Day 1 Git , cấu trúc 1 chương trình C

**Git status**: liệt kê ra tất cả những file đã được sửa đổi (thêm, xóa, sửa đổi), những file này sẵn sàng được add (dòng lệnh màu đỏ) với lệnh **git add** (copy đường dẫn dòng lệnh màu đỏ ở git status).

**Git add**: Trước khi add nhớ check lại branch đúng bằng lệnh git branch -a, nếu branch chưa đúng thì dùng lệnh git checkout ‘tên branch đúng’,

**Git commit** -m “<tin nhắn commit>”)

**Git push** origin HN22\_FR\_CPP\_01.

**Git pull :** lấy code mới nhất từ web.

Biến và các kiểu dữ liệu biến

Cấu trúc 1 chương trình C++

* Documentation: chú thích mục đích viết chương trình
* Link section: header file, namespace
* Definition section: Khai báo hằng số và gán giá trị (keywords **typedef** int myInt)
* Global declaration: Khai báo biến global và class…
* Function definition section: Định nghĩa các hàm cần dùng
* Main function: vị trí mà complie bắt đầu thực thực thi chương trình

**Header guard**: #ifndef …………………..#endif: dùng để tránh các lỗi gây ra do định nghĩa lại một biến hay 1 hàm có cùng tên khi ta include giữa các file khác nhau.

Data type:

* C/C++ nguyên thủy: int, char, long, double, float, void, **wide char**,…
* Derived: function, array, pointer, referrence.
* Người dùng định nghĩa: Struct, class, **union**, enum,…

**Datatype modifier:** signed, unsigned, long(integer, double), **short(integer)**

Day 2

Con trỏ, mảng, chuỗi, struct

Con trỏ

* **Con trỏ là gì?**
* **Con trỏ sử dụng như thế nào?**
* **Null pointer và invalid pointer?**
* **Các phép toán với con trỏ?**
* **Ép kiểu cho con trỏ?**
* **Pointer và reference?**
* **Bộ nhớ động?**

Con trỏ là gì?

* Con trỏ là 1 biến mà nó lưu giữ địa chỉ của 1 biến khác.
* Kích thước con trỏ là giống nhau cho mọi kiểu dữ liệu (là 4 byte hoặc 8 byte tùy theo từng hệ điều hành 32 bit hoặc 64 bit mà chương trình đang chạy)

Cách khai báo con trỏ, cách sử dụng con trỏ?

* Khai báo: datatype \* nameptr;
* Lưu giữ địa chỉ của 1 biến: int x = 2; int \*p = &x;
* Truy cập giá trị tại địa chỉ con trỏ đang trỏ đến: cout << \*p;
* Thay đổi giá trị tại địa chỉ con trỏ đang trỏ đến: \*p = 3;

Null pointer và invalid pointer?

* Null pointer là hằng với giá trị 0, nếu con trỏ là null pointer thì nó không trỏ đi đâu cả.
* Invalid pointer là 1 con trỏ trỏ tới vùng nhớ không xác định.

Phép toán với con trỏ?

* Tăng giảm (++, --, +=, -=, +, -)
* So sánh (==, != ..)

Ép kiểu con trỏ?

Vd: char arr[8] = [1,2,3,4,5,6,7,8];

Char \*cp = arr;

cp++ // giá trị là 2

Int \*ip = (int\*) p; // giá trị là 2

Ip++ // giá trị là 6,

Pointer và reference?

Reference là 1 alias(bí danh, tên gọi khác) của 1 biến đã tồn tại

Khác nhau giữa reference và con trỏ: không thể có null reference và reference phải được khởi tạo ngay khi được tạo ra, sau khi khởi tạo thì nó không thể reference đến biến khác.

Memory layout in c++:

* **Text segment**: chứa mã lệnh của chương trình
* **Initialize data segment**: chứa biến toàn cục, biến static, hằng số được gán gia trị ngay khi tạo ra.
* **Uninitialize data segment (bss):** chứa biến toàn cục, biến static chưa được gán giá trị khi tạo ra
* **Stack segment**: chứa biến cục bộ quản lý bởi CPU theo cơ chế FILO.
* **Heap segment**: dùng để cấp phát động khi chương trình chạy

Bộ nhớ động ?

Khi dùng con trỏ thì ta cấp phát bộ nhớ động cho nó (bằng lệnh new or calloc, relloc, malloc) , khi dùng xong thì phải giải phóng vùng nhớ (bằng lệnh delete) đó cho hệ điều hành. Vùng nhớ này là vùng nhớ **heap**.

Array

* **Mảng là gì?**
* **Cách sử dụng mảng?**
* **Mảng và con trỏ?**
* **Mảng nhiều chiều?**

Mảng là gì?

* Mảng là cấu trúc dữ liệu lưu trữ 1 tập hợp tuần tự có kích thước cố định của các phần tử cùng loại.
* Các phần tử trong mảng có địa chỉ ô nhớ liền kề, liên tục nhau.

Cách sử dụng mảng?

* Khai báo mảng: int arr[4];
* Khai báo cùng khởi tạo mảng: int arr[4] = {1,2,3,4};
* Truy cập phần tử trong mảng: cout << arr[2] ; // giá trị = 3.

Mảng và con trỏ?

* Mảng là 1 con trỏ hằng, con trỏ hằng thì chỉ thay đổi giá trị mà nó trỏ tới, không thể trỏ tới địa chỉ của 1 biến khác.
* Vd: int arr[3]; không thể thực hiện arr++,

Mảng nhiều chiều?

Khai báo và sử dụng tương tự mảng 1 chiều

Vd: int arr[2][3]; int arr[2][3] = {1,2,3,4,5,6}; int arr[2][3] = {{1,2},{3,4},{5,6}},;

String

* **String là gì?**
* **Cách sử dụng các hàm với string**
* **Cách sử dụng class string**

String là gì?

* String là 1 mảng 1 chiều lưu trữ các phần tử của chuỗi, kết thúc bằng kí tự rỗng “\0”.
* Vd: char str[] = “hello”;

Các hàm xử lý với chuỗi? Khai báo với thư viện **string.h**

* **Strcpy**(s1,s2); //copy s2 vào s1
* **Strcat**(s1,s2); //nối s2 vào s1
* **Strcmp**(s1,s2); trả về 0 nếu 2 chuỗi như nhau, trả về bé hơn 0 nếu s1 < s2, trả về lớn hơn 0 nếu s1 > s2.
* **Strlen**(s1); // trả về chiều dài của chuỗi s1. Vd: “hello” có len = 5.
* **Strstr**(s1,s2) // trả về 1 con trỏ trỏ tới lần xuất hiện đầu tiên của s2 tại s1 vd: s1 = “Day la 1 vi du ve strstr()”, s2 = “vi” 🡪 char\* sub = strstr(s1,s2); // sub = “vi du ve strstr()”.
* **Strchr**(s1,s2)// tương tự như strstr(s1,s2) nhưng s2 ở đây là 1 ký tự.

Cách sử dụng thư viện với chuỗi char.

Thư viện sử lý chuỗi **string**

Sử dụng toán tử “+”

Str.size()🡪 trả về chiều dài chuỗi.

Structure

* **Struct là gì?**
* **Cách sử dụng struct?**
* **Cách sử dụng con trỏ trong struct**
* **Keyword: typedef**

Struct là gì?

* Struct là 1 kiểu dữ liệu do người dùng tự định nghĩa, các phần tử trong struct có thể có các kiểu dữ liệu khác nhau.
* Struct thường được sử dụng để tổ hợp các thuộc tính của 1 đối tượng đơn giản.

Cách sử dụng struct?

* Định nghĩa 1 struct: struct ten\_struct{……};
* Truy nhập các thuộc tính trong struct bằng toán tử “.”.

Con trỏ struct?

* Vd: struct books \*b; p**->** thuộc tính

Keyword typedef

* Typedef struct{…} ten\_viet\_tat;
* Struct tenstruct{…}; typedef struct tenstruct ten\_viet\_tat.

Reference: <https://www.tutorialspoint.com/cplusplus/index.htm>

Day 3

Phạm vi của biến, name space, Các phép toán, control flow

Phạm vi của 1 biến

Lifetime: block lifetime , user define lifetime, whole program lifetime.

Linkage: block scope, file scope, multi file scope.

* **Biến global**: là biến được khai báo ở ngoài tất cả các hàm, có thể được truy xuất ở bất kỳ đâu trong chương trình và có lifetime xuyên suốt tới khi chương trình kết thúc.
* **Biến local:** biến được khai báo trong 1 phạm vi (scope, 2 dấu ngoặc kép), thông thường chúng chỉ tồn tại trong phạm vi đó, ngay khi ra ngoài phạm vi đó, biến đó sẽ bị hủy, và không thể được truy cập. Tuy vậy ta vẫn có thể kéo dài vòng đời của biến bằng từ khóa static.
* Khi biến local và global cùng tên **thì biến local sẽ che phủ biến global** tức là khi gọi và sử dụng biến thì ưu tiến dùng biến local, muốn sử dụng biến global ta phải sử dụng ::ten\_bien,

Từ khóa extern và static.

**Từ khóa extern**: Khi chương trình gồm nhiều module thì ta phải chia code thành nhiều file (.h, .cpp) và khi ta muốn sử dụng 1 biến được định nghĩa trong các file này như 1 biến global thì ta sẽ sử dụng từ khóa **extern**.

Ứng dụng từ khóa extern có 2 cái:

* 1 là mở rộng linkage của 1 biến const được khai báo ở globle của 1 file ( đặc điểm là ở 2 chỗ khai báo sẽ đều có từ khóa extern và chỉ có 1 chỗ được phép khởi tạo

Ví dụ: có 2 file sub.cpp và main.cpp

Sub.cpp: Const int x = 10;

Ta muốn sử dụng biến này ở file main.cpp thì ta cần phải thêm từ khóa extern ở trước const để **mở rộng linkage** cho nó và phải forward declaration biến x.

* Sub.cpp sẽ là **extern const x = 10;**
* Main.cpp **extern const x**; // chỉ được phép khai báo.
* Còn 1 ứng dụng nữa là mở rộng phạm vi của 1 biến global sang 1 file khác

ví dụ: file main.cpp ta có biến globle int x = 10; ở file sub.cpp ta muốn sử dụng biến này như 1 biến global thì ta cũng cần forward declaration biến x như sau sub.cpp extern int x; // chỉ khai báo, không initialize.

**Static**: dùng để hạn chế phạm vi truy cập của 1 biến globle chỉ ở trong file này. Khi biến static ở trong 1 hàm thì nó có tác dụng kéo dài lifetime của biến đó tới khi chương trình kết thúc, biến static sẽ lưu giữ các giá trị từ lần gọi hàm trước đó.

**Name space:** Cho phép nhóm các biến các hàm vào 1 nhóm khi các hàm biến đó cùng tên với các hàm, các biến khác. Để khi gọi hàm ta sử dụng tên namespace để phân biệt các hàm cùng tên.

*rude: chỉ khai báo namespace ở global scope. Nó có thể lồng nhau, không có access modifier (public, private), không cần dấu chấm phẩy ở sau dấu ngoặc nhọn kết thúc namespace.*

**Toán tử**

* Số học
* Logic
* Quan hệ
* Bitwise
* Gán
* Toán tử khác: sizeof; (condition)?x:y; \*; ->

***Thứ tự các toán tử ……***

**Vòng lặp, rẽ nhánh.**

* If(){} else if{} else{}
* For(){}
* Switch(key) case 1:{ … break;} case 2:{ … break;} defaul{}
* While(condition){}
* Do{}while(condition)

Day 4

Hàm, truyền tham chiếu, tham trị, con trỏ

* Hàm cách gọi hàm
* Pass by value, reference, pointer
* Return by value, reference, pointer
* Function pointer

Hàm và cách gọi hàm

* Hàm là gi?
* Cấu trúc 1 hàm
* Tham số và đối số?
* Khai báo hàm, nguyên mẫu hàm

Hàm là gì?

* Hàm là 1 tập hợp các câu lệnh có thể có các input , thực thi 1 công việc nào đó.

Cấu trúc 1 hàm?

* Kiểu\_trả\_về tên\_hàm(danh\_sách\_tham\_số){…};
* Khai báo, định nghĩa hàm
* Kiểu trả về void, vẫn có thể return;

Argument và parameter?

* **Parameter** là 1 biến, sử dụng trong phương thức định nghĩa 1 hàm.
* **Argument**: đối số, truyền vào hàm khi gọi hàm sử dụng

Pass by value, pointer, reference.

* **Pass by value:** Tạo 1 bản copy giá trị, gửi copy tới hàm khi gọi hàm, giá trị biến argument sau khi truyền vào hàm sẽ k đổi.
* **Pass by pointer**: truyền vào hàm là địa chỉ của biến argument, do vậy sau khi gọi hàm, nếu trong hàm biến truyền vào thay đổi giá trị thì biến argument truyền vào hàm cũng thay đổi giá trị.
* **Pass by reference:** truyền vào hàm là 1 tên gọi khác của chính argument, do vậy, do vậy trong hàm thay đổi, thì biến truyền vào cũng thay đổi giá trị.
* **Default parameter:** sẽ được xếp phía sau trong danh sách các tham số của hàm

Return value, pointer, reference

* **Return value:** là cách đơn giản, an toàn, và hay sử dụng nhất, khi return thì 1 bản copy giá trị sẽ được trả về.
* **Return address**: thực hiện nhanh do trả về địa chỉ của giá trị, nhưng dễ gây lỗi do địa chỉ trả về của biến đó, nếu chỉ là biến cục bộ trong hàm thì sẽ bị hủy ngay khi ra khỏi hàm, do vậy thường sử dụng return về biến global hoặc static.
* **Return reference:** nó cũng thực hiện nhanh, hữu ích khi trả về đối tượng là struct hoặc class, nhưng cũng tương tự return address biến trả về sẽ bị hủy khi out of scope.

Function pointer

* Khi ta muốn truyền 1 hàm làm tham số đầu vào của 1 hàm khác ta sử dụng function poiner.
* Function pointer giống như 1 pointer ngoại trừ việc nó trỏ tới 1 hàm, thay vì 1 biến.
* Định nghĩa 1 function pointer:

kieu\_du\_lieu (\*ten\_function\_ponter)(danh\_sach\_tham\_s0

Buổi 5

Debug chương trình

* Syntax error
* Runtime error: runtime error + logic error.

**Syntax error**: xuất hiện lỗi sai về cú pháp, các rule đã được định nghĩa trước.

**Runtime error:** vd chia 1 số cho 0, truy nhập 1 phần tử nằm ngoài phạm vi của mảng, list…; truy cập null pointerd; detete con trỏ khi k cấp phát động, lỗi logic (bug)

**Logic error:** chương trình chạy không đúng theo yêu cầu, sảy ra khi chương trình chương trình chạy được, nhưng kết quả không đúng: vòng lặp vô tận, sai về thứ tự thực hiện phép toán, nhầm lẫn trong các khối if\_else, không break trong switch case.

Buổi 6

OOP

* **Tính đóng gói (ẩn thông tin)**: về cơ bản thì 1 chương trình gồm có dữ liệu và các đoạn mã function chức năng nào đó, khi 1 thành phần này được gom lại với nhau sẽ tạo thành 1 class. Do vậy 1 class có các dữ liệu và các phương thức thao tác với dữ liệu. Tính đóng gói liên quan chặt chữ tới tính trừu tượng (ẩn dấu dữ liệu) . Các chi tiết dữ liệu về đối tượng sẽ được ẩn khỏi người dùng bằng các cấp độ access modifier, chỉ cho phép người dùng truy cập đối tượng thông qua các giao thức được công khai, thông qua đó người dùng có thể sử dụng đối tượng mà không biết chi tiết nó thực hiện như thế nào.

**Lợi ích**: ẩn dấu thông tin, bảo vệ dữ liệu, dễ thay đổi khi cần sửa chữa.

* **Tính kế thừa:** kế thừa từ các class trước, tăng mức độ sử dụng lại của code (re use code)
* **Đa hình**: kế thừa các phương thức từ class cha, override, hàm ảo (virtual function)
* **Tính trừu tượng**: ta chỉ hiển thị nội dung cần thiết cho người dùng ẩn đi các chi tiết hoạt động không cần mà người dùng không cần thiết biết. Có 2 cách để trừu tượng hóa dữ liệu là trừu tượng hóa sử dụng các class và trừu tượng hóa sử dụng các tập file.h. Trong class thì việc trừu tượng hóa được thức hiện qua các access modifier public private.

**Class and object**

* 1 class sẽ chứa data (member variable) và function ( method)
* Object là 1 biến có kiểu dữ liệu đặc biệt được định nghĩa như tên 1 class

**Acess modifier**: public, private, protected: giới hạn lại mức độ truy cập của các biến và phương thức của class

**Overload** : cùng tên, khác tham số

**Override:** cùng tên, cùng kiểu, số lượng các tham số

**Constructor**: hàm đặc biệt tự động được gọi khi 1 đối tượng của class được tạo.

**Default constructor:** là các constructor mà không có tham số truyền vào

*Constructor phải có tên cùng với tên class.*

Mặc định c++ sẽ tự có 1 default constructor, nhưng khi ta tạo 1 bất kỳ 1 constructor nào thì constructor default của c++ sẽ tự động mất.

**Copy construtor:**

**Destructor:** chỉ có 1 và k có tham số truyền vào, sẽ tự động gọi khi object bị hủy, hữu ích khi ta sử dụng cấp phát động trong chính class. Khi kế thừa thì tạo cha trước, tạo con sau, còn hủy thì hủy con trước, hủy cha sau.

Buoi 7 Class

Tại sao cần chia giữa khai báo và định nghĩa 1 class

Giảm độ phức tạp của 1 class. Khi thay đổi gì đó trong header file thì sẽ yêu cầu compile lại toàn bộ các file, mà khi thay đổi gì đó trong file.cpp thì chỉ việc compile lại file.cpp đó 🡪 thực hiện build nhanh hơn, tiện hơn khi sửa đổi gì đó.

**Biến static**

* Thuộc class ( bất kì object nào, hoặc k cần object đếu có thể lấy được giá trị của biến static), nó k thuộc riêng object nào.
* Biến static có thể nằm ở public, private và cả protected.
* Khởi tạo cho static data member in class:
* Nó không thể được khởi tạo giá trị từ constructor, hay tại dòng nó được khai báo, mà phải được khởi tạo ở global scope (out of line của class) bằng toán tử phân giải phạm vi :: (*bắt buộc phải khởi tạo giá trị cho biến static mới dùng được*).
* Nếu không phải là const thì biến static không thể được khởi tạo giá trị ngay trong file .h mà phải được khởi tạo ở ngoài tất cả các hàm ( khởi tạo bên trong file .cpp).
* Ở C++ 17 thì ta có thể khai báo và khởi tạo cho static data member bằng từ khóa inline vd: inline static int x = 10; 🡪 sau đó, ta đã có thể sử dụng biến này.
* Khi ta không khởi tạo giá trị cho nó thì ta sẽ k thể gọi nó ra và sử dụng được.
* Các hàm trong class đều có thể truy nhập giá trị của biến static này.
* Nó chia sẻ giá trị cho tất cả các object thuộc class.
* Trong phạm vi trong class thì cú pháp sử dụng biến class giống như biến bình thường khác, bên ngoài class ta có thể truy nhập biến static này thông qua object bằng toán tử .( như biến khác), hoặc không cần thông qua object mà thông qua tên class với toán tử phân giải phạm vi::.

Các **hàm static** trong class:

* Giống như các biến static bên trong class, các hàm static cũng không phụ thuộc vào đối tượng của class.
* Cho phép gọi một hàm thành viên static bằng cách sử dụng đối tượng và toán tử ".". Nhưng nên gọi các thành viên static bằng cách sử dụng tên lớp và toán tử phân giải phạm vi.
* Các hàm thành viên tĩnh chỉ được phép truy cập các thành viên dữ liệu kiểu static hoặc các hàm thành viên static khác, chúng không thể truy cập các thành viên không phải kiểu static của class.
* Nó không có con trỏ this.

**Con trỏ this:** Nó là 1 object của 1 class, bản chất là tham chiếu của 1 đối tượng thuộc class khi được tạo.

**Friend function and Friend class**

* Tại sao dùng friend: khi ta chỉ muốn chia sẻ 1 biến chỉ ở trong 1 class khác, hoặc 1 hàm khác không chia sẻ công khai ở public. (hạn chế phạm vi biến đó trong 1 số hàm)
* **Friend function** không phải 1 thành viên thuộc class do vậy con trỏ this của class k thể tham chiếu tới nó.
* Friend class: ví dụ classA là friend của classB, khi đó, trong classA ta khai báo object classB thì object này có thể truy xuất cả thành phần private của classB, **friend class không có tính 2 chiều**

**Const class object**. Chỉ truy nhập được các biến và hàm được đánh dấu là const trong class.

Buổi 8 Overloading override

Overloading

Cùng tên hàm, khác số lượng hoặc kiểu tham số ( khác kiểu trả về, k liên qua gì; vd: khác kiểu tham số, giống y hệt tham số truyền vào sẽ báo lỗi)

Overriding

Giống nhau y hệt tên hàm, số lượng tham số, và kiểu trả về chỉ xuất hiện ở class kế thừa từ 1 class khác.

**Hạn chế của overloading operator:**

* Số toán hạng của toán tử không được thay đổi.
* Không thể có 1 toán tử mới( chỉ có thể overload lại các toán tử này)
* Không thể overload lại các phép toán chuẩn ví dụ: cộng 2 số nguyên.

**Quy tắc cần tuân thì khi nạp chồng toán tử:**

* Thứ tự ưu tiên của các toán tử
* Số lượng toán hạng
* Không tạo 1 oprator, chỉ sử dụng trong danh sách hạn chế có sẵn
* Không thể overload lại các phép toán chuẩn ví dụ: cộng 2 số nguyên

Có 2 cách để nạp chồng 1 toán tử:

* viết dưới dạng 1 hàm của class
* viết dưới dạng 1 hàm global, nếu muốn hàm truy cập được private của class thì ta phải khai báo hàm này là friend của class

Ví dụ: Với toán tử cộng ta có thể thực hiện theo 2 cách là member function hoặc non member function

Phanso operator+(const Phanso &other)

{

Phanso Kq;

Kq.ts = this->ts \* other.ms + other.ts \* this->ms;

Kq.ms = other.ms \* this->ms;

return Kq;

}

Friend Phanso operator+(const Phanso &c1, const Phanso &c2)

{

Phanso Kq;

Kq.ts = c1.ts \* c2.ms + c2.ts \* c1.ms;

Kq.ms = c1.ms \* c2.ms;

return Kq;

}

Triển khai overload toán tử =

myClass& myClass::operator=(const myClass& other)

{

…

Return \*this;

}

Triển khai overload toán tử >>, <<

…

Shallow copy:

* Là quá trình copy các thuộc tính (không phải static) của 1 object đang tồn tại sang 1 object mới. nếu thuộc tính là value thì shadow copy là copy giá trị bit by bit, còn thuộc thính là reference thì thuộc tính này của 2 object sẽ cùng trỏ tới 1 vùng nhớ.
* Default copy constructor và phép gán = là shallow.
* Không nên sử dụng shallow copy khi class có thuộc tính là reference (con trỏ) vì khi đó nếu ta thay đổi hoặc hủy thuộc tính đó ở 1 object thì object còn lại cũng bị thay đổi hoặc bị hủy.

Deep copy

Là quá trình copy các thuộc tính (không phải static) của 1 object đang tồn tại sang 1 object mới, nếu thuộc tính là value thì deep copy sẽ copy giá trị bit by bit, còn thuộc thính là reference thì ta chỉ copy giá trị của thuộc tính đó.

Buoi 9 Kế thừa

Mối quan hệ giữa 2 class: “is a” và “has a”

* Kế thừa là mối quan hệ “is a”. Khi 1 đối tượng class con được tạo thì 1 đối tượng base cũng sẽ được tạo.
* Mối quan hệ “has a” thì là trong 1 class có data member là 1 class khác, nó có thể khai báo nhiều object thuộc class khác vào 1 class,

Khái niệm kế thừa:

Kế thừa là một đặc tính quan trọng trong hướng đối tượng, một class có thể kế thừa các thuộc tính và phương thức từ 1 class khác, class kế thừa class khác gọi là class con ( class derived, subclass), còn class được kế thừa gọi là class cha ( class base, superclass)

Vd: **class cha** là **class** **động vật** có các thuộc tính cân nặng, chiều cao, tên gọi lại có các **class con** khác như : **mèo, chó, gà, lơn**, kế thừa các thuộc tính chung từ **class** **động vật,** và có các thuộc tính riêng như tiếng kêu, và các hành vi riêng của từng loài.

Khi sử dụng các class kế thừa nhau thì chỉ có con trỏ lớp cha mới trỏ tới con trỏ trong lớp con được, còn ngược lại khi 1 con trỏ được khai báo là thuộc lớp con thì k thể trỏ ngược lại lớp cha.

Tại sao cần kế thừa:

**Reusable** Tăng tính tái sử dụng code🡪 Giảm công sức code

**Dễ maintain:** khi muốn sửa đối gì liên quan các thuộc tính từ class cha thì sẽ dễ dàng hơn.

Dễ dàng thiết kế hệ thống kiến trúc từ cấp cao tới cấp thấp.

Quá trình constructor và initialize

Khi construct 1 class thì nó sẽ construct 1 class bass nhất ( cha trước, con sau)

Ngược lại hàm hủy ở các class lại được thực hiện ngược lại, destructor con trước, cha sau.

Access modifier: public, protected, private trước có luôn

Overriding: trước có rồi

Hiding inherited functionality

Ta có thể thay đổi phạm vi truy cập **riêng** (**chung** thì là kiểu kế thừa là public, protected hay private) của 1 data member or function member được kế thừa từ lớp cha sang public hoặc private ở class con.

Vd mở rộng phạm vi hàm printx() từ protected sang public ở obj thuộc class kế thừa.:

classBase{

Protected:

Int x;

Printx(){…}

}

ClassDerived: public classBase{

Public:

Using classBase::Printx();

}

Vd hạn chế (hiding) phạm vi hàm printx() từ public sang private ở obj thuộc class kế thừa

classBase{

public:

Int x;

Printx(){…}

}

ClassDerived: public classBase{

pravate:

Using classBase::Printx();

}

Khi đó object thuộc class con sẽ không thể gọi hàm printx() ở hàm main được nữa

Đa kế thừa vấn đề diamond (xem bổ sung sau)

Buổi 10 Đa hình, trừu tượng

Đa hình:

Các hàm cùng tên nhưng khi gọi lại có thể thực hiện theo các các khác nhau.

* **Static**: compile time : overloading function, overloading operator
* **Dynamic**: run time: virtual ( overriding: chỉ sảy ra khi có kế thừa)

Từ khóa **final**:

* **Khi final đi với hàm**: khi redefine 1 hàm trong class con sử dụng từ khóa final thì các lớp kế thừa class con tiếp theo đằng sau này sẽ không thể redefine function này nữa, nhưng các class con cùng cấp (cùng kế thừa 1 class cha nào đó, và muốn redefine 1 hàm từ class cho đó) với class con chứa hàm với từ khóa final này vẫn redefine được.
* **Khi final đi với class**: thì các class sau đó sẽ không thể kế thừa tiếp class có từ khóa final.

Virtual function

**Vitual table**: mỗi 1 class base có các member function có từ khóa virtual đều có 1 virtual table, các class con kế thừa từ base cũng sẽ có virtual table. Khi 1 con trỏ được khai báo kiểu base trỏ tới địa chỉ của 1 đối tượng thuộc class con, thì khi nó gọi hàm được overide nó sẽ ưu tiên gọi hàm đó ở class con.

**Virtual destructor** :Khi xóa một object kiểu **derived** mà có 1 con trỏ kiểu **base** đang trỏ tới thì sẽ sảy ra vấn đề undifine behavior. Để khắc phục vấn đề này thì hàm hủy tại base phải có từ khóa virtual.

Vd: classBase{

public:

virtual Printx(){…}

}

ClassDerived: public classBase{

public:

printx(){ádfasd}

}

Int main(){

classBase \*base = new Derived; // lúc này nó sẽ gọi hàm constructor Base -> constructor Derived;

delete base; // nó chỉ gọi hàm destructor base, destructor derived sẽ k được gọi nếu hàm hủy tại base k có từ khóa virtual }

Do vậy bất kì khi nào 1 class có các class khác kế thừa từ nó thì destructor của nó phải có virtual.

Trừ tượng ( Abstraction)

**Pure virtual**: không có thân hàm (không có phần định nghĩa) , và phải có gán **=** 0.

Note: *Class kế thừa từ class cha ( trong class cha có pure virtual) thì trong class con kế thừa* ***phải bắt buộc định nghĩa (implement)*** *hàm có tên kiểu (pure virtual trong class cha)*

**Abstract class**: là 1 class chứa ít nhất 1 pure virtual function. Abstract class sẽ cung cấp các interface cho các subclass. (*pure virtual function cũng được gọi là abstract funciton*)

**Interface class** là 1 class mà tất cả các hàm đều là pure virtual.

*Note: Không thể tạo 1 đối tượng (object) từ abstract class hoặc interface class, mà chỉ có thể tạo từ các subclass.*

**Object slicing:** xảy ra khi ta gán 1 object thuộc class con cho 1 object thuộc 1 class cha, khi đó các data member thuộc riêng các class con sẽ “sliced” off and bị mất

**Ép kiểu giữa các class kế thừa: Bổ sung thêm**

Dynamic casting: upcasting, downcasting

Kiểm tra xem class có từ khóa virtual không, và kiểm tra null sau khi cast.

Buổi 11 Template

Template: khuôn mẫu giúp đơn giản hóa việc overloading.

Template function:

1 template có thể có các loại parameter khác nhau, nhưng bắt buộc phải sử dụng đủ các loại parameter khác nhau đó.

template<class T1, class T2>

T1 maxx(T1 a, T1 b){ //Error do T2 k duoc su dung

return (a > b)?a:b;

}

T2 minn(T1 a, T2 b) { //Error do khong khai bao template<class T1, class T2>

return (a > b)?a:b;

}

Template class:

template<class T1, class T2>

class mpair{

public:

mpair(T1 a, T2 b){ // Ta co the dinh nghia ham ngay trong than class

x = a; nhu sau

y = b;

}

T1 first();

T2 second(){

return y;

}

private:

T1 x;

T2 y;

};

template<class T1, class T2>

T1 mpair<T1, T2>::first(){ // trong class function member co the k can su dung du paramter

return x; // van ok, khac void function template.

}

int main(){

mpair<int, float> a(2,2.4);

mpair<float, int> b(4.4,3);

cout << a.first() << ", " << a.second() << endl;

cout << b.first() << ", " << b.second() << endl;

return 0;}

* Ta cũng có thể định nghĩa 1 function template riêng trong 1 class bình thường khác.
* Template non-type parameter:

Template<class T1, int size>

* Function Template specification: xử lý ngoại lệ với riêng 1 loại dữ liệu nào đó trong 1 template chung.

template<class T1, class T2>

class mpair{

public:

mpair(T1 a, T2 b){

x = a;

y = b;

}

void first(){

cout << x << endl;

}

void second(){

cout << y << endl;

}

private:

T1 x;

T2 y;

};

template<>

void mpair<bool, bool>::first(){

if(x != 0){

cout << "True" << endl;

}else{

cout << "False" << endl;

}

}

template<>

void mpair<bool, bool>::second(){

if(y != 0){

cout << "True" << endl;

}else{

cout << "False" << endl;

}

}

int main(){

mpair<bool, bool> a(2,2.4);

a.first();

a.second();

return 0;

}

* Ta cũng có class template specificatioin tương tự:

template <> // the following is a template class with no templated parameters

class mpair<bool,bool> // we're specializing Storage8 for bool

{…}

Buổi 12 Smart pointer, Multi thread

Smart pointer

Do con trỏ thuần ( raw pointer) thường phức tạp khi sử dụng, khó kiểm soát khi nó được cấp phát động, dễ gây ra sự memory leak nếu không được delete đúng cách.

**Memory leack**:

* 1 pointer được new nhiều lần, mà chỉ được delete 1 lần.
* khi một hàm được return về 1 con trỏ, khi gọi hàm nhiều lần để gán cho 1 biến nào đó nhưng ta chỉ delete biến đó 1 lần, thì cũng gây ra leak memory.
* Smart pointer sẽ hỗ trợ việc tự động hủy khi ta new 1 con trỏ.
* **Unique pointer:**

Cùng 1 thời điểm, mỗi 1 object (resource ) chỉ có 1 pointer trỏ tới và quản lý nó( owner).

* **Shared pointer:**
* Hoạt động tương tự như unique pointer nhưng nó cho phép có nhiều share pointer cùng trỏ tới 1 object. Object sẽ không bị delete khi tất cả các owner của nó out of scope. Được thực hiện bằng cách thêm biến **reference count** lưu trữ số owner của object đó.
* Mỗi shared ponter cùng 1 lúc cũng chỉ quản lý (owner) của 1 object. Khi new rồi gán cho object khác, thì cái cũ sẽ tự delete.
* Circular referrence: -> weak pointer.
* **Weak pointer**

Luôn lệ thuộc vào 1 shared pointer để hoạt động.

Program, Process, Thread

Process: thực thi trên RAM(code segment, data segment, bss(data uninitial), heap segment, stack segment.

Thread: 1 đoạn code (logic) có thể chạy song song với 1 đoạn code khác(logic). Một process có thể có nhiều thread khác nhau. Các thread trong 1 process có thể share memory cho nhau.

**So sánh thread và process.?**

*Ưu điêm thread so với process:*

* Thread nhanh hơn hơn khi chuyển qua lại giữa các thread, thực thi nhanh hơn.
* Độ phức tạp thread thấp hơn process.
* Sharing giữa các resource.

*Data race.: mutex, auto…, Deadlock,*

Buổi 13 Cấu trúc dữ liệu và giải thuật

Cấu trúc dữ liệu

Cấu trúc dữ liệu là cách lưu trữ, tổ chức dữ liệu có thứ tự, có hệ thống để dữ liệu có thể được sử dụng một cách hiệu quả.

Dữ liệu và thông tin:

* Dữ liệu là các số liệu thô chưa được tinh chỉnh, mơ hồ, không mang bất kì ý nghĩa nào, được lưu trữ trong máy tính với đơn vị byte/bit
* Thông tin là đầu ra của dữ liệu được xử lý, nó dựa vào dữ liệu. mang theo ý nghĩa riêng biệt

Array và vector:

* Array (mảng tĩnh): các phần tử cùng loại dữ liệu, được lưu trữ liên tiếp trong các ô nhớ và có số phần tử cố định
* Vector(mảng động): khá giống với array nhưng nó có kích thước tùy chỉnh dễ dàng. Đặc điểm nó có size và maxsize. *Khi add thêm phần tử tới maxsize+1 thì nó sẽ tự động clone vector này sang 1 vùng nhớ khác có 1 maxsize lớn hơn.*

Danh sách liên kết:

* Tổ chức bởi các node, mỗi 1 node sẽ có có data và next(next là con trỏ trỏ tới phần tử tiếp theo), để xác định 1 links list thì cần có phần tử đầu và phần tử cuối.
* Danh sách liên kết đơn:
* Danh sách liên kết đôi:
* Đánh giá: Truy xuất chậm hơn so với array, vector, nhưng nó ưu điểm tốt hơn trong việc **insert, delete** 1 phần tử nào đó ở vị trí bất kỳ.

Stack:

* Thực hiện theo cơ chế LIFO, xếp chồng data lên, data trên cùng(phần tử mới thêm sau cùng) sẽ được lấy ra đầu tiên.

Queue:

* Thực hiện theo cơ chế FIFO, có các cơ chế như: enqueue, dequeue, (peek, isEmpty, isFull),

Hask table:

* Là cấu trúc dữ liệu lưu trữ data theo key và value, mỗi phần tử trong hask table là các cặp key và value, thường được sử dụng trong việc tìm kiếm.
* Có 2 cách khác nhau để mở rộng 1 hask table: mở rộng size, mở rộng data thành 1 list

Tree

Mục đích: giảm độ phức tạp của việc tìm kiếm

Các giải thuật khi duyệt các phần tử của tree

In-oder traversal

Pre-oder traversal

Post-oder traversal

Heap

Graph

Cấu trúc dữ liệu STL của C++ ( chỉ việc sử dụng thư viện có sẵn)

#include<vector>

* + Iterator:
  + Const iterator:
* Khi gán = giữa 2 đối tượng vector thì nó mặc định là shallow copy, nhưng khi đối tượng được gán thay đổi thì nó sẽ tự động chuyển sang deep copy. Khi gọi const iterator thì đối tượng đó sẽ không bị thay đổi nên vẫn sẽ giữ là shallow copy.

#include<stack>

#include<list> // khi sử dụng iterator duyệt list thì k thay đổi thêm, xóa list

#include<map>

Buổi 14 Thuật toán

Thuật toán so sánh

Mục đích: sau khi sắp xếp thì việc tính toán hoặc tìm kiếm trên danh sách dễ dàng hơn.

Các điểm để đánh giá thuật toán:

* Thời gian
* Bộ nhớ: tránh sử dụng đệ quy(do đệ quy sử dụng nhiều bộ nhớ)
* Độ ổn định: Dựa trên sự thay đổi vị trí giữa các phần tử giống nhau.

Các loại thuật toán sắp xếp:

* Bubble sort:
* Selection sort:
* Insertion sort:
* Merge sort:
* Quick sort:

Bubble sort: Xét lần lượt các sặp phần tử liên tiếp, nếu sau lớn hơn trước thì swap. Vd:

Selection sort: độ phức tạp giống với buble sort nhưng số lần swap it hơn.vd:

Insertion sort: nếu danh sách gần đúng rồi thì nó sẽ chạy rất nhanh(ứng dụng trong các trường hợp như bản xếp hạng, ít biến động thứ tự các phần tử)

Merge sort: ưu điểm: chạy nhanh, độ phức tạp O(Nlog(N)). Nhược điểm: tốn bộ nhớ.

Quick sort:

Search

Linear: tuyến tính

Binary: tìm kiếm nhị phân

Chỉ sử dụng khi Mảng đã được sort rồi.

Thư viện có sẵn std

#include<algorithm>

bubble sort

**ý tưởng:**

* xét lần lượt các cặp 2 phần tử liên tiếp
* nếu phần tử đứng sau nhỏ hơn đứng trước ta đổi chỗ 2 phần tử (nói cách khác, phần tử nhỏ nhất sẽ nỗi lên trên)
* lặp lại đến khi không còn 2 phần tử nào thỏa mãn

**ưu điểm:**

* code đơn giản, dễ hiểu
* không tốn thêm bộ nhớ do đệ quy

**nhược điểm:** độ phức tạp O(N^2) không đủ nhanh với dữ liệu lớn.

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

for(int j = 0; j < n - 1 -i; j++){

if(a[j] > a[j+1]){

swap(a[i], a[j]);

}

}

selection sort:

**ý tưởng**:

* chia mảng thành 2 mảng con, mảng đã sắ xếp và chưa sắp xếp
* tại mỗi lần lặp thuật toán, phần tử nhỏ nhất của mảng con chưa sắp xếp sẽ được chuyển về đoạn đã sắp xếp

**ưu điểm**: code đơn giản dễ hiểu

**nhược điểm**: độ phức tạp O(N^2), không đủ nhanh với dẽ liệu lớn

void selectionSort(int a[], int n){

int i, j , min\_idx;

//di chuyển ranh giới của mảng đã sắp xếp và chư sắp xếp

for(i = 0; i < n- 1; i++){

//Tim phan tu nho nhat mang chua sap xep

min\_idx = i;

for(j = i + 1; j < n; j++){

if(a[j] < a[min\_idx])

min\_idx = j;

//Đổi chỗ phần tử nhỏ nhất với phần tử đầu tiên

swap(arr[min\_idx],arr[i]]);

}

}

Insertion sort

**Ý tưởng:**

* ta sẽ sắp xếp lần lượt từng đoạn gồm 1 phần tử đầu tiên, 2 phần tử đầu tiên, .. N phần tử
* giả sử ta đã sắp xếp xong i phần tử của mảng. Để sắp xếp i +1 phần tử đầu tiên ta tìm giá trị phù hợp với phần tử thứ i+1 và chèn vào đó

**ưu điểm:** nếu danh sách đã gần đúng thứ tự thì sắp xếp chèn sẽ chạt nhât nhanh ( ví dụ sắp xếp higscore trong game)

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

//Tìm vị trí phù hợp cho i

int j = i;

while(j > 0 && data[i] < data[j-1]) --j;

//Đưa i về đúng vị trí

int temp = data[i];

for(int k = i; k > j; k--){

data[k] = data[k-1];

}

data[j] = temp;

}

quick sort

**ý tưởng:**

* chia dãy thành 2 phần, phần lớn và phần nhỏ
* chọn 1 khóa pivot
* nhwnwngx phần tử lớn hơn pivot chia vào phần lớn
* những phần tử nhỏ hơn pivot chia vào phần nhỏ
* gọi đệ quy sắp xếp 2 phần

**ưu điểm:** chạy nhanh

**nhược điểm:**

* tùy thuộc vào cách chia 2 phần mà độ phức tạp của thuật toán khác nhau
* không ổn định

void quickSort(int a[], int left, int right){

int i = left, j = right;

int pivot = a[left+rand()%(right-left)];

//chia dãy thành 2 phần

while(i <- j){

while(a[i] < pivot) ++i;

while(a[j] > pivot) --j;

if(i <=j){

swap(a[i], a[j]);

++i;

--j;

}

}

//Gọi đệ quy để sắp xép các nửa

if(left < j) quickSort(a, left, j);

if(i < right) quickSort(a,i,right);

}

Search

linear search:

binary search;

CnU

black box test:

test chỉ quan tâm với đầu vào và đầu ra không quan tâm chi tiết thực hiện bên trong code.

unit test

integrated test: kết hợp các unit test để test

system test: test toàn bộ tính năng

acceptant test: người dùng, khách hàng tự dùng test.

các thành phần của test case

test case id

mô tả test case

thủ tục test ( test step)

expected output

white box

kiến trúc:

statement testing

branch testing

path coverage

Bảo mật thông tin, quy trình phát triển phần mềm

RnD:

Quy trinh phat trien du an

phân tích yêu cầu

phân tích thiết kế (UML)

code

test

UML: chuẩn ngôn ngữ mô tả thiết kế

kiến trúc UML:

dạng kiến trúc: class diagram

dạng hành vi: sequence diagram

use case diagram gồm:

actor (hình người)

use-case (hành vi - hình vòng elip)

mối quan hệ (mũi tên):

giữa các actor (kế thừa: dùng mũi tên tam giác trắng)

bao gồm: include

mở rộng: extend

khái quát/chi tiết: generalization/specification

class diagram:

public: +

protected: #

private: -