Contents

[Getting Started – Bắt Đầu: 1](#_Toc157157164)

[Preprocessor Directive – Tiền Xử Lý Điều Hướng: 2](#_Toc157157165)

[Program – Chương Trình: 7](#_Toc157157166)

[Memory – Bộ Nhớ: 8](#_Toc157157167)

[Stream: 9](#_Toc157157168)

[Namespace: 12](#_Toc157157169)

[Primitive Data Type – Kiểu Dữ Liệu Cơ Bản: 13](#_Toc157157170)

[Pointer: 14](#_Toc157157171)

[Enum: 21](#_Toc157157172)

[Operator – Toán Tử: 22](#_Toc157157173)

[Loop – Vòng Lặp: 26](#_Toc157157174)

[Condition – Điều Kiện: 27](#_Toc157157175)

[Function – Hàm: 28](#_Toc157157176)

[Class: 32](#_Toc157157177)

[Quick Code – Mã Nhanh: 35](#_Toc157157178)

[Cheat Engine: 35](#_Toc157157179)

[Debug: 36](#_Toc157157180)

Getting Started – Bắt Đầu:

1. Phần Mở Rộng Của C++ File?

* Đéo quan trọng, nhưng thông thường xài “.cpp”

1. Cách Cài Đặt C++?

* Vào Link này
* <https://sourceforge.net/projects/orwelldevcpp/>
* Rồi Click “Download” + chờ tải + mở File vừa tải + chọn “English” + Click “OK” + nhấn “I Agree” + nhấn “Next” + nhấn “Install” + chờ cài đặt + bỏ Tick “Run Dev-C++ …” + Click “Finish”
* Khi này, trong thư mục “C:/Program Files (x86)” sẽ xuất hiện thư mục “Dev-Cpp”
* Tiếp theo, chạy Dev C++ = cách Click vào biểu tượng của nó ngoài Desktop + Click “Next” 2 lần + Click “OK” + Click “Yes” + thoát Dev C++
* Cuối cùng, thêm “C:\Program Files (x86)\Dev-Cpp\MinGW64\bin” vào biến hệ thống “Path”, trong thư mục này sẽ có chứa trình biên dịch C++, GNU Debugger, …
* Để từ các File C++ ra 1 EXE duy nhất phải qua 2 bước
* Bước 1, biên dịch ra các File C++ Object với phần mở rộng “.o”, mỗi File C++ Object thực chất là mã nhị phân giống trong EXE nhưng không thể chạy

g++ <Đường Dẫn Tới Từng File C++ Có Phần Mở Rộng> -c

* Các File C++ Object mặc định sẽ trùng tên với File C++ tương ứng
* Ví dụ

g++ foo.cpp bar.cpp -c

* Khi này 2 File “foo.o” và “bar.o” sẽ được tạo ra
* Bước 2, liên kết các File C++ Object với nhau và với các thư viện, ví dụ trường hợp ta muốn tách dự án ra nhiều File C++, và không muốn dùng lệnh Import, nhưng vẫn muốn từ File này sử dụng hàm của File khác, = cách tạo hàm nguyên mẫu trong File này nhưng chưa viết phần thân cho nó, mà viết phần thân trong File khác

g++ <Đường Dẫn Tới Các File C++ Object Có Phần Mở Rộng> -o <Đường Dẫn Tới File EXE Không Cần Phần Mở Rộng>

* Trong các File C++ Object phải chứa đúng 1 File có hàm main làm ngõ vào
* Ví dụ

g++ foo.o bar.o -o alice

* Để tạo ra File EXE ngay thay vì làm 2 bước, nhập lệnh sau vào CMD

g++ <Bản> <Đường Dẫn Đến Các File C++ Có Phần Mở Rộng> -o <Đường Dẫn Đến File EXE Không Cần Phần Mở Rộng> <Bit>

* Mặc định

|  |  |
| --- | --- |
| <Bản> | -std=c++98 |
| <Bit> | -m64 |

* Đặt <Bit> = -m32 để nó biên dịch ra chương trình 32 Bit, nếu không sẽ biên dịch ra 64 Bit

|  |  |
| --- | --- |
| <Bản> | Phiên bản |
| -std=c++14 | C++ 2014 |
| -std=c++11 | C++ 2011 |
| -std=c++98 | C++ 1998 |

* Ví dụ

g++ -std=c++11 foo.cpp -o foo -m32

* Để biên dịch kèm theo thông tin Debug như Index dòng của các lệnh, kiểu dữ liệu biến, … vào File EXE, thêm Flag -g
* Ví dụ

g++ -g -std=c++11 foo.cpp -o foo -m32

1. Cách Show Phiên Bản G++?

* Vào CMD nhập

g++ --version

1. Cách Show Phiên Bản C++?

* Phiên bản được lưu vào hằng toàn cục sau

\_\_cplusplus

* Giá trị tương ứng

|  |  |
| --- | --- |
| 199711 | C++ 1998 |
| 201103 | C++ 2011 |
| 201300 | C++ 2014 |

1. Comment?

* Comment được xử lí trước tiền xử lý điều hướng, nghĩa là bản chất thay Comment = không gì cả
* Để Comment từ vị trí A trên 1 dòng đến hết dòng, đặt // tại A
* Ví dụ

foo // bar bob

alice john

* Tương đương

foo

alice john

* Khi trình biên dịch bắt gặp // nó sẽ ngay lập tức nhảy tới dòng tiếp theo đéo quan tâm kí tự tiếp theo sau //
* Để Comment 1 đoạn Code, đặt chúng trong cặp /\* và \*/
* Ví dụ

foo /\* concu

alice john

con buom\*/ bu lon

* Tương đương

foo

bu lon

* Khi trình biên dịch bắt gặp /\*, nó sẽ bỏ qua từng kí tự một đéo quan tâm kí tự đó là gì cho đến khi bắt gặp \*/ thì mới chạy Code tiếp

Preprocessor Directive – Tiền Xử Lý Điều Hướng:

1. Cách Hoạt Động?

* Thực thi trước trình biên dịch, công đoạn này gọi là tiền xử lí, làm các việc như Copy Paste Code, bỏ qua không biên dịch 1 đoạn Code khi một điều kiện nào đó được thỏa mãn, …

1. Macros?

* Khi bạn viết như sau

#define <Nhãn> <Nội Dung Thay Thế>

* <Nhãn> chỉ bao gồm 1 từ
* Trong giai đoạn tiền xử lí, bất cứ chỗ nào từ dòng lệnh này trở đi có từ <Nhãn> đứng riêng lẻ giống như 1 biến, không nằm trong dấu nháy kép, sẽ bị thay thế bởi <Nội Dung Thay Thế>, ngoại trừ các chỗ trong các Preprocessor Directive khác
* Nếu sau khi thay thế, <Nội Dung Thay Thế> lại chứa 1 Macro khác, thì Macro này tiếp tục được thay thế, cứ như vậy cho đến khi không còn Macro, trong công đoạn này, mỗi Macro khác nhau chỉ thực hiện thay thế 1 lần
* Ví dụ

#define ga foo

#define bar foo bob

#define foo bar

ga

* Tương đương

foo bob

* Đầu tiên, ga biến thành foo, sau đó foo biến thành bar, sau đó bar biến thành

foo bob, nhưng Macro foo đã được dùng rồi nên dừng lại ở đây không thế nữa

* <Nội Dung Thay Thế> có thể là bất cứ thứ gì, kể cả câu lệnh, vì nó sẽ Copy thẳng sang
* Ví dụ

#define foo cout << bar << endl;

int bar = 1;

foo

int far = 2;

* Tương đương

int bar = 1;

cout << bar << endl;

int far = 2;

* Có thể khai báo Macro trùng tên nhiều lần, khi này cái cũ sẽ bị xóa, lấy cái mới nhất
* Trong trường hợp <Nhãn> có dạng 1 hàm, các tham số cũng sẽ được thế y chang những gì bạn Pass
* Ví dụ

#define foo(a, b) a + b\*4

int bar = 5;

cout << foo(bar, 2+5) << endl;

* Tương đương

int bar = 5;

cout << bar + 2+5\*4 << endl;

* Đặt dấu # đằng trước 1 biến sẽ bao nó trong cặp dấu nháy kép
* Ví dụ

#define foo(a, b) cout << #a << b;

* Tương đương

#define foo(a, b) cout << "a" << b;

* Ví dụ trong hàm ta chỉ nhận 2 tham số là a và b, trong Macro ta không thể viết cout << ab được vì tham số ab không tồn tại, thay vào đó ta có thể thêm ## giữa các biến để nối chúng lại với nhau mà không có khoảng trống
* Ví dụ

#define foo(a, b, c) cout << a##b##c;

foo(bar, bob, john)

* Tương đương

cout << barbobjohn;

* Để xóa bỏ sự tồn tại của Macro kể từ dòng Code sau trở đi

#undef <Nhãn>

* Nếu Macro có dạng hàm, thì không <Nhãn> chỉ cần tên hàm, không cần đối số
* Để khai báo Macro trên nhiều dòng, đặt dấu \ sau mỗi dòng trước khi xuống dòng
* Ví dụ

#define foo bar long mu \

123 deh hieu \

That la\

ngu

int bar = 2;

foo

* Tương đương

int bar = 2;

bar long mu 123 deh hieu That langu

1. Bỏ Qua Không Biên Dịch 1 Đoạn Code?

#if <Điều Kiện>

<Đoạn Code>

#endif

* <Điều Kiện> được viết dưới dạng biểu thức toán học trong C++ như bình thường, chỉ được phép chứa số, toán tử, và nhãn của Macro đã được định nghĩa
* Nếu <Điều Kiện> trả về True, thì <Đoạn Code> sẽ được biên dịch, còn trả về False, thì bị bỏ qua không biên dịch
* ở giữa #if và #endif, ta có thể thêm #else hoặc #elif, cơ chế tương tự như Else và Else If trong câu điều kiện, nếu có #else thì nó phải được đặt dưới cùng
* ví dụ

#if <Điều Kiện 1>

<Đoạn 1>

#elif <Điều Kiện 2>

<Đoạn 2>

#elif <Điều Kiện 3>

<Đoạn 3>

#else

<Đoạn 4>

#endif

* Nếu <Điều Kiện 1> trả về True, <Đoạn 1> được biên dịch, <Đoạn 2, 3, 4> bị bỏ qua không biên dịch, nếu trả về False, thì <Đoạn 1> không được biên dịch, mà chuyển sang kiểm tra <Điều Kiện 2>, nếu nó trả về True thì <Đoạn 2> được biên dịch, <Đoạn 3, 4> không được biên dịch, nếu trả về False thì <Đoạn 2> không được biên dịch, tiếp tục chuyển sang kiểm tra <Điều Kiện 3>, nếu True,

<Đoạn 3> được biên dịch, <Đoạn 4> không được biên dịch, nếu False, thì

<Đoạn 3> không được biên dịch, và <Đoạn 4> được biên dịch do nó sau #else

* <Điệu Kiện> của #if hoặc #elif có thể bao gồm toán tử defined như sau

defined <Nhãn Macro>

* <Nhãn Macro> phải là 1 từ
* Khi này biểu thức trên sẽ trả về True nếu Macro tương ứng đã được khai báo, trả về False nếu chưa được khai báo hoặc đã bị xóa
* Ta có thể sử dụng tiền xử lí sau

#ifdef <Nhãn Macro>

* Tương đương

#if defined <Nhãn Macro>

* Hoặc để trả về True khi Macro không tồn tại và False khi tồn tại

#ifndef <Nhãn Macro>

* Tương đương

#if !defined <Nhãn Macro>

1. Header File?

* Các Header File tiêu chuẩn của C++ được lưu ở thư mục “C:\Program Files (x86)\Dev-Cpp\MinGW64\lib\gcc\x86\_64-w64-mingw32\4.9.2\include\c++”
* Khi bạn viết

#include <<Tên Header File Tiêu Chuẩn>>

* Thì trình biên dịch sẽ mò tới đường dẫn trên để chạy Header File có tên là

<Tên Header File Tiêu Chuẩn>, tương đương việc thay thế cái dòng lệnh này = nguyên nội dung trong Header File đó luôn

* Bất cứ File nào đặt vào đường dẫn trên đều trở thành Header File tiêu chuẩn và có thể được Import như trên
* Cú pháp của Code trong Header File giống cú pháp bình thường
* Ví dụ

#include <iostream>

#include <foo.h>

* Để Import Header File do mình tự tạo ra thì viết

#include "<Đường Dẫn Tới File>"

* Cách viết đường dẫn y chang trong Batch
* Ví dụ

#include "..\bar.h"

* Phần mở rộng Header File đéo quan trọng, bạn đặt hay không thì tùy

1. Cách Không Import Thư Viện Nhiều Lần?

* Giả sử ta có File A là File C++ chính, là ngõ vào của chương trình, trong A ta có Import File B và C, trong C ta cũng lại Import B, như vậy tương đương với việc ta Copy Paste 2 lần B vào A, để tránh việc này, trong File B phải có tiền xử lí sau, đặt ở đâu cũng được

#pragma once

* Khi này, đầu tiên B được Import vào A như thường, nhưng khi C được Import vào A, thì gặp lệnh Import B trong C, thì do B đã Import vào A rồi, nó không được Import nữa

1. Cách Không Padding Giữa Các Thuộc Tính Trong Class?

#pragma pack(<N>)

* <N> phải là lũy thừa của 2, ví dụ 1, 2, 4, 8, …
* Khi gặp chỉ thị trên, thì tất cả các khai báo Class hoặc Struct phía dưới, ví dụ Class A, khai báo int a, double c, char d, giữa các thuộc tính này sẽ Padding tùy theo <N>, nghĩa là mỗi thuộc tính với kích thước từ <N> trở lên sẽ phải đặt ở chỗ sao cho ô Byte đầu tiên của nó có địa chỉ chia hết cho <N>, đồng thời kích thước của A cũng phải chia hết cho <N>, thuộc tính đầu tiên vẫn đặt ở địa chỉ chia hết cho 16
* Ví dụ

#pragma pack(4)

struct foo {

int a;

short b;

char c;

double d;

char e;

int f;

short g;

}

* Khi này các thuộc tính được lưu như sau

|  |  |  |
| --- | --- | --- |
| Địa chỉ | Nội dung |  |
| 93 |  |  |
| 92 |  |  |
| 91 | Padding | foo |
| 90 |
| 89 | g |
| 88 |
| 87 | f |
| … |
| 84 |
| 83 | Padding |
| … |
| 81 |
| 80 | e |
| 79 | d |
| … |
| 72 |
| 71 | Padding |
| 70 | c |
| 69 | b |
| 68 |
| 67 | a |
| … |
| 64 |

* Như vậy dễ thấy nếu <N> = 1, thì không có Padding
* Nếu <N> = 0, thì trở lại Padding mặc định, tức là <N> = kích thước của thuộc tính có kích thước lớn nhất

1. Template?

* Có tác dụng tạo ra hàm hoặc Class tại Run Time, không phải tại Compile Time, hàm chạy xong thì bị xóa khỏi bộ nhớ, không như hàm thông thường được lưu cố định trong phần nhớ Text

template<<Danh Sách Biến Thế>> <Hàm Hoặc Class>

* Nghĩa là mỗi Template chỉ có tác dụng cho 1 hàm hoặc Class duy nhất đằng sau nó
* <Danh Sách Biến Thế> gồm các khai báo biến ngăn cách bởi dấu phẩy
* <Hàm Hoặc Class> là đoạn mã khai báo hàm hoặc Class, mà bên trong nó, có sử dụng các <Biến Thế>
* Khi gọi <Hàm Hoặc Class> như gọi hàm, hoặc tạo Object cho Class, …, ta cần thế các <Biến Thế> = giá trị cụ thể
* Trường hợp hàm

<Hàm><<Các Giá Trị Biến Thế>>(<Các Tham Số Hàm>);

* Ví dụ

template <typename foo, typename boo, int bar, char alice>

void far(foo dark, foo light, int near){

boo haha = dark + light + near;

cout << bar << alice;

}

* Khi hàm far được gọi, ta phải Pass cho nó giá trị cụ thể của các <Biến Thế>
* Biến foo, boo có kiểu dữ liệu cơ bản là typename, tức là nó có giá trị là int, float, … theo nghĩa đen
* Biến bar có kiểu dữ liệu cơ bản là int, nên nó phải có giá trị là 1, 2, hay 10, …, tương tự alice

far<int, float, 4, 'c'>(10, 8, 45);

* Khi lệnh trên chạy, hàm far mới được tạo ra trong RAM, và nó như sau, thế giá trị các <Biến Thế> vào

void far(int dark, int light, int near){

float haha = dark + light + near;

cout << 4 << 'c';

}

far(10, 8, 45);

* Ta có thể tạo hàm mới với kiểu dữ liệu cơ bản mới

far<double, char, 8, '\n'>(1.23, 8.99, 9);

* Nếu Template chỉ bao gồm các <Biến Thế> kiểu typename, và <Đối Tượng> là hàm với phần tham số có Full các <Biến Thế>, thì không cần phải Pass giá trị cụ thể của <Biến Thế> khi gọi hàm, nó tự động hiểu từ tham số Pass vào
* Ví dụ

template<typename a, typename b>

void foo(a foo, b bob, int c){

cout << 1;

}

foo('c', 4, 5);

* Khi này a tự động = char, b tự dộng = int

Program – Chương Trình:

1. Cách Hoạt Động Của 1 File EXE Được Biên Dịch Từ File C++?

* Khi chạy File EXE này, ngay lập tức nó sẽ chạy thẳng hàm “main” trong đó, hàm này phải trả về Integer, nếu không trả về gì cả, tự động trả về 0, đây chính là Exit Code

1. Nhảy Đến Câu Lệnh Bất Kì?

* Y chang Go To trong Batch
* Ví dụ

int bar = 5;

goto foo;

bar = 10;

foo: bar++;

* Ta có

|  |  |
| --- | --- |
| foo | 6 |

1. Cấu Trúc Chương Trình?

* Ta chỉ được phép khai báo hàm và biến, không được dùng các câu lệnh nếu chúng không nằm trong hàm
* Ví dụ, Code sau sai

int a;

a = 4;

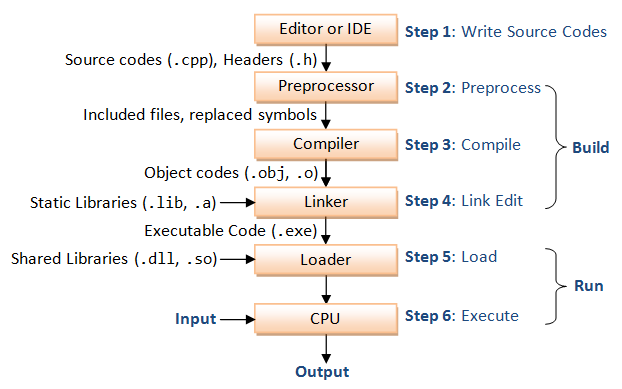
int main(){

return 0;

}

1. Linker?

* Diễn biến từ lúc viết Code đến lúc chạy chương trình



* Linker có tác dụng nối các đối tượng và thư viện rời rạc lại với nhau thành 1 File EXE hoàn chỉnh để thực thi

Memory – Bộ Nhớ:

1. Thứ Tự Bit Little Endian?

* Khi bạn lưu 1 biến trong máy tính, ví dụ có dạng nhị phân

01011111 00011111 11111111 00000000

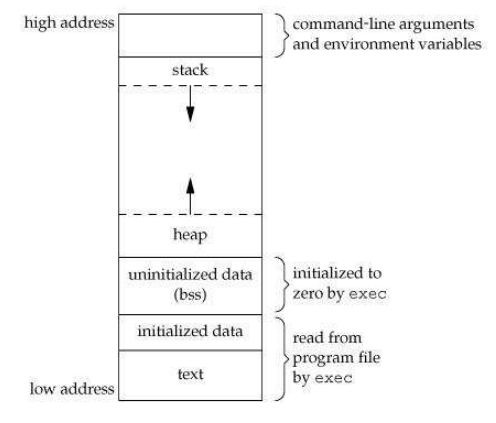
* Thì theo chiều tăng dần địa chỉ, nó được lưu như sau

00000000 11111111 00011111 01011111

* Như vậy, thứ tự Byte bị đảo ngược

1. Tổng Quan Bộ Nhớ?

* Khi bắt đầu chương trình, C++ ngay lập tức chia các vùng bộ nhớ trong RAM như sau



* Đây là thanh RAM, chiều tăng dần địa chỉ từ dưới lên
* Khi chương trình chạy 1 hàm, hàm này sẽ được đẩy vào Stack, hệ thống sẽ cấp phát vùng nhớ cho nó, vùng này gọi là khung hàm, khi 1 hàm A gọi tới 1 hàm B, thì khung hàm của B sẽ được tạo và đặt dưới khung hàm của A trong Stack
* Ví dụ khi A đang chạy, các Object mặc định được lần lượt tạo ra trong A sẽ được đặt trong khung hàm của A lần lượt từ trên xuống, Object có kích thước K thì sẽ phải đặt ở địa chỉ sao cho ô Byte đầu tiên của nó chia hết cho K, riêng Object không thuộc kiểu dữ liệu cơ bản phải được đặt sao cho ô Byte đầu tiên của nó chia hết cho 16, do đó giữa các Object có thể có khoảng trống (Padding)
* Ví dụ, ta tạo lần lượt các Object sau, Class foo có kích thước 15 Byte

int a, foo b, int c, double d, char f, char g, int h

* Thì chúng sẽ được lưu trong Stack như sau, giả sử ban đầu a được lưu ở 80

|  |  |
| --- | --- |
| Địa chỉ | Nội dung |
| 83 | a |
| … |
| 80 |
| 79 | Padding |
| 78 | b |
| … |
| 64 |
| 63 | c |
| … |
| 60 |
| 59 | Padding |
| … |
| 56 |
| 55 | d |
| … |
| 48 |
| 47 | f |
| 46 | g |
| 45 | Padding |
| 44 |
| 43 | h |
| … |
| 40 |

* Kích thước khung hàm được xác định trước bởi trình biên dịch dựa vào số biến trong hàm
* Ví dụ A gọi B, B chạy xong thì khung hàm của nó sẽ biến mất, sau đó A chạy xong thì khung hàm của A cũng biến mất
* Khung hàm của hàm main được đặt trên cùng
* Kích thước Stack cố định, do đó nếu chồng chất quá nhiều khung hàm cùng lúc sẽ bị lỗi tràn Stack
* Heap là bộ nhớ động, ta có thể ngẫu nhiên chọn 1 ô trong đó để lưu giá trị, càng đặt nhiều biến vào Heap thì các biến tiếp theo càng bị đặt ở phía trên
* Stack nhanh hơn Heap do cấp phát bộ nhớ thường xuyên trên này, do đó các tập lệnh có thể được lưu trong bộ nhớ đệm của CPU, đồng thời cấu trúc đơn giản
* BSS là nơi lưu trữ các biến nào chưa được khởi tạo giá trị
* Phần vùng nhớ khởi tạo lưu các Object toàn cục, Object tĩnh và hằng toàn cục, đó là lí do vì sao Object tĩnh trong hàm giá trị có thể thay đổi được sau mỗi lần gọi
* Ngoài ra ta cũng có thể lưu biến trong Register để tăng tốc độ xử lí
* Text là vùng nhớ dùng để lưu chương trình, tức là các Instruction, mỗi hàm trong chương trình sẽ được lưu ở đây dưới dạng các dãy Instruction liên tục, khi gọi hàm thì các Instruction này sẽ được thực thi lần lượt

Stream:

1. Buffer?

* Bản chất của Console giống như 1 File, bạn ghi/đọc dữ liệu vào Console = ghi/đọc dữ liệu vào File
* Object cout và printf mặc định lưu trữ con trỏ đến Buffer của Console
* Object cin và scanf mặc định lưu trữ con trỏ đến Buffer của bàn phím
* Bạn có thể thay đổi con trỏ của cout hoặc cin thành con trỏ vào Buffer của 1 File
* Khi dùng lệnh in như cout hay printf, thì các kí tự được gửi vào Buffer nó chĩa vào trước
* Các kí tự trong Buffer sau đó thỏa mãn 1 điều kiện nào đó thì mới bị đẩy tới đích như Console hoặc File
* Trong C++ chỉ có 2 Buffer có đích là Console tương ứng với printf và cout, tạm gọi là A và B, tuy nhiên mặc định B chưa được sử dụng bởi cout, và cả printf và cout đều sử dụng chung A, giả sử kích thước của A chứa tối đa 1023 kí tự, tương tự B, thì toàn bộ 1023 kí tự trong Buffer này sẽ được đẩy ra màn hình một khi Buffer chứa Full 1023 kí tự, sau đó làm rỗng nó
* Toàn bộ kí tự trong A hoặc B cũng tự động bị đẩy ra khi dùng lệnh flush ép nó đẩy ra, hoặc ngay khi chèn kí tự xuống dòng vào nó, hoặc khi chương trình kết thúc, riêng đối với A, dữ liệu trong nó tự động bị đẩy ra do hệ điều hành sau mỗi khoảng thời gian ngắn
* Tương tự Buffer của File y chang cách hoạt động của B, thêm việc khi File Object ứng với nó đóng lại thì dữ liệu cũng được đẩy vào đích
* Đồng thời toàn bộ kí tự trong Buffer mà cout chõ đến cũng bị đẩy ra đích ngay khi chương trình gặp lệnh cin, sau đó chờ người dùng nhập Input, bạn cũng có thể dùng lệnh gỡ ràng buộc để nó không bị đẩy khi gặp lệnh cin, lí do cho sự ràng buộc này là vì sẽ tốt hơn khi câu hỏi được đẩy hết ra ngoài rồi nhập Input hợp lí hơn
* 1 Buffer có thể được trỏ đến từ nhiều Object, mỗi Object sẽ có các cờ riêng cho nó không ảnh hưởng tới cờ của Object khác như fixed, left, right, …, và khi dùng Object này ghi vào Buffer thì sẽ dùng cờ tương ứng của Object đó để định dạng thứ được ghi vào
* Trong C++ chỉ có 2 Buffer chính dùng để lưu dữ liệu nhập từ bàn phím ứng với scanf và cin, tạm gọi là X và Y, tuy nhiên mặc định Y chưa được sử dụng bởi cin, và cả printf và cin đều sử dụng chung X
* Bất cứ khi nào scanf hoặc cin được gọi, nếu trong Buffer nó dùng chứa dữ liệu theo đúng định dạng nó yêu cầu, thì dữ liệu đó sẽ được đẩy ra, đích là địa chỉ nào đó như địa chỉ của các biến
* Ta có thể tạo ra đối tượng tương tự cin và scanf nhưng sử dụng bộ đệm được Copy từ 1 File sang
* Khi thoát khỏi chế độ mặc định, tức là cout không dùng A nữa mà dùng B, cin không dùng X nữa mà dùng Y, printf vẫn dùng A, scanf vẫn dùng X, thì tốc độ chương trình trong khoản nhập xuất sẽ nhanh hơn do không còn phải dung hòa nữa

1. Printf?

* Thuộc thư viện <iostream>

printf(<String Nội Suy>, <Giá Trị Nội Suy 1>, <Giá Trị Nội Suy 2>, …);

* Các kí tự nội suy trong <String Nội Suy> sẽ được thay lần lượt = các

<Giá Trị Nội Suy>

* Cú pháp kí tự nội suy

%<Kiểu Cách><Độ Rộng>.<Số Chữ Số Thập Phân><Kiểu Dữ Liệu Cơ Bản>

* Các kiểu dữ liệu cơ bản

|  |  |
| --- | --- |
| d | int |
| ld | long |
| lld | long long |
| lu | unsigned long |
| c | char |
| f | float, double viết bình thường, mặc định dùng cờ fixed 6 chữ số sau dấu phẩy |
| E, e | float, double dùng cờ scientific viết theo kiểu khoa học |
| s | Mảng char |

* Các kiểu cách

|  |  |
| --- | --- |
| + | Số dương sẽ hiện thêm dấu “+” đằng trước, tương đương dùng cờ showpos |
| - | Căn lề trái thay vì căn lề phải trong phần không gian được xác định bởi <Độ Rộng>, tương đương dùng cờ left |
| 0 | Lấp đầy phần không gian còn lại = số 0 |

* Ví dụ

printf("Hello Gay : %+15.3f", 3.145644879);

* Nội dung thêm vào Buffer

Hello Gay : +3.146

1. Scanf?

* Thuộc thư viện <iostream>

scanf(<String Nội Suy>, <Địa Chỉ 1>, <Địa Chỉ 2>, …);

* Lệnh trên sẽ tạm dừng chương trình cho đến khi Buffer do scanf sử dụng chứa chuỗi kí tự giống định dạng trong <String Nội Suy> hoặc dư hoặc không trùng định dạng, đồng thời cho phép người dùng nhập từ bàn phím
* Mỗi lần người dùng nhập thứ gì đó rồi nhấn “Enter”, thì thứ gì đó sẽ được đẩy vào Buffer do scanf sử dụng
* Khi chương trình hêt tạm dừng, dữ liệu trong Buffer sẽ được xử lí, gọi toàn bộ chuỗi kí tự trong Buffer này là A, kí tự đầu tiên của A là kí tự tồn tại lâu nhất trong A
* <String Nội Suy> trước khi xử lí sẽ rút gọn tất cả các chuỗi dấu cách liền nhau thành 1 dấu cách, đồng thời nếu trước kí tự nội suy chưa có dấu cách thì sẽ thêm 1 dấu cách, ví dụ "ab cd eff%d vailon" thành "ab cd eff %d vailon", gọi String này là B
* Bước 1, 2 con trỏ C và D trỏ vào kí tự đầu tiên của A và B
* Bước 2, nếu D là kí tự thường, thì đối chiếu với C, nếu C và D trùng nhau thì cả 2 cùng dịch sang phải 1 kí tự, nếu không thì kết thúc đọc
* Bước 3, nếu D là kí tự dấu cách, thì xét C, nếu C là kí tự thường thì đứng yên tại chỗ, nếu là dấu cách hoặc kí tự xuống dòng thì C dịch sang phải cho đến khi gặp kí tự không phải dấu cách hay kí tự xuống dòng
* Bước 4, nếu D ở vị trí kí tự nội suy, thì xét chuỗi ngắn nhất từ vị trí C trở đi có định dạng giống kí tự nội suy, rồi chuyển kiểu dữ liệu của chuỗi này theo kiểu của kí tự nội suy, rồi gán vào <Địa Chỉ 1>, sau đó C dịch chuyển tới vị trí kế sau chuỗi ngắn nhất này và D dịch chuyển sang kí tự tiếp theo
* Cứ tiếp tục như vậy, <Địa Chỉ 2>, <Địa Chỉ 3>, … sẽ lần lượt được gán giá trị, nếu trong quá trình lặp mà gặp phải giá trị không hợp lệ thì sẽ ngừng gán và lệnh trên kết thúc, đồng thời chuỗi dư còn lại vẫn sẽ còn tồn tại trong Buffer
* Nếu số <Địa Chỉ> không đủ so với số kí tự nội suy thì chương trình sẽ văng
* Cú pháp kí tự nội suy

%<Có Bỏ Qua><Độ Rộng><Kiểu Dữ Liệu Cơ Bản>

* Các kiểu dữ liệu cơ bản

|  |  |
| --- | --- |
| d | int |
| f | float |
| s | Mảng char đã xác định kích thước, đối số địa chỉ thì chỉ cần ghi tên mảng, không cần &, chỉ đọc đến kí tự ngay trước dấu cách hoặc kí tự xuống dòng đầu tiên, ghi đè chứ không xóa rồi ghi lại từ đầu, chuỗi đọc được sẽ thêm kí tự \0 ở cuối trước khi ghi đè vào mảng ban đầu, ghi đè từ vị trí đầu tiế |
| [^<Kí Tự>] | Mảng char đã xác định kích thước, đối số địa chỉ thì chỉ cần ghi tên mảng, không cần &, đọc đến kí tự ngay trước kí tự <Kí Tự> đầu tiên, cơ chế ghi giống ở trên |
|  |  |

* <Độ Rộng> có thể là 1 số hoặc không có gì cả, nếu không có gì thì sẽ đọc số lượng kí tự tối đa có thể, nếu là 1 số thì chỉ đọc tối đa đến số đó kí tự, có thể đọc ít, dấu chấm cũng tính là 1 kí tự, nếu giá trị đọc vượt quá kích thước của kiểu dữ liệu thì sẽ bị Clip, vì dụ nếu Max là 100 mà đọc được 1234, thì sẽ giá trị đọc từ 1234 giảm còn 100, hoặc Min là -100 mà đọc được -1234 thì sẽ từ -1234 lên

-100

* <Có Bỏ Qua> có thể là kí tự \* hoặc không có kí tự gì cả, nếu là kí tự \* thì giá trị đọc được không được gán vào địa chỉ nào cả, do đó địa chỉ hiện tại tiếp tục chờ kí tự nội suy tiếp theo để gán giá trị cho nó
* Ví dụ 1

int foo;

float bar;

scanf("concu %da%f", &foo, &bar);

* Chuỗi nội suy được xử lí thành "concu %da %f"
* Ta có, [E] là nhấn Enter

|  |  |  |
| --- | --- | --- |
|  | foo | bar |
| Trước khi nhập | 123 | 45.56 |
| "concu789a1.34[E]" | 789 | 1.34 |
| "concu 789 a 1.34[E]" | 789 | 45.56 |
| "concu 789a 1.34[E]" | 789 | 1.34 |
| "concu 789[E]" | 789 | 45.56 |
| "concu[E]  789a [E]  1.34[E]" | 789 | 1.34 |
| "con cu789a1.34[E]" | 123 | 45.56 |
| " con cu789a1.34[E]" | 123 | 45.56 |

* Ví dụ 2

int foo;

float alice;

scanf(" %3da%\*2d%5f concu", &foo, &alice);

* Chuỗi nội suy được xử lí thành " %3da %\*2d %5f concu"
* Ta có

|  |  |  |
| --- | --- | --- |
|  | foo | alice |
| Trước khi nhập | 123 | 7.89 |
| "357a99671.3987[E]" | 357 | 671.3 |
| " 357a 99 6.71thangcho[E]" | 357 | 6.71 |
| " 357a 997 6.71thangcho[E]" | 357 | 78 |
| " 357a 9 6.71thangcho[E]" | 357 | 6.71 |
| " 35789a 9 6.71thangcho[E]" | 357 | 7.89 |
| " 35 a 9 6.71thangcho[E]" | 35 | 7.89 |

* Ví dụ 3

int foo = 3;

scanf("%d ", &foo);

* Ở trường hợp này, nếu bạn chỉ nhập, ví dụ 14 rồi nhấn Enter, thì Console vẫn tiếp tục chờ bạn nhập tiếp, đó là do dấu cách đằng sau %d, nên bạn nhập bao nhiêu dấu cách đằng hay kí tự xuống dòng đằng sau 14, thì nó vẫn tính vào dấu cách này, và chỉ dừng lại khi bạn nhập 1 kí tự khác dấu cách và kí tự xuống dòng
* Ta có

|  |  |  |
| --- | --- | --- |
|  | foo | Buffer |
| Trước khi nhập | 3 |  |
| "357 [E]  [E]  [E]  [E]  ab[E]" | 357 | ab |

1. Kí Tự Thoát (Escape Character)?

* Carriage Return

\r

* Đây là kí tự đưa con trỏ trở về đầu dòng, và những gì được ghi sau đó sẽ đè lên những kí tự cũ
* Ví dụ

ggnore\rlon

* In ra

lonore

* Line Feed

\n

* Đây là kí tự đưa con trỏ xuống dòng tiếp theo, giữ nguyên hoành độ
* Ví dụ

ggnore\nlon

* In ra

ggnore

lon

* Tuy nhiên, khi biên dịch Code C++ trong hệ điều hành Windows, tất cả kí tự \n sẽ bị thay thế bằng \r\n, đây coi như là 1 kí tự, tên là kí tự xuống dòng (New Line)
* Back Space
* \b
* Kí tự này lùi con trỏ 1 kí tự, đồng thời xóa kí tự ở vị trí lùi
* Ví dụ

ggnore\bc

* In ra

ggnorc

Namespace:

1. Bản Chất Namespace?

* Chỉ có tác dụng gộp nhiều hàm hoặc thuộc tính vào chung 1 gói, để truy cập hàm hoặc thuộc tính thì phải thông qua tên của gói này
* Mục đích để tránh xung đột khi tạo ra 2 hàm hay thuộc tính có tên giống nhau
* Namespace không phải 1 phần của thư viện, thư viện chỉ là cái cung cấp phương thức cho Namespace
* Các Object nằm ở ngoài cùng, tức là cùng cấp mới main, đều nằm trong Namespace to nhất, tạm gọi là root, tên của Namespace này là không có gì cả

1. Cách Tạo 1 Namespace?

namespace <Tên Namespace> {

<Khai Báo Các Hàm Hoặc Thuộc Tính>

}

* Ví dụ

namespace foo {

int bar(){

return 5;

}

int fool = 1;

}

* Để sử dụng hàm hay thuộc tính của Namespace

<Tên Namespace>::<Hàm Hoặc Thuộc Tính>

* Ví dụ

int boo = foo::bar();

* Ta có

|  |  |
| --- | --- |
| boo | 5 |

* Trường hợp muốn truy xuất thuộc tính hoặc hàm thuộc Namespace root

::<Thuộc Tính Hoặc Hàm>

* Ta có thể tạo ra 1 thuộc tính hoặc hàm cho Namespace khi ở ngoài Namespace bằng cách sử dụng Object chưa khởi tạo và hàm nguyên mẫu
* Với thuộc tính

<Class Trả Về> <Tên Namespace>::<Tên Thuộc Tính> = <Giá Trị>;

* Với hàm

<Class Trả Về> <Tên Namespace>::<Tên Hàm>(<Các Tham Số>){

<Làm Gì Đó>

}

* Ví dụ

namespace foo {

int bar;

float alice(int, bool);

}

int foo::bar = 5;

float foo::alice(int a, bool b){

cout << 1;

}

* Ta có thể sử dụng cách khai báo này cho Class, thay mỗi từ namespace thành class
* Ví dụ

class foo {

int bar;

float alice(int, bool);

}

int foo::bar = 5;

float foo::alice(int a, bool b){

cout << 1;

}

1. Cách Đưa Các Hàm Và Thuộc Tính Ra Khỏi Namespace?

using namespace <Tên Namespace>;

* Lệnh trên sẽ thêm vào danh sách tìm kiếm của trình biên dịch

<Tên Namespace>

* Ví dụ trong Namespace foo có thuộc tính bar, để truy cập bar, ta sẽ viết foo::bar, nhưng nếu ta chỉ viết bar, thì khi này trình biên dịch sẽ rà soát trong danh sách tìm kiếm của nó, xem Namespace nào có thuộc tính bar để trả về, nếu trong danh sách tìm kiếm có nhiều Namespace cùng chứa bar, thì ta không thể truy xuất bar
* Ví dụ

using namespace foo;

Primitive Data Type – Kiểu Dữ Liệu Cơ Bản:

1. Kích Thước Các Kiểu Dữ Liệu Cơ Bản?

|  |  |  |
| --- | --- | --- |
| Kiểu | Số Bit | Cách mã hóa |
| int | 32 | Y chang số nguyên có dấu |
| unsigned | 32 | Y chang số nguyên không dấu |
| long | 32 | Y chang số nguyên có dấu |
| short | 16 | Y chang số nguyên có dấu |
| float | 32 | Y chang Floating Point, Bit đầu làm dấu, 8 Bit kế làm Exponent, 23 Bit cuối làm Mantissa |
| double | 64 | Y chang Floating Point, Bit đầu làm dấu, 11 Bit kế làm Exponent, 52 Bit cuối làm Mantissa |
| char | 8 | Y chang số nguyên có dấu, nhưng khi in ra thì ra kí tự tương ứng trong ASCII |
| bool | 8 | True là 00000001, False là 00000000 |
| std::string | 64 |  |
| unsigned long long | 64 | Y chang số nguyên không dấu |
| long long | 64 | Y chang số nguyên có dấu |
| unsigned long | 32 | Y chang số nguyên không dấu |
| unsigned short | 16 | Y chang số nguyên không dấu |
| unsigned char | 8 | Y chang số nguyên không dấu, nhưng khi in ra thì ra kí tự tương ứng trong ASCII |
| signed char | 8 | Y chang số nguyên có dấu, nhưng khi in ra thì ra kí tự tương ứng trong ASCII |
| long double | 128 | Y chang Floating Point, nhưng dùng 80 Bit cuối thay vì Full, 48 Bit đầu đều = 0, trong 80 Bit cuối, Bit đầu làm dấu, 15 Bit kế làm Exponent, 64 Bit cuối làm Mantissa, lưu ý Mantissa này đã bao gồm cả 1. ở Bit đầu rồi |
| int\* | 64 | Pointer vào biến có kiểu int |

1. Cơ Chế Chuyển Đổi Kiểu Dữ Liệu Cơ Bản Ngầm Định?

* Khi lấy 1 số có kiểu dữ liệu cơ bản bé hơn tính toán với 1 số có kiểu dữ liệu cơ bản lớn hơn thì số có kiểu dữ liệu cơ bản bé sẽ được đổi sang kiểu dữ liệu cơ bản lớn, kết quả cũng sẽ có kiểu dữ liệu cơ bản lớn, nếu cùng kiểu mà 1 cái không dấu thì coi không dấu to hơn
* Kiểu dữ liệu cơ bản mặc định của giá trị

|  |  |
| --- | --- |
| Giá trị | Kiểu dữ liệu cơ bản mặc định |
| Số ghi theo kí hiệu khoa học hoặc số có dấu chấm có độ lớn không vượt quá khoảng 1.8 \* 10308 | double |
| |Số nguyên| ∈ [0, 231 – 1] | int |
| |Số nguyên| ∈ [231, 261 – 1] | long long |

* Các số ngoài các khoảng này không được dùng độc lập mà phải gán vào 1 biến có kiểu dữ liệu cơ bản đủ lớn để chứa nó
* Để số thực mặc định không phải double mà là float thì ta thêm “f” vào sau, ví dụ 1234.5f

1. Ép Kiểu (Type Casting)?

* Trả về giá trị ở kiểu dữ liệu cơ bản khác

(<Kiểu Dữ Liệu Cơ Bản Mới>) <Giá Trị>

* Cách ép kiểu như trên gọi là C Style Casting
* Ví dụ

float foo = (int) 3.14;

* Ta có

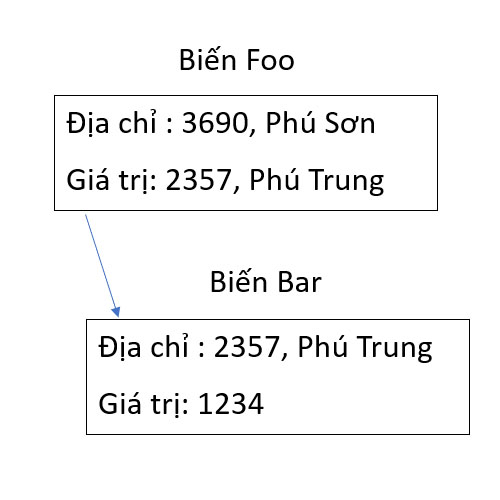
|  |  |
| --- | --- |
| foo | 3 |

1. C++ Style Casting?

* Bản chất = C Style Casting nhưng có Check xem sự chuyển đổi giữa 2 kiểu có hợp lí không

Pointer:

1. Minh Họa Pointer?



* Pointer là Object lưu trữ giá trị là địa chỉ của Object khác
* Kích thước mọi Pointer là 8 Byte
* Để trả về địa chỉ của 1 Object

&<Object>

* Để tạo 1 Pointer

<Pointer Class> <Pointer> = <Địa Chỉ>;

* Nếu <Địa Chỉ> = 0 hoặc False thì tương đương nullptr
* <Pointer Class> sẽ có cú pháp sau

<Class Của Object Pointer Chĩa Vào>\*<Hằng Hay Không Hằng>

* <Class Của Object Pointer Chĩa Vào> bao gồm cả int\*, float\*, int\*\*, … tức là Pointer chĩa vào Pointer
* Nếu chỉ định <Hằng Hay Không Hằng> là const, thì bạn được 1 Const Pointer, nó giống Const Object thông thường, tức là bạn không thể thay đổi cái địa chỉ chứa trong nó
* Sẽ có 4 trường hợp
* 1 Pointer không phải Const Pointer chĩa vào 1 Object không phải Const Object, khi này bạn có thể tùy ý thay đổi cái địa chỉ lưu trong Pointer và có thể thông qua Pointer để thay đổi giá trị cho Object
* 1 Pointer không phải Const Pointer chĩa vào 1 Const Object, khi này bạn có thể tùy ý thay đổi cái địa chỉ lưu trong Pointer nhưng không thể thông qua Pointer để thay đổi giá trị cho Object
* 1 Const Pointer chĩa vào 1 Object không phải Const Object, khi này bạn không thể thay đổi cái địa chỉ đã lưu trong Pointer lúc mới khởi tạo, nhưng có thể thông qua Pointer này để thay đổi giá trị cho Object
* 1 Const Pointer chĩa vào 1 Const Object, khi này bạn vừa không thể thay đổi cái địa chỉ đã lưu trong Pointer lúc mới khởi tạo vừa không thể thông qua Pointer này để thay đổi giá trị cho Object
* Ví dụ
* 1 Pointer không phải Const Pointer chĩa vào 1 Object không phải Const Object

int bar = 5;

int\* foo = &bar;

* Để trả về Object bị Pointer chĩa vào

\*<Pointer>

* Ví dụ

int far = \*foo;

* Ta có

|  |  |
| --- | --- |
| Far | 1234 |

* Để thay đổi giá trị của Object được Pointer chĩa vào

\*<Pointer> = <Giá Trị Mới>;

* Để trả về giá trị của địa chỉ kế tiếp

<Pointer>[<Số Bước>]

* Ví dụ

int boo = 5;

int\* bar = &boo;

int foo = bar[4];

* Ở đây, thay vì dùng thập lục phân, ta dùng thập phân cho dễ nhìn
* Ban đầu, biến boo tạo ra ở ô Byte thứ 100, Pointer bar tạo ra ở ô Byte thứ 92, khi này bar[0] sẽ chỉ vào 4 Byte từ 100 đến 103, nghĩa là chỉ vào boo, bar[1] sẽ chỉ vào 4 Byte từ 104 đến 107, …, bar[4] sẽ chỉ vào 4 Byte từ 116 đến 119, như vậy giả sử các Byte từ 116 đến 119 chứa các Bit sau

|  |  |  |  |
| --- | --- | --- | --- |
| 01010111 | 10111011 | 10101101 | 11011111 |

* Thì từ các Bit này quy đổi sang int, và như vậy foo = –542262441, lưu ý phải đảo các Byte lại trước khi quy đổi
* Tùy theo kích thước của <Class Của Object Pointer Chĩa Vào> mà nhảy 4 Byte, 8 Byte, …
* Ta cũng có thể thay đổi giá trị của vài ô Byte bằng cách này
* Ví dụ

bar[4] = 45;

* Trả về <Pointer 2> cách <Pointer 1> một số bước

<Pointer 2> = <Pointer 1> + <Số Bước>;

* Ví dụ

\*(foo + 2) = 5;

* Tương đương

foo[2] = 5

* Để trả về số bước giữa 2 Pointer

<Số Bước> = <Pointer 1> - <Pointer 2>

* Lấy địa chỉ của <Pointer 1> trừ địa chỉ của <Pointer 2>, sau đó chia cho kích thước của Class của Object do 2 Pointer chĩa vào, được <Số Bước>
* <Pointer 1> và <Pointer 2> phải chĩa vào các Object thuộc chung Class
* <Số Bước> thuộc Class long long
* Ví dụ

int a[5] = {14, 25, 39, 45, 59};

cout << &a[4] - &a[1];

* Buffer

3

1. Void Pointer?

* Là Pointer mà nó có thể chĩa đến bất kì Object nào, nghĩa là nó chỉ có thông tin về địa chỉ của Object
* Khi cần thì bạn có thể chuyển lại về dạng Pointer thường
* Ví dụ

int a = 5;

void\* b = &a;

int\* c = (int\*) b;

* Đây là ví dụ cho ép kiểu Pointer, tức là tạo ra 1 Pointer mới cũng chĩa vào thứ mà Pointer cũ chĩa vào, nhưng khi đọc và ghi thứ đó thông qua Pointer mới, sẽ khác khi ghi và đọc thông qua Pointer cũ, do thuộc Class khác nhau, nên biểu diễn nhị phân sẽ mang những ý nghĩa khác nhau
* Có thể dùng mẹo sau để xem các Bit của 1 Object thuộc kiểu dữ liệu cơ bản là số nguyên dưới dạng thập lục phân

cout << (void\*) <Số Nguyên>;

* <Số Nguyên> phải có kích thước 8 Byte, do kích thước Pointer 8 Byte
* Ví dụ

unsigned long long foo = 123

cout << (void\*) foo;

* Màn hình CMD

0x7b

* 123 = 01001011, do đây được hiểu là địa chỉ nên sẽ hiện 7b

1. Ép 1 Pointer Chĩa Vào Const Object thành 1 Pointer Chĩa Vào Object Không Phải Const Object?

* Giả sử bạn tạo 1 Object không phải Const Object thuộc Class int, gọi là foo, sau đó bạn tạo 1 Pointer bar đến Object này, nhưng đối xử với địa chỉ của Object này như 1 Const Object, tức là không thể thông qua bar để thay đổi giá trị của foo

int foo = 5;

const int\* bar = &foo;

* Khi này, bạn có thể trả về phiên bản đã được ép kiểu của bar từ Const Pointer thành không phải Const Pointer

const\_cast<<Class Của Object Mà Pointer Cũ Trỏ Tới>>(<Pointer Cũ>)

* Lưu ý <Class Của Object Mà Pointer Cũ Trỏ Tới> đã bỏ đi từ khóa const
* Ví dụ

int\* alice = const\_cast<int\*>(bar);

\*alice = 7;

* Khi này foo sẽ có giá trị = 7

1. Copy Bit?

* Có 2 Pointer, A và B, chĩa vào 2 địa chỉ khác nhau, theo chiều tăng dần địa chỉ, bạn muốn Copy 2 Byte từ vị trí của A và Paste nó vào vị trí của B

memcpy(<B>, < A>, <Số Byte Muốn Copy>);

* Ví dụ

memcpy(B, A, 2);

* Ta có

|  |  |
| --- | --- |
| A | 11111111 00000000 10110101 |
| B trước | 10101110 10110110 10110111 10011010 10110110 |
| B sau | 11111111 00000000 10110111 10011010 10110110 |

1. Tham Chiếu (Reference)?

* Tham chiếu bản chất là 1 Object ta dùng nhiều tên để gọi
* Tạo 1 tham chiếu đến 1 Object đã tạo

<Class Của Object> & <Tham Chiếu> = <Tên Biến>;

* Ví dụ

int & foo = bar;

* Ở đây, foo và bar là 1, nên khi gọi foo, ta đang gọi bar và ngược lại, do đó, nếu thay đổi giá trị của foo cũng sẽ thay đổi bar và ngược lại
* <Tham Chiếu> một khi đã gắn với 1 Object thì không thể gắn lại vào Object khác

1. L Value Và R Value?

* L Value giống như tên biến, R Value giống như giá trị, nghĩa là L Value có thể đứng bên trái và phải của toán tử gán, trong khi R Value chỉ được phép đứng bên phải
* Ví dụ

foo = 5;

* Thì foo là L Value còn 5 là R Value, ta không thể viết

5 = 4;

1. Mảng?

* Bản chất 1 mảng là 1 Object mà các thuộc tính của nó thuộc chung 1 Class, tạo thành dãy các Byte liên tiếp lưu trong Stack, ví dụ mảng int thì dãy từng 4 Byte một, theo chiều tăng dần địa chỉ, lưu ý không có Padding giữa các phần tử trong mảng, ví dụ mảng của Class kích thước 15 Byte, thì từng 15 Byte một xếp khít nhau, tuy nhiên, phần tử đầu tiên vẫn phải được đặt ở chỗ sao cho ô Byte đầu tiên của nó chia hết cho kích thước Class, nếu Class không phải kiểu dữ liệu cơ bản, thì chia hết 16
* Cách khởi tạo giá trị ban đầu cho mảng y chang như khởi tạo giá trị ban đầu cho Object thuộc 1 Class nào đó
* Ví dụ tạo 1 mảng với giá trị ban đầu

<Class Của Các Phần Tử> <Object Mảng>[] = <Các Giá Trị Khởi Tạo>;

* [] gọi là toán tử Subscript
* Ví dụ cụ thể

int foo[] = {1, 2, 3, 4, 5};

* Khi tạo như này, thì <Kích Thước Mảng> tự động = số phần tử của

<Các Giá Trị Khởi Tạo>

* Cách viết khác y chang dựa trên cách khởi tạo giá trị cho Object của 1 Class nào đó

<Class Của Các Phần Tử> <Object Mảng>[]<Các Giá Trị Khởi Tạo>;

* Tạo 1 mảng không có giá trị ban đầu

<Class Của Các Phần Tử> <Object Mảng>[<Kích Thước Mảng>];

* Khi này tất cả phần tử của mảng có giá trị ngẫu nhiên
* Ví dụ

int foo[10];

* <Object Mảng> hoạt động khá giống 1 Const Pointer, nhưng Pointer này không có kích thước, địa chỉ của Pointer này = địa chỉ nó chĩa vào = địa chỉ của phần tử đầu tiên
* Dựa vào <Kích Thước Mảng>, <Object Mảng> sẽ được đặt trong bộ nhớ tại ví trí sao cho Byte cuối chạm đít hoặc cách 1 đoạn ngắn ô Byte đầu tiên của Object được khai báo trước đó
* Có thể tạo mảng mà các phần tử là Pointer
* Nếu <Kích Thước Mảng> là tên của 1 biến số nguyên thì vẫn cho phép, khi này mảng được tạo gọi là mảng biến thiên chiều dài, nghĩa là kích thước của nó sẽ được xác định tại thời gian chạy, không phải tại lúc biên dịch, tuy nhiên sử dụng lệnh sizeof trên mảng này sẽ báo lỗi do không biết trước kích thước
* Ví dụ, tạo mảng mà các phần tử là Pointer chĩa vào các địa chỉ chứa các Object thuộc Class int

int \*foo[] = {&bar, &boo};

* Ta có thể chỉ định <Kích Thước Mảng> và khởi tạo giá trị ban đầu cho 1 số phần tử

<Class Của Các Phần Tử> <Object Mảng>[<Kích Thước Mảng>] =

<Các Giá Trị Khởi Tạo>;

* Số phần tử của <Các Giá Trị Khởi Tạo> không được vượt quá

<Kích Thước Mảng>

* Khi này các phần tử còn lại chưa được khởi tạo tự động = 0
* Ví dụ

int foo[5] = {1, 2};

int bar[5] = {};

* Ta có

|  |  |
| --- | --- |
| foo | [1, 2, 0, 0, 0] |
| bar | [0, 0, 0, 0, 0] |

1. Mảng Đa Chiều?

* Dùng nhiều ngoặc nhọn lồng nhau, ví dụ để tạo mảng 3 chiều 2 x 3 x 4

int foo[2][3][4] = {

{

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12}

},

{

{11, 12, 13, 14},

{15, 16, 17, 18},

{19, 110, 111, 112}

}

};

* Truy xuất phần từ có Index (0, 1, 2)

foo[0][1][2]

* Chiều đầu tiên là chiều lớn nhất
* foo[0], foo[1] tương đương mảng 2 chiều
* foo[0][0], foo[0][1], … tương đương mảng 1 chiều
* cho A là mảng N chiều, B là mảng con N – 1 chiều, k là 1 số nguyên, ta có biểu thức

\*(A + k) = A[k] = B

* Ví dụ

\*(foo[0] + 2) = foo[0][2] = {9, 10, 11, 12}

\*foo[1][2] = \*(foo[1][2] + 0) = foo[1][2][0] = 19

* Tương tự như mảng 1 chiều, mỗi hàng hay cột, …, chúng ta có thể không khai báo hết giá trị của các phần tử, chỉ cần khai báo 1 số phần tử

1. Pointer Vào Nguyên Mảng?

* Bây giờ ta muốn khi + 1, thì Pointer sẽ bước 1 bước đi hết cái mảng
* Giả sử foo là mảng 3 chiều 2 x 3 x 4 chứa các Object thuộc Class int, bar là Pointer vào nguyên mảng foo

int (\*bar)[2][3][4] = &foo;

* Quy ước cách hiểu lệnh này như sau
* Thay (\*bar) thành X, ta có

int X[2][3][4]

* Đéo khác gì lệnh khai báo mảng int thông thường, nghĩa là \*bar là 1 mảng int, do đó bar phải là 1 Pointer đến mảng int, để khi đặt dấu \* đằng trước, nó sẽ trả về mảng int, ở đây là foo
* Giả sử bar đang chĩa vào địa chỉ 100, là địa chỉ của foo[0][0][0], thì bar + 5 sẽ chĩa vào địa chỉ 580, do kích thước foo = 2 x 3 x 4 = 24 phần tử, x 4 Byte mỗi phần tử = 96 Byte, x 5 mảng = 480 Byte
* Ví dụ khác

int (\*\*bob)[2][3][4] = &bar;

* bob chính là Pointer đến bar, dễ thấy \*\*bob đúng thật trả về mảng foo là mảng int 2 x 3 x 4

1. Tham Chiếu Vào Nguyên Mảng?

* Tương tự như Pointer vào nguyên mảng, cho mảng A đa chiều, ta muốn dùng tên gọi khác để truy xuất A, chỉ cần đặt & đằng trước tên tham chiếu, rồi đóng ngoặc cả 2 lại
* Ví dụ

int A[2][3][4];

int (&B)[2][3][4] = A;

* Khi này B tương đương A

1. Char Pointer?

* Địa chỉ được lưu trong 1 Char Pointer khi in ra thì sẽ bị chuyển từ dạng thập lục phân sang mã ASCII, do đó để in ra dưới dạng thập lục phân, ta cần ép kiểu Pointer sang kiểu khác, những vẫn chĩa vào Char ban đầu, thông thường dùng Void Pointer

(void\*) &<Char>

* Ví dụ

char foo = 'v';

cout << (void\*) &foo << endl;

* 1 mảng Char hoạt động y chang mảng với Class khác,tuy nhiên khi tạo mảng mà không chỉ định kích thước, thì kích thước tự động = số phần tử + 1, phần tử cuối = \0, đồng thời nếu chỉ định kích thước thì số phần tử phải không vượt quá kích thước – 1, kí tự \0 sẽ tự động thêm kế sau phần tử cuối
* Khi in 1 mảng Char, thì nó không in địa chỉ của phần tử đầu tiên mà in ra chuỗi giá trị của các phần tử dưới dạng ASCII cho đến khi gặp kí tự \0 thì dừng lại, do đó nếu muốn in ra địa chỉ thì cần ép kiểu sang kiểu Pointer khác
* Ngoài cách khởi tạo thông thường, ta còn có thể sử dụng dấu nháy kép
* Ví dụ

char foo[] = "concac";

* Tương đương

char foo[] = {'c', 'o', 'n', 'c', 'a', 'c'};

1. Tạo Object Trong Heap?

* Để chiếm 1 dãy các ô Byte gồm <Số Ô Byte> ô Byte tự do liên tiếp trong Heap, gọi dãy này là Block

void\* <Pointer> = malloc(<Số Ô Byte>);

* <Pointer> được tạo trong Stack, nó trỏ đến ô Byte đầu tiên của Block, hay ô Byte có địa chỉ nhỏ nhất
* Các ô Byte này ban đầu có giá trị ngẫu nhiên
* Sau khi chạy lệnh trên các ô Byte trong Block được đánh dấu là không tự do
* Ta có thể chuyển <Pointer> từ Void Pointer thành Pointer chõ vào Class nào đấy để có thể thay đổi giá trị của các ô Byte này
* Ta có thể thay đổi giá trị của cả các ô Byte bên ngoài Block, nhưng những giá trị đó có thể bị ghi đè bởi các tác vụ khác
* Ví dụ

int\* foo = (int\*) malloc(10);

foo[0] = 11;

foo[12] = 154;

* Tương tự như trên

void\* <Pointer> = calloc(<Số Phần Tử>, <Kích Thước Phần Tử>);

* Tương đương lệnh sau, nhưng khởi tạo các ô Byte với giá trị 0

void\* <Pointer> = malloc(<Số Phần Tử> \* <Kích Thước Phần Tử>);

* Để mở rộng kích thước Block hoặc thu nhỏ lại

void\* <Pointer Mới> = realloc(<Pointer Cũ>, <Số Ô Byte Mới>);

* Nếu <Số Ô Byte Mới> không vượt quá <Số Ô Byte Cũ>, thì đây là thu hẹp Block, <Pointer Mới> = <Pointer Cũ>, một số phần tử cuối của Block sẽ được đánh dấu lại thành tự do
* Nếu <Số Ô Byte Mới> > <Số Ô Byte Cũ>, xét 2 trường hợp
* Nếu vẫn còn không gian trống phía trước, tức là vẫn còn các ô Byte tự do phía trước đủ để mở rộng Block cũ thành Block mới, thì

<Pointer Mới> = <Pointer Cũ>, một số phần tử kế tiếp sẽ bị đánh dấu thành không tự do

* Nếu phía trước không đủ không gian, thì sẽ chọn 1 vùng không gian mới, Copy Block cũ sang, đồng thời giảm thu hẹp kích thước Block cũ về 0, <Pointer Cũ> vẫn chĩa vào Block cũ, <Pointer Mới> thì chĩa vào Block mới
* <Pointer Cũ> có thể là Pointer chõ vào bất kì Class nào, không quan trọng
* Để chủ động thu hẹp kích thước Block về 0

free(<Pointer>);

* Toàn bộ ô Byte trong Block bị <Pointer> chĩa vào sẽ được đánh dấu lại thành tự do
* Sau khi chương trình kết thúc, mọi Block trong Heap do chương trình tạo ra vẫn còn tồn tại thì sẽ bị giảm kích thước về 0
* Ngoài ra ta còn có thể dùng từ khóa new để tạo Object trong Heap

<Class Của Object>\* <Pointer> = new <Class Của Object>;

* ví dụ

int\* foo = new int;

* Tương đương

int\* foo = (int\*) malloc(4);

* Ta có thể khởi tạo giá trị trước

int\* foo = new int(10);

1. Tạo Mảng Trong Heap?

* Ví dụ tạo mảng 1 chiều Class int gồm 3 phần tử, gọi mảng này là A

int\* foo = new int[3];

* Ta có thể khởi tạo giá trị ban đầu như sau

int\* foo = new int[3]{1, 2, 3};

* Khi này foo là 1 Int Pointer trong Stack, nó chĩa vào phần tử đầu tiên của A
* Ví dụ truy cập phần tử có Index 1

foo[1]

* Ví dụ tạo mảng 2 chiều Class int kích thước 2 x 3, gọi mảng này là B

int (\*foo)[3] = new int[2][3]{

{1, 2, 3},

{4, 5, 6}

};

* Khi này foo là 1 Pointer trong Stack, nó chĩa vào mảng int gồm 3 phần tử, nghĩa là nó chĩa vào B[0]
* Ta có B[0] = foo[0], B[1] = foo[1], B[0][0] = foo[0][0], B[1][2] = foo[1][2], …
* Nếu ta chưa biết được kích thước mảng lúc biên dịch, thì có thể tạo mảng 2D bằng cách tạo 1 mảng gồm các Pointer đến mảng
* Ví dụ tạo mảng 2 chiều Class int gồm 3 hàng, số phần tử mỗi hàng có thể khác nhau

int \*\* foo = new int\*[3]{

new int[4]{4, 5, 6, 7},

new int[2]{1, 2},

new int[5]

};

* Cách truy cập tương tự như khi kích thước mảng đã xác định trước
* Tương tự, ví dụ tạo mảng 3 chiều

int (\*foo)[3][4] = new int[2][3][4];

1. Xóa Mảng Trong Heap?

* Đối với Object đơn lẻ, thì phải dùng lệnh sau, nếu không sẽ văng, tương đương lệnh free

delete <Pointer Đến Object>;

* Đối với mảng 1 chiều, thì phải dùng lệnh sau, nếu không sẽ văng, tương đương lệnh free

delete[] <Pointer Đến Mảng>;

* Đối với mảng 2 chiều trở lên, thì có thể dùng 1 trong 2 lệnh trên
* Sau khi xóa Pointer, bạn cần gán Null Pointer cho nó, nếu muốn delete 2 lần mà không bị lỗi

1. Function Pointer?

* Là Pointer chĩa vào Instruction đầu tiên của hàm trong phần nhớ Text
* Giả sử cho hàm foo trả về int, nhận 2 tham số đầu vào là float và double, ta muốn tạo Pointer bar chĩa vào foo

int (\*bar)(float, double) = foo;

* Lệnh trên được hiểu như sau
* Thế (\*bar) thành X, ta được

int X(float, double)

* Dễ thấy đây đéo khác gì hàm nguyên mẫu, do đó X là hàm giống foo, nói cách khác, \*bar là hàm foo, do đó bar phải là Pointer đến foo, để khi đặt dấu \* đằng trước sẽ trả về foo, vế phải là foo bởi vì tên hàm = Pointer đến hàm
* Do bar là Pointer đến hàm, nên ta có thể dùng nó để gọi hàm

bar(3.5, 7.9);

* Để in ra giá trị của Function Pointer, nghĩa là in ra địa chỉ của nó chõ vào

cout << (void\*) <Function Pointer>;

1. Handle?

* Chỉ là 1 Pointer đến 1 tài nguyên nào đó, ví dụ File “foo” được tải từ đĩa cứng vào RAM, đồng thời ta tạo 1 Pointer đến File này, thì nó là 1 Handle
* 1 tài nguyên ví dụ như 1 File khi đang mở trong RAM, có thể được nhiều Handle chĩa tới, khi không còn Handle nào chĩa tới File này nữa, thì Block của nó trong RAM sẽ được tự do

Special Value – Giá Trị Đặc Biệt:

1. Các Hằng Toàn Cục Mặc Định?

CHAR\_BIT = 8

1. Pointer Đặc Biệt?

nullptr

* Là Pointer chõ đến không gì cả

Enum:

1. Tại Sao Dùng Enum?

* Giả sử ta muốn tạo 1 Object mà nó chỉ lưu tên các diễn viên nổi tiếng, nhưng địt mẹ làm như vậy thì tốn bộ nhớ vãi, thay vào đó ánh xạ tên mỗi diễn viên vào 1 số nguyên, điều này được thực hiện = Enum
* Để tạo 1 Enum Class

enum <Enum Class> {<Tên 1>, <Tên 2>, … };

* Để tạo 1 Object chỉ được phép có giá trị trong <Enum Class>

<Enum Class> <Object>;

* Ta chỉ có thể gán cho <Object> các giá trị là <Tên 1>, <Tên 2>, …, phải viết đúng tên
* Bản chất các giá trị <Tên 1>, <Tên 2>, … là hằng số nên ta có thể dùng chung bất kì đâu
* Ví dụ

enum foo {bar, bob, alice};

foo far = bob;

int john = bar;

* Các giá trị đằng sau

|  |  |
| --- | --- |
| bar | 0 |
| bob | 1 |
| alice | 2 |
| far | 1 |

* Các giá trị đằng sau là số nguyên bắt đầu từ 0, ta cũng thay đổi những giá trị này = cách gán cho <Tên 1>, <Tên 2>, …, các giá trị nguyên, khi này các giá trị đằng sau sẽ bắt đầu đếm lại từ chỗ gán
* Ví dụ

enum foo {bar, bob, alice = 5, a, b, c = -2, d, e};

* Ta có

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| bar | bob | alice | a | b | c | d | e |
| 0 | 1 | 5 | 6 | 7 | –2 | –1 | 0 |

* Bạn không ghi <Enum Class> cũng được
* Ví dụ

enum {ga, foo};

* Tương đương

const int ga = 0;

const int foo = 1;

* Cho dù là 2 Enum Class khác nhau thì cũng không được phép có phần tử với giống nhau, ví dụ sau sẽ lỗi do ALICE xuất hiện 2 lần

enum foo {BAR, ALICE};

enum john {ALICE, DARK};

Operator – Toán Tử:

1. Toán Tử Gán?

* =, +=, …
* Nguyên biểu thức ở vế phải sẽ được tính trước, sau đó mới gán cho vế trái, đồng thời câu lệnh gán cũng sẽ trả về giá trị này
* Ví dụ

int a = 4;

b = (a += 5) \* 6

* Ta có

|  |  |
| --- | --- |
| a | 9 |
| b | 54 |

1. Kí Hiệu Khoa Học?

<Số Bất Kì>e<Mũ>

* <Mũ> phải là số nguyên
* Ví dụ

123.456e05

12.456e+4

1. Trả Về Giá Trị Thập Phân Từ Thập Lục Phân?

0x<Thập Lục Phân>

* Ví dụ

int foo = 0xABC;

* Ta có

|  |  |
| --- | --- |
| foo | 2748 |

1. Trả Về Giá Trị Thập Phân Từ Bát Phân?

0<Bát Phân>

* Ví dụ

int a = 00014, b = 014;

* Ta có

|  |  |
| --- | --- |
| a | 12 |

1. Toán Tử Dịch Bit?

* Để dịch sang trái

<Biến Đã Dịch> = <Object> << <Dịch Sang Trái Bao Nhiêu Đơn Vị>

* <Object> và <Dịch Sang Trái Bao Nhiêu Đơn Vị> phải là số nguyên
* Ví dụ

foo = bar << 5

* Ta có

|  |  |
| --- | --- |
|  | Dạng nhị phân |
| bar | 101 |
| foo | 10100000 |

* Để dịch sang phải

<Object Đã Dịch> = <Object> >> <Dịch Sang Phải Bao Nhiêu Đơn Vị>

* Ví dụ

foo = bar >> 3

* Ta có

|  |  |
| --- | --- |
|  | Dạng nhị phân |
| bar | 10111011 |
| foo | 10111 |

1. Cộng Thêm 1 Và Trả Về Giá Trị Trước Đó?

<Object Trước Đó> = <Object Sắp Cộng>++;

* Không được phép ghi 5++, 3++, …
* Ví dụ

foo = bar++;

* Ta có

|  |  |
| --- | --- |
| bar trước | 5 |
| foo | 5 |
| bar sau | 6 |

1. Cộng Thêm 1 Và Trả Về Giá Trị Sau Khi Đã Cộng?

<Kết Quả> = ++<Object Sắp Cộng>;

* Ví dụ

foo = ++bar;

* Ta có

|  |  |
| --- | --- |
| bar trước | 5 |
| foo | 6 |
| bar sau | 6 |

1. Phép Chia Nguyên?

<Thương> = <Số Bị Chia> / <Số Chia>;

* <Số Bị Chia> và <Số Chia> phải nguyên
* <Thương> sẽ là phần nguyên của phép chia
* Ví dụ

foo = bar / boo;

* Ta có

|  |  |
| --- | --- |
| bar | –8 |
| boo | 3 |
| foo | –2 |

1. Phép Chia Thập phân?

<Thương> = <Số Bị Chia> / <Số Chia>;

* <Số Bị Chia> hoặc <Số Chia> hoặc cả 2 phải là số thập phân
* <Thương> sẽ là số thập phân, = kết quả của phép chia
* Ví dụ

foo = bar / boo;

* Ta có

|  |  |
| --- | --- |
| bar | 8. |
| boo | 3 |
| foo | 2.66667 |

1. Phép Lấy Dư?

<Dư> = <Số Bị Chia> % <Số Chia>;

* <Số Bị Chia> và <Số Chia> phải nguyên
* Bản chất, ban đầu <Số Bị Chia> sẽ / <Số Chia>, kết quả giống phép chia nguyên, được <Thương>, sau đó ta có công thức

<Dư> = <Số Bị Chia> – <Số Chia> \* <Thương>

* Ví dụ

foo = bar % boo;

* Ta có

|  |  |
| --- | --- |
| bar | –13 |
| boo | 5 |
| foo | –3 |

1. Toán Tử And?

<Z> = <X> && <Y>;

* Nó sẽ kiểm tra <X> trước, nếu giá trị <X> quy đổi về 0 thì <Z> = 0 = False, <Y> bị bỏ qua không được thực hiện, nếu <X> quy đổi về 1 thì kiểm tiếp <Y>, …, cho đến khi không còn giá trị nào để kiểm nữa mà tất cả đều quy đổi khác 0 thì

<Z> = 1

1. Toán Tử So Sánh?

* ==, >, <, >=, …
* Trả về 1 = True hoặc 0 = False
* Cẩn thận khi so sánh số Floating Point
* Ví dụ

float a = 0.7;

bool b = a == 0.7;

* Ta có b = False, điều này là do a là float còn 0.7 bên phải là double, nên độ chính xác của chúng khác nhau, do đó giá trị khác nhau

1. Toán Tử Bitwise And?

<Z> = <X> & <Y>;

* <X> và <Y> phải có giá trị nguyên
* <Z> được tính = cách chuyển <X> và <Y> thành dạng nhị phân rồi And với nhau từng Bit 1, sau đó chuyển lại về nguyên

1. Toán Tử Bitwise Not?

<X’> = ~<X>;

* <X> phải có giá trị nguyên
* Lấy toàn bộ Bit trong ô nhớ rồi đảo lại, sau đó chuyển lại về nguyên

1. Toán Tử Bitwise EX – OR?

<Z> = <X> ^ <Y>;

* Tương tự Bitwise AND, chỉ đổi AND thành EX – OR

1. Toán Tử 3 Ngôi (Ternary)?

<Kết quả> = <Điều Kiện> ? <True> : <False>;

* Ví dụ

foo = 2 > 5 ? 6 : 7;

* Ta có

|  |  |
| --- | --- |
| foo | 7 |

1. Dấu Phẩy?

* Để thực hiện 1 lúc nhiều câu lệnh gán trên 1 dòng Code, dùng dấu phẩy
* Ví dụ

foo = 5, bar = 4, far, boo;

* Cũng như trên, nhưng trả về giá trị của biểu thức cuối cùng

foo = (bar = 5, boo, bar += 2);

* Ta có

|  |  |
| --- | --- |
| foo | 7 |

* Không được phép sử dụng 2 lần từ khóa khai báo Class, ví dụ như sau là sai

int a = 2, int b = 1;

* Để khai báo nhiều Object thuộc cùng 1 Class, ví dụ

int a = 4, b, c = 8;

* Tuy nhiên đối với Pointer, thì phần kẻ từ dấu \* chỉ dành cho Object tương ứng
* Ví dụ

const int \* const a = &c, b = 6;

* Tương đương

const int \* const a = &c;

const int b = 6;

* Tương tự đối với mảng, thì phần kể dấu [ chỉ dành cho Object tương ứng
* Ví dụ

const int foo[5] = {1, 2, 3, 4, 5}, bar = 6;

* Tương đương

const int foo[5] = {1, 2, 3, 4, 5};

const int bar = 6;

1. Độ Ưu Tiên Các Toán Tử?

|  |  |
| --- | --- |
| Độ ưu tiên | Toán tử |
| 1 | A::B, ::A |
| 2 | A++, A--, A[], A.B, A->B |
| 3 | ++A, --A, !A, \*A, &A ~A, (<Ép Kiểu>) |
| 4 | \*, /, % |
| 5 | +, - |
| 6 | >>, << |
| 7 | <, <=, >, >= |
| 8 | ==, != |
| 9 | & |
| 10 | ^ |
| 11 | | |
| 12 | && |
| 13 | || |
| 14 | Toán tử 3 ngôi |
| 15 | Các toán tử gán |

* Cùng độ ưu tiên thì tính từ trái sang phải
* Để dễ dàng tính toán, ta ngoặc lần lượt các cụm có độ ưu tiên giống nhau rồi tính từng cụm 1 từ trái sang phải
* Ví dụ

i = 1;

t = ++i || j++ && ++k = (++i) || (j++ && ++k)

* Ở đây, (++i) trả về 2, quy đổi về True, do đó vế phải toán tử Or không được thực hiện, hay biến j và k không được + 1

1. Nạp Chồng Toán Tử (Operator Overloading)?

* Bản chất của 1 toán tử là 1 hàm, hàm này có thể đã được nạp chồng nhiều lần để toán tử có thể áp dụng trên nhiều Class khác nhau
* Ta chỉ có nạp chồng các toán tử mà C++ cho phép
* Cú pháp tương tự như khi định nghĩa hàm
* Thứ tự tham số = thứ tự toán hạng
* Số toán hạng phải = số toạn hạng của toán tử khi chưa nạp chồng, ví dụ toán tử + cần 2 toán hạng, thì khi thực hiện nạp chồng +, thì nó cũng phải áp dụng lên 2 toán hạng hay 2 tham số
* Khi nạp chồng, độ ưu tiên toán tử không đổi

<Class Phép Toán Trả Về> operator <Kí Hiệu Toán Tử> (<Các Toán Hạng>){

<Làm Gì Đó Và Trả Về Giá Trị>

}

* Ví dụ
* Ta tạo 1 Class foo mới và thực hiện phép cộng giữa 2 đối tượng thuộc Class này

class foo {

public:

int a;

int b;

};

int operator + (foo haha, foo huhu){

return haha.a \* huhu.b;

}

foo bar = {4, 5}, bob = {7, 8};

cout << bar + bob;

* Buffer

32

* Ta có thể thực hiện việc nạp chồng bên trong 1 Class, khi này không cần phải chỉ định tham số đầu tiên, nó sẽ tự động là Object gọi đến toán tử này, nói cách khác Object này là toán hạng đầu tiên, và từ khóa this trong đây sẽ chỉ vào nó
* Ví dụ

class foo {

public:

int a;

int b;

int operator + (foo huhu){

return a \* huhu.b;

}

};

foo bar = {4, 5}, bob = {7, 8};

cout << bar + bob;

* Buffer

32

Loop – Vòng Lặp:

1. For Loop?

for (<Lệnh Đầu>; <Điều Kiện>; <Lệnh Cuối>){

<Làm Gì Đó>

}

* <Lệnh Đầu> sẽ được chạy 1 lần duy nhất trước khi bắt đầu vòng lặp, nếu khai báo biến trong này thì sẽ như biến cục bộ trong hàm, nhưng vẫn có thể sử dụng lại biến đã khai báo ở bên ngoài và cập nhật nó
* Mỗi lần lặp, sẽ kiểm tra <Điều Kiện> trước, nếu nó trả về True, thì <Làm Gì Đó> được chạy và sau đó <Lệnh Cuối> sẽ chạy, còn nếu trả về False, thì kết thúc vòng lặp
* Ví dụ

for (int i = 0; i < 5; i++){

bar = bar + 2;

}

1. Kết Thúc 1 Vòng Lặp Ngay?

break;

* Nếu nhiều vòng lặp chồng lên nhau thì chỉ kết thúc vòng lặp trong cùng

1. Kết Thúc Lần Lặp Hiện Tại Ngay Không Chạy Code Phía Dưới Nữa Mà Qua Kiểm Tra Điều Kiện Lần Lặp Tiếp Theo?

continue;

1. While Loop?

while (<Điều Kiện>){

<Làm Gì Đó>

}

* Khi bắt gặp lệnh này, đầu tiên Check <Điều Kiện>, nếu True thì <Làm Gì Đó> thực thi, rồi quay trở lại, nếu False thì cút

1. Do While Loop?

do {

<Làm Gì Đó>

} while (<Điều Kiện>);

* Khi gặp lệnh này, <Làm Gì Đó> được thực thi trước, rồi mới Check <Điều Kiện>, nếu True, quay lại, False thì cút

1. 1 Số Vòng Lặp?

* Đếm a, a + 1, a + 2, …, n

for (int i = a; i <= n;, i++){

cout << i;

}

1. Range Based For Loop?

* Là cách viết ngắn gọn khi bạn muốn lặp qua các phần tử của 1 mảng, Vector, …

for (<Class Của Các Phần Tử> <Object Lặp> : <Mảng, Vector, …>){

<Làm gì đó>

}

* <Object Lặp> qua mỗi vòng lặp sẽ mang giá trị của từng phần tử trong

<Mảng, Vector, …>

* Ví dụ

for (int foo : {9, 8, 7, 6}){

cout << foo << endl;

}

* Buffer

9

8

7

6

Condition – Điều Kiện:

1. If Else?

* Trường hợp 1, không có Else

if (<Biểu Thức>) <Lệnh>

* Nếu <Biểu Thức> trả về True thì <Lệnh> được thực hiện, còn nếu trả về False thì <Lệnh> không được thực hiện
* Trường hợp 2, có Else

if (<Biểu Thức>) <Lệnh 1> else <Lệnh 2>

* Nếu <Biểu Thức> trả về True thì <Lệnh 1> được thực hiện, còn nếu trả về False thì <Lệnh 2> được thực hiện
* Else sẽ đi kèm với If ngay trước nó, nếu lệnh If ngay trước đó đã có Else rồi thì trước đó nữa, …
* Ví dụ khi ta viết

if (foo > 5) if (foo < 4) cout << 4; else cout << 5; else cout << 6;

* Thì tương đương

if (foo > 5){

if (foo < 4) cout << 4;

else cout << 5;

}

else cout << 6;

* Ví dụ khi ta viết

if (foo == 1) cout << 4;

else if (foo == 2) cout << 5;

else if (foo == 3) cout << 6;

else cout << 7;

* Thì tương đương

if (foo == 1) cout << 4;

else {

if (foo == 2) cout << 5;

else {

if (foo == 3) cout << 6;

else cout << 7;

}

}

* Nghĩa là lệnh trong trường hợp Else If, <Lệnh 2> là 1 lệnh If

1. Switch Case?

switch (<Giá Trị>){

case <Giá Trị 1>: <Dãy Các Câu Lệnh 1>

case <Giá Trị 2>: <Dãy Các Câu Lệnh 2>

…

default: <Dãy Các Lệnh Mặc Định>

case <Giá Trị N>: <Dãy Các Câu Lệnh N>

…

}

* <Giá Trị> sẽ lần lượt được so sánh với <Giá Trị 1>, <Giá Trị 2>, … cho đến khi nào nó tìm được giá trị đúng = nó hoặc gặp “default”, tại đó, tất cả các

<Dãy Các Câu Lệnh> trở đi đều được thực hiện

* Các <Giá Trị 1>, <Giá Trị 2>, … phải khác nhau, là hằng số, cùng Class là số nguyên với <Giá Trị>
* Ví dụ

int foo = 3;

switch (foo){

case 1: cout << 1;

case 2: cout << 2;

case 3: cout << 3;

cout << 4;

case 10: cout << 5;

default: cout << "Haha";

}

* Màn hình

|  |
| --- |
| 345Haha |

* Để không thực hiện tất cả <Dãy Các Câu Lệnh> thì thêm lệnh sau vào vị trí bạn muốn thoát khỏi khối Switch Case

|  |
| --- |
| break; |

Function – Hàm:

1. Cách Tạo 1 Hàm?

<Class Trả Về> <Tên Hàm>(<Các Tham Số>){

<Làm Gì Đó>

}

* Ví dụ

int foo(){

return 1;

}

* Nếu không có lệnh return, thì giá trị trả về ngẫu nhiên

1. Có Bao Nhiêu Cách Để Pass Tham Số?

* 2 cách, Pass = giá trị và Pass = tham chiếu
* Ví dụ Pass = giá trị

void foo(int a){

a = 5;

}

int c = 10;

foo(c);

* Khi viết như trên, Object a được tạo ra trong RAM, giá trị của c được Copy vào a, nên khi a thay đổi, c không thay đổi
* Tương đương với việc viết

int a = c;

* Ví dụ Pass = tham chiếu

void foo(int& a){

a = 5;

}

int c = 10;

foo(c);

* Khi viết như trên, a coi như tên gọi khác của c, nên khi a thay đổi, c ở ngoài cũng thay đổi
* Tương đương với việc viết

int& a = c;

* Khi Pass 1 mảng vào 1 hàm, bản chất là ta đang Pass Pointer vào phần tử đầu của mảng vào hàm
* Ví dụ

void foo(int\* a){

a[1] = 5;

}

int c[4] = {9, 8, 7, 6};

foo(c);

* Giá trị của c sau khi chạy đoạn Code trên sẽ là {9, 5, 7, 6}
* Tương đương với việc viết

int\* a = c;

* Ngoài ra thay vì viết

void foo(int\* a)

* Ta cũng có thể viết

void foo(int a[])

* Hoặc

void foo(int a[<Kích Thước Bất Kì>])

* Kết quả là y chang đéo khác gì
* Khi Pass như những cách trên, thì thông tin về số Byte của mảng bị xóa, do đã chuyển về Pointer, nên khi Pass mảng vào hàm, thì cần Pass thêm kích thước của nó
* Cách khác hay hơn, đó là sử dụng Pointer đến nguyên mảng
* Ví dụ

void foo(int (\*a)[11]){

int \* b = \*a;

int bar = sizeof(\*a);

}

int c[11] = {9, 8, 7, 6};

foo(&c);

* Khi này b chính là mảng c, và bar chính là số Byte mà c chiếm, tức là = 11 \* 4 = 44 Byte
* Cách hay hơn nữa là Pass tham chiếu
* Ví dụ

void foo(int (&a)[11]){

a[0] = 99;

int bar = sizeof(a);

}

int c[11] = {9, 8, 7, 6};

foo(c);

* Khi này a chính là c, nên thay đổi a trong foo thì c cũng đổi, bar chính là số Byte c chiếm
* Tên tham số có thể có hoặc không, khi không có tên tham số, thì bạn sẽ không thể sử dụng chúng trong hàm, nhưng vẫn phải Pass
* Ví dụ

int foo(int, float){

return 1;

}

foo(4, 6.8);

* Bạn có thể chỉ định giá trị mặc định cho tham số, khi này mặc dù không có phần hiện thực hàm thì vẫn tính là bạn đã hiện thực hàm

1. Hàm Nguyên Mẫu (Function Prototype)?

* Cho trình biên dịch biết ta sẽ khai báo hàm này sau, nó sẽ không báo lỗi khi ta sử dụng hàm mà đéo khai báo nó trước, dùng cái này khi chia dự án ra nhiều File C++, rồi liên kết với nhau qua trình liên kết, File này sử dụng hàm của File kia thông qua hàm nguyên mẫu

<Class Trả Về> <Tên Hàm>(<Các Class Của Tham Số>);

* Ví dụ

int foo(int, float);

int main(){

cout << foo(2, 4);

}

int foo(int a, float b){

return a + b;

}

* Hàm nguyên mẫu có tác dụng chủ yếu khi bạn có 2 hàm mà nó gọi lẫn nhau, nếu hàm A gọi hàm B, nhưng do B chưa khai báo nên trình biên dịch báo lỗi, hoặc hàm B đã khai báo trước, nhưng trong B lại gọi A, mà A khai báo sau, thế cũng báo lỗi, để ngăn chặn điều, này, dùng hàm nguyên mẫu
* Ví dụ

void bar(); // hàm nguyên mẫu

void foo(){

cout << 1;

bar(); // nếu đéo có hàm nguyên mẫu khúc này sẽ báo lỗi

}

void bar(){

cout << 2;

foo(); // gọi ngược lại foo tạo thành đệ quy

}

int main(){

foo();

}

* Chương trình trên sẽ in ra liên tục “1212121212…” đéo ngừng, kiểu như Flip Flop
* Ở hàm nguyên mẫu, tên tham số có thể có hoặc không, thậm chí khác tên khi thực hiện hàm sau đó, vì trình biên dịch đéo quan tâm
* Có thê chỉ định giá trị mặc định cho tham số ở hàm nguyên mẫu
* Ví dụ

int foo(int, float b = 4.8, int c = 2);

* Khi này ở phần thực hiện hàm, ta không được phép chỉ định giá trị mặc định lại cho tham số đã chỉ định trước ở hàm nguyễn mẫu
* Tiếp ví dụ ở trên

int foo(int a, float b, int c){

cout << a << endl << b << endl << c;

}

foo(6);

* Buffer

6

4.8

2

* Khi sử dụng giá trị mặc định cho tham số, và ta lại nạp chồng hàm, có thể gặp tình huống sau

int foo(int a, int b, int c = 4){

cout << 1;

}

int foo(int a, int b){

cout << 2;

}

Foo(4, 5);

* Tại đây, trình biên dịch sẽ không biết ta gọi tới hàm foo ở trên hay ở dưới, và nó sẽ báo lỗi, nghĩa là bất cứ khi nào có từ 2 phiên bản nạp chồng tương thích với các tham số truyền vào thì sẽ báo lỗi

1. Scope?

* Một Object sẽ có 4 kiểu là mặc định, tĩnh, Register, Extern
* Object kiểu mặc định hoạt động theo cơ chế bộ nhớ của C++, tức là tạo trong Stack, và biến mất khi khung hàm chứa nó biến mất
* Object tĩnh y chang mặc định, chỉ có điều khi {} chứa nó kết thúc, nó sẽ không biến mất khỏi RAM vì được lưu trong phần vùng nhớ khởi tạo, và khi khối {} chứa nó được thực thi lại, thì sẽ không tạo ra Object mới mà dùng Object cũ, nghĩa là ở lần thực thi thứ 2, câu lệnh khai báo Object tĩnh này sẽ được bỏ qua, đồng thời nếu không khởi tạo giá trị ban đầu, nó tự động = 0
* Để tạo 1 Object tĩnh

static <Class Của Object> <Object> = <Giá Trị Khởi Tạo>;

* Ví dụ

static int foo = 5;

* Object kiểu Register y chang mặc định, chỉ có điều nó không lưu trong RAM và lưu trong thanh ghi ở CPU, và biến mất khỏi thanh ghi khi khối {} chứa nó kết thúc, do lưu sẵn trong thanh ghi nên khi tính toán, nó không cần tải từ RAM vào thanh ghi nữa
* Để tạo 1 Object kiểu Register

register <Class Của Object> <Object> = <Giá Trị Khởi Tạo>;

* Tuy nhiên, Object kiểu Register đã bị khai tử
* Object kiểu Extern y chang mặc định, chỉ có điều nó trở thành Object toàn cục trong vòng đời của khối {} chứa nó
* Để tạo 1 Object kiểu Extern

extern <Class Của Object> <Object>;

1. Hàm Tĩnh?

* Xét 1 hàm nằm tại cấp cao nhất của File, tức là không nằm trong Class hay hàm nào, nằm ngoài hàm main, khi này nếu không chỉ định gì thì mặc định đây là hàm toàn cục, khi ta thực hiện liên kết các File C++ Object, thì giả sử ta không sử dụng hàm này chỗ nào, đồng thời trong hàm này ta lại sử dụng 1 hàm A của File C++ Object khác, thông qua hàm nguyên mẫu, nhưng ta đã xóa hàm đó khỏi File khác rồi, thì khi liên kết sẽ xảy ra lỗi, do trình liên kết cho rằng đây là hàm toàn cục, nên nó có thể được sử dụng bởi các File C++ Object khác được liên kết với File chứa nó, mặc dù thực tế là đéo, và nó thực hiện liên kết những gì ở trong hàm này, và nó phát hiện A, nhưng A đéo tồn tại, nên báo lỗi
* Để tránh trường hợp trên, thêm từ khóa static vào trước khai báo hàm, khi này hàm sẽ chỉ có phạm vi nhìn thấy trong File chứa nó, nghĩa là các File C++ Object khác không thể sử dụng được nó, và do đó, trình liên kết bỏ qua hàm này do nó đéo được sử dụng, nên đéo báo lỗi

1. Cách Gọi Hàm?

* Hàm chỉ có thể được gọi thông qua Pointer, Pointer này phải là Function Pointer và chĩa vào Instruction đầu tiên trong dãy các Instruction ứng với hàm muốn gọi
* Chính tên hàm cũng là 1 Pointer như trên
* Cú pháp gọi hàm

<Pointer>(<Các Tham Số>);

1. Truyền Tham Số Cho Chương Trình Từ CMD?

* Bước 1, mở CMD, nhập lệnh sau

<Đường Dẫn Đến File EXE Không Có Phần Mở Rộng> <Các Tham Số>

* <Các Tham Số> đều là String, nếu đặt trong dấu ngoặc kép thì tính là 1 tham số, nếu không thì mặc định ngăn cách nhau bởi 1 chuỗi khoảng trắng
* <Đường Dẫn Đến File EXE Không Có Phần Mở Rộng> cũng tính là tham số, và nó là phần tử đầu tiên của mảng tham số
* Ví dụ

bar\foo "thang lon" 123 abc vai cut "ao456"

* Thì File “foo.exe” sẽ được truyền tham số lần lượt là "bar\foo", "thang lon", "123", "abc", "vai", "cut", "ao456"
* Tham số sẽ được truyền vào hàm main, để nhận chúng, dùng cú pháp

int main(int <Số Tham Số>, char\*\* <Pointer Đến String>){

<Làm Gì Đó>

}

* Tiếp tục ví dụ trên, thì khi Pass các tham số vào, <Số Tham Số> sẽ nhận giá trị = số phần tử mảng tham số = 7
* Để trả về Pointer đến 1 tham số khi này đã là 1 mảng Char hay 1 String trong bộ nhớ

<Pointer Đến String>[<Index>]

1. Hàm Inline?

* Ví dụ ta có 1 hàm foo, là 1 hàm cực đơn giản, trả về bình phương của 1 số, giả sử trong chương trình ta gọi hàm này liên tiếp tỉ lần, thì mỗi lần sẽ phải tốn công đẩy tham số vào Stack, chuyển con trỏ Instruction tới Instruction đầu tiên của hàm, chạy hàm, lưu giá trị trả về, …, để đỡ phải chuyển qua chuyển lại, ta dùng hàm Inline, bằng cách đặt từ khóa inline trước khai báo hàm
* Ví dụ

inline void foo(int& b){

b\*=b;

}

* Khi này, sau khi biên dịch xong xuôi, bất cứ chỗ nào hàm foo được gọi đều sẽ thế = nguyên đoạn mã của foo, nên không cần phải mất công gọi hàm mà chạy thẳng theo Flow của chương trình luôn
* Tuy nhiên, từ khóa inline sẽ vô dụng với các hàm phức tạp, như bên trong chứa vòng lặp, đệ quy, Object tĩnh, …, nghĩa là trình biên dịch sẽ bỏ qua từ khóa này

1. Hàm Con (Nested Function)?

* Không thể định nghĩa 1 hàm bên trong 1 hàm
* Chỉ có thể sử dụng hàm nguyên mẫu của A bên trong hàm B để sử dụng A trong B khi nó chưa được định nghĩa
* Ví dụ

void foo(){

void bar();

bar();

}

void bar(){

cout << 1;

}

foo();

* Buffer

1

* Tương tự, bạn không thể định nghĩa 1 Class bên trong 1 hàm

1. Hàm Trả Về Tham Chiếu?

* Thông thường ta thấy các hàm giả về giá trị toàn là R Value, để cho nó trả về L Value, nghĩa là tham chiếu đến 1 Object nào đó, thì chỉ định kiểu trả về của hàm là kiểu tham chiếu, và giá trị trả về phải là tên 1 Object nào đấy
* Ví dụ

int c;

int& foo(){

return c;

}

* Khi này ta có thể viết

foo() = 9;

* Và khi in ra c, ta được 9, đó là do foo() trả về L Value nên ta có thể gán giá trị cho nó
* Ta cũng có thể gán lại tham chiếu này cho biến khác

int& bar = foo();

bar = 10;

* Khi này c sẽ bằng 10

Class:

1. Cách Hoạt Động?

* Quy định các kiểu dữ liệu như int, float, … là Class int, Class float, …, ta tạm gọi chúng là kiểu dữ liệu cơ bản, Struct cũng xem là Class luôn
* Các hàm trong Class ta gọi là phương thức, ngoài Class thì gọi là hàm bình thường
* Các Object trong Class ta gọi là thuộc tính, ngoài Class thì gọi là Object bình thường
* Khi 1 Class được tạo ra, nó sẽ lưu ở phần nhớ Text, gọi là bản mẫu
* Khi 1 Object được tạo ra từ Class, nó sẽ dựa vào bản mẫu để tạo ra các thuộc tính riêng cho Object này, cơ chế tạo thuộc tính y chang như khi tạo biến bên ngoài Class, nghĩa là các thuộc tính này sẽ lưu trong Stack, các Object dùng chung hàm trong bản mẫu
* Bản chất Object không tồn tại, khi bạn dùng tên Object để truy xuất thuộc tính hoặc hàm, thực chất tương đương với việc bạn dùng tên biến để truy xuất giá trị, hoặc dùng tên hàm để gọi hàm, điều này được thực hiện bởi trình biên dịch
* Các chế độ của thuộc tính như công khai, bảo vệ, ẩn, chỉ có tác dụng khi biên dịch
* Thuộc tính hoặc phương thức ở chế độ công khai, bạn có thể truy cập nó từ bất kì đâu
* Thuộc tính hoặc phương thức ở chế độ ẩn, bạn chỉ có thể truy cập nó bên trong phần định nghĩa Class, bên trong hàm bạn hoặc Class bạn, không thể truy cập trong Class kế thừa
* Thuộc tính hoặc phương thức ở chế độ bảo vệ, y chang khi ở chế độ ẩn, nhưng có thể truy cập trong Class kế thừa

1. Tạo 1 Class?

class <Tên Class> {  
 <Khai Báo Mặc Định>

public:

<Tên Class>(<Các Tham Số Khởi Tạo>){

<Đây Là Constructor>

<Lệnh Trong Này Sẽ Thực Thi Ngay Khi Object Được Tạo>

}

~<Tên Class>(){

<Đây Là Destructor>

<Lệnh Trong Này Sẽ Thực Thi Khi Object Bị Xóa>

}

<Khai Báo Gì Đó Công Khai>

private:

<Khai Báo Gì Đó Ẩn>

protected:

<Khai Báo Gì Đó Được Bảo Vệ>

};

* Bạn có thể nạp chồng nhiều Constructor với các thể loại tham số khác nhau, một khi bạn đã viết 1 Constructor với số lượng tham số > 0, thì bạn bắt buộc phải tạo Constructor mặc định với 0 tham số
* Destructor của Object sẽ được gọi khi Object bị xóa, đó là khi khung hàm chứa Object được thu hồi, hoặc khi bạn dùng lệnh delete lên Object, không được gọi khi dùng lệnh free, do free không quan tâm đến Pointer Pass vào nó chõ về Object thuộc Class gì
* Nếu bạn dùng lệnh delete mà Object lại đéo có Destructor thì sẽ Crash
* Dễ thấy Constructor và Destructor không trả về thứ gì
* Ngay khi tạo mảng Object thuộc Class A, thì mỗi Object sẽ gọi Constructor của riêng nó, Constructor này là Constructor không nhận đối số nào trong dãy các Constructor nạp chồng
* Tạo 1 Object của 1 Class

<Tên Class> <Object>(<Các Tham Số Khởi Tạo>);

* Trong trường hợp không có <Các Tham Số Khởi Tạo> thì phải viết như sau

<Tên Class> <Object>;

* Ví dụ

car Honda;

* Các thuộc tính ở phần <Khai Báo Mặc Định> sẽ tự động ẩn
* Toàn bộ thuộc tính được khai báo trong Class sẽ được lưu trong Stack trong các theo đúng thứ tự khai báo, vẫn có Padding, ví dụ khai báo a, b, c thuộc kiểu int thì 4 ô Byte đầu dành cho a, 4 ô Byte kế dành cho b, 4 ô Byte cuối dành cho c, theo chiều tăng dần địa chỉ, tiếp tục khai báo e, f thuộc kiểu double, thì 4 ô Byte sau c dành cho việc Padding, 8 ô Byte sau đó nữa lưu e, 8 Byte kế lưu f, tiếp tục khai báo g thuộc kiểu char, thì 1 ô Byte tiếp lưu g, và 7 ô Byte kế làm Padding, lí do phần cuối của Class phải Padding là để địa chỉ của ô Byte tiếp theo sau 7 ô Byte này chia hết cho kích thước của thuộc tính có kích thước lớn nhất, ở đây double lớn nhất = 8, nói cách khác, nó tương đương với việc Clas này chịu ảnh hưởng của chỉ thị #pragma pack(<Kích Thước Thuộc Tính Lớn Nhất>), lưu ý những điều trên áp dụng cho tất cả thuộc tính, không quan tâm ẩn hay công khai hay bảo vệ
* Địa chỉ của <Object> là địa chỉ của thuộc tính đầu tiên của nó trong Stack
* Để trả về kích thước của Class, tức là tổng số Byte mà các thuộc tính trong Class chiếm trong Stack, bao gồm cả các Padding

sizeof(<Tên Class Hoặc 1 Object Của Class>)

* Để trả về kích thước của Class do 1 Pointer chĩa vào

sizeof(\*<Pointer>)

* Ta có thể khởi tạo giá trị ban đầu của các thuộc tính công khai ngay khi khởi tạo Object bằng cú pháp sau, thứ tự gán = thứ tự khai báo

<Tên Class><Object> = {<Giá Trị Thuộc Tính 1>, <Giá Trị Thuộc Tính 2>, …};

* Hoặc

<Tên Class><Object>{<Giá Trị Thuộc Tính 1>, <Giá Trị Thuộc Tính 2>, …};

* <Giá Trị Thuộc Tính> phải có kiểu dữ liệu cơ bản bé hơn kiểu dữ liệu cơ bản của thuộc tính được khai báo, chúng sẽ được gán lần lượt vào từng ô nhớ trong Stack ứng với các thuộc tính, nên không cần phải khởi tạo hết
* Đồng thời tất cả thuộc tính trong Class phải là công khai thì mới khai báo được kiểu này
* Ví dụ
* Class car có 4 thuộc tính chưa được gán giá trị là int a, b, float c, char d

car Honda = {4, 6, 7.8};

* Khi này Honda là 1 Object của Class car và giá trị của các thuộc tính của nó là

a = 4, b = 6, c = 7.8, d = không xác định

* Ta cũng có thể Copy toàn bộ thuộc tính trong Stack của 1 Object B cho 1 Object A khác khi khởi tạo

<Tên Class><Object A> = <Object B>;

* Hoặc nếu đã khởi tạo <Object A> rồi

<Object A> = <Object B>;

* Trong trường hợp ta ta không biết Class mà <Object B> thuộc về

auto <Object A> = <Object B>;

* Khi này <Object A> tự động thuộc có Class của <Object B>
* Khi khai báo Object thuộc kiểu dữ liệu cơ bản, thì các cách khai báo sau đây tương đương

<Kiểu Dữ Liệu Cơ Bản> <Object> = <Giá Trị>;

<Kiểu Dữ Liệu Cơ Bản> <Object>(<Giá Trị>);

<Kiểu Dữ Liệu Cơ Bản> <Object> = {<Giá Trị>};

<Kiểu Dữ Liệu Cơ Bản> <Object>{<Giá Trị>};

* Ví dụ

int foo = 5;

int foo(5);

int foo = {5};

int foo{5};

1. Khai Báo Hàm Ngoài Class?

* Ta đã nói về vấn đề này ở phần Namespace
* Khi khai báo hàm bên trong Class, thì tương đương với việc nó là 1 hàm Inline, nghĩa là khi Object nào đó gọi hàm này trong Code, thì chỗ đó sẽ được thay = hàm này luôn để cho đỡ tốn công gọi hàm, tuy nhiên hiệu ứng này chỉ áp dụng cho những hàm đơn giản
* Còn khai báo ngoài Class thì sẽ không có hiệu ứng này
* Để khai báo được hàm ngoài Class, thì bên trong Class đó bắt buộc phải có hàm nguyên mẫu
* Có thể khai báo Constructor bên ngoài Class như sau

<Tên Class>::<Tên Class>(<Các Tham Số>){

<Làm Gì Đó>

}

* Và Destructor

<Tên Class>::~<Tên Class>(<Các Tham Số>){

<Làm Gì Đó>

}

1. Trả Về 1 Object Của 1 Class?

<Class>(<Các Tham Số>)

* Lệnh trên sẽ gọi Constructor của <Class> và Pass cho nó <Các Tham Số>, sau đó trả về Object tương ứng
* Ví dụ

foo bar = foo(3, 4);

* Ở đây, chỉ có Constructor với 2 đối số nguyên là được gọi, Constructor với không đối số không được gọi

1. Tạo Tên Mới Cho Class?

typedef <Tên Class Cũ> <Tên Class Mới>;

* Khi này ta có thể vừa dùng <Tên Class Cũ> và <Tên Class Mới> để chỉ tới Class
* Hoặc đặt tên ngay khi tạo Class
* Ví dụ

typedef class {

int a;

float b;

} foo;

1. Const Object?

* 1 Const Object phải được gán giá trị khởi tạo ngay khi khai báo
* Để tạo 1 Object mà không thể thay đổi giá trị thuộc tính của nó trong Stack

const <Tên Class> <Object> = <Các Giá Trị Khởi Tạo>;

* Sau lệnh trên, bạn không thể gán giá trị cho các thuộc tính của <Object>
* Ngoài ra, từ khóa const có thể đặt bất kì chỗ nào, bên trái hay bên phải

<Tên Class> đều được

* Ví dụ

foo const bar = {4};

* Tương đương

const foo bar = {4};

* Để trình biên dịch báo lỗi nếu chúng ta khởi tạo giá trị cho Const Object không phải là 1 Const Object hoặc 1 giá trị cụ thể, thay từ khóa const thành constexpr, khi này nếu ta gán giá trị của Object B cho Object A chịu ảnh hưởng của constexpr, thì B phải là Const Object
* Const hợp với Class được coi là 1 Class duy nhất, ví dụ Class foo khác Class const foo

1. Volatile Object?

* Giả sử ta có foo = false, và ta tạo 1 Thread mà sau 1 lúc nó sẽ chỉnh foo về true
* Ta biên dịch Code bình thường thì không sao, nhưng khi biên dịch với việc tối ưu hóa, nghĩa là làm cho Code ngắn hơn, thì có thể trình biên dịch sẽ đéo biết foo có thay đổi, và do đó thế foo thành false hết, và chương trình lỗi
* Để tránh điều này, sử dụng Volatile Object

volatile <Tên Class> <Object> = <Các Giá Trị Khởi Tạo>;

* Ví dụ
* foo là Object thuộc Class bar, với 2 thuộc tính là int a, b

volatile bar foo = {4, 5};

* Khi này, trình biên dịch sẽ không thế 2 thuộc tính a và b thành 4 và 5 hết nữa vì nó biết foo có thể thay đổi
* Ngoài ra, từ khóa volatile có thể đặt bất kì chỗ nào, trái phải <Tên Class> đều được
* Volatile hợp với Class được coi là 1 Class duy nhất, ví dụ Class foo khác Class volatile foo

1. Phương Thức Hằng?

* Khi khai báo hàm trong 1 Class, ta có thể sử dụng cú pháp sau

<Class Trả Về> <Tên Hàm>(<Các Tham Số>) const {

<Làm Gì Đó>

}

* Từ khóa const ở đây sẽ chỉ định hàm này không thể thay đổi giá trị của thuộc tính của Object mà nó được gọi từ, nếu ta cố tình gán lại giá trị cho thuộc tính ở trong hàm này, trình biên dịch sẽ báo lỗi
* Bản chất tất cả thuộc tính nếu xuất hiện trong hàm này đều được thêm const vào đầu, do đó có thể phát sinh lỗi khó phát hiện như việc 1 toán tử không chấp nhận 1 đối số là const
* Ví dụ như sau sẽ báo lỗi

class foo {

int a = 4;  
 public:

void bar(int alice) const {

a = 8;

}

}

1. Thuộc Tính Tĩnh Nhưng Không Hằng?

* Để khai báo 1 thuộc tính dạng này trong 1 Class A, dùng cú pháp

static <Class Của Thuộc Tính Không Có Const> <Thuộc Tính>;

* Sau đó, phải khởi tạo giá trị cho <Thuộc Tính> bên ngoài Class A như sau

<Class Của Thuộc Tính Không Có Const> <Tên Class A>::<Thuộc Tính> =

<Giá trị Khởi Tạo>;

* Khi này, do <Thuộc Tính> được khởi tạo ở cấp cao nhất, nghĩa là cùng cấp hàm main, khi đó nó được coi như 1 Object toàn cục, lưu trong phần nhớ dành cho các Object toàn cục, Object tĩnh và hằng toàn cục, được chia sẻ bởi tất cả các Object của Class A, nên nếu 1 Object thay đổi nó thì các Object khác cũng nhận thấy sự thay đổi, đồng thời có thể truy xuất nó thông qua Class A như sau, nếu nó công khai

<Tên Class A>::<Thuộc Tính>

* Ví dụ

class foo {

static int a;

}

int foo::a = 4;

1. Thuộc Tính Hằng Tĩnh?

* Để khai báo 1 thuộc tính dạng này trong 1 Class A, dùng cú pháp

static <Class Của Thuộc Tính Có Const> <Thuộc Tính> = <Giá Trị Khởi Tạo>;

* Nếu không khởi tạo giá trị cho <Thuộc Tính> trong Class A, thì bạn phải khởi tạo giá trị cho nó ngoài Class A như sau

<Class Của Thuộc Tính Có Const> <Tên Class A>::<Thuộc Tính> =

<Giá trị Khởi Tạo>;

* Cơ chế lưu trữ tương tự thuộc tính tĩnh nhưng không hằng, tuy nhiên bạn sẽ không thể thay đổi giá trị cho nó sau khi khởi tạo

1. Phương Thức Tĩnh?

* Để khai báo 1 phương thức tĩnh trong 1 Class A, đặt static trước khai báo phương thức, bạn có thể định nghĩa phương thức bên trong hoặc bên ngoài Class
* Các phương thức tĩnh trong Class A chỉ được phép sử dụng các thuộc tính tĩnh trong Class A, đồng thời bạn có thể truy xuất phương thức tĩnh thông qua Class A như sau

<Tên Class A>::<Phương Thức>

1. Truy Cập Thuộc Tính Object Thông Qua Pointer?

* Cho A là Object nào đấy, B là Pointer chĩa tới A, A có hàm foo, để gọi hàm foo từ B, dùng cú pháp

B->foo()

* Tương tự để truy cập thuộc tính bar của A từ B

B->bar

1. This?

* Khi từ khóa this được sử dụng trong 1 phương thức, thì khi phương thức này được gọi bởi 1 Object A, thì this tương đương với Pointer mà Pointer này chĩa đến địa chỉ của Object A
* Ví dụ

class foo {

int alice = 2;  
 public:

void bar(){

this->alice = 5;

}

}

foo john;

john.bar();

* Khi này thuộc tính alice của john sẽ = 5
* Ta có thể thay cụm this-> thành <Tên Class>:: với công dụng y chang, ví dụ

class foo {

int alice = 2;  
 public:

void bar(){

foo::alice = 5;

}

}

foo john;

john.bar();

* Khi này thuộc tính alice của john sẽ cũng = 5

1. Hàm Bạn (Friend Function)?

* Giả sử ta có Class A và hàm B ở ngoài không thuộc Class A, khi này B không thể truy xuất các thuộc tính cũng như phương thức ẩn hoặc được bảo vệ trong Class A, tương tự không thể truy xuất chúng thông qua Object thuộc Class A được tạo trong B
* Để có thể làm được đều trên, ta cần đặt lệnh sau vào bất kì chỗ nào trong Class A

friend <Hàm Nguyễn Mẫu Của B>;

* Ví dụ

class foo {

int b = 4;

static int const a = 5;

friend int bob(float);

};

int bob(float alice){

foo daubuoi;

daubuoi.b = 8;

cout << foo::a;

}

* Ở đây, hàm bob có thể truy xuất và thay đổi thuộc tính ẩn b của Object daubuoi thuộc Class foo và truy xuất thuộc tính vừa tĩnh vừa hằng vừa ẩn là a của Class foo

1. Class Nguyên Mẫu (Class Prototype)?

* Tương tự hàm nguyên mẫu, giả sử ta cần sử dụng Class A trong khi nó chưa được khai báo, thì ta cần nói với trình biên dịch là ta sẽ khai báo nó sau

class <Tên Class A>;

* Ví dụ

class foo;

foo bar = {1};

class foo {

public:

int a;

}

1. Class Bạn (Friend Class)?

* Giả sử ta có Class A và Class B khác nhau, khi này các phương thức C trong B không thể truy xuất các thuộc tính cũng như phương thức ẩn hoặc được bảo vệ trong Class A, tương tự không thể truy xuất chúng thông qua Object thuộc Class A được tạo trong C
* Để có thể làm được đều trên, ta cần đặt lệnh sau vào bất kì chỗ nào trong Class A

friend <Class Nguyên Mẫu Của B>;

* Ví dụ

class foo {

int b = 4;

static int const a = 10;

friend class boo;

};

class boo {

void bob(){

foo alice;

alice.b = 6;

cout << foo::a;

}

};

* Ở đây, hàm bob trong Class boo có thể truy xuất và thay đổi thuộc tính ẩn b của Object alice thuộc Class foo và truy xuất thuộc tính vừa tĩnh vừa hằng vừa ẩn là a của Class foo

1. Kế Thừa?

* 1 Class có thể kế thừa nhiều Class, khi này ta nói Class này là Class con, các Class nó kế thừa từ gọi là các Class cha

class <Class Con> : <Kiểu 1> <Class Cha 1>, <Kiểu 2>, <Class Cha 2>, … {

<Làm Gì Đó>

}

* <Kiểu> chỉ có thể là public, protected hoặc private, nếu không chỉ định thì nó là private
* Bản chất là dùng lại bản mẫu đã lưu của các Class cha trong phần nhớ Text, đồng thời mở rộng ra thêm vài thuộc tính và phương thức, do đó, Class con sẽ dùng chung các thuộc tính tĩnh cũng như hàm tĩnh với Class cha, nên nếu thuộc tính tĩnh thay đổi ở Class cha hoặc Class con, thì thằng còn lại cũng sẽ thay đổi
* Nhưng để cho dễ hình dung, ta coi nó như Copy nội dung từ <Class Cha> đặt vào <Làm Gì Đó>
* Nếu <Kiểu> là public, thì bê nguyên văn bản trong <Class Cha> đặt vào

<Làm Gì Đó>, trừ các thuộc tính và phương thức ẩn trong <Class Cha> được bê sang nhưng không thể truy cập

* Nếu <Kiểu> là protected, thì tương tự public, trừ việc các thuộc tính và phương thức công khai trong <Class Cha> khi bê sang sẽ trở thành được bảo vệ
* Nếu <Kiểu> là private, thì tương tự public, trừ việc các thuộc tính và phương thức công khai lẫn được bảo vệ trong <Class Cha> khi bê sang đều sẽ trở thành ẩn
* 2 thuộc tính hoặc hàm ở 2 <Class Cha> có cùng tên được bê sang thì bạn không thể truy cập chúng vì sự mơ hồ
* Ví dụ

class foo : public boo, protected alice, private bob, public john {

protected: int a = 5;

};

* Riêng Constructor và Destructor thì cũng Copy sang nhưng luôn ở chế độ công khai, còn hàm bạn hay Class bạn không được Copy
* Những gì được Copy sang sẽ được đặt ở trước phần <Làm Gì Đó>, theo thứ tự là nội dung của <Class Cha 1> sẽ nằm trên đầu, rồi đến <Class Cha 2>, …
* Khi này ta sẽ có 1 danh sách các Constructor và Destructor từ trên xuống theo đúng thứ tự <Class Cha 1>, <Class Cha 2>, … với nguyên mẫu khác nhau
* Khi 1 Object thuộc 1 Class được tạo ra, thì các Constructor trong danh sách Constructor của nó sẽ được gọi lần lượt từ trên xuống, mà do có nhiều nguyên mẫu khác nhau của Constructor của Class cha, ta cần phải làm gì đó để truyền danh sách tham số lần lượt cho các Constructor
* Tương tự khi Object này bị xóa, thì các Destructor trong danh sách Destructor sẽ được gọi từ dưới lên, theo thứ tự <Class Con>, <Class Cha Cuối>, …,

<Class Cha 1>, riêng cái này thì không có gì phải bàn vì nó đéo nhận tham số

* Tương tự khi có 1 Class kế thừa <Class Con>, thì toàn bộ danh sách Constructor và Destructor của nó cũng được bê sang
* Để truyền tham số từ Constructor của <Class Con>, cho Constructor của

<Class Cha>

<Tên Class Con>(<Các Tham Số>) :

<Tên Class Cha 1>(<Các Giá Trị 1>),

<Tên Class Cha 2>(<Các Giá Trị 2>),

…

{

<Đây Là Phần Thực Thi Constructor Của Class Con>

}

* Khi các Class cha nhận được tham số rồi, thì nó có thể tiếp tục truyền tham số cho Class ông nội của nó nữa
* Nếu không truyền tham số, hoặc bạn không viết <Tên Class Cha>(<Các Giá Trị>) luôn, thì Constructor được gọi ở Class cha sẽ là Constructor với không đối số
* Lưu ý thứ tự của <Tên Class Cha> không quan trọng, vì thứ tự Constructor đã được xác định trong danh sách Constructor
* Ví dụ

class boo {

public:

boo(){

cout << "boo ";

}

boo(int a){

cout << a << " ";

}

~boo(){

cout << "boodie ";

}

};

class haha {

public:

haha(){

cout << "haha ";

}

haha(int a, int b){

cout << a << " " << b << " ";

}

~haha(){

cout << "hahadie ";

}

};

class foo : public boo, public haha {

public:

foo(){

cout << "foo ";

}

foo(int john) : boo(john + 4), haha(john + 1, john + 10){

cout << "foo ";

};

~foo(){

cout << "foo chet ";

}

};

class cuoi : public foo {

public:

cuoi(){

cout << "cuoi ";

}

cuoi(int gg) : foo(gg + 5){

cout << gg << " ";

}

~cuoi(){

cout << "cuoi chet ";

}

};

cuoi zoro(4);

* Khi Object zoro thuộc Class cuoi được tạo và được truyền đối số là 4, thứ được thêm vào Buffer là

13 10 19 foo 4

* Giải thích
* Khi bạn ghi zoro(4), thì Constructor ở dòng màu đỏ có nguyên mẫu hợp lệ, sau đó nó truyền gg + 5, tức là 9 cho Constructor của foo, khi này ở foo, Constructor dòng màu cam có nguyên mẫu hợp lệ, do đó truyền john + 4, tức là 13 cho Constructor của boo, đồng thời truyền john + 1, tức là 10, và john + 10, tức là 19 cho Constructor của haha, tại boo, Constructor màu xanh có nguyên mẫu hợp lệ, và ở haha, Constructor màu xám có nguyên mẫu hợp lệ, do đó thứ tự gọi Constructor sẽ là xanh, xuất ra 13, rồi xám, xuất ra 10, 19, rồi cam, xuất ra "foo ", rồi đỏ, xuất ra 4
* Khi Object zoro bị xóa, thứ được thêm vào Buffer là

cuoi chet foo chet hahadie boodie

* Cái này thì không cần giải thích do thứ tự quá rõ ràng
* Ta cũng có thể sử dụng cú pháp sau để khởi tạo giá trị cho các thuộc tính, kể cả hằng

<Tên Class>(<Các Tham Số>) :

<Tên Thuộc Tính 1>(<Giá Trị 1>),

<Tên Thuộc Tính 2>(<Giá Trị 2>),

…

{

<Đây Là Phần Thực Thi Constructor Của Class Con>

}

* Ví dụ

class foo {

private:

const int a;

float b;

public:

foo(int gg, float hh): a(gg), b(hh){

cout << 1;

}

};

foo bar(7, 5.8);

* Khi này thuộc tính a của Object bar sẽ = 7 và thuộc tính b của nó = 5.8

1. Phương Thức Ảo (Virtual Method)?

* Khi 1 khai báo ít nhất 1 phương thức ảo trong 1 Class thì nó trở thành Class ảo, một Class cũng là Class ảo nếu Class cha của nó là Class ảo
* Class ảo tạo ra từ việc khai báo phương thức ảo trong 1 Class sẽ được tự động thêm 1 thuộc tính là Virtual Pointer vào đầu danh sách các thuộc tính, và do đó Virtual Pointer sẽ được lưu trong bộ nhớ
* Class ảo tạo ra từ việc kế thừa Class ảo khác, thì không tạo ra thêm Virtual Pointer mà nó dùng lại Virtual Pointer của các Class cha
* Khi gán 1 Object thuộc Class con cho Object thuộc Class cha, giả sử Class con kế thừa từ nhiều Class cha, thì nó sẽ chọn vùng nhớ tương ứng với Class cha đó để gấn, vẫn sử dụng các hàm ở Class cha
* Giả sử Class cha có hàm foo, Class con ta có định nghĩa lại hàm foo, và ta gán địa chỉ của Object thuộc Class con cho Pointer A chĩa vào Object thuộc Class cha, thì A sẽ đối xử với địa chỉ này như Class cha, tuy nhiên nếu trong địa chỉ này có 8 Byte Virtual Pointer, thì khi dùng A gọi tới hàm foo, nó sẽ gọi tới foo của Class con thay vì Class cha
* Trước tiên, đặt từ khóa virtual trước khai báo phương thức C trong Class cha
* Bước 2, khởi tạo 1 Pointer chĩa vào A nhưng kiểu của nó phải là Pointer chĩa vào Object thuộc Class cha
* Bước 3, từ Pointer này gọi đến phương thức C, khi này C được gọi sẽ là C đã bị ghi đè trong A, nếu không có từ khóa virtual, thì C được gọi sẽ là C ban đầu trong Class cha
* Ví dụ

class foo {

public:

virtual void boob(){

cout << 2 << endl;

}

void khongao(){

cout << 3 << endl;  
 }

};

class boo : public foo {

public:

void boob(){

cout << 1 << endl;

}

void khongao(){

cout << 4 << endl;  
 }

};

foo\* alice = new boo();

alice->boob();

alice->khongao();

* Buffer

1

3

* Giải thích
* Ta tạo ra Pointer alice, nó có kiểu foo\* tức là nó xem những gì mình chĩa vào là Object thuộc Class foo, nhưng ta lại gán 1 Object thuộc Class boo cho nó
* Khi này do phương thức boob của Class foo là ảo nên alice->boob sẽ được hiểu là chĩa đến phương thức boob trong boo, còn phương thức khongao là không ảo nên alice->khongao được hiểu là chĩa đến phương thức khongao trong foo
* Xét trường hợp ta tạo Object của Class con trong Heap, và dùng Pointer A chĩa vào Class cha để chĩa tới Object này, thì khi xóa Object này thông qua A, Destructor của Class con sẽ đéo được gọi, do đó cần khai báo Destructor của Class cha là phương thức ảo, khi này sẽ đảm bảo Destructor của Class con sẽ được gọi trước rồi đến Destructor của Class cha
* Dễ thấy Pointer đến Class cha chĩa vào Class con không thể gọi phương thức có trong Class con nhưng không có trong Class cha, để làm được điều này, ta có thể ép kiểu Pointer từ Class cha sang Class con, ví dụ

foo \* fool = new bar();

((bar\*)) fool -> john();

* Ở ví dụ trên, foo là Class cha của Class bar, hàm john chỉ có ở Class bar, không có trong Class foo, nên ta không thể dùng fool để gọi john, do đó ta phải ép kiểu của fool từ foo \* sang bar \* đê có thể gọi hàm john trong Class bar
* Bạn có thể hiện thực phương thức ảo ở ngoài Class, không ghi từ khóa virtual

1. Phương Thức Thuần Ảo (Pure Virtual Method)?

* Là phương thức ảo nhưng được gán = 0
* Ví dụ

virtual void bob() = 0;

* Viết kiểu trên bình thường sẽ lỗi nhưng với phương thức ảo thì không, và bạn có thể thực hiện hàm bob ở Class con, nói chung chả khác đéo gì phương thức ảo thông thường
* Class chứa ít nhất 1 phương thức thuần ảo thì gọi là Abstract Class, Class kế thừa từ Abstract Class mà không hiện thực lại hàm thuần ảo, thì cũng gọi là Abstract Class
* Abstract Class không được phép được tạo Object, nếu cố tình tạo sẽ báo lỗi

1. Struct?

* Y chang Class, chỉ khác ở chỗ thay từ khóa class thành struct, và nếu không chỉ định gì thì thuộc tính và hàm trong nó là công khai

1. Union?

* Y chang Struct, nghĩa là có thể tạo phương thức, Constructor, Destructor, … trong nó, chỉ khác ở chỗ thay từ khóa struct thành union, và tất cả thuộc tính của nó lưu ở cùng 1 địa chỉ, kích thước của nó = kích thước của thuộc tính có kích thước lớn nhất, do đó nó cũng được đặt ở địa chỉ chia hết cho kích thước này
* Hệ quả là thay đổi 1 thuộc tính sẽ làm thay đổi tất cả thuộc tính còn lại
* Khi khởi tạo 1 Union, chỉ được Pass 1 giá trị, và giá trị này sẽ được gán cho thuộc tính đầu tiên, các thuộc tính tiếp theo tự động bị ảnh hưởng
* Ví dụ

union foo {

int a, b;

double c;

char d;

int e;

private:

void haha(){  
 cout << 1;

}

};

foo bar = {4};

* Hình ảnh bộ nhớ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Địa chỉ | Dữ liệu |  |  |  |  |
| 113 | 0 |  |  |  |  |
| 112 | 0 |  |  |  |  |
| 111 | 0 |  |  |  |  |
| 110 | 0 |  |  | c = 1.97626 x 10–323 | bar |
| 109 | 0 |  |  |
| 108 | 0 |  |  |
| 107 | 0 |  | a, b, e = 4 |
| 106 | 0 |  |
| 105 | 0 |  |
| 104 | 4 | d = 4 |

Quick Code – Mã Nhanh:

1. Di Chuyển Con Trỏ Đến Vị Trí Nào Đó Trên Màn Hình?

#include <windows.h>

void gotoxy(short x, short y)

{

COORD c = { x, y };

SetConsoleCursorPosition(GetStdHandle(STD\_OUTPUT\_HANDLE) , c);

}

* Cách dùng

gotoxy(<Hoành Độ>, <Tung Độ>);

* <Hoành Độ> có chiều dương từ trái qua phải
* <Tung Độ> có chiều dương từ trên xuống dưới
* Bắt đầu từ (0, 0)

Cheat Engine:

1. Danh Sách Process?

* Để mở cửa sổ danh sách Process
* Click biểu tượng “Select a process to open” góc trái trên
* Sẽ có 3 Tab ứng với 3 loại Process, chương trình C++ sẽ thuộc loại “Processes”
* Để chuyển sang làm việc trên Process nào đó
* Chọn Process nào đó trong cửa sổ danh sách Process + Click “Open”

1. Memory Viewer?

* Để mở cửa sổ Memory Viewer
* Click “Memory View” bên trái giữa
* Ở đây sẽ hiện những ô nhớ của Process được chọn, gồm 2 phần
* Phần bên trên sẽ biên dịch giá trị các ô nhớ sang Instruction
* Phần bên dưới sẽ hiện nguyên mã thập lục phân
* Quy ước 2 số Hexa tương ứng 1 Byte, phần bên dưới mỗi địa chỉ cách nhau 16 Byte, mỗi Byte được liệt kê bên phải, “0123…” = dãy các Byte này viết theo ASCII
* Dùng cuộn chuột hoặc thanh cuộn để cuộn lên xuống địa chỉ
* Ở phần bên trên, địa chỉ có dạng <Tên Process> + <Offset>, địa chỉ

<Tên Process> sẽ là ngõ vào chương trình, hay Instruction đầu tiên, <Offset> viết dưới dạng thập lục phân

* Để nhanh chóng di chuyển đến địa chỉ nào đó, phải chuột vào phần trên hoặc dưới + nhấn “Ctrl” + “G” + nhập địa chỉ muốn tới + nhấn “Enter”
* Các giá trị nằm ngoài phạm vi Process sẽ hiện “??”

1. Địa Chỉ Hiện Ra Có Đúng Không?

* Đây là địa chỉ ảo, đéo phải địa chỉ thật trong RAM

1. Tự Động Chuyển Sang Làm Việc Trên 1 Process Khi Cheat Engine Mới Mở?

* Vào Tab “Edit” + chọn “Settings” + vào “General Settings” + tại mục “Automatically attach to processes named” + nhập Full tên Process có dấu chấm phẩy đằng sau, ví dụ “gg.exe;” + Click “OK”

Debug:

1. GDB (GNU Debugger)?

* Vào CMD, nhập lệnh sau để gắn Debugger vào File EXE, lưu ý khi biên dịch File EXE phải kèm theo thông tin Debug như Index dòng của các lệnh, kiểu dữ liệu của biến, …

gdb <Đường Dẫn Đến File EXE Không Có Phần Mở Rộng>

* ví dụ

gdb foo\bar\bob

* Khi này Debugger sẽ bắt đầu khởi chạy, File EXE chưa chạy
* Từ đây bạn có thể thực hiện các thao tác Debug
* Để thoát khỏi trình Debug, nhập

q

* Để khởi chạy File EXE, nhập

r <Các Tham Số Hàm Main>

* Ví dụ

r thangngu thangngoc

* Chương trình sẽ chạy một lèo trong cùng CMD với GDB rồi kết thúc, bạn có thể chạy nhiều lần, có thể nhập Input từ bàn phím
* Để tạo 1 Break Point để chương trình dừng lại tại thời điểm nào đó để ta Debug, nhập lệnh sau trước khi chạy

b <Dòng Bao Nhiêu>

* <Dòng Bao Nhiêu> có thể ghi là main, khi này Break Point sẽ được đặt ở dòng lệnh kế sau dòng khai báo hàm main
* Ví dụ

b main

b 14

b 6

* Nếu tạo 2 Break Point tại cùng 1 dòng thì chỉ có tác dụng = 1 Break Point
* Ví dụ bạn tạo Break Point tại các dòng 5, 9, 14, 18
* Đầu tiên chương trình chạy các lệnh từ dòng 1 đến hết dòng 4
* Sau đó chuyển sang Debugger, nếu Debugger ra lệnh tiếp tục, thì sẽ tiếp tục chạy từ đầu dòng 5 đến hết dòng 8, …
* Để tiếp tục chạy từ vị trí hiện tại đến Break Point tiếp theo, nhập

c

* Nếu không còn Break Point nào thì sẽ chạy hết chương trình và kết thúc Process ứng với File EXE
* Nếu vô tình cũng có 1 File cùng tên File EXE nhưng có phần mở rộng “.cpp”, 2 File này nằm chung 1 thư mục, thì khi dừng tại Break Point nào, thì dòng ứng với Break Point đó trong File CPP sẽ được in ra màn hình, bất kể là nó có phải File nguồn hay rác