1. **What is Object-Oriented Programming? Lập trình hướng đối tượng là gì?** 
   1. **Procedural programming**: Lập trình hướng thủ tục

* Tập trung xây dựng các function và hầu hết sử dụng function để thực hiện các chức năng bằng cách truyền các tham số vào hàm
* Giới hạn của lập trình thủ tục:

+ Function phải biết structure of data.

+ Khi chương trình trở nên lớn và phức tạp hơn thì rất khó để hiểu, maintain, debug, reuse code

* 1. **What is Object-Oriented Programming**?

+ Classes and Objects

+ Encapsulation: Tính đóng gói – Dấu thông tin

+ Reusability: Tính tái sử dụng cao

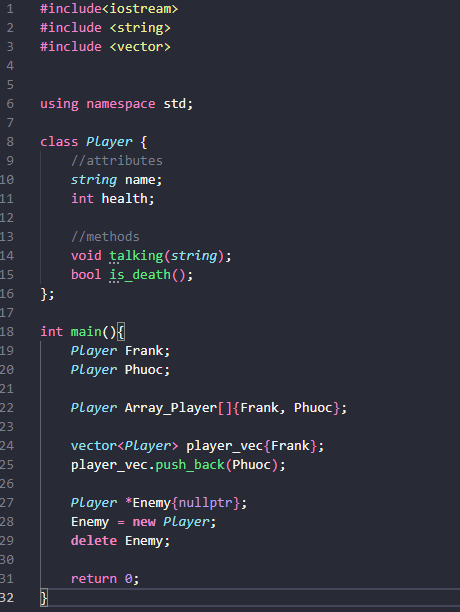
+ Inheritance: Tính kế thừa

+ Polymorphism: Tính đa hình

1. **Classes and Objects**
   1. **Classes:**

* Class là các bản vẽ để tạo ra Object
* Bao gồm **Attributes**( data) và **methods**(functions)
* Có thể hide data và methods
* Provide public interface
  1. **Objects:**
* Được tạo ra từ class
* Represent a specific instance of class: Đại diện cho một trường hợp cụ thể của class.
* Có thể tạo nhiều object từ 1 class
* Có thể sử dụng đượ các methods được defined trong class.

1. **Declare Classes and Objects:**

* 
* Khi ta viết Player(Nam,90) thì đây là 1 R value nha, vì không có tên cho nó.

1. **Accessing Class Members**

* Ta có thể access: **Class attributes** và **Class Methods**
* Sử dụng **dot operator “.”** Ví dụ: Frank.name; (\*Enemy).name;
* Sử dụng **arrow operator** “**->**” Ví dụ: Enemy->name = (\*Enemy).name

1. **Access Modifier**

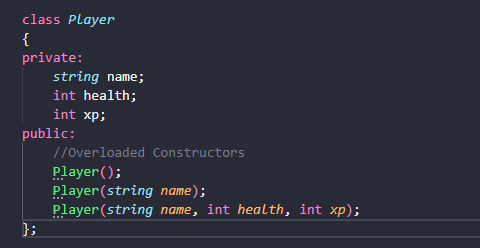
* Gồm 3 loại:
  + **Public**: accessible everywhere
  + **Private:** accessible bởi **members** của class hoặc **friends** của class
  + **Protected**: Giống private và cho phép lớp kế thừa truy cập nữa.

1. **Implementing Member Methods:** Triển khai các methods

* Giống như các triển khai function bình thường
* Ta không cần truyền các **attributes** của class vào methods
* Có thể implement ở trong hoặc ngoài class ( Ngoài class thì use “**::**”trước tên của function)
* Có thể declare class ở 1 file.h, sau đó trong file.c ta implement method, chỉ cần include file.h và use toán tử phân giải phạm vi “**::”.** Để sử dụng function trong main ta include file.h vào file main.cpp là ok

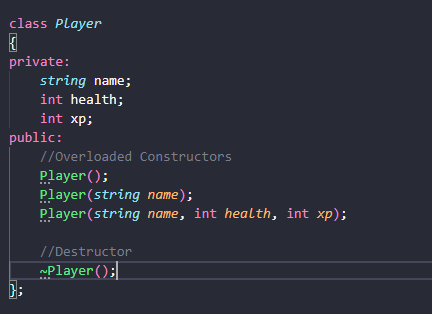
1. **Constructor and Destructor:**
   1. **Constructor and Destructor**
2. **Constructor**

* Là một member method đặc biệt **được gọi khi một object được tạo**.
* Có tên trùng với tên Class và rất hữu dụng trong việc khởi tạo các giá trị cho object.
* Không return gì cả và có thể **overload**

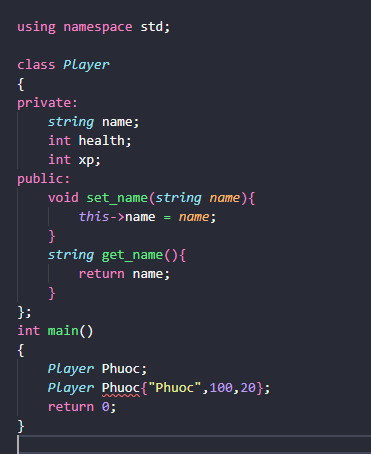
****

1. **Destructor:**

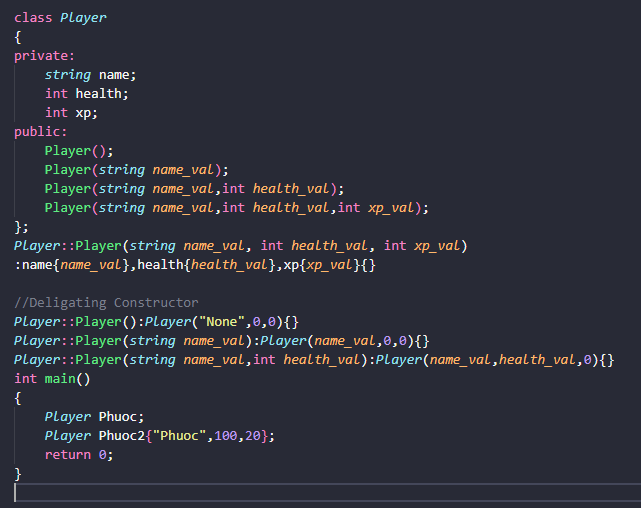
* Là một member method đặc biệt, có tên trùng với tên Class nhưng có dấu ~ đằng trước.
* Được gọi tự động khi 1 object out scope hay object bị destroyed.
* Không có return type và parameter
* Chỉ có duy nhất 1 destructor và không thể overload.
* Hữu dụng trong việc giải phóng bô nhớ và tài nguyên.

****

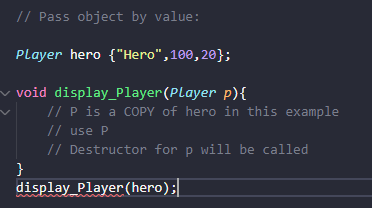
* 1. **Default Constructor**
* A default constructor is a constructor that doesn’t take any argument. It has no parameters. It is also called a zero-argument constructor.
* The compiler automatically creates an implicit default constructor if the programmer does not define one.
* Nói chung là khi tạo 1 constructor không đối số, hoặc để complier tạo thì chỉ có thể khai báo object không có tham số mà thôi, nếu không là lỗi ngay

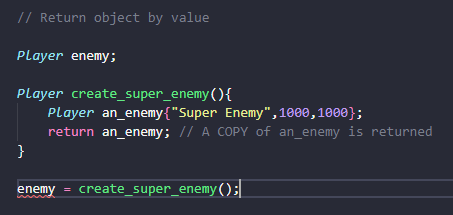


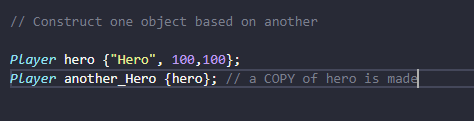
* 1. **Overloading Constructor**
* 1 class có thể có nhiều constructor nếu cần thiết
* Dựa vào thông tin lúc create object, constructor tương ứng sẽ được gọi. Nếu có sự mơ hồ nào đó, compiler sẽ không đoán mà báo lỗi luôn.
  1. **Constructor Initialization lists**
* Thay vì khởi tạo giá trị của các data trong thân hàm constructor, ta có thể sử dụng initialization list
* Cách này hiệu suất hơn vì trước khi thực hiện body thì các data đã đươc khởi tạo xong rồi.
* ****
  1. **Delegating Constructors**
* Để tránh duplicated code, từ 1 constructor ta có thể call một constructor khác trong **initialization list**.
* Bản chất của việc này là một constructor gọi 1 constructor khác và tất nhiên các lệnh trong body của constructor được gọi cũng được thực hiện

****

* 1. **Constructor Parameters and Default Values**
* Sử dụng default values là phương pháp hiệu quả để một constructor có thể đảm nhận nhiều kiểu object được tạo.
* Trường hợp dưới đây thì constructor với 3 arguments được gọi 3 lần
* ****
* **Lưu ý:** Khi dùng default values ta không thể sử dụng các Constructor khác nếu không sẽ dẫn đến lỗi complier
  1. **Copy Constructor**
* Khi objects được copy thì C++ phải tạo ra một object mới từ 1 object có sẵn
* Các trường hợp xảy ra **copy constructor**:
  + Pass object vào by value như là một tham số vào hàm
  + Return một object by value từ 1 hàm
  + Tạo một object mới từ 1 object đã tồn tại trong cùng 1 class

****

****

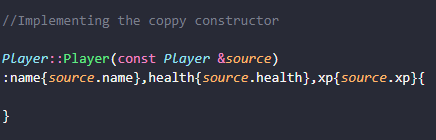
****

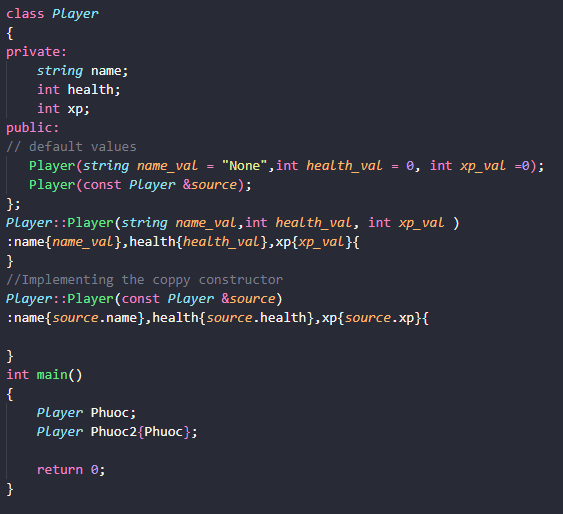
* Nếu không tự viết 1 copy constructor thì complier sẽ tự động tạo 1 default copy constructor. Nói chung ngoại trừ tường hợp sử dụng raw pointer ra thì dùng default copy cũng ok
* Khi có raw pointer thì copy constructor sẽ chỉ copy pointer chứ không phải copy giá trị nó đang trỏ tới.
  + **Declaring the Copy constructor:**

**Type ( const Type &source);**

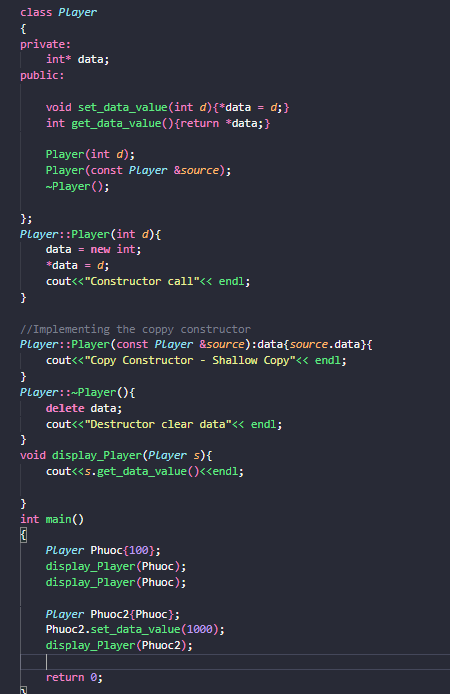
****

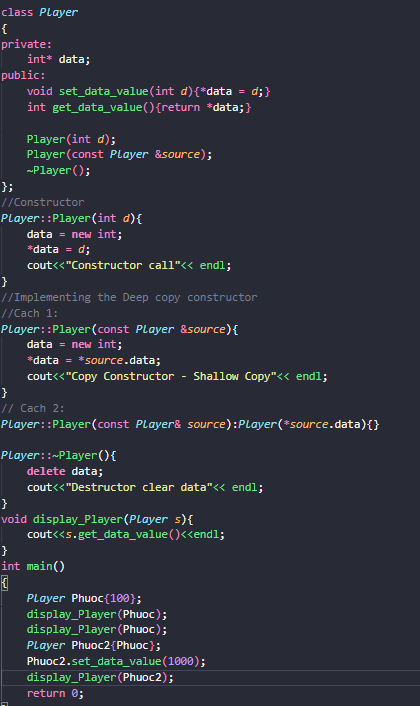
* Trong đó sử dụng tham chiếu **&source** vì nếu truyền tham trị thì ta sẽ tạo 1 copy khác của source, copy constructor sẽ bị gọi liên tục.
* Dùng **const** ở đây có nghĩa là ta không modify source
  + **Implementing the Copy constructor**

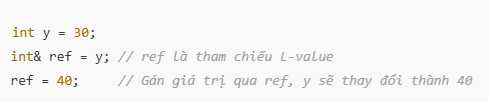


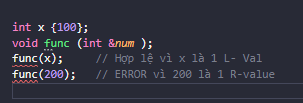
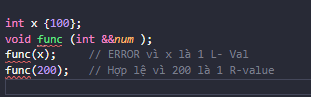
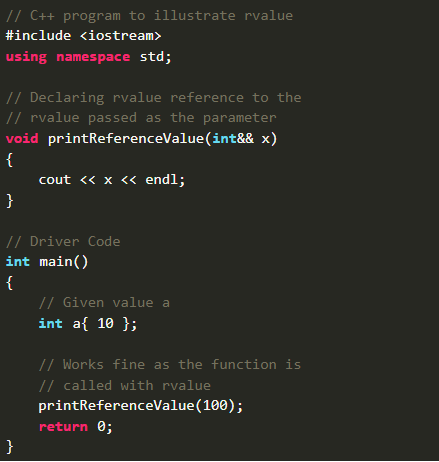


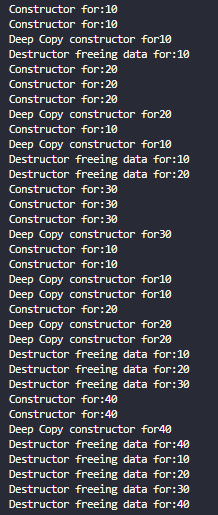
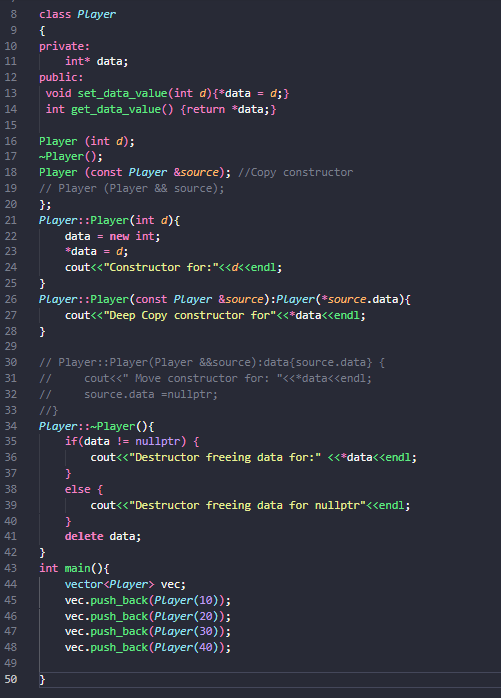
* 1. **Shallow Copy with the Copy Constructor**
* Nói cơ bản thì Shallow copy là ta chỉ copy giá trị của pointer chứ không copy giá trị mà nó đang trỏ tới.
* Điều này có nghĩa là 2 pointer đang cùng trỏ về 1 vùng nhớ, sẽ dẫn đến việc khi ta giải phóng bộ nhớ ở destructor thì object còn lại sẽ trỏ đến một vùng nhớ đã bị giải phóng -> Lỗi

****

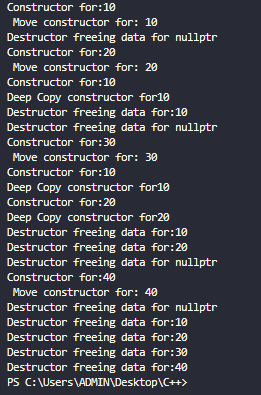
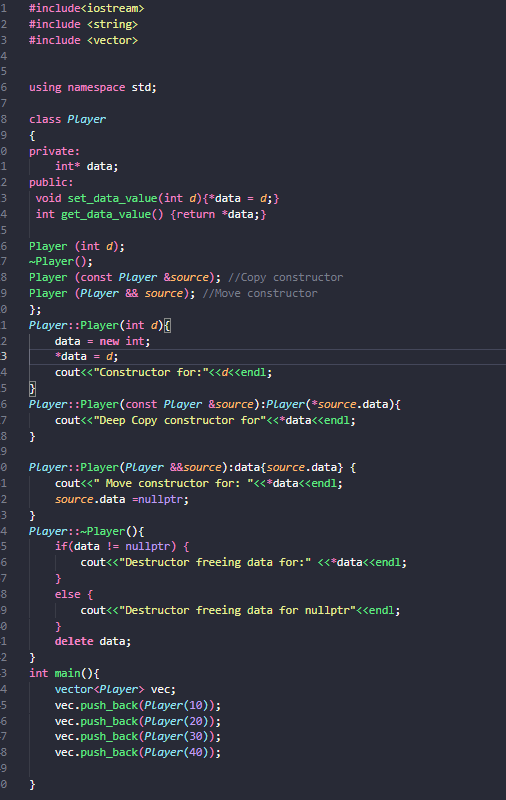
* Trong trường hợp trên, có 4 lần copy constructor được thực hiện. Lần đầu là khi gọi hàm **display\_Player()**, hàm này pass by value. Nghĩa là 1 object s được copy từ object Phuoc, tuy nhiên đây là shallow copy nên 2 raw pointer của 2 object đều đang trỏ đến 1 vùng nhớ. Sau khi kết thúc gọi hàm thì Destructor của s được gọi, giải phóng vùng nhớ s đang trỏ đến. Lúc này pointer trong Phuoc sẽ trỏ đến một vùng nhớ không hợp lệ. Do đó khi ta gọi hàm display\_Player() lần thứ 2 sẽ bị lỗi.
* Sau đó khi ta tạo Phuoc2 từ Phuoc thì bản chất lại shallow copy 1 lần nữa, nhưng object Phuoc chứa pointer đang trỏ đến 1 vùng invalid nên cũng sẽ bị lỗi.
  1. **Deep Copy with the Copy Constructor**
* Deep copy không chỉ copy the pointer mà còn copy data mà pointer trỏ đến. mỗi copy sẽ có storage độc nhất trên heap.
* Sử dụng deep copy khi có raw pointer là thành viên của class.
* ****
* Ta có thể triển khai Deep copy constructor theo 2 cách như trên, ở ví dụ tương tự này, khi ta gọi hàm display\_Player, copy constructor được gọi nhưng ở đây sẽ không xảy ra hiện tượng như trước vì object s được copy từ Phuoc nhưng nó trỏ đến 1 vùng nhớ hoàn toàn khác, vì vậy kgi kết thúc gọi hàm, s bị free không làm ảnh hưởng gì đến vùng nhớ mà con trỏ trong Phuoc đang trỏ tới cả, nên không có lỗi xảy ra.
* Tương tự khi tạo Phuoc2 từ Phuoc thì giá trị của Phuoc 2 lúc này sẽ giống Phuoc, tuy nhiên khi ta thay đổi data mà con trỏ trong Phuoc2 thì không ảnh hưởng tới data mà con trỏ trong Phuoc trỏ tới.
  1. **Move Constructor \*\*\***
* Khái niệm về L-value và R-Value:
  + **L – Value**:
    - Là value có name và **addressable**: nghĩa là có thể truy cập địa chỉ
    - Có thể **modifiable** nếu không phải là constant
    - **Tham chiếu L-Value:** Dùng để tham chiếu đến một L-Value

****

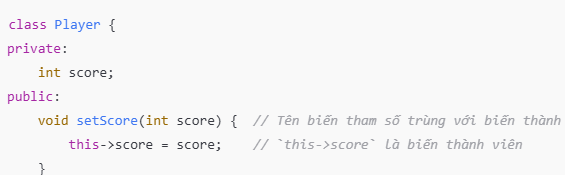
* + **R-Value:** 
    - Là value **non-addressable**: không thể truy cập địa chỉ và **non-asignable** : không thể gán.
    - Nằm bên phải toán tử gán
    - Là 1 literal
    - Là 1 giá trị tạm thời **non-modifiable**
    - Dùng std::move() để ép buộc một đối tượng trở thành R-value.
    - **Tham chiếu R -Value**: sử dụng **&&**
      * ****
      * ****
      * ****
* **Usage:**
  + + When **copy constructors** are **called many times** automatically due to the copy semantics of C++
  + + Copy constructors doing deep copying can **cause bottle neck**
  + -> Move semantics & Move constructor: **Move object** rather than copy
  + + **Recommended** when you have **a raw pointer**
  + + Copy elision - C++ may optimize copying away completely (Return Value Optimization - RVO)
  + Ta nói qua về việc thắt nút cổ chai: Khi một đối tượng được sao chép bằng copy constructor, nếu đối tượng gốc chứa dữ liệu lớn (mảng, container) hay tài nguyên động thì việc copy rất phức tạp và tốn thời gian. Nếu copy chỉ để dùng tạm thời rồi xóa thì quá không cần thiết, vậy nên ta dùng **Move semantics**
* A move constructor is a special type of constructor in C++ that is used to create a new object from the already existing object of the same type, but instead of making a copy of it, it makes the new object point to the already existing object in the memory, leaving the source object in a valid but unspecified state.
* Lý do sử dụng:   
  - việc sao chép các đối tượng lớn hoặc phức tạp như vector, string, hoặc tài nguyên động gây tốn kém tài nguyên, do đó thay vì copy nó thì ta di chuyển nó.
* **Bản chất của Move constructor:** 
  + Ta chuyển quyền sở hữu tài nguyên từ đối tượng cũ sang đối tượng mới sau đó vô hiệu hóa con trỏ đối tượng cũ.
  + Khi ta sử dụng move constructor, vd: **Class1 object1 = move(object2);** ( std::move nếu chưa dùng namespace), thì ta đang chuyển object2 từ L value sang R value, hàm move sẽ trả về 1 R Value Reference cho đối tượng đó. Sau đó Move constructor sẽ được gọi.
  + Ta cũng có việc **Class1(100) là 1 R-value**
* **Cú pháp:** 
  + Class :: Class ( Class &&source)
    - Ta có thể thấy thì cú pháp khá giống copy constructor, tuy nhiên có 2 điểm khác:
      * **Không có từ khóa const**: vì chúng ta phải modify giá trị này về Null
      * && : R value reference

****

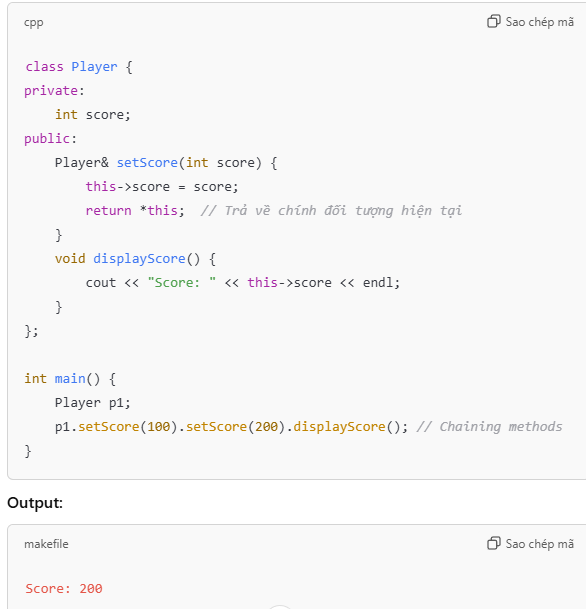
* Đọc đoạn code trên ta phải nhìn ra các vấn đề sau:
  + Copy constructor ở đây là deep copy vì nó copy cả data và raw pointer trỏ đến
  + Chúng ta đang comment Move constructor, vì khi có cả Move và Copy constructor thì khi tạo R value Player(10) thì sẽ ưu tiên gọi Move constructor. Ở đây cũng chưa hiểu vì sao R value lại truyền tđược vào Copy constructor.
  + Ta thấy Constructor được gọi liên tục, lý do là vì khi truyền 1 đối tượng vào hàm ta sẽ tạo 1 đối tượng copy, chưa hết: do vector sử dụng cơ chế mở rộng dung lượng, khi không gian hiện tại không đủ, nó sẽ tạo vùng nhớ mới với dung lượng lớn hơn, sao chép toàn bộ phần tử cũ sang vùng mới và hủy vùng cũ -> constructor và copy constructor bị gọi nhiều lần.

****

* **Vì Player(10) là 1 Rvalue nên khi định nghĩa Move constructor thì nó sẽ ưu tiên được gọi.** 
  1. **The “This” Pointer**
* Là 1 con trỏ đặc biệt, **trỏ đến đối tượng hiện tại mà một thành viên ( biến hoặc hàm) đang được gọi,** nói đơn giản thì khi mình tạo 1 object và sử dụng biến hoặc hàm trong class của object đó thì this pointer sẽ trỏ đến object này.
* Lưu ý rằng this pointer **chỉ xuất hiện trong các hàm thành viên (method) của class** mà thôi. Không xuất hiện bên ngoài.
* This pointer có 2 công dụng chính:
  + Phận biệt giữa biến thành viên và tham số của method:

****

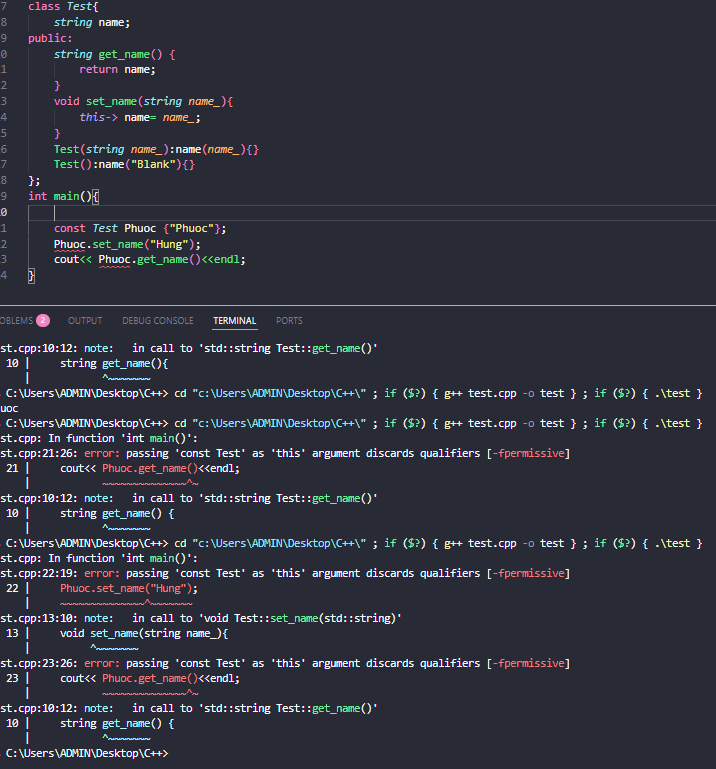
* + Trả về chính object hiện tại ( method chaining)

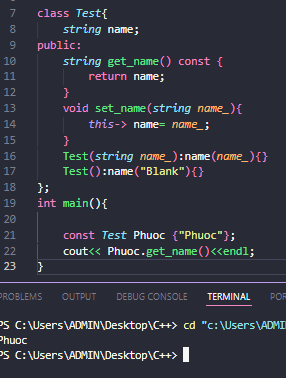
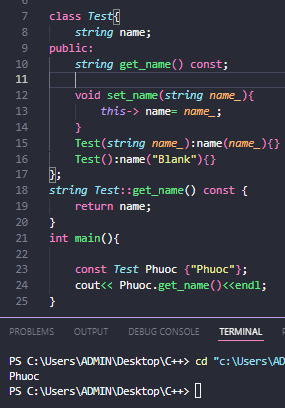
****

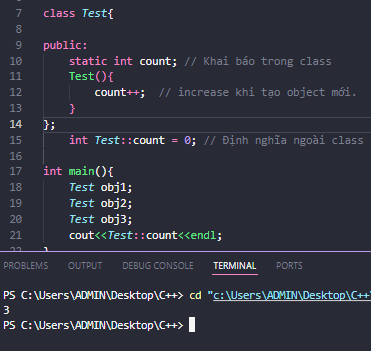
* 1. **Using Const with Classes**
* Có 2 kiểu dùng const, 1 là kiểu truyền **tham số vào function của class dưới dạng const** -> Không thể thay đổi tham số truyền vào này bên trong function.
* Ở đây ta sẽ tìm hiểu về kiểu thứ 2: **Const object**
  + Const object là như thế nào
  + Gọi 1 method của 1 const object
* Format:
  + **Const Player villain {"Villain", 100, 55};**

-> Cannot change the content of attributes after intialize.  
Nói chung là có 2 điểm cần lưu ý:  
- Phải khởi tạo ngay khi khai báo. Xử lý ở constructor làm sao khi khai báo object kiểu gì thì cũng phải có giá trị cho nói rồi.

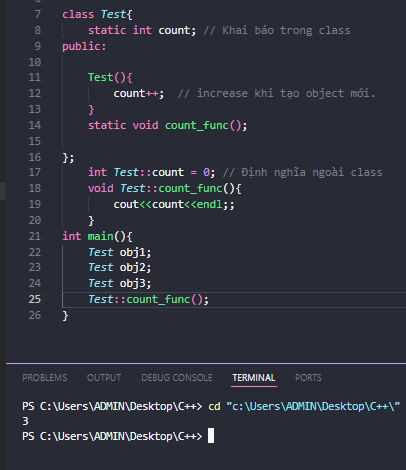
- Cannot change the content of attributes after intialize. Không thể thay đổi nội dung của data sau khi khởi tạo object.

- 

* Vấn đề tiếp theo là không thể gọi được method của const object vì compiler sẽ hiểu là đang có thay đổi nội dung của const object. Tuy nhiên trên thực tế thì chỉ có hàm set\_name() mới ảnh hưởng đến nội dung, chứ hàm get\_name() thì đéo. Vậy làm sao để gọi được method của const object mà không lỗi:   
    
  - Câu trả lời là ta thêm từ khóa “**const**” vào sau khai báo va định nghĩa method mà mình muốn gọi. Và tất nhiên là method này không làm thay đổi nội dung thật, chứ có thay đổi thì cũng cút.   
  
  1. **Static Class Members**
* Ta sẽ nói về static data member và static method
* **Static data member:**
  + Biến static thuộc về class, mọi object của class dùng chung biến này.
  + Không cần object để truy cập, truy cập thẳng từ class: Classname::data\_name
  + Khai báo trong lớp nhưng định nghĩa ngoài lớp.
  + Các object chia sẻ cùng 1 biến static.

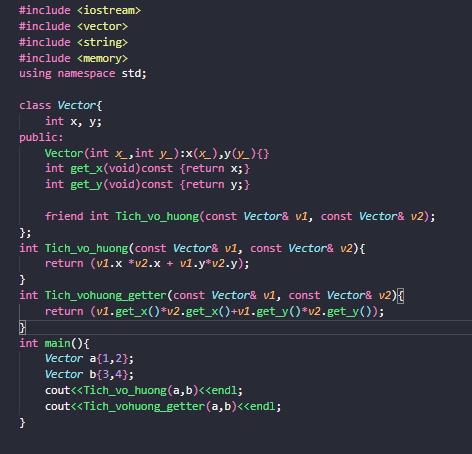
****

* **Static method:**
  + Không có con trỏ this nên không thể truy cập thành viên **Không static** của class.
  + Có thể gọi mà không cần đối tượng.
  + **Chỉ có thể truy cập biến static.**
  + Khai báo trong class, định nghĩa trong hay ngoài đều ok.

****

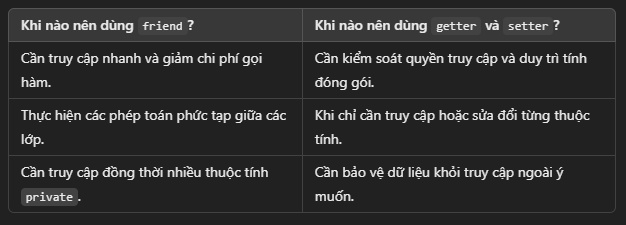
* 1. **Structs vs Classes**
* Điểm khác biệt duy nhất là ở struct thì access modifier mặc định là public, ở class là private, còn lại về thực tế là không khác gì nhau.
* **struct**: Mặc định public, kế thừa public, thường dùng cho dữ liệu đơn giản.
* **class**: Mặc định private, kế thừa private, dùng khi cần đóng gói dữ liệu và các tính năng phức tạp.

Ngoài sự khác biệt về mặc định, không có sự khác biệt về tính năng và khả năng sử dụng giữa struct và class trong C++.

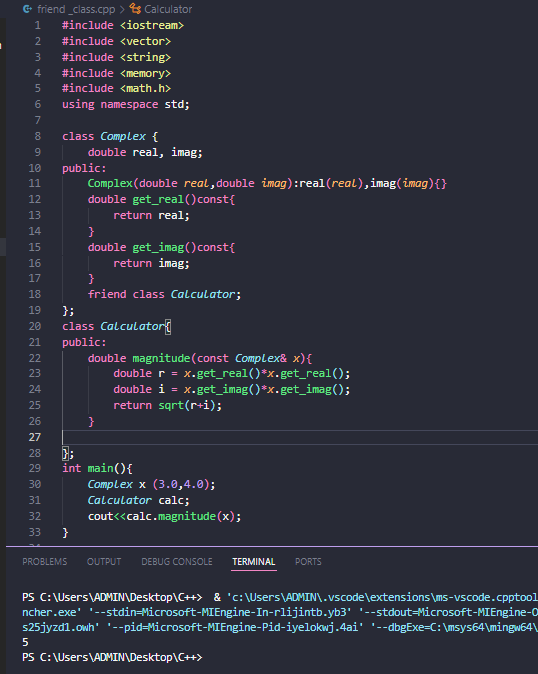
* 1. **Friends of a Class**
* **Friend function** của 1 class là 1 fucntion có quyền truy cập vào data private và protected của class. Hàm này không phải là thành viên của classm declare trong class, define bên ngoài class.
* **Friend class** là class mà object của nó có thể truy cập và các data private hoặc protected của class khác.
* Nói một cách đơn giản thì friend function hay friend class là các đơn vị có thể truy cập trực tiếp vào các data private và protect của 1 class.
* Câu hỏi đặt ra là sao không dùng getter, setter mà phải dùng friend, thì thật ra là tùy trường hợp, dùng getter setter sẽ tốn hiệu năng hơn vì phải thông qua 1 class trung gian nữa, giảm chi phí gọi hàm. Xem các ví dụ sau:
* 
* Đây là 1 trường hợp đơn giản khi ta muốn tính tích vô hướng của 2 vector. Tuy nhiên nếu dùng getter thì chi phí gọi hàm rất lớn, khi dùng nhiều lần, vậy nên mới đẻ ra friend function.

 gọi hai hàm getX() và getY(), tạo ra thêm chi phí gọi hàm.

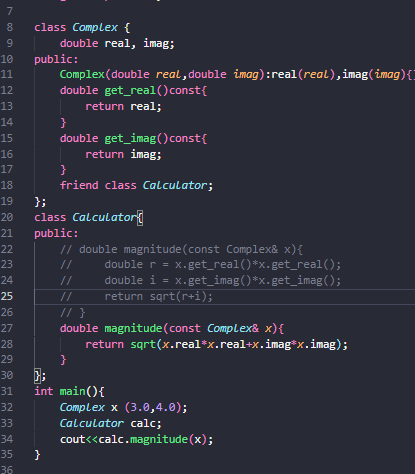
 friend truy cập trực tiếp vào private, giúp **tăng hiệu năng** trong các tác vụ tính toán nặng.



* Giờ ta qua có bài toán giữa 2 class như sau:



Đây là cách ta dùng getter, nói chung là để method của class Calculator có thể truy cập vào thành viên private thì ta dùng getter, tuy nhiên thì khá tốn chi phí gọi hàm.



* Khi ta dùng friend class thì nó như này, nhanh hơn vì truy cập luôn mà không tốn chi phi gọi hàm.

**II. Operator overloading**

* Trong C++, các toán tử (operators) như +, -, \*, /, ==, <<, >> thường được sử dụng với các kiểu dữ liệu cơ bản như int, float, double. Tuy nhiên, khi chúng ta làm việc với các lớp tự định nghĩa (user-defined classes), các toán tử này không hoạt động theo cách mặc định mà chúng ta mong muốn.
* Ví dụ:
* Nếu bạn có một lớp Complex để biểu diễn số phức và muốn cộng hai đối tượng Complex bằng cách dùng + như với kiểu int thì mặc định sẽ không thể thực hiện được.
* Khi đó, chúng ta cần nạp chồng toán tử (operator overloading) để xác định cách thức hoạt động của toán tử + với lớp Complex.

**Using function:**

Result = multiply\*=(add(a,b),divide(c,d));

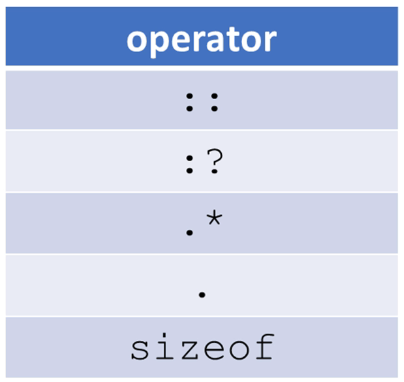
**Using member method:**

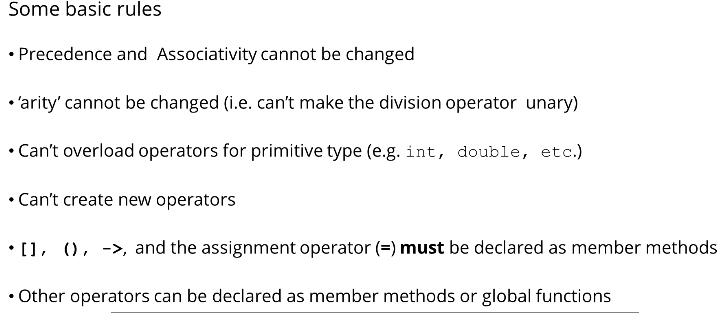
Result = (a.add(b)).multiply(c.divide(d));

**Using overloading operator:**

Result = (a+b)\*(c/d);

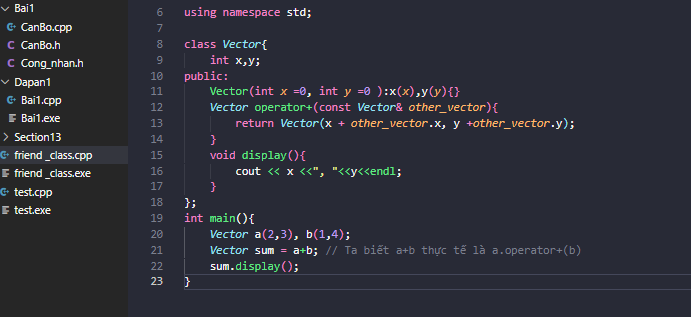
* Ta có thể nạp chồng tất cả các toán tử ngoại trừ:





* Các rules cơ bản là như sau:
  + Không thể thay đổi thứ tự ưu tiên và liên kết của toán tử: Nhân chia trước cộng trừ sau
  + Không thể thay đổi số lượng toán hạng mặc định của toán tử đó: + vẫn 2 toán hạng, toán tử 3 ngôi vẫn cần 3 toán hạng
  + Không thể overload cho các kiểu dữ liệu nguyên thủy
  + Không tạo toán tử mới
  + Các toán tử = , [], (), -> phải được khai báo là thành viên của lớp.

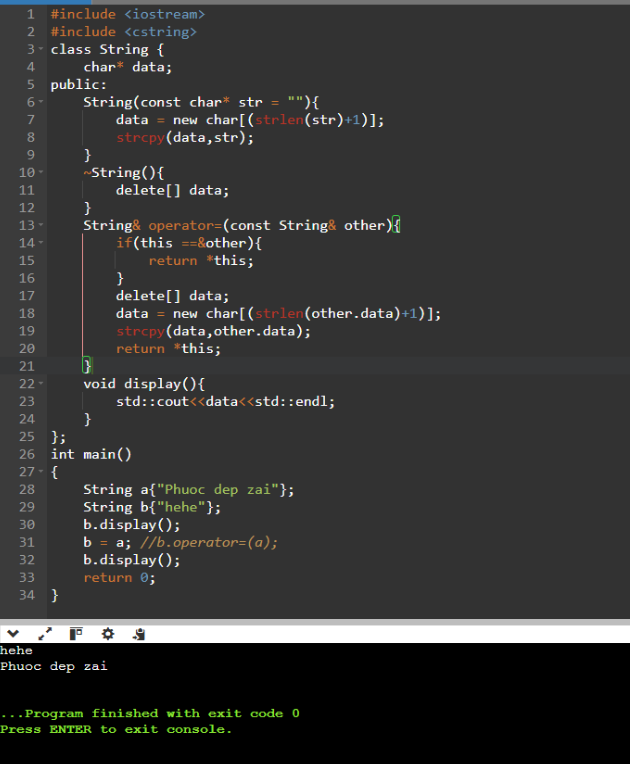
1. **Ta code thử 1 ví dụ về operator +:**

****

* Sau khi viết lại hàm operator+, thì khi ta viêt **a+b** thì thực ra nó sẽ là: **a.operator+(b)**

1. **Overloading the Assignment Operator (Copy)=**

* Toán tử gán = được sử dụng để sao chép giá trị từ 1 đối tượng này sang 1 đối tượng khác. Khi không định nghĩa toán tử gán thì trình biên dịch sẽ tự động sử dụng toán tử gán mặc định (shallow copy), tuy nhiên thì nếu data có con trỏ thì sẽ rất nguy hiểm. Ta nên tự định nghĩa việc overload để deep copy.
* Ta triển khai như sau:
  + Xử lý tự gán: Nếu this và other cùng là 1 đối tượng (kiểm tra giá trị 2 pointer)
  + Xóa dữ liệu cũ: Nếu đối tượng đã có dữ liệu trước đó, phải giải phóng bộ nhớ tránh rò rỉ
  + Sao chép dữ liệu từ đối tượng khác
  + Trả về **\*this** để cho phép phép gán liên tiếp.

****

2. Overloading the Assignment Operator (Move)

-  Khi b =a -> Vấn đề khi dùng Overloading the Assignment Operator (copy) là ta giải phóng bộ nhớ của b, cấp phát bộ nhớ mới cho b, rồi copy data từ a sang b. Đồng nghĩa với các object lớn, nhiều dữ liệu thì khá tốn chi phí. Nếu ta gán b= a mà sau đó a không còn được sử dụng, thì nên sử dụng Overloading the Assignment Operator (Move) để tránh tốn chi phí vì tái sử dụng bộ nhớ của a thay vì tạo mới 1 bản sao cho b.

-     String& operator=(String&& other) noexcept {

       if(this == &other){

           return \*this;

       }

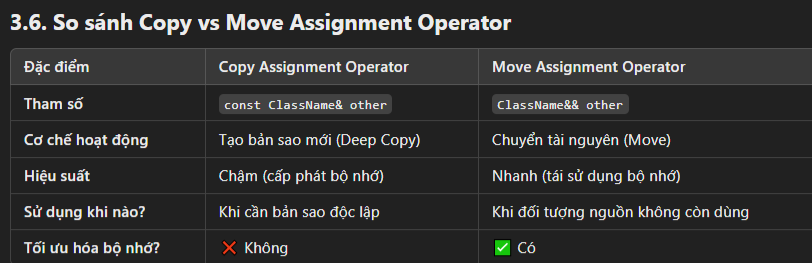
       delete[] data; // Giai phong bo nho trong heap

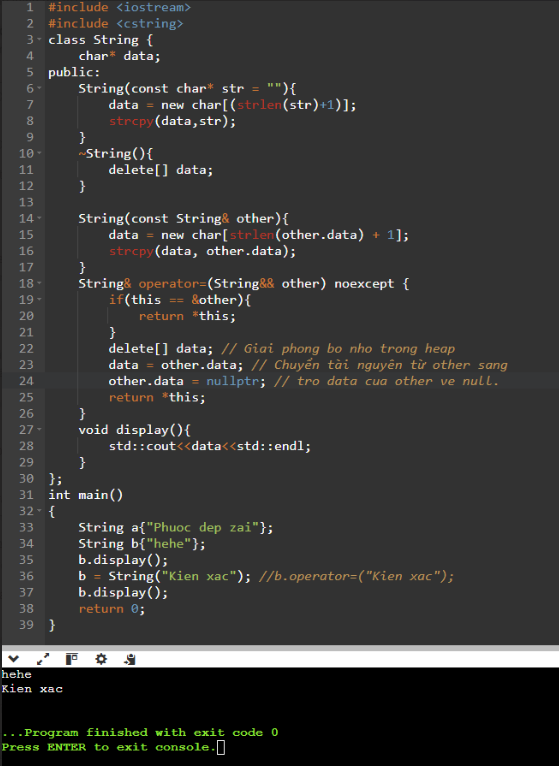
       data = other.data; // Chuyển tài nguyên từ other sang

       other.data = nullptr; // tro data cua other ve null.

       return \*this;

* Về bản chất thì data của object a đang trỏ đến 1 vùng nhớ x trong heap. Khi ta dùng Overloading the Assignment Operator (Move) **b = std::move(a)** (phải chuyển a về Rvalue) thì bản chất ta đang gán con trỏ data của b sang vùng x, sau đó chuyển con trỏ data của a về Null.

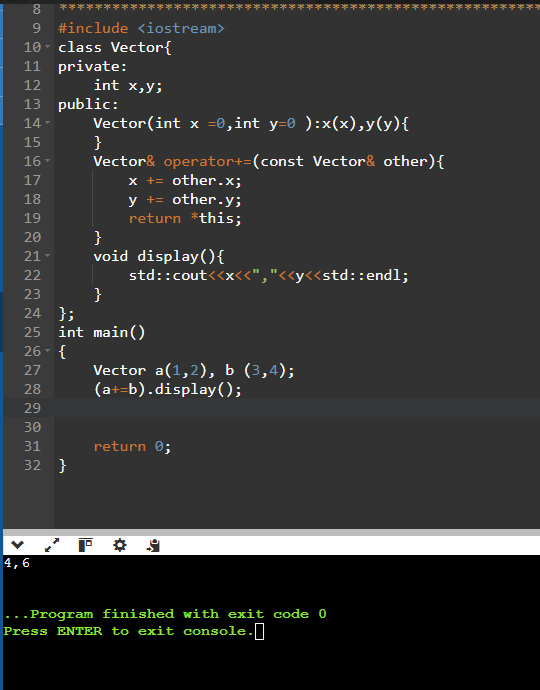




3. Overloading Operators as Member Functions

-  Nếu toán tử thay đổi đối tượng gọi -> nó nên là hàm thành viên.

-  Nếu toán tử **không thay đổi đối tượng gọi** và cần tính đối xứng giữa hai toán hạng, nó nên là **hàm toàn cục**.



4. Overloading Operators as Global Functions

- Khi nạp chồng toán tử dưới dạng hàm toàn cục, nó không thuộc về lớp nào, mà sử dụng các object của các lớp làm toán hạng.

-✔ **Dùng hàm toàn cục khi:**

* **Toán tử không thay đổi trạng thái của bất kỳ toán hạng nào**, mà chỉ trả về một kết quả mới.
* **Cần tính đối xứng giữa hai toán hạng** (không có toán hạng nào quan trọng hơn toán hạng kia).
* **Toán tử cần hoạt động với kiểu dữ liệu cơ bản và kiểu do người dùng định nghĩa** (int + Vector, Vector + int).

