify(), trình biên dịch sẽ dùng nó.

Chúng ta sẽ định nghĩa lại để xuất ra “I am a Derived”.

class Derived: public Base{

public:

Derived(int nValue):Base(nValue){}

int GetValue() { return m\_nValue; }

// Here's our modified function

void Identify() { cout << "I am a Derived" << endl; }

};

int main(){

Base cBase(5);

cBase.Identify();

Derived cDerived(7);

cDerived.Identify();

return 0;

}

Kết quả

I am a Base

I am a Derived

**Lưu ý: Đặc tính truy xuất hàm tái định nghĩa trong lớp dẫn xuất không bị ảnh hưởng bởi lớp cơ sở, mà tùy thuộc vào khai báo trong lớp dẫn xuất.**

**Hàm thừa kế được gọi là overwrite khi nó đươc tái định nghĩa trong lớp dẫn xuất, có cùng kiểu trả về, kiểu tham số, tên hàm và số lượng tham số.**

1. Gọi hàm thừa kế từ lớp dẫn xuất

Thỉnh thoảng chúng ta không muốn thay thế hàm thừa kế, mà thêm tính năng cho nó.

Ở phần trên, Derived::Identify() hoàn toàn ghi đè Base::Identify()!. Để thực hiện, chúng ta gọi Base::Identify() ở trong Derived::Identify() và định nghĩa thêm.

class Derived: public Base{

public:

Derived(int nValue):Base(nValue){}

int GetValue() { return m\_nValue; }

void Identify() {

Base::Identify(); // gọi Base::Identify()

cout << "I am a Derived"; // định nghĩa thêm

}

};

int main()

{

Base cBase(5);

cBase.Identify();

Derived cDerived(7);

cDerived.Identify();

return 0;

}

Kết quả

I am a Base

I am a Base

I am a Derived

1. Đổi đặc tính thành viên thừa kế

C++ không có khả năng xóa một hàm khỏi lớp, nhưng cho phép ẩn qua việc đổi đặc tính truy xuất.

Khi bạn tái định nghĩa hàm, trình biên dịch chỉ hiểu đặc tính truy xuất trong lớp dẫn xuất. Do đó chúng ta có thể tái định nghĩa hàm public thành private trong lớp dẫn xuất, và nó không thể truy xuất được nữa.

Không chỉ vậy, C++ cho phép khai báo lại đặc tính truy xuất trong lớp dẫn xuất mà không cần định nghĩa lại.

**Lưu ý: Để khai báo bạn dùng toán tử phạm vi :: , nhưng không có toán tử ()**

class Base{

private:

int m\_nValue;

public:

Base(int nValue): m\_nValue(nValue){}

protected:

void PrintValue() { cout << m\_nValue; }

};

class Derived: public Base{

public:

Derived(int nValue): Base(nValue){}

// Base::PrintValue có tính protected, nên public không truy xuất được

// Nhưng ta thay đổi bằng cách đặt vào khu vực public của lớp dẫn xuất

Base::PrintValue;

};

int main(){

Derived cDerived(7);

// Truy xuat duoc tu public

cDerived.PrintValue(); // In ra 7

return 0;

}

Tương tự ta đổi đặc tính thành viên dữ liệu thừa kế từ public sang private như sau:

class Base{

public:

int m\_nValue;

};

class Derived: public Base{

private:

Base::m\_nValue;

public:

Derived(int nValue) {

m\_nValue = nValue;

}

};

int main(){

Derived cDerived(7);

// Trinh bien dich bao loi vi m\_nValue bị dinh nghia Private trong lop dan xuat

cout << cDerived.m\_nValue;

return 0;

}

Tuy nhiên cách thiết kế lớp như vậy là quá dở. Thay vào đó, bạn nên thừa kế lớp Base với đặc tính private để che dấu toàn bộ dữ liệu.

**Lưu ý: Bạn chỉ nên thay đổi đặc tính của phương thức truy xuất của lớp cở sở để có thể truy xuất dữ liệu. không nên đổi đặc tính dữ liệu thành viên private của lớp cơ sở để truy xuất, vì lớp dẫn xuất không nên được phép thấy dữ liệu private lớp cơ sở.**

1. Đa thừa kế:

Một lớp trong C++ có thể thừa kế thành viên từ nhiều lớp và cú pháp mở rộng như sau:

class lớp\_dẫn\_xuất: access baseA, access baseB....

access là các đặc tính public, protected hay private và luôn khai báo trước tên lớp cơ sở. Mỗi lớp cơ sở trong danh sách thừa kế tách nhau bằng dấu phẩy.

Xem ví dụ dưới đây:

#include <iostream>

using namespace std;

// Base class Shape

class Shape {

public:

void setWidth(int w) { width = w;}

void setHeight(int h) { height = h;}

protected:

int width;

int height;

};

// Base class PaintCost

class PaintCost{

public:

int getCost(int area) { return area \* 70;}

};

// Derived class

class Rectangle: public Shape, public PaintCost{

public:

int getArea() { return (width \* height);}

};

int main(void){

Rectangle Rect;

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

// Print the total cost of painting

cout << "Total paint cost: $" << Rect.getCost(area) << endl;

return 0;

}

Kết quả

Total area: 35

Total paint cost: $2450

1. Vấn đề của đa thừa kế
2. Truy xuất hàm thành viên từ lớp cơ sở

Đa thừa kế làm tăng độ phức tạp của chương trình và là cơn ác mộng với việc mở rộng, bảo trì.

Đầu tiên là việc nhiều lớp cơ sở chứa hàm cùng tên:

#include <iostream>

using namespace std;

class USBDevice{

private:

long m\_lID;

public:

USBDevice(long lID): m\_lID(lID){}

long GetID() { return m\_lID; }

};

class NetworkDevice{

private:

long m\_lID;

public:

NetworkDevice(long lID): m\_lID(lID){}

long GetID() { return m\_lID; }

};

class WirelessAdaptor: public USBDevice, public NetworkDevice{

public:

WirelessAdaptor(long lUSBID, long lNetworkID):

USBDevice(lUSBID),

NetworkDevice(lNetworkID)

{}

};

int main(){

WirelessAdaptor c54G(5442, 181742);

cout << c54G.GetID(); // GetID() nao se duoc goi?

return 0;

}

Bạn đã thấy vấn đề ở trên chưa? Trình biên dịch sẽ gọi GetID() của USBDevice hay NetworkDevice?.

Tuy nhiên, có một cách là bạn dùng toán tử phạm vi USBDevice::GetID(), trình biên dịch sẽ hiểu là hàm của lớp USBDevice được gọi:

int main()

{

WirelessAdaptor c54G(5442, 181742);

cout << c54G.USBDevice::GetID();

return 0;

}

Vấn đề trên trở nên phức tạp hơn với 4 hay 6 lớp cơ sở thừa kế, việc xung đột tên thành viên phải được giải quyết tốt qua việc thiết kế lớp.

1. Thừa kế con thoi

Thứ hai cũng là vấn đề khá trầm trọng, xảy ra khi hai lớp cơ sở lại dẫn xuất từ cùng 1 lớp cơ sở.

|  |  |
| --- | --- |
| class PoweredDevice{};    class Scanner: public PoweredDevice{};    class Printer: public PoweredDevice{};    class Copier: public Scanner, public Printer{}; | http://www.learncpp.com/images/CppTutorial/Section11/PoweredDevice.gif |

Scanner và Printer đều dẫn xuất từ poweredDevice, tuy nhiên Copier sẽ không kết hợp các đặc tính từ cả Scanner và Printer

Ở đây có rất nhiều vấn đề phát sinh:

* Liệu Copier sẽ có một hay hai bản sao cơ sở PoweredDevice khi khởi tạo?
* Giải quyết các loại tham chiếu thế nào?

Hầu hết lỗi phát sinh có thể giải quyết qua việc khai báo phạm vi của thành viên thừa kế, nhưng sẽ thêm công việc bảo trì cho bạn cũng như thời gian phát triển ứng dụng.

Một lời khuyên là bạn nên thiết kế lớp theo cấu trúc cây phân cấp, giúp dễ bảo trì và mở rộng, đồng thời ít phát sinh lỗi hơn.

1. Có nên đa thừa kế?

Các lỗi phát sinh ở trên sẽ không bao giờ gặp khi bạn dùng đơn thừa kế.

Nhiều ngôn ngữ OOP như Samlltalk, PHP, không hỗ trợ đa thừa kế.

Một vài ngôn ngữ hiện đại như Java , C# hạn chế đơn thừa kế từ một lớp thông thường, nhưng cho phép đa thừa kế từ interface class (chúng ta sẽ đề cập sau).

Ý nghĩa đằng sau việc không cho phép đa thừa kế trong những ngôn ngữ này là tối thiểu hóa sự phức tạp và lỗi.

Nhiều lập trình viên kinh nghiệm tin rằng đa thừa kế trong C++ nên tránh bằng mọi giá vì những lỗi tiềm ẩn phía sau. Bạn phải sử dụng nó một cách khôn ngoan nếu muốn.

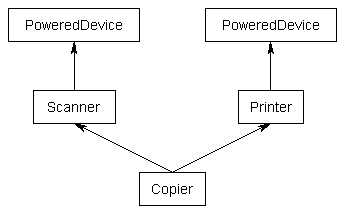
Một điều thú vị là bạn đang dùng một lớp đa thừa kế mà không biết: thư viện <iostream> có đối tượng **cin** và **cout**.

1. Lớp cơ sở ảo

(yêu cầu đã học qua về virtual)

Phần này chúng ta thảo luận sau hơn về vấn đề “thừa kế con thoi”

Một ví dụ với nhiều hàm dựng:

class PoweredDevice{

public:

PoweredDevice(int nPower){

cout << "PoweredDevice: " << nPower << endl;

}

};

class Scanner: public PoweredDevice{

public:

Scanner(int nScanner, int nPower): PoweredDevice(nPower){

cout << "Scanner: " << nScanner << endl;

}

};

class Printer: public PoweredDevice{

public:

Printer(int nPrinter, int nPower): PoweredDevice(nPower){

cout << "Printer: " << nPrinter << endl;

}

};

Nếu bạn tạo đối tượng lớp Copier, mặc định là đối tượng này sẽ dẫn xuất từ lớp Printer và Scanner như sơ đồ trên.

Tạo hàm main như sau:

int main(){

Copier cCopier(1, 2, 3);

}

Kết quả

PoweredDevice: 3

Scanner: 1

PoweredDevice: 3

Printer: 2

Như bạn thấy, hàm dựng PoweredDevice gọi hai lần.

Nhưng nếu bạn chỉ muốn một bản sao của PoweredDevice tạo ra cho cả Scanner và Printer thì sao?

Để chia sẻ lớp cơ sở, chúng ta phải ảo hóa trong sơ đồ phân cấp. Nó tạo ra lớp cơ sở ảo duy nhất chia sẻ cho lớp dẫn xuất.

Ví dụ:

class PoweredDevice{

};

class Scanner: virtual public PoweredDevice{

};

class Printer: virtual public PoweredDevice{

};

class Copier: public Scanner, public Printer{

};

Khi bạn tạo đối tượng Copier, bạn sẽ chỉ lấy một bản sao của PoweredDevice. Tuy nhiên, nó dẫn tới một vấn đề: nếu Scanner và Printer cùng chia sẻ PoweredDevice, vậy lớp nào chịu trách nhiệm tạo nó?

Câu trả lời là lớp Copier. Hàm dựng Copier chịu trách nhiệm tạo PoweredDevice.

Do đó, chỉ có một lần là Copier đươc pehp1 gọi hàm dựng trực tiếp Consequently, this is one time when Copier is allowed to call a non-immediate-parent constructor directly:

class PoweredDevice{

public:

PoweredDevice(int nPower){

cout << "PoweredDevice: " << nPower << endl;

}

};

class Scanner: virtual public PoweredDevice{

public:

Scanner(int nScanner, int nPower): PoweredDevice(nPower){

cout << "Scanner: " << nScanner << endl;

}

};

class Printer: virtual public PoweredDevice{

public:

Printer(int nPrinter, int nPower): PoweredDevice(nPower){

cout << "Printer: " << nPrinter << endl;

}

};

class Copier: public Scanner, public Printer

{

public:

Copier(int nScanner, int nPrinter, int nPower):

Scanner(nScanner, nPower),

Printer(nPrinter, nPower),

PoweredDevice(nPower){}

};

int main(){

Copier cCopier(1, 2, 3);

}

Kết quả

PoweredDevice: 3

Scanner: 1

Printer: 2

Đầu tiên, lớp cơ sở ảo sẽ được tạo trước nhất.

Thứ hai, hàm dựng Scanner và Printer vẫn gọi hàm dựng của PoweredDevice. Nếu chúng ta tạo một bản Scanner và Printer, từ khóa “virtual” bị bỏ qua, những hàm dựng sẽ được gọi và áp dụng luật kế thừa bình thường.

Thứ ba, nếu một lớp thừa kế một hay nhiều lớp có lớp cơ sở ảo, thì lớp dẫn xuất cuối cây phân cấp có trách nhiệm khởi tạo lớp cơ sở ảo.

Trong trường hợp này, Copier thừa kế hai lớp Printer và Scanner, cả hai đều thừa kế từ lớp ảo PoweredDevice. Copier là lớp dẫn xuất cuối, nên phải khởi tạo PoweredDevice.

Chú ý rằng điều này vẫn đúng dù trong trường hợp thừa kế đơn.

### Tính đa hình

1. Con trỏ và tham chiếu tới lớp cơ sở của đối tượng dẫn xuất

Ở phần trước, các bạn đã học về tính thừa kế. Phần chính chương này nói về khía cạnh quan trọng và mạnh mẽ của thừa kế là hàm ảo.

Vậy hàm ảo là gì? Tại sao chúng ta cần hàm ảo?

Khi bàn về lớp dẫn xuất, bạn hiểu rằng lớp dẫn xuất gồm 2 phần: phần của mỗi lớp kế thừa, và phần của chính nó.

Ta sẽ dùng lại lớp Base và lớp Derived:

class Base{

protected:

int m\_nValue;

public:

Base(int nValue): m\_nValue(nValue){}

const char\* GetName() { return "Base"; }

int GetValue() { return m\_nValue; }

};

class Derived: public Base{

public:

Derived(int nValue): Base(nValue){}

const char\* GetName() { return "Derived"; }

int GetValueDoubled() { return m\_nValue \* 2; }

};

Hàm main() sẽ khai báo cả con trỏ và tham chiếu của lớp Derived:

int main(){

Derived cDerived(5);

cout << "cDerived is a " << cDerived.GetName() << " and has value " << cDerived.GetValue() << endl;

Derived &rDerived = cDerived;

cout << "rDerived is a " << rDerived.GetName() << " and has value " << rDerived.GetValue() << endl;

Derived \*pDerived = &cDerived;

cout << "pDerived is a " << pDerived->GetName() << " and has value " << pDerived->GetValue() << endl;

return 0;

}

Kết quả

cDerived is a Derived and has value 5

rDerived is a Derived and has value 5

pDerived is a Derived and has value 5

Tuy nhiên, vì Derived có phần của Base, câu hỏi nữa là liệu C++ có để chúng ta gán con trỏ hay tham chiếu lớp Base tới đối tượng Derived?

Câu trả lời là hoàn toàn có thể:

int main(){

Derived cDerived(5);

// 2 khai báo đều hợp lệ!

Base &rBase = cDerived;

Base \*pBase = &cDerived;

cout << "cDerived is a " << cDerived.GetName() << " and has value " << cDerived.GetValue() << endl;

cout << "rBase is a " << rBase.GetName() << " and has value " << rBase.GetValue() << endl;

cout << "pBase is a " << pBase->GetName() << " and has value " << pBase->GetValue() << endl;

return 0;

}

Kết quả:

cDerived is a Derived and has value 5

rBase is a Base and has value 5

pBase is a Base and has value 5

rBase và pBase là tham chiếu và con trỏ của lớp Base, chúng chỉ có thể thấy thành viên của Base (hoặc bất kì lớp cở nào nào được Base dẫn xuất).

Dù Drived::GetName() đã ghi đè Base::GetName(), con trỏ/tham chiếu Base cũng không thấy. Nên chúng sẽ gọi Base::GetName().

Một ví dụ khác:

class Animal{

protected:

std::string m\_strName;

/\*Chung ta dat ham dung Animal trong protected de public khong the khoi tao Animal truc tiep

\* Nhung cho phep lop dan xuat co the khoi tao duoc

\*/

Animal(std::string strName): m\_strName(strName){}

public:

std::string GetName() { return m\_strName; }

const char\* Speak() { return "???"; }

};

class Cat: public Animal{

public:

Cat(std::string strName): Animal(strName){}

const char\* Speak() { return "Meow"; } //Ghi de

};

class Dog: public Animal{

public:

Dog(std::string strName): Animal(strName){}

const char\* Speak() { return "Woof"; } //Ghi de

};

int main(){

Cat cCat("Fred");

cout << "cCat is named " << cCat.GetName() << ", and it says " << cCat.Speak() << endl;

Dog cDog("Garbo");

cout << "cDog is named " << cDog.GetName() << ", and it says " << cDog.Speak() << endl;

Animal \*pAnimal = &cCat;

cout << "pAnimal is named " << pAnimal->GetName() << ", and it says " << pAnimal->Speak() << endl;

pAnimal = &cDog;

cout << "pAnimal is named " << pAnimal->GetName() << ", and it says " << pAnimal->Speak() << endl;

return 0;

}

Kết quả:

cCat is named Fred, and it says Meow

cDog is named Garbo, and it says Woof

pAnimal is named Fred, and it says ???

pAnimal is named Garbo, and it says ???

1. Cách dùng con trỏ, tham chiếu lớp

Giả sử bạn cần in ra tên và âm thanh của các con thú. Nếu không dùng con trỏ, tham chiếu, bạn sẽ viết thế này:

void Report(Cat &cCat){

cout << cCat.GetName() << " says " << cCat.Speak() << endl;

}

void Report(Dog &cDog){

cout << cDog.GetName() << " says " << cDog.Speak() << endl;

}

Nhưng nếu bạn phải in ra 30 con vật khác nhau thì sao? Phải khai báo 30 hàm?

Một sự lãng phí thời gian vô cùng lớn.

void Report(Animal &rAnimal){

cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;

}

Còn nếu bạn khai báo như trên dùng tham chiếu lớp Animal cũng không làm việc vì nó chỉ gọi Animal::GetName() và Animal::Speak().

Xét ví dụ khác:

int main()

{

Cat acCats[] = { Cat("Fred"), Cat("Tyson"), Cat("Zeke") };

Dog acDogs[] = { Dog("Garbo"), Dog("Pooky"), Dog("Truffle") };

for (int i=0; i < 3; i++)

cout << acCats[i].GetName() << " says " << acCats[i].Speak() << endl;

for (int i=0; ii< 3; i++)

cout << acDogs[i].GetName() << " says " << acDogs[i].Speak() << endl;

return 0;

}

Dùng vòng lặp có vẻ ổn cho mảng đối tượng Cat và Dog. Nhưng với 30 loài, phải dùng 30 vòng lặp?

Giải pháp sau cũng không làm việc được:

int main()

{

Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");

Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");

// mảng con trỏ animal, trỏ tới mỗi đối tượng Cat và Dog

Animal \*apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson, &cZeke };

for (int i=0; i < 6; i++)

cout << apcAnimals[i]->GetName() << " says " << apcAnimals[i]->Speak() << endl;

return 0;

}

Hai cách dùng trên đều chung một vấn đề là tham chiếu, con trỏ lớp cơ sở không thể thấy được thành viên lớp dẫn xuất

Giải pháp ở đây chính là hảm ảo

1. Hàm ảo – quá tải con trỏ, tham chiếu lớp

Hàm ảo là một hàm đặc biệt để giải quyết mọi phiên bản hàm của lớp dẫn xuất với cùng một cách.

Khả năng này còn gọi là tính đa hình

Cú pháp:

Virtual <tên\_hàm> (danh sách tham số){định nghĩa};

Lưu ý: Hàm ảo và lớp ảo là hai khái niệm khác nhau.

Ví dụ

#include <iostream>

using namespace std;

class Base{

public:

virtual const char\* GetName() { return "Base"; }

};

class Derived: public Base{

public:

virtual const char\* GetName() { return "Derived"; }

};

int main(){

Derived cDerived;

Base &rBase = cDerived;

cout << "rBase is a " << rBase.GetName() << endl;

return 0;

}

Kết quả

rBase is a Derived

Vì Base::GetName() là hàm ảo, nên trình biên dịch sẽ kiểm tra xem có lớp dẫn xuất nào không. Vì đối tượng tham chiếu rBase đại diện có một phần là của lớp Derived, trình biên dịch sẽ tìm hiểu mọi lớp thừa kế giữa Base và Derived và dùng phiên bản hàm dẫn xuất phù hợp nhất mà nó thấy.

Trong trường hợp này là Derived::GetName().

Lật lại ví dụ về lớp Animal ở phần trước, chúng ta sẽ khai báo virtual cho Speak()

#include <iostream>

#include <string>

using namespace std;

class Animal{

protected:

string m\_strName;

/\*hàm dựng protected để Animal không bị khởi tạo trực tiếp từ public

\* mà phải từ hàm dựng lớp dẫn xuất\*/

Animal(std::string strName): m\_strName(strName){}

public:

std::string GetName() { return m\_strName; }

virtual const char\* Speak() { return "???"; }

};

class Cat: public Animal{

public:

Cat(std::string strName): Animal(strName){}

const char\* Speak() { return "Meow"; }

};

class Dog: public Animal{

public:

Dog(std::string strName): Animal(strName){}

const char\* Speak() { return "Woof"; }

};

Khai báo hàm Report() để in tên và tiếng nói của các loài vật:

void Report(Animal &rAnimal){

cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;

}

int main(){

Cat cCat("Fred");

Dog cDog("Garbo");

Report(cCat);

Report(cDog);

return 0;

}

Kết quả:

Fred says Meow

Garbo says Woof

rAnimal đang tham chiếu tới phần vùng nhớ lớp Animal của đối tượng truyền vào. Khi rAnimal.Speak() được gọi, vì là hàm ảo nên trình biên dịch sẽ tìm các lớp thừa kế giữa Animal và đối tượng truyền vào và chọn hàm dẫn xuất phù hợp nhất nếu thấy.

Ở đây lần lượt là Cat::Speak() và Dog::Speak.

Tương tự cho ví dụ về mảng con trỏ Animal:

int main(){

Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");

Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");

Animal \*apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson, &cZeke };

for (int iii=0; iii < 6; iii++)

cout << apcAnimals[iii]->GetName() << " says " << apcAnimals[iii]->Speak() << endl;

return 0;

}

Kết quả

Fred says Meow

Garbo says Woof

Pooky says Woof

Truffle says Woof

Tyson says Meow

Zeke says Meow

**Lưu ý: Bạn chỉ cần khai báo virtual cho hàm thành viên lớp cơ sở, trình biên dịch sẽ kiểm tra, không cần khai báo cho lớp dẫn xuất.**

**Hàm lớp dẫn xuất phải là overwrite thì trình biên dịch mới giải quyết được tức phải trùng cú pháp với hàm lớp cơ sở**

1. Truy xuất phương thức override từ lớp cơ sở

Để truy xuất phương thức override từ lớp cơ sở chúng ta dùng toán tử phạm vi “::” .Ví dụ:

class Base{

public:

virtual const char\* GetName() { return "Base"; }

};

class Derived: public Base{

public:

virtual const char\* GetName() { return "Derived"; }

};

int main(){

Derived cDerived;

Base &rBase = cDerived;

// Gọi hàm Base::GetName() thay vì Derived::GetName()

cout << rBase.Base::GetName() << endl;

}

1. Từ khóa override

Có khả năng khi ghi đè hàm, chúng ta vô tình tạo ra các phương thức thành viên mới do cú pháp khai báo không khớp giữa lớp cơ sở và lớp dẫn xuất:

class Base

{

virtual void A(float=0.0);

virtual void B() const;

};

class Derived: public Base

{

virtual void A(int=0); // Tham số là int thay vì float, xem như triển khai hàm mới

virtual void B(); // Khai báo hàm thành viên non-const, xem như triển khai hàm mới

};

Kết quả là hàm A và B của lớp Base vẫn có thể được gọi thay vì của lớp Derived

C++11 giới thiệu từ khóa **override** cho phép khai báo tường minh phương thức sẽ ghi đè. Nếu việc ghi đè thất bại, trình biên dịch sẽ báo lỗi. Ví dụ:

class Base

{

virtual void A(float=0.0);

virtual void B() const;

virtual void C();

void D();

};

class Derived: public Base

{

virtual void A(int=0) override; // lỗi vì Derived::A(int) không override Base::A(float)

virtual void B() override; // lỗi vì Derived::B() không override Base::B() const

virtual void C() override; // ok! Derived::C() overrides Base::C()

void D() override; // oke!

};

Chúng ta nên dùng **override** để kiểm khả năng ghi đè, tránh các lỗi không có chủ ý.

(If you’re wondering why this was implemented as an identifier rather than a keyword, I presume this was done so that the name “override” can be used as a normal variable name in other contexts. If it had been defined as a keyword, it would be reserved in all contexts, which might break existing applications)

1. Từ khóa final

C++11 có thêm từ khóa **final**, được dùng khi bạn không muốn cho phép người khác ghi đè hàm ảo hay thừa kế lớp.

Ví dụ dùng final để cấm ghi đè hàm thành viên ảo:

class Base

{

virtual void A() final; // final identifier marks this function as non-overrideable

};

class Derived: public Base

{

virtual void A(); // cố gắng override final function Base::A() sẽ bị lỗi khi biên dịch

};

Ngăn không cho thừa kế:

class Base final // final identifier marks this class as non-inheritable

{

};

class Derived: public Base // cố gắng override class Base sẽ bị lỗi khi biên dịch

{

};

There are some legitimate reasons to make functions or classes final. For example, the most common use of final is to ensure that an immutable class stays immutable. An immutable class is a specially-designed class whose state cannot be modified after it is created. Without the final identifier, a derived class could add functions that could cause the class to become mutable. If the base class is made final, it cannot be subclassed, and this is avoided.

However, generally speaking, unless you have a really good reason, use of final should generally be avoided. And if you do use the final keyword, document why, as it will likely not be obvious to whoever inherits your code.

1. Trả về tham chiếu, con trỏ của hàm ảo

Một lưu ý khi các bạn làm việc với hàm ảo trả về kiểu tham chiếu, con trỏ

Ví dụ:

class Base{

public:

// GetThis() trả về con trỏ tới Base

virtual Base\* GetThis() { return this; }

};

class Derived: public Base{

public:

// Thông thường hàm override phải trả về đồi tượng cùng kiểu với hàm base. Tuy nhien, vì Derived dẫn xuất từ Base, nên trình biên dịch sẽ trả về Derived\* thay vì Base\*

virtual Derived\* GetThis() { return this; }

};

1. Hàm hủy ảo

Dù C++ tự cung cấp hàm hủy mặc định nếu bạn không khai báo, nhưng trường hợp trong định nghĩa lớp có dùng đối tượng cấp phát động, thì bạn phải khai báo hàm hủy để giải phóng vùng nhớ.

Trong lớp thừa kế, bạn nên thiết lập hàm hủy ảo cho lớp. Ví dụ:

class Base{

public:

    ~Base(){

        cout << "Calling ~Base()" << endl;

    }

};

class Derived: public Base{

private:

    int\* m\_pnArray;

public:

    Derived(int nLength)    {

        m\_pnArray = new int[nLength];

    }

    ~Derived() // note: not virtual{

        cout << "Calling ~Derived()" << endl;

        delete[] m\_pnArray;

    }

};

int main(){

    Derived \*pDerived = new Derived(5);

    Base \*pBase = pDerived;

    delete pBase;

    return 0;

}

pBase là kiểu con trỏ Base, khi pBase bị xóa, chương trình kiểm tra hàm hủy Base có phải hàm ảo? Nếu không phải, thì nó chỉ gọi hàm hủy của Base mà thôi. Kết quả in ra là:

Calling ~Base()

Nhưng mục đích của chúng ta là gọi hàm hủy của Derived, nên phải đặt hàm hủy ảo cho Base:

class Base{

public:

    virtual ~Base(){

        cout << "Calling ~Base()" << endl;

    }

Hàm main() sau khi chạy sẽ in:

Calling ~Derived()

Calling ~Base()

Một lần nữa, để làm việc tốt với đối tượng thừa kế, luôn đặt tính ảo cho hàm hủy.

1. Phép gán ảo

Không giống như hảm hủy ảo, ảo hóa toán tử gán thực sự là ý tưởng tồi.

Phần này chỉ khuyên bạn không thiết lập tính ảo cho phép gán.

1. Hạn chế của hàm ảo

Hầu như lúc nào bạn cũng cần hàm ảo, tại sao không làm cho mọi phương thức đều ảo?

Câu trả lời ở đây là tính hiệu quả. Việc giải quyết hàm ảo sẽ mất nhiều thời gian hơn hàm thông thường.

Thêm vào đó, trình biên dịch cũng phải cấp phát thêm vùng nhớ cho con trỏ mỗi đối tượng của lớp có một hay nhiều hàm ảo.

Chúng ta sẽ nói rõ hơn vấn đề này trong phần sau.

1. Interface (Abstract class)
2. Giới thiệu

Một interface mô tả hành vi và khả năng của một lớp trong C++ , nó không triển khai bất kỳ phương thức nào cả.

C++ interface triển khai dùng lớp trừu tượng, đừng nhầm lẫn với kiểu dữ liệu trừu tượng (Danh sách liên kết .. )

Một lớp được xem là trừu tượng nếu chỉ có ít nhất 1 hàm thuần ảo. Hàm thuần ảo được đặt “=0” phía sau khai báo:

class Box{

public:

// pure virtual function

virtual double getVolume() = 0;

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

1. Mục đích

Mục đích lớp trừu tượng là cung cấp lớp cơ sở cho thừa kế. Không thể dùng lớp trừu tượng để tạo đối tượng và dùng chuyên cho khai báo interface.

Mọi cố gắng tạo đối tượng từ abstract class sẽ bị lỗi khi biên dịch

Do đó, các subclass từ abstract class cần định nghĩa từng hàm ảo cho riêng nó, nếu có hàm thuần ảo chưa override, trình biên dịch sẽ báo lỗi.

1. Thiết kế:

Hệ thống hướng đối tượng sử dụng abstract base class để cung cấp interface chuẩn và phổ biến cho các ứng dụng bên ngoài. Thông qua thừa kế, các lớp dẫn xuất hoạt động cùng một cách.

Khả năng của ứng dụng sẽ được cung cấp bởi hàm thuần ảo trong lớp interface. Việc triển khai hàm thuần ảo nằm trong lớp dẫn xuất tùy vào yêu cầu ứng dụng.

Nền tảng này cho phép các ứng dụng được mở rộng và phát triển một cách dễ dàng.

### Cơ chế hàm ảo

1. Địa chỉ hàm

Trong phần nay và tiếp theo, chúng ta sẽ tìm hiểu cách hàm ảo được triển khai.

Khi ứng dụng C++ được thực thi, nó thực thi tuần tự, bắt đầu từ hàm main(). Khi hàm gọi triệu tập, thời đểm thực thi nhảy tới phần đầu của hàm gọi. Vậy làm sao máy tính biết được?

Khi biên dịch, trình biên dịch chuyển từng dòng lệnh thành một hay nhiều dòng mã máy. Mỗi dòng mã máy được gán một địa chỉ độc nhất. Ở đây không có khác biệt đối với hàm gọi.

Khi hàm được gọi, nó chuyển thành mã máy và gán địa chỉ. Do đó, mỗi hàm đều phải có địa chỉ theo ngôn ngữ máy.

Binding là quá trình chuyển tên biến và tên hàm thành địa chỉ mã máy. Chúng ta sẽ chỉ tìm hiểu về binding hàm

1. Early binding

Ví dụ về lệnh gọi hàm trực tiếp:

#include <iostream>

void printValue(int value)

{

std::cout << value;

}

int main()

{

printValue(5); // This is a direct function call

return 0;

}

Lệnh gọi hàm trực tiếp được phân giải bằng tiến trình early binding, hay còn gọi là static binding, nghĩa là trình biên dịch hay trình liên kết có khả năng ánh xạ tên hàm hay biến với địa chỉ mã máy.

Mọi hàm đều có địa chỉ riêng biệt, nên khi trình biên dịch gặp lệnh gọi hàm, nó thay thế lệnh gọi bằng chỉ thị mã máy nhảy tới địa chỉ của hàm

Ví dụ về trình tính toán dùng early binding:

#include <iostream>

int add(int x, int y)

{

return x + y;

}

int subtract(int x, int y)

{

return x - y;

}

int multiply(int x, int y)

{

return x \* y;

}

int main()

{

int x;

std::cout << "Enter a number: ";

std::cin >> x;

int y;

std::cout << "Enter another number: ";

std::cin >> y;

int op;

do

{

std::cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";

std::cin >> op;

} while (op < 0 || op > 2);

int result = 0;

switch (op)

{

// gọi hàm với early binding

case 0: result = add(x, y); break;

case 1: result = subtract(x, y); break;

case 2: result = multiply(x, y); break;

}

std::cout << "The answer is: " << result << std::endl;

return 0;

}

Vì add(), subtract(), and multiply() đều là lệnh gọi trực tiếp, trình biên dịch sẽ dùng early binding để ánh xạ.

1. Late Binding

Trong vài ứng dụng, không thể nào biết trước được hàm nào sẽ được gọi cho tới khi run time. Đây được gọi là quá trình late binding (hay dynamic binding).

Trên C++, một cách để late binding là dùng con trỏ hàm với toán tử gọi hàm của nó ().

Ví dụ lệnh gọi hàm add():

#include <iostream>

int add(int x, int y)

{

return x + y;

}

int main()

{

// Tạo con trỏ hàm và khiến nó trỏ tới hàm add

int (\*pFcn)(int, int) = add;

std::cout << pFcn(5, 3) << std::endl; // add 5 + 3

return 0;

}

Lệnh gọi hàm thông qua con trỏ hàm được gọi là lệnh gọi gián tiếp.

Ví dụ ứng dụng tính toán sử dụng con trỏ hàm thay vì lệnh gọi trực tiếp:

#include <iostream>

int add(int x, int y)

{

return x + y;

}

int subtract(int x, int y)

{

return x - y;

}

int multiply(int x, int y)

{

return x \* y;

}

int main()

{

int x;

std::cout << "Enter a number: ";

std::cin >> x;

int y;

std::cout << "Enter another number: ";

std::cin >> y;

int op;

do

{

std::cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";

std::cin >> op;

} while (op < 0 || op > 2);

// Tạo con trỏ hàm pFcn (yes, the syntax is ugly)

int (\*pFcn)(int, int);

// Set con trỏ hàm trỏ tới hàm

switch (op)

{

case 0: pFcn = add; break;

case 1: pFcn = subtract; break;

case 2: pFcn = multiply; break;

}

// Call the function that pFcn is pointing to with x and y as parameters

// This uses late binding

std::cout << "The answer is: " << pFcn(x, y) << std::endl;

return 0;

}

Khi ta gọi hàm qua con trỏ, trình biên dịch không thể sử dụng early binding để phân giải hàm gọi pFcn(x,y) vì nó không thể chỉ ra được hàm nào mà pFcn sẽ trỏ tới tại thời điểm biên dịch do nó lệ thuộc vào điều kiện switch.

Late binding có hiệu suất hơi thấp so với early binding. Trong late binding, ứng dụng phải đọc địa chỉ giữ bởi con trỏ và sau đó nhảy tới đó.

Late binding tốn thêm một bước, tuy nhiên late binding có ưu điểm là linh hoạt.

1. Bảng ảo

Để triển khai hàm ảo, C++ dùng hình thái đặc biệt của late binding với tên gọi bảng ảo. Bảng ảo là bảng tra cứu hàm dùng để ánh xạ hàm gọi theo phương pháp late binding.

Bảng ảo thỉnh thoảng có tên khác như virtual function table hay virtual method table, hay dispatch table

Phần này chúng ta sẽ tìm hiểu quá trình đọc bảng ảo

* Đầu tiên: mọi lớp dùng hàm ảo (hay hàm ảo được thừa kế) đều có bảng ảo riêng, bảng ảo này đơn giản là mảng static được trình biên dịch tạo ra trong thời đểm biên dịch.

Bảng ảo sẽ cấp 1 entry cho 1 hàm ảo của lớp, mỗi entry đơn giản là con trỏ hàm trỏ tới hàm ảo nằm trong lớp thấp nhất trong cây thừa kế.

* Thứ hai: trình biên dịch thêm vào con trỏ ẩn cho lớp cơ sở, chúng ta gọi là \*\_\_vptr. \*\_\_vptr được thiết lập tự động khi một thực thể của lớp tạo ra, khi đó con trỏ \*\_\_vptr được dùng để tham chiếu tới bảng ảo của lớp.

Khác với con trỏ \*this, \*this là tham số hàm dùng bởi trình biên dịch để ánh xạ tham chiếu chính nó, \*\_\_vptr là con trỏ thực sự.

Hệ quả là nó khiến cho đối tượng trở nên lớn hơn vì kích thước con trỏ. \*\_\_vptr sẽ được lớp dẫn xuất thừa kết.

Cho ví dụ cây thừa kế đơn giản gồm 3 cấp:

class Base

{

public:

virtual void function1() {};

virtual void function2() {};

};

class D1: public Base

{

public:

virtual void function1() {};

};

class D2: public Base

{

public:

virtual void function2() {};

};

Vì có 3 lớp nên trình biên dịch sẽ thiết lập 3 bảng ảo cho : Base, D1 và D2. Đồng thời thêm con trỏ ẩn trỏ tới base class dùng hàm ảo.

Để hiểu cơ chế này hoạt động thế nào, chúng ta demo ví dụ sau:

class Base

{

public:

FunctionPointer \*\_\_vptr;

virtual void function1() {};

virtual void function2() {};

};

class D1: public Base

{

public:

virtual void function1() {};

};

class D2: public Base

{

public:

virtual void function2() {};

};

Khi đối tượng được tạo, con trỏ \*\_\_ptr được set trỏ tới bảng ảo của lớp đó.

1. Xây dựng bảng ảo

Chúng ta sẽ tìm hiểu về cách bảng ảo được tạo.

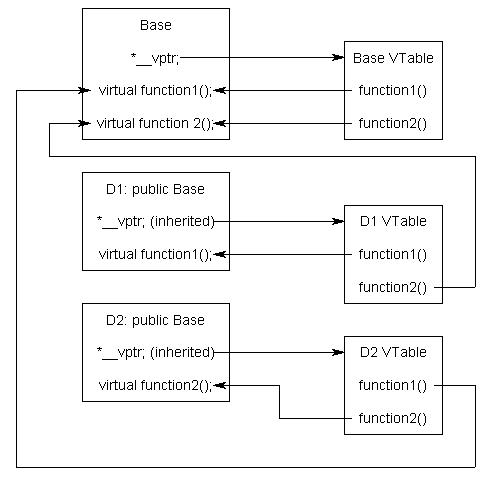
Xét lại ví dụ phần trước, mỗi bảng ảo chỉ có 2 entry (một cho function1, một cho function2). Khi tạo entry, entry đó phải trỏ tới hàm ảo của lớp thấp nhất trên cây thừa kế phân cấp

Bảng ảo cho lớp Base khá đơn giản. Đối tượng của lớp Base chỉ có thể truy xuất thành viên của Base, không có quyền truy xuất hàm của D1 hay D2. Do đó, bảng ảo của Base gồm 2 entry trỏ tới Base::function1() và Base::function2().

Bảng ảo D1 vừa có thể truy xuất thành viên D1 và cả Base. Tuy nhiên D1 đã override function1, do đó entry function1 phải trỏ tới D1::function1() và entry còn lại trỏ tới Base::function2().

Bảng ảo D2 tương tự, entry function1 trỏ tới Base::function1() và entry còn lại trỏ tới D2::function2()

Hình minh họa cơ chế



Ví dụ tạo còn trỏ kiểu Base\* dPtr, nó chỉ trỏ tới phần Base của đối tượng d1, tuy nhiên con trỏ \*\_\_vptr sẽ vẫn trỏ tới bảng ảo của D1. Do đó, dù dPtr là kiểu con trỏ Base, nó vẫn dùng bảng ảo D1 để tham chiếu hàm ảo.

int main()

{

D1 d1;

Base \*dPtr = &d1;

dPtr->function1();

}

Khi gọi dPtr->function1(), ứng dụng nhận biết đây là hảm ảo, nó sẽ tham chiếu tới bảng ảo qua con trỏ \*\_\_vptr.

ứng dụng sẽ tra cứu phiên bản của function1() và gọi tới D1::function1()

Xét trường hợp con trỏ kiểu Base trỏ tới đối tượng Base thực sự:

int main()

{

Base b;

Base \*bPtr = &b;

bPtr->function1();

}

Con trỏ \_\_vptr\* sẽ trỏ tới bảng ảo của Base nên nó sẽ tham chiếu tới hàm Base::function1(), vì đây là phiên bản thấp nhất có thể gọi.

Bằng cách dùng các bảng ảo, trình biên dịch và ứng dụng có thểm đảm bảo hàm gọi được ánh xạ đến đúng hàm ảo, thậm chí bạn đang dùng con trỏ hay tham chiếu tới lớp cơ sở nào đi nữa.

Việc gọi hàm ảo sẽ tốn chi phí hơn hàm non-virtual vì hai lý do:

* Phải dùng con trỏ \*\_\_vptr để tham chiếu tới bảng ảo
* Sau đó tra cứu bảng ảo để tìm đúng phiên bản hàm ảo
* Sau đó gọi hàm

Tuy nhiên, trên các dòng máy tính hiện đại ngày nay thì chi phí đó khá nhỏ.

Ngoài ra, bất kì lớp nào dùng hàm ảo đều phải có con trỏ \_\_vptr\*, do đó mỗi đối tượng của lớp đó tạo ra sẽ gia tăng kích thước bằng 1 con trỏ.

Hàm ảo vô cùng mạnh mẽ, nhưng phải tốn chi phí cho nó.

### Khả năng quá tải (Toán tử và hàm)

C++ cho phép bạn có nhiều hơn một định nghĩa cho một hàm hay một toán tử trong cùng một phạm vi, gọi là quá tải hàm và quá tải toán tử.

Khai báo quá tải là khai báo cùng tên với khai báo trước đó trong cùng phạm vi, nhưng có đối số hàm và định nghĩa khác nhau.

Khi bạn gọi một hàm hay toán tử quá tải, trình biên dịch xác định định nghĩa phù hợp nhất để dùng với tiêu chí so sánh kểu đối số truyền vào hay kiểu toán tử với kiểu tham số đi sau trong định nghĩa.

Quá trình chọn lọc hàm hay toán tử quá tải phù hợp nhất được gọi là phân tích quá tải.

1. Quá tải hàm trong C++:

Bạn có thể có nhiều định nghĩa cho cùng một hàm trong cùng phạm vi. Định nghĩa của các hàm này phải khác nhau bởi kiểu dữ liệu hoặc số lượng đối số hàm. Bạn không thể quá tải hàm chỉ bằng kiểu dữ liệu trả về khác nhau.

Ví dụ về quá tải hàm **print()** , để in ra các kiểu dữ liệu khác nhau:

#include <iostream>

using namespace std;

class printData {

public:

void print(int i) { cout << "Printing int: " << i << endl; }

void print(double f) { cout << "Printing float: " << f << endl; }

void print(char\* c) { cout << "Printing character: " << c << endl; }

};

int main(void){

printData pd; // Call print to print integer

pd.print(5); // Call print to print float

pd.print(500.263); // Call print to print character

pd.print("Hello C++");

return 0;

}

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

1. Quá tải toán tử trong C++:

Bạn có thể tái định nghĩa hay quá tải hầu hết toán tử có sẵn trong C++. Do đó lập trình viên có thể dùng toán tử với kiểu dữ liệu người dùng định nghĩa

Quá tải toán tử là các hàm với cách đặt tên đặc biệt : từ khóa **operator** đứng trước toán tử cần định nghĩa. Cũng giống như hàm, quá tải toán tử phải có kiểu trả về và danh sách tham số

Box operator+(const Box& b);

Quá tải toán tử “+” cho phép cộng hai đối tượng lớp Box với nhau và trả về đối tượng Box mới. Quá tải toán tử có thể là hàm thông thường hay hàm thành viên. Trường hợp trên là hàm thành viên nên chỉ có một đối số.

Trong trường hợp chúng ta định nghĩa hàm trên như là hàm thông thường thì phải truyền hai đối số như sau:

Box operator+(const Box& a, const Box& b);

Ví dụ sau cho bạn nắm rõ hơn về quá tải dùng hàm thành viên. Đối tượng khác được truyền vào đối số hàm, đối tượng gọi hàm quá tải này được truy xuất nhờ toán tử **this**:

#include <iostream>

using namespace std;

class Box {

public:

double getVolume(void){ return length \* breadth \* height; }

void setLength( double len ){ length = len; }

void setBreadth( double bre ){ breadth = bre; }

void setHeight( double hei ){ height = hei; }

// Overload + operator to add two Box objects.

Box operator+(const Box& b){

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

// Main function for the program

int main( ){

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Box Box3; // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

1. Toán tử có thể và không thể quá tải:

Danh sách toán tử có thể quá tải:

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

Danh sách toán tử không thể quá tải:

|  |  |  |  |
| --- | --- | --- | --- |
| :: | .\* | . | ?: |

1. Toán tử nhất nguyên

Toán tử nhất nguyên là toán tử chỉ có một toán hạng:

* Toán tử tăng (++) và giảm (--)
* Toán tử số âm (-)
* Toán tử luận lý phủ định (!)

Thông thường toán tử nhất nguyên xuất hiện bên trái của đối tượng ( !obj, -obj và ++obj..) nhưng thỉnh thoảng chúng có thể được dùng như hậu tố (obj++, obj--…)

Ví dụ sau quá tải toán tử tiền tố, hậu tố số âm (-)

#include <iostream>

using namespace std;

class Distance{

private:

int feet; // 0 to infinite

int inches; // 0 to 12

public:

// required constructors

Distance(){

feet = 0;

inches = 0;

}

Distance(int f, int i){

feet = f;

inches = i;

}

// method to display distance

void displayDistance() {

cout << "F: " << feet << " I:" << inches <<endl;

}

// overloaded minus (-) operator

Distance operator- () {

feet = -feet;

inches = -inches;

return Distance(feet, inches);

}

};

int main(){

Distance D1(11, 10), D2(-5, 11);

-D1; // apply negation

D1.displayDistance(); // display D1

-D2; // apply negation

D2.displayDistance(); // display D2

return 0;

}

Kết quả

F: -11 I:-10

F: 5 I:-11

Trong hàm quá tải minus -, dòng lệnh

return Distance(feet, inches)

có nghĩa hàm return về đối tượng lớp Distance cùng dữ liệu khởi tạo, sau khi kết thúc hàm, đối tượng này cũng bị hủy luôn.

1. Toán tử nhị nguyên

Toán tử nhị nguyên nhận hai đối số, bạn dùng toán tử nhị phân rất thường xuyên như phép cộng (+), trừ (-) và chia (/)

Ví dụ cách toán tử cộng (+) quá tải. Tương tự bạn có thể quá tải toán tử trừ (-), gán (=), nhân (\*) và chia (/)

#include <iostream>

using namespace std;

class Box{

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

public:

double getVolume(void) {

return length \* breadth \* height;

}

void setLength( double len ) {

length = len;

}

void setBreadth( double bre ) {

breadth = bre;

}

void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

Box operator+(const Box& b) {

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

};

// Main function for the program

int main( ){

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Box Box3; // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

Kết quả

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

1. Toán tử tiền tố, hậu tố ++; --

Toán tử tăng giảm phải được triển khai thành hai phiên bản: hậu tố và tiền tố.

Cú pháp triển khai tiền tố tương tự như toán tử nhất nguyên, tức không cần tham số. Riêng toán tử tăng giảm hậu tố, cần tham số nặc danh kiểu int - kỹ thuật hình nộm tham số để phân biệt với tiền tố.

Ví dụ interface lớp Digit:

class Digit

{

private:

int m\_digit;

public:

Digit(int digit=0) : m\_digit(digit){}

Digit& operator++(); // tiền tố

Digit& operator--(); // tiền tố

Digit operator++(int); // hậu tố

Digit operator--(int); // hậu tố

};

Triển khai đa năng toán tử cho Digit:

Digit& Digit::operator++(){

++m\_digit;

return \*this;

}

Digit& Digit::operator--(){

--m\_digit;

return \*this;

}

Digit Digit::operator++(int){

// Tạo đối tượng tạm với giá trị m\_digit hiện tại

Digit temp(m\_digit);

// Dùng tiền tố ++ tăng giá trị lên cho đối tượng hiện tại

++(\*this);

// trả về đối tượng tạm

return temp;

}

Digit Digit::operator--(int){

// Tạo đối tượng tạm với giá trị m\_digit hiện tại

Digit temp(m\_digit);

// Dùng tiền tố -- giảm giá trị lên cho đối tượng hiện tại

--(\*this);

// trả về đối tượng tạm

return temp;

}

Cả 2 phiên bản đều làm chung công việc – tăng giảm giá trị của đối tượng. Nhưng toán tử tiền tố trả về con trỏ \*this sau khi tăng giảm trên đối tượng.

Toán tử hậu tố, phải trả về trạng thái ban đầu của đối tượng, cách phổ biến là sử dụng đối tượng tạm thời khởi tạo trong phương thức để trả về trạng thái đối tượng trước khi tăng giảm.

Điều đó có nghĩa kiểu trả về của hậu tố phải là kiểu giá trị, không phải reference, sử dụng hậu tố sẽ tốn chi phí hơn tiền tố.

Cuối cùng, dùng toán tử tiền tố trong triển khai hậu tố làm công việc đơn giản hơn.

1. Toán tử quan hệ

Có nhiều toán tử quan hệ trong C++ (<, >, <=, >=, ==, …), dùng để so sánh các kiểu dữ liệu sẵn có.

Bạn có thể quá tải bất kì toán tử quan hệ nào, để so sánh hai đối tượng của lớp.

Ví dụ sau mô tả quá tải toán tử (<) và tương tự các toán tử quan hệ khác

#include <iostream>

using namespace std;

class Distance{

private:

int feet; // 0 to infinite

int inches; // 0 to 12

public:

// required constructors

Distance(){

feet = 0;

inches = 0;

}

Distance(int f, int i){

feet = f;

inches = i;

}

// method to display distance

void displayDistance() {

cout << "F: " << feet << " I:" << inches <<endl;

}

// overloaded minus (-) operator

Distance operator- () {

feet = -feet;

inches = -inches;

return Distance(feet, inches);

}

// overloaded < operator

bool operator <(const Distance& d){

if(feet < d.feet){

return true;

}

if(feet == d.feet && inches < d.inches){

return true;

}

return false;

}

};

int main(){

Distance D1(11, 10), D2(5, 11);

if( D1 < D2 ) {

cout << "D1 is less than D2 " << endl;

}

else {

cout << "D2 is less than D1 " << endl;

}

return 0;

}

Kết quả

D2 is less than D1

1. Toán tử xuất nhập

Các toán tử chèn stream (<<) và khai stream (>>) có thể quá tải cho phép đối tượng xuất , nhập như kiểu dữ liệu có sẵn.

Hàm quá tải cho các kiểu có sẵn nằm trong namespace std gồm hai tham số. Cú pháp hàm:

istream& operator>> (istream& in, <kiểu dữ liệu>& <tên biến>)

ostream& operator<< (ostream& out, <kiểu dữ liệu>& <tên biến>)

Điều quan trọng ở đây là phải đặt thuộc tính **friend** cho hàm quá tải của lớp vì nó phải được gọi mà không lệ thuộc đối tượng nào, hoặc bạn có thể khai báo hàm quá tải nằm ngoài lớp.

Ví dụ minh họa:

#include <iostream>

using namespace std;

class Distance{

private:

int feet; // 0 to infinite

int inches; // 0 to 12

public:

// required constructors

Distance(){

feet = 0;

inches = 0;

}

Distance(int f, int i){

feet = f;

inches = i;

}

friend ostream &operator<<( ostream &output, const Distance &D ){

output << "F : " << D.feet << " I : " << D.inches;

return output;

}

friend istream &operator>>( istream &input, Distance &D ){

input >> D.feet >> D.inches;

return input;

}

};

int main(){

Distance D1(11, 10), D2(5, 11), D3;

cout << "Enter the value of object : " << endl;

cin >> D3;

cout << "First Distance : " << D1 << endl;

cout << "Second Distance :" << D2 << endl;

cout << "Third Distance :" << D3 << endl;

return 0;

}

Kết quả:

$./a.out

Enter the value of object :

70

10

First Distance : F : 11 I : 10

Second Distance :F : 5 I : 11

Third Distance :F : 70 I : 10

1. Toán tử truy xuất [ ]

Toán tử [] dùng để truy xuất các phần tử trong mảng, bạn có thể quá tải toán tử này cho các đối tượng.

Ví dụ:

#include <iostream>

using namespace std;

const int SIZE = 10;

class safearay{

private:

int arr[SIZE];

public:

safearay(){// constructor.

register int i;

for(i = 0; i < SIZE; i++){

arr[i] = i;

}

}

int &operator[](int i){

if( i > SIZE ){

cout << "Index out of bounds" <<endl;

// return first element.

return arr[0];

}

return arr[i];

}

};

int main(){

safearay A;

cout << "Value of A[2] : " << A[2] <<endl;

cout << "Value of A[5] : " << A[5]<<endl;

cout << "Value of A[12] : " << A[12]<<endl;

return 0;

}

Kết quả

Value of A[2] : 2

Value of A[5] : 5

Index out of bounds

Value of A[12] : 0

1. Toán tử truy xuất thành viên (->)

Toán tử truy xuất thành viên có thể quá tải, đây được xem là mẹo. Nó được định nghĩa để đưa ra một kiểu lớp có hành vi giống con trỏ. Toán tử -> phải là hàm thành viên.

Nếu sử dụng, nó trả về kiểu phải là con trỏ hay đối tượng của lớp mà chúng ta chỉ định.

Toán tử -> thường được dùng khi liên kết với toán tử tham chiếu (\*) khi triển khai “con trỏ thông minh”. Những con trỏ này là các đối tượng có hành vi giống như con trỏ đối tượng truyền vào, chúng cho phép truy xuất tới các hàm thành viên thông qua toán tử quá tải ->

Tham chiếu quá tải -> có thể được định nghĩa như là toán tử hậu tố một ngôi. Ví dụ:

class Ptr{

//...

X \* operator->();

};

Các phương thức thành viên của lớp X được truy xuất gián tiếp bởi lớp Ptr thông qua quá tải ->. Chúng ta sử dụng như sau:

void f(Ptr p )

{

p->m = 10 ; // (p.operator->())->m = 10

}

Câu lệnh p->m được biên dịch thành (p.operator->())->m. Hãy xem thêm một ứng dụng của toán tử quá tải ->

1. Toán tử new và delete

[Tham khảo về cách quản lý vùng nhớ trên C++ tại đây](#_New/Delete:_cái_nhìn).

### Ép kiểu trong OOP

Ép kiểu tự định nghĩa được dùng ngầm định như một tính năng mở rộng cho ép kiểu khi khởi tạo đối tượng, truyền tham số hàm, giá trị trả về của hàm, biểu thức gán, biểu thức kiểm soát vòng lặp, lệnh selection và ép kiểu tường minh.

Có 2 loại ép kiểu người dùng tự định nghĩa là:

* Conversion constructors
* Conversion functions

1. Hàm dựng ép kiểu

Là hàm dựng chỉ nhận duy nhất một tham số được khai báo mà không có từ khóa **explicit**. Trình biên dịch sử dụng hàm dựng ép kiểu để chuyển đối tượng tham số sang kiểu của lớp.

class Y {

int a, b;

char\* name;

public:

Y(int i) {

};

Y(const char\* n, int j = 0) {

};

};

void add(Y) {

};

int main() {

Y obj1 = 2; // tương đương với obj1 = Y(2)

Y obj2 = "somestring";// tương đương với obj2 = Y("somestring",0)

obj1 = 10; // tương đương với obj1 = Y(10)

add(5); // tương đương với add(Y(5))

}

Đoạn code trên có sử dụng 2 hàm dựng để ép kiểu:

* Y(int i) để chuyển số nguyên sang đối tượng lớp Y.
* Y(const char\* n, int j = 0) để chuyển con trỏ chuỗi sang đối tượng lớp Y

Trình biên dịch sẽ không ngầm định ép kiểu nếu hàm dựng có khai báo từ khóa **explicit**.

Trình biên dịch sẽ chỉ sử dụng hàm dựng với khai báo **explicit** trong toán tử **new,** biểu thức **static\_cast** trong ép kiểu tường minh, và trong quá trình khởi tạo.

Xem ví dụ sau:

class A {

public:

explicit A() {

};

explicit A(int) {

};

};

int main() {

A z;

//A y = 1; not allow!

A x = A(1); //Gọi tường minh hàm dựng

A w(1); // Gọi tường minh hàm dựng

A\* v = new A(1); // Gọi tường minh hàm dựng

A u = (A) 1; // Ép kiểu tường minh

A t = static\_cast<A> (1); // Ép kiểu tường minh

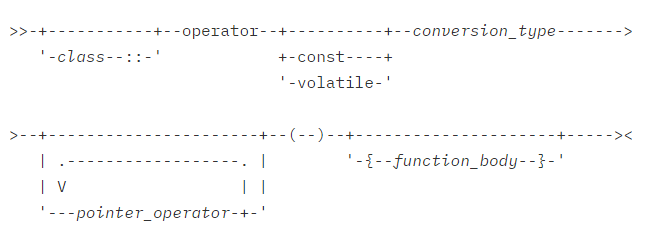
}

Dòng lệnh sau không hợp lệ A y = 1 vì đây là ép kiểu ngầm định, class A không có hàm dựng ép kiểu ngầm định.

1. Hàm ép kiểu

Có thể định nghĩa hàm thành viên ép kiểu, chuyển kiểu lớp hiện tại sang kiểu khác.

Cú pháp hàm thành viên ép kiểu



Một hàm ép kiểu thuộc class X sẽ làm thao tác ép kiểu đối tượng lớp X sang đối tượng khác.

Đoạn code sau khai báo hàm ép kiểu đối tượng sang kiểu int dùng operator int()

class Y {

int b;

public:

operator int();

};

Y::operator int() {

return b;

}

void f(Y obj) {

int i = int(obj);

int j = (int) obj;

int k = i + obj;

}

Cả 3 dòng lệnh trong hàm f() dùng hàm chuyển kiểu **Y::operator int()**

**Tên class, tên enum, tên typedef**, kiểu trả về hàm hay kiểu mảng không thể khai báo hay định nghĩa trong conversion\_type

* Không thể dùng hàm ép kiểu để chuyển đối tượng lớp A sang lớp A, lớp cơ sở của A hay void.
* Hàm ép kiểu **không có đối số,** và kiểu trả về được **ngầm định ép kiểu**.
* Hàm ép kiểu có thể **được thừa kế**.
* Hàm ép kiểu có thể là **virtual**, nhưng không được là **static**.

Từ khóa **auto** có thể dùng trong ép kiểu để tự suy ra kiểu trả về:

struct X {

operator int(); // OK

operator auto() -> short; // lỗi: phần kiểu trả về không nằm trong cú pháp hàm ép kiểu

operator auto() const { return 10; } // OK: tự luận kiểu

};

****

**Lưu ý: Hàm ép kiểu không được phép dùng template để tự luận ra kiểu trả về.**

### Big-5:

C++11 mặc định lớp sẽ có 5 hàm đặc biệt cung cấp sẵn: hàm hủy, hàm dựng và phép gán sao chép, hàm dựng và phép gán move.

Thỉnh thoảng, 5 hàm mặc định có thể chấp nhận được, thỉnh thoảng thì không.

1. Hàm hủy

Hàm hủy tự được gọi khi đối tượng ra khỏi phạm vi. Nó chỉ có trách nhiệm giải phóng tài nguyên cấp phát của đối tượng. (delete cho new, đóng file đang mở,…). Mặc định là áp dụng hàm hủy cho từng dữ liệu thành viên.

1. Hàm dựng sao chép và move

Nhiệm vụ của 2 hàm đặc biệt này là khởi tạo đối tượng mới từ đối tượng khác truyền vào.

* Hàm dựng sao chép được gọi nếu tham số là L\_value
* Hàm dựng move được gọi nếu tham số là R\_value.

Ví dụ lớp IntCell:

IntCell B = C; // Copy construct if C is lvalue; Move construct if C is rvalue

IntCell B { C }; // Copy construct if C is lvalue; Move construct if C is rvalue

Mặc định, hàm dựng sao chép sẽ áp dụng hàm dựng sao chép cho từng dữ liệu thành viên . Đối với dữ liệu kiểu đối tượng, tùy trường hợp hàm dựng sao chép hay move được gọi cho mỗi thành viên của chính nó.

1. Phép gán sao chép và move

Toán tử gán được gọi cho hai đối tượng đã khởi tạo.

* Phép gán sao chép được gọi khi vế phải là L\_value
* Phép gán move được gọi khi vế phải là R\_value

Mặc định, toán tử gán sao chép sẽ gọi toán tử gán sao chép cho từng thành viên dữ liệu.

1. Tính mặc định

Với dữ liệu thành viên là kiểu cơ bản như: int, double, vector<int>, string, và thậm chí vector<string> sẽ hoạt động tốt với hàm mặc định.

Tuy nhiên, khi lớp chứa thành viên dữ liệu con trỏ, có cấp phát thì chúng ta phải dùng Deep Copy như đã đề cập phần trước.

### Shallow copy và Deep copy

1. Hàm dựng và toán tử gán sao chép

Mặc dù toán tử gán khá dễ hiểu, nhưng để làm đúng lại cần chút mánh khóe vì hai lý do sau:

* Có vài trường hợp toán tử gán không được gọi như dự định.
* Phát sinh khi làm việc với cấp phát động

Mục đích của phép gán vẫn là sao chép giá trị các thành viên dữ liệu từ đối tượng này sang đối tượng kia.

Cents cMark(5); // Gọi hàm dựng của Cents

Cents cNancy = cMark; // Hàm dựng sao chép của Cents được gọi cho cNancy

Dòng lệnh thứ 2 xảy ra theo thứ tự nào?

BÀN THÊM VỀ HÀM DỰNG SAO CHÉP

Dòng lệnh thứ 2, kì vọng gọi phép gán quá tải. Tuy nhiên, nó không thực hiện! mà gọi hàm dựng sao chép (dùng để khởi tạo đối tượng từ một đối tượng có sẵn).

Mục đích hàm dựng sao chép và phép gán là khá tương đồng. Tuy nhiên, phép gán dùng để sao chép tới đối tượng đã có. Còn hàm dựng sao chép dùng để khởi tạo đối tượng mới từ đối tượng khác.

3 trường hợp hàm dựng sao chép được gọi thay vì phép gán:

* Khi đang khởi tạo 1 đối tượng với đối tượng khác (bằng phép gán hay hàm dựng sao chép)
* Truyền giá trị đối tượng vào đối số hàm
* Khi đối đượng trả về kiểu giá trị từ hàm

Triển khai lớp noname như sau:

class noname{

private:

int i;

public:

//ham dung mac dinh

noname(){

cout<<"constructor called"<<endl;

i = 0;

}

// ham dung sao chep

noname(const noname &c1){

cout<<"copy constructor called"<<endl;

this->i = c1.i;

}

//qua tai phep gan

noname &operator=(const noname &c1);

};

Có hai điều chúng ta cần lưu ý:

* Hàm dựng sao chép cho phép truy xuất dữ liệu private của đối tượng
* Tại sao phải truyền kiểu tham chiếu?

Điều thứ hai dễ hiểu là khi bạn truyền kiểu giá trị, trình biên xem đối tượng tạm là kiểu hằng .

Vậy còn trong quá tải phép gán thì sao? Chúng ta quá tải như sau:

//qua tai phep gan

noname &operator=(const noname &c1){

cout<<"Operator = called"<<endl;

//sao chep du lieu

this->i = c1.i;

//tra ve dia chi doi tuong hien tai

return (\*this);

}

C++ vẫn cho phép truy xuất đối dữ liệu private trong phép gán. Nhưng tại sao lại phải return (\*this), trả về tham chiếu tới đối tượng hiện tại?

Mục đích là triển khai phép gán theo chuỗi

cMark = cNancy = cFred = cJoe; // gán cJoe cho các dối tượng phía trước

Cuối cùng, C++ cho phép gán chính nó:

cMark = cMark; // gán hợp lệ

Hàm quá tải phép gán sẽ chẳng làm gì cả, nhưng trường hợp có cấp phát động bạn cần tránh điều trên. Bằng cách thêm bước kiểm tra địa chỉ:

//qua tai phep gan

noname &operator=(const noname &c1){

cout<<"Operator = called"<<endl;

//kiem tra dia chi

if (this == &c1)

return \*this;

//sao chep du lieu

this->i = c1.i;

//tra ve dia chi doi tuong hien tai

return \*this;

}

SAO CHÉP THÀNH VIÊN MẶC ĐỊNH

Cũng giống như hàm dựng, C++ cũng cung cấp phép gán mặc định cho đối tượng nếu không khai báo.

Vì C++ không hiểu cấu trúc lớp của bạn, nên C++ dùng phương thức sao chép thành viên thông minh (còn gọi là shallow copy).

1. Shallow Copying

Là phương thức C++ dùng cho hàm dựng và phép gán mặc định khi bạn không khai báo. Nó rất tiện cho bạn, khi dùng

cấu trúc lớp đơn giản và không có cấp phát động.

Nhưng khi làm việc với cấp phát động, shallow copy đưa nhiều rắc rối cho chúng ta hơn! Vì toán tử gán con trỏ chuẩn chỉ sao chép địa chỉ của con trỏ - không thực hiện cấp phát bộ nhớ hay sao chép nội dung của con trỏ tới.

class MyString{

private:

char \*buffer;

int length;

public:

MyString(char \*pchString="")

{

// Set độ dài chuỗi + 1 cho kí tự kết zero

length = strlen(pchString) + 1;

// Cấp phát bộ đệm với độ dài trên

buffer= new char[length];

// Sao chép chuỗi vào đệm

strncpy(buffer, pchString, length);

// set lại byte kết zero cho chắc

buffer[length-1] = '\0';

}

~MyString() // hàm hủy

{

// hủy cấp phát bộ đệm

delete[] buffer;

// set con trỏ bộ đệm về NULL / 0

buffer = 0;

}

char\* GetString() { return buffer; }

int GetLength() { return length; }

};

Bên trên xây dựng lớp chứa chuỗi có dùng cấp phát động, hàm dựng tham số có đầu vào là chuỗi. Nhưng không khai báo hàm dựng sao chép và phép gán quá tải.

Xét đoạn code sau, C++ sẽ dùng phương thức shallow copy:

int main(){

MyString cHello("Hello, world!");

{

MyString cCopy = cHello; // ham dung sao chep mac dinh duoc goi

// cCopy goes out of scope here

}

cout << cHello.GetString() << endl; // crash tai day

return 0;

}

cHello được khởi tạo, cấp phát vùng nhớ cho chuỗi tham số nhập vào.

Tại dòng lệnh thứ 2, rắc rối nằm ở đây! C++ dùng hàm dựng sao chép mặc định, shallow copy địa chỉ con trỏ bộ đệm từ cHello sang cCopy. Như vậy bộ đệm hai đối tượng cùng trỏ tới vùng nhớ.

Khi chương trình thực thi ra ngoài phạm vi {}, hàm hủy cCopy được gọi và đồng thời hủy cấp phát vùng nhớ bộ đệm của cCopy, đồng nghĩa vùng nhớ bộ đệm của CHello cũng bị hủy luôn.

Dòng lệnh cout đang in ra giá trị vùng nhớ không giá trị nên crash!

1. Deep Copy

Giải pháp cho trường hợp trên là bạn phải dùng deep copy cho tất cả con trỏ != NULL. Deep copy sao y đối tượng hay vùng nhớ trỏ tới, theo cách này, đối tượng cần được sao chép có thể làm mọi điều nó muốn mà không ảnh hưởng tới đối tượng khách.

Phương pháp này yêu cầu bạn phải tự hard coding cho hàm dựng sao chép và phép gán quá tải.

Chúng ta triển khai hàm dựng deep copy như sau:

// Khai báo hàm dựng sao chép

MyString(const MyString& cSource) {

// vì length không phải con trỏ, shallow copy nó

length = cSource.length;

// buffer là con trỏ cấp phát động nên dùng deep copy nếu !=NULL

if (cSource.buffer) {

// cấp phát vùng nhớ

buffer = new char[length];

// soa chép chuỗi

strncpy(buffer, cSource.buffer, length);

}

else

buffer = 0;

}

Thử chạy lại, bạn sẽ không thấy bị lỗi nữa.

Để quá tại phép gán dùng deep copy, bạn cần dùng mẹo nhỏ:

//quá tải phép gán

MyString& operator=(const MyString& cSource) {

// kiểm tra địa chỉ, tránh self-copy

if (this == &cSource)

return \*this;

// Hủy vùng nhớ hiện thời trước!

delete[] buffer;

// shallow copy length

length = cSource.length;

// kiểm tra buffer có = NULL?

if (cSource.buffer) {

// cấp phát mới

buffer = new char[length];

// sao chép chuỗi

strncpy(buffer, cSource.buffer, length);

}

else

buffer = 0;

return \*this; // Trả về địa chỉ cho phép cách gán liên tiếp

}

Việc hủy vùng nhớ và tái cấp phát theo kích thước mới, giúp bạn tránh trường hợp thiếu hụt vùng nhớ nếu chuỗi mới dài hơn.

1. Kiểm Tra Địa Chỉ Đối Tượng Gán, Tránh Self - Copy

Có nguyên nhân khiến ta phải làm như vậy:

* Tối ưu hóa hiệu suất: không cần phải làm việc vô bổ như vậy.
* Tránh mất thông tin :khi bạn hủy và tái cấp phát vùng nhớ, đồng nghĩa bạn đang hủy thông tin nằm trong vùng nhớ và sao chép thông tin rác vào vùng nhớ con trỏ.

1. Kết Luận

* Hàm dựng sao chép/phép gán mặc định chỉ thực hiện shallow copy, nơi cấu trúc lớp không dùng cấp phát động.
* Bạn phải dùng deep copy cho lớp có cấp phát động
* Phép gán quá tải thường code tương tự như hàm dựng sao chép, nhưng phải kiểm tra self-copy, return (\*this), và hủy vùng nhớ cấp phát trước khi deep copy.

1. Ngăn copy

Nếu bạn không muốn diễn ra việc sao chép, mẹo là khai báo chúng trong private:

class MyString{

private:

char \*buffer;

int length;

MyString& operator=(const MyString& cSource);

MyString(const MyString& cSource);

//rest of code...

};

C++ sẽ không tự động gọi hàm dựng sao chép/phép gắn mặc định.

### Ngoại lệ an toàn

##### Quan điểm về xử lí ngoại lệ

Mục đích chính của việc xử lí ngoại lệ là giúp hệ thống ổn định, thông suốt và tránh thất thoát tài nguyên. Các phương pháp truyền thống chỉ ra nhiều nhược điểm khi triển khai trên hệ thống lớn:

* Ngắt chương trình : là 1 phương án tồi, cần làm gì đó tốt hơn như xuất lỗi hay ghi log. Nếu là thư viện, nó sẽ không biết mục đích của chương trình là gì, không thể để một thư viện như vậy nhúng vào chương trình đòi hỏi không có crash.
* Trả về giá trị lỗi: thường không hiệu quả, một số hàm cũng trả về kiểu số. Cũng không thuận tiện khi mỗi lần gọi lại phải kiểm tra, điển hình là hàm printf(), nó trả về số âm khi xảy ra lỗi nhưng hiếm khi được lập trình viên để ý. Hàm dựng cũng không cần phải trả về gì cả.
* Trả về giá trị hợp lệ và kệ chương trình trong trạng thái lỗi: ví dụ một số hàm của C, có set biến ngoài errno

Double d = sqrt(-1.0)

Biến d vô nghĩa, sqrt set code lỗi cho errno vì tham số âm. Phương pháp này không phù hợp cho concurrency.

* Gọi hàm xử lí ngoại lệ: nôm na như **if (somethings\_wrong) handler().** Vấn đề lập tức trở thành “hàm xử lí ngoại lệ sẽ làm gì?” Nếu nó không thể xử lí tốt , sẽ ngắt chương trình hoặc trả về giá trị lỗi hay set code lỗi.. hay ném ra ngoại lệ. Nếu nó làm tốt như vậy thì chúng ta bắt lỗi để làm gì ?

Một đặc điểm khó chịu của ngoại lệ là phản hồi cuối cùng sẽ luôn là ngắt chương trình, bạn sẽ cần phải nhiều nỗ lực, cẩn thận hơn trong phát triển ứng dụng.Chương trình chỉ tự tắt khi có 1 ngoại lệ không bị bắt.

Các biện pháp như xuất lỗi, ghi log, hay hiện cửa sổ thông báo chỉ tốt cho việc debug. Những thư viện hiện thông báo, yêu cầu chỉ dẫn từ người dùng là không phù hợp, một thư viện tốt không nên “ba hoa” kiểu này. Bộ xử lí ngoại lệ có thể ghi log text hay XML cho hệ thống.

Ngoại lệ nên cung cấp phương thức để phát hiện lỗi không thể xử lí, và gửi nó cho một phần của hệ thống có đủ khả năng hồi phục và ghi thông lỗi đủ ý nghĩa nhất trong tình huống nhất định.

Lập trình viên không nên xem ngoại lệ là một cơ chế bắt, xử lí lỗi bình thường. Chúng ta nên coi đó như một cách hoạt động của hệ thống, một cách khác để trả về giá trị.

1. Khi nào không dùng ngoại lệ

Trong một vài tình huống đặc biệt, chúng ta không thể áp dụng cơ chế ngoại lệ:

* Tiến trình hoạt động trên hệ thống nhúng cần hoàn thành trong thời gian nhất định, vì **throw catch** sẽ mất khá khá thời gian
* Việc kiểm soát tài nguyên khi bạn dùng cơ chế không tự động như **new, delete.**

Lúc này ta phải dùng các phương pháp truyền thống, hãy dùng 2 phương pháp sau giải quyết:

* Giả lập cơ chế RAII: ban cho mọi hàm dựng phải set erro\_code, triển khai hàm thành viên invalid() trả về lỗi code, erro\_code = 0 nếu thành công, nếu hàm dựng không thể tạo, tức tài nguyên chưa rò rỉ thì erro\_code trả về giá trị khác zero. Chúng ta có thể test invalid() sau mỗi lần tạo đối tượng, giải quyết trường hợp thất bại.

void f(int n){

my\_vector<int> x(n);

if (x.invalid()) {

// ... deal with error ...

}

// ...

}

* Giả lập hàm có thể trả về giá trị lỗi hay throw exception: hàm có thể return về **pair<Value,erro\_code>** , chúng ta test erro\_code sau mỗi lần gọi hàm và xử lí lỗi nếu có.

void g(int n){

auto v = make\_vector(n); // return a pair

if (v.second) {

// ... deal with error ...

}

auto val = v.first;

// ...

}

Hai phương pháp hoạt động khá hiệu quả, nhưng còn luộm thuộm nếu so sánh với cơ chế exception có tính hệ thống

1. Xử lí ngoại lệ phân cấp

Chúng ta cần một chiến lược tổng quát cho ngoại lệ: giả định rằng hệ thống gồm 2 phần hoạt động độc lập, thì mỗi thành phần phải có khả năng giải quyết ngoại lệ và làm gì đó có nghĩa

Cơ chế xư lí phải đồng nhất trên toàn hệ thống, đơn giản và rõ ràng. Những thứ phức tạp không nên đem vào như là việc khôi phục lỗi.

Khiếm khuyết của hệ thống được phân cấp. Mỗi cấp sẽ phải xử lí lỗi trong phạm vi của nó, còn lại để cho cấp cao hơn. Theo thiết kế này, terminal() chỉ diễn ra khi cơ chế xử lí ngoại lệ gặp lỗi hoặc việc bắt các ngoại lệ chưa hoàn hảo, tức để lọi ngoại lệ. Tương tự, noexcept() cũng thoát ra nếu có lỗi khi việc phục hồi là không thể.

Triển khai mỗi hàm như một bức tường lửa : kiểm tra các điều kiện đầu vào, đảm bảo không có lỗi trong giai đoạn cuối. Là không phù hợp trong nhiều ngữ cảnh, con người. Với chương trình lớn:

* Khối lượng công việc sẽ quá lớn để đảm bảo độ tin cậy
* Chi phí hiệu suất bỏ ra quá nhiều
* Phương thức trên ngôn ngữ khác chưa chắc tuân theo quy tắc này
* Độ phức tạp chương trình vô tình trở thành gánh nặng cho tính tin cậy chấp nhận được

Phân chia chương trình thành các hệ thống nhỏ là cốt yếu, khả thi và kinh tế. Tất cả thư viện, hệ thống, các hàm interface nên được thiết kế theo cách này.

Phương pháp ngoại lệ tống quát yêu cầu chúng ta phải triển khai các chiến thuật khác nhau cho từng phần của hệ thống:

* Phải xác định được các vấn đề liên quan của hệ thống thành phần trong việc quản lý tài nguyên
* Trạng thái hệ thống thành phần sau khi nó để xảy ra lỗi là gì?
* Mục đích là phải để hệ thống thành phần tuân theo chiến lược ngoại lệ chung dù cách nó vận hành nội bộ không đồng bộ.

Thỉnh thoảng chúng ta phải chuyển cơ chế báo cáo lỗi dạng này sang dạng khác. Ví dụ như kiểm tra errno và throw ngoại lệ sau khi sử dụng C library hay ngược lại, catch exception và set errno trước khi trả về cho chương trình C từ C++ library:

void callC() // Gọi hàm C từ C++; chuyển err\_no sang throw

{

errno = 0;

c\_function();

if (errno) {

// ... dọn dẹp nội bộ, nếu cần

throw C\_blewit(errno);

}

}

extern "C" void call\_from\_C() noexcept // Gọi hàm C++ từ C; chuyển throw sang err\_no

{

try {

c\_plus\_plus\_function();

}

catch (...) {

// ... dọn dẹp nội bộ, nếu cần

errno = E\_CPLPLFCTBLEWIT;

}

}

Việc chuyển đổi này phải thực hiện đầy đủ trong chiến lược phân cấp ngoại lệ tổng quát. Nếu 1 hàm xảy ra ngoại lệ, nó không nên viện sự trợ giúp từ nơi gọi trong việc phục hồi hay thu hồi tài nguyên. Với những hệ thống hoạt động theo vòng tròn khép kín: chương trình trở nên khó hiểu khi đùn đẩy exception lên trên dẫn tới vòng lặp không giới hạn trong xử lí exception.

1. Exception và hiệu suất

Để sử dụng cơ chế exception hoàn toàn không tốn bất kì chi phí nào trong khi vẫn tương thích với C, thủ tục debug…

Để tiết kiệm chi phí cho việc Throw 1 ngoại lệ, không tốn hơn so với việc gọi hàm, chúng ta có thể làm được dù rất khó khăn

Dễ nhận ra rằng một nửa code trong chương trình truyền thống là để xử lí lỗi, bạn xem đoạn code sau:

void f(){

string buf;

cin>>buf;

// ...

g(1);

h(buf);

}

g() và h() có thể ném ra exception, f() phải đảm bảo buf được hủy đúng cách. Hãy so sánh cách dùng code để xử lí thay vì cơ chế exception:

bool g(int);

bool h(const char∗);

char∗ read\_long\_string();

bool f(){

char∗ s = read\_long\_string();

// ...

if (g(1)) {

if (h(s)) {

free(s);

return true;

}

else {

free(s);

return false;

}

}

else {

free(s);

return false;

}

}

Khi không dùng cơ chế exception, đoạn code trở nên rường rà, lòng vòng. Nên tốt nhất là để cơ chế exception tự lo dù tốn chi phí 1 chút.

Từ khóa **noexcept**, khá hữu ích cho việc tối ưu code.

void g(int) noexcept {}

void h(const string&) noexcept {}

Dù hàm C không thể ném ra ngoại lệ, nhưng vẫn dùng được **noexcept**. Trước khi khai báo, bạn nên cân nhắc xem liệu có xảy ra ngoại lệ hay không.

1. Tiêu chí exception safety

Trước đó, chúng ta gọi 1 tiến trình là an toàn ngoại lệ nếu nó giữ được trạng thái hệ thống tình trạng tốt sau khi nó kết thúc dù quăng ra 1 ngoại lệ.

Khi lý giải về tính ổn định đối tượng: nó nằm ở hàm dựng và các hàm thành viên xem như dại diện để làm việc cho tới khi nó bị hủy.

Vậy ổn định ở đây là hàm dựng đã hoàn tất và hàm hủy chưa đụng tới. Quan điểm đối với dữ liệu cũng tương tự.

Trước khi throw, phương thức phải đảm bảo các đối tượng còn hoạt động (chưa hủy). Cơ chế phục hồi, xử lí lỗi phải xuất ra một giá trị phù hợp cho ứng dụng hơn là exit trong mệnh đề catch.

Tất cả đối tượng trong thư viện chuẩn C++ được thiết kế theo nguyên tắc ngoại lệ an toàn, nó bảo đảm các tiêu chí:

1. Đảm bảo tính ổn định các đối tượng, không để thất thoát tài nguyên. Bạn có thể hủy hay gán đối tượng trả về từ mọi phương thức thư viện chuẩn (theo ISO)
2. Đảm bảo an toàn cho các phương thức quan trong: như push\_back(), insert() trên list và uninitialized\_copy()
3. Đảm bảo không throw một số phương thức: như std::swap trên container và pop\_back().

Tiêu chí thứ nhất và thứ 2 phải thỏa điều kiện:

* Phương thức không để phần tử container trong tình trạng không hợp lệ. Và không để rò rỉ tài nguyên như: vùng nhớ, File, locks, network connection và thread.
* Hàm hủy không được throw
* Hàm dựng phải throw: nếu xảy ra ngoại lệ trong quá trình khởi tạo thành viên dữ liệu phải có cách xử lí phù hợp

Điều kiện cho hàm hủy và hàm dựng là luật cốt lõi của C++ cho ngoại lệ an toàn

Cho ví dụ: khi xảy ra tình huống ngoại lệ, tài nguyên phải đảm bảo thu hồi trước khi throw.

void f(int i){

int∗ p = new int[10];

// ...

if (i<0) {

delete[] p; // hủy trước khi throw hoặc bị rò rỉ tài nguyên

throw Bad();

}

// ...

}

***Destructors do not throw exceptions (§iso.17.6.5.12).***

***The C++ language rules for partial construction and destruction ensure that exceptions thrown***

***while constructing subobjects and members will be handled correctly without special attention***

***from standard-library code (§17.2.3).***

Ngoài ra, chúng ta chỉ nên throw 1 exception duy nhất, tức phải cấu trúc lại code để không xổng một con chuột nào khỏi bẫy

##### Quản lý tài nguyên RAII

1. Triển khai hàm dựng, hàm hủy

Nguyên tắc cơ bản: có cấp phát , có thu hồi. Ví dụ thao tác trên FILE:

void use\_file(const char∗ fn) {

FILE∗ f = fopen(fn,"r");

// ... use f ...

fclose(f);

}

Ngoại lệ có thể xảy ra trong lần gọi fopen, khiến hàm exit trước khi fclose(). Các lỗi na ná thế này luôn xảy ra cho các phương thức không được triển khai an toàn ngoại lệ của C++.

Biện pháp thông thường là đặt bẫy try catch:

void use\_file(const char∗ fn) {

FILE\* f = fopen(fn,"r");

try {

}

catch (...) { // catch mọi ngoại lệ

fclose(f);

throw;

}

fclose(f);

}

Mệnh đề catch hoạt động đúng nguyên tắc: xử lí đóng file trước khi throw.

***Vấn đề của try catch là:***

* Cách xử lí dài dòng, nhạt nhẽo, tốn chi phí.
* Với tình huống xấu: code trở nên phức tạp khi có nhiều tài nguyên cấp phát và thu hồi.

Phân tích chi tiết, ta dễ nhận thấy thủ tục cấp phát luôn theo 1 chu trình như sau:

void funct(){

// cấp phát 1

// ...

// cấp phát n

// ... sử dụng ...

// thu hồi n

// ...

// thu hồi 1

}

Tài nguyên được cấp phát và thu hồi có sự tương quan với hành vi của hàm dựng và hàm hủy. Vậy chúng ta có thể quản lý tài nguyên theo kiểu đối tượng.

Triển khai lớp File\_ptr, hoạt động giống như FILE\*:

class File\_ptr {

FILE\* p;

public:

// Hàm dựng nhận tham số fopen, mở file n với method \*a

File\_ptr(const char∗ n, const char∗ a): p{fopen(n,a)} {

if (p==nullptr) throw runtime\_error{"File\_ptr: Can't open file"};

}

File\_ptr(const string& n, const char∗ a) :File\_ptr{n.c\_str(),a}{ }// open file n

// Hàm dựng nhận tham số FILE\*

explicit File\_ptr(FILE\* pp):p{pp} { // assume ownership of pp

if (p==nullptr) throw runtime\_error("File\_ptr: nullptr"};

}

// Triển khai Big 5…

˜File\_ptr() { fclose(p); }// Hàm hủy đóng FILE

// Quá tải toán tử FILE\*

operator FILE\*() { return p; }

};

Trong mọi tình huống, hàm hủy File\_ptr luôn được gọi khi ra ngoài phạm vi. Hàm dựng File\_ptr, throw nếu không thể mở file, đơn giản là kiểm tra nullptr cho con trỏ \*p.

Phương thức use\_file sẽ đơn giản hơn rất nhiều:

void use\_file(const char∗ fn){

File\_ptr f(fn,"r");

// ... use f ...

}

Dù phương thức thoát ra bình thường hay có ngoại lệ, hàm hủy vẫn hoạt động độc lập. Cơ chế RAII này cho phép tối ưu phần xử lí ngoại lệ.

1. Hàm dựng phải throw

Đối tượng được gọi là đã khởi tạo khi hàm dựng hoàn tất. Khi hàm dựng throw, cơ chế stack unwinding sẽ gọi hàm hủy cho đối tượng theo chuỗi: các đối tượng thành viên hay đối tượng đã khởi tạo thành công sẽ lần lượt được gọi hàm hủy trong quá trình unwinding.

**Tiêu chí thiết kế hàm dựng:**

* Đảm bảo đối tượng được khởi tạo đúng cách.
* Trường hợp thất bại, nó phải phục hồi trạng thái của hệ thống trước khi nó được tạo ra. Tức không để đối tượng thành viên trong trạng thái khởi tạo nửa vời.

Tiêu chí thứ 2 yêu cầu tất cả thành viên đều dùng RAII.

Xét một đối tượng X quản lý 2 loại tài nguyên : file x và mutex y

Khi quá trình cấp phát có thể thất bại và throw exception, có thể do mutex hay file hoặc cả 2, hàm dựng X không được nắm file hay mutex

Chúng ta dùng hai RAII Class, File\_ptr and std::unique\_lock, quá trình cấp phát thành công khi đối tượng RAII khởi tạo thành công

class Locked\_file\_handle {

File\_ptr p;

unique\_lock<mutex> lck;

public:

X(const char∗ file, mutex& m):

p{file ,"rw"}, // cấp phát ‘‘file’’

lck{m} {}// cấp phát mutex ‘‘m’’

// ...

};

RAII Class tự động giám sát và thu hồi tài nguyên, lập trình viên chẳng phải làm gì cả. Nếu ngoại lệ xảy ra sau khi File\_ptr khởi tạo, thì hàm hủy của File\_ptr tự động được gọi chứ không phải của unique\_lock.

1. Hàm hủy không được throw

Khác với hàm dựng, ngoại lệ không được xảy ra trong hàm hủy, do đang trong quá trình stack unwinding, trình biên dịch sẽ không biết được là tiếp tục quá trình stack unwinding hay xử lí ngoại lệ mới. Cuối cùng là chương trình sẽ phải tự ngắt.

Thay vào đó là ghi một tin nhắn vào log cho an toàn.

1. Dùng thư viện chuẩn

Hầu hết các container trong thư viện chuẩn đều được triển khai RAII. Khi so sánh với cách truyền thống new/delete, đã tiết kiệm chi phí và tránh lỗi rất nhiều.

Trường hợp đang sử dụng con trỏ trỏ tới đối tượng cấp phát nằm ngoài phạm vi, tốt nhất là dùng smart pointer để tự động thu hồi.

Tùy tình huống chúng ta dùng unique\_ptr hay shared\_ptr. [**Tham khảo ở đây**](#_C++_11_Sử)

1. Phương thức Finally

Người ta hay đề cập việc xây dựng một thủ tục “finally”, viết code tùy hứng, để dọn sạch tài nguyên khi xảy ra ngoại lệ.

Kỹ thuật này nhìn chung không tốt hơn RAII, vì nó sẽ là ad hoc (thực hiện theo chu trình tuần tự).

. First, we define a class that will execute an arbitrary action from its destructor.

template<typename F>

struct Final\_action {

Final\_action(F f): clean{f} {}

˜Final\_action() { clean(); }

F clean;

};

The ‘‘finally action’’ is provided as an argument to the constructor.

Next, we define a function that conveniently deduces the type of an action:

template<class F>

Final\_action<F> finally(F f){

return Final\_action<F>(f);

}

Finally, we can test finally():

void test()

// handle undiciplined resource acquisition

// demonstrate that arbitrar y actions are possible

{

int \*p = new int{7}; // probably should use a unique\_ptr (§5.2)

int \*buf = (int\*)malloc(100\*sizeof(int)); // C-style allocation

auto act1 = finally([&]{ delete p;

free(buf); // C-style deallocation

cout<< "Goodby, Cruel world!\n";

}

);

int var = 0;

cout << "var = " << var << '\n';

// nested block:

{

var = 1;

auto act2 = finally([&]{ cout<< "finally!\n"; var=7; });

cout << "var = " << var << '\n';

} // act2 is invoked here

cout << "var = " << var << '\n';

} // act1 is invoked here

var = 0

var = 1

finally!

var = 7

Goodby, Cruel world!

In addition, the memory allocated and pointed to by p and buf is appropriately deleted and free()d.

It is generally a good idea to place a guard close to the definition of whatever it is guarding.

That way, we can at a glance see what is considered a resource (even if ad hoc) and what is to be

done at the end of its scope. The connection between finally() actions and the resources they manip

ulate is still ad hoc and implicit compared to the use of RAII for resource handles, but using finally()

is far better than scattering cleanup code around in a block.

Basically, finally() does for a block what the increment part of a for-statement does for the for

statement (§9.5.2): it specifies the final action at the top of a block where it is easy to be seen and

where it logically belongs from a specification point of view. It says what is to be done upon exit

from a scope, saving the programmer from trying to write code at each of the potentially many

places from which the thread of control might exit the scope.

### Phong cách viết class

##### Ghi chú

Phong cách lập trình OOP vốn làm việc nhiều cho cấu trúc và hàm. Mỗi thành viên nên theo sau bởi các ghi chú để giải thích.

Tuy nhiên, ghi chú cho thủ tục hàm phải khác với ghi chú cho định nghĩa hàm. Theo lẽ thường, bạn sẽ để khối ghi chú đầy đủ trước thủ tục hàm.

Nếu áp dụng cho thành viên lớp, các khối ghi chú sẽ phá vỡ cấu trúc chương trình, là một trong các trường hợp gây ra rắc rối.

* **Bạn chỉ nên đặt ghi chú 1 dòng trước mỗi thủ tục hàm**
* **Đặt ghi chú đầy đủ trước định nghĩa hàm**

Còn về hàm thành viên inline, sẽ phải định nghĩa ngay trong hàm nên bạn cố gắng súc tích các ghi chú.

Nếu thân hàm 5 dòng trở lên, tốt nhất là khai báo thủ tục hàm rồi định nghĩa ngoài lớp.

Cấu trúc của một hàm thành viên nên rút gọn tối đa và phải có khối ghi chú đầy đủ.

Thông thường C++ sẽ có các hàm mặc định: hàm dựng, hàm dựng sao chép, hàm hủy và toán tử gán. Nếu bạn dùng hàm mặc định thì nên để ghi chú lại.

Ví dụ:

class queue {

private:

int data[100]; // Data stored in the queue

int first; // First element in the queue

int last; // Last element in the queue

public:

queue(); // khởi tạo hàng đợi

// queue(const queue &old\_queue)

// dùng hàm dựng sao chép mặc định

// queue operator = (const queue &old\_queue)

// dùng toán tử gán mặc định

// -queue()

// dùng hàm hủy mặc định

void put(int item);// đẩy dữ liệu vào hàng đợi

int get(void); // lấy dữ liệu khỏi hàng đợi

};

##### Hàm try-catch

1. Giới thiệu

Khối try-catch làm việc đủ hiệu quả trong hầu hết trường hợp, nhưng có một vài trường hợp cá biệt thì không. Xem ví dụ:

#include <iostream>

using namespace std;

class A

{

private:

int m\_x;

public:

A(int x) : m\_x(x)

{

if (x <= 0)

throw 1;

}

};

class B : public A

{

public:

B(int x) : A(x)

{

// Chuyện gì xảy ra nếu A khởi tạo thất bại và chúng muốn xử lí ở đây

}

};

int main()

{

try

{

B b(0);

}

catch (int)

{

std::cout << "Oops\n";

}

}

Tại hàm main, đối tượng class B được đặt trong khối try-catch, nếu hàm dựng class A ném ra ngoại lệ, khối try-catch hàm main sẽ bắt được. Do đó ứng dụng sẽ xuất ra lỗi:

Oops

Nhưng nếu chúng ta muốn xử lí ngoại lệ ngay trong khởi tạo class B? Rắc rối là khởi tạo class A được gọi trong danh sách khởi tạo của class B, trước khi đi vào thân hàm dựng của B.

Nên không có cách nào để đưa vào khối try-catch cả, giải pháp là dùng hàm try-catch

1. Hàm try-catch

Hàm try-catch được thiết kế cho phép xử lí ngoại lệ bên ngoài phạm vi thân hàm.

Cú pháp khá phức tạp, xem ví dụ:

#include <iostream>

class A

{

private:

int m\_x;

public:

A(int x) : m\_x(x)

{

if (x <= 0)

throw 1;

}

};

class B : public A

{

public:

B(int x) try : A(x) // thêm từ khóa try trước danh sách khởi tạo

{

}

catch (...) // note từ khóa catch phải cùng level với try

{

// Ngoại lệ từ danh sách khởi tạo thành viên hay thân hàm dựng đều bị bắt ở đây

std::cerr << "Construction of A failed\n";

// Nếu không tường minh re-throw ngoại lệ, nó vẫn được re-throw ngầm định

}

};

int main()

{

try

{

B b(0);

}

catch (int)

{

std::cout << "Oops\n";

}

}

Kết quả:

Construction of A failed

Oops

Có một nguyên tắc là bắt buộc phải re-throw ngoại lệ hiện tại trong hàm try-catch, nếu không ngoại lệ vẫn được re-throw ngầm định.

Đó là lý do tại sao “Oops” vẫn được in ra.

Hàm try-catch rất hiếm khi dùng với hàm non-member, vì không cần thiết, ngược lại thì hàm dựng luôn luôn cần.

1. Không dùng hàm try-catch để thu hồi tài nguyên

Khi khởi tạo đối tượng thất bại, hàm hủy của lớp sẽ không được gọi. Do đó, có người sẽ cố dùng hàm try-catch để dọn dẹp tài nguyên đã được cấp phát trước khi thoát.

Tuy nhiên, việc tham chiếu tới thành viên của đối tượng khởi tạo thất bại sẽ gây ra các hành vi không xác định vì đối tượng đã chết trước khi khối catch được thực thi. Nghĩa là bạn không thể dùng hàm try để dọn dẹp tài nguyên.

Hàm try được dùng chủ yếu cho log trước khi re-throw ngoại lệ vào stack, hay thay đổi kiểu ngoại lệ ném ra.

##### Kiểu dữ liệu lồng

1. Giới thiệu

Xem một ví dụ về class Fruit với enum FruitType:

#include <iostream>

enum FruitType

{

APPLE,

BANANA,

CHERRY

};

class Fruit

{

private:

FruitType m\_type;

int m\_percentageEaten = 0;

public:

Fruit(FruitType type) :

m\_type(type)

{

}

FruitType getType() { return m\_type; }

int getPercentageEaten() { return m\_percentageEaten; }

};

int main()

{

Fruit apple(APPLE);

if (apple.getType() == APPLE)

std::cout << "I am an apple";

else

std::cout << "I am not an apple";

return 0;

}

Kiểu enum FruitType có liên kết chặt và sử dụng nội bộ trong class Fruit, nên triển khai hai đối tượng này độc lập sẽ hơi khó hiểu.

1. Khai báo lồng

Trên C++, kiểu dữ liệu có thể được định nghĩa lồng bên trong lớp với thuộc tính truy xuất phù hợp.

Đoạn code trên có thể sửa lại như sau:

#include <iostream>

class Fruit

{

public:

enum FruitType

{

APPLE,

BANANA,

CHERRY

};

private:

FruitType m\_type;

int m\_percentageEaten = 0;

public:

Fruit(FruitType type) :

m\_type(type)

{

}

FruitType getType() { return m\_type; }

int getPercentageEaten() { return m\_percentageEaten; }

};

int main()

{

// Note: có thể khai báo FruitType thông qua class Fruit

Fruit apple(Fruit::APPLE);

if (apple.getType() == Fruit::APPLE)

std::cout << "I am an apple";

else

std::cout << "I am not an apple";

return 0;

}

Vì FruitType có thuộc tính truy xuất public nên có thể dùng khai báo ở ngoài phạm vi lớp.

Lúc này, class đóng vai trò như một namespace cho các kiểu dữ liệu lồng bên trong.

Lưu ý rằng enum class được xem là một namespace, nếu lồng FruitType bên trong Fuit như một enum class, chúng ta sẽ phải truy xuất enum APPLE thông qua Fruit::FruitType::APPLE

1. Các kiểu khác

Enum là kiểu phổ biến được lồng nhiều nhất trong lớp, C++ cho phép định nghĩa kiểu lồng với typedef, class, struct…

Giống như bất kì thành viên của lớp, lớp lồng có cùng thuộc tính truy xuất với thành viên của lớp bao ngoài.

Tuy nhiên, lớp lồng không có truy xuất đặc biệt tới con trỏ this của lớp bao ngoài.

Chúng ta có thể thấy thư viện STL C++ khai báo lớp lồng khá phổ biến với mô hình iterator.

##### Singleton

1. Static class

Việc khai báo một đối tượng global trên C++ để dùng mọi nơi là ý tồi trong thiết kế, tương tự cho biến hay hàm, vì chúng không tuân theo định luật bao đóng của hướng đối tượng.

Ngoài ra, trong hầu hết ngôn ngữ OOP hiện nay như Java hay C# không hỗ trợ biến hay hàm global.

Một giải pháp để triển khai một đối tượng duy nhất, chỉ chứa các phương thức static.

Một lớp static được nạp vào vùng nhớ khi thực thi chương trình bắt đầu và tồn tại cho tới khi kết thúc.

Các phương thức thành viên trong lớp static có thể gọi mà chả cần một thực thể nào, và cũng không được tạo đối tượng nào từ static class.

Tuy nhiên, static class không phải thiết kế tốt. Một static class sẽ có thể chia ra thành danh sách các hàm subroutine.

1. Mô hình singleton

Khi chỉ muốn tạo một thực thể của class theo chuẩn OOP đúng nghĩa, chúng ta sẽ dùng **Singleton** **.**

Singleton là một trong những mô hình có tính sáng tạo và tốt để tạo đối tượng.

Singleton chỉ duy trì 1 đối tượng duy nhất suốt vòng đời chương trình.

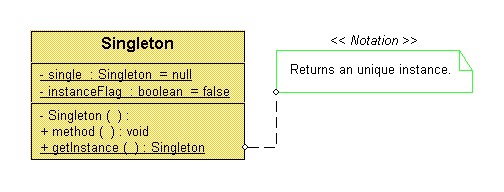
Singleton thường được dùng để kiểm soát truy xuất tài nguyên như kết nối cơ sở dữ liệu hay sockets.

Ví dụ: chúng ta có một license cho 1 kết nối tới Database chẳng hạn, một đối tượng kết nối Singleton đảm bảo luôn chỉ có 1 kết nối trong mọi thời điểm.

1. Triển khai

Triển khai Singleton khá dễ dàng với bất kì ngôn ngữ OOP nào. Có nhiều cách triển khai, nhưng đưa hàm dựng vào **private** và dùng 1 hàm thành viên **static** để khởi tạo và trả về thực thể của lớp là cách phổ biến nhất.

Mô hình Singleton :



#include <iostream>

using namespace std;

class Singleton{

private:

static Singleton \*single;

Singleton()

{

//private constructor

}

public:

static Singleton\* getInstance();

void method();

~Singleton()

{

instanceFlag = false;

}

};

Singleton\* Singleton::single = NULL;

Singleton\* Singleton::getInstance()

{

if(!single)

{

single = new Singleton();

instanceFlag = true;

return single;

}

else

{

return single;

}

}

void Singleton::method()

{

cout << "Method of the singleton class" << endl;

}

int main()

{

Singleton \*sc1,\*sc2;

sc1 = Singleton::getInstance();

sc1->method();

sc2 = Singleton::getInstance();

sc2->method();

return 0;

}

## C/C++ Các chuyên đề

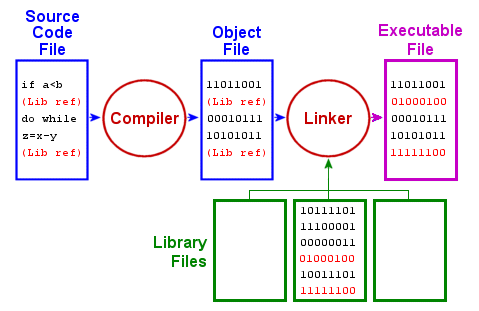
### Quá trình biên dịch

##### Cross-complier

1. Định nghĩa trình biên dịch Compiler

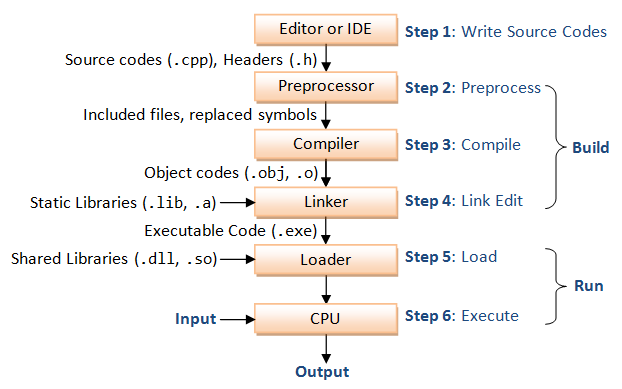
**Trình biên dịch**, còn gọi là **phần mềm biên dịch**, **compiler**, là một [*chương trình máy tính*](http://vi.wikipedia.org/wiki/Ph%E1%BA%A7n_m%E1%BB%81m) làm công việc dịch một chuỗi các câu lệnh được viết bằng một [*ngôn ngữ lập trình*](http://vi.wikipedia.org/wiki/Ng%C3%B4n_ng%E1%BB%AF_l%E1%BA%ADp_tr%C3%ACnh) (gọi là ngôn ngữ nguồn hay [*mã nguồn*](http://vi.wikipedia.org/wiki/M%C3%A3_ngu%E1%BB%93n)), thành một chương trình tương đương nhưng ở dưới dạng một [*ngôn ngữ máy tính*](http://vi.wikipedia.org/wiki/Ng%C3%B4n_ng%E1%BB%AF_m%C3%A1y) mới (gọi là ngôn ngữ đích) và thường là ngôn ngữ ở cấp thấp hơn, như ngôn ngữ máy. Chương trình mới được dịch này gọi *mã đối tượng*

1. Quá trình biên dịch

[](http://4.bp.blogspot.com/-AxCHzAMVJ70/VSvh4Kum_XI/AAAAAAAAUqw/ypEQu-U8Zvc/s1600/compile.gif)

Sơ đồ đơn giản

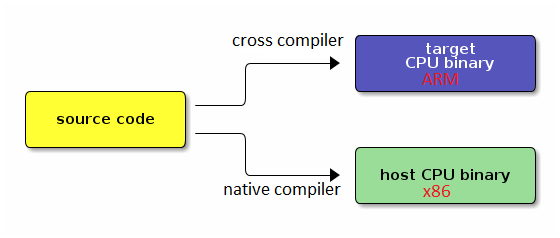
Ví dụ chương trình có sử dụng thư viện toán học <math> , vậy trong quá trình liên kết, bộ Linker sẽ tiến hành lấy mã của hàm printf() trong thư viện để kết hợp với mã thông thường khác.

[](http://2.bp.blogspot.com/-Zgpc8ehovHk/VSvlm6hSicI/AAAAAAAAUq8/750hTN2gOYg/s1600/CompilationProcess.png)

Sơ đồ chi tiết từ Code/Build/Run

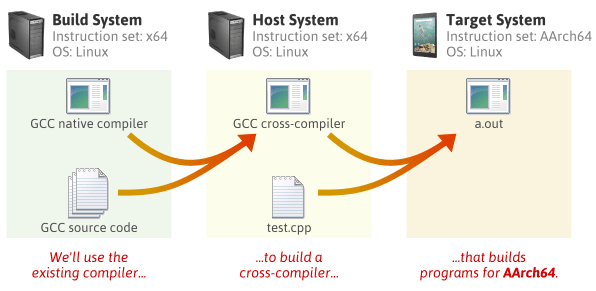
1. Cross Compiler /Toolchain là gì

Cross Compiler hay còn gọi là Toolchain được định nghĩa: là khả năng tạo ra mã lệnh thực thi trên nhiều platform của trình biên dịch.  
Ví dụ: một trình biên dịch có thể chạy trên Windows, nhưng vẫn có thể sinh ra code chạy trên môi trường Android, thì đây là cross compiler.

[](http://3.bp.blogspot.com/-8qxvZEmiqlU/VSvofIqj_lI/AAAAAAAAUrI/fXbCclylkgA/s1600/cross-compile.png)

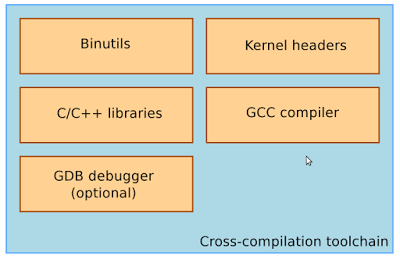
Có nghĩa là source code được viết trên máy tính chạy chíp Intel (x86 platform), thông qua một cross compiler sẽ cho ra file nhị phân (mã máy) có khả năng chạy được trên một nên tảng chip khác là ARM (ARM platform). Trên hình native compiler là trình biên dịch để tạo ra file nhị phân chạy trên chính máy tính đang dùng để viết source code.

1. Hệ thống Cross Compiler

[](http://4.bp.blogspot.com/-S_JgT5HhrHE/VSvsAksY-vI/AAAAAAAAUrU/nRjsob-exDU/s1600/build-gcc-cross-compiler.png)

Quá trình tạo ra và sử dụng tool chain liên quan đến 3 đối tượng:  
+ **Build** : là hệ thống/platform tạo ra tool chain, thường là các máy tính chạy nền tảng chip Intel x86, x84\_64 chạy hệ điều hành Linux hoặc Windows.  
+ **Host**: là hệ thống chạy chương trình tool chain để compile source của một chương trình ứng dụng nào đó, host cũng giống như Build, thường là các máy tính Intel chạy Linux hoặc Windows OS  
+ **Target**: là hệ thống chạy các chương trình (dưới dạng file nhị phân - mã máy) do hệ thống Host tạo ra, target thường là các hệ thống nhúng chạy chip ARM, MIPS, PowerPC, ...; tuy nhiên target cũng có thể là một máy tính bình thường chạy chip Intel.  
  
VD:  
BUILD=x86  
HOST=x86  
TARGTE=arm-linux

1. Các thành phần của complier

[](http://4.bp.blogspot.com/-_FdNNByaepU/VXOmkSp93hI/AAAAAAAAX3E/XOR6rHgsfLg/s1600/com.png)

1. Binutils

Là một tập các công cụ để tạo và quản lý file nhị phân (bin) của target CPU gồm

* **as**: là assembler, nó sinh ra mã nhị phân (binary code) từ assembler source code
* **ld**: trình liên kết (linker)
* **ar, ranlib**: sinh ra file nén .a, sử dụng như là thư viện
* **objdump, readelf, size, nm, strings**: phân tích file nhị phân
* **strip** : để loại bỏ những phần thừa trong file nhị phân để giảm kích thước của chúng

Thông thường để cross-compiler một chương trình ta phải cài đặt biến môi trường mới có thể compile đúng được

export PATH=/path/to/compiler/bin:$PATH #đường dẫn đến thư mục chứa as, ld, ...

export CROSS\_COMPILE=arm-none-linux-gnueabi-

export CC=${CROSS\_COMPILE}gcc

export CXX=${CROSS\_COMPILE}g++

export CPP=${CROSS\_COMPILE}cpp

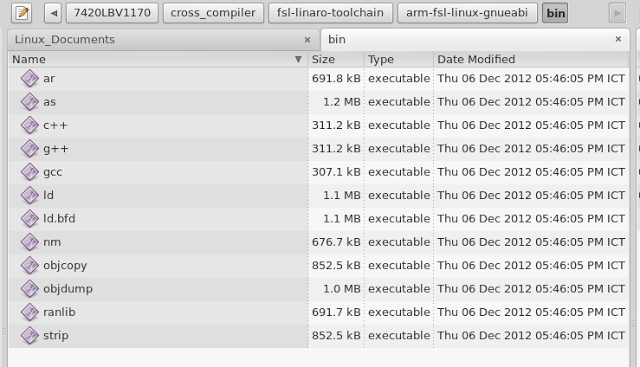
export AR=${CROSS\_COMPILE}ar

export AS=${CROSS\_COMPILE}as

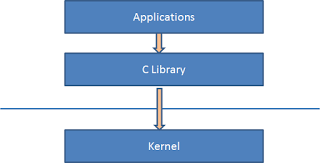
export LD=${CROSS\_COMPILE}ld

export RANLIB=${CROSS\_COMPILE}ranlib

export STRIP=${CROSS\_COMPILE}strip

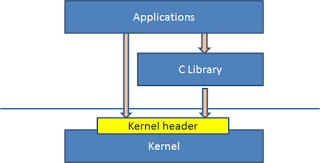
[](http://2.bp.blogspot.com/-Nf6p8sNuE2A/VXOsdpp-94I/AAAAAAAAX3U/XcOL6FCebM8/s1600/comp.png)

1. C/C++ Library

[](http://2.bp.blogspot.com/-wJsjYES4rJw/VXOuTYBTc0I/AAAAAAAAX3c/fkaDQKW5Fxo/s1600/c-lib.png)

Library được dùng làm interface giữa applications và kernel, cung cấp các C API chuẩn để dễ dàng phát triển ứng dụng. Một số libb có thể kể đến như: glibc, uClibc, eglibc, dietlibc, newlib, ...

1. Kernel header

[](http://3.bp.blogspot.com/-pUbez4HWxNw/VXOu_KRKYvI/AAAAAAAAX3k/eW7k0y0NecQ/s1600/kernel-header.png)

Kernel header cần thiết cho Applications và C Library để cung cấp các API giao tiếp với Kernel.

1. GCC compiler

**gcc, c++, g++** : compiler  
Trình biên dịch trong hệ thống Linux, compile cho rất nhiều ngôn ngữ và nhiều kiến trúc CPU khác nhau như ARM, MIPS, PowerPC, SuperH, x86. Tuy nhiên trong blog này chỉ đề cập đến ngôn ngữ C/C++ và kiến trúc CPU là ARM và x86.

1. GDB Debugger

Trình gỡ rối, trợ giúp cho quá trình phát hiện lỗi khi develop application.

1. Cú pháp GCC complier

Ví dụ biên dịch file main.c / main.cpp dùng Terminal command:

$export CFLAGS="-I./include -DDEBUG -Wall -g"

$export LDFLAGS+=" -L./lib -lm"

$**gcc** -c main.c ${CFLAGS} #tạo file object từ source

$**gcc** -o prog main.o ${LDFLAGS} #tạo file chương trình nhị phân từ file object

hoặc

$export CXXFLAGS="-I./include -DDEBUG -Wall -g"

$export LDFLAGS+=" -L./lib -lm"

$**g++** -c main.cpp ${CXXFLAGS} #tạo file object từ source

$**g++** -o prog main.o ${LDFLAGS} #tạo file chương trình nhị phân từ file object

Phía trên là format chung / cơ bản nhất của GCC compiler  
+**CFLAGS / CXXFLAGS**  
C Compiler Flags và C++ preprocessor flags để truyền các options vào trong compiler để thực hiện quá trình biên dịch file source thành các file object, gồm các thông tin:  
   - Đường dẫn các file header, bắt đầu với "-I", ví dụ: -I./include  
   - Các define được bắt đầu với "-D"; ví dụ: -DDEBUG, định nghĩa Macro DEBUG  
   - Các options đặc biệt khác của compiler như: -g để bật chức năng debug gdb của gcc compiler, -Wall để trace các warning trong quá trình compile  
  
+ **LDFLAGS :** Linker flags, dùng trong quá trình linking các thư viện, LDFLAGS chứa các thông tin:  
   - Đường dẫn đến thư viện, được bắt đầu bằng "-L"; ví dụ: -L./lib  
   - Các thư viện, bắt đầu với "-l", là viết tắt của lib; ví dụ: -lm tương ứng với libm, thư viện toán học, có sẵn trong hệ thống  
+ **gcc :** Compiler cho C source  
+ **g++ :** Compiler cho C++ source

##### Make file

1. Thiết kế chương trình

**Những vấn đề khi biên dịch**

Một chương trình đơn giản => chỉ có một vài file

Một chương trình “không đơn giản”:  
     - Nhiều dòng lệnh  
     - Nhiều module  
     - Nhiều người tham gia viết

**Vấn đề xảy ra:**

* Khó quản lý một file lớn (cả người và máy)
* Mỗi thay đổi cần thời gian biên dịch lâu
* Nhiều người lập trình không thể thay đổi cùng một file đồng thời
* Chương trình được phân ra thành nhiều module

**Giải pháp**: chia project ra thành nhiều file

**Mục tiêu:**

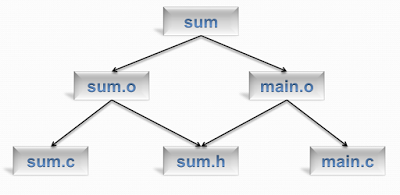
* Chia thành các module một cách đúng đắn
* Thời gian biên dịch ngắn nếu có sự thay đổi
* Dễ dàng bảo trì cấu trúc project và sự phụ thuộc

1. Makefile la gì
2. Định nghĩa

Makefile là một file dạng **script** chứa các thông tin:  
    - Cấu trúc project (file, sự phụ thuộc)  
    - Các lệnh để tạo file  
Lệnh **make** sẽ đọc nội dung Makefile, hiểu kiến trúc của project và thực thi các lệnh

1. Cấu trúc project

Cấu trúc và sự phụ thuộc của project có thể được biểu diễn bằng một DAG (Directed Acyclic Graph)  
Thí dụ:  
    - Chương trình chứa 3 file: **main.c, sum.c, sum.h**  
    - File **sum.h** được dùng bởi cả 2 file**main.c** và **sum.c**  
    - File thực thi là **sum**

[](http://3.bp.blogspot.com/-W_uhFqRcMyM/VYrCjQSycyI/AAAAAAAAYWc/HsJ_1Eqd1Ws/s1600/archi.png)

//sum.h

#ifndef SUM\_H\_

#define SUM\_H\_

#include <stdio.h>

int sum(int a, int b);

#endif /\* SUM\_H\_ \*/

//sum.c

#include "sum.h"

int sum(int a, int b){

return (a+b);

}

//main.c

#include <stdio.h>

#include "sum.h"

int main(int argc, char \*\*argv){

int x;

x= sum(1, 2);

printf("x = %d \n", x);

return 1;

}

Makefile:

sum: main.o sum.o

gcc -o sum main.o sum.o

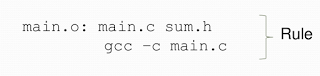
main.o: main.c sum.h

gcc -c main.c

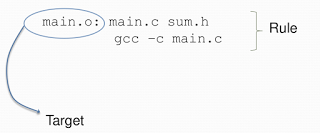
sum.o: sum.c sum.h

gcc -c sum.c

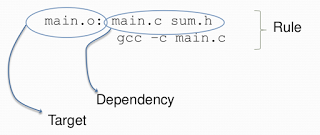
1. Phân tích cấu trúc Makefile:

[](http://4.bp.blogspot.com/-2TCd0KKCOWw/VYrDPOVsBEI/AAAAAAAAYXA/VNNhALfPirI/s1600/Screenshot.png)

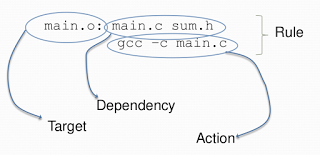
Rule: Có thể có nhiều Rule, trong ví dụ trên có 3 rule

[](http://3.bp.blogspot.com/-GD3yyfgdPYg/VYrDMDb3QyI/AAAAAAAAYWk/HSqisOGwPps/s1600/Screenshot-1.png)

Target

[](http://1.bp.blogspot.com/-cI1yS-UQBb8/VYrDNs8p26I/AAAAAAAAYWs/-r-4NlFncD4/s1600/Screenshot-2.png)

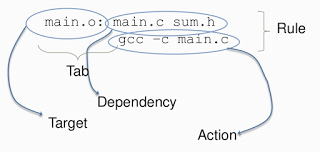
Dependency: Cần thiết để tạo ra Target

[](http://3.bp.blogspot.com/-SED2gxBKGqg/VYrDN-f_yTI/AAAAAAAAYWw/nR3v7mQnMow/s1600/Screenshot-3.png)

Action: Câu lệnh compile để tạo ra Target từ Dependency,

* option là **"-c"** để tạo ra file object từ file source code,
* option là **"-o"** để tạo ra file chương trình nhị phân từ file object

Mỗi Rule có thể không có hoặc có nhiều Action

[](http://1.bp.blogspot.com/-L0YIDEzwRWI/VYrDOh-y_cI/AAAAAAAAYW4/tYsgtf1fCBc/s1600/Screenshot-4.png)

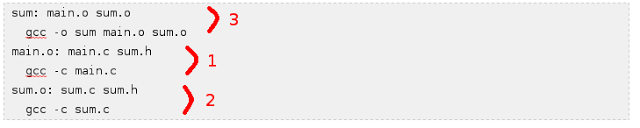
Action được thụt lùi vào một Tab (bằng 4 lần blank space) so với Target

**Thứ tự thực hiện:**

Khi bạn thực hiện lệnh make, chương trình make sẽ nhảy đến target đầu tiên là sum với mục đích để tạo ra nó, để làm được điều đó make đi kiểm tra lần lượt (từ trái qua phải: main.o -> sum.o) xem các dependency của sum đã tồn tại chưa. Denpendency đầu tiên là main.o chưa có, cần phải tìm rule nào đó mà ở đó main.o đóng vai trò là target, make tìm ra rule thứ 2 và nó nhảy đến thực hiện rule thứ 2 đề tạo ra main.o (lưu ý khi nó chạy rule 2 thì cũng giống y như khi chạy rule đầu tiên, có thể coi như là đệ quy). Sau khi tạo ra main.o, make trở về rule 1 để tiến hành kiểm tra tiếp xem dependency thứ hai là sum.o đã tồn tại chưa, sum.o chưa có vì thế make tiến hành các bước tương tự như đối với main.o. Sau khi tất cả các dependency được tạo ra, make mới có thể tạo ra file chạy cuối cùng là sum.

Vậy make thực hiện theo nguyên tắc / thứ tự như sau:

* Tạo ra các file object trước (**main.o,** **sum.o)**
* Tạo ra chương trình nhị phân cuối cùng từ các file object đã được tạo ra trước đó (**sum**)

[](http://2.bp.blogspot.com/-96LSR8TEPQk/VYrPpQzfPrI/AAAAAAAAYXk/ts5DD58oQEk/s1600/Screenshot-7.png)

**Compile & Execute**:

"**cd**" vào thư mục chứa project và thực hiện lệnh **make**, chương trình **sum** sẽ được tạo ra cùng với 2 file object là **sum.o** và **main.o** , chạy thử chương trình:

[ninhld@localhost ~]$

[ninhld@localhost ~]$ cd /home/ninhld/Github/eslinuxprogramming/Makefile

[ninhld@localhost Makefile]$

[ninhld@localhost Makefile]$ **make**

**gcc -c main.c**

**gcc -c sum.c**

**gcc -o sum main.o sum.o**

[ninhld@localhost Makefile]$

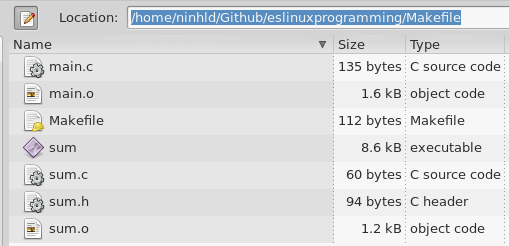
[ninhld@localhost Makefile]$

[ninhld@localhost Makefile]$ **./sum**

x = 3

[ninhld@localhost Makefile]$

[ninhld@localhost Makefile]$

[](http://1.bp.blogspot.com/-6egdNPATU-Q/VYrIX4hi7aI/AAAAAAAAYXU/AQ9DJMqX8Gg/s1600/Screenshot-6.png)

1. Nguyên lý biên dịch lại của Makefile

Việc compile lại project dựa vào hai yếu tố:

* Thời gian chỉnh sửa (date modified)
* Cây phụ thuộc trong Makefile

Có nghĩa là một khi Dependency thay đổi thì Target tương ứng cũng phải được compile lại.

Theo Makefile như trong ví dụ trên:

Nếu chỉnh sủa **sum.h** thì **main.o** và **sum.o** phải được tạo lại, mặt khác **main.o & sum.o** lại là dependency của **sum** nên **sum** sẽ được tạo lại

[ninhld@localhost Makefile]$ make

gcc -c main.c

gcc -c sum.c

gcc -o sum main.o sum.o

Nếu chỉnh sửa **sum.c** thì **sum.o** được tạo lại, đương nhiên **sum** phụ thuộc **sum.o** nên**sum** sẽ được tạo lại

[ninhld@localhost Makefile]$ make

gcc -c sum.c

gcc -o sum main.o sum.o

Khi viết Makefile tránh gộp tất cả lại làm một như dưới đây, vì khi đó nếu một trong các file source code thay đổi thì cũng phải compile lại tất cả các file khác, điều đó làm mất nhiều thời gian:

Makefile

sum: main.c sum.c sum.h

${CC} -o sum main.c sum.c sum.h

1. Khuôn dạng đầy đủ của Makefile

sum.h (giữ nguyên)

sum.c (giữ nguyên)

main.c (thêm #ifdef ... #endif)

#include <stdio.h>

#include "sum.h"

int main(int argc, char \*\*argv){

int x;

x= sum(1, 2);

#ifdef DEBUG

printf("x = %d \n", x);

#endif

return 1;

}

Makefile

.**PHONY**: all, install, clean

TARGET=sum

HDRS+= sum.h

CSRCS+= main.c sum.c

CPPSRCS+=

OBJSDIR=./build

OBJS:= $(patsubst %.cpp, $(OBJSDIR)/%.o, $(CPPSRCS))

OBJS+= $(patsubst %.c, $(OBJSDIR)/%.o, $(CSRCS))

CFLAGS += -I./include -DDEBUG -Wall -g

LDFLAGS += -L./lib -lm

CC:= gcc

CXX:= g++

**all**: ${TARGET}

${TARGET} : $(OBJS)

@echo " [LINK] $@"

@mkdir -p $(shell dirname $@)

@$(CXX) $(OBJS) -o $@ $(LDFLAGS)

$(OBJSDIR)/%.o: %.c $(HDRS)

@echo " [CC] $@"

@mkdir -p $(shell dirname $@)

@$(CC) -c $< -o $@ $(CFLAGS)

$(OBJSDIR)/%.o: %.cpp $(HDRS)

@echo " [CXX] $@"

@mkdir -p $(shell dirname $@)

@$(CXX) -c $< -o $@ $(CFLAGS)

**install**:

cp -rf ${TARGET} /usr/local/bin

**clean**:

rm -rf ${OBJSDIR}/\*.o

rm -rf ${TARGET}

1. all, install, clean, .PHONY

+ **all**  
Là target mặc định, khi bạn thực hiện lệnh **make** thì chương trình make sẽ tiến hành compile để tạo ra chương trình cuối cùng tương ứng với target này, trong ví dụ này là ${TARGET} tương ứng với sum; hai lệnh dưới đây là tương đương

make

make all

+ **install, clean**  
Các target do người dùng tự định nghĩa, tương ứng với các câu lệnh:

make install #thực hiện action cp

make clean #thực hiện action rm

+**.PHONY target**  
Target này có tác dụng chống xung đột cho chương trình **make** trong trường hợp trong thư mục source code có một file source nào đó trùng tên với các target trong Makefile, ví dụ all.c, install.c hay clean.c. Tốt nhất trong Makefile luôn dùng từ khóa này.

1. Phép gán: TARGET, CC, CXX, ...

Các từ in hoa như trên biểu diễn cho phép gán, có thể hiểu nó tương tự như #define trong C, nó giúp cho chương trình sáng sủa, dễ hiểu và rút ngắn Makefile đáng kể. Phép gán được thực hiện bằng các cách sau:

* "**=**" gán cố định
* "**:=**" tương tự như "="
* "**+=**" phép gán nối dài (append), ví dụ khi viết hai dòng

CSRCS+= main.c

CSRCS+= sum.c

=> CSRCS = main.c sum.c

Để lấy giá trị của phép gán thì dùng các cách sau:

${TARGET}

hoặc

$(CFLAGS)

* **HDRS / CSRCS /CPPSRCS**  
  Header , C source, C++ source file. Trong vd trên các file đều nằm trong folder gốc, nếu file nào nằm ở các thư mục con, bạn cũng cần chỉ đúng đường dẫn đễn file, vd:

HDRS+= ./src/sum.h

CSRCS+= main.c \

./src/sum.c

CPPSRCS+=

"\" là ký tự xuống dòng.

* **OBJSDIR / OBJS**  
  Thư mục chứa các object / các object
* **CFLAGS / LDFLASG**  
  Các flags chứa các options để pass vào cho compiler, xem lại [tại đây](http://eslinuxprogramming.blogspot.com/2015/04/cross-compiler.html) (mục 7).  
  Chú ý -DDEBUG tương ứng với define DEBUG để pass vào source code, ví dụ trong main.c ở trên
* **CC / CXX**  
  Compiler để compile, trong vd này sử dụng gcc/g++ để chạy trên máy tính chứ không phải board nhúng, khi [cross compiler](http://eslinuxprogramming.blogspot.com/2015/04/cross-compiler.html) thì bạn chỉ cần đặt đúng compiler tương ứng với platform cho board là được, vd:

CC=arm-linux-gcc

CXX=arm-linux-g++

1. Command Shell

Lệnh Shell được đưa vào trong cặp khóa ${ } như dưới đây:

OBJS:= $(patsubst %.cpp, $(OBJSDIR)/%.o, $(CPPSRCS))

mkdir -p $(shell dirname $@)

1. Makefile tương đương: $<, $@, $\*, $?, ...

|  |  |
| --- | --- |
| **Macro** | **Definition** |
| $< | Source code hiện tại để compile, tương ứng với %.c và %.cpp, ví dụ: main.c, sum.c |
| $@ | Target hiện tại tương ứng với $(OBJSDIR)/%.o, ví dụ: main.o, sum.o |
| $\* | Tương tự $< nhưng không có suffix, ví dụ: main, sum |
| $? | Danh sách dependency tương ứng với %.c $(HDRS), ví dụ: tương ứng với main.o là main.c và sum.h |

Được sử dụng trong đoạn Makefile chính:  
 ${TARGET} : $(OBJS)

@echo " [LINK] $@"

@mkdir -p $(shell dirname $@)

@$(CXX) $(OBJS) -o **$@** $(LDFLAGS)

$(OBJSDIR)/%.o: %.c $(HDRS)

@echo " [CC] $@"

@mkdir -p $(shell dirname $@)

@$(CC) -c $< -o $@ $(CFLAGS)

$(OBJSDIR)/%.o: %.cpp $(HDRS)

@echo " [CXX] $@"

@mkdir -p $(shell dirname $@)

@$(CXX) -c $< -o $@ $(CFLAGS)

Bạn đọc có thể trace các macro bằng các dòng lệnh sau:

@echo " [CC] $@ $< $\* $?"

@echo " [CXX] $@ $< $\* $?"

**Nguyên lý hoạt động:**  
Chỉ cần đoạn lệnh ngắn gọn trên là đủ để để compile tất cả các target được liệt kê trong OBJS thông qua ký tự đại diện "%"; compile main.o từ main.c, compile sum.o từ sum.c, tất nhiên có các dependency HDRS

1. Các ký tự khác

* @ ở đầu các Action trong Makefile. Loại bỏ các dòng trace mặc định của chương trình make khi đang compile
* -f :Nếu bạn đặt tên của script không phải mặc định là Makefile (ví dụ: MyMakefile) thì bạn cần thêm option -f vào

make -f MyMakefile

make -f MyMakefile install

make -f MyMakefile clean

...

1. Tạo Makefile theo kiểu module cho project lớn

Nếu project lớn có nhiều subfolder thì bạn nên chia thành nhiều phần cho dễ quản lý, cũng theo vd trên nhưng bây giờ giả sử sum.h và sum.c ở trong một subfolder gọi là src

[](http://3.bp.blogspot.com/-sASWN6bVxeI/VY65nSMvoEI/AAAAAAAAYZc/GjZTzBkrLsY/s1600/Screenshot-8.png)

Trong trường hợp này bạn thêm một makefile phụ là src.mk đặt trong thư mục src

src.mk

CSRCS+= ./src/sum.c

CPPSRCS+=

HDRS+= $(wildcard ./src/\*.h)

Makefile

.PHONY: all, install, clean

TARGET:=sum

**-include ./src/src.mk**

**CSRCS+= main.c**

OBJSDIR=./build

OBJS:= $(patsubst %.cpp, $(OBJSDIR)/%.o, $(CPPSRCS))

OBJS+= $(patsubst %.c, $(OBJSDIR)/%.o, $(CSRCS))

**INCDIR+= -I./src**

CFLAGS += -DDEBUG -Wall -g

LDFLAGS += -L./lib -lm

CC:= gcc

CXX:= g++

all: ${TARGET}

${TARGET} : $(OBJS)

@echo " [LINK] $@"

@mkdir -p $(shell dirname $@)

@$(CXX) $(OBJS) -o $@ $(LDFLAGS)

$(OBJSDIR)/%.o: %.c $(HDRS)

@echo " [CC] $@"

@mkdir -p $(shell dirname $@)

@$(CC) -c $< -o $@ $(CFLAGS) ${**INCDIR**}

$(OBJSDIR)/%.o: %.cpp $(HDRS)

@echo " [CXX] $@"

@mkdir -p $(shell dirname $@)

@$(CXX) -c $< -o $@ $(CFLAGS) ${**INCDIR**}

Cài đặt:

cp -rf ${TARGET} /usr/local/bin

Dọn dẹp:

rm -rf ${OBJSDIR}

rm -rf ${TARGET}

### Con trỏ hàm

##### Định nghĩa con trỏ hàm

Con trỏ hàm là một biến chứa địa chỉ hàm. Chúng ta có thể gọi hàm thông qua con trỏ này.

Liên quan tới cú pháp, có 2 loại con trỏ hàm

* Con trỏ tới hàm thông thường kiểu C hay hàm thành viên static C++
* Con trỏ tới hàm thành viên non-static C++

**Sự khác biệt cơ bản là mọi con trỏ tới hàm thành viên non-static cần tham số ẩn: con trỏ \*this tới một đối tượng của lớp. Luôn nhớ rằng 2 loại con trỏ này không tương thích với nhau**

##### Con trỏ hàm trên C

1. Cú pháp con trỏ hàm

Một con trỏ hàm có thể khởi tạo theo mẫu sau:

**<kiểu trả về> (\*<tên con trỏ>)(<danh sách đối số>);**

Ví dụ về con trỏ hàm nhận 3 tham số và trả về dữ liệu kiểu int.

int (\*pf)(float, char, char) = NULL;

1. Khởi tạo con trỏ hàm

Typedefs được dùng khá phổ biến cho các kiểu dữ liệu phức tạp như con trỏ hàm. Ví dụ:

typedef int (\*funcptr)();

Cách viết trên nghĩa là chúng ta đang định nghĩa kiểu con trỏ hàm **funcptr** trả về kiểu integer. typedef giúp khai báo dễ dàng hơn:

funcptr pfi;

typedef khá giống preprocessor nhưng tốt hơn, vì không có cách nào dùng preprocessor macro có thể định nghĩa con trỏ hàm

Ví dụ sau khai báo 3 biến con trỏ : pt2Function, pt2Member và pt2ConstMember. Hai dòng cuối là con trỏ tới thành viên non-static C++

// khai báo và khởi tạo = NULL

int (\*pf)(float, char, char) = NULL; // C

//dung typedef

typedef int (\*ptf)(float,char,char);

ptf pointerFunction = NULL;

//khai báo con trỏ hàm thành viên

int (obj::\*pt2Member)(float, char, char) = NULL;

int (obj::\*pt2ConstMember)(float, char, char) const = NULL;

1. Gán địa chỉ cho con trỏ hàm

Giống như con trỏ bình thường, bạn gán tham chiếu hàm (toán tử & tên hàm) cho con trỏ hàm. Khi dùng cho hàm thành viên C++, chúng ta phải dùng toán tử phạm vi kèm theo: tên\_class::tên\_thành\_viên

**Lưu ý: Mặc dù có thể bỏ qua toán tử địa chỉ trên hầu hết compiler, nhưng chúng ta nên viết rõ ràng để portable code dễ hơn**

// C

int DoIt (float a, char b, char c){ printf("DoIt\n"); return a+b+c; }

int DoMore(float a, char b, char c)const{ printf("DoMore\n"); return a-b+c; }

pt2Function = DoIt; // viết tắt

pt2Function = &DoMore; // cú pháp đầy đủ dùng toán tử địa chỉ

// C++

class TMyClass{

public:

int DoIt(float a, char b, char c){ cout << "TMyClass::DoIt"<< endl; return a+b+c;};

int DoMore(float a, char b, char c) const

{ cout << "TMyClass::DoMore" << endl; return a-b+c; };

};

pt2ConstMember = &TMyClass::DoMore; // cú pháp gán đúng dùng toán tử địa chỉ

pt2Member = &TMyClass::DoIt; // note: <pt2Member> vẫn có thể trỏ tới &DoMore hợp lệ

1. So sánh trên con trỏ hàm

Chúng ta vẫn dùng như cách thông thường

// C

if(pt2Function >0){ // check if initialized

if(pt2Function == &DoIt)

printf("Pointer points to DoIt\n"); }

else

printf("Pointer not initialized!!\n");

// C++

if(pt2ConstMember == &TMyClass::DoMore)

cout << "Pointer points to TMyClass::DoMore" << endl;

1. Gọi hàm từ con trỏ hàm

Gọi hàm trong C dùng con trỏ hàm với phép tham chiếu ngược tường minh, dùng toán tử nội dung \*. Cách khác, bạn có thể dùng con trỏ hàm thay vì tên hàm.

Trong C++, 2 toán tử \* và ->\* được dùng chung với nhau với đối tượng để gọi hàm thành viên non-static.

Nếu gọi hàm nằm trong một hàm thành viên thì chỉ cần dùng con trỏ \*this.

int result1 = pt2Function (12, 'a', 'b'); // C viết tắt

int result2 = (\*pt2Function) (12, 'a', 'b'); // C viết tường minh

TMyClass instance1;

int result3 = (instance1.\*pt2Member)(12, 'a', 'b'); // C++

int result4 = (\*this.\*pt2Member)(12, 'a', 'b'); // C++ gọi từ nội bộ class

TMyClass\* instance2 = new TMyClass;

int result4 = (instance2->\*pt2Member)(12, 'a', 'b'); // C++, đối tượng kiểu con trỏ

delete instance2;

1. Tham số con trỏ hàm

Nếu bạn đang triển khai một phương pháp sắp xếp, cho phép nơi gọi hàm chọn thứ tự sắp xếp cho dữ liệu, một số lập trình viên cần xếp tăng dần số khác lại thích giảm dần.

Bạn có thể đặt tham số là 1 flag nhận biết cách sắp xếp, nhưng vậy thì kém linh hoạt; hàm sắp chỉ cho phép cố định 1 kiểu so sánh (tăng và giảm)

Cách tốt hơn là cho người dùng tùy chọn bằng cách truyền một hàm vào hàm sắp xếp, hàm này nhận hai tham số và định nghĩa cách so sánh hai tham số này để tiến hành sắp xếp.

//C

int (\_\_stdcall \*CompareFunction)(const byte\*, const byte\*);

void DLLDIR \_\_stdcall Bubblesort(byte\* array,

int size,

int elem\_size,

int \*ComareFunction(const byte\*, const byte\*));

void DLLDIR \_\_stdcall Quicksort(byte\* array,

int size,

int elem\_size,

int \*ComareFunction(const byte\*, const byte\*));

1. Hàm trả về con trỏ hàm

Có một chút mẹo vặt, bạn phải thay đổi định nghĩa/khai báo toàn bộ con trỏ hàm.

Trong ví dụ sau đề ra 2 giải pháp.

1. Khai báo trực tiếp

//'Plus' và 'Minus'

float Plus(const float &f1, const float &f2);

float Minus(const float &f1, const float &f2);

//Hàm trả về con trỏ hàm GetPtr1 với tham số và định nghĩa của nó

float (\*GetPtr1(const char opCode))(float, float){

if(opCode == '+')

return &Plus;

else

return &Minus;

}

1. Dùng typedef định nghĩa con trỏ hàm

typedef float (\*pt2Func)(float, float);

// Khai báo hàm trả về con trỏ hàm

pt2Func GetPtr2(const char opCode){

if(opCode == '+')

return &Plus;

else

return &Minus;

}

// Execute example code

void Return\_A\_Function\_Pointer(){

cout << endl << "Executing 'Return\_A\_Function\_Pointer'" << endl;

// định nghịa con trỏ hàm và khởi tạo = NULL

float (\*pt2Function)(float, float) = NULL;

pt2Function=GetPtr1('+'); // get function pointer from function 'GetPtr1'

cout << (\*pt2Function)(2, 4) << endl; // call function using the pointer

pt2Function=GetPtr2('-'); // get function pointer from function 'GetPtr2'

cout << (\*pt2Function)(2, 4) << endl; // call function using the pointer

}

1. Mảng con trỏ hàm

Mảng các con trỏ hàm cho bạn khả năng chọn hàm qua chỉ số index. Cú pháp khá rắc rối, có 2 cách để dùng trong C,C++.

Khai báo Class TMyClass

class TMyClass{

public:

float plus(float a, float b){

cout<<"you re using plus 2 float num";

return a+b;

}

float minus(float a, float b){

cout<<"you re using minus 2 float num";

return a-b;

}

};

Và 2 hàm trên C

float plus(float a, float b){

cout<<"you re using plus 2 float num";

return a+b;

}

float minus(float a, float b){

cout<<"you re using minus 2 float num";

return a-b;

Dùng typedef định nghĩa con trỏ hàm, tạo mảng và khởi tạo về NULL

Mảng gồm 10 con trỏ hàm trả về kiểu int nhận tham số (float, char ,char)

// C

typedef int (\*pt2Function)(float, char, char);

pt2Member pfArray[10] = {NULL};

// C++ Sau khi triển khai lớp TMyClass

typedef float (TMyClass::\*pt2Member)(float, float);

pt2Member pfArray[10] = {NULL};

Hàm main kiểm tra

int \_tmain(int argc, \_TCHAR\* argv[])

{

pt2Member pfArray[10] = {NULL};

pfArray[0] = &TMyClass::minus;

pfArray[1] = &TMyClass::plus;

TMyClass instance;

cout << (instance.\*pfArray[0])(12, 6) << endl;

cout << (instance.\*pfArray[0])(12, 9) << endl;

cout << (instance.\*pfArray[1])(34, 1) << endl;

cout << (instance.\*pfArray[0])(89, 0) << endl;

return 0;

}

##### Đặc tính con trỏ hàm thành viên

Con trỏ hàm thành viên là một trong những cú pháp khó trong C++. Thậm chí đối với các lập trình viên kinh nghiệm cũng hay nhầm lẫn.

Bài viết này này sẽ giới thiệu về cơ chế của con trỏ hàm thành viên.

Trước hết, chúng ta xem một ví dụ code:

//mem\_fun1.cpp

#include <iostream>

class Foo{

public:

Foo(int i=0){ \_i=i;}

void f(){

std::cout<<"Foo::f()"<<std::endl;

}

private:

int \_i;

};

int main(){

Foo \*p=0; //con trỏ p = NULL

p->f();

}

**Output:**

**Foo::f()**

Tại sao con trỏ NULL lại có thể gọi tới hàm thành viên? Dường như trình biên dịch không quan tâm tới giá trị của con trỏ, chỉ quan tâm tới nó kiểu gì.

Câu hỏi này sẽ để dành phần sau giải đáp. Con trỏ hàm hoạt động nhờ cơ chế lately binding, địa chỉ của hàm được xác định trong thời gian biên dịch.

1. Cú pháp C++:

**Return\_Type (Class\_Name::\* pointer\_name) (Argument\_List);**

* Return\_Type: Kiểu trả về của con trỏ hàm.
* Class\_Name: Tên lớp mà hàm thành viên thuộc về.
* Argument\_List: danh sách đối số tương ứng với hàm thành viên.
* pointer\_name: tên con trỏ hàm.

Ví dụ ta có lớp Foo với hàm thành viên f() :

int Foo::f(string);

Chúng ta có thể khai báo con trỏ hàm tên là “fptr” có tham số string và kiểu trả về là int:

int (Foo::\*fptr) (string);

Cú pháp gán địa chỉ hàm cho con trỏ:

fptr= &Foo::f;

Chúng ta có thể vừa khai báo và khởi tạo cho con trỏ hàm trong 1 dòng lệnh:

int (Foo::\*fptr) (string) = &Foo::f;

Để gọi hàm thành viên qua con trỏ, chúng ta dùng toán tử thành viên (.\*) hay (->\*) trên đối tượng:

#include <iostream>

#include <string>

using std::string;

class Foo{

public:

int f(string str){

std::cout<<"Foo::f()"<<std::endl;

return 1;

}

};

int main(int argc, char\* argv[]){

int (Foo::\*fptr) (string) = &Foo::f;

Foo obj;

(obj.\*fptr)("str");//gọi: Foo::f() qua đối tượng trên stack

Foo\* p=&obj;

(p->\*fptr)("str");//gọi: Foo::f() qua con trỏ

}

Lưu ý: “.\*fptr” bind con trỏ hàm fptr tới đối tượng “obj”, ngược lại “->\*fptr” bind con trỏ hàm tới đối tượng trỏ bởi con trỏ “p”

1. Con trỏ hàm thành viên static

Con trỏ hàm thành viên không giữ địa chỉ chính xác giống như con trỏ thường. Đây chỉ là dạng địa chỉ tương đối

Xem ví dụ sau: chúng ta dùng hàm thành viên static trong lớp Foo

#include <iostream>

#include <string>

using std::string;

class Foo{

public:

static int f(string str){

std::cout<<"Foo::f()"<<std::endl;

return 1;

}

};

int main(int argc, char\* argv[]){

//int (Foo::\*fptr) (string) = &Foo::f;//lỗi

int (\*fptr) (string) = &Foo::f;//đúng

(\*fptr)("str");//gọi Foo::f()

}

Hàm static không được xem là thành viên lớp nên không có con trỏ \*this, nó giống như là hàm global hơn nhưng nằm trong phạm vi của lớp Foo.

Do đó chúng ta phải dùng cú pháp con trỏ hàm bình thường cho trường hợp này.

Nếu gán con trỏ hàm thành viên với hàm static sẽ có lỗi: " cannot convert 'int (\*)(std::string)' to 'int (Foo::\*)(std::string)".

1. Chuyển kiểu con trỏ hàm thành viên
2. Trường hợp non-virtual

Con trỏ hàm thành viên non-static không thể chuyển kiểu sang con trỏ thông thường. (Trừ khi quá cần thiết, thì có thể dùng kỹ thuật assembly)

Xem ví dụ về chuyển kiểu giữa các con trỏ hàm với nhau:

//memfun4.cpp

#include <iostream>

class Foo{

public:

int f(char\* c=0){

std::cout<<"Foo::f()"<<std::endl;

return 1;

}

};

class Bar{

public:

void b(int i=0){

std::cout<<"Bar::b()"<<std::endl;

}

};

class FooDerived:public Foo{

public:

int f(char\* c=0){

std::cout<<"FooDerived::f()"<<std::endl;

return 1;

}

};

int main(int argc, char\* argv[]){

typedef int (Foo::\*FPTR) (char\*);

typedef void (Bar::\*BPTR) (int);

typedef int (FooDerived::\*FDPTR) (char\*);

FPTR fptr = &Foo::f;

BPTR bptr = &Bar::b;

FDPTR fdptr = &FooDerived::f;

//Bptr = static\_cast<void (Bar::\*) (int)> (fptr); //lỗi

fdptr = static\_cast<int (Foo::\*) (char\*)> (fptr); //OK: contravariance

Bar obj;

( obj.\*(BPTR) fptr )(1);//gọi: Foo::f()

}

**Output:**

**Foo::f()**

Ví dụ trên dùng kỹ thuật "typedef" để định nghĩa kiểu con trỏ hàm. Nó giúp việc định nghĩa và khai báo biến con trỏ dễ dàng cho lập trình viên , con trỏ hàm "fptr" có kiểu là:

int (Foo::\*) (char\*);

VIệc ép kiểu con trỏ hàm thành viên của Foo sang Bar là không hợp lệ vì không tương thích kiểu:

BPTR Bptr = static\_cast<void (Bar::\*) (int)> (fptr);//error

Con trỏ hàm thành viên non-static và non-virtual được xem là strong type, không thể cast giữa các đối tượng không tương thích kiểu.

Trong ví dụ sau, dù chúng ta muốn gọi Bar::b(), nhưng Foo::f() mới thực sự được gọi.

FPTR fptr;

Bar obj;

(obj.\*(BPTR)fptr)(1);//goi: Foo::f()

Chúng ta sẽ tìm hiểu nguyên nhân trong phần con trỏ \*this.

1. Trường hợp virtual

Chúng ta sẽ định nghĩa lại hàm thành viên thành virtual cho cả 3 class:

#include <iostream>

class Foo{

public:

virtual int f(char\* c=0){

std::cout<<"Foo::f()"<<std::endl;

return 1;

}

};

class Bar{

public:

virtual void b(int i=0){

std::cout<<"Bar::b()"<<std::endl;

}

};

class FooDerived:public Foo{

public:

int f(char\* c=0){

std::cout<<"FooDerived::f()"<<std::endl;

return 1;

}

};

int main(int argc, char\* argv[]){

typedef int (Foo::\*FPTR) (char\*);

typedef void (Bar::\*BPTR) (int);

FPTR fptr = &Foo::f;

BPTR bptr = &Bar::b;

FooDerived objDer;

(objDer.\*fptr)(0);//gọi: FooDerived::f(), not Foo::f()

Bar obj;

(obj.\*(BPTR)fptr)(1);//gọi: Bar::b() , not Foo::f()

}

**Output:**

**FooDerived::f()**

**Bar::b()**

Khi hàm thành viên là virtual, con trỏ hàm sẽ có tính đa hình. Con trỏ tới hàm thành viên virtual có thể truyền một cách an toàn giữa các vùng địa chỉ khác nhau miễn là bố cục đối tượng có tính tương thích (theo Bjarne Stroustrup , "The C++ Programming Language)

Sở dĩ có thể gọi tới đúng hàm thành viên virtual của đối tượng là do trình biên dịch sẽ sinh ra và tham chiếu bảng hàm thành viên ảo cùng địa chỉ tương ứng cho mỗi đối tượng.

1. Mảng con trỏ hàm và ứng dụng

Ứng dụng quan trọng của con trỏ hàm thành viên là các sự kiện phản hồi tới inputs. Ví dụ về Class Printer và mảng con trỏ pmf.

#include <stdio.h>

#include <string>

#include <iostream>

class Printer{//Đối tượng máy in

public:

void Copy(char \* buff, const char \* source){//sao chép file

strcpy(buff, source);

}

void Append(char \* buff, const char \* source){//Nối file

strcat(buff, source);

}

};

enum OPTIONS { COPY, APPEND };//Hai chế độ chọn từ menu máy in.

typedef void(Printer::\*PTR) (char\*, const char\*);//con trỏ hàm thành viên

void working(

OPTIONS option, //chế độ làm việc

Printer\* machine, //con trỏ \*this tới đối tượng máy in thao tác

char\* buff,

const char\* infostr){

PTR pmf[2]= {

&Printer::Copy,

&Printer::Append

}; //mảng con trỏ hàm tương ứng với từng tác vụ của máy in

switch (option){

case COPY:

(machine->\*pmf[COPY])(buff, infostr);

break;

case APPEND:

(machine->\*pmf[APPEND])(buff, infostr);

break;

}

}

int main(){

OPTIONS option;

Printer machine;

char buff[40];//target

working(COPY, &machine, buff, "Strings ");

working(APPEND, &machine, buff, "are concatenated! ");

std::cout<<buff<<std::endl;

}

**Output:**

**Strings are concatenated!**

Trong đoạn code trên, phương thức **working** có trách nhiệm xử lí công việc máy in gồm:

* Chế độ in
* Máy in nào cần làm việc
* Input
* Output

Mảng con trỏ hàm thành viên tương ứng với từng tác vụ trong máy in. Con trỏ hàm cũng được áp dụng trong thư viện STL mem\_fun().

1. Con trỏ hàm với \*this

Trở lại với ví dụ đầu tiên, một con trỏ NULL lại có thể gọi tới hàm thành viên đối tượng? Đối với hàm thành viên non-virtual khi được gọi, trình biên dịch sẽ sinh ra code giống như sau:

Foo\* const this = p;

void Foo::f( Foo \*const this){

std::cout<<"Foo::f()"<<std::endl;

}

Vì thế hàm Foo::f có thể được gọi mà không bận tâm giá trị của “p” là gì, nó giống như hàm global có tham số con trỏ đối tượng “p” truyền vào như là tham số con trỏ “\*this” vậy

Con trỏ \*this không được tham chiếu trong hàm trong trường hợp này và do đó trình biên dịch vẫn đi qua. Nếu chúng ta muốn xuất ra dữ liệu của biến thành viên data\_i, thì trình biên dịch sẽ báo lỗi khi tham chiếu tới con trỏ \*this.

Đối với hàm thành viên virtual, phiên bản đúng của hàm thành viên sẽ được tìm trong bảng ảo tương ứng với con trỏ \*this.

Đó là lý do tại sao con trỏ hàm non-virtual, virtual, static được triển khai khác nhau.

##### Callback Functions

Một cách dùng khác của con trỏ hàm là thiết lập một “listerner” hay còn gọi là hàm “callback” được gọi tới khi một sự kiện xảy ra.

Cơ chế là khi một hàm được gọi, nó phát ra một cảnh báo và sau đó hàm callback được gọi tới.

Khi lập trình giao diện (GUI), người dùng thường xuyên tương tác với giao diện (click chuột, nhấn phím…), chương trình phải xử lí những “sự kiện” như vậy bằng cách dùng Callback function. (hàm xử lí sự kiện)

Hình dung bạn đang dùng thư viện GUI, có hàm create\_button() nhận 2 tham số vị trí để button xuất hiện, nội dung button, cuối cùng là hàm callback khi button được nhấn:

void create\_button( int x, int y, const char \*text, function callback\_func );

Giả sử rằng C,C++ có một kiểu con trỏ hàm chung gọi là “function”.

Khi button bị nhấn, callback\_func() được gọi tới. Hành vi hàm callback tùy thuộc vào button.

1. Trên C

Trở về ví dụ sắp xếp, với đề xuất dùng con trỏ hàm để triển khai phương thức so sánh tổng quát

Chúng ta có hàm qsort():

void qsort(void \***array**, size\_t **nmemb**, size\_t **size**, int(\*compar)(const void \*, const void \*) );

Chú ý cách dùng void \* áp dụng cho mọi loại dữ liệu (trong C++, chúng ta có thể dùng template trường hợp này, nhưng C++ cũng cho phép dùng con trỏ void\*) vì con trỏ void\* có thể trỏ tới mọi thứ.

Vì chúng ta không biết rõ kích thước từng phần tử trong mảng void\*, chúng ta phải đưa tham số phần tử nmemb vào qsort(), và kích thước **size** của phần tử. \***array** trỏ tới mảng cần sắp xếp.  
Cuối cùng, quan trọng nhất là đối số so sánh của qsort(). Đó là con trỏ hàm nhận 2 tham số void\* và trả về giá trị integer. Nó qui định cách sắp xếp các phần tử của mảng mà không cần phải viết định nghĩa bổ sung cho giải thuật sắp xếp.

Code:

// Dùng callback trong C triển khai quick sort

#include <stdlib.h> // due to: qsort

#include <time.h> // randomize

#include <stdio.h> // printf

// triển khai hàm so sánh nhận 2 tham số con trỏ void const

int CmpFunc(const void\* \_a, const void\* \_b)

{

// ép kiểu

const float\* a = (const float\*) \_a;

const float\* b = (const float\*) \_b;

if(\*a > \*b) return 1; // số thứ nhất lớn hơn trả về 1

else

if(\*a == \*b) return 0; // bằng nhau trả về 0

else return -1; // số thứ 2 lớn hơn thì trả về -1

}

// ví dụ dùng qsort()

void QSortExample()

{

float field[100];

srand(NULL); // khởi tạo bộ sinh số ngẫu nhiên

for(int c=0;c<100;c++) // khởi tạo dữ liệu cho mảng

field[c]=rand();

qsort((void\*) field,/\*số phần tử\*/ 100,/\*kích thước phân tử\*/sizeof(field[0]),/\*hàm so sánh\*/ CmpFunc);

// xuất 10 phần tử đầu tiên

printf("10 phần tử đầu tiên ...\n");

for(int c=0;c<10;c++)

printf("element #%d contains %.0f\n", c+1, field[c]);

printf("\n");

}

1. Trên C++

Địa chỉ hàm thành viên static là cố định khi biên dịch, ngược lại địa chỉ hàm thành viên non-static của mỗi đối tượng sinh ra lại khác nhau.

Do có địa chỉ cố định nên cách triển khai callback cho hàm thành viên static tương tự như trên C.

Để triển khai cho hàm non-static, ta bắt buộc phải truyền tham số địa chỉ đối tượng vào hàm gọi:

1. Để callback đến một hàm thành viên tùy ý, bạn phải viết 1 hàm thành viên tĩnh kiểu wrapper.
2. Sau đó ép kiểu con trỏ tới đối tượng mà bạn muốn gọi thành viên sang void\* và truyền nó vào wrapper như đối số bổ sung thêm hay qua biến toàn cục. (Tất nhiên là bạn phải truyền đối số gọi hàm cho hàm thành viên)
3. Wrapper sẽ ép kiểu con trỏ void\* sang con trỏ lớp tương ứng và gọi hàm thành viên.

Cách 1: Khai báo tham số là con trỏ tới đối tượng vào hàm callback

Lưu ý: cách này hữu ích nếu bạn thiết kế callback cho chính lớp.

// Callback tới hàm thành viên dùng đối số phụ

// Hàm 'DoItA' ngầm định callback tới hàm thành viên 'Display'.

// chúng ta triển khai hàm warrper “ Wrapper\_To\_Call\_Display “.

#include <iostream.h> // dùng cout

class TClassA {

public:

void Display(const char\* text) { cout << text << endl; };

static void Wrapper\_To\_Call\_Display(void\* pt2Object, char\* text);

/\* more of TClassA \*/

};

// static wrapper-function có thể callback tới Display()

void TClassA::Wrapper\_To\_Call\_Display(void\* pt2Object, char\* string) {

//ép kiểu

TClassA\* mySelf = (TClassA\*) pt2Object;

// gọi hàm thành viên

mySelf->Display(string);

}

// DoItA ngầm định callback tới con trỏ hàm, nó cũng có thể là member function

void DoItA(void\* pt2Object, void (\*pt2Function)(void\* pt2Object, char\* text)) {

/\* do something \*/

pt2Function(pt2Object, "hi, i'm calling back using a argument ;-)"); // make callback

}

// chạy code

void Callback\_Using\_Argument() {

TClassA objA;

DoItA((void\*) &objA, TClassA::Wrapper\_To\_Call\_Display);

}

Cách 2: Thay vì truyền tham số, ta khai báo biến toàn cục là con trỏ tới đối tượng lớp

#include <iostream.h> // cout

void\* pt2Object; // biến toàn cục trỏ tới đối tượng tùy ý

class TClassB {

public:

void Display(const char\* text) { cout << text << endl; };

static void Wrapper\_To\_Call\_Display(char\* text);

/\* more of TClassB \*/

};

// static wrapper-function callback tới Display()

void TClassB::Wrapper\_To\_Call\_Display(char\* string) {

// lưu ý: <pt2Object> phải trỏ tới đối tượng khởi tạo trước!

TClassB\* mySelf = (TClassB\*) pt2Object;

// gọi thành viên hàm

mySelf->Display(string);

}

// DoItB cũng có thể là member-function

void DoItB(void (\*pt2Function)(char\* text)) {

/\* do something \*/

pt2Function("hi, i'm calling back using a global ;-)"); // make callback

}

// chạy code

void Callback\_Using\_Global() {

TClassB objB;

pt2Object = (void\*) &objB;

//callback

DoItB(TClassB::Wrapper\_To\_Call\_Display);

}

1. Cú pháp thủ tục callback

Trình biên dịch mặc định thuộc tính \_\_cdecl cho thủ tục gọi hàm. Thủ tục gọi hàm báo cho trình biên dịch biết cách truyền đối số và cách sinh ra tên của hàm.

Ví dụ thủ tục \_\_stdcall, \_\_fastcall

Thủ tục gọi hàm thuộc nằm phía sau khai báo hàm: hàm và con trỏ hàm với thủ tục gọi khác nhau sẽ không tương thích.

// định nghĩa thủ tục gọi hàm

void \_\_cdecl DoIt(float a, char b, char c); // Borland and Microsoft

void DoIt(float a, char b, char c) \_\_attribute\_\_((cdecl)); // GNU GCC

Với trình biên dịch Borland và Microsoft, bạn khai báo thủ tục gọi hàm giữa kiểu trả về và tên hàm.

Trên môi trường window, bạn phải đặt thủ tục \_\_stdcall trước trên con trỏ hàm để có thể dùng được với các hàm Win32API. Hoặc có thể dùng macro CALLBACK định nghĩa trong windef.h

#define CALLBACK \_\_stdcall

Với GNU GCC, bạn dùng từ khóa \_\_attribute\_\_ phía sau định nghĩa/khai báo hàm kèm theo tên thủ tục trong ngoặc kép

1. Dùng tính đa hình và hàm ảo thay vì con trỏ hàm (C++)

Khi triển khai so sánh trên đối tượng, chúng ta có thể dùng hảm ảo thay vì con trỏ hàm. Ví dụ, khai báo hàm sắp xếp cpp\_qsort() nhận tham số con trỏ tới lớp cung cấp hàm ảo so sánh được gọi:

class Sorter{

public:

virtual int compare (const void \*first, const void \*second);

};

// cpp\_qsort, a qsort using C++ features like virtual functions

void cpp\_qsort(void \*base, size\_t nmemb, size\_t size, int (\*compar)( const void \*, const void \*));

Trong hàm cpp\_qsort(), mỗi lần so sánh 2 đối tượng, thì compar->compare sẽ được gọi.

Nếu có subclass override hàm ảo so sánh, hàm sắp xếp sẽ nhận hành vi mới. Ví dụ:

class AscendSorter : public Sorter{

virtual int compare (const void\*, const void\*) {

int first = \*(int\*)first\_arg;

int second = \*(int\*)second\_arg;

if ( first < second ) {

return -1;

} else if ( first == second ) {

return 0;

}

else {

return 1;

}

}

};

Sau đó truyền con trỏ tới đối tượng AscendSorter vào cpp\_qsort() để sắp xếp tăng dần.

1. Bạn thực sự không cần dùng con trỏ hàm?

Hàm ảo được triển khai khi dùng với con trỏ hàm. Trình biên dịch tự nhận biết và làm nốt phần việc còn lại.

Áp dụng tính đa hình với con trỏ hàm là kỹ thuật phổ biến (như trong Java), nhưng sẽ tốn chi phí để tạo đối tượng so với việc chỉ truyền con trỏ hàm đơn thuần.

1. Ưu điểm con trỏ hàm

Con trỏ hàm cung cấp phương tiện qui định cách thức làm việc.

Bạn có thể viết một hàm và thư viện linh hoạt cho phép lập trình viên chọn hành vi để tương tác bằng cách truyền con trỏ hàm vào đối số.

Chúng ta vẫn áp dụng được cho đối tượng với hàm ảo

##### Callback trong phát triển ứng dụng

1. Nền tảng thiết kế

Cơ chế callback trong hướng đối tượng hỗ trợ trong cả thiết kế component và ứng dụng. Nhà thiết kế nên có phương pháp chuẩn để cung cấp dịch vụ callback. Linh hoạt trong số lượng và kiểu tham số truyền, giá trị trả về. Vì component được thiết kế cho ứng dụng không cần biết hay ra lệnh cho đối tượng “called back” bởi component.

Lập trình viên ứng dụng, được cung cấp hàng loạt component với cơ chế call back chuẩn và một số đối tượng của lớp với hàm thành viên tương thích với kiểu trả về hàm callback, chỉ cần coding cho việc kết nối chúng lại với nhau. Hoặc họ chỉ phải tùy chỉnh caller class hay dẫn xuất một lớp mới.

Để thiết kế cơ chế này chúng ta cần:

* Hướng đối tượng - ứng dụng xây dựng trên các đối tượng. Hầu hết chức năng trong C++ chứa trong hàm thành viên, vốn không thể gọi qua một con trỏ hàm bình thường. Hàm thành viên Non-static hoạt động phụ thuộc vào đối tượng, do đó call back trong OOP phải có thông tin đối tượng nào được gọi.
* Kiểu an toàn – là một tính năng và lợi ích cơ bản của C++ , bất kì cơ chế callback nào của C++ phải an toàn. Nghĩa là chúng ta phải đảm bảo tất cả đối tượng được dùng đúng với interface, và luật này áp dụng chặt chẽ cho danh sách đối số, kiểu trả về và ép kiểu. Đó là cách tốt nhất để đảm bảo trình biên dịch làm việc tốt.
* Không móc nối : Mục đích cơ bản của callback là cho phép các components hoạt động độc lập có thể kết nối với nhau. Nếu cơ chế vô hình tạo ra sự lệ thuộc giữa caller và callee, nhiệm vụ coi như thất bại
* Xâm lấn kiểu dữ liệu - Một số cơ chế call back yêu cầu điều chỉnh , hay dẫn xuất, kiểu caller hay callee. Thực tế là đối tượng được kết nối tới đối tượng khác trong một ứng dụng thường không đả động gì tới kiểu dữ liệu của nó cả. Như ta thấy bên dưới, sự xâm lấn kiểu dữ liệu có thể làm giảm tính linh hoạt và tăng độ phức tạp của code ứng dụng.
* Tính tổng quát: khác biệt chính giữa các trường hợp callback là kiểu dữ liệu liên quan. Cơ chế callback nên biểu diễn tham số kiểu template. Template đảm bảo tính nhất quán của interface và tên trong tất cả trường hợp, và cung cấp phương tiện để có hỗ trợ cần thiết về code bởi trình biên dịch.
* Tính linh hoạt: Kinh nghiệm cho thấy hệ thống callback yêu cầu sự chính xác kiểu dữ liệu giữa hàm callback và hàm được gọi sẽ gây khó khăn khi dùng. Ví dụ khi bạn làm việc với callback truyền con trỏ đối tượng dẫn xuất derived\*, bạn muốn kết nối tới hàm được gọi nhận đối tượng là base\*.

1. Mô hình hiện tại
2. Con trỏ hàm

Chúng ta xét hàm ANSI C's qsort(). Để xây dựng 1 hàm đứng riêng lẻ, xu hướng thường dùng con trỏ tĩnh hay con trỏ toàn cục để đánh dấu đối tượng mục tiêu, hay dùng hàm callback và nhận thêm tham số ( hay dùng đối tượng hàm).

Phương pháp con trỏ static/global không đáp ứng yêu cầu vì tốn tại mối quan hệ callback qua các lần gọi. Ví dụ: tôi muốn kết nối Button tới đối tượng X này và Button khác thì kết nối tới X khác, trong suốt quá trình chạy ứng dụng

Phương pháp tham số phụ, nếu đáp ứng an toàn về kiểu, không móc nối giữa kiểu dữ liệu hàm gọi và hàm được gọi.

qsort() đạt được yêu cầu về an toàn kiểu dữ liệu. Ví dụ để lơ qua kiểu dữ liệu, nó nhận tham số kiểu (void\*). Không có gì ngăn cản chúng ta gọi qsort() cho mảng đối tượng apples nhưng lại truyền con trỏ hàm so sánh kiểu oranges!

Bạn sẽ thường xuyên thấy cơ chế này trong hàm apply của collection. Mục đích của hàm apply là cho phép lập trình viên truyền callback tới một collection và để nó gọi tới apply cho mỗi item trong collection. Nó thường có dạng này:

void apply(void (\*func)(T &theItem,void \*extraStuff),void \*theStuff);

Nhiệm vụ của bạn là triển khai hàm cho con trỏ \*func và truyền đúng tên hàm vào apply()

1. Single Rooted Hierarchy

Những cơ chế không an toàn kiểu này thường liên quan tới một số lớp cơ sở như Object hay EventHandler, và tận dụng việc ép kiểu từ con trỏ thành viên dẫn xuất tới con trỏ thành viên lớp cơ sở.

Kinh nghiệm cho thấy hệ thống single-rooted không thể làm việc nếu components đến từ nhiều nguồn

1. Biểu diễn tham số cho hàm gọi

Nhà thiết kế component phải biểu diễn tham số cho component dựa trên kiểu của hàm được gọi. Khai báo tham số không phù hợp trong nhiều hoàn cảnh , callback nằm trong số đó. Xem ví dụ sau:

class Button{

public:

virtual void click();

//...

};

template <class T>

class ButtonThatCallsBack:public class Button{

public:

ButtonThatCallsBack(T \*who,void (T::\*func)(void)): callee(who),callback(func){}

void click() {

(callee->\*callback)();

}

private:

T \*callee;

void (T::\*callback)(void);

};

class CDPlayer{

public:

void play();

//...

};

//Kết nối CDPlayer với Button

CDPlayer cd;

ButtonThatCallsBack<CDPlayer> button(&cd,&CDPlayer::play);

button.click(); //gọi cd.play()

ButtonThatCallsBack<CDPlayer> phải biết về CDPlayer và các phương thức trong interface liên quan tới nó. Sự kém linh hoạt xảy ra khi dữ liệu đối tượng được gọi trở thành dữ liệu nơi gọi. Tất cả code trong đối tượng ButtonThatCallsBack phải nhận ra được mối quan hệ với hàm được gọi, làm tăng tính liên kết trong hệ thống. Một ButtonThatCallsBack<X> khác với ButtonThatCallsBack<Y>.

Nếu component có nhiều mối quan hệ callback, nó không thể khai báo tham số hết được. Xem xét một Button muốn duy trì danh sách các “callees” để được thông báo theo sự kiện click. Vì kiểu dữ liệu của callee được đẩy vào kiểu của lớp Button, danh sách này phải vừa đồng nhất vừa không xác định rõ kiểu.

1. Cơ chế Callee Mix-In

Nhà thiết kế component có thể tạo ra một interface class để triển khai callback, và để lập trình viên ứng dụng có thể tự triển khai interface để kết nối lớp của họ tới component. Tôi gọi phương pháp này là “callee mix-in”.

Bên dưới là thiết kế của Button, cung cấp phương thức cảnh báo callback khi click chuột, khai báo một interface lồng bên trong là Notifiable với hàm thuần ảo notify().

Đối tượng sử dụng Button sẽ phải truyền hàm dựng của nó tới con trỏ đối tượng Notifiable \*callee, Button sẽ dùng cho cảnh báo sự kiện click:

class Button{

public:

class Notifiable{

public:

virtual void notify()=0;

};

Button(Notifiable \*who):callee(who){}

void click(){callee->notify();}

private:

Notifiable \*callee;

};

class CDPlayer{

public:

void play();

//...

};

CDPlayer không có đối tượng cảnh báo để truyền cho Button. Lập trình viên muốn có Button callback tới CDPlayer sẽ phải dẫn xuất lớp mới từ CDPlayer và Button::Notifiable, override hàm thuần ảo để có thể làm việc:

class MyCDPlayer:public CDPlayer, public Button::Notifiable{

public:

void notify() {play();}

};

Và sử dụng:

MyCDPlayer cd;

Button button(&cd);

button.click(); //gọi cd.play()

Cơ chế này an toàn, móc nối được Button và CDPlayer, tuy nhiên hầu như vô dụng trong thực tế.

Vấn đề với callee mix-in là sự xâm lấn kiểu dữ liệu. Nó ảnh hưởng bởi dữ liệu callee. Có 3 khuyết điểm:

* Đầu tiên là đa thừa kế, nếu callee là nhiều components, thì sẽ khó giải quyết do xung đột tên.
* Thứ hai, không thể dẫn xuất nếu nhà thiết kế ứng dụng lấy CDPlayer từ một API không thay đổi được (lưu ý khi lập trình thư viện: đây là vấn đề lớn trong cơ chế mix-in nói chung).
* Vấn đề thứ ba, xét phiên bản CDPlayer sau:

class CDPlayer{

public:

void play();

void stop();

//...

};

Để một Button vừa có thể gọi play() vừa gọi stop() là không thể. Cơ chế mix-in thất bại ở điểm này, nó chỉ hỗ trợ một tác vụ giữa caller/calle/member- function. MyCDPlayer chỉ có duy nhất 1 notify().

1. Callback dùng template functor
2. Cơ chế

Đối tượng hàm đơn giản là đối tượng có hành vi như con trỏ hàm, nó quá tải toán tử (), dùng cho việc gọi hàm. Thư viện “callback.h” triển khai một tập hợp template class Functor.

Các đối tượng Functor có thể lưu trữ dữ liệu callee cần thiết và cung cấp con trỏ hàm tới phương thức của callee.

Quan trọng nhất là nó hoạt động độc lập với callee. Components định nghĩa interface callback dùng lớp Functor

Hàm dựng nhận tham số Functor. Chúng ta sẽ dùng hàm teamplate makeFunctor(), nhận tham số là thông tin từ callee (Địa chỉ tới đối tượng callee và con trỏ hàm tới phương thức) và trả về đối tượng Functor đã khởi tạo.

Cơ chế này rất dễ sử dụng. Code :

#include <callback.h> //include the callback library header

#include <iostream.h>

class Button{

public:

Button(const Functor0 &uponClickDoThis):notify(uponClickDoThis)

{}

void click(){

notify(); //a call to operator()

}

private:

Functor0 notify; //note - held by value

};

//Some application stuff we'd like to connect to Button:

class CDPlayer{

public:

void play(){cout<<"Playing"<<endl;}

void stop(){cout<<"Stopped"<<endl;}

};

void wow()

{cout<<"Wow!"<<endl;}

void main() {

CDPlayer cd;

//makeFunctor nhận đối tượng và con trỏ hàm

Button playButton(makeFunctor(cd,&CDPlayer::play));

Button stopButton(makeFunctor(cd,&CDPlayer::stop));

//makeFunctor từ con trỏ hàm

Button wowButton(makeFunctor(&wow));

playButton.click(); // gọi cd.play()

stopButton.click(); // gọi cd.stop()

wowButton.click(); // gọi wow()

}

Component Button đã kết nối tới đối tượng và hàm của ứng dụng mà không cần biết gì về đối tượng kết nối tới, không cần thừa kế hay điều chỉnh đối tượng liên quan. Và kiểu dữ liệu an toàn.

Sơ đồ thiết kế Button là một interface callback không nhận tham số và trả về void. Nó chỉ chứa functor notify. Khi thới điểm tới, nó chỉ gọi functor notify(), quá tải () sẽ được gọi, giống như gọi con trỏ hàm.

Kết nối thứ gì đó tới component dùng callback khá đơn giản. Bạn chỉ cần khởi tạo Functor cho component bằng cách đặt makeFunctor() trong hàm dựng component. makeFunctor() được quá tải theo 2 phiên bản:

* Nhận con trỏ hàm: makeFunctor(&wow)
* Hay đối tượng và con trỏ hàm của nó: makeFunctor(cd, &CDPlayer::play)

Còn một vấn đề nữa, làm sao để callback service biết chính xác kiểu Functor nào mà bạn muốn tạo? Có 1 mẹo nhỏ là bạn truyền thêm một tham số hình nộm là con trỏ kiểu functor muốn tạo ra. Đơn giản đặt zero vào tham số này:

* makeFunctor((Functor0 \*)0, &wow)
* makeFunctor((Functor0 \*)0, cd, &CDPlayer::play)

Chúng ta sẽ dùng cú pháp này từ bây giờ.

Lớp Button ở trên chỉ cần hàm callback không có đối số và trả về void. Trường hợp components có thể nhận dữ liệu và có giá trị trả về thì sao?

Sự khác biệt duy nhất giữa các functor là khi khởi tạo, có bao nhiêu đối số, kiểu con trỏ hàm trả về. Vậy chúng ta chỉ cần triển khai một bộ các template Functor nhận 1, 2, 3,… tham số.

//Lớp Functor cung cấp trong Callback library:

Functor0 //Không template – Không tham số

Functor1<P1>

Functor2<P1,P2>

Functor3<P1,P2,P3>

Functor4<P1,P2,P3,P4>

Functor0wRet<RT>

Functor1wRet<P1,RT>

Functor2wRet<P1,P2,RT>

Functor3wRet<P1,P2,P3,RT>

Functor4wRet<P1,P2,P3,P4,RT>

Lý do phải triển khai nhiều vì

* class name là duy nhất không thể quá tải.
* Chỉ có thể quá tải toán tử () 1 lần mà thôi.
* Hậu tố “wRet” ám chỉ functor có trả về kiểu dữ liệu của con trỏ hàm.

Mỗi functor có một quá tải toán tử hàm () với dữ liệu trả về tương ứng. Ví dụ:

template <class P1>

class Functor1{

public:

void operator()(P1 p1)const;

//...

};

template <class P1,class P2,class RT>

class Functor2wRet{

public:

RT operator()(P1 p1,P2 p2)const;

//...

};

Những lớp functor này đủ khả năng đáp ứng callback cho thiết kế component, chúng được chuẩn hóa và hiệu quả cho dịch vụ callback, và cơ chế đơn giản để gọi hàm. Với template functor trong library, component chỉ cần chọn 1 trong số interface functor phù hợp.

Dưới đây là lớp DataEntryField dùng dịch vụ callback, truyền đối tượng hàm nhận tham số const String& và trả về kiểu bool:

#include <callback.h>

class DataEntryField{

public:

DataEntryField(const Functor1wRet<const String &,Boolean> &v):validate(v){}

void keyHit(const String & stringSoFar){

if(validate(stringSoFar))

// xử lí..

}

private:

Functor1wRet<const String &,Boolean> validate;

//functor validate gồm một quá tải Boolean operator()(const String &)

};

Ví dụ trên chỉ minh họa tính tổng quát của callback library.

Thư viện callback phải 100% an toàn cho trình biên dịch. Nếu bạn cố tạo functor từ thứ gì không tương thích, trình biên dịch sẽ báo lỗi. Tất cả hàm ảo không nên dùng.

Hệ thống khá linh hoạt nhờ teamplate hóa, nên kiểu dữ liệu giữa caller và callee ăn khớp khi biên dịch.

Khi làm việc với các con trỏ hàm. Kiểu trả về của hàm phải chuyển đổi ngầm sang kiểu trả về functor. Một functor có quá tải hàm kiểu void vẫn có thể built được từ hàm có giá trị trả về. Giá trị return xem như bỏ qua:

// Derived dẫn xuất public từ Base

void foo(Base &);

long bar(Derived &);

Functor1<Derived&> f1 = makeFunctor((Functor1<Derived&> \*)0,&foo);

//ok – chuyển kiểu ngầm – void operator()(Derived&)

f1 = makeFunctor((Functor1<Derived&> \*)0,&bar);

//ok – operator()(Derived&) sẽ vẫn là kiểu void – không return

class DataEntryField{

public:

DataEntryField(const Functor1wRet<const String &,Boolean> &v):validate(v){}

void keyHit(const String & stringSoFar){

if(validate(stringSoFar))

// xử lí..

}

private:

Functor1wRet<const String &,Boolean> validate;

//functor validate gồm một quá tải Boolean operator()(const String &)

};

Việc chuyển kiểu ngầm hay bỏ qua giá trị return sẽ thực hiện bởi trình biên dịch. (giữa derived và base, kiểu cơ bản, hay chuyển kiểu người dùng)

Thư viện đang thiếu khả năng kiểu - linh hoạt, như một phần mở rộng cần bổ sung cho ngôn ngữ lập trình vì sự giới hạn của con trỏ

Hàm dựng mặc định của functor sẽ nhận zero như là khởi tạo, đặt functor vào trạng thái “unset”, và xem như ép kiểu sang Boolen (false) để kiểm tra functor đẽ “set” chưa. Các lớp functor sẽ không phản hồi tới bất kỳ hàm ảo nào, bạn có thể thực hiện bất kỳ thao tác với đối tượng functor, nó cũng dễ dùng như con trỏ hàm

Tóm lại bạn đã hiểu được các vấn đề:

* Sử dụng callback service tử callback.h
* Triển khai callback service trong component
* Tạo functor cho các kiểu đối số đa dạng
* Kết nối bất kì đối tượng nào tới component dùng callback – gọi makeFunctor() cho đối tượng đó.

1. Sức mạnh Templates

As usual, what is easy for the user is often tricky for the implementor. Given the black-box descriptions above of the Functor classes and makeFunctor() it may be hard to swallow the claims of type-safety, transparent conversions, correct virtual function behavior etc. A look behind the curtain reveals not only how it works, but also some neat template techniques. Warning: most people find the pointer-to-member and template syntax used in the implementation daunting at first.

Obviously some sort of magic is going on. How can the Functor class, with no knowledge of the type or signature of the callee, ensure a type safe call to it, possibly with implicit conversions of the arguments? It can't, so it doesn't. The actual work must be performed by some code that knows both the functor callback signature and everything about the callee. The trick is to get the compiler to generate that code, and have the Functor to point to it. Templates can help out all around.

The mechanism is spread over three components - the Functor class, a Translator class, and the makeFunctor() function. All are templates.

The Functor class is parameterized on the types of the callback function signature, holds the callee data in a typeless manner, and defines a typed operator() but doesn't actually perform the work of calling back. Instead it holds a pointer to the actual callback code. When it comes time to call back, it passes the typeless data (itself actually), as well as the callback arguments, to this pointed-to function.

The Translator class is derived from Functor but is parameterized on both the Functor type \_and\_ the callee types. It knows about everything, and is thus able to define a fully type-safe static 'thunk' function that takes the typeless Functor data and the callback arguments. It constructs its Functor base class with a pointer to this static function. The thunk function does the work of calling back, turning the typeless Functor data back into a typed callee and calling the callee. Since the Translator does the work of converting the callee data to and from untyped data the conversions are considered 'safe'. The Translator isA Functor, so it can be used to initialize a Functor.

The makeFunctor() function takes the callee data, creates a Translator out of it and returns the Translator. Thus the Translator object exists only briefly as the return value of makeFunctor(), but its creation is enough to cause the compiler to lay down the static 'thunk' function, the address of which is carried in the Functor that has been initialized with the Translator.

All of this will become clearer with the details.

For each of the 10 Functor classes there are 2 Translator classes and 3 versions of makeFunctor(). We'll examine a slice of the library here, Functor1 and its associated Translators and makeFunctors. The other Functors differ only in the number of args and return values.

1. Đối tượng hàm

Vì đối tượng Functor chỉ là thực thể nằm trong caller, chúng phải chứa dữ liệu về callee. Chúng ta có thể thiết kế một lớp cơ sở chứa dữ liệu của đối tượng gọi (callee), không cần bận tâm xem callee là con trỏ hàm hay đối tượng để gọi hàm thành viên:

class FunctorBase{

public:

typedef void (FunctorBase::\*\_MemFunc)();

typedef void (\*\_Func)();

FunctorBase():callee(0),func(0){}

FunctorBase(const void \*c,const void \*f,size\_t sz)

{

if(c) //trường hợp là hàm thành viên

{

callee = (void \*)c;

memcpy(memFunc,f,sz);

}

else //trường hợp con trỏ hàm

{

func = f;

}

}

//for evaluation in conditions will be changed to bool when bool exists

operator int()const{return func||callee;}

class DummyInit{

};

////////////////////////////////////////////////////////////////

// Note: Đoạn code này tùy thuộc vào tất cả con trỏ hàm thành viên phải là cùng kích cỡ

// Ngược lại, bạn phải làm memFunc là lớn nhất

////////////////////////////////////////////////////////////////

union{

const void \*func;

char memFunc[sizeof(\_MemFunc)];

};

void \*callee;

};

All Functors are derived (protected) from this base. FunctorBase provides a constructor from typeless args, where if c is 0 the callee is a pointer-to-function and f is that pointer, else c is pointer to the callee object and f is a pointer to a pointer-to-member function and sz is that ptr-to-member-function's size (in case an implementation has pointer-to-members of differing sizes). It has a default constructor which inits to an 'unset' state, and an operator int to allow for testing the state (set or unset).

The Functor class is a template. It has a default constructor and the required operator() corresponding to its template parameters. It uses the generated copy constructor and assignment operators.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* one arg - no return \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template <class P1>

class Functor1:protected FunctorBase{

public:

Functor1(DummyInit \* = 0){}

void operator()(P1 p1)const

{

thunk(\*this,p1);

}

FunctorBase::operator int;

protected:

typedef void (\*Thunk)(const FunctorBase &,P1);

Functor1(Thunk t,const void \*c,const void \*f,size\_t sz):

FunctorBase(c,f,sz),thunk(t){}

private:

Thunk thunk;

};

The Functor class has a protected constructor that takes the same typeless args as FunctorBase, plus an additional first argument. This argument is a pointer to function (the thunk function) that takes the same arguments as the operator(), plus an additional first argument of type const FunctorBase &. The Functor stores this away (in thunk) and implements operator() by calling thunk(), passing itself and the other arguments. Thus it is this thunk() function that does the work of 'calling back'.

A key issue at this point is whether operator() should be virtual. In the first iteration of my mechanism the Functor classes were abstract and the operator()'s pure virtual. To use them for callbacks a set of derived template classes parameterized on the callee type was provided. This required that functors always be passed and held by reference or pointer and never by value. It also required the caller component or the client code maintain the derived object for as long as the callback relationship existed. I found the maintenance and lifetime issues of these functor objects to be problematic, and desired by-value syntax.

In the current mechanism the Functor classes are concrete and the operator() is non-virtual. They can be treated and used just like ptr-to-functions. In particular, they can be stored by value in the component classes.

1. Bộ phiên dịch Translator

Where does the thunk() come from? It is generated by the compiler as a static member of a template 'translator' class. For each Functor class there are two translator classes, one for stand-alone functions (FunctionTranslator) and one for member functions (MemberTranslator). The translator classes are parameterized by the type of the Functor as well as the type(s) of the callee. With this knowledge they can, in a fully type-safe manner, perform two important tasks.

First, they can initialize the Functor data. They do this by being publicly derived from the Functor. They are constructed with typed callee information and which they pass (untyped) to the functor's protected constructor.

Second, they have a static member function thunk(), which, when passed a FunctorBase, converts its callee data back into typed information, and executes the callback on the callee. It is a pointer to this static function which is passed to the Functor constructor.

template <class P1,class Func>

class FunctionTranslator1:public Functor1<P1>{

public:

FunctionTranslator1(Func f):Functor1<P1>(thunk,0,f,0){}

static void thunk(const FunctorBase &ftor,P1 p1)

{

(Func(ftor.func))(p1);

}

};

FunctionTranslator is the simpler of the two. It is parameterized by the argument type of the Functor and some ptr-to-function type (Func). Its constructor takes an argument of type Func and passes it and a pointer to its static thunk() function to the base class constructor. The thunk function, given a FunctorBase ftor, casts ftor's func member back to its correct type (Func) and calls it. There is an assumption here that the FunctorBase ftor is one initialized by the constructor (or a copy). There is no danger of it being otherwise, since the functors are always initialized with matching callee data and thunk functions. This is what is called a 'safe' cast, since the same entity that removed the type information also re-instates it, and can guarantee a match. If Func's signature is incompatible with the call, i.e. if it cannot be called with a single argument of type P1, then thunk() will not compile. If implicit conversions are required the compiler will perform them. Note that if func has a return it is safely ignored.

template <class P1,class Callee, class MemFunc>

class MemberTranslator1:public Functor1<P1>{

public:

MemberTranslator1(Callee &c,const MemFunc &m):

Functor1<P1>(thunk,&c,&m,sizeof(MemFunc)){}

static void thunk(const FunctorBase &ftor,P1 p1)

{

Callee \*callee = (Callee \*)ftor.callee;

MemFunc &memFunc(\*(MemFunc\*)(void \*)(ftor.memFunc));

(callee->\*memFunc)(p1);

}

};

MemberTranslator is parameterized by the argument type of the Functor, some class type (Callee), and some ptr-to-member-function type (MemFunc). Not surprisingly it's constructor is passed 2 arguments, a Callee object (by reference) and a ptr-to-member-function, both of which are passed, along with the thunk function, to the base class constructor. Once again, the thunk function casts the typeless info back to life, and then calls the member function on the object, with the passed parameter.

Since the Translator objects are Functor objects, and fully 'bound' ones at that, they are suitable initializers for their corresponding Functor, using the Functor's copy constructor. We needn't worry about the 'chopping' effect since the data is all in the base class portion of the Translator class and there are no virtual functions involved. Thus they are perfect candidates for the return value of makeFunctor()!

1. Phương thức makeFunctor

For each Functor class there are three versions of makeFunctor(), one for ptr-to-function and a const and non-const version for the object/ptr-to-member-function pair.

template <class P1,class TRT,class TP1>

inline FunctionTranslator1<P1,TRT (\*)(TP1)>

makeFunctor(Functor1<P1>\*,TRT (\*f)(TP1))

{

return FunctionTranslator1<P1,TRT (\*)(TP1)>(f);

}

The function version is straightforward. It uses the dummy argument to tell it the type of the functor and merely returns a corresponding FunctionTranslator. I mentioned above that the Func type parameter of FunctionTranslator was invariably a ptr-to-function type. This version of makeFunctor() ensures that by explicity specifying it as such.

template <class P1,class Callee,class TRT,class CallType,class TP1>

inline MemberTranslator1<P1,Callee,TRT (CallType::\*)(TP1)>

makeFunctor(Functor1<P1>\*,Callee &c,TRT (CallType::\* const &f)(TP1))

{

typedef TRT (CallType::\*MemFunc)(TP1);

return MemberTranslator1<P1,Callee,MemFunc>(c,f);

}

This is the gnarliest bit. Here makeFunctor is parameterized with the type of the argument to the Functor, the type of the callee, the type of the class of which the member-function is a member, the argument and return types of the member function. Whew! We're a long way from Stack<T> land! Like the ptr-to-function version, it uses the dummy first argument of the constructor to determine the type of the Functor. The second argument is a Callee object (by reference). The third argument is this thing:

TRT (CallType::\* const &f)(TP1)

Here f is a reference to a constant pointer to a member function of CallType taking TP1 and returning TRT. You might notice that pointer-to-member-functions are all handled by reference in the library. On some implementations they can be expensive to pass by value and copy. The significant feature here is that the function need not be of type pointer-to-member-of-Callee. This allows makeFunctor to match on (and ultimately work with) a ptr-to-member-function of some base of Callee. It then typedefs that bit and returns an appropriate MemberTranslator.

template <class P1,class Callee,class TRT,class CallType,class TP1>

inline MemberTranslator1<P1,const Callee,TRT (CallType::\*)(TP1)const>

makeFunctor(Functor1<P1>\*,const Callee &c,TRT (CallType::\* const &f)(TP1)const)

{

typedef TRT (CallType::\*MemFunc)(TP1)const;

return MemberTranslator1<P1,const Callee,MemFunc>(c,f);

}

This last variant just ensures that if the Callee is const the member function is also (note the const at the end of the third argument to the constructor - that's where it goes!).

That, for each of ten Functors, is the whole implementation.

1. Tổng kết

Callbacks are a powerful and necessary tool for component based object-oriented development in C++. They can be a tremendous aid to the interoperability of libraries. The template functor system presented here meets all the stated criteria for a good callback mechanism - it is object-oriented, compile-time type-safe, generic, non-type-intrusive, flexible and easy to use. It is sufficiently general to be used in any situation calling for callbacks. It can be implemented in the current language, and somewhat more elegantly in the proposed language.

This implementation of callbacks highlights the power of C++ templates - their type-safety, their code-generation ability and the flexibility they offer by accepting ptr-to-function and ptr-to-member-function type parameters.

Ultimately the greatest benefit is gained when class libraries start using a standard callback system. If callbacks aren't in the components, they can't be retrofitted. Upon publication of this article I am making this Callback library freely available in the hope that it will be adopted by library authors and serve as a starting point for discussion of a standard callback system.

### Liên kết Internal và External

Một đơn vị biên dịch ám chỉ một triển khai (.cpp) và tất cả header (.h/hpp) được include.

Nếu một đối tượng hay hàm nằm trong đơn vị biên dịch có liên kết nội bộ, thì chỉ có thể nhìn thấy bởi bộ liên kết bên trong đơn vị biên dịch.

Nếu đối tượng hay hàm có liên kết ngoại tuyến, thì bộ liên kết có thể nhìn thấy liên kết này khi xử lí đơn vị biên dịch chứa liên kết ngoại tuyến đó.

Biến và hàm **static** khi khai báo trong namespace global sẽ là liên kết nội bộ.

Biến và hàm **extern** mặc định là liên kết ngoại tuyến.

Trình biên dịch sẽ mặc định liên kết như sau:

* Biến non-const global mặc định có liên kết ngoại tuyến.
* Biến const global variables mặc định có liên kết nội bộ
* Hàm mặc định có liên kết ngoại tuyến.

1. Basics

Chúng ta qui ước thuật ngữ symbol đại diện cho bất kì thực thể mà bộ liên kết làm việc như biến, hàm, đối tượng…

1. Khai báo và định nghĩa

Một khai báo sẽ nói với trình biên dịch về sự tồn tại của symbol và làm nó có thể tham chiếu bởi symbol khác nằm bất kì đâu, mà không cần phải tường minh địa chỉ của symbol được tham chiếu tới.

Một định nghĩa sẽ nói trình biên dịch nội dung thân hàm, đối tượng…

Khi dữ liệu thành viên class là kiểu tham chiếu (con trỏ hay &), trong cùng một lúc, nó luôn được phép trỏ tới một kiểu đã khai báo, vì con trỏ chiếm vùng nhớ cố định là 4-bytes trên hệ 32-bit và 8-byte trên hệ 64-bits.

Khi tham chiếu tới nội dung con trỏ, thì phần định nghĩa trở nên cần thiết. Tương tự, với khai báo hàm, tất cả tham số và kiểu trả về chỉ cần khai báo, không cần định nghĩa.

Định nghĩa của tham số và kiểu trả về chỉ cần cho định nghĩa hàm.

Khai báo và định nghĩa của hàm rất rõ ràng như sau:

int f(); // declaration

int f() { return 42; } // definition

Đối với biến thì có khác, khai báo và định nghĩa thường không tường minh tách biệt. Ví dụ

int x;

Vừa khai báo nhưng cũng định nghĩa. Trong trường hợp này, hàm dựng mặc định của kiểu int sẽ không khởi tạo biến về zero.

Giá trị x ở trên sẽ tùy thuộc vào nội dung vùng nhớ mà nó đang giữ, vì kiểu cơ bản trên C++ không có khởi tạo mặc định.

Chúng ta có thể tường minh khai báo và định nghĩa biến tách biệt, bằng cách dùng từ khóa **extern**:

extern int x; // declaration

int x = 42; // definition

Tuy nhiên, nếu vừa dùng **extern** khai báo vừa khởi tạo giá trị thì cũng xem như đang định nghĩa biến global

extern int x = 5; // Vô tác dụng khi khai báo

int x = 5;

1. Chuyển tiếp khai báo

Chúng ta có thể sử dụng một khai báo mà chưa biết nó đã được định nghĩa hay chưa.

Bằng cách này, chúng ta không cần phải include đầy đủ phần định nghĩa (.cpp), chỉ cần phần khai báo là đủ (.h)

Ưu điểm là không cần phải biên dịch lại phần code sử dụng khai báo, trong khi cập nhật lại định nghĩa.

Lưu ý: symbol có thể khai báo nhiều lần, nhưng chỉ được định nghĩa một lần.

Ví dụ : có thể chuyển tiếp khai báo bao nhiều lần cũng được cho hàm hay class/struct

int f();

int f();

int f();

int f();

int f();

int f();

int f() { return 5; }

Trong khi thế này là bị cấm:

int f() { return 6; }

int f() { return 9; }

1. Đơn vị biên dịch

Đơn vị biên dịch được định nghĩa sẽ gồm tất cả file sau quá trình tiền xử lí, các macro, chỉ thị điều kiện #ifdef và #ifndef và #include….

Cho file header:

#ifndef HEADER\_HPP

#define HEADER\_HPP

#define VALUE 5

#ifndef VALUE

struct Foo { private: int ryan; };

#endif

int strlen(const char\* string);

#endif /\* HEADER\_HPP \*/

Và file program.cpp:

#include "header.hpp"

int strlen(const char\* string)

{

int length = 0;

while(string[length]) ++length;

return length + VALUE;

}

Sau quá trình tiền xử lí, đơn vị biên dịch sẽ có dạng:

int strlen(const char\* string);

int strlen(const char\* string)

{

int length = 0;

while(string[length]) ++length;

return length + 5;

}

1. Liên kết

Liên kết được hiểu là khả năng nhìn thấy các symbol khi xử lí file. Liên kết có 2 dạng là liên kết nội bộ và liên kết ngoại tuyến.

1. Liên kết ngoại tuyến

Symbol có liên kết ngoại tuyến có thể nhìn thấy tới bộ liên kết từ file khác.

Nó có thể chia sẽ giữa các đơn vị biên dịch.

Trên C,C++, từ khóa **extern** khai báo một symbol có liên kết ngoại tuyến:

extern int x;

extern void f(const std::string& argument);

Như đã nói, biến **global non-const** mặc định có liên kết ngoại tuyến, vậy giữa **extern** và **non-const global** có gì khác nhau?

Thực ra, **int x**, sẽ vừa định nghĩa, vừa khai báo biến global. Cũng như dùng **extern** mà lại khởi tạo giá trị vậy:

int x;

extern int x{};

// Hai trường hợp trên sẽ gây lỗi liên kết.

extern int x; // cú pháp đúng.

Ví dụ hàm f được khai báo liên kết ngoại tuyến trong file.hpp và cũng định nghĩa trong cùng một file:

// file.hpp

#ifndef FILE\_HPP

#define FILE\_HPP

extern int f(int x);

/\* ... \*/

int f(int x) { return x + 1; }

/\* ... \*/

#endif /\* FILE\_HPP \*/

Việc dùng **extern** ở đây là thừa, vì tất cả hàm gọi đều ngầm định là **extern**, tách biệt khai báo và định nghĩa cũng không cần thiết. Nên viết gọn lại:

// file.hpp

#ifndef FILE\_HPP

#define FILE\_HPP

int f(int x) { return x + 1; }

#endif /\* FILE\_HPP \*/

Tìm hiểu cách định nghĩa trên là không nên. Dùng hai file: a.cpp và b.cpp đều include file.hpp:

// a.cpp

#include "file.hpp"

/\* ... \*/

// b.cpp

#include "file.hpp"

/\* ... \*/

Trình biên dịch sẽ sinh ra hai đơn vị biên dịch cho 2 file .cpp

// TU A, from a.cpp

int f(int x) { return x + 1; }

/\* ... \*/

// TU B, from b.cpp

int f(int x) { return x + 1; }

/\* ... \*/

Sau khi biên dịch, bộ liên kết sẽ làm việc. Bộ liên kết khi thấy symbol **f** và tìm định nghĩa cho nó, nó tìm thấy 2 định nghĩa, từ 2 đơn vị biên dịch khác nhau.

Bộ liên kết sẽ dừng và báo cho bạn lỗi kiểu như:

duplicate symbol \_\_Z1fv in:

/path/to/a.o

/path/to/b.o

Tình huống trên vi pham luật 1 định nghĩa nên bộ liên kết báo lỗi.

1. Liên kết nội bộ

Symbol có liên kết nội bộ chỉ nhìn thấy trong đơn vị biên dịch mà thôi. Nên phân biệt khái niệm nhìn thấy được và truy xuất được.

Khi khai báo symbol có liên kết nội bộ trong file header, mỗi đơn vị biên dịch tách biệt include header này sẽ có một bản sao của symbol đó

Để khai báo symbol có liên kết nội bộ, C và C++ dùng từ khóa **static**.

Xem ví dụ:

//header.hpp:

static int variable = 42;

//file1.hpp:

void function1();

//file2.hpp:

void function2();

//file1.cpp:

#include "header.hpp"

void function1() { variable = 10; }

//file2.cpp:

#include "header.hpp"

void function2() { variable = 123; }

//main.cpp:

#include "header.hpp"

#include "file1.hpp"

#include "file2.hpp"

#include <iostream>

auto main() -> int

{

function1();

function2();

std::cout << variable << std::endl;

}

Vì biến static variable là liên kết nội bộ, nên 3 đơn vị biên dịch: file1.cpp, file2.cpp và main.cpp đều có bản sao riêng.

1. Namespace nặc danh

Trên C++ có 1 cách để khai báo một hay nhiều symbols có liên kết nội bộ: dùng namespace nặc danh.

Namespace nặc danh có thể dùng như một wrapper, các symbol nằm trong wrapper dù không khai báo tường minh **static** nhưng chúng có hành vi tương tự, tức chỉ nhìn thấy bên trong namespace mà thôi.

Trong mọi trường hợp nào thì:

namespace { int variable = 0; }

cũng giống như:

static int variable = 0;

1. Cách dùng

Trong hướng đối tượng, việc tạo ra nhiều bản sao sẽ gây lãng phí bộ nhớ khi mỗi đơn vị biên dịch đều sở hữu bản sao riêng.

Tương tượng khi bạn có một triển khai singleton (class chỉ khởi tạo duy nhất một thực thể), và đột nhiên bị nhân bản tràn lan, khiến singleton chả còn ý nghĩa gì nữa.

Một trường hợp khá thú vị là che giấu hàm trong đơn vị biên dịch cục bộ khỏi phạm vi global. Tưởng tượng, bạn đang có hàm subroutine chỉ dùng trong file1.h, nhưng cũng có một hàm subroutine khác chỉ dùng trong file2.h

Hai hàm subroutine này có cùng chức năng và cùng tên, nên có thể dùng **static** cho cả 2 hàm. Nếu không cùng include cả 2 header vào cùng một đơn vị biên dịch, thì chúng vẫn chỉ là liên kết nội bộ.

### Export/Import DLL

1. Windows DLL Files

Windows DLL files là một nhánh của PE file với vài điểm khác biệt:

* Phần mở rộng .DLL
* Entry point bắt đầu sẽ là DllMain() thay vì WinMain() hay main().
* Cờ flag DLL sẽ được set trong PE header.

DLLs có thể được nạp theo hai cách:

* Vào thời điểm nạp ứng dụng
* Gọi hàm LoadModule() Win32 API hay LoadLibrary() .

1. Export theo tên

Các hàm exported trong DLL file phải dùng cú pháp sau:

\_\_declspec(dllexport) Kiểu\_trả\_về Tên\_hàm(Danh\_sách\_tham\_số);

\_\_declspec(dllexport) Tên\_biến;

class/struct \_\_declspec(dllexport) Tên\_class/struct{ …//định nghĩa…};

Từ khóa "\_\_declspec" không phải của standard C, nhưng được nhiều trình biên dịch mở rộng hỗ trợ như một tính năng cho hàm và biến.

Microsoft C Compiler và GCC chạy trên môi trường window cho phép từ khóa **\_\_declspec**, và thuộc tính **dllexport**

Hàm có thể exported từ file exe, khi đó file thực thi .exe với hàm exported sẽ được gọi tương tự như file .dll. Tuy nhiên trường hợp này là khá hiếm.

Ví dụ:

class \_\_declspec(dllexport) nativeType{

private:

int i;

public:

nativeType();

~nativeType();

static void Test();

void Set(int i);

};

\_\_declspec(dllexport) void exported\_native\_Function(std::vector<nativeType>& v);

\_\_declspec(dllexport) extern nativeType native\_object;

Đối với class, có thể export tất cả thành viên và cả thành viên static, nhưng phải định nghĩa chúng.

Khi đã khai báo export cho class thì không được khai báo export cho từng thành viên trong class.

Có một ngoại lệ là chúng ta có thể export hàm thuần ảo virtual, tuy nhiên luôn phải định nghĩa hàm hủy của lớp trừu tượng và nó sẽ luôn được gọi sau khi gọi hàm hủy lớp dẫn xuất.

**Lưu ý:**

**\_\_declspec(dllexport)** không áp dụng cho hàm hay biến, đối tượng có thuộc tính **static.**

**\_\_declspec(dllexport)** có thể dùng cho đối tượng CLR nhưng không áp dụng cho các hàm hay hàm thành viên /**CLR**.

Khi đó muốn export, chúng ta phải chuyển thủ tục gọi hàm bằng cách dùng phần mở rộng:

* **\_\_stdcall**: cho hàm global có thủ tục gọi **\_\_crlcall**
* **\_\_thiscall**: cho hàm thành viên

1. Export theo Ordinal

Ngoài cách export hàm ra DLL là bằng tên, thông qua cú pháp **\_\_declspec(dllexport)**.

Ngoài ra, chúng ta có thể export hàm bằng số thứ tự với file .def. Với kỹ thuật này, chúng ta không cần phải can thiệp vào source code mà vẫn có thể export hàm và biến.

Để tìm số thứ tự của hàm, nối thứ tự của nó với tên hàm trong file .def

Nếu muốn tối ưu kích thước file DLL, dùng thuộc tính NONAME cho mỗi hàm export, số thứ tự sẽ chứa trong bảng export thay vì tên hàm.

Ưu điểm :

* Có thể tiết kiệm thời gian nếu có quá nhiều hàm export.
* Có thể export với các hàm gọi theo **\_\_clrcall**

**Lưu ý**:

Luật export vẫn giống như cách dùng **\_\_declspec(dllexport)**

1. Dùng file .def để Export

Module-definition (.def) file chứa một hay nhiều lệnh module mô tả nhiều thuộc tính cho DLL.

Nếu bạn không dùng từ khóa **\_\_declspec(dllexport)** để export hàm, biến thì DLL sẽ cần .def file.

File .def sẽ chứa các lệnh định nghĩa cho module như sau:

* Đầu tiên dùng lệnh **LIBRARY** để khai báo tên file DLL, bộ liên kết sẽ nhận diện đây là file .def và đặt tên DLL này trong thư viện import của DLL.
* Lệnh **EXPORTS** liệt kê tất cả tên và giá trị thứ tự của hàm hay biến export bởi DLL.
* Chúng ta sẽ đặng kí cho hàm tên export có dạng: tên\_hàm**@**số\_thứ \_tự

Số thứ tự phải từ 1 tới N, đây sẽ là số thứ tự hàm export qua DLL.

Ví dụ layout của một file .def:

LIBRARY BTREE

EXPORTS

Insert @1

Delete @2

Member @3

Min @4

Nếu dùng Wizard DLL MFC để tạo MFC DLL, thì wizard sẽ tạo một khung file .def cho bạn tự động và thêm vào project.

Thêm tên hàm cần exported vào file này, với các DLL không phải MFC, thì chúng ta phải tự tạo file .def và nạp vào project:

1. Tạo file .def với tên trùng với project và nằm cùng thư mục project
2. Trong Visual Studio, vào **Project->Properties**. Tới **Configuration Properties->Linker->Input** và gõ tên file DEF chúng ta vừa tạo trong trường **Module Definition File**.

Đối với từng project DLL, chúng ta phải làm các bước như trên, ví dụ khi **Debug** ta phải đặt output\_d.def trong cấu hình **Debug** cho thư viện output\_d.DLL, tương tự với cấu hình **Release**.

If you use the MFC DLL Wizard to create an MFC DLL, the wizard creates a skeleton .def file for you and automatically adds it to your project. Add the names of the functions to be exported to this file. For non-MFC DLLs, you must create the .def file yourself and add it to your project.

If you are exporting functions in a C++ file, you have to either place the decorated names in the .def file or define your exported functions with standard C linkage by using extern "C". If you need to place the decorated names in the .def file, you can obtain them by using the DUMPBIN tool or by using the linker /MAP option. Note that the decorated names produced by the compiler are compiler specific. If you place the decorated names produced by the Visual C++ compiler into a .def file, applications that link to your DLL must also be built using the same version of Visual C++ so that the decorated names in the calling application match the exported names in the DLL's .def file.

If you are building an extension DLL, and exporting using a .def file, place the following code at the beginning and end of your header files that contain the exported classes:

#undef AFX\_DATA

#define AFX\_DATA AFX\_EXT\_DATA

// <body of your header file>

#undef AFX\_DATA

#define AFX\_DATA

These lines ensure that MFC variables that are used internally or that are added to your classes are exported (or imported) from your extension DLL. For example, when deriving a class using DECLARE\_DYNAMIC, the macro expands to add a CRuntimeClass member variable to your class. Leaving out these four lines might cause your DLL to compile or link incorrectly or cause an error when the client application links to the DLL.

When building the DLL, the linker uses the .def file to create an export (.exp) file and an import library (.lib) file. The linker then uses the export file to build the DLL file. Executables that implicitly link to the DLL link to the import library when they are built.

Note that MFC itself uses .def files to export functions and classes from the MFCx0.dll.

1. Phân tích DLL Exports

Có nhiều cách để xác định các hàm được exported bởi DLL, chúng ta có thể dùng dumpbin theo cú pháp:

dumpbin /EXPORTS <dll file>

Xuất danh sách hàm export cùng với ordianl và RVA của chúng vào console.

1. Imports

Tương tự như hàm exports, module có thể import hàm vào một module khác. DLL file sẽ được nạp vào vùng nhớ process khi ứng dụng bắt đầu, và hàm sẽ được dùng như hàm local.

Khai báo hàm import cần có cú pháp như sau:

**\_\_declspec(dllimport) void MyFunction();**

Giúp trình biên dịch và bộ liên kết nhận biết hàm được import từ thư viện bên ngoài.

Khác với hành vi khi khai báo **\_\_declspec(dllimport)** và **\_\_declspec(dllexport)** cho kiểu non-class, thành viên dữ liệu static được phép định nghĩa trong module export.

1. Phân tích DLL Imports

Để xuất danh sách thư viện import ra console, dùng dumpbin với cú pháp:

dumpbin /IMPORTS <dll file>

Ngoài ra có thể dùng depends.exe thể liệt kê hàm import và export.

1. Export/Import hàm inline

Có thể áp dụng **dllexport** cho hàm inline, đoạn code sinh ra sẽ bỏ qua từ khóa inline cho hàm và export như hàm thông thường.

Ví dụ: biên dịch đoạn code sau thành DLL.

\_\_declspec(dllexport)

inline int times3(int i) { return i \* 3; }

int times9(int i) { return times3(times3(i)); }

Hàm times9 sẽ chèn thân hàm trong time3 mà không gọi tới times3.

* Còn một điểm nữa, nếu khai báo một con trỏ trỏ tới hàm inline, trình biên dịch sẽ phải sinh ra một phiên bản hàm non-inline và trả về địa chỉ hàm đó cho con trỏ. Trong trường hợp này, dllexport cũng làm con trỏ trỏ tới hàm nằm trong bảng export directory.Khi khai báo **dllimport** cho hàm inline, trình biên dịch sẽ inline thân hàm của nó vào nơi gọi như hàm inline export.
* Khi lấy địa chỉ của hàm import inline, địa chỉ của nó trong DLL sẽ được trả về như hàm import non-inline.
* Sử dụng hàm export inline là một vấn đề lớn, đặc biệt khi chúng bị ngầm định inline mà bạn không nhận ra.Ví dụ:

class \_\_declspec(dllexport) SimpleValue

{

public:

SimpleValue() : m\_value(0) { }

void setValue(int value);

int getValue() { return m\_value; }

private:

int m\_value;

};

Hàm dựng SimpleValue và SimpleValue::getValue() được export theo dạng inline! Do đó, bất kì thay đổi nào tới 2 hàm này sẽ phải biên dịch lại toàn bộ code gọi tới chúng.

* Nếu hàm inline sử dụng biến static cục bộ hay chuỗi hằng kí tự, thì nó được xem là biến static hay chuỗi hằng tại bất kì nơi nào. Ví dụ: chúng ta dùng biến static cục bộ là bộ đếm

int inline count()

{

static int c = 0;

return ++c;

}

* Khi update DLL có export hàm inline: các DLL import hàm inline bắt buộc phải biên dịch lại, vì chúng chèn code từ hàm import inline thay vì gọi hàm import theo cách thông thường.

1. Thừa kế và class export

Tất cả lớp cơ sở của export class đều phải được export, nếu không trình biên dịch sẽ cảnh báo.

Tất cả thành viên có thể truy xuất của class (public member) đều phải export.

Luật trên cho phép export class thừa kế từ import class, nhưng import class thừa kế từ export class thì không nên.

Theo luật truy xuất của C++, những gì mà người dùng có thể truy xuất tới DLL thì đều phải export, bao gồm thành viên private tham chiếu bởi hàm inline.

1. Luật Export/Import thành viên class

Các trường hợp dùng import và export cho từng thành viên class:

* Phiên bản hạn chế từ class interface: cung cấp một phiên bản không đầy đủ tính năng (bản dùng thử chẳng hạn)
* Khi người dùng DLL chỉ cần truy xuất vài thành viên mà thôi.

Khi export hàm ảo, phải export tất cả hàm ảo từ lớp cơ sở trong chuỗi thừa kế, hoặc ít nhất cung cấp phiên bản có thể sử dụng cho người dùng.

Hàm thành viên ảo khi khai báo export phải định nghĩa ngay trong phần khai báo.

Khi đã khai báo export thì phải có định nghĩa, nếu không trình biên dịch sẽ báo lỗi.

Trình biên dịch sẽ cảnh báo khi bạn khai báo import/export trong phần định nghĩa mà không phải trong phần khai báo class.

### Pre-comlied header

1. Mục đích

Precomplied header là file header (trên C/C++) được biên dịch thành một dạng thức, giúp xử lí nhanh hơn khi biên dịch.

Việc dùng precomplied header giảm đáng kể thời gian biên dịch, đặc biệt với header file lớn, chứa nhiều header file khác trong nhiều đơn vị biên dịch khác nhau.

Trên C/C++, nội dung header file sẽ được chèn vào source file nào khai báo #include cho nó.

Header file thinh thoảng chứa lượng lớn source code (như Window.h trên MS Window hay Cocoa/Cocoa.h trên OS X).

Điều này rất phổ biến trên những header file dùng teamplate như thư viện toán học Eigen và Boost C++. Chúng được viết hầu như theo header file, không dùng liên kết khi runtime.

Do đó, mỗi lần người dùng biên dịch ứng dụng, hàng loạt các thư viện header cũng phải biên dịch lại các thư viện cho các đối tượng dùng chung hay làm thư viện liên kết.

Một số trình biên dịch cho phép header file được biên dịch thành dạng thức nhanh hơn cho quá trình biên dịch. Dạng thức này gọi là precomplied header, và có phần mở rộng phổ biến là .pch trên Microsoft Visual C++ hay .gch trên GNU complier.

Ví dụ:

//header.hpp

...

//source.cpp

#include "header.hpp"

...

Khi biên dịch source.cpp lần đầu với tính năng precomplied header, trình biên dịch sẽ sinh ra file header .pch. Lần kế tiếp, nếu nhãn thời gian của header không thay đổi, trình biên dịch có thể bỏ qua giai đoạn biên dịch liên quan tới file **header .hpp** và dùng trực tiếp **header.pch**

1. Microsoft Visual C++ complier

stdafx.h là header file tạo bởi Microsoft Visual Studio IDE wizard. AFX trong stdafx.h viết tắt của Application Framework eXtensions.

Compatible compilers (Visual C++ 6.0 hay mới hơn) sẽ biên dịch file này trước để giảm thời gian xử lí. Visual C++ sẽ không biên dịch mọi thứ nằm trước chỉ thị **#include "stdafx.h"** trong source file, trừ khi tùy chọn /Yu không bị check

Tùy chọn /Yu, tức có sử dụng precomplied header, mặc định được check, nó có ý nghĩa rằng tất cả source code được include trong stdafx.h đã được biên dịch trước rồi.

Tùy chọn /Yc sẽ xác định file .cpp được chọn để tạo precompiled header

1. GNU complier

Precompiled headers được hỗ trợ cho GCC (3.4 về sau). Phương thức của GCC khá giống với Visual C++ complier, GCC lưu phiên bản precomplied của header file trong file **“.gch”**. Khi biên dịch source file, trình biên dịch kiểm tra xem file “.gch” có nằm trong cùng thư mục, sau đó dùng nó nếu có thể.

GCC chỉ có thể dùng phiên bản precomplied được biên dịch bởi cùng một cấu hình của compiler. Ngoài ra, chỉ có các chỉ thị tiền xử lý được phép nằm trước precomplied header mà thôi.

GCC tự động nhận diện hầu hết header file bởi phần mở rộng của chúng. Tuy nhiên, nếu việc nhận diện thất bại (bởi phần mở rộng không hỗ trợ), thì chúng ta có thể dùng tùy chọn cấu hình “–x” cho complier switch để đảm bảo file này được nhận diện như là header file.

1. C++Builder

Trong cấu hình project mặc định, C++ Builder complier ngầm định sinh ra file precompiled header cho tất cả header include trong module source, trừ khi #pragma hdrstop được tìm thấy.

Precompiled haeder được chia sẻ cho tất cả modules của project nếu cần. Ví dụ, khi làm việc với thư viện của Visual Component, header vcl.h được dùng rất phổ biến, đặt vcl.h vào precompiled header để chia sẻ cho các modules, tối ưu thời gian biên dịch.

C++ Builder có thể chỉ định header file nào dùng cho precompiled header, tương tự như tính năng trên Visual C++.

Tính năng "Precompiled Header Wizard" cho phép quét tất cả include header file trong tất cả source modules của project, phân loại chúng, tạo và test precompiled header cho các file source tự động.

1. Tips
2. Không pre-compile biến constant

Trình biên dịch không thể pre-compile header file nếu nó chứa **constant**.

Nếu muốn đặt biến **constant** trong header, phải tạo riêng header chứa các biến **constant**, và không pre-complie header này.

Các header nằm trong pre-compiled header không được include header chứa khai báo **constant**, chúng ta có thể dùng từ khóa extern để khai báo **constant** , còn định nghĩa để trong file source.

Lưu ý rằng vấn đề biến **constant** chỉ xảy ra khi pre-compile source file heaer. Nếu không nằm trong pre-compile header, có thể khai báo biến **constant** thoải mái trong header.

Ngoài ra, có thể dùng directive **#define** thay vì dùng **constant**

1. Không pre-compile template headers

Theo kinh nghiệm thực tế, trình biên dịch có thể pre-compile header chứa template class, nhưng pre-complied image sẽ luôn bị biên dịch lại trong quá trình incremental make.

Trình biên dịch luôn xử lí template theo cách này, một số template pre-compile chạy tốt, một số thì không, chúng ta phải để ý các log của quá trình biên dịch tìm ra nguyên nhân bị lỗi.

Để quá trình biên dịch ổn định, tốt nhất không nên đưa template header vào pre-compiled header.

1. Không pre-compile cho các header files thay đổi thường xuyên

Tất cả thay đổi nhỏ cho header file sẽ khiến pre-compile header phải biên dịch lại, do đó chỉ nên đưa các header hệ thống và thư viện bên ngoài vào pre-compiled header mà thôi

1. Tên header phải đúng trong chỉ thị include

Ví dụ:

// MAINFORM.CPP

#include <VCL.H>

#include "pch.h"

#pragma hdrstop

// SPLASH.CPP

#include <VCL.H>

#include "PCH.H" // mismatched case

#pragma hdrstop

Trong trường hợp này, trình biên dịch sẽ sinh ra và sử dụng pre-compile image cho MAINFORM.CPP, nhưng với SPLASH.CPP thì lại tạo một phiên bản pre-compiled image riêng.

Luật: pre-compiled header yêu cầu mọi chi thị include file trước #pragma hdrstop phải khai báo đúng tên, còn các chỉ thị include sau đó có thể không cần chính xác.

### Bộ logging cho C++

Logging là kỹ thuật giám sát và duy trì hệ thống phần mềm. Tác giả Petru giới thiệu một logging framework thỏa mãn 3 tiêu chí:

* Typesafe
* Thread-safe
* Portable.

Nó đơn giản là công cụ cung cấp thông tin trạng thái hệ thống. Cơ chế logging tốt có thể tiết kiệm thời gian debug và chi phí bảo trì ứng dụng.

Link source: www.ddj.com/code/.

##### 1st version

1. Bước đầu

Thiết kế log sẽ áp dụng theo log level:

//Listing One

// Log, version 0.1: a simple logging class

enum TLogLevel {

logERROR,

logWARNING,

logINFO,

logDEBUG,

logDEBUG1,

logDEBUG2,

logDEBUG3,

logDEBUG4

};

Phương thức ToString() nhận tham số kiểu enum và trả về tên log level tương ứng, trường hợp không khớp mặc định trả về macro NODEBUG:

#define NODEBUG "No Debug"

std::string ToString(const TLogLevel &level) {

switch (level) {

case logERROR:

return std::string("logERROR");

case logWARNING:

return std::string("logWARNING");

case logINFO:

return std::string("logINFO");

case logDEBUG:

return std::string("logDEBUG");

case logDEBUG1:

return std::string("logDEBUG1");

case logDEBUG2:

return std::string("logDEBUG2");

case logDEBUG3:

return std::string("logDEBUG3");

case logDEBUG4:

return std::string("logDEBUG4");

default:

return std::string(NODEBUG);

}

}

Triển khai lớp Log trong Listing One:

//Listing One

class Log

{

public:

Log();

virtual ~Log();

std::ostringstream& Get(TLogLevel level = logINFO);

public:

static TLogLevel& ReportingLevel();

protected:

std::ostringstream os;

private:

Log(const Log&);

Log& operator =(const Log&);

private:

TLogLevel messageLevel;

};

Phương thức wrapper ReportingLevel để set log level:

TLogLevel& Log::ReportingLevel() {

static TLogLevel ReportingLevel = logDEBUG;

return ReportingLevel;

}

Lớp log sử dụng thành viên kiểu std::ostringstream là “os” để tích lũy dữ liệu log. Phương thức Accessor Get() dùng để truy xuất tới thành viên dữ liệu “os”, đồng thời set level cho thành viên messageLevel:

std::ostringstream& Log::Get(TLogLevel level)

{

os << "- " << NowTime();

os << " " << ToString(level) << ": ";

os << std::string(level > logDEBUG ? 0 : level - logDEBUG, '\t');

messageLevel = level;

return os;

}

Sau khi tất cả dữ liệu được định dạng, hàm hủy của Log tiếp tục xuất ra standard error. Sử dụng:

Log().Get(logINFO) << "Hello " << username;

Dòng lệnh trên tạo ra đối tượng với level log là logINFO, dữ liệu đẩy vào đối tượng std::stringstream, định dạng và tích lũy, cuối cùng ghi chuỗi kí tự vào log file dùng fprintf().

Quá trình flush được khai báo trong hàm hủy: 2 bước cuối cùng khá quan trọng vì chúng liên quan đến threadsafe của cơ chế log. Hàm fprintf() được triển khai threadsafe, nên nếu log được dùng bởi nhiều thread khác nhau, thì output không bị xung đột.

Log::~Log()

{

if (messageLevel <= Log::ReportingLevel())

{

os << std::endl;

fprintf(stderr, "%s", os.str().c\_str());

fflush(stderr);

}

}

Trước khi tối ưu triển khai, chúng ta sẽ viết đoạn code chèn tab cho logging level, và ngắt dòng cho mỗi chuỗi văn bản.

Giúp các dòng log dễ đọc cho người dùng và cả máy.

Hàm main:

int main(int argc, char\*\* argv) {

Log::ReportingLevel() = logDEBUG1;

const int count = 3;

Log::Get(logDEBUG) << "A loop with " << count << " iterations";

for (int i = 0; i != count; ++i) {

Log::Get(logDEBUG) << "the counter i = " << i;

}

return 0;

}

**which outputs:**

**- 22:25:23.643 DEBUG: A loop with 3 iterations**

**- 22:25:23.644 DEBUG1: the counter i = 0**

**- 22:25:23.645 DEBUG1: the counter i = 1**

**- 22:25:23.645 DEBUG1: the counter i = 2**

Lùi đầu dòng và tabs giữa các đoạn giúp log sáng sủa hơn

1. Tối ưu

Framework log hiện tại có vấn đề hiệu suất, có thể dùng mẹo nhỏ là khai báo directive #define cho log:

#define LOG(level) \

if (level > Log::ReportingLevel()) ; \

else Log().Get(level)

Sử dụng:

LOG(logINFO) << "Hello " << username;

Để tối ưu hiệu suất, tức không cần logging, chúng ta sẽ truyền vào logging level cao hơn thành viên messageLevel.

Khi cần logging, chỉ cần truyền level thấp hơn messageLevel hiện tại của Log.

Phiên bản sau khi xử lí directive có dạng như sau:

if (logINFO > Log::ReportingLevel()){

}

else

Log().Get(logINFO) << "Hello " << username;

Việc kiểm tra messageLevel trong hàm hủy là không cần thiết:

Log::~Log()

{

os << std::endl;

fprintf(stderr, "%s", os.str().c\_str());

fflush(stderr);

}

Thiết kế log chỉ thực thi khi logging level hợp lệ, khi dùng macro cần tránh bug liên quan, ví dụ:

LOG(logINFO) << "A number of " << NotifyClients() << " were notified.";

Số lượng người dùng chỉ được thông báo chỉ khi log level là logINFO hay thấp hơn. Code đúng nên là:

const int notifiedClients = NotifyClients();

LOG(logINFO) << "A number of " << notifiedClients << " were notified.";

1. Generic log

Lớp Log đang bị gán cứng với stderrr, nếu thư viện Log tích hợp vào ứng dụng GUI, nó nên log vào file ASCII.

Người dùng thư viện log API nên xác định mục đích của log, không khó để thiết kế tham số cho log để thay đổi tới FILE\*, nhưng chúng ta nên cho nó cần câu hơn là cho nó con cá?

Phương pháp tốt hơn là thiết kế lại log, sử dụng lớp Output2File để quản lý giao tiếp, chính sách log phải linh hoạt với tham số, khác với việc log theo level.

Lớp Output2File sẽ cung cấp phương thức giao tiếp Output, nhận tham số std::string là:

static void Output(const std::string& msg);

Triển khai lớp template log nhận tham số template là lớp Output2File:

template <typename OutputPolicy>

Log<OutputPolicy>::~Log()

{

os<<std::endl;

OutputPolicy::Output(os.str());

}

Hàm hủy của log sẽ gọi tới Output của lớp policy để xuất chuỗi ra stream tương ứng.

Lớp Output2File dùng phương thức Stream để thiết lập con trỏ FILE\* output, xem Listing Two:

//Listing Two

class Output2FILE // triển khai của OutputPolicy

{

public:

static FILE\*& Stream();

static void Output(const std::string& msg);

};

inline FILE\*& Output2FILE::Stream()

{

static FILE\* pStream = stderr;

return pStream;

}

inline void Output2FILE::Output(const std::string& msg)

{

FILE\* pStream = Stream();

if (!pStream)

return;

fprintf(pStream, "%s", msg.c\_str());

fflush(pStream);

}

typedef Log<Output2FILE> FILELog;

#define FILE\_LOG(level) \

if (level > FILELog::ReportingLevel() || !Output2FILE::Stream()) ; \

else FILELog().Get(messageLevel)

Phương thức Output của lớp Output2File lấy con trỏ FILE\* từ phương thức Stream, dùng nó để xuất ra chuỗi kí tự đầu vào.

Đoạn code bên dưới sẽ mặc định dùng default stderr (error checking/handling omitted for brevity), tuy nhiên chúng ta có thể cấu hình FILE\* theo ý muốn:

FILE\* pFile = fopen("application.log", "a");

Output2FILE::Stream() = pFile;

FILE\_LOG(logINFO) << ...;

1. Thread safe

Lưu ý cho ứng dụng đa luồng: lớp OutPut2File sẽ tốt nếu bạn không dùng đa luồng.

Ngược lại, nếu log stream thay đổi theo runtime từ các thread khác nhau, bạn nên dùng khóa nội bộ tùy theo phương tiện cung cấp bởi thread platform, hay wrapper như Boost threads.

Listing 3 dưới đây hỗ trợ Boost threads:

//Lising Three

#include "boost\thread\mutex.hpp"

class Output2FILE

{

public:

static void Output(const std::string& msg);

static void SetStream(FILE\* pFile);

static FILE\*& Stream();

static boost::mutex GetMutex();

};

inline std::mutex& OutputPolicy::GetMutex(){

static std::mutex mtx;

return mtx;

}

inline FILE\*& Output2FILE::Stream()

{

static FILE\* pStream = stderr;

return pStream;

}

inline void Output2FILE::SetStream(FILE\* pFile)

{

boost::mutex::scoped\_lock lock(GetMutex());

Stream() = pFile;

}

inline void Output2FILE::Output(const std::string& msg)

{

boost::mutex::scoped\_lock lock(GetMutex());

FILE\* pStream = Stream();

if (!pStream)

return;

fprintf(pStream, "%s", msg.c\_str());

fflush(pStream);

}

Chúng ta cung cấp thêm phương thức SetStream() nhận tham số FILE\* để thiết lập stream. Khóa nội bộ sẽ được triển khai cho phương thức SetStream() và Output()

Triển khai log dùng khóa nội bộ sẽ chậm hơn, vì việc kiểm tra trong macro chưa đồng bộ - race condition sẽ không xảy ra.

1. Compile-Time Plateau Logging Level

Thỉnh thoảng, quá trình runtime khi có những log không cần thiết bị bị chậm khá nhiều. Hãy cung cấp phương tiện để triệt tiêu một số log tại thời điểm biên dịch bằng chỉ thị tiền xử lí FILELOG\_MAX\_LEVEL:

#ifndef FILELOG\_MAX\_LEVEL

#define FILELOG\_MAX\_LEVEL logDEBUG4

#endif

#define FILE\_LOG(level) \

if (level > FILELOG\_MAX\_LEVEL) ;\

else if (level > Log::ReportingLevel() || !Output2FILE::Stream()) ; \

else Log().Get(level)

Marco tiến hành 2 lần kiểm tra:

Quá trình kiểm tra thứ nhất với các log level cao hơn FILELOG\_MAX\_LEVEL thì triệt tiêu code.

Quá trình kiểm tra log level thứ hai giống như trước.

Việc tối ưu này có phạm vi trên diện rộng, bạn có thể tin tưởng nó hoạt động tốt trên mọi môi trường.

Để hình dung, chúng ta dùng Marco trong vòng lặp 3 lần với level cấu hình là DEBUG1:

bash-3.2$ g++ main.cpp

bash-3.2$ ./a.exe DEBUG1

- 22:25:23.643 DEBUG: A loop with 3 iterations

- 22:25:23.644 DEBUG1: the counter i = 0

- 22:25:23.645 DEBUG1: the counter i = 1

- 22:25:23.645 DEBUG1: the counter i = 2

bash-3.2$ g++ main.cpp -DFILELOG\_MAX\_LEVEL=3

bash-3.2$ ./a.exe DEBUG1

- 22:25:31.972 DEBUG: A loop with 3 iterations

1. Đánh giá hiệu suất

Có một kỹ thuật khá thú vị khi so sánh quá trình log trên các platform khác nhau hay cùng một flatform nhưng khác version (Linux – Window), với cấu hình trình biên dịch khác nhau (debug với optimize)

So sánh 2 output file với 2 chế độ log logINFO và logDEBUG4, có thể dễ dàng nhận ra mức độ chi tiết của log với level cao nhất, tất nhiên là file log sẽ rất lớn.

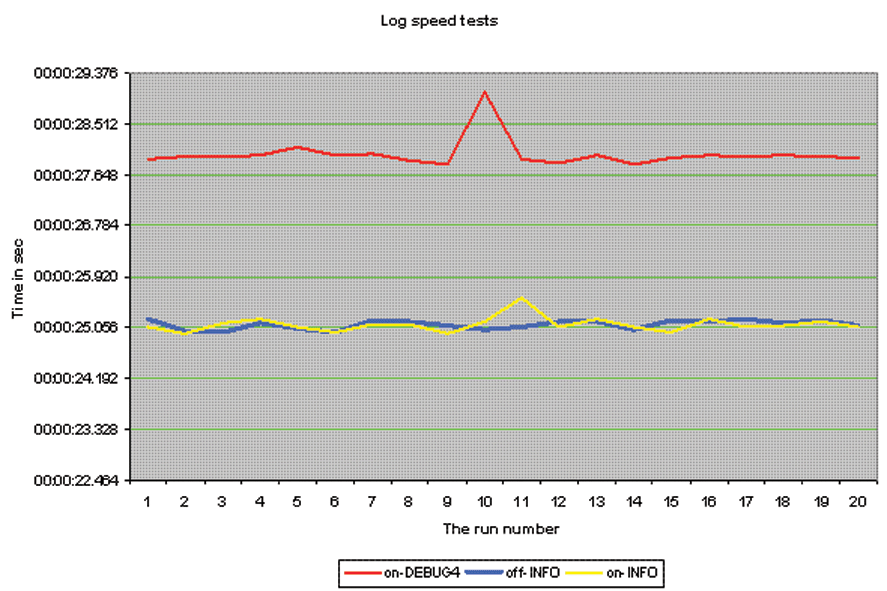
Một điểm rất hay khi dùng log framework là có thể kết hợp cho nhiều ứng dụng khác nhau (hay cho nhiều bản sao ứng dụng) đều dùng một file log duy nhất.

Chỉ cần mở cùng file từ nhiều ứng dụng khác nhau ở chế độ append – chúng ta đang log lại các thông tin từ nhiều process khác nhau.

Để đánh giá hiệu suất, chạy code trên 3 cấu hình:

* off-INFO. Tắt log tại thời điểm biên dịch.
* on-INFO.Bật log tại thời điểm biên dịch , tắt trong lúc chạy (dùng ở level INFO).
* on-DEBUG4. Bật log trong lúc chạy và biên dịch (dùng ở level DEBUG4).

Mỗi cấu hình chạy 20 lần. Hình 1 và bảng 1 minh họa thống kê



**Figure 1: Log speed tests.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | on-DEBUG4 | off-INFO | on-INFO |
| AVERAGE | 00:00:28.002 | 00:00:25.106 | 00:00:25.109 |
| STDEV | 0.0000029838 | 0.0000008605 | 0.0000014926 |

**Table 1**

Chi phí khi dùng log ở chế độ cao nhất DEBUG4 là không nhiều, rất đáng để sử dụng. Ngoài ra chúng ta có thể output thêm ThreadID vào log file khi có nhiều ứng dụng cùng chạy log, việc này khá hữu ích khi lọc thông tin

1. Cải thiện log

Log framework ở bài trước tỏ ra khá mạnh mẽ, đơn giản, chia thành nhiều level thông tin cho người dùng. Log level tùy thuộc vào mức độ quan trong và thông tin cần biết. Khi ứng dụng chạy, log level có hiệu ứng trên toàn phạm vi, nên chỉ có những dòng lệnh nào hợp lệ mới được log.

Log::ReportingLevel() = logINFO;

FILE\_LOG(logINFO) << "This log statement is enabled";

FILE\_LOG(logDEBUG) << "This log statement is disabled";

Chúng ta khuyến khích sử dụng log level ở cấp cao nhất đến khi code đủ tốt, hay khi cần kiếm bug nào đó. Ví dụ: log nội dung một vài biến rất quan trọng khi cần chắc chắn ứng dụng hoạt động tốt, nhưng nó chỉ là mớ hỗn độn đầy nhiễu loạn.

Khi code chạy ổn định, bạn có thể giảm dần log level để tăng hiệu suất cho ứng dụng.

Giả sử ứng dụng khi đã thành phẩm, rơi vào trạng thái tồi tệ và không hoạt động tốt. Bạn muốn đẩy log level lên cao hơn, nhưng nếu muốn thay đổi một số cấu hình cố định và khởi động lại ứng dụng, thì rất khó bắt được nguyên nhân trước đó. Tăng log level cũng tăng lượng dữ liệu log vào, khiến ứng dụng khó thích nghi trong môi trường sản xuất.

Trong phần này, chúng ta sẽ cung cấp khả năng “hot-enable” log mà không cần ngưng và khởi động lại ứng dụng.

Khi bạn tìm ra manh mối về nguyên nhân, có thể đưa log level về trạng thái cũ khi runtime.

Link source code hoàn chỉnh: http://twimgs.com/ddj/images/article/2009/code/logcpp2.zip

1. Some Ingredients

Chúng ta sẽ sử dụng line-level để bật/tắt log statements.

Chia sẻ dữ liệu giữa các thread dùng biến đơn giản và không cần khóa. Cách đơn giản và hiệu quả để giao tiếp giữa các thread là dùng biến chia sẻ. Chúng ta phải thỏa mãn điều kiện sau:

Đọc và ghi dữ liệu phải atomic (dùng kích thước theo word của máy)

Reading/writing nên dùng ranh giới vùng nhớ (thời điểm biên dịch hay runtime). Ứng dụng dùng biến global chia sẻ để báo hiệu khi nào log nên được bật hay tắt.

Một thread dùng để set biến với giá trị mong muốn, trong khi thread khác đọc biến. Thread set biến cần khóa, còn đọc biến thì không cần để việc chia sẻ dữ liệu hiệu quả.

Trường hợp định nghĩa biến staic local on-the-fly, bên trong vòng lặp. Ví dụ:

for (static int i = 0;;);

Cú pháp trên hợp lệ trên C++, và biến chỉ có thể nhìn thấy bên trong vòng lặp mà thôi. Bằng cách dùng vòng lặp như vậy cho log statements có thể đọc biến rất hiệu quả, không cần phải tìm kiếm.

Trường hợp định nghĩa biến local on-the-fly, bên trong lệnh if. Ví dụ

if (bool flag = false) ; else …

BIến chỉ có thể nhìn thấy bên trong lệnh if/else. Mục đích của những câu lệnh như vậy chỉ để đảm bảo bên trong vòng lặp được thực thi ít nhất 1 lần.

1. Cấu hình Logging Level theo Line Granularity

Ý tưởng cơ bản là định nghĩa biến static local cho mỗi lệnh log. Biến này dùng để bật/tắt từng lệnh log. Phương thức Set() sẽ giúp thay đổi giá trị bất kì biến static local nào.

Biến static local sẽ có các trạng thái:

* Uninitialized: log statements chưa được thực thi lần nào.
* Default: Người dùng không muốn thay đổi log level mặc định.
* Enabled: Người dùng muốn bật log, mà không cần quan tâm tới giá trị log level global.
* Disabled: Người dùng muốn tắt log, mà không cần quan tâm tới giá trị log level global.

Chúng ta dùng enum định nghĩa các trạng thái:

enum TEnabler {NOT\_INIT = -2, DEFAULT, DISABLE , ENABLE};

Nếu định nghĩa biến static local bên trong vòng lặp như trên, chúng ta có thể bao nó lại trong macro, người dùng chỉ cần sử dụng mà không cần quan tâm nội dụng của nó. Ví dụ:

#define LOGMAN(level) for(static TEnabler enabler = NOT\_INIT; …;…) FILE\_LOG().Get(logINFO)

Sử dụng Macro:

LOGMAN(logINFO) << "Hello world!"; // Một biến staic local được định nghĩa cho mỗi lệnh log

Được mở rộng thành:

for(static int enabler = NOT\_INIT; …;…) FILELog().Get(logINFO) << "Hello world!";

Để đảm bảo thân vòng lặp (lệnh log) thực thi không hơn một lần, dùng thêm lệnh if ở ngoài như sau:

#define LOGMAN(level) if (alreadyLogged = false) \

else for(static TEnabler enabler = NOT\_INIT; !alreadyLogged && …; alreadyLogged = true) FILE\_LOG().Get(logINFO)

Tính năng tối ưu của biên dịch sẽ nhận thấy khi nhánh if không bao giờ được thực thi (luôn false), thì việc kiểm tra sẽ bị bỏ đi hoàn toàn.

Phương thức LogEnable được định nghĩa để phát hiện nếu lệnh log hiện tại có được bật hay chưa:

bool LogEnabled(TEnabler& enabler, const TLogLevel& level, const char\* pFile, int line);

Macro LOGMAN():

#define LOGMAN(level) if (alreadyLogged = false) \

else for(static TEnabler enabler = NOT\_INIT; !alreadyLogged && LogEnabled(enabler, level, \_\_FILE\_\_, \_\_LINE\_\_); \

alreadyLogged = true) FILELog().Get(logINFO)

Triển khai LogEnabled():

bool LogEnabled(TEnabler& enabler, const TLogLevel& level, const char\* pFile, int line)

{

TEnabler safeEnabler = Read(enabler);

if (safeEnabler == NOT\_INIT)

{

Init(enabler, pFile, line);

safeEnabler = Read(enabler);

}

return ((safeEnabler == DEFAULT && FILE\_LOG::ReportingLevel().Get() >= level) || safeEnabler == ENABLED);

}

Phương thức Read(), Write() để ghi, viết giá trị của biến static chia sẻ. Chúng sẽ chèn ranh giới vùng nhớ, đảm bảo trật tự code được toàn vẹn và code làm việc tốt trên hệ máy đa nhân. Để làm được điều này, chúng ta dùng thư viện atomic\_ops của Boehm

Lưu ý rằng khởi tạo của biến staic local đã được thực hiện trước khi vòng lặp thực thi, tránh được race condition, vốn xảy ra khi biến bị cấp phát động.

Having very efficient code when the enabler variable is already initialized (enabler != NOT\_INIT) is very important. For that reason locks were avoided on the normal path (DLCP), and there is a possibility that the Init() function above is actually called multiple times for the same variable. Because the Init() function is called just once for each log statement, the locks can be used inside this function, and serialize these calls, without worrying about effciency.

What Init() does is to add the addresses of the static variable into a global map, having the key the file name and the line number identifying the log statement:

typedef std::pair<std::string, int> TKey;

typedef std::vector<TEnabler\*> TPtrLevels;

typedef std::pair<TPtrLevels, TEnabler> TValue;

typedef std::map<TKey, TValue> TMap;

Please note that the value is not a single address but a vector of addresses. Multiple log statements on a single line can be handled this way:

void foo(int i)

{

LOGMAN(logINFO) << i; LOGMAN(logINFO) << i;

}

Another case when different static variable will be defined on the same line is when used inside template functions/methods:

template <typename T>

void foo(const T& t)

{

LOGMAN(logDEBUG) << t; // "myfile.cpp", line = N

}

For each "T" type a different "static" variable will be generated.

foo(1); // a static variable is defined for "myfile.cpp", line = N

foo(1.2); // another static variable is defined for the same "myfile.cpp", line = N

Before implementing the Init() function, let's look at the implementation of the Set() function. It will allow changing the value of static variables identified by the file name and line number:

void Set(const char\* pFile, int line, TEnabler enabler)

{

boost::mutex::scoped\_lock l(GetMutex());

TKey key = std::make\_pair(std::string(pFile), line);

TMap::iterator i = GetMap().find(key);

if (i != GetMap().end())

{

TValue& value = i->second;

value.second = enabler;

TPtrLevels& levels = value.first;

for (TPtrLevels::iterator j = levels.begin(); j != levels.end(); ++j)

{

Write(enabler, \*\*j);

}

}

else

{

GetMap()[key] = std::make\_pair(TPtrLevels(), enabler);

}

}

The function checks if there are any static variable addresses in the map at the moment (for the file name and line number parameters). If there are any, it will set the value accordingly. If there are no values in the map yet (for example the Set() was called before the log statement at file name/number was first executed), it just saves the value in the map.

This is a possible implementation of the Init() function:

void Init(TEnabler& enabler, const char\* pFile, int line)

{

boost::mutex::scoped\_lock l(GetMutex());

TKey key = std::make\_pair(std::string(pFile), line);

TMap::iterator i = GetMap().find(key);

if (i != GetMap().end())

{

TValue& value = i->second;

value.first.push\_back(&enabler);

Write(value.second, enabler);

}

else

{

Write(DEFAULT, enabler);

TPtrLevels levels;

levels.push\_back(&enabler);

GetMap()[key] = std::make\_pair(levels, DEFAULT);

}

}

The function will insert in the global map the address of the static variable. If the key is not in the map it will use the DEFAULT value. If the key is in the map it will use whatever the value was already set. If Init() is called multiple times with the same static variable address (probably because of a race condition) the code checks if the address is already in the vector, and it ignores the second occurrence. However ignore this situation can be ignored, and accept in this situation having the same address multiple times in the vector. This is a very rare situation and the code will work fine either way anyway.

The Set() and Init() functions are the only ways to change the global map, and both are using the same mutex in to synchronize the access to the map. They are also using the Write() function in order to set the static shared variables, that will insert the appropriate memory barriers.

1. Sử dụng hiệu quả Line-level Logging

The implementation of this framework will be greatly simplified by using the new C++0x standard. Some of the features that are of interest here are implemented in the std::atomic<T> class, so there is no need to use the atomic\_ops library anymore (memory barriers). Also the fact the static variables can be safely initialized dynamically, and the resulting code is thread safe is a big help.

I also compared the previous version of log FILE\_LOG versus LOGMAN to see how big the overhead introduced by this mechanism is. The code executes same disabled log statements 100,000,000 times:

#include "logman.h"

int main(int argc, char\* argv[])

{

LogMan::Set("main3.cpp", 15, LogMan::DISABLED);

const int count = 100000000;

FILE\_LOG(logINFO) << "Start file log";

for (int i = 0; i != count; ++i)

{

FILE\_LOG(logDEBUG) << "this is not executed";

}

FILE\_LOG(logINFO) << "Start log man";

for (int i = 0; i != count; ++i)

{

LOGMAN(logINFO) << "this is disabled too";

}

FILE\_LOG(logINFO) << "End";

return 0;

}

The test produces the following results:

$> log2-test3.1

- 18:34:38.796640 INFO: Start file log

- 18:34:40.199079 INFO: Start log man

- 18:34:42.191984 INFO: End

It took 1.4 sec to execute the first loop and 2.0 sec to execute the second loop. This means the overhead introduced by this mechanism is low, albeit measurable.

While this sample can be used for measuring the overhead, it is actually not a very good example of usage. Instead of enabling/disabling logging statements using Set() functions calls (which are very sensitive to the line number where the statement is located) it is better to directly change the logging level of the log statement itself (and perhaps recompile the code).

A better use of this framework is in conjunction with a means that efficiently enables/disables at runtime individual log statements. This can be done by starting a separate thread that periodically reads (for example every second) a file that specifies the file names, line numbers that identifies the log statements, and the corresponding values. That thread has a very low CPU load, since most of the time it does nothing and it sleeps. For example:

$> cat enabler.txt

file1.cpp,30,1

file2.cpp,49,0

file3.cpp,100,-1

By passing these values to the Set() function you can dynamically adjust the logging level of any individual log statement, and enabling the log statement at file file1.cpp:30 and disabling the log statement in file2.cpp:49.

To help generate this enabler.txt, file the Init() function could be print in the logfile itself the file names and the file numbers for all the local static variable added into the global map. It is then a simple task to create a simple GUI that parses the log file and extracts this information, and then gives us a way to create the enabler.txt file by just checking some buttons.

1. Kết luận

This article presents a simple way to change the logging level of an application at the line-level granularity. The method employed is efficient, and the interface presented to the programmer is easy to use. The upcoming C++0x standard will simplify the implementation and it will make it even simpler.

##### 2nd version

1. Building a lightweight, thread-safe logger that can be redirected to different I/O channels

In this article, I want to discuss the importance of providing a logging component in your software project. From a very general point of view, any software can be developed in a way that allows the running code to provide a trace of its execution in a log file, then depending on the life phase of the product (development or production), it may be released with a different level of logging set. Having a logger becomes crucial when debugging is not possible or is inconvenient. Under some circumstances, debugging is not possible — for example, because the application runs on specific hardware or the problem is completely unclear. In those cases, collection and analysis of logs is crucial to pinpointing a problem.

At Nokia Siemens Networks, I developed the firmware running on NSN's Base Transceiver Station (BTS). A BTS is very complex hardware driven by impressive firmware. When something does not work (especially in a real network), there is no way to check where the problem traces back other than reading the BTS logs, a common logging platform every subsystem shares with all the other components, which provides prints on different severity levels.

A logger should be effective, efficient, and able to provide clear data. Most importantly, it has to guarantee proper functioning even when the whole system crashes. It makes no sense to write logs if the logger stops running at every crash. The logging platform, like the captain of a sinking ship, must "survive" until the end.

In this article, I develop a small logger that provides two levels of logging and three types of log severity (Error, Warning, and Debug). The code has been used in different projects of different sizes, is thread-safe, computationally lightweight, and easily customizable to work with different log output (to allowing remote logging for example).

Dr. Dobb's has previously published another implementation of a good C/C++ logger (see Logging In C++ and Logging In C++: Part 2). There are few important differences between what you'll see here and that solution. First and most important, the logger discussed here will flush the log messages immediately and will not buffer them, which is crucial when a serious destabilizing event occurs. (The implementation provided in Logging In C++: Part 2 provides the ability to enable or disable a single log print at runtime. The way this feature is implemented is very clever and I encourage you all to take a look at both articles.) Another difference in my implementation is that I use only C++ and STL functionalities (no Boost libraries). To understand the code presented, I expect that you'll be familiar with the variadic template concept of the new C++ standard.

1. First Look

What a logger should print depends on the kind of application it's written for. In my projects, I've used it to provide at least: the sequential numbering for the logged lines, date, and execution time elapsed since the beginning of the logging period (expressed in ms, it is also very useful to check the execution time for single operations), and some information about the severity of the logged item.

Making a call to the log function should be easy. The logger should be as little invasive as possible, both for the programmer than for the code. If you take a look at Listing One, you'll see how to invoke the logger function just by using the proper c-style macro:

Listing One: logger.hpp.

#ifndef LOGGER\_HPP

#define LOGGER\_HPP

#include "log.hpp"

static logging::logger< logging::file\_log\_policy > log\_inst( "execution.log" );

#ifdef LOGGING\_LEVEL\_1

#define LOG log\_inst.print< logging::severity\_type::debug >

#define LOG\_ERR log\_inst.print< logging::severity\_type::error >

#define LOG\_WARN log\_inst.print< logging::severity\_type::warning >

#else

#define LOG(...)

#define LOG\_ERR(...)

#define LOG\_WARN(...)

#endif

#ifdef LOGGING\_LEVEL\_2

#define ELOG log\_inst.print< logging::severity\_type::debug >

#define ELOG\_ERR log\_inst.print< logging::severity\_type::error >

#define ELOG\_WARN log\_inst.print< logging::severity\_type::warning >

#else

#define ELOG(...)

#define ELOG\_ERR(...)

#define ELOG\_WARN(...)

#endif

#endif

Line 4 shows a static instantiation of the logger class. The logger is a template that needs to be parameterized with a logging policy. In this case, I'm using a file logging policy, which means that all the output of the logger will be directed to a file on the physical disk.

Lines 6-8 and 15-17 are where the logging macros are defined. As you can see, there are two logging levels (the second level may be enabled to have a more verbose output), and three severity levels. The preferred logging level may be enabled or disabled by toggling the LOGGING\_LEVEL\_x definition. When those macro definitions are absent, a log invocation attempt will be resolved by the preprocessor in the usage of the definitions visible at lines 10-12 and 19-21, which do no logging.

Listing Two shows an example of the logger usage and the relative output:

Listing Two: Example.

#define LOGGING\_LEVEL\_1

#include "logger.hpp"

int main()

{

LOG("Starting the application..");

for( short i = 0 ; i < 3 ; i++ )

{

LOG("The value of 'i' is ", i , ". " , 3 - i - 1 , " more iterations left ");

}

LOG\_WARN("Loop over");

LOG\_ERR("All good things come to an end.. :(");

return 0;

}

Output:

0000000 < Wed Jan 09 12:33:18 2013 - 0000005 > ~ <DEBUG> :Starting the application..

0000001 < Wed Jan 09 12:33:18 2013 - 0000005 > ~ <DEBUG> :The value of 'i' is 0. 2 more iterations left

0000002 < Wed Jan 09 12:33:18 2013 - 0000005 > ~ <DEBUG> :The value of 'i' is 1. 1 more iteration left

0000003 < Wed Jan 09 12:33:18 2013 - 0000005 > ~ <DEBUG> :The value of 'i' is 2. 0 more iterations left

0000004 < Wed Jan 09 12:33:18 2013 - 0000005 > ~ <WARNING> :Loop over

0000005 < Wed Jan 09 12:33:18 2013 - 0000005 > ~ <ERROR> :All good things come to an end.. :(

1. The Logging Policy

Listing Three shows the policy interface and the implementation for the file policy, which uses a C++ ofstream to direct the log messages to the disk.

Listing Three: Log policy.

class log\_policy\_interface

{

public:

virtual void open\_ostream(const std::string& name) = 0;

virtual void close\_ostream() = 0;

virtual void write(const std::string& msg) = 0;

};

class file\_log\_policy : public log\_policy\_interface

{

std::unique\_ptr< std::ofstream > out\_stream;

public:

file\_log\_policy() : out\_stream( new std::ofstream ) {}

void open\_ostream(const std::string& name);

void close\_ostream();

void write(const std::string& msg);

~file\_log\_policy();

};

A logging policy should use a pure abstract class to describe how the interface for the policy works (line 5). This logger "policy" provides functionality related to the log writing, as the logger does not know where its output will be directed: Whether to disk, RAM, or even tunneled by a socket to a remote computer, it just calls a "write" function in the provided policy.

The policy interface simply provides three functions to open and close an output stream and a write operation. For the file logging policy, the code will use the C++ std::ofstream to direct the output to the disk. The complete implementation for the file logging policy is visible in the Listing Four.

Listing Four: The file log policy implementation.

void file\_log\_policy::open\_ostream(const std::string& name)

{

out\_stream->open( name.c\_str(), std::ios\_base::binary|std::ios\_base::out );

if( !out\_stream->is\_open() )

{

throw(std::runtime\_error("LOGGER: Unable to open an output stream"));

}

}

void file\_log\_policy::close\_ostream()

{

if( out\_stream )

{

out\_stream->close();

}

}

void file\_log\_policy::write(const std::string& msg)

{

(\*out\_stream)<<msg<<std::endl;

}

file\_log\_policy::~file\_log\_policy()

{

if( out\_stream )

{

close\_ostream();

}

}

There's nothing special to comment on here. If the stream-opening operation fails, the function throws a std::runtime\_error (line 6). Obviously, a different behavior can be provided.

In this implementation, an exception is thrown exception because the logging facility in my applications is not optional, but crucial (as it is for the firm I work for). In fact, the logging functionalities are seen as features of the software that the customer will use, so if this feature is not working properly, then it is often right to abort the application startup.

The Core Printing Functionality

Every call to a logging macro will be expanded in an invocation of the print function of the logger instance. This function actually does some formatting activities on the log message and use the write function from the used policy to stream out the log string. The Listing Five shows the print function implementation.

Listing Five: The print function.

template< typename log\_policy >

template< severity\_type severity , typename...Args >

void logger< log\_policy >::print( Args...args )

{

write\_mutex.lock();

switch( severity )

{

case severity\_type::debug:

log\_stream<<"<DEBUG> :";

break;

case severity\_type::warning:

log\_stream<<"<WARNING> :";

break;

case severity\_type::error:

log\_stream<<"<ERROR> :";

break;

};

print\_impl( args... );

write\_mutex.unlock();

}

Here, you should be familiar with the variadic functions: print is a variadic, which actually accepts as formal argument the parameter pack Args and one more template argument, severity. Actually, severity is an enum, which can be one of three values as shown in Listing Six:

Listing Six: The severity\_type enum.

enum severity\_type

{

debug = 1,

error,

warning

};

The severity provided as a template parameter is used in the switch statement at line 6 of Listing Five to add the proper severity description to the log message: debug ,warning, or error. The variable log\_stream used in the print function implementation is an attribute of the logger class, actually of a std::stringstream type. The first operation at line 5 is a lock request to write\_mutex, this is needed to ensure thread safety by guaranteeing that no more than one print operation is performed at the same time. This lock is released at line 19 after the operation has finished. Please note that the locking request is a blocking operation if the mutex is already acquired by a different thread, a wait-free version can use a buffering system to store the log messages until the lock is released. The call to print\_impl at line 18 is show next in Listing Seven.

Listing Seven: print\_impl implementation.

template< typename log\_policy >

void logger< log\_policy >::print\_impl()

{

policy->write( get\_logline\_header() + log\_stream.str() );

log\_stream.str("");

}

template< typename log\_policy >

template<typename First, typename...Rest >

void logger< log\_policy >::print\_impl(First parm1, Rest...parm)

{

log\_stream<<parm1;

print\_impl(parm...);

}

If you're not familiar with the variadic functions, you might not be able to understand why this function has two bodies. It's done this way to access the parameters in the variadic parameter pack. If you look at line 12, you'll see that the function is recursive, and at every call the first argument expanded from parm... is used to, let's say, fill parm1, while all the others are filling the argument pack Rest... .

The value of parm1 is stored at line 11, then the recursion happens again with one parameter less. At a certain point, Rest... will be empty, the last value will be printed and the last call to print\_impl performed. If parm... is empty, the recursion is ended by an invocation of the print\_impl version that started at line 1. This print\_impl version just makes a call to the write function of the logging policy, passing a std::string consisting of a log message preceded by the header, which contains data like the timestamp, the log line number, and so on.

1. The Logger Template

Now, let's look at the logger template body (Listing Eight), and at the implementation of the support functions that are used by the print operation.

Listing Eight: Logger template body.

template< typename log\_policy >

class logger

{

unsigned log\_line\_number;

std::string get\_time();

std::string get\_logline\_header();

std::stringstream log\_stream;

log\_policy\* policy;

std::mutex write\_mutex;

//Core printing functionality

void print\_impl();

template<typename First, typename...Rest>

void print\_impl(First parm1, Rest...parm);

public:

logger( const std::string& name );

template< severity\_type severity , typename...Args >

void print( Args...args );

~logger();

};

Lines 4-9 provide a private function and visible attributes such as log\_line\_number just to keep track of the current line number; for each print invocation the number will be increased by one. Get\_time and get\_logline\_header are support functions used to format the log message header, and they are implemented in Listing Nine.

Listing Nine: get\_time and get\_logline\_header implementation.

template< typename log\_policy >

std::string logger< log\_policy >::get\_time()

{

std::string time\_str;

time\_t raw\_time;

time( & raw\_time );

time\_str = ctime( &raw\_time );

//without the newline character

return time\_str.substr( 0 , time\_str.size() - 1 );

}

template< typename log\_policy >

std::string logger< log\_policy >::get\_logline\_header()

{

std::stringstream header;

header.str("");

header.fill('0');

header.width(7);

header << log\_line\_number++ <<" < "<<get\_time()<<" - ";

header.fill('0');

header.width(7);

header <<clock()<<" > ~ ";

return header.str();

}

Going back to Listing Eight, at line 10-13, the declaration of print\_impl is visible, followed at line 15 by the logger-constructor declaration. Lisiting Ten shows the constructor and destructor bodies.

Listing Ten: The logger constructor and destructor.

template< typename log\_policy >

logger< log\_policy >::logger( const std::string& name )

{

log\_line\_number = 0;

policy = new log\_policy;

if( !policy )

{

throw std::runtime\_error("LOGGER: Unable to create the logger instance");

}

policy->open\_ostream( name );

}

template< typename log\_policy >

logger< log\_policy >::~logger()

{

if( policy )

{

policy->close\_ostream();

delete policy;

}

}

Note that if the allocation at line 5 fails and it is not possible to create a log\_policy object, then a std::runtime\_error is thrown. As previously explained, no exception handling is performed here — after all, if this small logger is not able to allocate the amount of memory required by log\_policy, then something very weird is happening.

1. Conclusion

The simple logger described in this article can be used easily in any project to track code behavior during runtime, I think that its Achilles' heel is actually the need to lock the writing mutex in the print function. From one perspective, this is unavoidable because not all operating systems are able to provide atomic stream operations, but it introduces a source of inefficiency.

I think that a good logger should always provide a near constant execution time in any circumstance, which is problematic when threads might have to wait for mutexes to be released. However, in practice, unless numerous threads are logging, the operations are fast enough that there is no significant delay.

The source code for this article was tested with g++ 4.7.2 and requires the C++11 multithread functionality support to work properly; refer to http://tehsausage.com/mingw-std-thread-gcc-4-7 if you get into trouble when compiling this project.

### Bộ quản lý vùng nhớ cho C,C++

Cả C và C++ đều không có bộ dọn vùng nhớ tự động, nên dễ xảy ra rò rỉ. Ngày nay, các lớp thư viện chuẩn của C++ cung cấp nhiều khả năng tự quản lý, thu hồi vùng nhớ (như smart pointer, container STL…), các đối tượng tài nguyên được triển khai theo mô hình RAII tránh bị rò rỉ.

Tuy nhiên, việc quản lý vùng nhớ trong container lại tốn chi phí hiệu suất. Còn nếu triển khai bộ cấp phát trên từng class không phải chiến thuật tốt vì nó cục bộ, không dùng chung được.

Phần này giới thiệu hai mô hình cổ điển là : cấp phát theo dạng Bitmap và Free-list, cùng một số giải thuật

##### Tại sao phải quản lý vùng nhớ

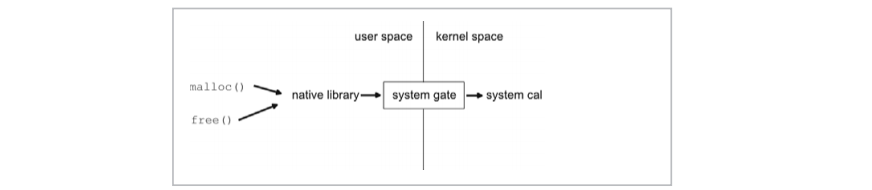
Malloc và new là phương thức cấp phát chung nhất của C,C++, và có sử dụng thread để tăng hiệu suất.

Khi liên tục yêu cầu cấp phát tài nguyên, hệ thống sẽ chạy chậm đi vì sự trung chuyển từ không gian người dùng với kernel.

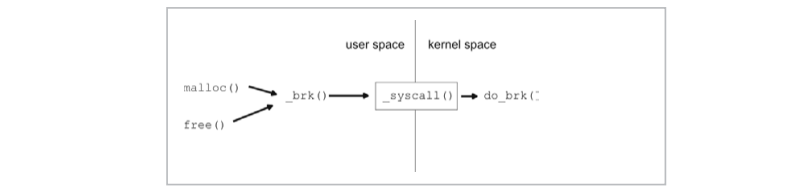
C,C++ không có bộ thu hồi tự động, nên dễ rò rỉ vùng nhớ. Trong chương trình lớn, việc này ảnh hưởng tới hiệu suất đáng kể, khi vùng nhớ RAM trở nên hạn chế thì mức độ truy xuất ổ cứng sẽ căng hơn.

##### Hàm gọi hệ thống và porting

Tùy theo mỗi trình biên dịch hoạt động trên các OS khác nhau, chúng cần dựa trên thư viện gốc của hệ điều hành để cấp phát, thu hồi tài nguyên. Thư viện gốc sẽ thông qua System gate để gọi các System call nằm trong hạt nhân kernel OS.

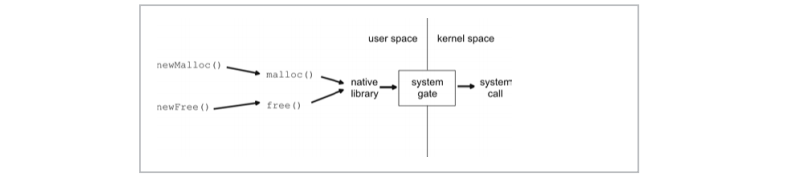


Trên Unix, malloc va free sử dụng phương thức brk() như một wrapper của thư viện gốc, gọi system call.



Còn trên Window sẽ dùng phương thức HEAPXXX để giao tiếp với kernel.

Do đó chúng ta phải thông qua malloc và free để giao tiếp với kernel.



##### Thiết kế

Bộ quản lý vùng nhớ phải đảm bảo các tiêu chí:

1. Tốc độ

Phải nhanh hơn khả năng cấp phát của trình biên dịch. Khi xử lí yêu cầu cấp phát hay thu hồi một cách liên tục, số lượng lớn, không được làm giảm tốc độ chương trình.

1. Hiệu quả

Phải thu hồi hết được tất cả tài nguyên cấp phát cho hệ thống sau khi chương trình kết thúc. Không để rò rỉ tài nguyên, xử lí được các trường hợp ngoại lệ (exception safe).

1. Thuận tiện

Người dùng chỉ cần phải thay đổi ít trong code khi tích hợp bộ quản lý.

1. Tính tương thích

Có khả năng tương thích trên nhiều platform

##### Kỹ thuật triển khai phổ biến

1. Yêu cầu cấp phát một mảng vùng nhớ lớn

Cấp phát một khoanh lớn khi chương trình khởi động, để phân phát cho từng yêu cầu cấp phát sau này.

Ưu điểm là hạn chế system call và cải thiện thời gian thực thi.

1. Tối ưu cho kích thước cấp phát phổ biến

Trong mọi chương trình, sẽ tồn tại kích thước yêu cầu cấp phát phổ biến, bộ quản lý cũng nên tối ưu xử lí các kích thước này tốt hơn.

1. Dùng pool quản lý memory thu hồi

Thu hồi vùng nhớ khi chương trình đang thực thi nên tổ chức thành hàng đợi trong bộ chứa.

Kỹ thuật này làm tối thiểu khả năng rò rỉ, tái sử dụng vùng nhớ thu hồi nên hạn chế system call, cải thiện tốc độ.

##### Đánh giá tốc độ new/delete

Ví dụ về Lớp Complex biểu diễn kiểu số phức gồm phần nguyên và phần phức.

class SoPhuc{

public:

Complex (double a, double b): r (a), c (b) {}

private:

double r; // phan thuc

double c; // phan phuc

};

int main(int argc, char\* argv[]){

SoPhuc \* array[1000];

for (int i = 0;i < 5000; i++)

//Cap phat 1k phan tu

for (int j = 0; j < 1000; j++)

array[j] = new SoPhuc (i, j);

//Thu hoi 1k phan tu

for (int j = 0; j < 1000; j++)

delete array[j];

return 0;

}

Vòng lặp bên trong cùng sẽ cấp phát và thu hồi cho 1k phần tử. 5000 vòng lặp bên ngoài cho 5k lần cấp phát-thu hồi, kết quả là 10 tỷ lần gọi giữa không gian user và kernel.

Biên dịch trên gcc-3.4.6 –Solaris Machine 10 mất trung bình 3.5 giây. Chúng ta cần phải ghi đè toán tử new/delete của đối tượng nếu muốn cải thiện hiệu suất hơn nữa.

##### Quá tải New/Delete:

Quản lý vùng nhớ trên C++ xoay quanh việc quá tải new/delete. Mỗi lớp khác nhau sẽ có chính sách cấp phát khác nhau, nếu không làm vậy, thì toán tử new/delete global phải được override lại.

Quá tải có thể triển khai theo 2 dạng:

void\* operator new(size\_t size);

void operator delete(void\* pointerToDelete);

2 phương thức đầu tiên là thủ tục cơ bản để cấp phát – thu hồi tài nguyên mặc định trên C++, chúng không gọi hàm dựng hay hủy nào cả. Ngược lại, hàm dựng sẽ gọi new, trong khi hàm hủy sẽ gọi delete.

//-Hay có tham số đối tượng quản lý vùng nhớ-

void\* operator new(size\_t size, MemoryManager& memMgr);

void operator delete(void\* pointerToDelete, MemoryManager& memMgr);

Chúng nên sử dụng cách thứ nhất để thuận tiện, người dùng không phải điều chỉnh code nhiều. Do đó, dĩ nhiên vì không dùng tham số nên chúng ta phải khai báo bộ quản lý vùng nhớ dạng biến toàn cục

**Toán tử new, new[ ], delete, và delete[ ] được quá tải như một wrappers:**

MemoryManager gMemoryManager; // Memory Manager, biến toàn cục

void\* operator new (size\_t size) { return gMemoryManager.allocate(size); }

void\* operator new[] (size\_t size) { return gMemoryManager.allocate(size); }

void operator delete (void\* pointerToDelete) { gMemoryManager.free(pointerToDelete); }

void operator delete[] (void\* arrayToDelete) { gMemoryManager.free(arrayToDelete); }

Lưu ý: Ở cách cấp phát cho mảng [], tham số size = số phần tử X kích thước phần tử.

Các phương thức trên sẽ là **Static** method nên không có con trỏ this.

##### Pool Memory

Các cơ chế cấp phát nên triển khai pool khi thu hồi vùng nhớ.

Nguyên tắc cơ bản:

* Duy trì một pool các đối tượng số phức trong memory manager có sẵn, các yêu cầu cấp phát sau này sẽ nằm trong pool.
* Nếu số đối tượng cần tạo quá tải pool, pool phải được mở rộng.
* Đối tượng deleted được trả về cho pool

Pool Memory giúp bộ quản lý không phải tìm kiếm, hiệu suất chương trình tốt hơn.

Pool được tổ chức thành danh sách:

block diagram image

Mỗi block trong pool:

* Chỉ cần con trỏ tới block kế tiếp

Chúng ta tạo cấu trúc **FreeStore**, một wrapper cho con trỏ \*next, **FreeStore\***  là một node của Pool Memory.

**Data structure FreeStore**

struct FreeStore{ FreeStore\* next;};

Đầu tiên triển khai interface iMemoryManager:

**Triển khai interface cơ bản cho Memory manager**

class IMemoryManager {

public:

virtual void\* allocate(size\_t) = 0;

virtual void free(void\*) = 0;

};

Chúng ta cũng khai báo **inline** khi định nghĩa allocate và free để tăng hiệu suất thực thi.

Pool sẽ là DSLK các **FreeStore**, chứa địa chỉ Block kế tiếp. Tất nhiên MemoryManager phải giữ con trỏ **head**, tới phần tử đầu tiên trong pool

Và cần **expandPoolSize**() để mở rộng pool khi cần thiết, **cleanup**() để thu hồi khi chương trình shutdown.

**Thiết kế SoPhuc với FreeStore**

#include <sys/types.h>

class MemoryManager: public IMemoryManager{

private:

FreeStore\* freeStoreHead;

void expandPoolSize ();

void cleanUp ();

FreeStore\* freeStoreHead;

public:

MemoryManager () {

freeStoreHead = 0;

expandPoolSize ();

}

virtual ~MemoryManager () {cleanUp ();}

virtual void\* allocate(size\_t);

virtual void free(void\*);

};

MemoryManager gMemoryManager;

class SoPhuc {

public:

SoPhuc (double a, double b): r (a), c (b) {}

inline void\* operator new(size\_t);

inline void operator delete(void\*);

private:

double r; // phan thuc

double c; // phan phuc

};

Thuật toán khi cấp phát:

1. Nếu **free-store** chưa tạo thì tạo, tới bước 3
2. Nếu **free-store** cạn kiệt, tạo **free-store** mới
3. Trả về phần tử đầu tiên của **free-**store và đánh dấu phần tử kế tiếp cho **free-store** head.

Thuật toán khi thu hồi:

1. Gán \*next trong con trỏ cần xóa tới **free-store head**.
2. Gán **free-store head** cho con trỏ cần xóa

Định nghĩa thủ tục cấp phát – thu hồi cho SoPhuc

inline void\* MemoryManager::allocate(size\_t size) {

if (0 == freeStoreHead)

expandPoolSize ();

FreeStore\* head = freeStoreHead;

freeStoreHead = head->next;

return head;

}

inline void MemoryManager::free(void\* deleted) {

FreeStore\* head = static\_cast <FreeStore\*> (deleted);

head->next = freeStoreHead;

freeStoreHead = head;

}

void\* SoPhuc::operator new (size\_t size) { return gMemoryManager.allocate(size); }

void SoPhuc::operator delete (void\* pointerToDelete) { gMemoryManager.free(pointerToDelete); }

Con trỏ FreeStore\* thực ra đang trỏ tới địa chỉ đối tượng SoPhuc. Tuy nhiên để chắc chắn, kích thước vùng nhớ mỗi block sẽ là kích thước lớn nhất giữa con trỏ FreeStore\* trỏ và đối tượng SoPhuc.

**Thủ tục expand và cleanUp của Pool**

#define POOLSIZE 32

void MemoryManager::expandPoolSize () {

size\_t size = (sizeof(SoPhuc) > sizeof(FreeStore\*))?sizeof(SoPhuc) : sizeof(FreeStore\*);

FreeStore\* head = reinterpret\_cast <FreeStore\*> (new char[size]);

freeStoreHead = head;

for (int i = 0; i < POOLSIZE; i++) {

head->next = reinterpret\_cast <FreeStore\*> (new char [size]);

head = head->next;

**Lưu ý: Các block vẫn dùng new/delete mặc định. Tuy nhiên, bạn cũng có thể dùng malloc/free, tốc độ truy xuất là như nhau**

}

head->next = 0;

}

void MemoryManager::cleanUp() {

FreeStore\* nextPtr = freeStoreHead;

while (nextPtr) {

freeStoreHead = freeStoreHead->next;

delete [] nextPtr; // Nhớ rằng con trỏ next vẫn là kiểu mảng char

nextPtr = freeStoreHead;

}

}

Phương thức **cleanUp** chỉ có ý nghĩa trả lại vùng nhớ cho hệ thống, và cần thuật toán xác định khi nào cần gọi.

Quay trở về với ví dụ ban đầu mất 3.5 giây. Biên dịch sử dụng trình quản lý vùng nhớ Pool, chỉ mất 0.67 giây

Ưu điểm Pool:

* Tái sử dụng vùng nhớ thu hồi, đẩy vào cuối pool, hạn chế sử dụng system call khi khởi tạo mới.
* Tốc độ thực thi nhanh

Khuyết điểm:

* Thiết kế này của chúng ta chỉ dùng đơn thread
* không chống rò rỉ vùng nhớ được nếu đối tượng không bị hủy.

Vấn đề rò rỉ có thể giải quyết bằng cách yêu cầu cấp phát block memory lớn đầu chương trình, và thông qua toán tử new/malloc để chia block này cho từng đối tượng. Theo đó, **cleanUp** không chỉ thu thồi từng đối tượng mà con toàn bộ block.

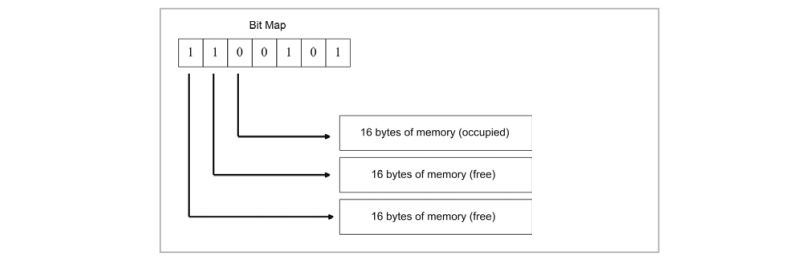
##### Ánh xạ bit (bitmap)

1. Cơ chế

Một kỹ thuật cao cấp cho cơ chế cấp phát đối tượng kích thước cố định là ánh xạ bit. Bộ quản lý chịu trách nhiệm phân chia vùng nhớ khởi tạo đầu chương trình, và thu hồi toàn bộ khi chương trình kết thúc, nên tránh được rò rỉ tài nguyên.

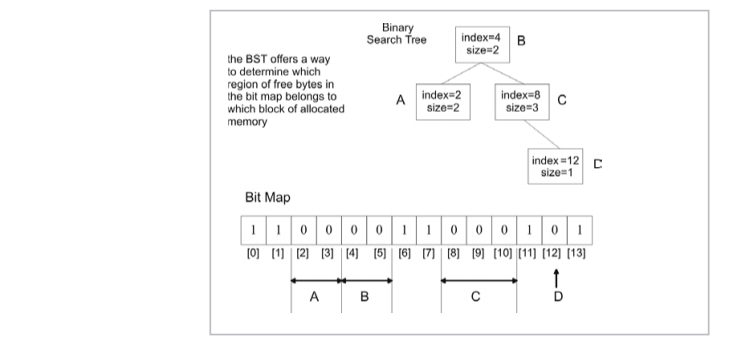
Memory được chia thành các khối kích thước bằng nhau, hay bằng kích thước đối tượng. Ánh xạ bit dùng mảng bitmap giám sát vùng nào đang trống hay đã cấp phát. Mỗi bit trong tập ánh xạ bit đại diện cho một khối nhớ.

Thường chúng ta qui định giá trị bit = 1 là block free, 0 là block cấp phát.



Để bổ khuyết, chúng ta sẽ dùng cấu trúc cây nhị phân cùng bitmap.

* Bitmap: dùng khi cấp phát
* Binary tree: dùng khi thu hồi, xác định bao nhiêu bit trong bitmap cần reset.



**Cấu trúc node Tree:**

* Địa chỉ vùng nhớ heap đã cấp phát
* Bit Index đầu tiên trong bitmap
* Số lượng bit

**Phương thức cấp phát:**

1. Phân giải số byte cấp phát trành số lượng bits cần (tùy theo bạn qui định 1-bit = ? byte).

2. Tìm kiếm thứ tự bit bắt đầu dãy bit trống liên tục đáp ứng số byte.

3. Nếu không thấy trả về NULL, nếu thấy sang bước 4.

4. Clear dãy bit sẽ cấp phát.

5. Tạo Binary Tree entry cho vùng nhớ cấp phát và chèn.

6. Trả về địa chỉ byte đầu tiên.

**Phương thức thu hồi:**

1. Dùng tham số địa chỉ để tìm entry trong Binary Tree.

2. Nếu không thấy thì retrun. Nếu thấy sang bước 3.

3. Dùng thông tin trong entry, set bits tương ứng trong bitmap sang trạng thái free.

1. Vấn đề thừa kế

Toán tử new/delete override trước đó sẽ không làm việc được, mà phải định nghĩa lại cho lớp thừa kế. Có 3 gải pháp:

* Để bộ quản lý memory duy trì 2 pools riêng biệt cho những lớp khác nhau. Tức có 2 phương thức cấp phát/thu hồi và 2 con trỏ pool head. Nhưng khó mở rộng được.
* Duy trì 1 pool cơ sở với kích thước block của lớp dẫn xuất lớn nhất. Phương thức cấp phát luôn trả về kích thước lớp dẫn xuất lớn nhất, bất chấp lớp dẫn xuất nào yêu cầu. Giải pháp này không hữu dụng lắm vì hao vùng nhớ
* Thứ 3 là tạo một bộ quản lý chung cho mọi loại kích thước cấp phát.

1. Tổng kết

Ưu điểm:

* Cơ chế bitmap gọn nhẹ
* Hạn chế phân mảnh vùng nhớ trong khi cấp phát

Khuyết điểm là tập ánh xạ bit :

* không cho biết bao nhiêu vùng nhớ đã cấp phát hay còn trống
* Không biết đối tượng sở hữu ,không có khả năng mở rộng vùng nhớ, chống phân mảnh

Nhiều OS sơ khởi dùng cơ chế này để quản lý vùng nhớ có kích thước cố định và nằm ngoài vùng kiểm soát của kernel. (Không gian địa chỉ kernel trên window là đia chỉ 2GB trở lên, Linux từ 1GB trở lên)

Ví dụ: Nếu 1-bit quản lý 16-bytes, 512 KB bitmap quản lý được 64 MB memory.

Điều này chưa tính tới chi phí của Binary Tree, là tạm chấp nhận được. Duyệt bit có thể tối ưu bằng cách xác định điểm bắt đầu thay vì đi từ đầu. Ví dụ bit thứ 18 sẽ nằm trong byte thứ 3.

##### Danh sách liên kết vùng nhớ

1. Cơ chế

Một chương trình có thể yêu cầu cấp phát theo 8, 16, 24, 32 và 64 bytes. Bộ quản lý sẽ duy trì 6 free-list cho mỗi kích thước tương ứng.

Tương tự như bitmap memory, free-list cũng quản lý memory chunk đầu chương trình, bộ quản lý tự động phân chia chunk cho từng free-list.

Khi yêu cầu cấp phát “m” bytes, bộ quản lý sẽ dò kích thước phù hợp nhất là “n” bytes > = “m”.

1. Bytes bảo vệ (lính canh)

Các block trong free-list có chứa các bytes đặc biệt ở cuối, gọi là bytes bảo vệ trong tiến trình heap memory.

Các hàm memcpy và memset có thể thao tác lấn ra ngoài vùng nhớ cấp phát, vi phạm chính sách vùng nhớ. Nhiều trình biên dịch thêm kí tự đặc biệt như là lính canh (bytes bảo vệ) cho block.

Bất kì truy xuất tới block, sẽ kiểm tra kí tự đặc biệt này. Nếu không hợp lệ tức nội dung block bị thay đổi trái phép.

Trong bài sẽ dùng 2 trong 4-bytes cuối như là lính canh, byte thứ 3 chứa kích thước đối tượng và byte cuối chứa flag (free hay alloc)

1. Thiết kế

* Bộ quản lý duy trì nhiều Memory Pool, mỗi Pool có block size khác nhau: 8, 16, 24, 32, 40…. (bytes). Các block size này được dùng khá thường xuyên. Số Block trong Pool qui định bởi pool size
* Một danh sách quản lý con trỏ được cấp phát . Khi chương trình kết thúc, chỉ cần toàn bộ con trỏ.

Mỗi con trỏ trong free-list quản lý một block memory tùy yêu cầu cấp phát, chúng ta sẽ duyệt danh sách để tìm block phù hợp.

Cấu trúc của block gồm:

* Payload: vùng nhớ cho đối tượng
* 4-byte bảo vệ ở cuối cùng gồm:   
  2 byte-lính canh: là một hằng giá trị ví dụ: 0xdead.  
  1 byte-kích thước block (8, 16, 24 ,32, 40 …)  
  1 byte-trạng thái: 1 là free, 0 là allocated

1. Giải thuật cấp phát

Tham số kích thước sẽ quyết định danh sách Memory chunk nào dùng cho cấp phát.

1. Nếu Pool còn, set byte trạng thái và trả về địa chỉ
2. Nếu Pool trống, khởi tạo Memory chunk mới, cấu hình 4-byte bảo vệ cho từng Block : giá trị byte lính canh, byte kích thước, byte trạng thái về zero,
3. Chèn địa chỉ Memory Chunk vào danh sách quản lý.
4. Trả về địa chỉ tới block đầu tiên trong Pool.
5. Giải thuật thu hồi

Chúng ta có thể kiểm tra địa chỉ thu hồi trong danh sách quản lý, nếu không có thì chẳng làm gì cả.

1. Kiểm tra byte lính canh, nếu không thấy thì trả về
2. Nếu tìm thấy byte lính canh, lấy byte kích thước, set byte-trạng thái về 1
3. Chèn địa chỉ vào Pool tương ứng
4. Tổng kết

Tùy theo chính sách cấp phát, thu hồi mà chúng ta có thể triển khai theo nhiều cách. Ví dụ danh sách quản lý có thể chứa thông tin kích thước block tương ứng với địa chỉ, như vậy chỉ cần 2 byte bảo vệ gồm byte lính canh và byte trạng thái.

* Chuỗi liên hoàn dùng chỉ dùng 1 cấp trúc DSLK đơn giản, nên tốn ít chi phí hơn so với bitmap indexing.
* Dễ mở rộng vùng nhớ, xử lí được nhiều kích thước
* Tính năng byte lính canh cho biết tình trạng vùng nhớ (bị xâm phạm hay không)
* Nếu quản lý vùng nhớ lớn hơn, sẽ tốn quá nhiều chi phí thời gian duyệt List.

##### Triển khai Multithread

Trong môi trường đa luồng, vùng nhớ cấp phát và thu hồi phải đảm bảo các tiến trình được đặt khóa mutex một cách hợp lý.

Bất kì thread nào khi truy xuất tới hai phương thức này, sẽ phải giữ khóa sau khi cấp phát hay thu hồi xong sẽ mở lại khóa cho thread khác. Nếu mutex đang bị khóa, thread sẽ chờ tới khi được mở.

Trên C, chúng ta dùng **pthread** từ thư viện <pthread.h>, code của hai phương thức cấp phát/thu hồi đặt trong **pthread\_mutex\_lock**() và **pthread\_mutex\_unlock**().

Trên lớp MemoryManager quản lý thêm thành viên dữ liệu khóa **pthread\_mutex\_lock**.

Hàm dựng phải gọi **pthread\_mutex\_init** để khởi tạo khóa, tương ứng hàm hủy gọi **pthread\_mutex\_destry** để hủy khóa.

Tuy nhiên, qua khảo sát, phiên bản đa luồng lại chay chậm hơn khá nhiều so với đơn thread.

### XML

##### Giới thiệu

Không thể phủ nhận được vai trò của XML trong quá trình phát triển ứng dụng. Thông thường được dùng để tạo rss, kết quả trả về từ server, lưu trữ dữ liệu...Vậy trong quá trình phát triển chúng ta cần phải thao tác khá là nhiều với những tài liệu XML này. Thì **TinyXML** là một trong những thư viện nhỏ gọn, hiệu quả, hổ trợ ngôn ngữ C++ giúp chúng ta làm được điều đó.

TinyXML là một thư viện hữu ích giúp chúc ta có thể làm việc được với tài liệu XML thông qua mô hình **DOM** (**D**ocument **O**bject **M**odel).

1. Cơ bản về XML

Cũng tương tự HTML là ngôn ngữ đánh dấu văn bản, nhưng XML một ưu điểm nổi là các thẻ đều do người dùng tự định nghĩa, không giống như HTML là phải tuân thủ theo các thẻ được xây dựng sẵn.

Ví dụ về HTML và XML:

**HTML**

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>Tiêu đề </title>
5. </head>
6. <body>
7. <p>Nội dung</p>
8. </body>
9. </html>

**XML**

1. <?xml version="1.0" encoding="utf-8" ?>
2. <Authors>
3. <Author id="01">Kevin La</Author>
4. <Author id="268">Nguyen Nghia</Author>
5. </Authors>

Một ví dụ XML có cấu trúc phúc tạp hơn:

1. <?xml version="1.0"?>
2. <catalog>
3. <book id="bk101">
4. <author>Gambardella, Matthew</author>
5. <title>XML Developer's Guide</title>
6. <genre>Computer</genre>
7. <price>44.95</price>
8. <publish\_date>2000-10-01</publish\_date>
9. <description>An in-depth look at creating applications with XML.</description>
10. </book>
11. <book id="bk102">
12. <author>Ralls, Kim</author>
13. <title>Midnight Rain</title>
14. <genre>Fantasy</genre>
15. <price>5.95</price>
16. <publish\_date>2000-12-16</publish\_date>
17. <description>A former architect battles corporate zombies,
18. an evil sorceress, and her own childhood to become queen
19. of the world.</description>
20. </book>
21. </catalog>

Lưu ý: Tài liệu XML bắt buộc phải có một thẻ root. Thẻ root này sẽ chứa tất cả các thẻ con nằm trong nó. Thẻ root ở hai ví dụ trên là <authors></authors> và <catalog></catalog>.

Để hiểu rõ hơn về tài liệu XML các bạn có thể tham khảo bài viết [Cơ Bản Về XML](http://www.stdio.vn/articles/read/40/co-ban-ve-xml) .

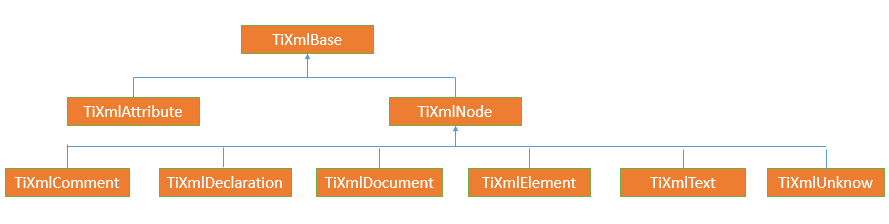
##### Thư viện TinyXML

1. Giới thiệu

TinyXML là một thư viện mã nguồn mở C++, thao tác với tài liệu XML thông qua mô hình DOM (Document Object Model).

Phiên bản mới nhất của thư viện này là **TinyXML 2.6.2**. Các bạn có thể download thư viện tại [**ĐÂY**](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/337/tinyxml_2_6_2.zip).

Sơ đồ phân cấp lớp của thư viện này được mô tả như hình dưới đây:

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/337/ss_1.png)

Với thông tin về mỗi lớp như sau:

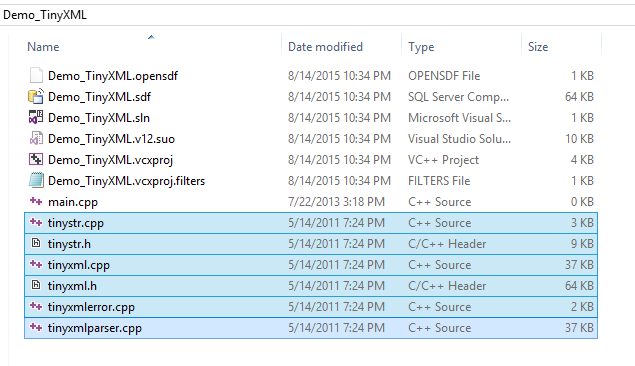
|  |  |
| --- | --- |
| Tên Class | Thông tin về Class |
| TiXmlAttribute | Đại diện cho thuộc tính, là một cặp name-value |
| TiXmlBase | Là class cơ sở của tất cả mọi class trong TinyXML |
| TiXmlComment | Đại diện cho comment trong tài liệu XML |
| TiXmlDeclaration | Đại diện cho những khai báo ở đầu file trong tài liệu XML |
| TiXmlDocument | Đại diện cho một tài liệu XML, chứa tất cả các thông tin về tài liệu này. |
| TiXmlElement | Đại diện cho một node trong tài liệu XML |
| TiXmlNode | Là class cơ sở trong mô hình DOM |
| TiXmlText | Đại diện cho nội dung của một node trong tài liệu XML |

1. Thêm thư viện vào Project

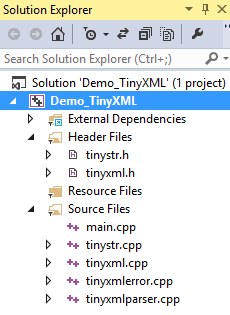
Tạo một **Empty Project Console** C++ với tên Project là **Demo\_TinyXML**. Sau đó download thư viện ở trên và giải nén. Trong thư mục vừa mới giải nén ta chỉ quan tâm tới những file cần thiết là các file:

* tinyxmlerror.cpp
* tinyxmlparser.cpp
* tinystr.cpp
* tinystr.h
* tinyxml.cpp
* tinyxml.h

Copy những file này vào thư mục project của chúng ta.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/337/ss_2.png)

Sau đó thêm những file vào project trên Visual Studio bằng cách chuột phải (right-click) vào project chọn**Add > Existing Item...** và chọn những file trên và nhấn **Add**để kết thúc. Và kết quả sẽ như sau:

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/337/ss_3.png)

Và thêm #include "tinyxml.h" vào đầu những file muốn sử dụng thư viện này:

1. #include "tinyxml.h"
2. Read

Trong bài viết này tôi sẽ thao tác với tài liệu XML có nội dung như sau, việc thao tác với những tài liệu XML khác hoàn toàn tương tự.

1. <?xml version="1.0" encoding="utf-8" ?>
2. <!--Demo read, write, edit XML document using TinyXML library-->
3. <Authors>
4. <Author id="01">
5. <Name>Kevin La</Name>
6. <Age>27</Age>
7. </Author>
8. <Author id="268">
9. <Name>Nguyen Van Nghia</Name>
10. <Age>20</Age>
11. </Author>
12. </Authors>

Tạo một file có tên là **Authors.xm**l có nội dung như trên và copy file này vào thư mục project của chúng ta.

Các bước để đọc một tài liệu XML:

Đọc tài liệu Authors.xml như sau:

**Bước 1**: Load tài liệu lên bộ nhớ.

1. TiXmlDocument doc("Authors.xml");
2. if (!doc.LoadFile())
3. {
4. printf("%s", doc.ErrorDesc());
5. return -1;
6. }

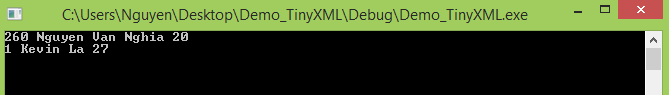
**Bước 2**: Lấy thông tin của node gốc (root element).

1. TiXmlElement\* root = doc.RootElement();

**Bước 3**: Lần lượt truy vấn qua các node con để lấy được thông tin cần thiết:

1. int id;
2. //Tìm phần tử con đầu tiên của node roor
3. TiXmlElement\* child1 = root->FirstChildElement();
4. //Lấy ra id của Author
5. child1->QueryIntAttribute("id", &id);
6. //Truy vấn đến phần tử con đầu tiên của child1
7. TiXmlElement\* name = child1->FirstChildElement();
8. //Truy vấn đến phần tử tiếp theo cùng cấp name
9. TiXmlElement\* age = name->NextSiblingElement();
11. //Xuất id
12. cout << id << " ";
13. //Xuất name
14. cout << name->GetText() << " ";
15. //Xuất age
16. cout << age->GetText() << endl;
18. //Truy xuất đến phần tử tiếp theo cùng cấp với child1
19. TiXmlElement\* child2 = child1->NextSiblingElement();
20. //Lấy ra id của Author
21. child2->QueryIntAttribute("id", &id);
22. //Truy vấn đến phần tử con đầu tiên của child2
23. name = child2->FirstChildElement();
24. //Truy vấn đến phần tử tiếp theo cùng cấp name
25. age = name->NextSiblingElement();
27. //Xuất thông tin của author
28. cout << id << " ";
29. cout << name->GetText() << " ";
30. cout << age->GetText() << endl;

Kết quả:

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/337/ss_4.png)

Giả sử có rất nhiều author, thì giải pháp tốt nhất là chúng ta sẽ sử dụng vòng lặp để duyệt qua tất cả các node:

1. TiXmlElement\* author = nullptr;
2. int id;

5. for (author = root->FirstChildElement(); author != NULL; author = author->NextSiblingElement())
6. {
7. author->QueryIntAttribute("id", &id);
8. TiXmlElement\* name = author->FirstChildElement();
9. TiXmlElement\* age = name->NextSiblingElement();
10. cout << id << " ";
11. cout << name->GetText() << " ";
12. cout << age->GetText() << endl;
13. }

Các bạn có thể tìm hiểu rõ hơn về các phương thức FirstChild,LastChild, PreviousSibling, NextSibling, NextSiblingElement để có thể truy vấn trong tài liệu XML một cách hiệu quả.

1. Write

Ở phần này tôi sẽ tạo lại một tài liệu XML có nội dung như **Authors.xml** nhưng lưu lại với tên là**Author\_Write.xml**.

Tài liệu XML là một mô hình phân cấp. Nghĩa là một một node trong tài liệu XML đồng thời là cha của node này và là con của node kia. Ví dụ trong tài liệu**Authors.xml** thì Authors là cha của các Author, các Author là anh chị em với nhau. Mỗi Author đều có thuộc tính id và có hai con là Name và Age. Cách tạo tài liệu XML cũng dựa trên điều này. Tạo ra các TiXmlElement và dùng phương thức có prototype như sau để thể hiện mối quan hệ trên:

1. TiXmlNode\* TiXmlNode::LinkEndChild (TiXmlNode \* addThis)

Tạo một đối tượng TiXmlDocument và thêm các node TiXmlDeclaration, TiXmlComment, đối tượng root TiXmlElement vào tài liệu này:

1. //Tạo đối tượng quản lý tài liệu XML
2. TiXmlDocument doc;
4. //Tạo chỉ thị của tài liệu XML bao gồm version, endcoding sau đó thêm dec vào tài liệu
5. TiXmlDeclaration \*dec = new TiXmlDeclaration("1.0", "utf-8", "");
6. //Thêm dec vào tài liệu
7. doc.LinkEndChild(dec);
9. //Tạo comment và thêm comment vào tài liệu
10. TiXmlComment \*cmt = new TiXmlComment("Demo read, write, edit XML document using TinyXML library");
11. doc.LinkEndChild(cmt);
13. //Tạo node root và thêm root vào tài liệu
14. TiXmlElement\* root = new TiXmlElement("Authors");
15. doc.LinkEndChild(root);

Tạo hai đối tượng Author và thêm vào node root

1. //Tạo Author1
2. TiXmlElement\* author1 = new TiXmlElement("Author");
3. //Set id cho author1
4. author1->SetAttribute("id", 1);
5. //Thêm author1 vào root
6. root->LinkEndChild(author1);

9. //Tạo Author2
10. TiXmlElement\* author2 = new TiXmlElement("Author");
11. //Set id cho author2
12. author2->SetAttribute("id", 268);
13. //Thêm author2 vào root
14. root->LinkEndChild(author2);

Với mỗi Author ta sẽ tạo hai node là Name, Age và set nội dung cho chúng, sau đó add vào node author tương ứng.

**Author1**

1. //Tạo node Name
2. TiXmlElement\* author1\_name = new TiXmlElement("Name");
3. author1->LinkEndChild(author1\_name);
4. TiXmlText\* name\_content\_1 = new TiXmlText("Kevin La");
5. author1\_name->LinkEndChild(name\_content\_1);
7. //Tạo node Age
8. TiXmlElement\* author1\_age = new TiXmlElement("Age");
9. author1->LinkEndChild(author1\_age);
10. TiXmlText\* age\_content\_1 = new TiXmlText("27");
11. author1\_age->LinkEndChild(age\_content\_1);

**Author2**

1. //Tạo node Name
2. TiXmlElement\* author2\_name = new TiXmlElement("Name");
3. author2->LinkEndChild(author2\_name);
4. TiXmlText\* name\_content\_2 = new TiXmlText("Nguyen Van Nghia");
5. author2\_name->LinkEndChild(name\_content\_2);
7. //Tạo node Age
8. TiXmlElement\* author2\_age = new TiXmlElement("Age");
9. author2->LinkEndChild(author2\_age);
10. TiXmlText\* age\_content\_2 = new TiXmlText("20");
11. author2\_age->LinkEndChild(age\_content\_2);

Lưu tài liệu XML.

1. doc.SaveFile("Authors\_Write.xml");

Dưới đây là toàn bộ souce code tại ra file **Authors\_Write.xml**

1. //Tạo đối tượng quản lý tài liệu XML
2. TiXmlDocument doc;
4. //Tạo chỉ thị của tài liệu XML bao gồm version, endcoding sau đó thêm dec vào tài liệu
5. TiXmlDeclaration \*dec = new TiXmlDeclaration("1.0", "utf-8", "");
6. //Thêm dec vào tài liệu
7. doc.LinkEndChild(dec);
9. //Tạo comment và thêm comment vào tài liệu
10. TiXmlComment \*cmt = new TiXmlComment("Demo read, write, edit XML document using TinyXML library");
11. doc.LinkEndChild(cmt);
13. //Tạo node root và thêm root vào tài liệu
14. TiXmlElement\* root = new TiXmlElement("Authors");
15. doc.LinkEndChild(root);

18. //Tạo Author1
19. TiXmlElement\* author1 = new TiXmlElement("Author");
20. //Set id cho author1
21. author1->SetAttribute("id", 1);
22. //Thêm author1 vào root
23. root->LinkEndChild(author1);

26. //Tạo Author2
27. TiXmlElement\* author2 = new TiXmlElement("Author");
28. //Set id cho author2
29. author2->SetAttribute("id", 268);
30. //Thêm author2 vào root
31. root->LinkEndChild(author2);
33. //Tạo node Name
34. TiXmlElement\* author1\_name = new TiXmlElement("Name");
35. author1->LinkEndChild(author1\_name);
36. TiXmlText\* name\_content\_1 = new TiXmlText("Kevin La");
37. author1\_name->LinkEndChild(name\_content\_1);
39. //Tạo node Age
40. TiXmlElement\* author1\_age = new TiXmlElement("Age");
41. author1->LinkEndChild(author1\_age);
42. TiXmlText\* age\_content\_1 = new TiXmlText("27");
43. author1\_age->LinkEndChild(age\_content\_1);

46. //Tạo node Name
47. TiXmlElement\* author2\_name = new TiXmlElement("Name");
48. author2->LinkEndChild(author2\_name);
49. TiXmlText\* name\_content\_2 = new TiXmlText("Nguyen Van Nghia");
50. author2\_name->LinkEndChild(name\_content\_2);
52. //Tạo node Age
53. TiXmlElement\* author2\_age = new TiXmlElement("Age");
54. author2->LinkEndChild(author2\_age);
55. TiXmlText\* age\_content\_2 = new TiXmlText("20");
56. author2\_age->LinkEndChild(age\_content\_2);
58. doc.SaveFile("Authors\_Write.xml");

Build và chạy chương trình, sau đó mở thư mục chứa project các bạn sẽ thấy file **Authors\_Write.xml**được tạo ra với nội dung giống như file **Authors.xml.**

1. Tham khảo

http://www.grinninglizard.com/tinyxmldocs/annotated.html - 15/08/2015

### MySQL++

##### Giới thiệu về cầu nối MySQL C++

MySQL Connector/C++ là cầu nối CSDL cho C++, giúp phát triển ứng dụng trên MySQL server.

1. Những lợi ích từ cầu nối MySQL C++:

* API tương thích với MySQL C API (thư viện MySQL client)
* Thuần C++
* Hỗ trợ JDBC 4.0, chuẩn công nhiệp API
* Hỗ trợ OOP
* Rút ngắn thời gian phát triển
* Bản quyền GPL với trường hợp ngoại lệ bản quyền FLOSS
* Cho phép ứng dụng thương mại hóa

1. Tính tương thích với JDBC

Cầu nối tướng thích với chuẩn API JDBC 4.0, dù vậy không triển khai hết API JDBC, chỉ bao gồm các lớp sau:

* **Connection**
* **DatabaseMetaData**
* **Driver**
* **PreparedStatement**
* **ResultSet**
* **ResultSetMetaData**
* **Savepoint**
* **Statement**

JDBC 4.0 API định nghĩa tới 450 phương thức lớp. Cầu nối C++ chỉ gồm 80% mà thôi.

Hỗ trợ trên nhiều platform: window, linux

1. MySQL Connector/C++ requires:

* MySQL 5.1 hay cao hơn
* Microsoft Visual Studio trên Windows

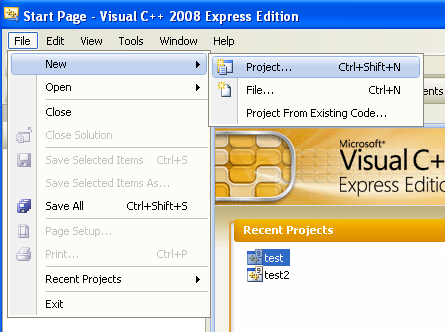
##### Cấu hình MySQL C++ với Visual studio

Thư viện tĩnh gồm file mysqlcppconn-static.lib. Bạn cần liên kết file này với project, đồng thời link cùng libmysql.dll và libmysql.lib. Chương trình sẽ cần libmysql.dll để thực thi.

Thư viện tĩnh gồm source file mysqlcppconn.dll, kèm theo thư viện khai báo trong file mysqlcppconn.lib. Chương trình khi chạy sẽ cần mysqlcppconn.dll và libmysql.dll để thực thi

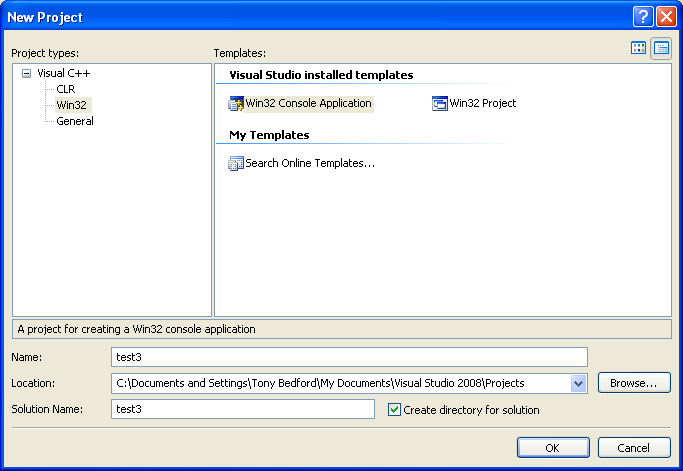
1. Khai đường dẫn thư viện
2. Select **File**, **New**, **Project** from the main menu.

**Figure 5.1 Creating a New Project**



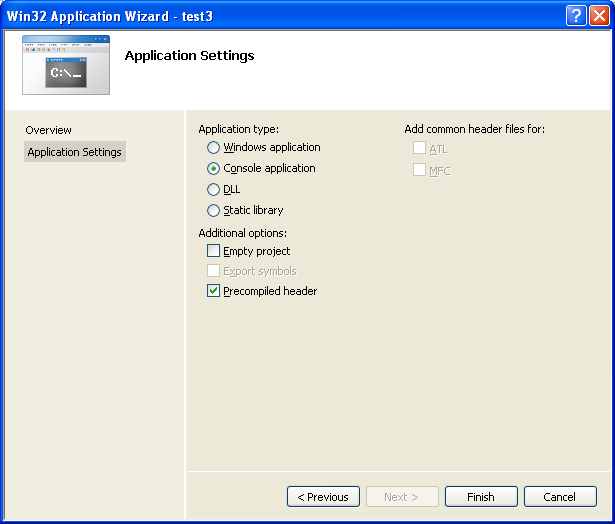
1. In the wizard, select **Visual C++**, **Win32**. From **Visual Studio Installed Templates**, select the application type **Win32 Console Application**. Enter a name for the application, then click OK, to move to the Win32 Application Wizard.

**Figure 5.2 The New Project Dialog Box**



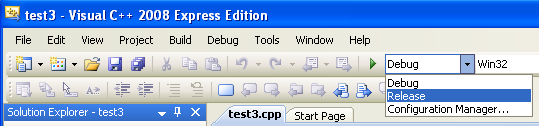
1. In the Win32 Application Wizard, click **Application Settings** and ensure the defaults are selected. The radio button **Console application** and the check box **Precompiled headers** are selected. Click Finish to close the wizard.

**Figure 5.3 The Win32 Application Wizard**



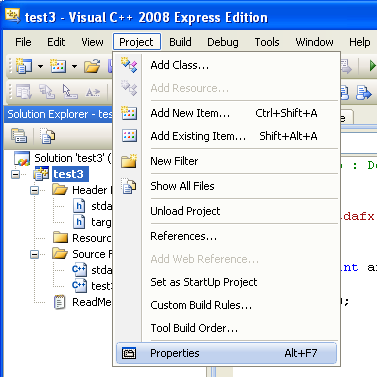
1. From the drop down list box on the toolbar, change from the default **Debug** build to the **Release** build.

**Figure 5.4 Selecting the Release Build**



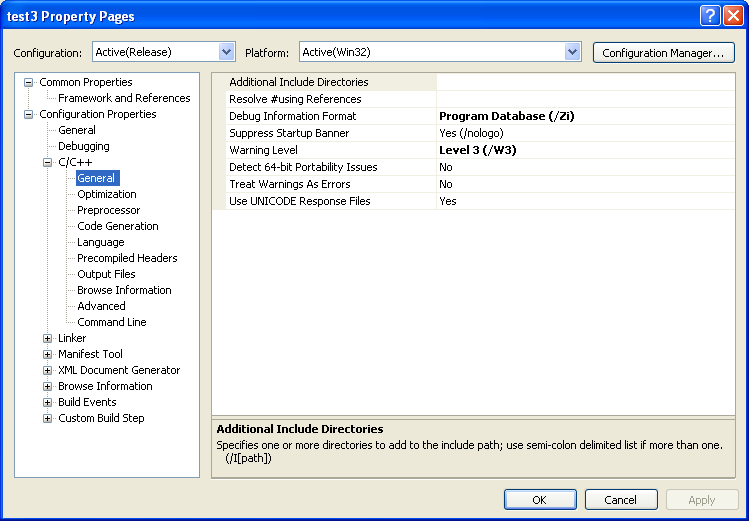
1. From the main menu select **Project**, **Properties**. This can also be accessed using the hot key ALT + F7.

**Figure 5.5 Selecting Project Properties from the Main Menu**



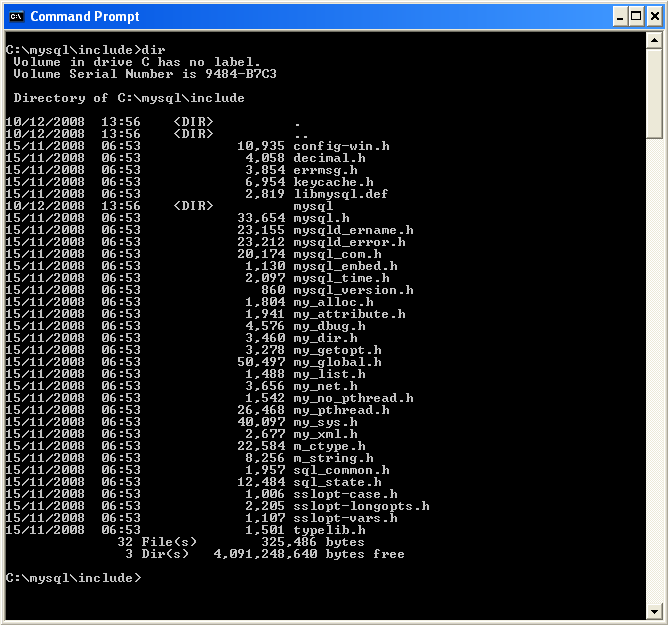
1. Under **Configuration Properties**, open the tree view.
2. Select **C++**, **General** in the tree view.

**Figure 5.6 Setting Properties**



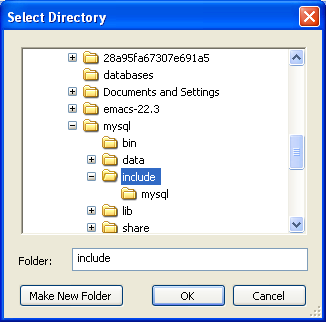
1. Ensure that Visual Studio can find the MySQL include directory. This directory includes header files that can optionally be installed when installing MySQL Server.

**Figure 5.7 MySQL Include Directory**



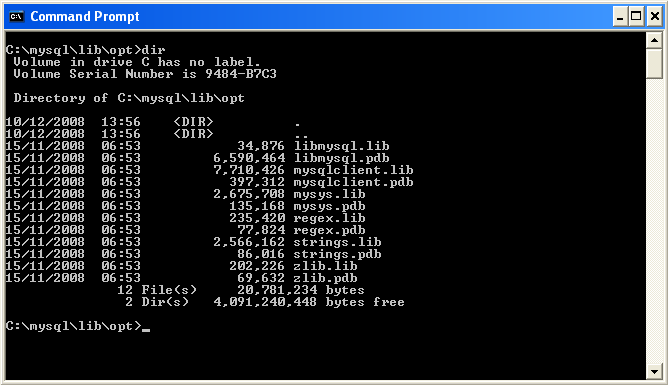
1. In the **Additional Include Directories** text field, add the MySQL include/ directory.

**Figure 5.8 Select Directory Dialog**



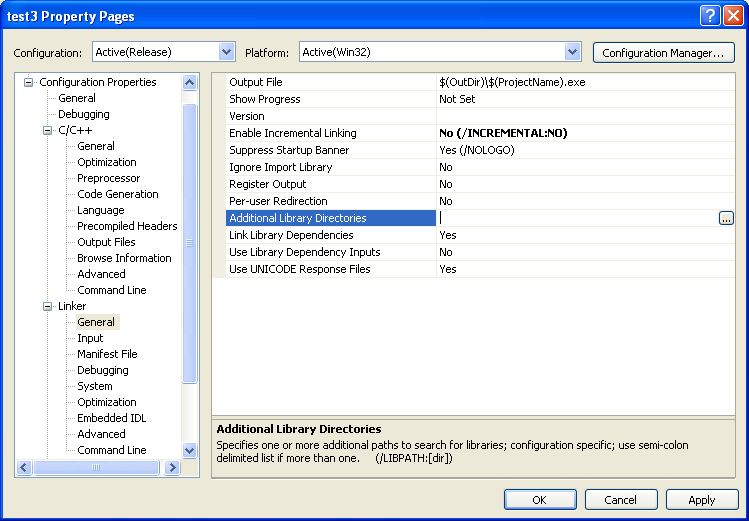
1. Also set the location of additional libraries that Visual Studio needs to build the application. These are located in the MySQL lib/opt directory, a subdirectory of the MySQL Server installation directory.

**Figure 5.9 Typical Contents of MySQL lib/opt Directory**



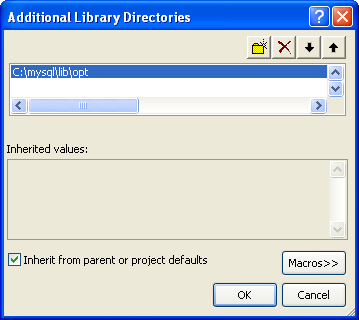
1. In the tree view, open **Linker**, **General**, **Additional Library Directories**.

**Figure 5.10 Additional Library Directories**



1. Add the lib/opt directory into the **Additional Library Directories** text field. This enables the library file libmysql.lib to be found.

**Figure 5.11 Additional Library Directories Dialog**

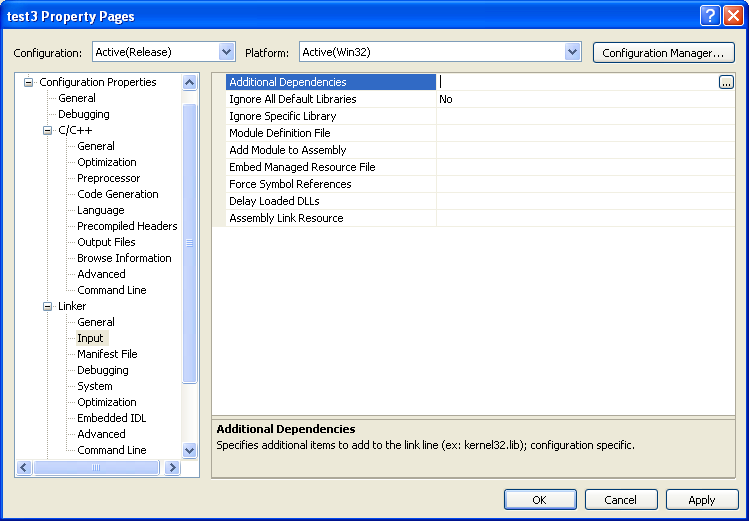


1. Static Build (khuyến cáo không nên vì static lib có thể không tương thích với phiên bản VS complier C++)

The remaining steps depend on whether you are building an application to use the Connector/C++ static or dynamic library. If you are building your application to use the dynamic library, see [Dynamic Build](https://dev.mysql.com/doc/connector-cpp/en/connector-cpp-apps-windows-visual-studio.html#connector-cpp-application-build-dynamic). If you are building your application to use the static library, carry out the following steps:

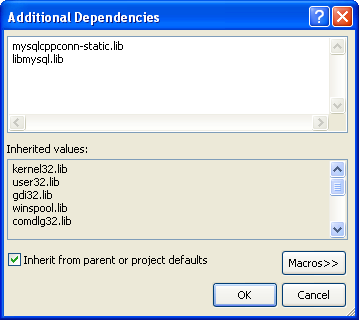
1. Open **Linker**, **Input**, **Additional Dependencies**.

**Figure 5.12 Additional Dependencies**



1. Enter mysqlcppconn-static.lib and libmysql.lib.

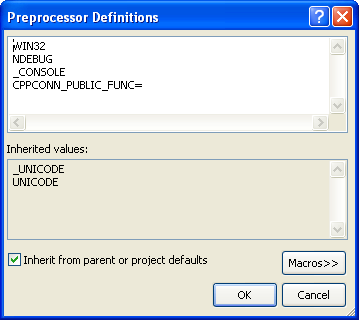
**Figure 5.13 Adding Additional Dependencies**



1. By default CPPCONN\_PUBLIC\_FUNC is defined to declare functions to be compatible with an application that calls a DLL. If you are building an application to call the static library, ensure that function prototypes are compatible with this. In this case, define CPPCONN\_PUBLIC\_FUNC to be an empty string, so that functions are declared with the correct prototype.

In the **Project**, **Properties** tree view, under **C++**, **Preprocessor**, enter CPPCONN\_PUBLIC\_FUNC= into the **Preprocessor Definitions** text field.

**Figure 5.14 Setting the CPPCONN\_PUBLIC\_FUNC Define**



**Note**

Make sure you enter CPPCONN\_PUBLIC\_FUNC= and not CPPCONN\_PUBLIC\_FUNC, so that it is defined as an empty string.

1. Dynamic build

Theo 2 bước sau:

1. Khai báo mysqlcppconn.lib trong Linker -> Input-> **Additional Dependencies** .
2. mysqlcppconn.dll phải nằm chung với thư mục chương trình chạy.

##### Bắt đầu chương trình mới

1. Khai báo thư viện

Bạn khai báo các thư viện cần thiết như bên dưới:

#include "mysql\_connection.h"

#include <cppconn/driver.h>

#include <cppconn/exception.h>

#include <cppconn/resultset.h>

#include <cppconn/statement.h>

1. Khai báo Driver cho MySQL

Khai báo đối tượng sql::Driver

sql::Driver \*driver;

Khởi tạo Driver cho MySQL

driver = sql::get\_driver\_instance();

1. Kết nối qua sql::Connection

Khai báo đối tượng Connection

sql::Connection \*con;

Khởi tạo kết nối tới MySQL Server dùng phương thức sql::Driver::connect(), gồm 3 tham số:

* Địa chỉ host và port (3306)
* User name
* password

con = driver->connect("tcp://127.0.0.1:3306", "user", "password");

Hủy đối tượng kết nối khi không dùng nữa

delete con;

Phải đảm bảo đối tượng sql::Connection hủy khi không dùng nữa, nhưng không nên hủy đối tượng Driver, cầu nối sẽ tự lo việc đó.

**Phương thức get\_mysql\_driver\_instance() sẽ gọi get\_driver\_instance()- vốn không dùng thread-safe. Để tránh các phương thức này xuất hiện trên nhiều thread một lúc, và việc gọi với một mutex để tránh thực đồng thời nhiều threads**

Phương thức để kiểm tra trạng thái kết nối

sql::Connection::isValid()

Phương thức kết nối lại

sql::Connection::reconnect()

1. Thực thi truy vấn sql::Statement

Khởi tạo đối tượng sql::Satement với **sql::Connection::createStatement()**

sql::Statement \*stmt = con->createStatement();

Dùng phương thức **sql::Satement::execute()** để thực thi SQL command.

stmt->execute("USE " EXAMPLE\_DB);

stmt->execute("DROP TABLE IF EXISTS test");

stmt->execute("CREATE TABLE test(id INT, label CHAR(1))");

stmt->execute("INSERT INTO test(id, label) VALUES (1, 'a')");

1. Chiết xuất dữ liệu với sql::ResultSet

Khai báo đối tượng sql::Resultset

sql::ResultSet \*res;

Để lấy kết quả trả về từ truy vấn, bạn phải dùng phương thức **sql::Statement::executeQuery()** hay **sql::PreparedStatement::executeQuery()**

res = stmt->executeQuery("SELECT id, label FROM test ORDER BY id ASC");

Dùng vòng lặp while với phương thức **sql::ResultSet::next()**

* Bạn có thể dùng thứ tự cột trong cấu trúc bảng CSDL để lấy dữ liệu, thứ tự cột luôn bắt đầu từ 1.  
   Ví dụ: res->getInt(1) trả về thông tin là kiểu integer của cột thứ nhất.
* Nhà sản xuất khuyên bạn dùng tên cột cho thuận tiện. Ví dụ: res->getString(“label”) trả về chuỗi dữ liệu từ cột label

while (res->next()) {

cout << "id = " << res->getInt(1); // getInt(1) returns the first column

cout << ", label = '" << res->getString("label") << "'" << endl;

}

****

**Nhớ hủy sql::Statement và sql::Connection, sql::ResulSet khi không dùng.**

delete res;

delete stmt;

delete con;

1. Insert, delete, update với sql::PreparedStatment

Các bước khởi tạo giống như trên

// ...

sql::Connection \*con;

sql::PreparedStatement \*prep\_stmt

// ...

Khởi tạo sql::PreparedStatement với sql::Connection::prepareStatement(“SQL command text”)

prep\_stmt = con->prepareStatement("INSERT INTO test(id, label) VALUES (?, ?)");

Dùng các phương thức sql::PreparedStatment::setInt(), setString(), setDouble()… cho các kiểu dữ liệu tương ứng trong cột CSDL. Đối số gồm số thứ tự cột và kèm theo dữ liệu nhập vào.

prep\_stmt->setInt(1, 1);

prep\_stmt->setString(2, "a");

prep\_stmt->execute();

Dùng sql::PreparedStatment::execute() cho mỗi thông tin cập nhật.

prep\_stmt->setInt(1, 2);

prep\_stmt->setString(2, "b");

prep\_stmt->execute();

Hủy đối tượng khi không dùng:

delete prep\_stmt;

delete con;

1. Xử lí ngoại lệ với sql::SQLException

Chúng ta để khối code trong khối try{} catch() {}

try {

/\* Create a connection \*/

…

/\* Connect to the MySQL test database \*/

…

/\* Access column data by alias or column name \*/

…

/\* Access column fata by numeric offset, 1 is the first column \*/

…

} catch (sql::SQLException &e){

Tham số catch() sẽ là đối tượng tham chiếu kiểu sql::SQL Exception, vốn được thừa kế từ lớp exception. Dùng phương thức sql::SQLException::

* what() để lấy thông tin lỗi
* getErrorCode() để lấy mã lỗi
* getSQLState() trả về chuỗi thông tin trạng thái

} catch (sql::SQLException &e){

cout << "# ERR: SQLException in " << \_\_FILE\_\_;

cout << "(" << \_\_FUNCTION\_\_ << ") on line " »

<< \_\_LINE\_\_ << endl;

cout << "# ERR: " << e.what();

cout << " (MySQL error code: " << e.getErrorCode();

cout << ", SQLState: " << e.getSQLState() << " )" << endl;

}

### Mã Hóa

1. Giới thiệu

Trong cuộc sống ngày nay, mạng internet có lẻ đã trở thành một thứ không thể thiếu đối với chúng ta. Mạng internet giúp chúng ta tra cứu thông tin, làm việc, học tập, kết nối với mọi người. Nếu thiếu nó thì có lẻ thế giới này sẽ trở lên lạc hậu hơn rất nhiều so với bây giờ.

Vậy có bao giờ bạn nghĩ đến thực chất internet là gì, mà tại sao nó lại quan trọng đến vậy? Về cơ bản, ta có thể coi internet như là một môi trường trung gian để trao đổi THÔNG TIN, mọi hoạt động trên internet đềuHẦU NHƯ CHỈ xoay quanh việc tiếp nhận, trao đổi và truyền nhận THÔNG TIN. Qua đó ta có thể thấy được thông tin quan trọng như thế nào.

*Thật sự thông tin còn quan trọng hơn những gì bạn tưởng. Một số thông tin có thể quan trọng hơn bất cứ thứ gì hữu hình trên thế giới này, nó có thể đáng giá hàng trăm tỷ, hàng tỷ tỷ đô la, hoặc có thể là vô giá.*

Vì thông tin là quan trọng, nên chúng ta sẽ cần phải có những phương pháp để bảo vệ nó.

1. Mã hóa là gì?

Mạng máy tính là một môi trường mở, những thông tin bạn gửi lên internet hoặc nhận về internet đều có thể bị nghe trộm. Do đó việc bảo mật những thông tin này là cần thiết, và một trong những cách để bảo mật thông tin hữu hiệu nhất hiện nay là mã hóa.

Có thể bạn cảm thấy xa lạ với khái niệm mã hóa. Nhưng nó là một thứ cực kì quan trọng, và hiện hữu ở rất nhiều nơi trong đời sống hàng ngày của chúng ta. Để biết được nó quan trọng như thế nào, và được sử dụng rộng rãi ra sao, hãy tưởng tượng: Nếu không có mã hóa, hệ thống ATM sẽ không tồn tại, sẽ không tồn tại chuỗi hệ thống ngân hàng, sẽ không có giao dịch mua bán online, internet sẽ không phát triển… Và nếu không có mã hóa, bạn sẽ không thể ngồi đây, ngay giờ này và đọc bài viết này, bởi không có nó thì internet sẽ không thể phát triển được như ngày nay.

*Giả sử con người chưa biết đến mã hóa, một hacker chỉ cần làm một thiết bị lắng nghe và chuyển đổi các gói packet được truyền đi trong mạng và gắng thiết bị này vào cáp mạng của một máy ATM nào đó. Khi bạn hoặc ai đó sử dụng máy ATM này để chuyển tiền, thiết bị này chỉ việc phân tích các packet chứa thông tin giao dịch được truyền đi, và chuyển đổi số tài khoản mà bạn muốn gửi thành số tài khoản của anh ta. Và thế là, tất cả số tiền giao dịch của máy ATM đó sẽ chảy vào túi của anh ta, trong khi anh ta chỉ việc ngồi máy lạnh sơi cà phê! Nếu vậy thì làm sao máy ATM có thể tồn tại? Vì nó quá thiếu an toàn nên sẽ không ai dùng đến nó, và sẽ không ai tạo ra nó.*

Ta có thể dễ dàng khái quát, mã hóa là một phương pháp bảo vệ thông tin, bằng cách chuyển đổi thông tin từ dạng rõ (Thông tin có thể dễ dàng đọc hiểu được) sang dạng mờ (Thông tin đã bị che đi, nên không thể đọc hiểu được. Để đọc được ta cần phải giải mã nó). Nó giúp ta có thể bảo vệ thông tin, để những kẻ đánh cắp thông tin, dù có được thông tin của chúng ta, cũng không thể hiểu được nội dung của nó.

*Lấy ví dụ, khi bạn muốn gửi thư cho bạn mình, và trong đó chứa những thông tin quan trọng mà bạn không muốn ai biết (Giả sử nội dung ban đầu là "Ngày mai xăng tăng giá đó"). Do đó bạn muốn bảo mật thông tin này, để dù có người cố tình đọc trộm nội dung thì cũng không thể hiểu, thì bạn sẽ mã hóa nó (Giả sử bạn mã hóa thành "fd%$23fDd432FDs4#@Vdserf3%$3"). Xong khi đưa đến bạn mình, bạn sẽ bày cho họ cách giải mã để họ có thể hiểu được nội dung thư.*

1. Thuật toán mã hóa

Thuật toán mã hóa là một thuật toán nhằm mã hóa thông tin của chúng ta, biến đổi thông tin từ dạng rõ sang dạng mờ, để ngăn cản việc đọc trộm nội dung của thông tin (Dù hacker có được thông tin đó cũng không hiểu nội dung chứa trong nó là gì).

Thông thường các thuật toán sử dụng một hoặc nhiều key (Một chuỗi chìa khóa để mã hóa và giải mã thông tin) để mã hóa và giải mã (Ngoại trừ những thuật toán cổ điển). Bạn có thể coi key này như một cái password để có thể đọc được nội dung mã hóa. Người gửi sẽ dùng key mã hóa để mã hóa thông tin sang dạng mờ, và người nhận sẽ sử dụng key giải mã để giải mã thông tin sang dạng rõ. Chỉ những người nào có key giải mã mới có thể đọc được nội dung.

Nhưng đôi khi "kẻ thứ ba" (hacker) không có key giải mã vẫn có thể đọc được thông tin, bằng cách phá vỡ thuật toán. Và có một nguyên tắc là bất kì thuật toán mã hóa nào cũng đều có thể bị phá vỡ. Do đó không có bất kì thuật toán mã hóa nào được coi là an toàn mãi mãi. Độ an toàn của thuật toán được dựa vào nguyên tắc:

* Nếu chi phí để giải mã một khối lượng thông tin lớn hơn giá trị của khối lượng thông tin đó thì thuật toán đó được tạm coi là an toàn. (Không ai lại đi bỏ ra 50 năm để giải mã một thông tin mà chỉ mang lại cho anh ta 1000 đô).
* Nếu thời gian để phá vỡ một thuật toán là quá lớn (giả sử lớn hơn 100 năm, 1000 năm) thì thuật toán được tạm coi là an toàn.

1. Phân loại các phương pháp mã hóa

Có rất nhiều loại phương pháp mã hóa khác nhau đã ra đời. Mỗi loại có những ưu và nhược điểm riêng. Ta có thể phân chia các phương pháp mã hóa thành 4 loại chính:

* Mã hóa cổ điển
* Mã hóa một chiều
* Mã hóa đối xứng
* Mã hóa bất đối xứng

1. Mã hóa cổ điển

Đây là phương pháp mã hóa đầu tiên, và cố xưa nhất, và hiện nay rất ít được dùng đến so với các phương pháp khác. Ý tưởng của phương pháp này rất đơn giản, bên A mã hóa thông tin bằng thuật toán mã hóa cổ điển, và bên B giải mã thông tin, dựa vào thuật toán của bên A, mà không dùng đến bất kì key nào. Do đó, độ an toàn của thuật toán sẽ chỉ dựa vào độ bí mật của thuật toán, vì chỉ cần ta biết được thuật toán mã hóa, ta sẽ có thể giải mã được thông tin.

*Một ví dụ về phương pháp mã hóa cổ điển: Giả sử bạn mã hóa bằng cách thay đổi một kí tự trong chuỗi cần mã hóa thành kí tự liền kề (“Di hoc ve” thành “Ek ipd xg”). Thì bất cứ người nào, chỉ cần biết cách bạn mã hóa, đều có thể giải mã được.*

1. Mã hóa một chiều

Đôi khi ta chỉ cần mã hóa thông tin chứ không cần giải mã thông tin, khi đó ta sẽ dùng đến phương pháp mã hóa một chiều (Chỉ có thể mã hóa chứ không thể giải mã). Thông thường phương pháp mã hóa một chiều sử dụng một hàm băm (hash function) để biến một chuỗi thông tin thành một chuỗi hash có độ dài nhất định. Ta không có bất kì cách nào để khôi phục (hay giải mã) chuỗi hash về lại chuỗi thông tin ban đầu.

*Hàm băm (Hash function) là một hàm mà nó nhận vào một chuỗi có độ dài bất kì, và sinh ra một chuỗi kết quả có độ dài cố định (Gọi là chuỗi hash), dù hai chuỗi dữ liệu đầu vào, được cho qua hàm băm thì cũng sinh ra hai chuỗi hash kết quả khác nhau rất nhiều. Ví dụ như đối với kiểu dữ liệu Hash-table, ta có thể coi đây là một dạng kiểu dữ liệu mảng đặc biệt mà index nó nhận vào là một chuỗi, nó được định nghĩa bằng cách bên trong nó chứa một mảng thông thường, mỗi khi truyền vào index là một chuỗi, thì chuỗi này sẽ đi qua hàm băm và ra một giá trị hash, giá trị này sẽ tương ứng với index thật của phần tử đó trong mảng bên dưới.*

Đặc điểm của hash function là khi thực hiên băm hai chuỗi dữ liệu như nhau, dù trong hoàn cảnh nào thì nó cũng cùng cho ra một chuỗi hash duy nhất có độ dài nhất định và thường nhỏ hơn rất nhiều so với chuỗi gốc, và hai chuỗi thông tin bất kì dù khác nhau rất ít cũng sẽ cho ra chuỗi hash khác nhau rất nhiều. Do đó hash function thường được sử dụng để kiểm tra tính toàn vẹn của dữ liệu.

*Giả sử bạn có một file dữ liệu định up lên mạng, và bạn muốn người dùng có thể kiểm tra xem dữ liệu họ down về có chính sát dữ liệu mình up lên hay không. Thì bạn sẽ dùng một hash function để băm dữ liệu của file đó ra một chuỗi hash, và gửi kèm cho người dùng chuỗi hash này. Khi đó, người dùng chỉ việc dùng đúng hash function đó để tìm chuỗi hash hiện tại của file down về, rồi so sánh với chuỗi hash ban đầu, nếu hai chuỗi này giống nhau thì dữ liệu down về vẫn toàn vẹn.*

Ngoài ra có một ứng dụng mà có thể bạn thường thấy, đó là để lưu giữ mật khẩu. Vì mật khẩu là một thứ cực kì quan trọng, do đó ta không nên lưu mật khẩu của người dùng dưới dạng rõ, vì như vậy nếu bị hacker tấn công, lấy được CSDL thì hacker có thể biết được mật khẩu của người dùng. Do đó, mật khẩu của người dùng nên được lưu dưới dạng chuỗi hash, và đối với server thì chuỗi hash đó chỉnh là “mật khẩu” đăng nhập (lúc đăng nhập thì mật khẩu mà người dùng nhập cũng được mã hóa thành chuỗi hash và so sánh với chuỗi hash trong CSDL của server). Dù hacker có lấy được CSDL thì cũng không tài nào có thể giải mã được chuỗi hash để tìm ra mật khẩu của người dùng.

Thuật toán mã hóa một chiều (hàm băm) mà ta thường gặp nhất là MD5 và SHA.

1. Mã hóa đối xứng

Mã hóa đối xứng (Hay còn gọi là mã hóa khóa bí mật) là phương pháp mã hóa mà key mã hóa và key giải mã là như nhau (Sử dụng cùng một secret key để mã hóa và giải mã). Đây là phương pháp thông dụng nhất hiện nay dùng để mã hóa dữ liệu truyền nhận giữa hai bên. Vì chỉ cần có secret key là có thể giải mã được, nên bên gửi và bên nhận cần làm một cách nào đó để cùng thống nhất về secret key.

Để thực hiện mã hóa thông tin giữa hai bên thì:

* Đầu tiên bên gửi và bên nhận bằng cách nào đó sẽ phải thóa thuận secret key (khóa bí mật) được dùng để mã hóa và giải mã. Vì chỉ cần biết được secret key này thì bên thứ ba có thể giải mã được thông tin, nên thông tin này cần được bí mật truyền đi (bảo vệ theo một cách nào đó).
* Sau đó bên gửi sẽ dùng một thuật toán mã hóa với secret key tương ứng để mã hóa dữ liệu sắp được truyền đi. Khi bên nhận nhận được sẽ dùng chính secret key đó để giải mã dữ liệu.

Vấn đề lớn nhất của phương pháp mã hóa đối xứng là làm sao để “thỏa thuận” secret key giữa bên gửi và bên nhận, vì nếu truyền secret key từ bên gửi sang bên nhận mà không dùng một phương pháp bảo vệ nào thì bên thứ ba cũng có thể dễ dàng lấy được secret key này.

Các thuật toán mã hóa đối xứng thường gặp: DES, AES…

1. Mã hóa bất đối xứng

Mã hóa bất đối xứng (Hay còn gọi là mã hóa khóa công khai) là phương pháp mã hóa mà key mã hóa (lúc này gọi là public key – khóa công khai) và key giải mã (lúc này gọi là private key – khóa bí mật) khác nhau. Nghĩa là key ta sử dụng để mã hóa dữ liệu sẽ khác với key ta dùng để giải mã dữ liệu. Tất cả mọi người đều có thể biết được public key (kể cả hacker), và có thể dùng public key này để mã hóa thông tin. Nhưng chỉ có người nhận mới nắm giữ private key, nên chỉ có người nhận mới có thể giải mã được thông tin.

Để thực hiện mã hóa bất đối xứng thì:

* Bên nhận sẽ tạo ra một gặp khóa (public key và private key). Bên nhận sẽ dữ lại private key và truyền cho bên gửi public key. Vì public key này là công khai nên có thể truyền tự do mà không cần bảo mật.
* Bên gửi trước khi gửi dữ liệu sẽ mã hóa dữ liệu bằng thuật toán mã hóa bất đối xứng với key là public key từ bên nhận.
* Bên nhận sẽ giải mã dữ liệu nhận được bằng thuật toán được sử dụng ở bên gửi, với key giải mã là private key.

Điểm yếu lớn nhất của mã hóa bất đối xứng là tốc độ mã hóa và giải mã rất chậm so với mã hóa đối xứng, nếu dùng mã hóa bất đối xứng để mã hóa dữ liệu truyền – nhận giữa hai bên thì sẽ tốn rất nhiều chi phí.

Do đó, ứng dụng chỉnh của mã hóa bất đối xứng là dùng để bảo mật secret key cho mã hóa đối xứng: Ta sẽ dùng phương pháp mã hóa bất đối xứng để truyền secret key của bên gửi cho bên nhận. Và hai bên sẽ dùng secret key này để trao đổi thông tin bằng phương pháp mã hóa đối xứng.

Thuật toán mã hóa bất đối xứng thường thấy: RSA.

1. Ứng dụng

Khi cần bảo mật thông tin truyền đi giữa các ứng dụng chat, hoặc các ứng dụng có truyền nhận thông tin bí mật giữa client-server, thì ta có thể sử dụng kết hợp phương pháp mã hóa bất đối xứng và phương pháp mã hóa đối xứng để đảm bảo dữ liệu đó sẽ được bảo mật.

Ta sẽ thực hiện bằng cách dùng phương pháp mã hóa bất đối xứng để truyền secret key từ bên gửi cho bên nhận, và dùng key này để mã hóa, giải mã thông tin.

Ta có thể thực hiện theo quy trình sau:

* Bên nhận sinh ra cặp khóa public key và private key (dùng để mã hóa secret key của bên gửi) và gửi cho bên gửi một cách công khai, không cần bảo mật.
* Bên gửi sinh ra một secret key (dùng để mã hóa dữ liệu), mã hóa secret key này bằng thuật toán mã hóa bất đối xứng với key mã hóa là public key của bên nhận, sau đó truyền cho bên nhận.
* Bên nhận nhận dữ liệu và giải mã nó bằng thuật toán mã hóa bất đối xứng được sử dụng ở bên gửi, với key giải mã là private key. Khi đó sẽ ra được secret key dùng để mã hóa dữ liệu.
* Sau đó mỗi khi cần truyền dữ liệu thì bên gửi sẽ mã hóa dữ liệu đó bằng secret key trước khi gửi, và bên nhận cũng sẽ giải mã dữ liệu bằng secret key đó.

## Generic Programming C++

### Cast kiểu trong C,C++

Chuyển đổi kiểu dữ liệu (type-casting) là một kỹ năng quan trọng trong lập trình C/C++. Một số bài toán phức tạp không thể giải được do dữ liệu không tương thích, qua quá trình type-casting đã trở nên đơn giản hơn bao giờ hết. Vì thế việc tìm hiểu và thành thạo trong việc casting là rất quan trọng và cần thiết đối với lập trình viên. Bài viết này sẽ giới thiệu đến các bạn một số phương pháp casting thông dụng và hiệu quả.

1. Luật ép kiểu C++
2. C-style casts

Trong C, ép kiểu thông qua toán tử (<Kiểu\_dữ\_liệu>)

int i1 = 10;

int i2 = 4;

float f = (float)i1 / i2;

C++ cho phép bạn kiểu của C để ép kiểu như sau:

int i1 = 10;

int i2 = 4;

float f = float(i1) / i2;

Ép kiểu theo chuẩn C không bị kiểm tra trong thời gian biên dịch, cách ép kiểu như vậy có nguy cơ vì sự khác biệt cấu trúc dữ liệu phức tạp trong C++

Nhin chung, ép kiểu chuẩn C không an toàn, bạn cân nhắc trước khi dùng .

1. Type\_cast C++

Bản chất của type-casting không phải là biến đổi kiểu dữ liệu này thành kiểu dữ liệu khác. Thực chất type-casting chỉ thay đổi cách đọc các byte dữ liệu của một biến, hoặc thay đổi độ dài sẽ được đọc của vùng nhớ mà biến đó trỏ tới. Do đó khi casting, dữ liệu sẽ không bị thay đổi, nhưng dữ liệu có thể sẽ được hiển thị khác với lúc ban đầu, tuỳ theo tính tương thích giữa kiểu dữ liệu trước và sau khi casting.

Bản thân toán tử gán = cũng đã được overload lại để phục vụ cho việc casting các kiểu dữ liệu thông thường như int, float, char, … Đối với các kiểu dữ liệu do người dùng định nghĩa, ta cũng có thể overload lại nhưng tồn tại một số nhược điểm như sau:

* Khai báo phức tạp, tốn công sức và thời gian khi cần casting nhiều kiểu dữ liệu khác nhau.
* Khi nâng cấp và mở rộng dự án phải bổ sung thêm, dễ sai sót dẫn đến lỗi.
* Ép kiểu không tường minh đôi lúc gây khó hiểu cho lập trình viên.

C++ đã cung cấp cho chúng ta một số công cụ hỗ trợ việc casting một cách tổng quát, có thể sử dụng với bất kỳ kiểu dữ liệu nào. Sử dụng chúng, ta hạn chế được các nhược điểm kể trên. Nhưng tốn chi phí để dùng.

Cú pháp chung của các kiểu casting:

1. type\_cast <new\_type> (expression);

Trong đó, new\_type là kiểu dữ liệu cần chuyển đổi, expression là đối tượng cần được chuyển sang kiểu dữ liệu mới. Để lưu trữ lại, ta cần khai báo một biến có kiểu dữ liệu new\_type và gán kết quả sau khi casting vào biến đó.

1. Static\_cast C++

**static\_cast** có nhiều điểm tương đồng với **dynamic\_cast**, ngoại trừ không thực hiện kiểm tra tính tương thích giữa đối tượng và kiểu dữ liệu mới. Do đó, việc kiểm tra tính tương thích và đảm bảo của quá trình casting được trao cho lập trình viên.

Do không kiểm tra tính tương thích giữa đối tượng và kiểu dữ liệu nên **static\_cast** tốn ít chi phí bộ nhớ hơn so với **dynamic\_cast.**

Cú pháp:

**static\_cast <type-id> ( expression )**

Cuyển kiểu cho biểu thức expression sang kiểu type-id

Xem ví dụ:

char c = 97;

cout << static\_cast<int>(c) << endl; // in ra 97, không phải 'a'

static\_cast dùng tốt cho các kiểu cơ bản:

int i1 = 10;

int i2 = 4;

float f = static\_cast<float>(i1) / i2;

Thuận lợi chính của static\_cast là trình biên dịch có kiểm tra, tránh được các lỗi vô ý. Static\_cast không mạnh như ép kiểu chuẩn C

Dùng khi bạn muốn làm rõ các chuyển kiểu ngầm định

int i = 48;

char ch = i; // chuyển kiểu ngầm int sang char

Ép kiểu từ int (4-bytes) sang char (1-byte) không an toàn, trình biên dịch sẽ cảnh báo. Ép kiểu với static\_cast sẽ thông báo với trình biên dịch rằng bạn hiểu rõ các nguy cơ và bạn thực sự muốn làm vậy.

int i = 48;

char ch = static\_cast<char>(i);

1. dynamic\_cast (địa chỉ và tham chiếu) C++

**dynamic\_cast** được sử dụng với con trỏ (pointer) hoặc tham chiếu đến các lớp. Cú pháp:

dynamic\_cast < new\_type > ( expression )

* new\_type: thường là kiểu tham chiếu tới subclass
* expression: lvalue tham chiếu tới baseclass đang cần cast sang kiểu new\_type.   
  Nếu tham số là NULL thì con trỏ null cũng được trả về.

Nếu đối tượng không tương thích với kiểu dữ liệu mới, chương trình sẽ trả về **NULL** (đối với con trỏ) hoặc **throw exception std::bad\_cast** nếu new\_type là kiểu tham chiếu.

**dynamic\_cast** được sử dụng khi ta không biết kiểu dữ liệu thực sự của đối tượng trước khi casting. Do đó nếu đã biết chính xác kiểu dữ liệu của đối tượng thì không nên sử dụng nó vì **dynamic\_cast** đòi hỏi chi phí cao cho việc sử dụng.

Chúng ta có thể dùng dynamic\_cast để Upcast, C-style cast và static\_cast cũng thể Upcast.

Nếu tham số **expression** là kiểu tham chiếu có tính đa hình và **new\_type** là con trỏ void, dynamic\_cast sẽ return về con trỏ tới kiểu đối tượng thấp nhất trên cây thừa kế

* Nếu **new**\_**type** là subclass **expression** thì cast thành công, trường hợp này gọi là Downcast.
* Nếu **new**\_**type** và **expression** cùng là subclass của một base class thì cast thành công, trường hợp này gọi là sidecast.
* Nếu không thể kiểm tra được, con trỏ null của new\_type sẽ return, std::bad\_cast được ném ra

Ví dụ sau minh hoạ sử dụng **dynamic\_cast** để Downcast các đối tượng thừa kế**:**

1. #include <iostream>
2. class Animal{
3. public:
4. Animal(){}
5. virtual ~Animal(){}
6. //...
7. };
9. class Cat : public Animal{
10. public:
11. Cat(){}
12. ~Cat(){}
13. //...
14. };
16. class Bird : public Animal{
17. public:
18. Bird(){}
19. ~Bird(){}
20. //...
21. };
23. #define ANIMAL\_COUNT 5
24. int main(){
25. Animal\*\* animal\_list = new \*Animal[ANIMAL\_COUNT];
27. for(int i = 0; i < ANIMAL\_COUNT; i++){
28. if(Some\_logical\_expressions)
29. animal\_list[i] = new Cat();
30. else
31. animal\_list[i] = new Bird();
32. }
34. for(int i = 0; i < ANIMAL\_COUNT; i++){
35. if(dynamic\_cast<Cat\*>(animal\_list[i])) {
36. //Do something
37. }
38. else if(dynamic\_cast<Bird\*>(animal\_list[i])){
39. //Do something else
40. }
41. }
43. for(int i = 0; i < ANIMAL\_COUNT; i++)
44. if animal\_list[i];
45. if [] animal\_list;
47. return 0;
48. }

Khi dynamic\_cast được dùng trong hàm dựng hay hàm hủy (gián tiếp hay trực tiếp), **expression** đang là đối tượng trong quá trình khởi tạo/hủy, thì **expression** xem như là kiểu đối tượng thấp nhất trên cây thừa kế. Nếu new\_type không phải tham chiếu tới base class hay chính class của **expression** thì không xác định được hành vi thực thi.

1. reinterpret\_cast C++

**reinterpret\_cast** thao tác với các con trỏ tương tự như các toán tử chuyển đổi thông thường. Kết quả của việc casting đơn giản là việc copy dữ liệu quản lý bởi một con trỏ sang con trỏ khác. Do đó nó có thể casting giữa tất cả các kiểu dữ liệu, kể cả các lớp không có quan hệ với nhau.

Khi sử dụng **reinterpret\_cast**, lập trình viên cần hiểu rõ mình đang làm gì, vì hầu hết quá trình casting với **reinterpret\_cast** đều hợp lệ và không thông báo lỗi trong quá trình build.

1. const\_cast C++

Biến hằng là kiểu Rvalue, không thể thay đổi giá trị hay truyền như một đối số hàm

**const\_cast** được sử dụng để thêm hoặc loại bỏ **const** khỏi một biến. **const**\_**cast** là cách duy nhất để thay đổi tính hằng của biến. (Đọc thêm [Thao tác với Constant trong C++](#_Thao_Tác_Với_2))

Ví dụ

1. #include <stdio.h>
3. void echo(char\* str){
4. printf("%s", str);
5. }
7. int main(){
8. const char \*str = "STDIO Tutorial";
10. //echo(str); //Error incompatible type
11. echo(const\_cast<char\*>(str)); //OK
13. return 0;
14. }
15. Typeid C++

Chuẩn ANSI-C++ định nghĩa một toán tử mới là typeid, cho phép kiểm tra kiểu dữ liệu của một biến.

typeid được định nghĩa trong header <typeinfo>, do đó các lập trình viên cần khai báo thư viện này trước khi sử dụng.

Cú pháp sử dụng:

1. typeid (expression)

Kết quả trả về một tham chiếu tới const object của kiểu dữ liệu. Ta có thể so sánh hai typeid thông qua toán tử == và != đã được overload. Ngoài ra ta có thể lấy được một chuỗi kí tự mô phỏng kiểu dữ liệu hoặc tên lớp thông qua phương thức name() của typeid.

Sử dụng lại cấu trúc kế thừa ở ví du trên, ta xét ví dụ sau:

1. int main()
2. {
3. Animal\* animal = new Animal();
4. Animal\* cat = new Cat();
5. //Check to know if 2 variable’s type are the same or not
6. if(typeid(animal) == typeid(cat))
7. printf("%s\n", typeid(cat).name());
8. else
9. printf(“type are different\n”);
10. printf(typeid(static\_cast<Cat\*>(cat)).name());
12. return 0;
13. }

Kết quả được in ra trên màn hình:

class Animal \*

class Cat \*

Ta thấy dù đã được khởi tạo qua toán tử new, bản chất của lớp Cat vẫn là Animal. Do đó typeid vẫn trả về một const object có kiểu Animal. Qua đó ta thấy được tầm quan trọng của việc casting trong OOP.

### Template trong C++

1. Template là gì?

“Template” là từ khóa trong C++, chúng ta có thể hiểu rằng là nó một kiểu dữ liệu trừu tượng, đặc trưng cho các kiểu dữ liệu cơ bản. “Template” là từ khóa báo cho trình biên dịch rằng đoạn mã sau đây định nghĩa cho nhiều kiểu dữ liệu và mã nguồn của nó sẽ được compile sinh ra tương ứng cho từng kiểu dữ liệu trong quá trình biên dịch.

Có 2 loại “template” cơ bản:

* Function template: là một khuôn mẫu hàm, cho phép định nghĩa các hàm tổng quát thao tác cho nhiều kiểu dữ liệu.
* Class template: là một khuôn mẫu lớp, cho phép định nghĩa các lớp tổng quát cho nhiều kiểu dữ liệu.

1. Tham số template mặc định

Xét ví dụ sau

template <class Type = int> //mặc định kiểu int

class Point{

Type x;

Type y;

};

Point<> point;

Trong ví dụ trên khi khai báo thể hiện của khuôn mẫu lớp là Point<> point; do không có kiểu truyền vào, lúc này trình biên dịch sẽ mặc định là kiểu đó là int do lúc đầu đã gán tham số kiểu mặc định là  int khi khai báo kiểu trong template.

Lưu ý: tham số kiểu mặc định phải nằm ngoài cùng bên phải trong danh sách các tham số template

1. Hàm template

Đoạn code sau để hoán vị giữa hai số, nếu ta cần hoán vị giữa 2 số kiểu float hoặc double,… thì ta lại phải định nghĩa các hàm cho từng loại kiểu dữ liệu:

void Swap( int &x, int &y){

int Temp = x;

x = y;

y = Temp;

}

Khi dùng “template”, ta chỉ cần định nghĩa một hàm duy nhất cho các kiểu dữ liệu: int, float, double, …

template <class Type>

void Swap( Type &x, Type &y){

Type Temp = x;

x = y;

y = Temp;

}

void main(){

int x = 5, y = 10;

float a = 5.5f, b = 3.0f;

Swap(x, y);

Swap(a, b);

}

Vậy cách thức nó hoạt động ra sao, ta hãy xét hoạt động của trình biên dịch khị gặp lời gọi hàm Swap(x, y) với tham số truyền vào là kiểu int:

* Khi gọi Swap(x, y), trình biên dịch tìm xem có hàm Swap() nào đã được khai báo với tham số kiểu int hay chưa, nếu có thì sẽ liên kết với hàm đó.
* Nếu chưa nhưng lại tìm thấy từ khóa “**template**” với hàm **Swap**() được truyền vào 2 tham số cùng kiểu với nhau (lúc này là kiểu Type), trình biên dịch kiểm tra 2 tham số x và y có cùng kiểu dữ liệu với nhau hay không
* Nếu cùng kiểu thì trình biên dịch lại kiểm tra có hàm **Swap**() gồm 2 tham số kiểu int đã được sinh ra trước đó hay chưa
* Nếu có thì lời gọi hàm sẽ liên kết với hàm Swap() đã được sinh ra
* Nếu chưa thì khi đó trình biên dịch sẽ sinh ra hàm **Swap**(int &x, int &y), tương tự với **Swap**() kiểu float.

Vì vậy trong quá trình biên dịch sẽ sinh ra 2 hàm Swap() cho 2 loại kiểu dữ liệu trên.

Trong ví dụ trên tôi đã chỉ ra cách khai báo hàm mẫu, trong đó:

* “Type” chỉ là một tên riêng thể hiện cho một kiểu dữ liệu tổng quát.
* “class” ta có thể thay thế bằng “typename”, ở đây nó không có sự khác biệt.

Ngoài ra ta có thể dùng prototype cho nguyên mẫu hàm giống như ta làm cho các hàm thông thường

template <class T> void Swap( T &x, T &y);

void main(){

int x = 3, y = 4;

float a = 1.2f, b = 2.2f;

Swap(x, y);

Swap(a, b);

}

template <class T> void Swap( T &x, T &y){

T z = x;

x = y;

y = z;

}

**Lưu ý: để dùng hàm template cho tham biến truyền vào, bạn phải khai báo tham số kiểu tham chiếu.**

Trong ví dụ trên ta chỉ hoán ví giữa 2 tham số cùng kiểu, vậy với khác kiểu thì sau? Ta chỉ cần khai báo thêm một kiểu dữ liệu tổng quát

template <class T, class X> void Swap( T &x, X &y);

//hoặc

template <typename T, typename X> void Swap( T &x, X &y);

1. Quá tải hàm Templates

Nguyên mẫu hàm (function template) đều có tính chất của một hàm thông thường, nó cho phép chúng ta nạp chồng (overload function)

template <class T, class X> T Sum( T x, X y){return x + y;}

template <class T, class X> X Sum( T x, X y, T z){return x + y + z;}

**Lưu ý: muốn nạp chồng hàm thì các tham số truyền vào ở các hàm phải khác nhau**

1. Template với class

Ta xét ví dụ sau

class Point{

int x;

int y;

};

Nếu muốn khai báo khuôn mẫu lớp với kiểu tùy ý ta làm như sau

template <class Type>

class Point{

Type x;

Type y;

};

Cú pháp khai báo đối tượng template Point kiểu integer:

Point<int> P1;

Trong ví dụ trên là khai báo Point với 2 thuộc tính cùng kiểu với nhau, chúng ta có thể định nghĩa với 2 thuộc tính có kiểu dữ liệu khác nhau​

template <class TypeOne, class TypeTwo>

class Point{

TypeOne x;

TypeTwo y;

Public:

Point();

};

// định nghĩa lớp

template <class TypeOne, class TypeTwo>

Point<TypeOne, TypeTwo>::Point(){

x = 0;

y = 0;

}

**Lưu ý: Ta phải khai báo template cùng tham số kiểu trước mỗi định nghĩa hàm thành viên.**

1. Template với struct

Chúng ta khai báo cấu trúc node dùng template như sau:

//dinh nghia cau truct node

template<class Y>

struct node {

Y data;

node \*next;

node \*prev;

//ham dung danh sach khoi tao

node(Y data) : data(data) {this->next = this->prev = NULL;}

};

Khai báo biến struct template tương tự như class template:

int main(){

struct node<int> \*p = new node<int>(6);

delete p;

return 0;

}

1. Kết hợp Template struct và class

Xét ví dụ về khai báo lớp DSLK. Ta khai báo lớp lồng node:

//xay dung lop link list

template<class T>

class linklist {

private:

//dinh nghia cau truct node

struct node {

T data;

node \*next;

node \*prev;

//ham dung danh sach khoi tao

node(T data) : data(data) { this->next = this->prev = NULL; }

};

node \*p\_head;

node \*p\_tail;

int size;

/\*Các phương thức…\*/

};

**Lưu ý: Phía sau khai báo template, bạn không thể đặt từ khóa typedef, trình biên dịch sẽ báo lỗi.**

1. Chuyên hóa Lớp template

Trong nhiều trường hợp làm việc với template, bạn có một phiên bản chung cho tất cả kiểu dữ liệu có thể - lớp vector<> cũng được triển khai chính xác như vậy. Ý tưởng chuyên hóa là để ghi đè triển khai mặc định để xử lí cho kiểu dữ liệu cụ thể theo cách khác.

Ví dụ, hầu hết vector triển khai như một mảng với kiểu dữ liệu cho trước, bạn muốn tiết kiệm vùng nhớ và triển khai vector<bool> như là vector kiểu integer với mỗi bit tương ứng với một entry trong vector. Vậy thì bạn sẽ phải có hai vector class riêng biệt. Lớp đầu tiên:

template <typename T>

class vector{

// các khai báo hàm, linh tinh…

private:

T\* vec\_data; // chúng ta chứa dữ liệu theo khối vùng nhớ cấp phát động

int length; // số phân tử được dùng

int vec\_size; // kích thước thật của vector

};

Triển khai lớp template vector kiểu bools sẽ hơi khác, chúng ta chú trọng xử lí từng bit trực tiếp

Trước hết , bạn vẫn phải khai báo template nhưng với danh sách tham số rỗng:

template <>

tên class theo sau phải có kiểu dữ liệu cụ thể: class tên\_class<kiểu>

template <>

class vector <bool>{

// interface

private:

unsigned int \*vector\_data; // mỗi bit integer là một entry kiểu boolean

int length;

int size;

};

Lưu ý rằng phiên bản chuyên hóa template sẽ hoàn hảo nếu có interface khác so với phiên bản chung, giữa chúng không liên quan về code dù cùng là template vector.

Mục đích chính của chuyên hóa trong trường hợp này là tính hiệu quả vùng nhớ cho kiểu bool. Ngoài ra bạn cũng có thể triển khai các chuyên hóa template cho từng kiểu dữ liệu cụ thể vì lý do nào đó.

Trường hợp khác, khi bạn muốn chuyên hóa template cho một kiểu teamplate nào đó không nằm trong triển khai của bạn. Ví dụ: bạn có template sortedVector, yêu cầu phải định nghĩa toán tử lớn hơn “>”, còn các lớp đối tượng lại viết bởi người khác, lại không định nghĩa quá tải “>” nhưng lại có hàm so sánh. Bạn sẽ phải chuyên hóa template để xử lí những lớp này theo một cách riêng.

1. Chuyên hóa template từng phần:

Chuyên hóa từng phần có nguồn gốc tương tự như chuyên hóa toàn phần ở trên. Thay vì triển khai cho một kiểu dữ liệu cụ thể, bạn viết một teamplate chuyên hóa cho một tính năng nào đó nhưng vẫn để lớp chọn các tính năng khác như một phần của template.

Trở lại với ý tưởng mở rộng lý thuyết vector, chúng ta có sortedVector: chúng ta cần có cách để so sánh giữa hai phần tử. Chúng ta có thể dùng toán tử logic “>” nếu kiểu dữ liệu đã được quá tải hoặc chuyên hóa nếu chưa có.

Nhưng bây giờ chúng ta cần có con trỏ đối tượng trong vector đã sắp xếp. Chúng ta có thể sắp xếp chúng theo giá trị của con trỏ, chỉ dùng “>” (từ nhỏ tới lớn)

template <typename T>

class sortedVector{

public:

void insert (T val) {

if ( length == vec\_size ) // length là số phần tử {

vec\_size \*= 2; // chúng ta sẽ bỏ qua khả năng overflow!

vec\_data = new T[vec\_size]; // cấp phát vùng nhớ mới gấp đôi

}

++length; // tăng số phần tử lên

// chúng ta sẽ bắt đầu từ điểm cuối, duyệt từng phần tử cho tới khi tìm thấy chỗ để chèn phần tử mới

int pos;

for( pos = length; pos > 0 && val > vec\_data[pos - 1]; --pos ) {

vec\_data[pos] = vec\_data[pos - 1];

}

vec\_data[pos] = val;

}

// các hàm khac...

private:

T \*vec\_data;

int length;

int size;

};

Chú ý vòng lặp for, chúng ta đang so sánh trực tiếp giữa 2 phần tử kiểu “T”. Nhìn từ bên ngoài có vẻ ổn, nhưng nếu các phần tử kiểu con trỏ thì sao? Thường chúng ta vẫn xét theo giá trị chứ không phải địa chỉ con trỏ. Đối với trường hợp này, vòng lặp for phải dùng toán tử truy xuất nội dung:

for( pos = length; pos > 0 && \*val > \*vec\_data[pos - 1]; --pos )

Cái chúng ta cần ở đây là dùng chuyên hóa một phần dựa trên kiểu template là con trỏ hay đối tượng thực.

Cú pháp khai báo chuyên hóa một phần để xử lí con trỏ bất kì như sau:

template <typename T>

class sortedVector<T \*>{

public:

// khai báo lại hàm insert tương tự:

insert( T \*val ){

if ( length == vec\_size ) // length : số phần tử {

vec\_size \*= 2; // bỏ qua khả năng tràn bộ đệm!

vec\_data = new T[vec\_size];

}

++length; // tăng số phân tử lên

// vòng lặp for sẽ khai báo dùng toán tử truy xuất nội dung con trỏ

int pos;

for( pos = length; pos > 0 && \*val > \*vec\_data[pos - 1]; --pos ) {

vec\_data[pos] = vec\_data[pos - 1];

}

vec\_data[pos] = val;

}

// các hàm khac...

private:

T\*\* vec\_data; // mảng các con trỏ kiểu T\*

int length;

int size;

};

Chú ý cú pháp chuyên hóa lớp:

Thứ nhất: vẫn khai báo kiểu template với tham số T. Khai báo class template phải có tham số template là con trỏ \*T phía sau:

template <typename T>

class sortedVector<T \*>

Điều này báo cho trình biên dịch biết rằng ta đang chuyên hóa cho mọi kiểu con trỏ template.

Thứ hai: T vẫn là kiểu được trỏ tới, bản thân nó không phải con trỏ. Ví dụ: kiểu sortedVector<int\*>, T là kiểu int.

Điểm quan trọng là con trỏ không phải kiểu dữ liệu, nó chỉ trỏ tới kiểu dữ liệu T.

Bạn cần cẩn thận một chút khi triển khai code: vec\_data là T\*\*, vì chúng ta cần mảng động chứa các con trỏ T\*.

Nhưng có thực sự bạn muốn sortedVector hoạt động như vậy: sắp xếp cho kiểu con trỏ?

Có một nguyên nhân thực tế: khi bạn cấp phát động cho mảng các đối tượng, hàm dựng mặc định phải được gọi để khởi tạo từng đối tượng. Nếu không có hàm dựng mặc định (nếu mọi đối tượng cần dữ liệu để khởi tạo), thì bạn cần danh sách con trỏ tới các đối tượng, nhưng bạn cũng cần chúng để sắp xếp các đối tượng.

Lưu ý: Bạn cũng cần chuyên hóa cho đối số của template – Ví dụ: nếu bạn có kiểu fixedVector cho phép người dùng vừa xác định kiểu, vừa khai báo độ dài cố định cho vector (tránh cấp phát động mở rộng vùng nhớ).

Cú pháp:

template <typename T, unsigned length>

class fixedVector { ... };

Cú pháp chuyên hóa từng phần cho kiểu bool:

template <unsigned length>

class fixedVector<bool, length> {...};

Chú ý: tham số template chỉ còn “length” và khai báo tên lớp cũng gồm kiểu bool và tham số length – length được gọi là tham số non-type của template. Chỉ ra rằng đây là tham số kiểu số nguyên không dấu.

Vậy làm sao trình biên dịch chọn kiểu chuyên hóa nào để dùng khi có hàng tá sự kết hợp giữa kiểu chung, chuyên hóa từng phần, toàn phần… ?

Luật chung: chọn chuyên hóa teamplate đầy đủ nhất trước – chuyên hóa này có đối số template

The general rule of thumb is that the compiler will pick the most specific template specialization--the most specific template specialization is the one whose template arguments would be accepted by the other template declarations, but which would not accept all possible arguments that other templates with the same name would accept.

Ví dụ: để sortedVector<int\*> sắp xếp theo vùng nhớ, bạn tạo chuyên hóa toàn phần cho kiểu int\*.

Trình biên dịch mặc định sẽ chọn triển khai chuyên hóa nào đầy đủ nhất dành cho kiểu con trỏ. Ở đây sẽ là chuyên hóa int\* phù hợp nhất vì là chuyên hóa toàn phần, chứ không phải kiểu con trỏ khác như double\*chẳng hạn. Chuyên hóa int\* là tham số chung cho các teamplate con trỏ khác.

1. Tham số None-type template

None-type template parameters là tham số mẫu mặc định, được xác định và trình biên dịch không sinh ra kiểu khác trong suốt thời gian biên dịch.

template <class Type, int N>

class Stack{

//interface

};

Trong ví dụ trên N là kiểu int đã được xác định rõ từ trước và không thay đổi trong thời gian biên dịch

1. Template và Friend

Như chúng ta thường thấy các lớp và các hàm có thể được khai báo là friend của nhau. Đối với khuôn mẫu lớp cũng vậy, chúng ta có thể khai báo friend giữa khuôn mẫu lớp và hàm toàn cục, hoặc hàm của các lớp khác,..

template <class Type>

class Point{

public:

friend void Sum(Type X);

friend void Sum();

};

template <class Type>

void Sum(Type X){printf("a friend of only a class template with same type\n");}

void Sum(){printf("a friend of every class template\n");}

Trong ví dụ trên thì hàm Sum() sẽ là hàm bạn với Point<int>, Point<float>,… còn nếu Type là int thì lúc này Sum(int X) là hàm bạn với Point<int> nhưng sẽ không là bạn với Point<float>

1. Template và thành viên tĩnh

Tương tự như nontemplate class, chúng ta có thể định nghĩa thành viên tĩnh trong khuôn mẫu lớp, và thành viên tĩnh phải được khởi tạo ở phạm vi toàn cục. Nhưng thành viên tĩnh trong khuôn mẫu lớp khác với nontemplate class là thời điểm khởi tạo, đối với nontemplate class thì sẽ được khởi tạo khi bắt đầu chương trình, nhưng còn đối với khuôn mẫu lớp thì lại khởi tạo trong thời gian biên dịch.

template <class Type>

class Point{

public:

static int N;

};

Template <class Type> int Point<Type>::N = 0;

void main(){

Point<int> t1;

t1.N++;

Point<float> t2;

t2.N++;

Point<int> t3;

t3.N++;

printf("T1: %d\n", t1.N);

printf("T2: %d\n", t2.N);

printf("T3: %d\n", t3.N);

}

Kết quả in ra sẽ là 2 1 2. Nguyên nhân là khi ta khai báo Point <int> t1; lúc này trình biên dịch tìm xem Point<int> đã được khởi tạo chưa, lúc này chưa thì trình biên dịch sẽ khởi tạo ra đối tượng Point với kiểu cụ thể là int, thành viên tĩnh N được khởi tạo từ đây, sau khi thực thi dòng t1.N++; lúc này N = 1.

Đối với Point<float> t2 cũng tương tự như với Point<int> t1, kết quả khi thực thi dòng t2.N++; lúc này N = 1. Đối với  Point<int> t3, lúc này lớp Point với kiểu int đã được khởi tạo rồi, thành viên tĩnh N của t3 được sử dụng chung với t1, lúc này khi t3.N++; thì N = 2. Vì vậy khi in kết quả lên màn hình sẽ ra kết quả 2 1 2

Như tôi nói ở phần khuôn mẫu hàm thì “typename” và “class” là như nhau, nhưng trong trường hợp này thì lại phải dùng “typename”

template <typename Type>

class Point{

Type p;

};

Ở đây A là một kiểu của Type vì thế mà p được hiểu là một con trỏ kiểu Type::A\*, khi đó nếu muốn khai báo thể hiện của template Point thì làm như sau

class Q{

public:

typedef int A;

};

Point<Q> z;

 Còn nếu như ta bỏ “typename”

template <class Type>

class Point{

Type::A \* p;

};

Lúc này Type::A \* p được hiểu là tích giữa biến static A của lớp Type với một biến p nào đó. Ngoài ra đối với khuôn mẫu lớp ta có thể kế thừa giống như những các lớp thông thường

template <class Type>

class Point{

protected:

Type x;

Type y;

};

template <class Type>

class Point3D: public Point<Type>{

Type z;

};

Template cho phép chúng ta tổng quát hóa kiểu dữ liệu, nhưng đồng thời cũng hỗ trợ chúng ta thao tác, xử lý chuyên môn trên trên một kiểu hay nhiều nhất định

template <class Type>

class Point

{

public:

Point() {

printf(“Not explicit specialization\n”);

}

};

template<>

class Point<int>{

public:

Point() {

printf(“Explicit specialization\n”);

}

};

Như các bạn đã thấy trong từng ví dụ, việc sử dụng từ khóa “template” làm giảm rất nhiều thời gian và công sức phải code lại một hàm mà dùng cho nhiều kiểu dữ liệu, dễ dàng bảo trì, phát triển hay thay đổi  mã nguồn.

1. Template và generic programming

Hàm template findMax() triển khai template với đối số Comparable

/\*\*

\* Trả về phần tử lớn nhất trong mảng a.

\* giả sử rằng a.size( ) > 0.

\* Đôi tượng bắt buộc phải quá tải phép so sánh operator< và operator=

\*/

template <typename Comparable>

const Comparable & findMax( const vector<Comparable> & a ){

int maxIndex = 0;

for( int i = 1; i < a.size( ); ++i )

if( a[ maxIndex ] < a[ i ] )

maxIndex = i;

return a[ maxIndex ];

}

Trong hàm main, findMax được mở rộng khi dùng với nhiều kiểu dữ liệu đối số khác nhau. Khi biên dịch, sự mở rộng cho mỗi kiểu mới đều được tự động sinh ra code cho chương trình, đây gọi là **code** **bloat**.

int main( ){

vector<int> v1( 37 );

vector<double> v2( 40 );

vector<string> v3( 80 );

vector<IntCell> v4( 75 );

// code bloat sinh ra thêm cho vector, không thể nhìn thấy

cout << findMax( v1 ) << endl; // OK: Comparable = int

cout << findMax( v2 ) << endl; // OK: Comparable = double

cout << findMax( v3 ) << endl; // OK: Comparable = string

cout << findMax( v4 ) << endl; // Không hợp lệ; chưa quá tải operator<

return 0;

}

Trình biên dịch phát sinh lỗi tại findMax(v4), khi đối tượng IntCell chưa quá tải phép so sánh.

**Một lưu ý khi triển khai đối số template: Đối tượng tham số truyền vào phải có sự đảm bảo về quá tải toán tử so sánh và kiểu hàm dựng nào cần phải được triển khai**

Chúng ta thường triển khai template cho các đối tượng ngoài kiểu cơ bản, đó là lý do tại sao kiểu trả về là tham chiếu hằng

Có một số luật ngầm khi triển khai hàm template. Hầu hết lỗi xảy ra là tham số template truyền vào không khớp với đối tượng mong muốn triển khai của lập trình viên, vì template không qui định tính chính xác về kiểu của tham số (thông qua ép kiểu ngầm).

Chúng ta phải có cách để kiểm tra tính chính xác cho tham số template, dù khá phức tạp.

**Lưu ý: khi tồn tại phiên bản template và non-template đều khớp, phiên bản non-template được ưu tiên trước.**

**Nếu tồn tại 2 phiên bản template gần như khớp, thì xem như không hợp lệ.**

Lớp memory cell triển khai dùng tham số template Object có cung cấp hàm dựng mặc định không tham số, hàm dựng sao chép, phép gán mặc định sao chép

/\*\*

\* Triển khai lớp template memory cell.

\*/

template <typename Object>

class MemoryCell{

public:

explicit MemoryCell( const Object & initialValue = Object{ } ): storedValue{ initialValue } { }

const Object & read( ) const{ return storedValue; }

void write( const Object & x ){ storedValue = x; }

private:

Object storedValue;

};

Lưu ý rằng tham số hàm dựng sao chép không thể là zero vì zero không phải kiểu đối tượng có giá trị,thay vào đó phải dùng khai báo Object{ } không tham số.

Hàm main mở rộng lớp memoryCell với kiểu cơ bản và không cơ bản: MemoryCell<int> và MemoryCell<string>.

int main( ){

MemoryCell<int> m1;

MemoryCell<string> m2{"hello"};

m1.write( 37 );

m2.write( m2.read( ) + "world" );

std::cout << m1.read( ) << std::endl << m2.read( ) << std::endl;

return 0;

}

Nếu chỉ dùng một tham số template, code chúng ta sẽ tốn thêm ít cho cú pháp định nghĩa lớp. Rất nhiều lớp template được triển khai theo cách bên trên.

Mặc dù thường chúng ta tách biệt interface và triển khai riêng rẽ, nhưng hiện tại biên dịch như vậy đối với lớp template không hoạt động tốt trên nhiều platform. Do đó, nhiều trường hợp, toàn bộ lớp với triển khai sẽ được đặt hết trong file header .h.

Triển khai của hầu hết thư viện STL đều theo cách này, giúp chúng ta đỡ tốn thời gian khai báo cú pháp phức tạp.

1. Template với tham số tùy ý
2. Template với tham số ellip

**Variadic template** là kiểu template có thể nhận một chuỗi tùy ý tham số cho template với bấy kì kiểu dữ liệu nào

Cả hàm và lớp đều có thể áp dụng. Ví dụ về template tham số tùy ý

template<typename... Arguments>

class VariadicTemplate;

Tất cả cách tạo thể hiện class template này đều hợp lệ:

VariadicTemplate<double, float> instance;

VariadicTemplate<bool, unsigned short int, long> instance;

VariadicTemplate<char, std::vector<int>, std::string, std::string, std::vector<long long>> instance;

Số lượng đối số template có thể là zero:

VariadicTemplate<> instance;

Tất cả đều hợp lệ trên C++11.Tuy nhiên, nếu tạo template như sau:

template<typename T, typename... Arguments>

class VariadicTemplate;

Thì phải có ít nhất một đối số template cho kiểu T, trừ khi T là kiểu tham số mặc định đã khai báo tường minh:

template<typename T = int, typename... Arguments>

class VariadicTemplate;

Trên C++11, templates tham số tùy ý đem lại một ý nghĩa khác cho toán tử “…”, nó phức tạp hơn:

template<typename... Arguments>

void SampleFunction(Arguments... parameters);

Ví dụ trên là một hàm template, đối số trong template tùy ý được gọi là bộ tham số. Bộ tham số này sẽ được mở ra bên trong tham số hàm. Ví dụ khi bạn tạo hàm gọi như sau

SampleFunction<int, int>(16, 24);

...nó tương đương với hàm template:

template<typename T, typename U>

void SampleFunction(T param1, U param2);

1. Cú pháp sizeof...()

Một cách dùng khác với template tham số tùy ý là toán tử sizeof…(). Không giống với toán tử sizeof, dùng để xác định kích thước kiểu, toán tử sizeof… () dùng để xác định bao nhiêu kiểu dùng trong tham số tùy ý của template. Ví dụ:

template<typename... Arguments>

class VariadicTemplate{

private:

static const unsigned short int size = sizeof...(Arguments);

};

1. Cú pháp 2 toán tử ellip liền nhau (......):

Trong một vài trường hợp, 2 toán tử “…” đi liền nhau (……) hoặc có thể ghi cách ra (… …)

Cả hai cách viết có thể dùng thêm dấu phẩy ở giữa hoặc không có đều chấp nhận được.  
Kiểu cú pháp này xuất hiện với hàm template tùy ý:

template<typename... Arguments>

void SampleFunction(Arguments......){

}

Hay:

template<typename... Arguments>

void SampleFunction(Arguments... ...){

}

template<typename... Arguments>

void SampleFunction(Arguments..., ...){

}

Để cho dễ đọc, nên dùng cú pháp cuối có dấu phẩy phân cách, để tránh các tình huống hiểu nhầm.

1. Thừa kế và danh sách khởi tạo:

Khi tiển khai template tham số tùy ý cho class, nó có thể dùng khi thừa kế và danh sách khởi tạo hàm dựng. Thừa kế được hưởng lợi từ variadic template:

template<typename... BaseClasses>

class VariadicTemplate : public BaseClasses...

Nếu muốn tạo hàm dựng bên trong class **VariadicTemplate** dùng danh sách khởi tạo để gọi tất cả hàm dựng trong **BaseClassese**, chúng ta làm như sau:

template<typename... BaseClasses>

class VariadicTemplate : public BaseClasses...{

public:

VariadicTemplate(BaseClasses&&... base\_classes) : BaseClasses(base\_classes)...{

}

};

Để hiểu thêm cách dùng toán tử Rvalue && có thể tham khảo đường dẫn sau

http://thbecker.net/articles/rvalue\_references/section\_01.html

1. Chuyên hóa variadic class template:

Giống như template, variadic class template cũng có chuyên hóa. Ví dụ lại về chuyên hóa template:

template<typename T>

class Template{

public:

void SampleFunction(T param){

}

};

template<>

class Template<int>{

public:

void SampleFunction(int param){

}

};

Ví dụ về chuyên hóa variadic template:

template<typename... Arguments>

class VariadicTemplate{

public:

void SampleFunction(Arguments... params){

}

};

template<>

class VariadicTemplate<double, int, long>{

public:

void SampleFunction(double param1, int param2, long param3){

}

};

Template là một tính năng mạnh mẽ trên C++, đáng tin cậy để triển khai delegates và tuples, variadic template cung cấp một giải pháp về typesafe tốt hơn, thay thế cho cơ chế C-style ellips (…)

### Functor – đối tượng hàm trong C++

1. Tiền đề

Hình dung chúng ta đang duyệt mail box, và cần tác vụ tự động sắp xếp khi click vào tiêu đề từng field: sender, title, subject , date,…

Nếu triển khai con trỏ hàm trường hợp này. Vấn đề xảy ra:

* Có rất nhiều cách để sắp xếp tin nhắn, bạn phải hard code cho từng cách
* Nếu truyền tham số field vào, không có cách nào để truyền vào hàm so sánh của STL’s sort

Giải pháp là dùng đối tượng hàm (function object) trong C++. Đối tượng hàm có hành vi như hàm hay con trỏ hàm vậy.

1. Cách hoạt động:

Xem ví dụ sau:

myFunctorClass functor;

functor( 1, 2, 3 );

Toán tử () khá đặc biệt, bạn có thể quá tải bao nhiêu tham số tùy ý trong khi các toán tử khác có tham số cố định. Khi toán tử () được gọi, nó cho rằng đối tượng hàm là chính nó đang được gọi.

Một điều tuyệt vời nữa là vòng đời đối tượng rất linh hoạt so với hàm bình thường. Bạn có thể dùng hàm dựng functor để nhúng thông tin cần dùng cho triển khai quá tải toán tử () vào.

Ví dụ sau tạo lớp functor với hàm dựng nhận đối số integer. Khi đối tượng lớp được “gọi”, thông qua phép quá tải (), nó trả về tổng giữa giá trị “\_x” hiện tại với đối số vừa truyền vào của functor

class myFunctorClass{

public:

myFunctorClass (int x) : \_x( x ) {}

int operator() (int y) { return \_x + y; }

private:

int \_x;

};

int main(){

myFunctorClass addFive( 5 );

std::cout << addFive( 6 );

return 0;

}

Hàm dựng xem như phương tiện để cung cấp thông tin cần thiết cho functor làm việc.

1. Triển khai functor

Chúng ta sẽ viết functor để sắp xếp mail, triển khai operator() nhận 2 tham số là 2 message khác nhau, và cần hàm dựng nhận thông tin trường cần sắp xếp theo.

class MessageSorter{

public:

// hàm dựng nhận tham số trên field

MessageSorter (const std::string& field) : \_field( field ) {}

bool operator() (const Message& lhs, const Message& rhs) {

// dùng phương thức getHeader() để so sánh dữ liệu

return lhs.getHeader( \_field ) < rhs.getHeader( \_field );

}

private:

std::string \_field;

};

Dùng STL vector chứa danh sách thư và dùng .sort() sắp xếp, tham số thứ 3 là functor:

std::vector<Message> messages;

// read in messages

MessageSorter comparator;

sort( messages.begin(), messages.end(), comparator );

1. So sánh con trỏ hàm với functor

Do cách triển khai con trỏ hàm và đối tượng hàm khác nhau nên chúng không tương thích mặc dù giá trị trả về giống nhau.

Nên một hàm triển khai tham số con trỏ hàm sẽ không chấp nhận đối tượng hàm và ngược lại.

1. Functor, con trỏ hàm và Templates

Nếu muốn con trỏ hàm hay functor đều có thể truyền tham số cùng một hàm, bạn cần dùng template. Template sẽ tự suy diễn đúng kiểu cho functor và con trỏ hàm.

Ví dụ, hàm tìm số đúng, nhận tham số vector là dữ liệu đầu vào và trả về kết quả vector gồm tất cả số mà con trỏ hàm trả về true.

std::vector<int> find\_matching\_numbers (std::vector<int> vec, bool (\*pred)(int)){

std::vector<int> ret\_vec;

std::vector<int>::iterator itr = vec.begin(), end = vec.end();

while ( itr != end ) {

if ( pred( \*itr ) ) {

ret\_vec.push\_back( \*itr );

}

++itr;

}

}

Chúng ta sẽ dùng teamplate giúp hàm nhận cả tham số functor lẫn con trỏ hàm

template <typename FuncType>

std::vector<int> find\_matching\_numbers( std::vector<int> vec, FuncType pred )

{

std::vector<int> ret\_vec;

std::vector<int>::iterator itr = vec.begin(), end = vec.end();

while ( itr != end ) {

if ( pred( \*itr ) ) {

ret\_vec.push\_back( \*itr );

}

++itr;

}

}

Kết hợp với typedef, template sẽ giúp bạn khai báo con trỏ hàm dễ dàng hơn rất nhiều.

1. Functors vs. Virtual Functions

Functor và hàm ảo có quan hệ khá chặt. Chúng có cùng vấn đề:

* Làm cách nào để cùng một đoạn code có thể triển khai nhiều giải thuật linh hoạt?

Giả sử ta có hàm doMath() nhận 3 tham số: 2 kiểu integer và thứ 3 để xác định “Loại toán nào áp dụng!”

template <FuncType>

int doMath (int x, int y, FuncType func){

return func( x, y );

}

Cách khác là tạo một lớp interface có phương thức thuần ảo computeResult:

class MathComputer{

virtual int computeResult (int x, int y) = 0;

};

int doMath (int x, int y, MathComputer\* p\_computer){

return p\_computer->computeResult( x, y );

}

Cả functor và hàm thuần ảo đều giúp bạn lựa chọn giải thuật phù hợp cho chương trình.

Sự khác biệt lớn nhất là dùng hàm ảo sẽ không áp dụng được con trỏ hàm và kém linh hoạt hơn functor.

### Sử dụng thư viện STL

##### Tổng quan về C++ STL

Bạn có thể tham khảo lý thuyết về [**C++ Template tại đây**](#_Template_trong_C++)

C++ STL (Thư viện template chuẩn) là một bộ các lớp template vô cùng mạnh mẽ cung cấp các lớp và hàm được template hóa, triển khai các thuật toán và cấu trúc dữ liệu như vector, link-list, queues và stack.

Điểm thuận lợi là bạn không cần phải viết và debug, và STL đã làm tốt công việc của nó để cung cấp một phiên bản hiệu quả, hợp lý các lớp này. Mặc dù STL khá là phức tạp khi được template hóa.

Các thành phần cốt lõi của C++ STL bao gồm:

|  |  |
| --- | --- |
| **Thành phần** | **Mô tả** |
| Containers | Là bộ chứa (hay mảng) các đối tượng cụ thể. Có nhiều loại khác nhau như deque, list, vector, map… |
| Algorithms | Giải thuât triển khai trên bộ chứa đối tượng. Cho phép khởi tạo, sắp xếp, tìm kiếm và biến đổi |
| Iterators | Bộ lặp dể duyệt qua các phần tử đối tượng. Bộ này chứa các thành phần con của bộ chứa.. |

Cả ba thành phần trên được định nghĩa rất nhiều hàm giúp chúng ta thao tác trên mảng các đối tượng dễ dàng.

1. STL container

Chia làm nhiều lớp chứa dùng trong nhiều tình huống khác nhau. Nói chung, lớp chứa rơi vào 3 nhóm sau:

* Lớp chứa tuần tự
* Lớp chứa liên kết
* Lớp chứa nối tiếp

1. Lớp chứa tuần tự

Gồm 3 lớp : vector, deque và list. Tương ứng định nghĩa trong 3 thư viện <vector>, <deque>, <list>

Có vai trò duy trì trật tự của nhiều phần tử trong lớp chứa. Một trong đặc tính của lớp chứa tuần tự là có thể chèn một phần tử vào vị trí bất kì.

Công dụng phổ biến nhất là biểu diễn mảng: nếu bạn chèn 4 phần tử vào mảng, các phần tử sẽ nằm đúng vị trí bạn muốn.

* Vector: trong STL là một mảng động có khả năng tự mở rộng vùng nhớ cần thiết để chứa các phần tử. Lớp Vector cho phép truy xuất ngẫu nhiên phần tử qua toán tử [] hay phương thức .at(), và chèn, xóa phần tử cuối khá nhanh

Đoạn code minh họa chèn 6 phần tử cuối Vector:

vector<int> vect;

for (int nCount=0; nCount < 6; nCount++)

vect.push\_back(nCount); // chèn vào cuối mảng

* Deque: là lớp hàng đợi kép, là một mảng động có thể giãn theo cả 2 đầu.

deque<int> deq;

for (int nCount=0; nCount < 3; nCount++) {

deq.push\_back(nCount); // chèn vào cuối mảng

deq.push\_front(10 - nCount); // chèn vào đầu mảng

}

* List : là lớp chứa kiểu danh sách liên kết kép, mỗi phần tử chứa con trỏ trỏ tới phần tử kế tiếp và con trỏ trỏ tới phần tử phía sau.

Lớp List chỉ cho truy xuất phần tử đầu và cuối của danh sách – không cho truy xuất ngẫu nhiên.

Nếu bạn muốn tìm một phần tử ở giữa danh sách, bạn phải duyệt danh sách. List cho phép bạn chèn phần tử khá nhanh nếu biết vị trí.  
Chúng ta sẽ dùng Iterator để duyệt danh sách (sẽ đề cập phần sau)

Mặc dù lớp STL string và wstring không được chuẩn hóa trong lớp chứa tuần tự, nhưng bạn có thể dùng vector thay thế với kiểu dữ liệu <char> hay <wchar>.

1. Lớp chứa liên kết

Các lớp này tự động sắp xếp dữ liệu nhập khi dữ liệu được chèn vào bộ chứa lớp. Mặc định, lớp chứa liên kết so sánh các phần tử qua toán tử (<).

* **set** là bộ chứa để lưu trữ các phần tử độc nhất, không cho phép đúp phần tử. Các phần tử được sắp xếp theo giá trị.
* **multiset** là một set cho phép đúp số phần tử.
* **map** (hay gọi là mảng liên kết) là một set có mỗi phần tử là một cặp **key** - giá trị. Key dùng để sắp xếp và đánh chỉ mục dữ liệu, và **key** phải là duy nhất.
* **multimap** (hay gọi là từ điển) là một bản đồ cho phép đúp các **key**. Từ điển cuộc sống là một multimap: **key** là thế giới, giá trị là ý nghĩa của thế giới.

Tất cả key được sắp xếp tằng dần, và bạn có thể tra cứu giá trị bằng **key**. Một vài từ có nhiều ý nghĩa khác nhau, đó là lý do tại sao từ điển phải là dạng multimap hơn là map.

1. Lớp chứa tiếp nối

Là kiểu đặc biệt dành cho đối tượng người dùng cụ thể, ban có thể chọn lớp chứa nào cần làm việc.

* Stack : ngăn xếp là lớp chứa vận hành kiểu **LIFO** (Last In, Fisrt Out), thao tác rất đơn giản chỉ có chèn (pushed) và xóa (poped) từ phần tử cuối. Stack dùng **deque** như lớp chứa tuần tự mặc định (mặc dù vector lại có vẻ phù hợp hơn), nhưng chúng ta có thể dùng **Vector** hay **List** cũng được.
* Queue: hàng đợi là lớp chứa kiểu **FIFO** (First In, First Out), các phần tử có thể chèn (pushed) vào cuối và xóa (poped) từ phía trước.  
   Queue dùng **deque** làm lớp chứa mặc định, nhưng ta có thể dùng **List**.
* Priority queue: hàng đợi ưu tiên có phần tử luôn được sắp xếp qua toán tử (<). Khi phần tử được đẩy vào (pushed), nó sẽ được sắp xếp ngay trong hàng đợi.  
  Xóa phần tử ở phía trước hàng đợi sẽ trả quyền ưu tiên lại cho phần tử còn lại.

##### Iterator

1. Lý thuyết về Iterator

Iterator là kiểu dùng để duyệt các phần tử trong container. Iterator chia theo chức năng gồm nhiều loại:

* InputIterator
* OutputIterator
* ForwardIterator
* BidirectionalIterator
* RandomAccessIterator

Có thể coi Iterators như là một con trỏ trừu tượng.

Một kiểu đối tượng **IT** được xem là Iterator nếu nó thỏa mãn các điều kiện:

* **It** có move constructor
* **It** có move assignment
* **It** có hàm hủy
* lvalues kiểu **It** có thể swap
* std::iterator\_traits<**It**> có thành viên value\_type, difference\_type, reference, pointer, và iterator\_category
* Gọi R là một biến lvalue của **IT** thì có thể truy xuất được giá trị với (\*R) và có thể tăng dần (++R) trả về \***IT**

1. Truy xuất nội dung iterators

Chúng ta truy xuất nội dung Iterator thông qua toán tử (\*)

Iterator không thể tham chiếu tới nội dung nếu chúng đi qua iterator **end** hay đứng trước iterator **begin**. Những Iterator như vậy vẫn có thể truy xuất trong một số phương thức nhưng chúng không liên quan tới bất kì mảng nào.

Một Iterator null khởi tạo dùng hàm dựng mặc định thì con trỏ bên trong của nó không trỏ tới đâu cả.

1. Các phép toán trên iterator

Một iterator được ảo hóa hoàn hảo như một con trỏ, trỏ tới các phần tử của lớp chứa, với một bộ các quá tải toán tử định nghĩa rất phong phú:

* Toán tử truy xuất nội dung \* : Iterator trả về phần tử hiện tại mà Iterator đang trỏ tới.
* Toán tử ++ và -- : Chuyển Iterator tới phần tử kế tiếp hay trở về phần tử trước.
* Toán tử == và != : so sánh để xác định 2 Iterator có trỏ tới cùng phần tử không. Để so sánh giá trị, bạn dùng toán tử nội dung trước khi so sánh.
* Toán tử = : Gán Iterator tới vị trí mới (thường là phần tử đầu hay cuối của lớp chứa). Để gán giá trị, bạn cũng phải truy xuất nội dung trước khi gán.
* Hàm **std:advance**(<iterator>, <step>): nhảy Iterator lên/xuống tương ứng với giá trị dương/âm của **step**

1. Sử dụng Iterator

Trên hầu hết STL container đều hỗ trợ 2 phương thức lấy iterator:

* iterator begin( ): trả về iterator đại diện cho phần tử đầu tiên trong container.
* iterator end( ): trả về iterator được đánh dầu là kết thúc trong container (nằm sau phần tử cuối cùng).

Đây là đoạn code dùng vòng lặp in tất cả phần tử vector trong C++11:

#include <vector>

#include <iostream>

using namespace std;

void example()

{

vector<int> v = { 1,2,3,4 };

for( int i = 0; i != v.size( ); ++i )

cout << v[ i ] << endl;

}

Và nếu sử dụng iterator thì là dạng tương tự như:

for( vector<int>::iterator itr = v.begin( ); itr != v.end( ); itr.??? )

cout << itr.??? << endl;

Vậy chúng ta dùng phương thức gì trên iterator để có thể duyệt như cách tăng giá trị biến I trong vòng lặp bên trên?

1. Các phương thức lấy iterator

Mỗi lớp chứa gồm 4 hàm thành viên cơ bản để dùng với toán tử gán (=) là:

* begin() : trả về iterator đại diện cho phần tử đầu tiên của lớp chứa.
* end() : trả về iterator đại diện phần tử cuối của lớp chứa.
* cbegin() : trả về hằng iterator đại diện phần tử đầu tiên của lớp chứa.
* cend() : trả về hằng iterator đại diện phần tử cuối của lớp chứa.

Các phương thức trên giúp duyệt danh sách dễ dàng: lặp qua các phần tử cho tới khi gặp end().

Cuối cùng là tất cả lớp chứa đều có 2 loại iterator:

* container::iterator là iterator dùng dể truy xuất và có thể thay đổi giá trị phần tử.
* container::const\_iterator là iterator chỉ để truy xuất nhưng không thể thay đổi giá trị (lưu ý: đây không phải là biến hằng)

1. Thao tác trên Iterator

Iterator được hỗ trợ quá tải toán tử so sánh !=, ==, và có hàm dựng sao chép, toán tử gán. Do đó, sẽ có những phương thức sử dụng toán tử đa năng của iterator, chúng ta có thể liệt kệ:

* itr++ và ++itr: chuyển tới iterator đại diện phần tử kế tiếp trong container.
* \*itr: trả về tham chiếu tới phần tử mà iterator đại diện. Tham chiếu này có thể hoặc không thay đổi giá trị được (bàn sau).
* itr1==itr2: so sánh xem 2 iterator có cùng tham chiếu tới cùng 1 phần tử không.
* itr1!=itr2: trả về true nếu 2 iterator tham chiếu tới 2 phần tử khác nhau, hoặc false nếu ngược lại.

Vòng lặp trong đoạn code trên sẽ được tối ưu như sau:

for( vector<int>::iterator itr = v.begin( ); itr != v.end( ); ++itr )

cout << \*itr << endl;

Ngoài ra chúng ta có thể dùng đồng thời phép tham chiếu và vừa tăng iterator \*itr++. Dùng cấu trúc While như sau:

vector<int>::iterator itr = v.begin( );

while( itr !=v.end( ) )

cout << \*itr++ << endl;

1. Các thao tác trên container cần Iterators

Có 3 phương thức phổ biến nhất cần iterator trên List/vector là thêm, xóa hay chèn phần tử

* insert(iterator **pos**, const Object &**x**): chèn đối tượng x theo sau vị trí pos trong List
* iterator erase( iterator **pos** ): xóa phần tử tham chiếu bởi iterator pos trong List.

Thời gian chèn và xóa node trên list luôn là O(1), nhưng vector là O(n). Hàm erase trả về iterator đại diện phần tử theo sau phần tử mới xóa.

Iterator **pos** sẽ không còn hiệu lực.

* iterator erase( iterator **start**, iterator **end** ): xóa tất cả phần tử từ iterator start cho tới, nhưng không bao gồm **end**. Để xóa toàn bộ List thì chỉ cần gọi c.erase( c.**begin**( ), c.**end**( ) ).

1. Hiệu suất trên container

Chúng ta duyệt List và dùng earase trên mỗi phần tử sau phần tử đầu tiên.

Phương thức erase trên List là tuyến tính, mỗi lần gọi mất một khoảng thời gian nhất định, nhưng trên vector sẽ tốn bình phương thời gian vì mỗi lần gọi không hiệu quả.

Chúng ta sử dụng template để da dạng hóa các container.

template <typename Container>

void removeEveryOtherItem( Container & lst ){

auto itr = lst.begin( ); // itr: Container::iterator

while( itr != lst.end( ) )

{

itr = lst.erase( itr );

if( itr != lst.end( ) )

++itr;

}

}

Dùng từ khóa **auto** để trình biên dịch tự nhận biết kiểu iterator của bất kì container nào.

Một list<int>, mất 0.039 giây cho 800,000 phần tử, và 0.073 giây cho 1,600,000 phần tử, thời gian tăng tuyến tính theo số lượng phần tử.

Tương tự, vector<int>, mất 5 phút cho 800,000 phần tử và 20 phút cho 1,600,000 phần tử, tức là thời gian tăng gấp 4, cho lương phần tử tăng gấp 2.

1. Iterator và const\_iterators

\*iterator không những trả về giá trị mà còn tham chiếu tới đối tượng trong container. Lợi ích là chúng ta hoàn toàn có thể thay đổi giá trị cho từng phần tử.

Phương thức sau dùng generic code cho mọi kiểu, áp dụng trên cả vector hay list, thời gian thực thi tuyến tính:

template <typename Container, typename Object>

void change( Container & c, const Object & newValue ){

typename Container::iterator itr = c.begin( );

while( itr != c.end( ) )

\*itr++ = newValue;

}

Giả sử rằng chúng ta không muốn thay đổi dữ liệu và truyền tham chiếu hằng cho Container c. Trình biên dịch đảm bảo không cho phép các hàm thành viên mutator của c được gọi.

template <typename Container>

void print( const list<int> & lst, ostream & out = cout ){

typename Container::iterator itr = lst.begin( );

while( itr != lst.end( ) ){

out << \*itr << endl;

\*itr = 0; // Hợp lệ? !!!

++itr;

}

}

Đoạn code trên không hợp lệ và sẽ không được biên dịch.

STL cung cấp giải pháp cho vấn đề trên, bằng kiểu dữ liệu hằng iterator lồng bên trong container là **const\_iterator**.

Điểm khác biệt chủ yếu của iterator và const\_iterator là toán tử tham chiếu nội dung \*const\_iterator là kiểu r\_value.

Ngoài ra trình biên dịch bắt buộc bạn phải dùng const\_iterator để duyệt hằng container.

Tương tự như begin/end, chúng ta có phương thức hằng để lấy const\_iterator như sau:

* iterator begin( )
* const\_iterator begin( ) const
* iterator end( )
* const\_iterator end( ) const

Khi biến container là non constant, mutator begin sẽ được gọi. Ngược lại, container là hằng thì phiên bản begin const được gọi.

Nếu dùng **auto**, trình biên dịch sẽ tự nhận biết kiểu theo từng hoàn cảnh một cách chính xác hoàn toàn.

C++11 cho phép viết code làm việc dù kiểu container không có hàm thành viên begin và end.

Hàm không thành viên begin và end được định nghĩa để dùng begin(c) thay thế c.begin()tại bất kỳ nơi nào được phép

Generic code dùng begin(c) thay vì c.begin() thuận lợi hơn, nhờ làm việc được với container có hay không có thành viên begin và end.

template <typename Container>

void print( const Container & c, ostream & out = cout ) {

if( c.empty( ) )

out << "(empty)";

else

{

auto itr = begin( c ); // itr is a Container::const\_iterator

out << "[ " << \*itr++; // Print first item

while( itr != end( c ) )

out << ", " << \*itr++;

out << " ]" << endl;

}

}

Ngoài ra, hàm tự do begin và end trong C++11 có sử dụng **auto** và **decltype**, nên chúng tự động xác định được kiểu trả về thông qua kiểu tham số truyền vào.

template<typename Container>

auto begin( Container & c ) -> decltype( c.begin( ) )

{

return c.begin( );

}

template<typename Container>

auto begin( const Container & c ) -> decltype( c.begin( ) )

{

return c.begin( );

}

1. Duyệt vector<>

Ví dụ về iterator duyệt qua một vector

int main(){

vector<int> vect;

for (int nCount=0; nCount < 6; nCount++)

vect.push\_back(nCount);

vector<int>::const\_iterator it; // khai báo iterator hằng

it = vect.begin(); // gán cho phần begin() của vector

while (it != vect.end()) // lắp tới khi gặp end()

{

cout << \*it << " "; // in ra giá trị mà nó duyệt qua

++it; // trỏ sang phần tử kế tiếp

}

cout << endl;

return 0;

}

Kết quả

0 1 2 3 4 5

1. Duyệt List<>

Ví dụ tương tự cho list:

#include <iostream>

#include <list>

using namespace std;

int main(){

list<int> li;

for (int nCount=0; nCount < 6; nCount++)

li.push\_back(nCount);

list<int>::const\_iterator it;

it = li.begin();

while (it != li.end())

{

cout << \*it << " ";

++it;

}

cout << endl;

}

Kết quả

0 1 2 3 4 5

1. Khoảng cách giữa 2 phần tử

std::distance trả về khoảng cách giữa first và last

template< class InputIt >

typename std::iterator\_traits<InputIt>::difference\_type

distance( InputIt first, InputIt last );

* First: iterator trỏ tới phần tử đầu tiên
* Last: iterator trỏ tới phần tử cuối cùng
* InputIt : tham số template đại diện cho kiểu iterator.

Giá trị trả về có thể âm nếu iterator truy xuất ngẫu nhiên và **first** lại nằm ở **last**

Hàm có độ phức tạp tuyến tính. Tuy nhiên, nếu InputIt thỏa mãn RandomAccessIterator thì độ phức tạp là hằng số.

Ví dụ:

#include <iostream>

#include <iterator>

#include <vector>

int main()

{

std::vector<int> v{ 3, 1, 4 };

std::cout << "distance(first, last) = "

<< std::distance(v.begin(), v.end()) << '\n'

<< "distance(last, first) = "

<< std::distance(v.end(), v.begin()) << '\n';

}

Output:

distance(first, last) = 3

distance(last, first) = -3

1. Tại sao phải dùng iterator

Iterator cho phép bạn duyệt lớp chứa dễ dàng mà không cần biết cấu trúc của lớp chứa thế nào.

Toán tử truy xuất chỉ số phần tử [] chỉ được định nghĩa trên lớp chứa vector, nhưng nhiều lớp chứa khác không có (Như list, forward\_list hay các lớp chứa liên kết)

Iterator giúp bạn làm việc độc lập trên nhiều lớp chứa nếu bạn không chắc về khả năng truy xuất ngẫu nhiên hay phương thức .**size**()

Để sử dụng hiệu quả các lớp STL Algorithm, bắt buộc bạn phải dùng iterator. Iterator thể hiện sức mạnh rõ rệt khi kết hợp giữa STL algorithm và hàm thành viên của lớp chứa.

****Trong bài tiếp theo, bạn sẽ xem ví dụ về cách dùng iterator để chèn phần tử vào một danh sách (mà không dùng toán tử [] để truy xuất trực tiếp phần tử).

**Một lưu ý: iterator phải được cung cấp trong mỗi lớp cơ bản, vì iterator cần biết cách một lớp được triển khai. Do đó, iterator luôn cột chặt với lớp chứa cụ thể.**

##### STL Algorithm

1. Giới thiệu

Gồm một lượng lớn các giải thuật chung định nghĩa trong thư viện <algorithm> để làm việc với các lớp chứa. Cho phép các bạn tìm kiếm, sắp xếp, chèn, xóa và sao chép phần tử.

**Lưu ý rằng các giải thuật được triển khai như là các hàm toàn cục, có sử dụng iterator.**

Mỗi giải thuật chỉ định nghĩa một lần, nhưng lại làm việc với mọi lớp chứa có iterator (kể cả lớp chứa bạn tự định nghĩa).

Mặc dù STL Algorithm giúp bạn thao tác dễ dàng, nhanh chóng trên lớp chứa nhưng có điểm tối là : một vài lớp chứa lại không làm việc với Algorithm, gây ra lặp vô hạn hay tốn hiệu suất.

Chúng ta khai báo một list như sau:

list<int> li;

for (int nCount=0; nCount < 6; nCount++)

li.push\_back(nCount);

1. Phần tử nhỏ nhất và lớn nhất

Dùng std::min\_element và std::max\_elment:

template< class ForwardIt >

ForwardIt min\_element( ForwardIt first, ForwardIt last );

template< class ForwardIt >

ForwardIt max\_element(ForwardIt first, ForwardIt last);

* first, last: là forward iterators qui định khoảng cần tìm

Vi dụ:

#include <algorithm>

#include <iostream>

#include <vector>

#include <cmath>

static bool abs\_compare(int a, int b)

{

return (std::abs(a) < std::abs(b));

}

int main()

{

std::vector<int> v{ 3, 1, -14, 1, 5, 9 };

std::vector<int>::iterator result;

result = std::max\_element(v.begin(), v.end());

std::cout << "max element at: " << std::distance(v.begin(), result) << '\n';

result = std::max\_element(v.begin(), v.end(), abs\_compare);

std::cout << "max element (absolute) at: " << std::distance(v.begin(), result);

}

Output:

max element at: 5

max element (absolute) at: 2

Phiên bản có dùng tham số thứ 3 để định nghĩa cách so sánh:

template< class ForwardIt, class Compare >

ForwardIt min\_element( ForwardIt first, ForwardIt last, Compare cmp );

template< class ForwardIt, class Compare >

ForwardIt max\_element(ForwardIt first, ForwardIt last, Compare cmp);

* cmp: là đối tượng hàm để so sánh các phần tử với nhau. std::min\_element sẽ dùng phép toán (<), std::max\_elment dùng phép toán (>), cmp phải có dạng:

Ví dụ:

1. Tìm kiếm std::find

Cú pháp:

template< class InputIt, class T >

InputIt find( InputIt first, InputIt last, const T& value );

Để tìm ra iterator trong lớp chứa, sau đó dùng

* first, last: iterator hay con trỏ qui định khoảng cần tìm
* value: giá trị cần tìm

Hàm trả về Iterator hay con trỏ tới phần tử tìm thấy hoặc trả về **last** nếu không tìm thấy.

The overloads with a template parameter named ExecutionPolicy report errors as follows:

If execution of a function invoked as part of the algorithm throws an exception and ExecutionPolicy is one of the three standard policies, std::terminate is called. For any other ExecutionPolicy, the behavior is implementation-defined.

If the algorithm fails to allocate memory, std::bad\_alloc is thrown.

#include <iostream>

#include <algorithm>

#include <vector>

#include <iterator>

int main()

{

int n1 = 3;

int n2 = 5;

std::vector<int> v{0, 1, 2, 3, 4};

auto result1 = std::find(std::begin(v), std::end(v), n1);

auto result2 = std::find(std::begin(v), std::end(v), n2);

if (result1 != std::end(v)) {

std::cout << "v contains: " << n1 << '\n';

} else {

std::cout << "v does not contain: " << n1 << '\n';

}

if (result2 != std::end(v)) {

std::cout << "v contains: " << n2 << '\n';

} else {

std::cout << "v does not contain: " << n2 << '\n';

}

}

Output:

v contains: 3

v does not contain: 5

1. Sắp xếp std::sort

Chúng ta khai báo một vector như sau:

vector<int> vect;

vect.push\_back(7);

vect.push\_back(-3);

vect.push\_back(6);

vect.push\_back(2);

vect.push\_back(-5);

vect.push\_back(0);

vect.push\_back(4);

Dùng hàm

**sort**(<iterator\_bắt\_đầu>,<iterator\_kết\_thúc>,<hàm\_tính\_toán>)

* iterator\_bắt\_đầu: có thể là địa chỉ, con trỏ đầu mảng hay iterator bắt đầu lớp chứa
* iterator\_kết\_thúc: địa chỉ, con trỏ đầu mảng hay iterator kết thúc lớp chứa
* hàm\_tính\_toán: qui định hàm sắp xếp theo thuộc tính nào của đối tượng phần tử trong mảng/lớp chứa.

**Lưu ý: hai tham số đầu tiên có thể tùy biến, không nhất thiết luôn là đầu hay cuối, có thể là một đoạn trong mảng hay lớp chứa cần sắp xếp.**

ĐỂ SẮP XẾP TĂNG DẦN

**sort**(vect.begin(), vect.end()); // sắp xếp

vector<int>::const\_iterator it; // khai báo iterator chỉ đọc

for (it = vect.begin(); it != vect.end(); it++) // duyệt

cout << \*it << " ";

cout << endl;

DÙNG THAM SỐ THỨ 3

Tham số này qui định hai tiêu chí sau:

* Tăng hay giảm dần
* Sắp xếp theo thuộc tính nào (của đối tượng)

Hàm tính toán phải có dạng như sau:

**bool** <tên\_hàm>(**const** <kiểu> &<đối\_tượng\_1>, **const** <kiểu> &<đối\_tượng\_2>){

**return** <đối\_tượng\_1>.<thuộc\_tính> <toán\_tử\_quan\_hệ> <đối\_tượng\_1>.<thuộc\_tính>;

}

* <toán\_tử\_quan\_hệ> : hàm sort() dựa vào biểu thức quan hệ trong hàm tính toán để sắp xếp (> giảm dần, < tăng dần) theo thuộc tính nào của đối tượng.

Ví dụ

Khai báo cấu trúc Person:

struct Person{

string name;

int age;

};

Khai báo hàm tính toán sắp tăng dần cho hai thuộc tính

// hàm sắp theo name

bool sortByName(const Person &p1, const Person &p2) { return p1.name < p2.name; }

// hàm sắp theo age

bool sortByAge(const Person &p1, const Person &p2) { return p1.age < p2.age; }

Ngoài ra bạn có thể quá tải hàm tính toán cho các thuộc tính tăng hay giảm dần. Sau đó truyền đúng tên hàm cần vào sort()

Sắp xếp theo name:

// Sap xep theo name

sort(people.begin(), people.end(), **sortByName**);

for (Person &n : people)

cout << n.name << " ";

Sắp xếp theo age:

// Sap xep theo age

sort(people.begin(), people.end(), **sortByAge**);

for (Person &n : people)

cout << n.age << " ";



**Lưu ý hàm sort() sẽ không làm việc với lớp chứa list – lớp này đã có hàm thành viên .sort() , làm việc hiệu quả hơn so với phiên bản chung của algorithm.**

1. Đảo ngược mảng std::reverse

Cú pháp

template< class BidirIt >

void reverse( BidirIt first, BidirIt last );

* first, last: Khoảng phần tử cần đảo vị trí.

Exceptions

The overload with a template parameter named ExecutionPolicy reports errors as follows:

If execution of a function invoked as part of the algorithm throws an exception and ExecutionPolicy is one of the three standard policies, std::terminate is called. For any other ExecutionPolicy, the behavior is implementation-defined.

If the algorithm fails to allocate memory, std::bad\_alloc is thrown.

Ví dụ:

#include <vector>

#include <iostream>

#include <iterator>

#include <algorithm>

int main()

{

std::vector<int> v({1,2,3});

std::reverse(std::begin(v), std::end(v));

std::cout << v[0] << v[1] << v[2] << '\n';

int a[] = {4, 5, 6, 7};

std::reverse(std::begin(a), std::end(a));

std::cout << a[0] << a[1] << a[2] << a[3] << '\n';

}

Output:

321

7654

1. Xử lí tự động collection bằng std::for\_each

Mục đích của chúng ta là xử lí cùng một phương thức trên hàng loạt phần tử trong collection

Cú pháp:

template< class InputIt, class UnaryFunction >

UnaryFunction for\_each( InputIt first, InputIt last, UnaryFunction f );

Tham số thứ 3 sẽ là con trỏ hàm hay Functor nhận 1 tham số cho phần tử trên collection. Hàm phải có dạng như sau:

template<class Type>

void fun(const Type &a);

template<class Type>

void fun(Type a);

Kiểu **Type** phải khớp với kiểu dữ liệu của **iterator**. Nếu con trỏ hàm có trả về thì sẽ bị bỏ qua.

Lưu ý: Nếu tham số thứ 3 là Functor thì không được dùng hoặc phải **delete** hàm dựng và phép gán Movie semantic

Ví dụ:

struct Sum{

Sum(){sum = 0;}

void operator()(int n) { sum += n; }

int sum;

};

int main(){

std::vector<int> nums{4,5,3,9,16,268};

// gọi Sum::operator() cho mỗi phần tử

Sum s = std::for\_each(nums.begin(), nums.end(), Sum());

// xuất từng phần tử

for (auto const &n : nums) {

std::cout << ' ' << n;

}

std::cout << '\n';

std::cout << "sum: " << s.sum << '\n';

return 0;

}

Kết quả:

4 5 3 9 16 268

sum: 305

1. Sao chép mảng std::copy\_backward

**Std::copy\_backward** sẽ sao chép các phần tử tuần tự theo chiều ngược (phần tử cuối chép trước) trong khoảng [first, last) của mảng cũ sang mảng mới.

Cú pháp:

template< class BidirIt1, class BidirIt2 >

BidirIt2 copy\_backward( BidirIt1 first, BidirIt1 last, BidirIt2 d\_last );

* first, last: iterator hoặc con trỏ thuần qui định phạm vi
* d\_last : hàm sẽ sao chép tới phần tử cuối cùng là d\_last sang mảng mới
* Tham số template **BidirIt** phải là iterator 2 chiều.

Hàm trả về iterator hoặc vị trí phần tử cuối cùng được sao chép.

Sẽ xảy ra lỗi nếu d\_last không phải phần tử cuối trong mảng mới, khi đó phải dùng **std::copy**

Ví dụ:

#include <algorithm>

#include <iostream>

#include <vector>

int main()

{

std::vector<int> from\_vector;

for (int i = 0; i < 10; i++)

from\_vector.push\_back(i);

std::vector<int> to\_vector(15);

std::copy\_backward(from\_vector.begin(), from\_vector.end(), to\_vector.end());

std::cout << "to\_vector contains: ";

for (auto i: to\_vector) {

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

}

}

Output:

**to\_vector contains: 0 0 0 0 0 0 1 2 3 4 5 6 7 8 9**

****

**Lưu ý: When copying overlapping ranges, std::copy is appropriate when copying to the left (beginning of the destination range is outside the source range) while std::copy\_backward is appropriate when copying to the right (end of the destination range is outside the source range).**

1. Tìm phần tử cận trên std::upper\_bound

std::upper\_bound dùng để tìm vị trí phần tử có giá trị cận trên trong mảng [first, last).

Phiên bản 1 dùng quá tải phép toán (<) của đối tượng để so sánh các phần tử:

template< class ForwardIt, class T >

ForwardIt upper\_bound( ForwardIt first, ForwardIt last, const T& value );

* first, last: iterator hay kiểu con trỏ đối tượng đầu và cuối trong mảng phần tử cần xét
* value: giá trị tiệm cận trên
* Hàm trả về iterator hoặc vị trí phần tử tìm thấy đầu tiên trong khoảng [**first**, **last**) có giá trị <= **value** hoặc trả về **last** nếu không tìm thấy phần tử nào.

std::upper\_bound phiên bản 2 dùng đối tượng hàm **comp** định nghĩa cách so sánh phần tử trong mảng:

template< class ForwardIt, class T, class Compare >

ForwardIt upper\_bound( ForwardIt first, ForwardIt last, const T& value, Compare comp );

Khoảng [first, last) nên được sắp xếp trước khi dùng std::upper\_bound

Cả phiên bản 1 và 2 dùng quá tải toán tử (>) giữa phần tử so sánh với value: Element > value hay comp(Element, value)

Triển khai **comp** có cú pháp:

template<class Type1, class Type2 >

bool pred(const Type1 &a, const Type2 &b);//const để không làm thay đổi giá trị phần tử

* Type1 : đại diện cho kiểu iterator **ForwardIt**
* Type2 : đại diện cho đối tượng kiểu **T**

Ví dụ: sử dụng std::upper\_bound trên vector

#include <algorithm>

#include <iostream>

#include <iterator>

#include <vector>

int main()

{

std::vector<int> data = { 1, 1, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6 };

auto lower = std::lower\_bound(data.begin(), data.end(), 4);

auto upper = std::upper\_bound(data.begin(), data.end(), 4);

std::copy(lower, upper, std::ostream\_iterator<int>(std::cout, " "));

}

Output:

4 4 4

1. Tìm phần tử cận dưới std::lower\_bound

std:: lower \_bound dùng để tìm vị trí phần tử có giá trị cận dưới cho trước trong mảng [first, last).

Phiên bản 1 dùng quá tải phép toán (<) của đối tượng để so sánh các phần tử:

template< class ForwardIt, class T >

ForwardIt lower\_bound( ForwardIt first, ForwardIt last, const T& value );

* first, last: iterator hay kiểu con trỏ đối tượng đầu và cuối trong mảng phần tử cần xét
* value: giá trị tiệm cận dưới
* Hàm trả về iterator hoặc vị trí phần tử tìm thấy đầu tiên trong khoảng [**first**, **last**) có giá trị >= **value** hoặc trả về **last** nếu không tìm thấy phần tử nào.

std::lower\_bound phiên bản 2 dùng đối tượng hàm **comp** định nghĩa cách so sánh phần tử trong mảng:

template< class ForwardIt, class T, class Compare >

ForwardIt lower\_bound( ForwardIt first, ForwardIt last, const T& value, Compare comp );

Khoảng [first, last) nên được sắp xếp trước khi dùng std::lower\_bound

Cả phiên bản 1 và 2 dùng quá tải toán tử (<) giữa phần tử so sánh với value: Element < value hay comp(Element, value)

Triển khai **comp** có cú pháp:

template<class Type1, class Type2 >

bool pred(const Type1 &a, const Type2 &b);//const để không làm thay đổi giá trị phần tử

* Type1 : đại diện cho kiểu iterator **ForwardIt**
* Type2 : đại diện cho đối tượng kiểu **T**

##### Std::vector

Tương tự như std::array, thư viện chuẩn C++ cung cấp một template khác là std::vector, để làm việc với mảng động an toàn và thuận lợi hơn.

Std::vector có các tính năng gần giống với mảng tĩnh truyền thống.

Chúng ta sẽ tìm hiểu tại sao std::vector lại trở thành công vụ hữu dụng và linh hoạt nhất trong C++.

1. Giới thiệu std::vector

Ra mắt từ C++ 03, std::vector cung cấp các chức năng mảng động để quản lý vùng nhớ được cấp phát. Nghĩa là bạn có thể tạo mảng với kích thước tùy biến trong thời gian chạy, nhưng không cần thao tác cấp phát và hủy cấp phát vùng nhớ qua **new** và **delete** như mảng động truyền thống.

Std::vector được khai báo trong thư viện **<vector>**

1. Khởi tạo mảng

Cú pháp:

std::vector<kiểu\_dữ\_liệu> tên\_biến;

std::vector<kiểu\_dữ\_liệu> tên\_biến = {danh sách khởi tạo};

std::vector<kiểu\_dữ\_liệu> tên\_biến {danh sách khởi tạo chuẩn};// C++11

Bạn không cần cung cấp số phần tử mảng khi khai báo, std::vector cấp phát tĩnh theo nhu cầu người dùng.

Ví dụ:

#include <vector>

std::vector<int> array; // khai báo không có số phần tử

std::vector<int> array2 = { 9, 7, 5, 3, 1 }; // dùng danh sách khởi tạo

std::vector<int> array3 { 9, 7, 5, 3, 1 }; // dùng khởi tạo chuẩn (C++11 trở đi)

1. Truy xuất phần tử

Bạn vẫn dùng toán tử [] để truy xuất, nhưng không kiểm tra ranh giới mảng. Hàm thành viên at() giúp bạn truy xuất phần tử mảng có kiểm tra ranh giới.

array[2] = 2;

array.at(3) = 3;

Trên C++11, bạn có thể gán danh sách khởi tạo cho std::vector

array = { 0, 1, 2, 3, 4 }; // okay, array size is now 5

array = { 9, 8, 7 }; // okay, array size is now 3

1. Tự động hủy vùng nhớ

Khi một biến vector đi ra ngoài phạm vi, nó tự động hủy vùng nhớ cấp phát mà nó quản lý (nếu cần). Không những thế, nó còn ngăn bộ nhớ bị thiếu hụt.

Xem đoạn code sau:

void doSomething(bool earlyExit){

int \*array = new int[5] { 9, 7, 5, 3, 1 };

if (earlyExit)

return 0;

// do stuff here

delete[] array; // never called

}

Nếu **earlyExit** = true, hàm trả về và vùng nhớ cấp phát không bị hủy.

Nếu biến **array** là kiểu vector, vùng nhớ cấp phát tự động bị hủy ngay lập tức khi đi ra ngoài phạm vi hàm (bất chấp hàm có thoát ra sớm hay không)

1. Kích thước mảng

Std::vector có lưu trữ kích thước nội bộ, bạn dùng hàm thành viên **size()** để lấy:

#include <iostream>

#include <vector>

using namespace std;

int main(){

vector<int> array { 9, 7, 5, 3, 1 };

cout << "The size is: " << array.size() << '\n';

return 0;

}

Kết quả:

The size is: 5

1. Co giãn kích thước

Co giãn mảng động truyền thống khá phức tạp với các phần tử là đối tượng. Đây là một nguyên nhân mà bạn nên dùng std::vector.

Bạn dùng hàm thành viên std::vector<>::resize(kích\_thước)

#include <vector>

#include <iostream>

using namespace std;

int main(){

vector<int> array { 0, 1, 2 };

array.resize(5); // set size to 5

cout << "The size is: " << array.size() << '\n';

for (auto const &element: array)

cout << element << ' ';

return 0;

};

Kết quả

The size is: 5

0 1 2 0 0

**Lưu ý: Khi co giãn, giá trị phần tử được bảo lưu.**

**Phần tử mới sẽ được khởi tạo giá trị mặc định (vd: kiểu số là zero)**

1. Nén dữ liệu kiểu bools

Std::vector có một mẹo để nén vùng nhớ mảng các giá trị luận lý.

Đoạn code sau minh họa việc khai báo mảng std::vector kiểu bool chứa 8 chân trị trong 1-bytes nhớ

#include <vector>

#include <iostream>

using namespace std;

int main(){

vector<bool> array { true, false, false, true, true };

cout << "The size is: " << array.size() << '\n';

for (auto const &element: array)

cout << element << ' ';

return 0;

};

Kết quả

The size is: 5

1 0 0 1 1

The way the boolean version of std::vector works behind the scenes is pretty complicated. When you use the subscript Thao tác trên std::vector

1. Thao tác trên vector

Đây là một số thao tác phổ biến :

* **Push\_back(value)** : hàm thành viên chèn một phần tử có giá trị **value** vào cuối vector, tự mở rộng kích thước nếu cần.
* **Size() :** lấy số phần tử của vector

Xem ví dụ sau:

#include <iostream>

#include <vector>

using namespace std;

int main(){

// tao vector kiểu int

vector<int> vec;

int i;

// xuất kích thước bạn đầu của vector

cout << "vector size = " << vec.size() << endl;

// đẩy 5 phần tử vào cuối vector

for(i = 0; i < 5; i++){

vec.push\_back(i);

}

// xuất kích thước sau khi mở rộng của vector

cout << "extended vector size = " << vec.size() << endl;

// xuất giá trị 5 phần tử đầu của vector

for(i = 0; i < 5; i++){

cout << "value of vec [" << i << "] = " << vec[i] << endl;

}

// dùng iterator để truy xuất giá trị các phần tử

vector<int>::iterator v = vec.begin();

while( v != vec.end()) {

cout << "value of v = " << \*v << endl;

v++;

}

return 0;

}

vector size = 0

extended vector size = 5

value of vec [0] = 0

value of vec [1] = 1

value of vec [2] = 2

value of vec [3] = 3

value of vec [4] = 4

value of v = 0

value of v = 1

value of v = 2

value of v = 3

value of v = 4

1. Khai báo mảng hai chiều

Bạn có thể tạo std::vector của std::vectors. Bạn dùng một chút mẹo vặt trong cú pháp :

vector<vector<int>> vector2Chieu;

Nếu bạn muốn khai báo trước kích thước vector , làm như sau:

int dong = 5;

int cot = 5;

vector<vector<int>> vector2Chieu(dong,vector<int>(cot));

vector<vector<int>> vector2Chieu(5,vector<int>(5));

Trong C++11, bạn có thể khởi tạo qua danh sách khởi tạo chuẩn:

vector<vector<int>> vector2Chieu { { 1, 1, 1 }, { 2, 2, 2 } };

1. Tổng kết

Phần này giới thiệu một cách cơ bản về lớp khuôn mẫu std::vector. Trong bài tiếp theo, chúng ta tìm hiểu các tính năng khác của vector, kích thước vùng nhớ và kích thước mảng cũng như cách quản lý vùng nhớ cấp phát.

Với các ưu điểm

* Tự quản lý vùng nhớ
* Bảo lưu nội bộ kích thước mảng
* Khả năng co giãn an toàn

Chúng ta nên dùng std::vector thay cho mảng động

##### std::array C++11

Mảng tĩnh và động có sẵn trong C++ có điểm dở là: khi tham chiếu mảng bằng con trỏ, thông tin về kích thước của mảng là không biết được, và mảng tĩnh hay có vấn đề khi điều chỉnh kích thước vùng nhớ cấp phát.

Để giải quyết, thư viện chuẩn C++11 đã phát triển một bộ quản lý mảng, std::array và std::vector.

Được biết tới bắt đầu từ C++11, std::array cho phép tạo mảng tĩnh. Với std::array, bạn không dùng con trỏ để tham chiếu khi truyền vào hàm.

std::array được khai báo trong thư viện **<array>**

1. Khởi tạo mảng

Cú pháp khai báo:

std::array < kiểu\_dữ\_liệu, số\_phần\_tử > tên\_biến;

std::array < kiểu\_dữ\_liệu, số\_phần\_tử > tên\_biến = {danh sách khởi tạo};

std::array < kiểu\_dữ\_liệu, số\_phần\_tử > tên\_biến {danh sách khởi tạo chuẩn } ; //C++11

Ví dụ khai báo std::array

#include <array>

std::array<int, 3> myarray; // khai báo mảng số nguyên có độ dài là 3

Giống như mảng tĩnh, kích thước std::array phải định nghĩa trước. Std::array cũng có danh sách khởi tạo hay khởi tạo chuẩn C++11.

std::array<int, 5> myarray = { 9, 7, 5, 3, 1 }; // danh sách khởi tạo

std::array<int, 5> myarray2 { 9, 7, 5, 3, 1 }; // khởi tạo chuẩn C++11

Khác với mảng tĩnh, std::array không thể tạo mảng mà chưa khai báo kích thước khi dùng danh sách khởi tạo

std::array<int, > myarray = { 9, 7, 5, 3, 1 }; // error, chưa khai báo kích thước

Ngoài ra bạn có thể dùng toán tử gán cho danh sách khởi tạo

myarray = { 0, 1, 2, 3, 4 }; // okay

myarray = { 9, 8, 7 }; // okay, phần tử 3 và 4 sẽ bằng zero

myarray = { 0, 1, 2, 3, 4, 5 }; // error, danh sách khởi tạo dài hơn kích thước mảng

1. Tự động hủy mảng

Chúng ta không cần gọi phương thức hủy nào cả, std::array sẽ tự hủy khi ra ngoài phạm vi.

1. Truy xuất phần tử với at()

Bạn vẫn dùng toán tử [] để truy xuất phần tử : :

std::cout << myarray[1];

myarray[2] = 6;

Nhưng toán tử [] sẽ không kiểm tra ranh giới phần tử mảng. Bạn nên dùng phương thức .**at(<index>)** thay thế vì nó có kiểm tra ranh giới mảng:

std::array<int, 5> myarray { 9, 7, 5, 3, 1 };

myarray.at(1) = 6; // gán phần tử thứ 1 = 6

myarray.at(9) = 10; // phần tử thứ 9 không có, error

Ở đoạn mã trên, trình biên dịch kiểm tra phần tử thứ nhất có tồn tại không, nếu có nó trả về tham chiếu tới phần tử thứ nhất và gán 6 vào.

Tuy nhiên vì phần tử thứ 9 không có, nó sẽ ném error, ngắt chương trình luôn ( thực ra là nó ném ra một ngoại lệ xử lí lỗi std::out\_of\_range mà chúng ta sẽ nói trong phần [**Exception Handling**](#_C++_-_Error))

1. Số phần tử mảng

Bạn dùng hàm thành viên **size()** để lấy số phần tử mảng.

std::array<double, 5> myarray { 9.0, 7.2, 5.4, 3.6, 1.8 };

std::cout << "size: " << myarray.size();

Kết quả:

size: 5

1. Truyền tham số

Bạn truyền tham số std::array vào hàm như sau:

#include <iostream>

void printSize(const std::array<double, 5> &myarray){

std::cout << "size: " << myarray.size();

}

int main(){

std::array<double, 5> myarray { 9.0, 7.2, 5.4, 3.6, 1.8 };

printSize(myarray);

return 0;

}

Kết quả:

size: 5



**Lưu ý: thành viên size() trả về số phần tử mảng chứ không phải kích thước vùng nhớ của mảng.**

**Bạn luôn truyền tham chiếu hằng std::array vào hàm, để trình biên dịch không tạo bản sao của mảng.**

1. Duyệt mảng

Bạn dùng vòng lặp for-each duyệt std::array

std::array<int, 5> myarray { 9, 7, 5, 3, 1 };

for (auto &element : myarray) //Mẹo là dùng biến auto để trình biên dịch tự nhận diện kiểu dữ liệu

std::cout << element << ' ';

Bạn tham khảo từ khóa auto C++11 [**tại đây**](#_Từ_khóa_auto)

Bạn chú ý vòng lặp for dùng biến hằng tham chiếu const &**element**, để không tạo ra bản sao của phần tử **myarray** và tránh ánh hưởng tới hiệu suất chương trình.

Bạn duyệt mảng dùng for với chiều dài mảng myarray.**size**() cũng được.

1. Sắp xếp

Bạn dùng phương thức std::sort() trong thư viện <algorithm> cùng với iterator để sắp xếp std::array :

#include <iostream>

#include <array>

#include <algorithm> // for std::sort

int main(){

std::array<int, 5> myarray { 7, 3, 1, 9, 5 };

std::sort(myarray.begin(), myarray.end()); // sắp xếp mảng tăng dần

std::sort(myarray.rbegin(),myarray.rend()); // sắp xếp mảng giảm dần

for (const auto &element : myarray)

std::cout << element << ' ';

return 0;

}

Kết quả

1 3 5 7 9

Hàm sort() có sử dụng **iterators,** chúng ta tìm hiểu trong phần khác.

1. Tổng kết

std::array là sự thay thế hoàn hảo cho mảng tĩnh:

* Nó hiệu quả, hỗ trợ nhiều thao tác thuận lợi , nhanh chóng mà không tốn thêm nhiều bộ nhớ.
* Điểm yếu duy nhất là cú pháp khai báo hơi rường rà, phải khai báo số phần tử (hàm dựng đối tượng không tính sẵn cho bạn khi khởi tạo) và không thể co giãn kích thước như std::vector.

##### Set và map trong STL

STL hỗ trợ thêm 2 container hỗ trợ tìm kiếm nhanh hơn so với vector và list: set và map.

Bảo đảm chi phí các thao tác chèn, xóa ,tìm kiếm theo logN.

Set và map dùng triển khai top-dow red-black tree

1. Sets

Set là cotainer có thứ tự, không cho phép đúp phần tử, cách truy xuất trên set cũng tương tự như vector, list.

Kiểu lồng iterator và const\_iterator cho phép duyệt set, nhiều phương thức triển khai trên vector, list đều có trên set:

* Begin
* End
* Size
* Empty

Phương thức print template cũng sẽ làm việc với set:

template <typename Container>

void print( const Container & c, ostream & out = cout ) {

if( c.empty( ) )

out << "(empty)";

else

{

auto itr = begin( c ); // itr is a Container::const\_iterator

out << "[ " << \*itr++; // Print first item

while( itr != end( c ) )

out << ", " << \*itr++;

out << " ]" << endl;

}

}

Các phương thức riêng triển khai trên set là : insert, remove và search cơ bản.

Vì set không cho phép đúp phần tử, nên có khả năng insert sẽ thất bại. Insert sẽ trả về iterator đại diện cho phần tử mới chèn, hoặc phần tử đã tồn tại trên set khi thất bại.

Cách trả về iterator khá tiện lợi khi cần xóa, vì chúng ta đã biết được vị trí phần tử, tránh thao tác tìm kiếm không cần thiết.

1. Pair

STL còn định nghĩa thêm lớp template **pair**, gồm 2 thành viên struct để truy xuất 2 phần tử theo cặp. Có 2 cách chèn:

* **pair**<iterator,bool> insert( const **Object** & x ): Hàm insert 1 tham số đã mô tả ở trên.
* **pair**<iterator,bool> insert( **iterator** hint, const **Object** & x ): Hàm insert 2 tham số định nghĩa thêm **hint**: ám chỉ vị trí phần tử chèn x nên tới

Nếu chỉ dẫn **hint** chính xác, thao tác chèn sẽ rất nhanh với O(1). Ngược lại, insert sẽ dùng thuật toán của nó.

Ví dụ: đoạn code sau chạy rất nhanh dùng insert 2 tham số

#include <set>

void initSet(){

set<int> s;

for( int i = 0; i < 1000000; ++i )

s.insert( s.end( ), i );

}

Có nhiều phiên bản erase:

* **int** erase( const **Object** & x ): xóa x nếu tìm thấy và trả về số phần tử đã xóa ( 0 hoặc 1)
* **iterator** erase( **iterator** itr ): Xóa theo vị trí iterator cung cấp, trả về iterator của phần tử kế tiếp, itr trở nên vô giá trị.
* **iterator** erase( **iterator** start, **iterator** end ): Xóa tất cả phần tử có iterator bắt đầu từ start , nhưng không gồm end.

Set không dùng contain() mà triển khai find() , trả về iterator đại diện của phần tử (hoặc iterator end nếu không tìm thấy).

* **iterator** find( const **Object** & x ) const;

Theo mặc định, việc sắp xếp dùng đối tượng hàm less<Object>, triển khai gọi toán tử bé hơn (<) của tham số Object.

Ngoài ra khi khởi tạo set, chúng ta có thể triển khai đối tượng hàm so sánh cho tham số thứ hai của set.

Ví dụ: một set string, không kiểm tra hoa thường dùng đối tượng hàm CaseInsensitiveCompare

set<string,CaseInsensitiveCompare> s;

s.insert( "Hello" ); s.insert( "HeLLo" );

cout << "The size is: " << s.size( ) << endl;

1. Maps

Kiểu map dùng chứa các phần tử chỉ gồm khóa và giá trị: <keyType, valueType>.

Key phải là duy nhất, nhưng nhiều key có thể ánh xạ tới cùng 1 giá trị.

Key được duy trì theo trật tự sắp xếp nhất định.

Hành vi của map giống với set, tất cả hàm so sánh xoay quanh key. Map hỗ trợ :

* begin, end, size và empty
* các phương thức insert, find và erase

Iterator là một cặp key-value.

Phương thức insert() nhận đối tượng pair<keyType,valueType> .

Phương thức find() nhận tham số key, trả về iterator, nếu không tìm thấy trả về iterator end()

Các phương thức trên hơi tốn thời gian, may mắn là ta có toán tử truy xuất ngẫu nhiên [ ] được quá tải cho map:

**ValueType** & operator[] ( const **KeyType** & key );

* Nếu key tồn tại trên map, tham chiếu tới giá trị được trả về.
* Nếu key không tìm thấy, nó được chèn với giá trị mặc định và trả về tham chiếu giá trị.

Giá trị mặc định ở đây sẽ là tham số zero cho hàm dựng hay zero với kiểu cơ bản.

Do quan điểm triển khai như trên nên map không có accessor cho toán tử [ ], ví dụ khi truyền tham số là const reference cho map, toán tử [ ] sẽ không dùng được.

1. Truy xuất set- map

Có hai kỹ thuật truy xuất trong map:

* Dùng toán tử [ ] chèn và gán giá trị
* Dùng iterator

map<string,double> salaries;

salaries[ "Pat" ] = 75000.00; //Chèn Pat, gán giá trị 75000

cout << salaries[ "Pat" ] << endl;

cout << salaries[ "Jan" ] << endl; //Chèn Jan, gán giá trị mặc định 0.0

map<string,double>::const\_iterator itr;

itr = salaries.find( "Chris" ); //Dùng find lấy iterator tương ứng với key = “Chris”

if( itr == salaries.end( ) )

cout << "Not an employee of this company!" << endl;

else

cout << itr->second << endl;

Nếu không tìm thấy itr sẽ trỏ tới iterator end(). Nếu thấy, chúng ta có thể dùng itr->second để thay đổi giá trị.

## Chuyên đề C++ 11

### C++11 – Chuỗi hằng kí tự

1. Chuỗi người dùng tự định nghĩa

C++ provides literals for a variety of built-in types (2.14 Literals):

123 // int

1.2 // double

1.2F // float

'a' // char

1ULL // unsigned long long

0xD0 // hexadecimal unsigned

"as" // string

However, in C++98 there are no literals for user-defined types. This can be a bother and also seen as a violation of the principle that user-defined types should be supported as well as built-in types are. In particular, people have requested:

"Hi!"s // string, not ``zero-terminated array of char''

1.2i // imaginary

123.4567891234df // decimal floating point (IBM)

101010111000101b // binary

123s // seconds

123.56km // not miles! (units)

1234567890123456789012345678901234567890x // extended-precision

C++11 supports ``user-defined literals'' through the notion of literal operators that map literals with a given suffix into a desired type. For example:

constexpr complex<double> operator "" i(long double d) // imaginary literal

{

return {0,d}; // complex is a literal type

}

std::string operator""s (const char\* p, size\_t n) // std::string literal

{

return string(p,n); // requires free store allocation

}

Note the use of constexpr to enable compile-time evaluation. Given those, we can write

template<class T> void f(const T&);

f("Hello"); // pass pointer to const char\*

f("Hello"s); // pass (5-character) string object

f("Hello\n"s); // pass (6-character) string object

auto z = 2+1i; // complex(2,1)

The basic (implementation) idea is that after parsing what could be a literal, the compiler always check for a suffix. The user-defined literal mechanism simply allows the user to specify a new suffix and what is to be done with the literal before it. It is not possible to redefine the meaning of a built-in literal suffix or augment the syntax of literals. A literal operator can request to get its (preceding) literal passed ``cooked'' (with the value it would have had if the new suffix hadn't been defined) or ``uncooked'' (as a string).

To get an ``uncooked'' string, simply request a single const char\* argument:

Bignum operator"" x(const char\* p)

{

return Bignum(p);

}

void f(Bignum);

f(1234567890123456789012345678901234567890x);

Here the C-style string "1234567890123456789012345678901234567890" is passed to operator"" x(). Note that we did not explicitly put those digits into a string.

There are four kinds of literals that can be suffixed to make a user-defined literal

* integer literal: accepted by a literal operator taking a single unsigned long long or const char\* argument.
* floating-point literal: accepted by a literal operator taking a single long double or const char\* argument.
* string literal: accepted by a literal operator taking a pair of (const char\*, size\_t) arguments.
* character literal: accepted by a literal operator taking a single char argument.

Note that you cannot make a literal operator for a string literal that takes just a const char\* argument (and no size). For example:

string operator"" S(const char\* p); // warning: this will not work as expected

"one two"S; // error: no applicable literal operator

The rationale is that if we want to have ``a different kind of string'' we almost always want to know the number of characters anyway.

Suffixes will tend to be short (e.g. s for string, i for imaginary, m for meter, and x for extended), so different uses could easily clash. Use namespaces to prevent clashes:

namespace Numerics {

// ...

class Bignum { /\* ... \*/ };

namespace literals {

operator"" X(char const\*);

}

}

using namespace Numerics::literals;

1. Hằng kí tự Raw

In many cases, such as when you are writing regular expressions for the use with the standard regex library, the fact that a backslash (\) is an escape character is a real nuisance (because in regular expressions backslash is used to introduce special characters representing character classes). Consider how to write the pattern representing two words separated by a backslash (\w\\\w):

string s = "\\w\\\\\\w"; // I hope I got that right

Note that the backslash character is represented as two backslashes in a regular expression. Basically, a "raw string literal" is a string literal where a backslash is just a backslash so that our example becomes:

string s = R"(\w\\\w)"; // I'm pretty sure I got that right

The original proposal for raw strings presents this as a motivating example

"('(?:[^\\\\']|\\\\.)\*'|\"(?:[^\\\\\"]|\\\\.)\*\")|" // Are the five backslashes correct or not?

// Even experts become easily confused.

The R"(...)" notation is a bit more verbose than the "plain" "..." but "something more" is necessary when you don't have an escape character: How do you put a quote in a raw string? Easy, unless it is preceded by a ):

R"("quoted string")" // the string is "quoted string"

So, how do we get the character sequence )" into a raw string? Fortunately, that's a rare problem, but "(...)" is only the default delimiter pair. We can add delimiters before and after the (...) in "(...)". For example

R"\*\*\*("quoted string containing the usual terminator (")")\*\*\*" // the string is "quoted string containing the usual terminator (")"

The character sequence after ) must be identical to the sequence before the (. This way we can cope with (almost) arbitrarily complicated patterns.

The initial R of a raw string can be preceded by an encoding-prefix: u8, u, U, or L. For example u8R"(fdfdfa)" is an UTF-8 string literal.

### C++11 –khởi tạo với { }

##### Khởi tạo C++ 11

1. Khởi tạo biến

Chúng ta có 4 cú pháp cơ bản cho việc khởi tạo giá trị của biến:

X a1 {v}; // Khuyến khích nên dùng thường xuyên , C++ 11

X a2 = {v}; // Cú pháp trên C

X a3 = v; // Cú pháp trên C

X a4(v);

Cú pháp đầu tiên có thể dùng trong nhiều hoàn cảnh vì nó rõ ràng, ít lỗi hơn. Đây là điểm mới trong C++11.

Hai cú pháp dùng “=” là kiểu cũ từ C. Do thói quen từ trước, chúng ta hay dùng “=” khi khởi tạo giá trị:

int x1 = 0;

char c1 = 'z';

Tuy nhiên, khởi tạo dùng { } phải tuân theo luật ép kiểu cơ bản:

* Chuyển kiểu số nguyên. Ví dụ: char sang int thì được, nhưng int không ép kiểu sang char được.
* Chuyển kiểu số dấu chấm động. Ví dụ: float sang double được phép, nhưng double sang float thì không được.
* Số dấm chấm động và số nguyên không chuyển kiểu qua lại được.

Ví dụ:

void f(double val, int val2)

{

int x2 = val; // val==7.9, x2 sẽ = 7

char c2 = val2; // val2==1025, c2 sẽ = 1

int x3 {val}; // error : possible truncation

char c3 {val2}; // error : possible narrowing

char c4 {24}; // OK: 24 có thể đại diện cho char

char c5 {264}; // error (assuming 8-bit chars): 264 không thể đại diện cho char

int x4 {2.0}; // error : no double to int value conversion

// ...

}

Như vậy khởi tạo dùng { } khá nghiêm ngặt.

1. Trường hợp dùng auto

Lưu ý khi dùng từ khóa **auto** để suy luận ra kiểu với cách khởi tạo dùng { }

Ví dụ:

auto z1 {99}; // z1 bị hiểu là kiểu danh sách initializer\_list<int>

auto z2 = 99; // z2 là kiểu int

Khuyến cáo chỉ nên dùng khai báo tường minh “=” với auto, không dùng “{}”.

1. Khởi tạo với đối tượng

{ } định nghĩa cách mới trong việc khởi tạo đối tượng:

vector<int> v1 {99}; // v1 là vector có 1 phần tử có giá trị 99

vector<int> v2(99); // v2 là vector có size = 99 phần tử với giá trị = 0

Các kiểu đối tượng nên có triển khai tường minh, phân biệt giữa hai kiểu khởi tạo:

vector<string> v1{"hello!"}; // v1 : vector có 1 phần tử = "hello!"

vector<string> v2("hello!"); // error : vector không có hàm dựng() nhận chuỗi

Kết luận: Nên dùng { } để khởi tạo giá trị thay cho cách khác, trừ khi bạn có lý do chính dáng để không dùng.

1. Giá trị khởi tạo mặc định

Khởi tạo trống { } ngầm định giá trị mặc định được dùng:

int x4 {}; // x4 = 0

double d4 {}; // d4 = 0.0

char∗ p {}; // p = nullptr

vector<int> v4{}; // v4 là vector rỗng : size = capacity = 0

string s4 {}; // s4 = ""

Lý do duy nhất cho việc không khởi tạo giá trị khi khai báo biến là bộ đệm.

Chúng ta có thể dễ dàng khởi tạo giá trị cho buf:

char buf[max] {}; // Set Mọi kí tự = 0

Nhưng sẽ tốn chi phí vì bộ đệm chỉ dùng để đọc dữ liệu.

Nếu không có khởi tạo { }, biến toàn cục, namespace, local static hay thành viên static khởi tạo giá trị mặc định { }. Ví dụ:

int a; // int a{} tức a = 0

double d; // double d{}tức d = 0.0

Đối tượng và biến local trên free store (hay cấp phát động) không có giá trị mặc định, trừ khi chúng có hàm dựng mặc định khởi tạo giá trị.

##### std::initializer\_list

1. Danh sách khởi tạo { }

Các đối tượng có nhiều thành viên dữ liệu, sẽ cần danh sách khởi tạo gồm nhiều giá trị nằm trong cặp { }

int a[] = { 1, 2 }; // Khởi tạo mảng int giống như trên C

struct S { int x, string s };

S s = { 1, "Helios" }; // Khởi tạo struct

complex<double> z = { 0, pi }; // dùng hàm dựng

vector<double> v = { 0.0, 1.1, 2.2, 3.3 }; // dùng hàm dựng nhận tham số kiểu danh sách

Các dòng khai báo trên đều thừa “=”, chỉ để nhấn mạnh đây là trường hợp khởi tạo mà thôi.

Hàm dựng khởi tạo dùng () mang nghĩa giống như lệnh gọi ‘‘hàm’’. Để tường minh rằng bạn đang dùng “khởi tạo mặc định”, nên dùng { }.

complex<double> z1(1,2); // danh sách khởi tạo kiểu hàm

complex<double> f1(); // giống khai báo hàm

complex<double> z2 {1,2}; // danh sách khởi tạo C++11 {1,2}

complex<double> f2 {}; // danh sách khởi tạo mặc định C++11 {0,0}

Lưu ý rằng danh sách khởi tạo { } không ảnh hưởng bởi luật ép kiểu.

Khi dùng với **auto**, danh sách { } hiểu là **std::initializer\_list<T>** :

auto x1 {1,2,3,4}; // x1 là initializer\_list<int>

auto x3 {1.0,2}; // error : không thể suy ra kiểu của {1.0,2} (§6.3.6.2)

1. Tham số std::initializer\_list

Hàm dựng danh sách khởi tạo {} nhận tham số kiểu **std::initializer\_list**

Một số container vector, map… có hàm dựng, phép gán danh sách { }:

vector<double> v = { 1, 2, 3.456, 99.99 };

list<pair<string,string>> languages = {

{"Nygaard","Simula"},

{"Richards","BCPL"},

{"Ritchie","C"}

};

map<vector<string>,vector<int>> years = {

{ {"Maurice","Vincent", "Wilkes"},{1913, 1945, 1951, 1967, 2000} },

{ {"Martin", "Richards"} {1982, 2003, 2007} },

{ {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }

};

Để triển khai cho tham số hàm, chỉ cần khai kiểu **std::initializer\_list<T>**

void f(initializer\_list<int>);

f({1,2});

f({23,345,4567,56789});

f({}); // danh sách trống

years.insert({

{"Bjarne","Stroustrup"},

{1950, 1975, 1985}

});

Danh sách có độ dài tùy biến nhưng phải có kiểu đồng nhất, tức mọi phần tử phải cùng kiểu template T hay chuyển kiểu ngầm sang T.

1. Độ ưu tiên initializer\_list

Luật:

• Hàm dựng mặc định ưu tiên hơn so với danh sách khởi tạo

• Hàm dựng danh sách khởi tạo ưu tiên hơn so với đối số thông thường

Xét:

struct X {

X(initializer\_list<int>);

X();

X(int);

};

X x0 {}; // Trống: hàm dựng mặc định hay initializer-list? (hàm dựng mặc định)

X x1 {1}; // hàm dựng đối số int hay initializer-list 1 phân tử? (hàm dựng initializer-list)

1. Sử dụng initializer\_lists

Bạn phải khai báo header: <initializer\_list>.

* Giống như container, initializer\_list<T> được triển khai iterator với begin, end và size().Tuy nhiên, initializer\_list không hỗ trợ truy xuất ngẫu nhiên [ ], nên phải dùng vòng lặp:

void f(initializer\_list<int> args)

{

for (int i = 0; i!=args.size(); ++i)

cout << args.begin()[i] << "\n";

}

//Tương tự với:

void f(initializer\_list<int> args)

{

for (auto p=args.begin(); p!=args.end(); ++p)

cout << ∗p << "\n";

}

//Dùng auto luận kiểu:

void f(initializer\_list<int> args)

{

for (auto x : args)

cout << x << "\n";

}

* Các phần tử initializer\_list không cho phép đổi giá trị

int f(std::initializer\_list<int> x, int val)

{

∗x.begin() = val; // error : cố gắng thay đổi giá trị initializer-list element

return ∗x.begin(); // OK

}

void g()

{

for (int i=0; i!=10; ++i)

cout << f({1,2,3},i) << '\n';

}

1. Triển khai initializer\_lists cho hàm dựng

Vì không thay đổi giá trị nên không thể áp dụng Move constructor cho initializer\_list mà phải sao chép từng phần tử.

Xem triển khai hàm dựng initializer-list cho vector:

template<class E>

class Vector {

public:

Vector(std::initializer\_list<E> s); // hàm dựng initializer-list

// ...

private:

int sz;

E\* elem;

};

template<class E>

Vector::Vector(std::initializer\_list<E> s):sz{s.size()} // set vector size

{

reserve(sz); // mở rộng capacity

uninitialized\_copy(s.begin(), s.end(), elem); // khởi tạo phần tử cho mảng elem

}

Phương thức chuẩn template std:: uninitialized\_copy() sao chép nội dung từ iterator bắt đầu tới iterator kết thúc vào con trỏ elem.

1. Khởi tạo trực tiếp và sao chép

Bạn thử xem trường hợp khởi tạo giá trị phần tử cho vector:

vector<vector<double>> vs = {

{10,11,12,13,14}, // OK: vector 5 phần tử ép kiểu ngầm

{10}, // OK: vector 1 phần tử

10, // error : vector<double>(int) không tương thích kiểu

vector<double>{10,11,12,13}, // OK: vector 5 phần tử

vector<double>{10}, // OK: vector 1 phần tử 10.0

vector<double>(10), // OK: vector 10 phần tử giá trị mặc định 0.0

};

Trong ví dụ trên dùng hàm dựng initializer-list, tham số khởi tạo có thể là initializer-list hay vector<double>.

**std::vector<int>**(int) là rất tường minh:

vector<double> v1(7); // OK: v1 có 7 elements

vector<double> v2 = 9; // error : lỗi ép kiểu từ int -> vector

void f(const vector<double>&);

void g()

{

v1 = 9; // error : lỗi ép kiểu từ int -> vector

f(9); // error : lỗi ép kiểu từ int -> vector

}

Nhưng **std::vector<int>(initialize\_list<int>**) thì không . Thay thế () với {}, nhận kết quả hoàn toàn khác:

vector<double> v1 {7}; // OK: v1 có 1 element (= 7)

vector<double> v2 = {9}; // OK: v2 has one element (= 9)

void f(const vector<double>&);

void g()

{

v1 = {9}; // OK: v1 có 1 element (= 9)

f({9}); // OK: f nhận tham số initializer\_list {9}

}

Như vậy bạn có thể dùng initialize\_list khi khởi tạo với hàm dựng hay truyền tham số vào hàm:

vector<double> v1 {7,8,9}; // OK: v1 có 3 phần tử {7,8,9}

vector<double> v2 = {9,8,7}; // OK: v2 có 3 phần tử {9,8,7}

void f(const vector<double>&);

void g()

{

v1 = {9,10,11}; // OK: v1 có 3 phần tử {9,10,11}

f({9,8,7,6,5,4}); // OK: f gọi với tham số list {9,8,7,6,5,4}

}

Tương tự với kiểu chuỗi:

vector<string> v1 { "Anya"}; // OK: v1 has one element (with the value "Anya")

vector<string> v2 = {"Courtney"}; // OK: v2 has one element (with the value "Courtney")

void f(const vector<string>&);

void g()

{

v1 = {"Gavin"}; // OK: v1 now has one element (with the value "Gavin")

f({"Norah"}); // OK: f is called with the list {"Norah"}

}

### C++11 - Lambda

1. Giới thiệu

Có nhiều ngôn ngữ triển khai hàm Nặc Danh (Anonymous Function) như Python, Perl, C#, JavaScript, ... và ở phiên bản C++11 Lambda chính là tên gọi của kỹ thuật này.

Hàm nặc danh là hàm có triển khai thân hàm nhưng khác với các hàm khác là nó không có tên hàm.

Bài viết này cùng với các bài viết về C++11 khác nhằm giới thiệu một số tính năng mới của C++11.

1. Khái niệm Lambda

Lambda hay còn gọi là hàm nặc danh, nó có thể dùng để truyền vào một hàm khác và sử dụng một lần. Khác với các cách thông thường, ta định nghĩa hàm sau đó dùng tên hàm truyền vào một hàm khác.

Lambda đã được giới thiệu ở nhiều ngôn ngữ như Java, C#, JavaScript, và ở phiên bản C++11 đã có tính năng này.

Lợi ích của lambda là không nhất thiết phải khai báo tên hàm ở một nơi khác, mà có thể tạo ngay một hàm (dùng 1 lần). Như vậy, ta sẽ giảm được thời gian phải khảo sát hàm. Để làm rõ hơn về khái niệm này, ta sẽ xét 2 ví dụ sau.

Ví dụ 1 - Khai báo hàm sau đó truyền hàm vào hàm khác

1. #include <iostream>
2. #include <vector>
3. using namespace std;
4. void stdio\_doing(int n){
5. n = n + 1;
6. cout << n;
7. }
8. void for\_each (vector<int> v1, void (\*func)(int a)){
9. for (auto i = v1.begin(); i != v1.end(); i++){
10. func(\*i);
11. }
12. }
14. void main(){
15. vector<int> v;
16. v.push\_back(0);
17. v.push\_back(1);
18. v.push\_back(2);
19. v.push\_back(3);
21. for\_each(v, stdio\_doing);
22. }

Ví dụ 1 - Truyền thẳng biểu thức lambda (hàm) vào hàm khác

1. #include <iostream>
2. #include <vector>
4. using namespace std;
6. void for\_each (vector<int> v1, void (\*func)(int a))
7. {
8. for (auto i = v1.begin(); i != v1.end(); i++)
9. {
10. func(\*i);
11. }
12. }
14. void main()
15. {
16. vector<int> v;
17. v.push\_back(0);
18. v.push\_back(1);
19. v.push\_back(2);
20. v.push\_back(3);
22. for\_each(v, [](int a){
23. a = a + 1;
24. cout << a;
25. });
26. }

Xem xét ví dụ 2: ta đã không cần hiện thực trước hàm stdio\_doing mà đến khi thực sự sử dụng for\_each, ta mới cần hiện thực "stdio\_doing" này.

*Điều thú vị nữa ở trên là tôi đã tặng cho bạn một mẹo nhỏ hiện thực "vòng lặp" for\_each.*

1. Cú pháp
2. [capture](parameters){
3. block\_statements
4. }

Cú pháp của Lambda cực kỳ đơn giản, ta chỉ cần bắt đầu với

[](){ // Code của bạn ở đây }

Ta thử tạo 1 biểu thức lambda đơn giản nhất có thể như codes bên dưới.

1. #include <iostream>
3. using namespace std;
5. void main()
6. {
7. void (\*f)() = []() {
8. cout << "lambda ";
9. cout << "expression";
10. };
12. f();
13. }

Sau khi tạo xong biểu thức, "sẵn tiện" tôi gán địa chỉ nó vào một con trỏ hàm khác (vì nếu không có đối tượng nào kiểm soát địa chỉ của hàm này, thì không có cách nào sử dụng được). sau đó tôi gọi "hàm" này vận hành. Nâng cấp hơn cho biểu thức trên, ta có thể thử trả về 1 giá trị, dĩ nhiên là con trỏ hàm f cũng phải có kiểu trả về là kiểu tương ứng với kiểu trả về trong biểu thức lambda (các bạn có thể tự trải nghiệm điều này). Tương tự, bạn có thể thử nghiệm truyền các parameters vào biểu thức này. Riêng giữa 2 dấu ngoặc vuông [capture], tôi sẽ giải thích thêm về capture.

Xét ví dụ bên dưới để hiểu rõ hơn về capture, ta thêm int stdio = 5 và muốn sử dụng nó trong biểu thức lambda. Điều đơn giản nhất bạn có thể nghĩ ra vài cách để sử dụng stdio trong biểu thức là xem nó như một tham số và truyền vào biểu thức tại cặp ngoặc (). Tuy nhiên, ta không muốn sử dụng như vậy, ta sẽ dùng cách khác, một sự thật hiển nhiên là đôi lúc khung sườn (kiểu trả về và tham số đầu vào) của hàm do một lập trình viên khác viết, hoặc cụ thể hơn ta khảo sát cách thức hoạt động của for\_each phía trên sẽ thấy được sự bất tiện này nên ta không thể sử được phương pháp này. Vấn đề là ta sẽ sử dụng[capture] để cho phép biểu thức lambda sử dụng các biến bên ngoài và sử dụng nó như thế nào cũng do capture quy định.

1. #include <iostream>
3. using namespace std;
5. void main()
6. {
7. int stdio = 5;
9. int (\*f)() = []() {
10. cout << "lambda ";
11. cout << "expression";
12. cout << stdio;
13. return 1;
14. };
16. cout << f();
17. }

Codes trên sẽ được trình biên dịch báo lỗi tại dòng cout << stdio vì [] không có gì, tức là không cho phép bất cứ giá trị nào được "xen" vào biểu thức từ phía ngoài. Vậy ta có các cách thức nào để "xen" vào? Ta có thể sửa biểu thức trên như sau (tôi thêm dấu bằng vào cặp ngoặc vuông, ám chỉ việc cho phép các lambda sử dụng bản sao của các giá trị bên ngoài).

1. #include <iostream>
2. using namespace std;
3. void main(){
4. int stdio = 5;
6. int (\*f)() = [=]() {
7. cout << "lambda ";
8. cout << "expression";
9. cout << stdio;
10. return 1;
11. };
13. cout << f();
14. }
15. Các kiểu capture

|  |  |
| --- | --- |
| [] | Không được phép sử dụng bất kỳ biến bên ngoài |
| [=] | Được phép sử dụng các biến bên ngoài, nhưng là dạng sao chép giá trị của biến đó (by value) |
| [&] | Được phép sử dụng chính biến đó (by reference) |
| [=,&a] | Đây là một dạng nâng cao, bạn có thể cho phép tất cả các biến khác được sử dụng giá trị, và chỉ định thêm cho biến a được sử dụng chính nó (by reference) |
| [this] | Cho phép sử dụng this (OOP) như một bản sao, bạn có thể thử các dạng capture trên với OOP để khám phá thêm. |

1. Phần bổ sung

bổ sung thêm cách tạo ra lambda express và gọi thực thi tại nơi tạo ngay.

Ta có if (biểu thức), trong ý định này tức là ta khai báo biểu thức trong if và mong đợi nó sẽ thực thi ngay thay vì phải chờ gọi, ta cần làm như sau. [] {// Codes của bạn ở đây } ( ) Bạn cần lưu ý vị trí của dấu ngoặc tròn () khi tạo ra một biểu thức.

*[](){} sẽ tạo ra 1 hàm, nhưng nó chưa được gọi (ta phải gọi nó sau).  
[]{}() sẽ tạo ra 1 hàm, và gọi thực thi hàm này ngay.*

Bạn có thể đọc thêm về C++14 - Generic Lambda :: [www.stdio.vn/articles/read/349/c14-generic-lambda](http://www.stdio.vn/articles/read/349/c14-generic-lambda)để hiểu thêm về sự tiến hóa của Lambda.

### C++11 - Từ khóa auto

1. Giới thiệu

Từ phiên bản C++03 (hay C++03 trở về trước), C++ đã đủ tốt để chúng ta phát triển ứng dụng nhanh, theo xu thế, nó tiến hóa hơn để việc phát triển ứng dụng nhanh hơn nữa. Trong bài viết này, tôi đề cập từ khóaauto trong C++11 vì nó dễ gây nhầm lẫn với từ khóa auto trong phiên bản C++03.

1. Từ khóa auto ở C++03

Với từ khóa auto, một biến sẽ chỉ định cho trình biên dịch biết rằng, **nó có thời gian "sống" trong một phạm vi cục bộ**. Các biến auto sẽ không được khởi tạo tự động, chính vì vậy, dòng code sau sẽ sinh ra lỗi ở dòng thứ 2

1. auto int STDIO\_uninitialized\_local\_variable;
2. int STDIO\_implicit\_auto = STDIO\_uninitialized\_local\_variable; // Error

Dòng code thứ 2 sẽ lỗi vì biến ta đang sử dụng một biến chưa được khởi tạo STDIO\_uninitialized\_local\_variable. Theo logic thì bạn không thể gán một biến chưa có giá trị gì cho một biến khác. Bạn có thể sửa lại thành code như sau, trình biên dịch sẽ chấp nhận

1. auto int STDIO\_uninitialized\_local\_variable = 0;
2. int STDIO\_implicit\_auto = STDIO\_uninitialized\_local\_variable;
3. Tinh giản từ khóa auto

Chính vì từ khóa auto chỉ sử dụng cho biến cục bộ (local), nên trình biên dịch cho phép tinh giản từ khóa này và ta chỉ cần khai báo như bên dưới, cách khai báo 2 biến STDIO\_a và STDIO\_b là tương đương

1. auto char STDIO\_a = 'x';
2. char STDIO\_b = 'y';

Về sau C++11, đã bỏ đặc tính này, thay vào ý nghĩa mới cho auto

1. Từ khóa auto ở C++11

Với từ khóa auto ở phiên bản C++11 nó có ý nghĩa là, **tự động nhận dạng kiểu dữ liệu** thông qua kiểu dữ liệu của giá trị khởi tạo ra nó.

Trình biên dịch có thể nhận diện auto ở C++11 vì đằng sau nó không khai báo kiểu dữ liệu. Ví dụ

1. auto STDIO\_c = 'c'; // STDIO\_c kiểu char do 'c' kiểu char
2. auto STDIO\_i = 10; // STDIO\_i kiểu int do 10 kiểu int
3. auto STDIO\_p = new int[3]; // STDIO\_p kiểu int\* do new int[3] trả về kiểu int\*
5. Std::list<int> li; // STDIO\_li\_i kiểu list<int>::iterator do li.begin() trả về kiểu list<int>::iterator
6. auto STDIO\_li\_i = li.begin();

*****Lưu ý rằng, với auto thì biến phải được khởi tạo ngay để trình biên dịch nhận dạng, và khi đã nhận dạng kiểu dữ liệu thì không được thay đổi kiểu dữ liệu nữa.*

1. Sử dụng tham chiếu với auto

Auto được sử dụng rất nhiều với container trong thư viện STL, thứ nhất là đơn giản khi khai báo. Nguyên tắc là chúng ta phải dùng biến tham chiếu

auto & whichList = theLists[ myhash( x, theLists.size( ) ) ];

Cách viết trên rút gọn thủ tục khai báo dài dòng của iterator.

Trình biên dịch tự luận ra kiểu whichList là iterator của container theLists.

Ý nghĩa thứ hai là tránh việc sao chép giá trị từ phần tử container, nếu bạn không dùng tham chiếu như cú pháp bên dưới:

auto whichList = theLists[ myhash( x, theLists.size( ) ) ];

Trình biên dịch lại hiểu whichList là bản sao giá trị phần tử container theLists.

Tương tự với vòng lặp for each:

for( auto x : arr ) // sai

++x;

Cái chúng ta cần là x phải là kiểu tham chiếu cho mỗi phần tử trong container. Đoạn code nên là :

for( auto &x : arr ) // ok

++x;

1. Lợi ích và điều bất lợi

Lợi ích của auto trong C++11 là khá lớn trong việc tăng tốc phát triển ứng dụng. Đôi lúc các kiểu dữ liệu rất to lớn và dài, việc sử dụng "tự động chọn kiểu" như auto sẽ giúp cho code ngắn gọn vì "dù gì cũng phải gõ lại đúng kiểu", bớt các công việc tay chân, như ví dụ trên ta thấy list<int>::iterator là kiểu dữ liệu gõ khá dài.

Ví dụ:

Concurrency::task<Platform::Array<byte>^> //trích trong một project mẫu của Direct3D Windows Phone 8 với XAML

Nhược điểm của loại biến này là sự không tường minh trong mã nguồn, gây ra việc khó tra cứu nguồn gốc (kiểu dữ liệu) của một biến.

1. Lời kết

Công nghệ thật tuyệt vời, khi nó làm cho việc phát triển ứng dụng nhanh hơn và làm cho độ tin cậy của sản phẩm cao và nó chỉ phù hợp với các nhà phát triển và các lập trình viên có kinh nghiệm.

##### C++11 - Từ Khóa decltype

1. Giới thiệu

Mỗi một biến hay một giá trị đều chứa thông tin về kiểu dữ liệu của nó và decltype trong C++11 giúp ta "lấy kiểu" và sử dụng kiểu của một biến hay giá trị đó.

Sau khi thực hiện xong bài viết về [C++11 - Từ Khóa auto](http://www.stdio.vn/articles/read/54-c11-tu-khoa-auto) nhận thấy decltype là một từ khóa có mối quan hệ với auto, nên tôi thực hiện thêm bài viết này.

1. decltype là gì?

Trong C++11, từ khóa decltype dùng để yêu cầu lấy kiểu dữ liệu của một biến hoặc một biểu thức/giá trị. Xét các ví dụ sau

1. // STDIO\_i có kiểu là int vì 14 có kiểu là int
2. decltype(14) STDIO\_i;
4. // STDIO\_d có kiểu là double vì 14+3.0 có kết quả là 17.0 có kiểu là double
5. decltype(14 + 3.0) STDIO\_d;
7. // STDIO\_li\_i có kiểu là list<int>::iterator do li.begin() có kiểu là list<int>::iterator
8. list<int> li;
9. decltype(li.begin()) STDIO\_li\_i;
10. decltype và auto

Nếu bạn chưa biết về từ khóa auto, vui lòng đọc bài viết về [C++11 - Từ Khóa auto](http://www.stdio.vn/articles/read/54-c11-tu-khoa-auto) trước.

Từ khóa auto hỗ trợ việc không cần xác định kiểu, vì kiểu sẽ được xác định thông qua giá trị hay biến được gán. Còn decltype có thể xem là một phương pháp thay thế cho auto khá hiệu quả.

1. auto m = 100;
2. decltype(100) n = 100;

Cả 2 trường hợp trên, biến m và n đều sẽ có kiểu là int.

1. Hàm trả có kết quả trả về là auto

Ta phải chỉ định kiểu trả về của hàm dạng sau

1. auto getInscreaseAge(int age) -> decltype(age)
2. {
3. return age++;
4. }

Như cách viết hàm trên, auto sẽ không thể xác định được kiểu dữ liệu trả về, với cú pháp mới trong C++11 với giá trị trả về kiểu auto, bạn phải "chỉ định" kiểu trả về là gì bằng cách thêm -> return\_type như trường hợp trên, ta có kiểu trả về sẽ là int (do age kiểu int).

Cách viết hàm trên chưa thể hiện được lợi ích của việc dùng auto và decltype trong trường hợp này vì thật ra ta có thể viết đơn giản như sau

1. int getInscreaseAge(int age)
2. {
3. return age++;
4. }
6. // or
8. auto getInscreaseAge(int age) -> int
9. {
10. return age++;
11. }

Ta sẽ sử dụng nó kèm với template.

1. template <typename T1, typename T2>
2. auto getAdd(T1 arg1, T2 arg2) -> decltype(arg1 + arg2)
3. {
4. return (arg1 + arg2);
5. }

Lựa chọn sử dụng decltype cho kiểu sẽ được trả về, vậy kết quả cuối cùng sẽ quy định kiểu được trả về là kiểu gì, tất cả sẽ do kiểu truyền vào quyết định kiểu trả về chứ không phải bắt buộc ta phải biết trước kiểu trả về là gì (được quyền lựa chọn linh động hơn kiểu trả về).

### C++11 Từ khóa decltype

1. decltype là gì?

Trong C++11, từ khóa decltype dùng để yêu cầu lấy kiểu dữ liệu của một biến hoặc một biểu thức/giá trị. Xét các ví dụ sau

1. // STDIO\_i có kiểu là int vì 14 có kiểu là int
2. decltype(14) STDIO\_i;
4. // STDIO\_d có kiểu là double vì 14+3.0 có kết quả là 17.0 có kiểu là double
5. decltype(14 + 3.0) STDIO\_d;
7. // STDIO\_li\_i có kiểu là list<int>::iterator do li.begin() có kiểu là list<int>::iterator
8. list<int> li;
9. decltype(li.begin()) STDIO\_li\_i;
10. decltype và auto

Nếu bạn chưa biết về từ khóa auto, vui lòng đọc bài viết về [C++11 - Từ Khóa auto](http://www.stdio.vn/articles/read/54/c11-tu-khoa-auto) trước.

Từ khóa auto hỗ trợ việc không cần xác định kiểu, vì kiểu sẽ được xác định thông qua giá trị hay biến được gán. Còn decltype có thể xem là một phương pháp thay thế cho auto khá hiệu quả.

1. auto m = 100;
2. decltype(100) n = 100;

Cả 2 trường hợp trên, biến m và n đều sẽ có kiểu là int.

1. Hàm trả trả về kiểu auto

Xem khai báo hàm trả về kiểu auto:

1. int getInscreaseAge(int age){
2. return age++;
3. }
4. // hay dùng auto fucn\_name(param) -> return\_type
5. auto getInscreaseAge(int age) -> int
6. {
7. return age++;
8. }

C++11 qui định bạn phải "chỉ định" kiểu trả về cho hàm auto bằng cú pháp -> return\_type như trên, kiểu trả về sẽ là int (do age kiểu int).

Mục đích chính của việc dùng auto và decltype là sử dụng cho template:

1. template <typename T1, typename T2>
2. auto getAdd(T1 arg1, T2 arg2) -> decltype(arg1 + arg2)
3. {
4. return (arg1 + arg2);
5. }

Lựa chọn sử dụng decltype cho kiểu sẽ được trả về, vậy kết quả cuối cùng sẽ quy định kiểu được trả về là kiểu gì, tất cả sẽ do kiểu truyền vào quyết định kiểu trả về chứ không phải bắt buộc ta phải biết trước kiểu trả về là gì (được quyền lựa chọn linh động hơn kiểu trả về).

### C++11 – Tham chiếu Rvalue và Move semantic

1. Đối tượng tạm thời trả về

Trước C++11, có tồn tại một bệnh khó chữa khiến chương trình C++ chậm đi: việc tạo ra các đối tượng tạm thời.

Thỉnh thoảng các đối tượng này được tối ưu bởi trình biên dịch (ví dụ như trả về giá trị tối ưu). Nhưng không phải tất cả đều như vậy, có khi kết quả lại là sự sao chép lãng phí.

Đối tượng dự phòng là một khái niệm rất thông dụng và nó xuất hiện trong phần lớn các câu lệnh mà chúng ta viết ra. Ví dụ:

int a = 4 + (3 \* b);

Ở dòng code trên, chương trình cần sử dụng đến 2 đối tượng tạm thời,

* Đầu tiên thực hiện tính 3 \* b rồi gán vào đối tượng tạm thời thứ 1
* Sau đó lấy giá trị 4 cộng với đối tượng tạm thời 1 rồi lấy kết quả gán vào đối tượng tạm thời thứ 2

Để rõ hơn ta có thể viết lại cách mà chương trình thực hiện như sau:

int temp1 = 3 \* b;

int temp2 = 4 + temp1;

int a = temp2;

Hai đoạn code trên sẽ không khác nhau mấy về chi phí thực thi. Chỉ khác ở chỗ đoạn code đầu chương trình sử dụng temporary object, còn đoạn code thứ hai tự ta sử dụng named object để chứa giá trị tạm thời.

Ta thường sẽ gặp các loại đối tượng tạm thời sau:

* Giá trị trung gian trong quá trình tính toán của biểu thức. Ví dụ: int a = 4 + (3 \* b);
* Biến trả về từ hàm.
* Biến được khởi tạo nhưng không đặt tên. Ví dụ: if (string("stdio.vn") == string("www.stdio.vn"));

1. Lvalue và Rvalue

Lvalue là một biểu thức , biến có thể lấy được địa chỉ, bạn có thể thực hiện phép gán lên chúng. Ví dụ:

int a = 1; //a là lvalue

Lvalue còn là tham chiếu trả về từ hàm:

int x;

int& getRef (){ return x;}

getRef() = 4;

getRef() trả về địa chỉ của biến x, sau đó gán 4 cho x.

Rvalue đại diện bởi các đối tượng tạm thời. Ví dụ:

int x;

int getVal (){ return x;}

getVal();

getVal() là rvalue- nó trả về giá trị của x, chỉ là giá trị tạm thời.

Ví dụ về đối tượng rvalue:

string getName (){return "Alex";}

getName() trả về chuỗi khởi tạo bên trong hàm. Bạn dùng nó để gán cho biến khác:

string name = getName();

Nhưng không thể gán vào rvalue một giá trị hay một lvalue.

1. Tham chiếu Rvalue

Có loại tham chiếu nào dùng để tham chiếu đến rvalue không? Nếu từ trước C++11 thì loại tham chiếu này không tồn tại, nhưng từ C++11 đã có thêm một khái niệm mới "rvalue reference" – tham chiếu tới rvalue. Kiểu dữ liệu rvalue reference có thể được khai báo bằng cách thêm dấu "&&". Ví dụ string&& là một "rvalue reference to string".

*Kí hiệu "&&" có thể dễ gây nhầm lẫn đây là một "reference to reference", nhưng đây là kí hiệu của rvalue reference, và trong C++ cũng không có khái niệm "reference to reference".*

Dù tên gọi của "rvalue reference" có từ "tham chiếu", nhưng nó thường không được dùng vào mục đích để tham chiếu đến một rvalue. Mà mục đích chính của nó là để giúp ta nhận ra đâu là lvalue, và đâu là rvalue. Vì chỉ có rvalue reference mới có thể tham chiếu đến rvalue, nên nếu ta có 2 hàm có cùng tên và giá trị trả về, một hàm nhận đối số là string& và một hàm nhận đối số là string&& thì:

* Nếu giá trị truyền vào là một lvalue thì hàm có đối số là string& sẽ được gọi.
* Nếu giá trị truyền vào là một rvalue thì hàm có đối số là string&& sẽ được gọi.

Nói một cách ngắn gọn, rvalue là một loại tham chiếu mới mà nó chỉ có thể được dùng để tham chiếu tới rvalue. Và nhờ đó mà ta có thể định nghĩa được "move semantics" để giảm thiểu tối đa những chi phí copy dữ liệu vô ích (copy dữ liệu từ đối tượng tạm).

1. Copy semantics

Copy semantics nghĩa là ta thực hiện copy trạng thái (hay dữ liệu) của đối tượng này sang đối tượng khác, sau đó thì cả 2 đối tượng này sẽ có trạng thái (hay dữ liệu) như nhau. Đối với lập trình viên C++ thì có lẽ khái niệm này khá quen thuộc, khi chúng ta định nghĩa "copy constructor" hay “copy assignment operator” (toán tử gán) thì ta đã định nghĩa nó theo copy semantics – nghĩa là ta đã định nghĩa cho nó thực hiện việc copy trạng thái (dữ liệu) của đối tượng nguồn sang đối tượng đích.

Nhưng có một điểm của copy semantics mà ta luôn phải để ý, đó là ta chỉ thực hiện copy trạng thái (dữ liệu) của đối tượng nguồn sang đối tượng đích, còn trạng thái (dữ liệu) của đối tượng nguồn vẫn y nguyên như cũ. Nhưng đôi khi đó không phải là điều mà chúng ta muốn, mà điều chúng ta muốn là "dịch chuyển" trạng thái (dữ liệu) của đối tượng nguồn sang đối tượng đích, sau đó thì trạng thái (dữ liệu) của đối tượng nguồn sẽ ở trạng thái default (vì dịch chuyển là ta thực hiện "lấy" chứ không phải copy, cũng giống như chúng ta thực hiện "cut file", sau khi copy xong file đó sang vị trí khác thì file gốc sẽ bị xóa).

1. Move semantics

Tìm hiểu tại sao đoạn code sau là điều khủng khiếp trong C++03. (Nhưng lại tốt trên C++11)

#include <vector>

#include <iostream>

using namespace std;

//hàm gấp đôi từng giá trị phần tử vector

vector<int> doubleValues (const vector<int>& v){

vector<int> new\_values;

new\_values.reserve(v.size());

for (auto itr = v.begin(), end\_itr = v.end(); itr != end\_itr; ++itr ) {

new\_values.push\_back( 2 \* \*itr );

}

return new\_values;

}

int main(){

vector<int> v;

for ( int i = 0; i < 100; i++ ) v.push\_back( i );

// gấp đôi từng phần tử của vector v

v = doubleValues( v );

}

Để tìm hiểu xem vấn đề gì sẽ xảy ra thì ta sẽ "thực thi bằng tay":

* Bước 1 - Chương trình tạo ra một mảng động vector<int> "v"
* Bước 2 - Nó sẽ nhảy vào thực hiện đoạn code trong hàm doubleValues().
* Bước 3 - Nó sẽ tạo ra một đối tượng tạm bằng cách copy dữ liệu từ mảng new\_value.
* Bước 4 - Đối tượng tạm sẽ được gán cho đối tượng vector v bằng cách copy dữ liệu từ mảng tạm

Chưa xét đến chi phí thực hiện, ta nhận thấy rằng, ngay trước dòng "return " , ta đã có được mảng kết quả, nhưng sau đó, chỉ vì để trả về và gán kết quả vào biến v mà ta đã phải thực hiện copy mảng đó 2 lần. Ta dễ dàng thấy được việc này quá vô ích, tại sao ta không gán con trỏ mảng của đối tượng new\_values (con trỏ bên trong nó, để trỏ tới mảng mà nó quản lý) cho con trỏ mảng của đối tượng v? Nếu làm được như vậy thì ta sẽ tránh được hoàn toàn việc copy dữ liệu tới 2 lần.

*Vector là một loại kiểu dữ liệu mảng tự co giãn (Mảng tự động tăng kích thước khi ta chèn nhiều phần tử vào mảng hơn kích thước hiện tại của nó). Bên trong nó có một con trỏ dùng để trỏ tới mảng mà nó quản lý, khi mảng đã đầy mà chương trình tiếp tục thêm phần tử vào thì nó sẽ tự giãn ra bằng cách cấp phát lại 1 mảng mới kích thước gấp 1.5 lần và copy các phần tử của mảng cũ sang mảng mới.*

Giải pháp tối ưu trong trường hợp này là gán trực tiếp con trỏ mảng bên dưới của đối tượng new\_values sang đối tượng tạm, rồi đối tượng tạm sang đối tượng sortedArray mà không cần phải thực hiện việc tạo mảng mới và copy dữ liệu mảng. Và đó là khi ta cần tới khái niệm "Move semantics".

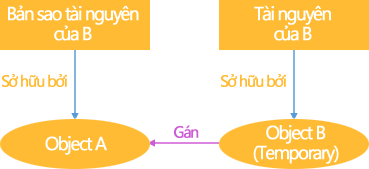
*Đối với những chương trình nhỏ thì có thể ta không để ý. Nhưng đối với những chương trình lớn, việc tạo ra các đối tượng tạm thời tốn rất nhiều chi phí. Do đó, từ C++11 đã được đưa vào thêm 2 khái niệm mới "rvalue reference" và "move semantic" để giúp giải quyết, tối ưu và giảm thiểu chi phí cho việc copy trạng thái (dữ liệu) của đối tượng tạm thời.*

1. Phương pháp Move semantics

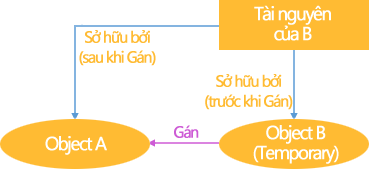
Lúc này, ta muốn gán đối tượng tạm B cho một đối tượng A. Ta cũng sẽ có 2 lựa chọn:

* Một là cấp phát 1 lượng tài nguyên tương tự cho A rồi thực hiện gán
* Hai là tận dụng tài nguyên của B mà khỏi cần tạo mới, vì dù gì thì tài nguyên của B sẽ bị hủy đi ngay sau đó (cùng với B, vì B là đối tượng tạm).

Phương án tối ưu nhất trong tình huống này luôn là phương án hai, nghĩa là khi ta thực hiện copy dữ liệu từ đối tượng tạm thì thay vì cấp phát một lượng tài nguyên tương đương, thì ta trực tiếp chuyển quyền sở hữu tài nguyên đó cho đối tượng đích. Và khi đó thì ta đã thực hiện việc copy dựa theo "move semantics".

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/28/content/ss_1.png)

Hình 1 - Phương án tạo mới tài nguyên cho A khi copy từ đối tượng tạm (Copy semantics)

*[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/28/content/ss_2.png)*

Hình 2 - Phương án chuyển quyền sở hữu trực tiếp tài nguyên của B cho A (Move semantics)

Do vậy ta có thể dễ dàng định nghĩa "move semantics" như sau:

*Move semantics nghĩa là ta thực hiện hành động copy bằng cách move tài nguyên (chuyển quyền sở hữu tài nguyên) từ đối tượng nguồn (đối tượng cần copy) sang đối tượng đích (đối tượng copy) thay vì phải tạo mới và copy lại tài nguyên đó.*

C++03 chưa có khái niệm ‘move semantics’. Nên C++03 nhận biết được đâu là đối tượng “lâu dài” – lvalue – hay đâu là đối tượng tạm – rvalue. Nhưng may thay C++11 đã đưa ra một khái niệm mới là rvalue reference – như đã đề cập ở trên, giúp chúng ta có thể xác định đối số nào là lvalue (đối tượng “lâu dài”) và đối số nào là rvalue (đối tượng tạm).

‘Move semantics’ chỉ là một giải pháp để giúp tối ưu việc copy trong một số trường hợp, chứ nó không thể thay thế được ‘copy semantics’. Do đó, thông thường những hàm gán sẽ được định nghĩa cả 2 phiên bản, một phiên bản dựa theo ‘copy semantics’ và một phiên bản dựa theo ‘move semantics’.

* Hàm đầu tiên sẽ nhận đối số là ‘lvalue reference đến kiểu của tham số (Type&), và trong hàm này ta sẽ định nghĩa việc copy theo "copy semantics".
* Hàm thứ hai sẽ nhận đối số là ‘rvalue reference đến kiểu của tham số (Type&&), và trong hàm này ta sẽ định nghĩa việc copy theo "move semantics".

1. So sánh Move semantics và Copy semantics

Để dễ dàng nắm được cách định nghĩa "move semantics" thì ta sẽ so sánh nó với "copy semantics"

| **MOVE SEMANTICS** | **COPY SEMANTICS** |
| --- | --- |
| Tham số đầu vào là một rvalue reference | Tham số đầu vào là một lvalue reference |
| Trường hợp thành viên dữ liệu của đối tượng đích đã cấp phát, phải thu hồi tài nguyên, nếu chưa thì bỏ qua bước này.  Thực hiện "shallow copy" các trường của đối tượng nguồn qua đối tượng đích.  Sau đó gán các con trỏ của đối tượng nguồn về NULL. (Vì theo "move semantics" thì ta thực hiện chuyển quyền sở hữu tài nguyên của đối tượng nguồn cho đối tượng đích, do đó đối tượng nguồn sẽ không còn sở hữu tài nguyên này nữa). | Thực hiện "deep copy" các trường của đối tượng nguồn qua đối tượng đích. |

Để dễ hiểu hơn ta có ví dụ triển khai move cho phương thức thành viên Assign:

1. class Array{
2. private:
3. int\* \_pData;
4. int \_n;
6. public:
7. Array(int n) {
8. \_pData = new int[n];
9. \_n = n;
10. }
12. /\* Được gọi khi đối số truyền vào là lvalue=> Hàm này sẽ định nghĩa theo "copy semantics" \*/
13. const Array& Assign(const Array& value) {
14. this->\_pData = new int[value.\_n];
15. this->\_n = value.\_n;
16. for (int i = 0; i < value.\_n; i++)
17. this->\_pData[i] = value.\_pData[i];
18. return \*this;
19. }
21. /\* Được gọi khi đối số truyền vào là rvalue=> Hàm này sẽ định nghĩa theo "move semantics" \*/
22. const Array& Assign(Array&& value) {
23. this->\_pData = value.\_pData;
24. value.\_pData = nullptr;
25. this->\_n = value.\_n;
26. return \*this;
27. }
28. };
29. int main(){
30. Array b(1000000);
31. Array a(1);
32. /\* Vì b là lvalue => gọi hàm Assign(const Array&)để thực hiện copy dựa theo "copy semantics" \*/
33. a.Assign(b);
34. /\* Array(1000000) tạo ra một đối tượng tạm, là một rvalue.Do đó hàm Assign(Array&&)
35. được gọi để thực hiện copy dựa theo Move semantics \*/
36. a.Assign(Array(1000000));
37. }

Lưu ý: Trong ví dụ trên, phương thức nhận tham số Lvalue phải kiểm tra với con trỏ \*this, tránh sao chép chính nó. Phương thức nhận tham số Rvalue, phải kiểm tra thành viên **pData** với **nullptr** xem đã cấp phát chưa để thu hồi vùng nhớ trước khi được gán.

1. std::move

Khi cần thực hiện ‘move semantics’ từ đối tượng Lvalue sang, đó là lúc ta cần sử dụng đến **std::move(&Object),** hàm sẽ trả về một Rvalue reference của tham số đối tượng truyền vào.

Về cơ bản hàm std::move() đơn thuần sử dụng **static\_cast** để trả về tham chiếu Rvalue của đối tượng, chúng ta cũng có thể cast bằng tay**:**

**static\_cast<Type&&>(object)**

Có một số trường hợp đặc biệt mà ta buộc phải sử dụng std::move, như đối với unique\_ptr. Unique\_ptr là một loại con trỏ mà tài nguyên nó sở hữu chỉ được sở hữu bởi duy nhất một đối tượng.

Unique\_ptr vô hiệu hóa triển khai copy constructor và copy assignment, vì việc gán một con trỏ cho một con trỏ khác đồng nghĩa việc hai con trỏ cùng trỏ tới một tài nguyên, vi phạm nguyên tắc của unique\_ptr.

1. #include <memory>
2. using namespace std;
3. int main(){
4. unique\_ptr<int> p1(new int[1000000]);
5. unique\_ptr<int> p2(p1);/\* Bị lỗi, vì unique\_ptr không có copy constructor \*/
6. unique\_ptr<int> p3(std::move(p1));/\* Tài nguyên p1 sở hữu sẽ được chuyển sang cho p3, còn p1 trỏ tới null \*/
7. }
8. std::swap và std::move

Một trong những phương thức dùng move semantic là **std::swap**(). Thông thường swap() copy được thực hiện qua 3 bước dưới đây:

void swap( double & x, double & y ){

double tmp = x;

x = y;

y = tmp;

}

void swap( vector<string> & x, vector<string> & y ){

vector<string> tmp = x;

x = y;

y = tmp;

}//Hoan vi dung dung copy

Tuy nhiên, khi làm việc với cấu trúc dữ liệu lớn, phương pháp trên quá tốn chi phí. Để tiết kiệm, chúng ta dùng move thay vì copy. Nhưng điều kiện là bên phải phép gán là r\_value, và đối tượng có hỗ trợ move.

Ví dụ: nếu vector<string> hỗ trợ move, biến x là r\_value, thì x có thể move sang cho tmp. Tương tự nếu y là r\_value, y có thể move sang cho x.

Vector là kiểu có hỗ trợ move, tuy nhiên, x y và tmp là L\_value nên phải dùng static cast để ép kiểu vế phải phép gán thành R\_value: **vector<string> &&** :

//Hoan vi dung dung type cast

void swap( vector<string> & x, vector<string> & y ){

vector<string> tmp = static\_cast<vector<string> &&>( x );

x = static\_cast<vector<string> &&>( y );

y = static\_cast<vector<string> &&>( tmp );

}

Cú pháp của static cast khá rường rà. Đơn giản hơn, chúng ta dùng std::move, có thể chuyển bất kì L\_value (hay R\_value) nào thành R\_value.

Nếu bạn để trống tham số thì std::move chẳng move gì cả.

std::move được triển khai trong std::swap. std::swap là 1 phần của thư viện chuẩn và làm việc với mọi kiểu dữ liệu:

void swap( vector<string> & x, vector<string> & y ){

vector<string> tmp = std::move( x );

x = std::move( y );

y = std::move( tmp );

}

//Hoan vi dung move semantic

1. Move constructor và Move assignment

Move semantics thường được áp dụng để định nghĩa move constructor và move assignment operator, giúp loại bỏ những chi phí không cần thiết khi tạo một đối tượng mới dựa trên đối tượng tạm, hoặc gán một đối tượng tạm cho một đối tượng nào đó.

Move constructor chỉ khác copy constructor ở chổ đối số đầu vào của nó là rvalue reference (Data\_type&&) thay vì lvalue reference (Data\_type&), và bên trong nó sẽ định nghĩa việc copy dựa theo "move semantics", thay vì "copy semantics" như của copy constructor. Cũng tương tự như giữa move assignment operator và copy assignment operator.

Move constructor và move assignment operator sẽ chỉ được gọi khi đối số truyền vào là một rvalue, do đó, nó chỉ dùng được khi ta thực hiện việc copy từ đối tượng tạm. Vì vậy nên ta cũng cần phải định nghĩa thêm copy constructor và copy assignment operator để dùng trong trường hợp đối số truyền vào là một lvalue.

Để rõ hơn ta có ví dụ sau:

1. class Array{
2. private:
3. int\* \_pData;
4. int \_n;
5. public:
6. Array(int n) {
7. \_pData = new int[n];
8. \_n = n;
9. }
11. /\* Copy constructor: Nó chỉ được gọi khi đối số truyền vào là lvalue\*/
12. Array(const Array& value) {
13. this->\_n = value.\_n;
14. this->\_pData = new int[this->\_n];
15. for (int i = 0; i < this->\_n; i++)
16. this->\_pData[i] = value.\_pData[i];
17. }
19. /\* Move constructor: Nó được gọi KHI đối số truyền vào là rvalue (temporary object) \*/
20. Array(Array&& value){
21. this->\_n = value.\_n;
22. this->\_pData = value.\_pData;
23. value.\_pData = nullptr;
24. }
25. };
27. int main(){
28. Array b(1000000);
29. Array a(b); //b là lvalue,copy constructor được gọi
30. Array a(Array(1000000)); // Array(1000000) là rvalue, move constructor được gọi
31. }
32. Copy Assignment và Move Assignment

Triển khai Move assignment: nhận tham số là R\_value, chúng ta sẽ dùng phương thức hoán vị std::swap để Move giá trị cho từng thành viên dữ liệu đối tượng tạm sang đối tượng hiện tại, dù sao đối tượng tạm cũng bị hủy.

Đối với triển khai Copy assignment: chúng ta không dùng phương pháp Deep Copy truyền thống. Mà sẽ dùng Move constructor để khởi tạo đối tượng tạm, sau đó dùng std::swap cho đối tượng tạm và đối tượng hiện tại.

Cả 2 đều trả về tham chiếu tới con trỏ \*this.

Phương pháp Move dùng std::swap cho đối tượng tạm khá phổ biến và hiệu quả.

Lấy lại ví dụ về lớp Array:

1. class Array{
2. private:
3. int\* \_pData;
4. int \_n;
5. public:
6. Array(int n) {
7. \_pData = new int[n];
8. \_n = n;
9. }
11. /\* Copy assignment: Nó chỉ được gọi khi đối số truyền vào là lvalue\*/
12. Array& operator=(const Array& value) {
13. this->\_n = value.\_n;
14. this->\_pData = new int[this->\_n];
15. for (int i = 0; i < this->\_n; i++)
16. this->\_pData[i] = value.\_pData[i];
17. return \*this;
18. }
20. /\* Move assignment: Nó được gọi KHI đối số truyền vào là rvalue (temporary object) \*/
21. Array& operator=(Array&& value){
22. this->\_pData = std::move(value.\_pData);
23. this->\_ n = std::move(value.\_n);
24. return \*this;
25. }
26. };
27. Truyền tham chiếu Rvalue

Chúng ta sẽ dùng std::move để truyền tham chiếu R\_value vào đối số hàm, với điều kiện phương thức hàm phải triển khai đối số Rvalue.

Xét ví dụ về phương thức push\_front chèn node vào vị trí iterator đầu tiên trong container List

/\* Phương thức push\_ ủy nhiệm cho insert để xử lí node chèn\*/

void push\_front( Object && x ){ insert( begin( ), std::move(x) ); }

iterator insert( iterator itr, Object && x ){ //Triển khai…}

Object là tham số template của lớp. Phương thức push\_front sẽ gọi insert để xử lí, đồng thời truyền tham số R\_value của đối tượng x là std::move(x).

Nếu truyền thẳng biến x mà không dùng std::move(x), trình biên dịch sẽ hiểu bạn đang truyền giá trị của đối tượng tạm.

### C++ 11 – null pointer

Thông thường ta sử dụng NULL để đánh dấu cho một con trỏ đang "không trỏ tới đâu cả", và định nghĩa của NULL chính là giá trị 0. Ở C++11 chúng ta có thêm một lựa chọn mới đó là nullptr.

1. NULL là gì?

**NULL** có thể bị hiểu nhầm là một dạng giá trị đặc biệt để đánh dấu vùng nhớ không trỏ tới đâu cả, nhưng thật sự nó lại rất đơn giản. Và **NULL** xuất phát từ điều như sau

#define NULL 0

Vậy NULL chính là đại diện của giá trị 0. Khai báo int\* p = NULL; cũng chính là int\* p = 0;

1. Vấn đề nảy sinh với NULL

Ta có thể khai báo int\* p = 0; nhưng không thể khai báo int\* p = 10; trừ phi ta ép kiểu int\* p = (int\*)10;

Vì 10 là kiểu số nguyên. Và chính vì NULL hay 0 có tính chất khá đặc biệt (không cần ép kiểu nên có thể xảy ra trường hợp gây bối rối như sau

1. void setSinName(int nameId)
3. void setSinName(char\* name)
5. int main()
6. {
7. setSinName(NULL);
9. return 0;
10. }

Visual Studio 2012 đã chọn gọi hàm có đối số là kiểu int thay vì char\* khiến lập trình viên bối rối và có thể tự gây nhầm lẫn, đây là một thảm họa trong thiết kế.

Vậy, **nullptr** chính là giải pháp hữu hiệu để xác định và phân biệt giữa 1 loại dữ liệu đánh dấu **null** cho pointer và một giá trị **NULL** (thật tế lại là giá trị 0).

1. nullptr dùng riêng cho pointer

C++11 dùng từ khóa nullptr để đại diện cho con trỏ NULL, điều này tường minh hơn cho trình biên dịch.

Ta dùng **nullptr** để truyền vào hàm nếu muốn dùng hàm có liên quan con trỏ, ta sẽ giải quyết được bài toán gây bối rối

void setSinName(int nameId)

void setSinName(char\* name)

int main()

{

setSinName(nullptr);

return 0;

}

1. std::nullptr\_t

nullptr không phải là cách khai báo duy nhất và ép kiểu ngầm cho tất cả kiểu con trỏ (kể cả bool), nullptr là kiểu riêng biệt, có thể dùng decltype để rút ra kiểu của nullptr:

**decltype( nullptr )**

Có một cách tiện hơn là dùng **std::nullptr\_t**

**std::nullptr\_t** là kiểu con trỏ null, bạn có thể dùng như là đối số hàm dựng hay đối số hàm khi cần báo trình biên dịch rằng tham số này là một con trỏ rỗng. Ví dụ:

void func( std::nullptr\_t );

1. Cách sử dụng
2. // p1 nếu so sánh với p2 thì p1 = p2
3. int\* p1 = nullptr;
4. int\* p2 = NULL;
6. // Trình biên dịch báo lỗi vì d không phải kiểu con trỏ
7. int d = nullptr;
9. // Mẹo nhỏ - gán như bên dưới sẽ thành công
10. int e = (int)nullptr;
12. // Phép gán này thành công và b = false
13. bool b = nullptr;

### C++ 11 lý thuyết smart pointer

1. Quản lý tài nguyên
2. Đặt vấn đề

*"Great power comes with great responsibility"*

Chúng ta thường nghe lập trình viên C++, thậm chí những chuyên gia nói rằng quản lý bộ nhớ trong C++ là một vấn đề rất nhức óc vì ta phải tự mình giải phóng những vùng nhớ không còn được dùng nữa. Nhưng nghĩ lại thì điều đó liệu có đúng? Như những gì ta đã được học thì việc ta cần làm chỉ là delete những vùng nhớ mà ta đã cấp phát. Khi xây dựng đối tượng thì ta chỉ cần thực hiện việc delete những vùng nhớ cấp phát bởi đối tượng đó trong destructor của nó. Vậy tại sao mọi người lại nghĩ nó phức tạp?

Đúng là nó sẽ không phức tạp, cho đến một ngày đẹp trời, ngày mà bạn mở máy, bật Visual Studio lên để tiếp tục hoàn thiện dự án của mình, thì bỗng nhiên một vấn đề oái oăm xuất hiện: "Nếu ta có một tài nguyên được sử dụng bởi nhiều đối tượng (ở nhiều nơi) trong chương trình, thì mình sẽ thực hiện hủy tài nguyên đó ở đâu?”. Và đó chính là khi bạn phải đối mặt với vấn đề nhức óc của việc quản lý tài nguyên trong C++.

1. Đi tìm giải pháp

Như ta thấy, nếu mỗi tài nguyên của chúng ta chỉ được sử dụng bởi duy nhất một đối tượng, thì ta chỉ việc thực hiện giải phóng tài nguyên đó trong destructor của đối tượng. Nhưng mọi việc sẽ hoàn toàn khác khi ta có những loại tài nguyên chia sẻ, được sử dụng bởi nhiều đối tượng. Khi đó ta sẽ không biết phải thực hiện giải phóng tài nguyên đó ở đâu – hay nói cách khác, ta sẽ không biết đối tượng nào chịu trách nhiệm giải phóng tài nguyên đó.

Do vậy, để giải quyết được vấn đề này thì ta phải sử dụng đến một khái niệm có mức trừu tượng hóa cao hơn (So với những khái niệm ta thường dùng trong lập trình) – Đó là "Quyền sở hữu".

1. Từ thực tế vào lý thuyết

Tài nguyên là một khái niệm có lẽ đã quá quen thuộc với chúng ta (tài sản, nhà đất, hay bất cứ thứ gì ta sở hữu đều có thể coi là tài nguyên). Khi ta sở hữu một tài nguyên nào đó, thì ta có quyền sở hữu đối với tài nguyên đó, nghĩa là ta có mọi quyền và trách nhiệm đối với tài nguyên. Quyền ở đây có thể bao gồm quyền sử dụng, quyền mua bán, quyền chuyển đổi… Còn trách nhiệm ở đây chính là mọi trách nhiệm pháp lý đối với tài nguyên.

Giả sử, bạn có một chiếc xe Air Blade. Thì bạn sẽ có quyền sử dụng, mua bán và chuyển nhượng. Nhưng việc sở hữu cũng đồng nghĩa bạn sẽ phải chịu mọi trách nhiệm pháp lý đối với chiếc xe đó. Giả sử như nếu chiếc xe đó được sử dụng để gây hành vi phạm pháp thì bạn sẽ là người đầu tiên phải đứng ra chịu trách nhiệm.

Khái niệm quyền sở hữu ra đời giúp chúng ta có thể quản lý tài nguyên tốt hơn. Vì khi đó ta biết chính xác đối tượng nào đang sở hữu nó, ai có thể sử dụng nó, và ai chịu trách nhiệm với nó.

Luật sở hữu nhà đất được nhà nước đặt ra nhằm xác định quyền và nghĩa vụ đối với người sở hữu đất. Cũng tương tự như vậy, luật sở hữu trí tuệ được đặt ra nhằm xác định quyền và nghĩa vụ của những người sáng chế (những người sở hữu ý tưởng).

Nếu chúng ta có thể áp dụng khái niệm về “Quyền sở hữu” đối với tài nguyên trong chương trình thì có thể vấn đề về quản lý tài nguyên sẽ trở nên dễ dàng hơn rất nhiều. Vì lúc này ta sẽ biết được ai sở hữu tài nguyên, ai có quyền sử dụng, và ai phải chịu trách nhiệm thu hồi tài nguyên đó.

1. Quyền sở hữu tài nguyên

Khái niệm quyền sở hữu tài nguyên xoay quanh vấn đề quyền và nghĩa vụ của đối tượng đó đối với một tài nguyên. Nghĩa là, để áp dụng được khái niệm này vào giải quyết vấn đề quản lý tài nguyên trong lập trình, ta cần phải xác định quyền và nghĩa vụ của các đối tượng sở hữu.

Dễ thấy, khi một đối tượng sở hữu tài nguyên, thì nó sẽ có quyền sử dụng và hủy tài nguyên đó, ngoài ra thì nó còn có quyền được chuyển quyền sở hữu của mình cho đối tượng khác (Sẽ được đề cập rõ hơn ở phần sau). Ngoài ra thì nó còn có nghĩa vụ phải hủy đi tài nguyên mà mình sở hữu khi tài nguyên đó không còn được dùng đến nữa.

Qua đó, ta có thể dễ dàng định nghĩa "Quyền sở hữu" như sau:

Khi một đối tượng sở hữu một tài nguyên nào đó, thì nó có quyền truy xuất, sử dụng, trao và chuyển quyền sở hữu. Và trách nhiệm của nó là phải hủy tài nguyên đó khi không còn được sử dụng nữa.

Do vậy, nếu ta áp dụng tốt "Quyền sở hữu" để quản lý các tài nguyên trong chương trình thì ta sẽ không phải gặp phải những vấn đề nhức óc trong quản lý tài nguyên (như là tài nguyên chia sẻ chung), hoặc sẽ không còn phải đau đầu với các vấn đề tương tự như memory leak nữa.

1. Raw Pointer và quyền sở hữu

Giả sử khi ta có một đối tượng, trong nó có một con trỏ thông thường (Hay còn gọi là raw pointer) trỏ đến một tài nguyên, thì đối tượng này đã có thể sử dụng tài nguyên đó. Do vậy, theo một ngữ nghĩa nào đó, ta có thể coi như đối tượng này đã sở hữu tài nguyên (Vì nó được quyền sử dụng). Nhưng nếu ta xét theo đúng khái niệm, thì đối tượng này vẫn chưa thật sự nắm "Quyền sở hữu". Bởi vì "Quyền sở hữu" bao gồm cả quyền và nghĩa vụ. Nó chỉ mới có quyền sử dụng, chứ chưa thực hiện nghĩa vụ hủy đi tài nguyên đó khi nó không còn được sử dụng nữa.

Thông thường, nếu đối tượng trên tự cấp phát tài nguyên, và tài nguyên đó chỉ mình nó sử dụng, thì nó có thể thực hiện thu hồi trong destructor của mình. Nhưng trường hợp chúng ta xét ở đây là trường hợp tổng quát, nghĩa là tài nguyên đó có thể không do đối tượng này tạo ra, và không chỉ được sử dụng bên trong nó. Do vậy, trong "phạm vi" của mình thì nó không có cách nào để thực hiện trách nhiệm hủy đi tài nguyên khi tài nguyên này không còn được sử dụng nữa.

Do vậy, chúng ta cần tới một loại con trỏ mới, thông minh hơn, để có thể giúp các đối tượng thật sự nắm được "Quyền sở hữu" – bằng cách thực hiện nghĩa vụ hủy đi tài nguyên khi nó không còn được sử dụng nữa. Và đó cũng chính là lý do mà "Smart pointer" đã ra đời.

1. Smart Pointer trong C++

Smart pointer là một loại dữ liệu để giả lập các raw pointer, và bổ sung thêm các khả năng mà raw pointer không có như tự động quản lý bộ nhớ, kiểm tra truy xuất ngoài vùng được cấp phát… Để đạt được điều này, người thiết kế C++ đã tạo ra các đối tượng để "đóng gói” các raw pointer (Các đối tượng này còn được gọi là proxy object, được thiết kế dựa theo proxy pattern). Để các đối tượng này hành xử như một raw pointer thật sự, thì các toán tử đặc trưng của pointer thông thường như " \* "  và " → "  đều được overload lại trong các đối tượng này. Nhờ vậy mà nó trông rất giống các raw pointer, và ta có thể sử dụng nó như là một raw pointer thật sự. Ngoài ra thì nó còn được định nghĩa để tự động xử lý các vấn đề liên quan đến quản lý bộ nhớ tự động.

Cách mà các smart pointer trong C++ hoạt động, để thực hiện việc tự động quản lý tài nguyên, đều dựa vào khái niệm "Quyền sở hữu tài nguyên". Do đó trong C++ có 3 loại smart pointer chính là:

* **unique\_ptr: Loại smart pointer đại diện cho "Quyền sở hữu duy nhất". Nghĩa là tài nguyên mà unique\_ptr trỏ tới chỉ có thể được sở hữu bởi duy nhất một đối tượng.**
* **shared\_ptr: Loại smart pointer này đại diện cho "Quyền sở hữu chia sẻ". Nghĩa là tài nguyên mà shared\_ptr trỏ tới là tài nguyên chia sẻ, có thể được sở hữu bởi nhiều đối tượng.**
* **weak\_ptr: Loại smart pointer này đại diện cho "Quyền sở hữu yếu". Nghĩa là đối tượng nắm trong tay weak\_ptr trỏ tới một tài nguyên thì nó chỉ có quyền được sử dụng tài nguyên đó, chứ không có quyền và nghĩa vụ hủy đi tài nguyên.**

Các loại smart pointer trên xuất hiện từ C++11, từ trước C++11 vẫn có một loại smart pointer nữa đó là auto\_ptr. Nhưng loại pointer này không còn được khuyến khích sử dụng nữa, vì nó không định nghĩa hành vi "Chuyển quyền sở hữu" một cách rõ ràng, và dễ dàng gây ra lỗi nếu người dùng không nắm rõ nó. Do vậy nó đã được thay bằng unique\_ptr kể từ C++11 (Nhưng auto\_ptr vẫn còn giữ lại trong C++11 để đảm bảo tính tương thích ngược).

1. unique\_ptr
2. Giới thiệu

Điểm mấu chốt của smart pointer trong C++ nằm ở vấn đề về "Quyền sở hữu". Do đó, khi nhắc đến bất kì một smart pointer nào trong C++, thì điều đầu tiên mà chúng ta phải nhắt đến là cách mà nó thực thi "Quyền sở hữu" để thực hiện tự động quản lý bộ nhớ.

unique\_ptr trong C++ đại diện cho "Quyền sở hữu duy nhất". Nghĩa là tài nguyên mà unique\_ptr trỏ tới chỉ được sở hữu bởi duy nhất một đối tượng, và trên lý thuyết một tài nguyên chỉ được trỏ tới bởi duy nhất một unique\_ptr (nếu lập trình viên không vi phạm nguyên tắc của unique\_ptr). Có lẻ đến đây, chúng ta sẽ tự hỏi : "Tại sao lại giới hạn quyền sở hữu của tài nguyên mà unique\_ptr trỏ tới là duy nhất?". Để trả lời cho câu hỏi đó chúng ta phải xét trên trên nhiều yếu tố.

Đầu tiên là về tính phổ biến của loại tài nguyên. Tài nguyên chiếm phổ biến nhất trong chương trình thông thường là tài nguyên chỉ được sở hữu bởi duy nhất một đối tượng tại một thời điểm (nếu chúng ta biết cách sử dụng hiệu quả chúng), điển hình là các mảng cấp phát động, và các field member được cấp phát trên heap trong đối tượng (Giả sử bạn có một lớp Matrix, bên trong nó có con trỏ để trỏ tới mảng hai chiều chứa dữ liệu của ma trận, và thường mảng hai chiều này chỉ được sử dụng bên trong lớp Matrix, do đó tài nguyên này chỉ được sở hữu duy nhất bởi một đối tượng Matrix).

Điểm thứ hai mà có lẽ chúng ta dễ dàng nhận ra, đó là về vấn đề performance. Khi ta giới hạn được phạm vi của tài nguyên, chúng ta sẽ quản lý nó một cách dễ dàng hơn, do đó việc quản lý sẽ gây tốn ít chi phí hơn.

1. Đặc điểm

unique\_ptr đại diện cho "Quyền sở hữu duy nhất", tài nguyên mà nó trỏ tới chỉ được sở hữu bởi duy nhất một đối tượng, và tài nguyên đó cũng chỉ được trỏ tới bởi duy nhất một unique\_ptr. Do đó ta không thể thực hiện "gán thông thường" (bằng copy constructor hoặc toán tử copy assignment) đối với unique\_ptr, vì khi đó thì cả hai unique\_ptr sẽ cùng trỏ tới một tài nguyên, gây vi phạm nguyên tắc của unique\_ptr. Thay vào đó, để chuyển tài nguyên mà unique\_ptr này trỏ tới sang unique\_ptr kia, ta sử dụng move constructor hoặc toán tử move assignment (Được định nghĩa dựa theo "Move semantics", để hiểu rõ hơn phần này, các bạn có thể đọc thêm bài "Rvalue Reference và Move Semantics").

1. Chi phí sử dụng

Thông thường, các pointer được sử dụng rất thường xuyên trong chương trình, do đó, chi phí (overhead) của nó là một điều ta cần phải quan tâm đầu tiên, vì dù nó chỉ gây tốn lượng chi phí nhỏ, nhưng cũng có thể đủ để làm chậm đáng kể performance của chương trình.

Thực tế, vì unique\_ptr chỉ quản lý các tài nguyên "sở hữu duy nhất", nên việc định nghĩa nó rất dễ dàng, và hầu như nó hoàn toàn không tốn thêm bất kì chi phí nào so với khi sử dụng raw pointer. Bên trong unique\_ptr chỉ chứa duy nhất một con trỏ để trỏ tới tài nguyên mà nó quản lý (trừ trường hợp bạn có sử dụng thêm deleter cho unique\_ptr) nên hầu như kích thước của unique\_ptr cũng bằng kích thước của một raw pointer. Ngoài ra thì việc truy xuất giá trị (sử dụng toán tử "\*"), và truy xuất thành viên (sử dụng toán tử "->") của unique\_ptr chỉ tốn thêm một lần truy xuất bộ nhớ so với raw pointer. Do vậy, hầu như ta có thể dùng unique\_ptr để thay thế hoàn toàn cho raw pointer đối với các "tài nguyên sở hữu duy nhất", vì lợi ích mà nó mang lại lớn hơn rất nhiều so với chi phí cần thiết.

1. shared\_ptr
2. Giới thiệu

Khác với unique\_ptr, shared\_ptr sẽ đại diện cho "Quyền sở hữu chia sẻ". Nghĩa là tài nguyên mà shared\_ptr trỏ tới là tài nguyên chia sẻ, có thể được sở hữu bởi nhiều đối tượng cùng một lúc. Nhờ shared\_ptr mà chúng ta có thể dễ dàng quản lý các tài nguyên được sử dụng bởi nhiều đối tượng một cách dễ dàng. Khi sử dụng shared\_ptr với các loại tài nguyên này, ta hoàn toàn không còn cần phải quan tâm việc thu hồi nó, không còn phải nhức óc để nghĩa xem tài nguyên đó sẽ phải được thu hồi khi nào, và ở đâu.

shared\_ptr được dùng để quản lý các tài nguyên được chia sẻ bởi nhiều đối tượng, nhưng chúng ta vẫn có thể dùng nó với tất cả các loại tài nguyên khác, shared\_ptr như một "smart pointer tổng quát", có thể sử dụng trong tất cả các trường hợp, và chúng ta có thể dùng shared\_ptr để thay thế hoàn toàn raw pointer, và coi nó là một công cụ quản lý tài nguyên tự động cực kì hiệu quả. Nhưng như chúng ta thường thấy, trường hợp tổng quát bao giờ cũng là trường hợp có mức tối ưu trung bình. shared\_ptr có thể gây tốn rất nhiều chi phí và việc lạm dụng nó thật sự sẽ làm giảm performance của chúng ta một cách đáng kể.

1. Chi phí sử dụng

Tài nguyên chia sẻ là tài nguyên có thể được sử dụng bởi nhiều đối tượng trong chương trình, việc quản lý các tài nguyên này sẽ rất khó khăn và gây tốn rất nhiều chi phí. Để quản lý loại tài nguyên này, shared\_ptr được định nghĩa phức tạp hơn unique\_ptr, sử dụng cơ chế "Reference counting" để đảm bảo khi không còn shared\_ptr trỏ tới tài nguyên đó nữa thì tài nguyên sẽ bị thu hồi, và sử dụng biến atomic để bảo thread-safe khi tài nguyên đó được sử dụng bởi các đối tượng nằm ở các thread khác nhau.

Do vậy, tùy vào trường hợp cụ thể, và tùy vào cách mà tài nguyên đó sẽ được sở hữu (duy nhất hay chia sẻ?) để ta quyết định có nên sử dụng shared\_ptr hay không.

1. Vấn đề gặp phải

shared\_ptr giúp làm giúp chúng ta việc tự động quản lý tài nguyên, để giúp giảm thiểu các vấn đề như memory leak. Nhưng cơ chế "Reference counting" mà shared\_ptr sử dụng vẫn có một nhược điểm lớn, và nó sẽ gây ra memory leak nếu gặp phải trường hợp nhất định. Bản thân cơ chế "Reference counting" thực hiện việc thu hồi tài nguyên tự động dựa vào số lượng con trỏ đang trỏ tới tài nguyên đó, khi số lượng con trỏ này về 0 thì tài nguyên sẽ được thu hồi. Nhưng nếu giả sử ta có hai tài nguyên A và B (là hai đối tượng), và hai tài nguyên này sở hữu lẫn nhau (A có một shared\_ptr trỏ tới B, và B có một shared\_ptr trỏ tới A), khi đó thì "Reference counter" (Biến đếm con trỏ trỏ tới tài nguyên đó) của cả 2 shared\_ptr trong A và B không bao giờ về 0, do vậy sẽ gây ra memory leak, vì A và B không bao giờ được thu hồi tự động. Tình huống này được người ta gọi là "Circular references".

Để giải quyết vấn đề về "Circular references" thì trong C++ xuất hiện thêm một loại smart pointer nữa, gọi là weak\_ptr.

1. weak\_ptr
2. Giới thiệu

Vì khi sử dụng shared\_ptr, đôi khi ta sẽ gặp phải vấn đề về "Circular references", gây memory leak. Do vậy weak\_ptr được sinh ra để giải quyết vấn đề đó.

weak\_ptr đại diện cho "Quyền sở hữu yếu". Nghĩa là đối tượng sở hữu tài nguyên bằng weak\_ptr thì nó chỉ có quyền sử dụng, chứ không có quyền hay nghĩa vụ thu hồi tài nguyên. Hay nói cách khác, cơ chế "Reference counting" chỉ dựa vào số lượng shared\_ptr đang trỏ tới tài nguyên, chứ không quan tâm tới việc tài nguyên đó đang được trỏ tới bởi bao nhiêu weak\_ptr (Dù đang có 5 weak\_ptr đang trỏ tới tài nguyên đó, nhưng không còn shared\_ptr nào trỏ tới nữa thì tài nguyên đó vẫn bị thu hồi).

Do vậy, weak\_ptr được sử dụng với các shared\_ptr trong trường hợp "Circular dependencies" (có thể gây nên "Circular references").

1. Chi phí sử dụng

weak\_ptr cũng có một điểm yếu rất lớn khi sử dụng. Đó là, mỗi lần muốn sử dụng tài nguyên mà weak\_ptr trỏ tới, ta cần phải thực hiện câu lệnh lock() để tạo ra 1 shared\_ptr trỏ tới tài nguyên đó (Sẽ được nói rõ hơn ở "Phần 2: Sử dụng smart pointer"), mà chi phí để copy-constructing một shared\_ptr (Tạo ra 1 đối tượng shared\_ptr bằng copy constructor của nó, thông qua một shared\_ptr khác đã trỏ tới tài nguyên) là rất lớn so với các câu lệnh tính toán thông thường.

1. Kết luận

C++11 có 3 loại smart pointers để phục vụ cho những mục đích khác nhau, do đó, tùy vào cách mà tài nguyên đó được sở hữu ta sẽ lựa chọn loại smart pointer phù hợp để tránh lỗi cũng như giảm thiểu chi phí khi sử dụng nó. Do vậy, việc hiểu rõ về smart pointer và cách nó hoạt động là rất cần thiết để sử dụng đúng cách và hiệu quả.

### C++ 11 Sử dụng smart pointer

1. Mô hình RAII

Smart pointer triển khai dựa trên mô hình lớp RAII để quản lý vùng nhớ cấp phát động. Đây là kỹ thuật nhằm liên kết vòng đời của tài nguyên (memory, thread, socket, file, mutex, database connection… - và tất cả mọi thức tồn tại trong một thời gian nào đó) tới một đối tượng.

Các lớp vẫn dùng phương thức cũ như open/close, lock/unlock, hay init/copyFrom/destroy được xem như non-RAII class.

1. Kết hợp hàm dựng và hàm hủy

Thư viện chuẩn của C++ được thiết kế chống rò rỉ dựa trên cơ chế quản lý dùng hàm dựng/hàm hủy.

Quan trọng hơn, phương pháp này có quan hệ mật thiết với xử lí ngoại lệ. Ví dụ về kỹ thuật dùng lớp thư viện chuẩn **lock** để quản lý mutex

mutex m; // used to protect access to shared data

// ...

void f(){

**unique\_lock**<mutex> lck {m}; // acquire the mutex m

// ... manipulate shared data ...

}

Thread không thực thi tới khi hàm dựng đối tượng **unique\_lock** nhận được mutex, khi hàm kết thúc, hàm hủy **unique\_lock** tự động mở khóa mutex khi thread của phương thức f() hoàn tất.

Kỹ thuật RAII là ý tưởng xử lí tài nguyên trong C++, triển khai trên nhiều container như vector, map, string và iostream.

1. Thư viện chuẩn RAII

Có thể kể ra là: [std::string](http://en.cppreference.com/w/cpp/string/basic_string), [std::vector](http://en.cppreference.com/w/cpp/container/vector), [std::thread](http://en.cppreference.com/w/cpp/thread/thread), và nhiều lớp khác, chỉ cần khai báo sử dụng mà không phải tường minh thủ tục hủy đối tượng.

Ngoài ra, thư viện chuẩn cung cấp các RAII wrapper là:

* [std::unique\_ptr](http://en.cppreference.com/w/cpp/memory/unique_ptr) và [std::shared\_ptr](http://en.cppreference.com/w/cpp/memory/shared_ptr) để quản lý vùng nhớ cấp phát động.
* [std::lock\_guard](http://en.cppreference.com/w/cpp/thread/lock_guard), [std::unique\_lock](http://en.cppreference.com/w/cpp/thread/unique_lock), [std::shared\_lock](http://en.cppreference.com/w/cpp/thread/shared_lock) để quản lý mutexes.

1. Unique\_ptr

**Cú pháp khởi tạo:**

**std::unique\_ptr<int> p1(new int)** // khởi tạo đối tượng integer, con trỏ p1 sở hữu đối tượng này

**std::unique\_ptr<int> p1(new int(100))** // khởi tạo đối tượng integer có giá trị 100

**std::unique\_ptr<int[]> p1(new int[10])** // khởi tạo mảng 100 đối tượng integer

Để chuyển quyển sở hữa 2 con trỏ, dùng **std::move**()

#include <iostream>

#include <memory>

struct Foo{

Foo() { std::cout << "Foo::Foo\n"; }

~Foo() { std::cout << "Foo::~Foo\n"; }

void bar() { std::cout << "Foo::bar\n"; }

};

void f(const Foo &){

std::cout << "f(const Foo&)\n";

}

int main(){

std::unique\_ptr<Foo> p1(new Foo); // p1 sở hữu Foo

if (p1) p1->bar()

{

std::unique\_ptr<Foo> p2(std::move(p1)); // giờ p2 sở hữa Foo

f(\*p2); // \* được quá tải trả về đối tượng Foo

p1 = std::move(p2); // trả quyền sở hữa về p1

std::cout << "destroying p2...\n";

}// p2 tự động hủy khi ra khỏi phạm vi

if (p1) p1->bar();// -> quá tải để có thể trỏ tới thành viên của Foo

}// Foo tự động hủy khi con trỏ p1 ra khỏi phạm vi

Không dùng các phép toán trên con trỏ unique\_ptr được. Bạn để ý 2 phương thức:

* Release(): bỏ quyền sở hữu tài nguyên, nhưng không thu hồi tài nguyên của con trỏ
* Reset(): bỏ quyền sở hữu và thu thồi tài nguyên con trỏ

****

**Lưu ý: chỉ con trỏ non-const unique\_ptr mới có thể trao quyền sở hữu. Tuy nhiên, const unique\_ptr chỉ tồn tại trong phạm vi khai báo mà thôi.**

1. Shared\_ptr

**Cú pháp:**

**std::shared\_ptr<int> p(new int)** // tạo đối tượng với new

**std::shared\_ptr<int[]> p(new int[100])** // tạo mảng đối tượng với new

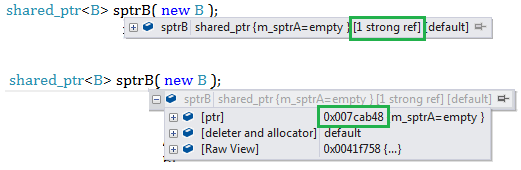
**std::shared\_ptr<int> p = std::make\_shared<int>()** // tạo đối tượng với make\_shared

**std::shared\_ptr<int> p = std::make\_shared<int>(100)** // tạo đối tượng giá trị 100 với make\_shared

**std::shared\_ptr<int[]> p = std::make\_shared<int>[5]** // tạo mảng 5 đối tượng Int với make\_shared

**std::shared\_ptr<Base> p = std::make\_shared<Derived>()** // cho phép chuyển kiểu ngầm

Bạn có thể lấy số tham chiếu đang trỏ tới tài nguyên của shared\_ptr bằng cách dùng phương thức use\_count( ). Khi debug, để ý thông số stong\_ref của shared\_ptr



shared\_ptr cũng đã quá tải các toán tử \*, -> cho giống với con trỏ thuần.

**Kiểm tra trạng thái tài nguyên**

* get( ) : trả về tài nguyên sở hữu bởi con trỏ shared\_ptr.
* reset( ) : bỏ quyền sở hữa đối với tài nguyên. Nếu shared\_ptr là chủ sở hữu cuối cùng, tài nguyên tự động được thu hồi.
* unique: kiểm tra xem có phải tài nguyên sở hữu bởi duy nhất đối tượng shared\_ptr hiện tại.
* operator bool: kiểm tra xem đối tượng shared\_ptr có sở hữu tài nguyên hay không. Có thể dùng với câu lệnh if.

**Vấn đề phát sinh:**

**Tình huống 1: không nên khởi tạo shared\_ptr từ con trỏ thuần**

Xét đoạn ví dụ sau

#include <memory>

using namespace std;

void main( ){

shared\_ptr<int> sptr1( new int );

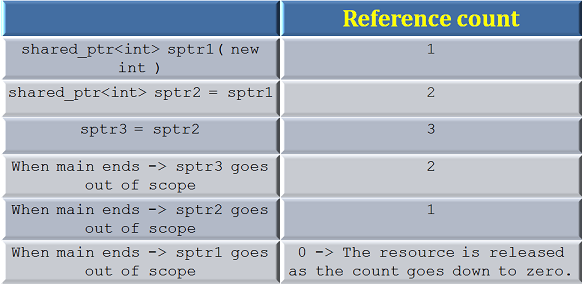
shared\_ptr<int> sptr2 = sptr1;

shared\_ptr<int> sptr3;

sptr3 = sptr2;

}

Bảng theo dõi bộ đếm tham chiếu tới tài nguyên con trỏ:



Đoạn code ở trên không có vấn đề, xem phần dưới:

void main( ){

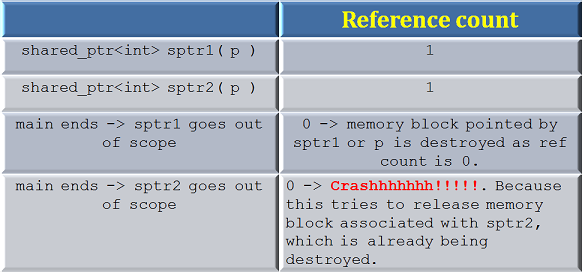
int\* p = new int;

shared\_ptr<int> sptr1( p);

shared\_ptr<int> sptr2( p );

}

Chương trình bị crash vì tài nguyên con trỏ sptr2 đã bị hủy.



**Tình huống 2: tham chiếu vòng**

#include <memory>

#include <iostream>

using namespace std;

class B;

class A{

public:

A() : m\_sptrB(nullptr) { };

shared\_ptr<B> m\_sptrB;

};

class B{

public:

B() : m\_sptrA(nullptr) { };

shared\_ptr<A> m\_sptrA;

};

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void main( ){

shared\_ptr<B> sptrB(new B );

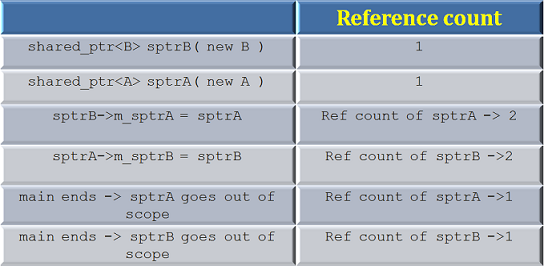
shared\_ptr<A> sptrA(new A );

sptrB->m\_sptrA = sptrA;

sptrA->m\_sptrB = sptrB;

}

Con trỏ thành viên shared\_ptr của A đang trỏ tới B và của B đang trỏ tới A, trường hợp này ta gọi là tham chiếu vòng. Xem bảng theo dõi số tham chiếu:



Tham chiếu tới sptrA và sptrB còn 1 khi ra khỏi main. Do đó tài nguyên không được thu hồi.

Đễ tránh tham chiếu vòng, C++ cung cấp con trỏ weak\_ptr.

1. Weak\_ptr

Được sinh ra để giải quyết vấn đề tham chiếu vòng. Con trỏ weak \_ptr  cũng được quá tải toán tử \*, ->. Nó chỉ có quyền sử dụng tài nguyên chứ không sở hữu hay có nghĩa vụ phải hủy. Bộ đếm tham chiếu sẽ bỏ qua weak \_ptr.

**Cú pháp:**

**weak\_ptr<int> wptr(shared\_ptr<int>(new int))** //nhận shared\_ptr là tham số

**weak\_ptr<int> wptr = sptr** //gán với shared\_ptr

Mỗi khi con trỏ weak pointer tạo ra, bộ đếm tham chiếu của nó tự động tăng lên một.

Xem ví dụ:

#include <iostream>

#include <memory>

using namespace std;

class Test{

//implement...

};

void main( ){

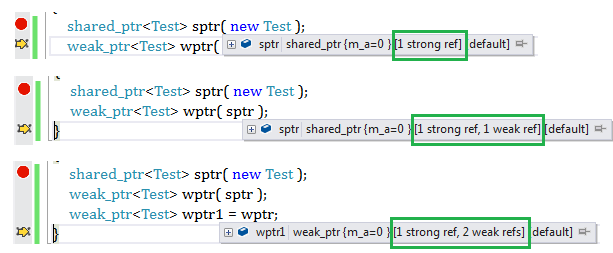
shared\_ptr<Test> sptr( new Test );

weak\_ptr<Test> wptr( sptr );

weak\_ptr<Test> wptr1 = wptr;

}

Debug, theo dõi số tham chiếu trên shared/weak pointer.



Phép gán giữa 2 weak pointer cũng làm tăng số tham chiếu trên bộ đếm của weak pointer.

**Kiểm tra tài nguyên của weak\_ptr:**

Để biết tài nguyên của shared\_ptr mà weak\_ptr trỏ tới bị thu hồi hay chưa, có 2 cách:

1. Dùng use\_count( ) : trả về số tham chiếu shared\_ptr (strong reference).
2. Dùng expired( ): để biết xem weak\_ptr đã hết hạn chưa, cách này nhanh hơn cách 1

**Cast shared\_ptr từ weak\_ptr:**

Để lấy shared\_ptr từ weak\_ptr dùng phương thức lock( ) .

void main( ){

shared\_ptr<Test> sptr(new Test);

weak\_ptr<Test> wptr(sptr);

shared\_ptr<Test> sptr2 = wptr.lock();

}

Hay ép kiểu trực tiếp weak\_ptr sang shared\_ptr.

**Giải quyết tham chiếu vòng với weak\_ptr:**

#include <iostream>

#include <memory>

using namespace std;

class B;

class A{

public:

A() : m\_a(5) {};

weak\_ptr<B> m\_sptrB;

int m\_a;

};

class B{

public:

B() : m\_b(10) {};

weak\_ptr<A> m\_sptrA;

int m\_b;

};

void main(){

shared\_ptr<B> sptrB(new B );

shared\_ptr<A> sptrA(new A );

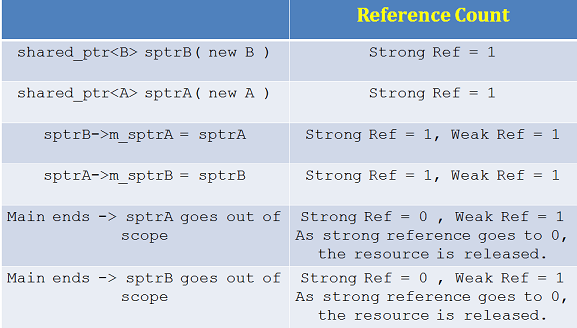
sptrB->m\_sptrA = sptrA;

sptrA->m\_sptrB = sptrB;

sptrA->PrintSpB( );

}

Bảng đếm tham chiếu:



1. Nên dùng con trỏ nào?

Tùy thuộc vào cách bạn dùng tai nguyên, nếu muốn chia sẻ trên nhiều con trỏ/ nhiều phạm vi, dùng shared\_ptr.

Ngược lại, tài nguyên không chia sẻ thì dùng unique\_ptr.

Tuy nhiên shared\_ptr sẽ tốn hiệu suất hơn unique\_ptr vì phải duy trì bảng đếm số tham chiếu shared\_ptr và weak\_ptr.

### C++11 OOP – override – delete & default

##### Override

1. Giới thiệu

Override là việc viết lại một phương thức (method) trong lớp dẫn xuất (derive class) mà ở lớp cơ sở (base class).

1. class CReptile{
2. public:
3. char\* getClassName(int sign){
4. if (sign > 0){
5. return "CReptile";
6. }
7. }
8. };
10. class CSnake: public CReptile{
11. public:
12. char\* getClassName(int sign){
13. if (sign > 0){
14. return "CSnake";
15. }
16. }
17. };

Ở codes trên, phương thức getClassName() trong class CSnake override phương thức getClassName() trong class CReptile .

1. Rủi ro của codes trên
2. class CReptile
3. {
4. public:
5. char\* getClassName(int sign)
6. {
7. if (sign > 0){
8. return "CReptile";
9. }
10. }
11. };
13. class CSnake: public CReptile
14. {
15. public:
16. char\* getClassName(char sign){
17. if (sign > 0){
18. return "CSnake";
19. }
20. }
21. };

Đối số trong phương thức getClassName trong class CReptile mang kiểu int, do sơ suất ta có thể làm cho đối số trong getClassName trong class CSnake mang kiểu char. Và mục đích ban đầu ta override hàm getClassName đã thất bại, đây được xem như một hàm mới.

1. Từ khóa override trong C++11

Trong phiên bản C++11 có hỗ trợ thêm một từ khóa là override, ý nghĩa của từ khóa này sẽ giúp ta kiểm soát việc một hàm có thật sự là override một phương thức của lớp mà nó đang kế thừa hay không, nếu không tồn tại phương thức mà nó đang "nghĩ" rằng nó đang override tại lớp mà nó kế thừa thì trình biên dịch sẽ báo lỗi khi biên dịch. Ta thêm từ khóa override vào cuối tên phương thức như sau **(khảo sát dòng 16)**.

1. class CReptile{
2. public:
3. char\* getClassName(int sign) {
4. if (sign > 0)
5. return "CReptile";
6. }
7. };
9. class CSnake: public CReptile{
10. public:
11. char\* getClassName(char sign) **override** {
12. if (sign > 0)
13. return "CSnake";
14. }
15. };

Do phương thức getClassName(char sign) không có tồn tại trên lớp cơ sở, nên khi biên dịch sẽ lỗi. Từ đó, ta có thể biết rằng, ta đang "tạo ra một hàm mới" hay gọi là overload thay vì override.

##### Delete & default

**Delete** trong OOP mang nghĩa vô hiệu một hàm thành viên:

class X {

// ...

X& operator=(const X&) = delete; // ngăn không cho sao chép

X(const X&) = delete;

};

Ngược lại, **default** dùng để khai báo tường minh rằng hàm sao chép này là mặc định:

class Y {

// ...

Y& operator=(const Y&) = default; // hành vi sao chép mặc định

Y(const Y&) = default;

};

Khai báo mặc định một hành vi có vẻ là hơi thừa. Tuy nhiên, để đề phòng trường hợp các lập viên tồi khi định nghĩa toán tử gán, khai báo **default** cũng phổ biến.

**Default** giúp trình biên dịch biết dùng hành vi nào, ít lỗi, tối ưu cho đối tượng.

Cơ chế của **default** và **delete** áp dụng mọi hàm thành viên.

struct Z {

// ...

Z(long long); // có thể khởi tạo với kiểu long long

Z(long) = delete; // nhưng long thì không

};

## Chuyên đề C++14

### Generic Lambda

1. Giới thiệu

Trong C++11 - Lambda :: [www.stdio.vn/articles/read/56/c11-lambda](http://www.stdio.vn/articles/read/56/c11-lambda) đã được hỗ trợ và giúp giải quyết được 1 vấn đề tồn đọng lâu nay đó là, hàm chỉ được gọi tại 1 chỗ thì định nghĩa hàm tại chỗ đó thay vì định nghĩa 1 hàm dùng chung nhưng thật ra chỉ gọi ở 1 vị trí trong codes (cần lưu ý là Lambda khác vớiInline Functions Trong C++ :: [www.stdio.vn/articles/read/13/inline-functions-trong-c](http://www.stdio.vn/articles/read/13/inline-functions-trong-c)).

Với C++14, Lambda đã làm nhiều hơn như vậy, không đơn thuần là nó là cách thức gõ code và mã của nó khi biên dịch sẽ được tạo ra ngay, mà nó có vai trò như 1 template, và vị trí nào sử dụng Lambda này với các kiểu dữ liệu tương ứng thì trình biên dịch sẽ sinh ra hàm tương ứng kiểu dữ liệu đó.

1. Tiền đề bài viết

Sau bài viết C++14 - Từ Khóa auto Và Khả Năng Tự Động Nhận Kiểu Khi Hiện Thực Một Hàm ::[www.stdio.vn/articles/read/348/c14-tu-khoa-auto-va-kha-nang-tu-dong-nhan-kieu-khi-hien-thuc-mot-ham](http://www.stdio.vn/articles/read/348/c14-tu-khoa-auto-va-kha-nang-tu-dong-nhan-kieu-khi-hien-thuc-mot-ham)tôi thấy để bổ sung đầy đủ hơn cho auto thì cần chia sẻ thêm điều này nữa.

1. Đối tượng hướng đến

Dành cho các độc giả có hứng thú với điều mới lạ nhưng cần thiết phải nắm được C++11 ở mức cơ bản nhất là phần C++14 - Từ Khóa auto Và Khả Năng Tự Động Nhận Kiểu Khi Hiện Thực Một Hàm ::[www.stdio.vn/articles/read/348/c14-tu-khoa-auto-va-kha-nang-tu-dong-nhan-kieu-khi-hien-thuc-mot-ham](http://www.stdio.vn/articles/read/348/c14-tu-khoa-auto-va-kha-nang-tu-dong-nhan-kieu-khi-hien-thuc-mot-ham)và C++11 - Lambda :: [www.stdio.vn/articles/read/56/c11-lambda](http://www.stdio.vn/articles/read/56/c11-lambda).

1. Nhắc lại Lambda trong C++11

Trong C++11, biểu thức Lambda được định nghĩa như sau

1. #include <iostream>
3. int main()
4. {
5. auto max = [](int a, int b) {
6. return a > b ? a : b;
7. };
9. std::cout << max(1, 2);
11. return 0;
12. }

Dòng 5, 6, 7 là biểu hiện của biểu thức Lambda. Đọc thêm bài C++11 - Lambda ::[www.stdio.vn/articles/read/56/c11-lambda](http://www.stdio.vn/articles/read/56/c11-lambda) để nắm lại kiến thức về biểu thức này.

1. Vấn đề của C++11

Ở C++11, biểu thức Lambda kiểu như sau sẽ không được chấp nhận.

1. auto max = [](auto a, int b) {
2. return a > b ? a : b;
3. };
5. std::cout << max(1, 2);
6. std::cout << max(1.2, 5.6);

Trình biên dịch sẽ báo lỗi ở dòng 1 vì auto a không được C++11 thiết kế chức năng đó, nó không có khả năng biết được a là kiểu gì.

1. Lambda trong C++14

Ở C++14, Lambda cho phép auto trong việc truyền tham số, xét ví dụ dưới đây (với C++11 thì không thể, nhưng với C++14 thì có thể).

1. auto max = [](auto a, int b) {
2. return a > b ? a : b;
3. };
5. std::cout << max(1, 2);
6. std::cout << max(1.2, 5.6);

Vấn đề sâu hơn là tại sao nó lại có thể biết trước được kiểu dữ liệu để biên dịch? Chắc chắn là không.

Bạn lưu ý dòng 5 và dòng 6, với a là 1 thì kiểu dữ liệu là int và với a là 1.2 thì kiểu dữ liệu sẽ là double. Điều này làm ta liên tưởng đến template Trong C++ :: [www.stdio.vn/articles/read/16/template-trong-c](http://www.stdio.vn/articles/read/16/template-trong-c) và điều liên tưởng này là đúng. Tại thời điểm biên dịch, thì Lambda chưa được biên dịch mà trình biên dịch chỉ "note" lại template của hàm. Cho đến những nơi gọi hàm và chỉ định kiểu cụ thể, trong trường hợp trên là dòng 5 thì 1 chính là kiểu int và dòng 6 thì 1.2 là kiểu double, lúc này trình biên dịch mới dựa vào template đó và sinh ra hàm phù hợp.

Lưu ý, điều này chưa áp dụng cho 1 hàm, ta chưa có Generic Function, nhưng nếu có, có thể template cho hàm không còn tồn tại nữa.

### Từ khóa Auto

1. Giới thiệu

Trong C++14, từ khóa auto đã được cập nhật nhiều hơn về mặt tính năng, đó là khả năng nhận dạng kiểu dữ liệu được trả về từ trong một hàm, với C++11 chỉ hỗ trợ cho biểu thức Lambda.

Để trải nghiệm được C++14 bạn cần GCC 4.8+ hoặc Visual Studio 2015.

1. Nhắc lại auto trong C++11

C++11 - Từ Khóa auto :: [www.stdio.vn/articles/read/54/c11-tu-khoa-auto](http://www.stdio.vn/articles/read/54/c11-tu-khoa-auto) có khả năng tự động nhận dạng được kiểu dữ liệu của khai báo biến hoặc kiểu trả về trong 1 lambda.

1. int sum(int a, int b)
2. {
3. return a + b;
4. }
6. int main()
7. {
8. // i kieu int vi 5 la kieu int
9. auto i = 5;
11. // c kieu char vi 'k' la kieu char
12. auto c = 'k';
14. // s kieu const char\* vi "www.stdio.vn" kieu const char\*
15. auto s = "www.stdio.vn";
17. // r kieu int vi ham sum co kieu tra ve la int
18. auto r = sum(1, 2);
20. // max co kieu tra ve la int vi Lambda return la 1 kieu int
21. // max se co dinh nghia day du la auto max = [](int a, int b)
22. {
23. return a > b ? a : b;
24. };
26. // x la kieu int vi max co kieu tra ve la int
27. auto x = max(1, 2);
29. return 0;
30. }

Trong C++11, không có khả năng dự đoán được kiểu trả về của một hàm do đó, cú pháp sau sẽ không tồn tại.

1. auto sum(int a, int b)
2. {
3. return a + b;
4. }
5. auto trong C++14

Với C++14, trình biên dịch hỗ trợ thêm tính năng nhận dạng kiểu dữ liệu được trả về từ hàm dựa vào kiểu dự liệu trong hàm trả về.

1. auto func(int a, int b)
2. {
3. if (a > b)
4. return a;
5. else
6. return b;
7. }
9. int main()
10. {
11. int t = func(1, 2);
13. return 0;
14. }

Như hàm func trên, kiểu trả về của a (dòng 4) và b (dòng 6) đều là kiểu int nên trình biên dịch sẽ dự đoán được auto ở dòng 1 phải là kiểu int và biên dịch sẽ thành công.

Đánh đố trình biên dịch hơn như sau

1. auto func(int a, double b)
2. {
3. if (a > b)
4. return a;
5. else
6. return b;
7. }

Ở dòng 1 thì a là kiểu int còn b là kiểu double, do đó, việc return sẽ gây ra 2 trường hợp là int haydouble (dòng 4 và 6) làm bối rối trình biên dịch không biết lựa chọn trường hợp nào. Trong trường hợp này, ta có thể đơn giản hóa là quay lại cách code thông thường đó là chỉ định kiểu trả về là kiểu gì hoặc kiểu trả về theo kiểu của Lambda hoặc nếu kiểu dữ liệu quá dài, để đỡ tốn thời gian có thể kết hợp thêm decltype (xem thêm C++11 - Từ Khóa decltype :: [www.stdio.vn/articles/read/55/c11-tu-khoa-decltype](http://www.stdio.vn/articles/read/55/c11-tu-khoa-decltype)).

1. // Cach thong thuong
2. int func(int a, double b)
3. {
4. if (a > b)
5. return a;
6. else
7. return b;
8. }
10. // Cach nhu Lambda
11. auto func(int a, double b) -> int
12. {
13. if (a > b)
14. return a;
15. else
16. return b;
17. }
19. // Cach nhu Lambda ket hop decltype
20. auto func(int a, double b) -> decltype(a)
21. {
22. if (a > b)
23. return a;
24. else
25. return b;
26. }

Công nghệ hỗ trợ chúng ta phát triển sản phẩm nhanh, nhưng nó dành cho nhà phát triển, bạn đọc thuộc nhóm đang phát triển sản phẩm thì sử dụng công nghệ là đúng nhưng những bạn đọc đang trong quá trình học tập theo tôi nên học cách chế tạo ra bánh xe, không nên dùng bánh xe vào thời điểm này.