C++

1. **OOP:**
2. **Class:**

- class is a user-defined data type that contains properties and methods.

- Object is an instance of class

1. **Abstraction: Tính trừu tượng**

- Tính trừu tượng là ẩn những thông tin không cần thiết, và việc triển khai thực tế của 1 ứng dụng khỏi user.

- abstraction just provides essential information about data to outside. Background details and implementation are hidden.

- 2 loại abstraction trong OOP:

+ Data Abstraction (trừu tượng hoá dữ liệu): là data của object không bị hiển thi ra bên ngoài. Nếu cần lấy data của object thì dùng get, set. This is abstraction in class.

+ Process Abstraction (trừu tượng hoá quy trình): là ẩn đi chi tiết của các function của object. This is abstraction in header file.

1. **Encapsulation: tính đóng gói**

- Dữ liệu và thông tin quan trọng của 1 đối tượng được đóng gói lại để các đối tượng khác bên ngoài ko thể làm thay đổi thông tin của đối tượng đó.

- private: chỉ có thể truy cập trong phạm vi của class đó, các class khác ko thể truy cập

- protected: có thể truy cập từ trong phạm vi của chính class đó hoặc các subClass kế thừa từ class đó.

- public: có thể truy cập từ bất cứ class nào.

1. **Polymorphism:**

- là 1 phương thức có thể đưa ra các kết quả khác nhau phụ thuộc vào kiểu data được xử dụng.

- Operator overloading: (nạp chồng toán tử): is re-define operators with custom data types. This is a compile polymorphism.

+ là bạn định nghĩa lại toán tử đã có trên kiểu dữ liệu người dùng tự định nghĩa để dể dàng thể hiện các câu lệnh trong chương trình.

- Function overloading: is using a function name to perform different tasks. Parameter of function is different data types. This is a compile polymorphism.

- Function override: Using in inheritance. This is runtime polymorphism

1. **Inheritance:**

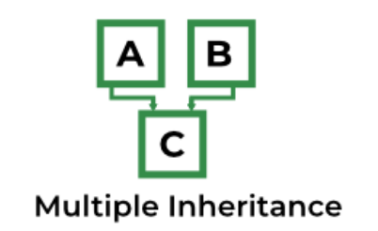
- Inheritance is capability of a class to inherit properties and methods from another class.

- 5 types of inheritance:

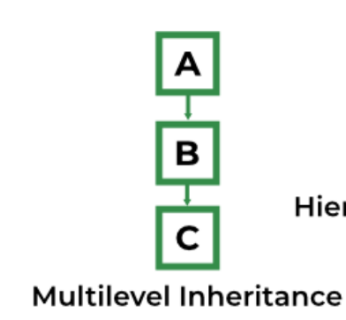
+ single inheritance: child class derived directly from base class.



+ Multiple inheritance: child class derives from multiple base classes.



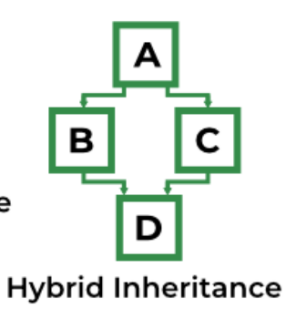
+ Multi-level inheritance: child class derives from a class which is also derived from another base class.



+ Hierarchical inheritance: multiple child classes derived from a base class.



+ hybrid inheritance:



- Is static property initialized in constructor function of class?

+ No. Because static variable are not tied with any instances of a class.

- Can constructor be virtual function?

+ No.

+ virtual function được xử lý ở runtime. Và nó phụ thuộc vào loại object đã được khởi tạo rồi.

+ constructor dùng để khởi tạo object, việc gọi constructor thì được xác định ở complile time dựa vào loại object đang được tạo.

1. **Abstract class:**

<https://www.geeksforgeeks.org/pure-virtual-functions-and-abstract-classes/?ref=header_search>

- là class có chứa 1 hoặc nhiều phương thức trừu tượng (virtual method)

- virtual function must be overridden in child classes.

- pure virtual function is declared by assigning 0 in the declaration.

- Cannot instantiate an abstract class.

Test t;

- class A is abstract class, and A has a pure virtual function. Class B inherits class A. If B doesn’t override the virtual function of A, B will become a abstract class.

- 1 class chỉ có thể kế thừa 1 abstract class.

- Các function trong abstract class có thể là public hoặc protected để child classes có thể override virtual function.

- Dùng abstract class khi: 1 nhóm các đối tượng có cùng bản chất, kế thừa từ 1 class thì sử dụng abstract class.

1. **Interface:**

- là class không có các thuộc tính (attributes) mà chỉ có các phương thức ảo.

- không thể khởi tạo 1 instance của interface.

- Các Interface có thể extend với nhau.

- 1 Class có thể implement nhiều Interface.

- Dùng Interface khi:

+ 1 nhóm đối tượng không cùng bản chất nhưng có các hành động giống nhau thì dùng Interface.

+ Khi bạn muốn tạo dựng một bộ khung chuẩn gồm các chức năng mà những module hay project cần phải có. Giống như sau khi nhận requirement của khách hàng về team ngồi với nhau và phân tích các đầu mục các tính năng của từng module, sau đó triển khai vào code viết các interface như đã phân tích,để các bạn dev có thể nhìn vào đó để thực hiện đủ các tính năng (khi đã implement rồi thì không sót một tính năng nào ^^).

- Nhược điểm: mỗi khi thêm method vào Interface, phải implement method này ở tất cả các class implement interface đó.

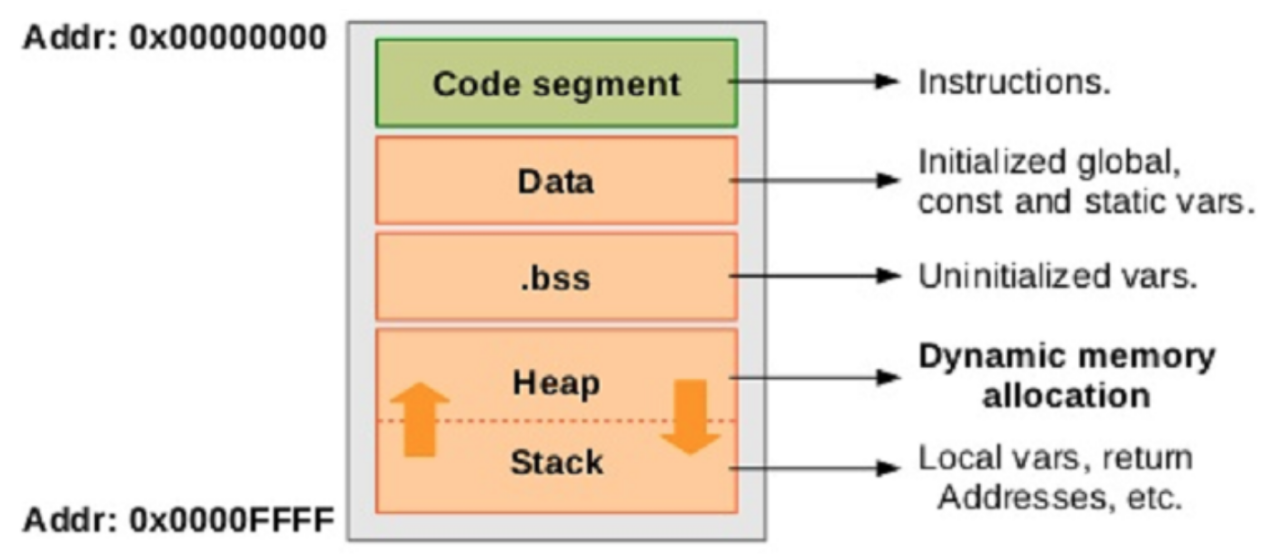
1. **Advantages of OOP**

- khả năng mở rộng và tái xử dụng code

- dễ dàng maintain.

- It provides better data security.

1. **Memory manager:**
2. **Phân vùng trên bộ nhớ ảo:**



- code segment (text segment):

+ là nơi lưu trữ các mã lệnh đã được biên dịch của chương trình máy tính.

+ khi chạy chuơng trình, đoạn code sẽ đc chuyển thành các mã lệnh và đc lưu trữ ở code segment, rồi được chuyển đến CPU để xử lý.

- Data segment: là phân vùng sử dụng để khởi tạo giá trị cho các kiểu biến static và global.

- Bss segment: lưu trữ các biến static, global nhưng chưa được khởi tạo giá trị cụ thể.

- Heap segment:

+ dùng để cấp phát bộ nhớ động, thông qua dynamic memory allocation: new,...

+ bộ nhớ trên heap segment sẽ ko tự động giải phóng cho đến khi chương trình kết thúc -> fai giải phóng bộ nhớ bằng manual: dùng delete function.

+ vùng nhớ cấp phát phải được quản lý bởi ít nhất 1 con trỏ => có thể giải phóng bộ nhớ manual.

+ cấp phát bộ nhớ trên Heap sẽ chậm hơn các phân vùng khác.

+ Heap có kích thước khá lớn.

- Stack segment:

+ dùng để cấp phát bộ nhớ cho các parameter của function và local variable.

+ Stack có kích thước vùng nhớ hạn chế => nêú cấp phát vùng nhớ cho variable vượt quá size của Stack -> stack sẽ bị overflow.

+ Cấp phát bộ nhớ trên Stack khá nhanh.

+ kích thước vùng nhớ cấp phát trên stack fai được khai báo rõ trước khi biên dịch.

- Compare memory allocation in stack and heap:

|  |  |
| --- | --- |
| Stack | Heap |
| Auto deallocated when variable goes out of scope | Required delete memory to deallocate memory. |
|  |  |

1. **Dynamic memory allocation:**

- in C:

+ malloc: allocate raw memory on the heap

+ calloc: allocate memory on heap and initialize it’s memory address to 0

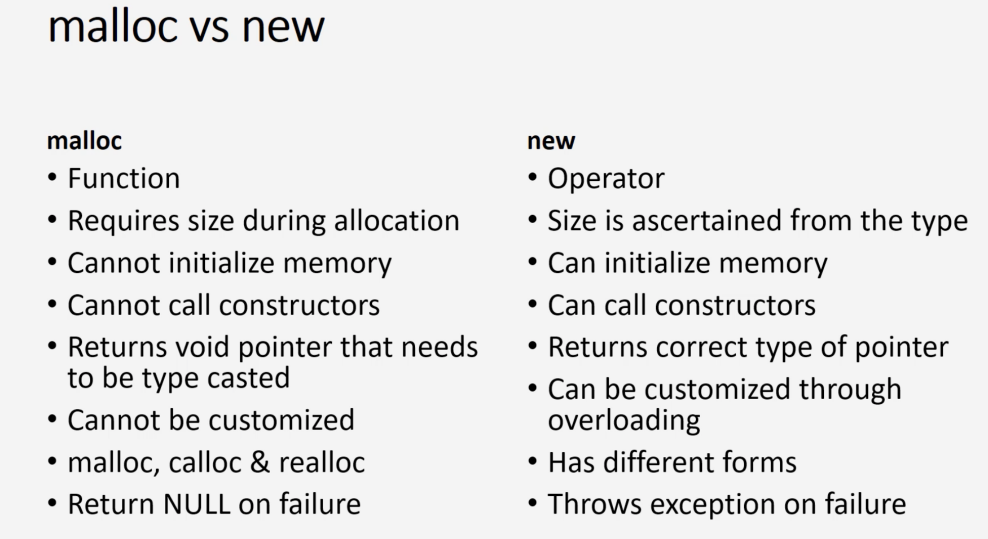
+ realloc: allocate lager chunk of memory for an existing allocation

+ free: deallocate memory

- in C++:

+ new: allocate memory on the heap

+ delete: deallocate memory



1. **Các tránh memory leak trong C++:**

- Sử dụng cấp phát động 1 cách cẩn thận:

+ Khi sử dụng new để cấp phát bộ nhớ thì fai dùng delete để giải phóng bộ nhớ.

+ Khi cấp phát bộ nhớ fai nhớ giải phóng bộ nhớ khi không sử dụng nữa.

- Sử dụng các con trỏ thông minh:

+ shared\_ptr: cho phép nhiều con trỏ tham chiếu đến 1 vùng nhớ và sẽ giải phóng vùng nhớ khi không còn con trỏ nào tham chiếu vào. Nên để shared\_ptr có thể tự free memory thì fai giảm reference count = 0.

+ unique\_ptr:

- tự động khởi tạo đối tượng khi được khai báo -> đối tượng luôn được tạo và sẵn sàng sử dụng.

- sẽ tự động giải phóng khi không được dùng đến nữa

- không thể copy 1 unique\_ptr, phải dùng move()

+ weak\_ptr:

- dùng cùng shared\_ptr

- weak\_ptr có thể giữ một tham chiếu tới một shared object được quản lý bởi shared\_ptr, nhưng nó không làm tăng reference count.

- khi sử dụng một weak\_ptr, chúng ta cần kiểm tra xem đối tượng mà nó trỏ tới còn valid hay không.

- weak\_ptr thường được sử dụng nhằm tránh việc tham chiếu vòng của shared\_ptr.

- Sử dụng function destructor:

+ trong các class, bạn có thể tự định nghĩa hàm destructor() để giải phóng tài nguyên cần thiết khi object bị huỷ.

- Quản lý quyền sở hữu:

+ theo dõi quyền sở hữu của object và con trỏ để đảm bảo rằng các con trỏ được giải phóng hoặc chuyển quyền sở hữu trước khi object bị huỷ.

- Sử dụng các công cụ kiểm tra:

+ static analysic tool

+ memory profiling tool

1. **Threads:**

- Có thể tạo thread cho:

+ a function

+ a function pointer

+ a lambda expression

+ member function of class.

- join():

+ main thread will wait until the thread finished, then it finishes program.

- detach():

+ tách child thread khỏi parent thread.

+ Cho phép child thread được chạy ngay lập tức từ parent thread. Các thread sẽ chạy riêng biệt.

1. **pass parameter to thread:**

+ pass value: sẽ tạo ra 1 bản copy của variable ấy -> có thể leak of memory

+ pass reference: không tạo ra bản copy. Nên sử dụng hàm ref(), cref()

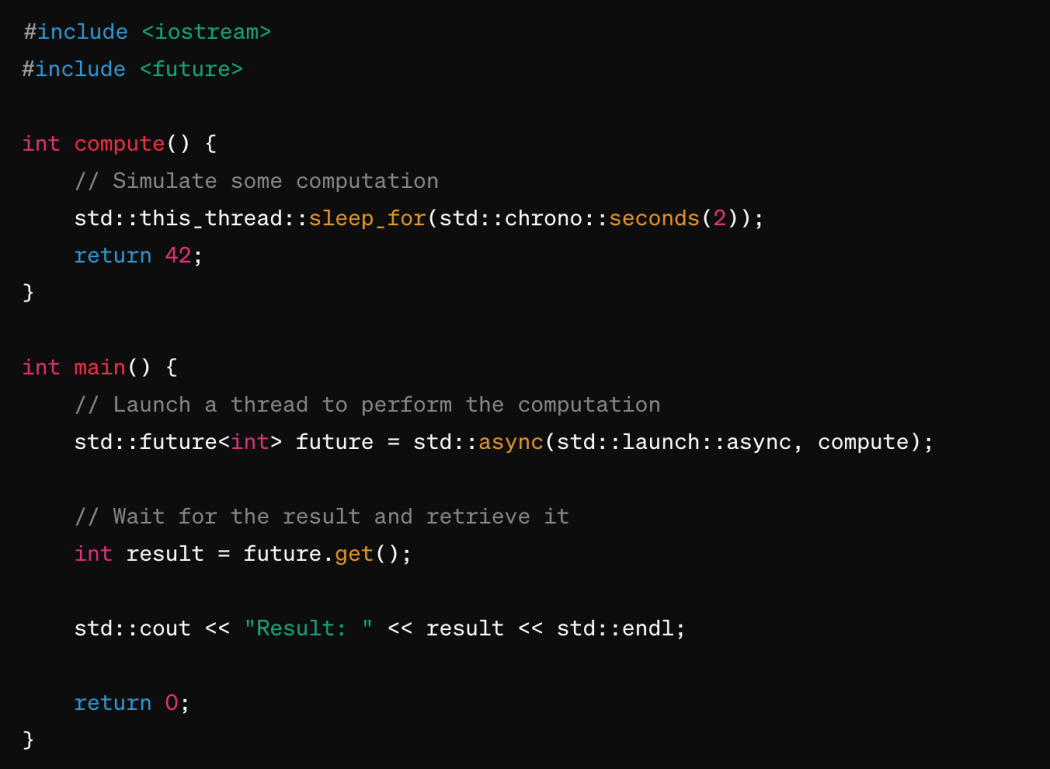
1. **Get return value of Thread:**

- C1: using package\_task and future value:

+ package\_task doesn’t create a new thread. So need using Thread to create a new thread.

- C2: using async() and future():

+ async() creates a new thread.



- C3: using promise and future:

|  |
| --- |
| #include <iostream>  #include <future>  #include <thread>  int compute() {  // Simulate some computation  std::this\_thread::sleep\_for(std::chrono::seconds(2));  return 42;  }  int main() {  // Create a promise and get a future from it  std::promise<int> promise;  std::future<int> future = promise.get\_future();  // Launch a thread to perform the computation  std::thread thread([&promise]() {  // Perform the computation  int result = compute();  // Set the result in the promise  promise.set\_value(result);  });  // Wait for the result and retrieve it  int result = future.get();  std::cout << "Result: " << result << std::endl;  // Join the thread  thread.join();  return 0;  } |

1. **Multi thread:**
2. **Synchronous:**

- Mutex: dùng để đồng bộ truy cập vào các tài nguyên chung giữa các thread. Đảm bảo tại 1 thời điểm chỉ có 1 thread có quyền truy cập vào object.

+ lock(): try to lock mutex. If it’s not success, it will wait until successful.

+ try\_lock(): lock mutex. If it’s not success, it will return immediately.

- lock\_guard: mutex sẽ tự động unlock khi out of scope -> không cần unlock()

- unique\_lock:

+ Không cần fai lock object khi khởi tạo, mà có thể gọi hàm lock() sau, khi nào cần thiết.

+ có thể chuyển quyền sở hữu giữa các unique\_lock thông qua move(). unique\_lock không thể copy.

+ unique\_lock tốn bộ nhớ , truy cập chậm hơn lock\_guard do unique\_lock chứa 1 flag để lưu thông tin liệu mutex hiện tại đang được sở hữu bởi unique\_lock hay không.

- Condition\_variable: is events to notify between threads. Đồng bộ thứ tự thực thi giữa các threads.

+ notify\_one(): signal a thread

+ notify\_all(): signal all thread which a waiting



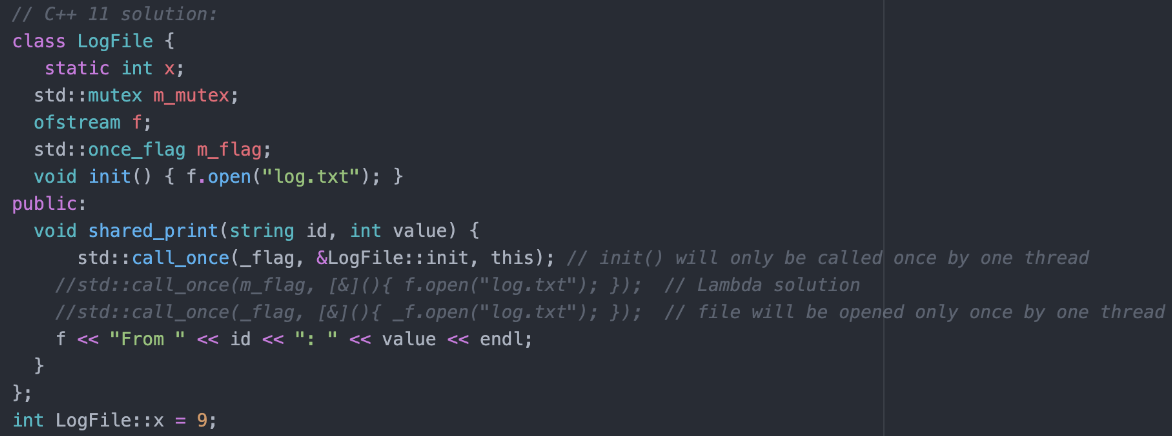
- Promise and future:

- Semaphore:

1. **Tránh khởi tạo object nhiều lần giữa các thread:**

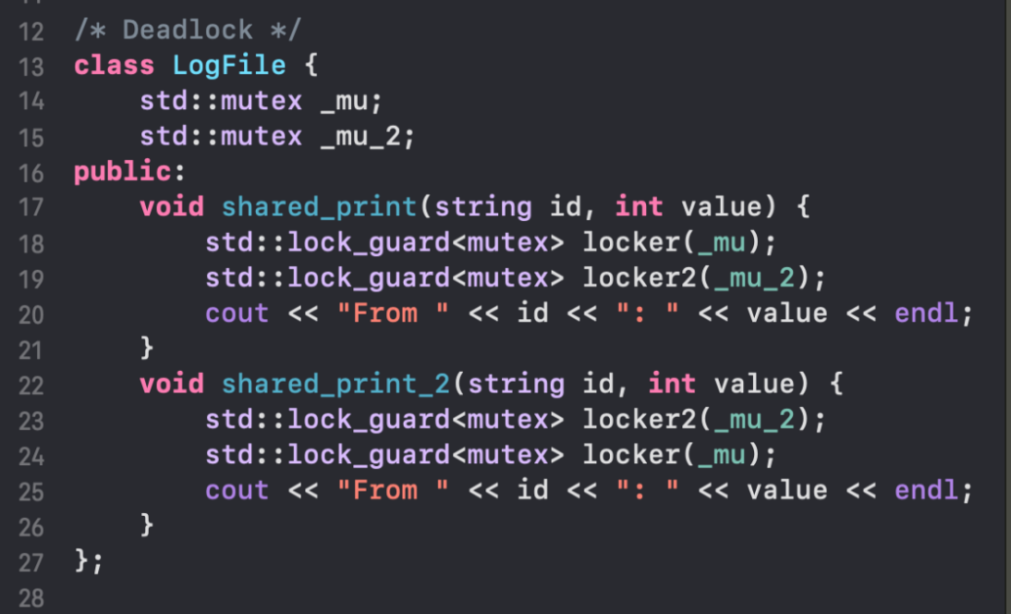
- dùng lazy initialization: std::call\_once, std::once\_flag.

- Các object, function chỉ được gọi 1 lần bởi 1 thread dù đang chạy trong concurrency có nhiều threads.



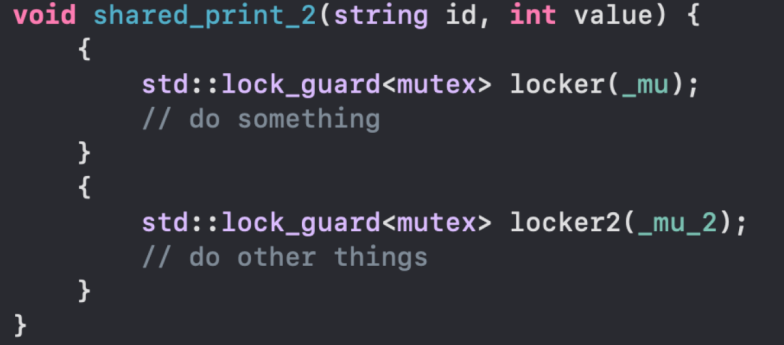
1. **Avoid deadlock:**

- Deadlock occurs in using mutex. Thread is locked without unlock -> program is hang.

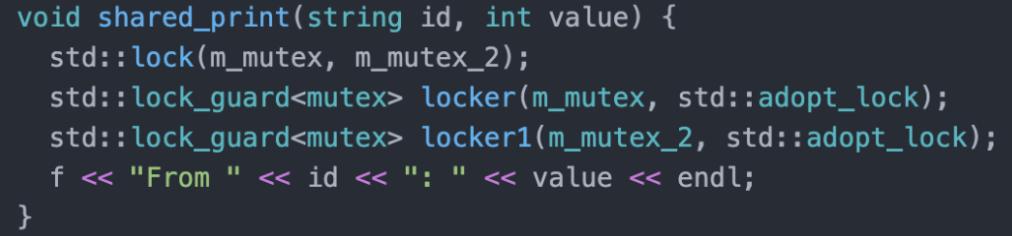


- Methods to avoid deadlock:

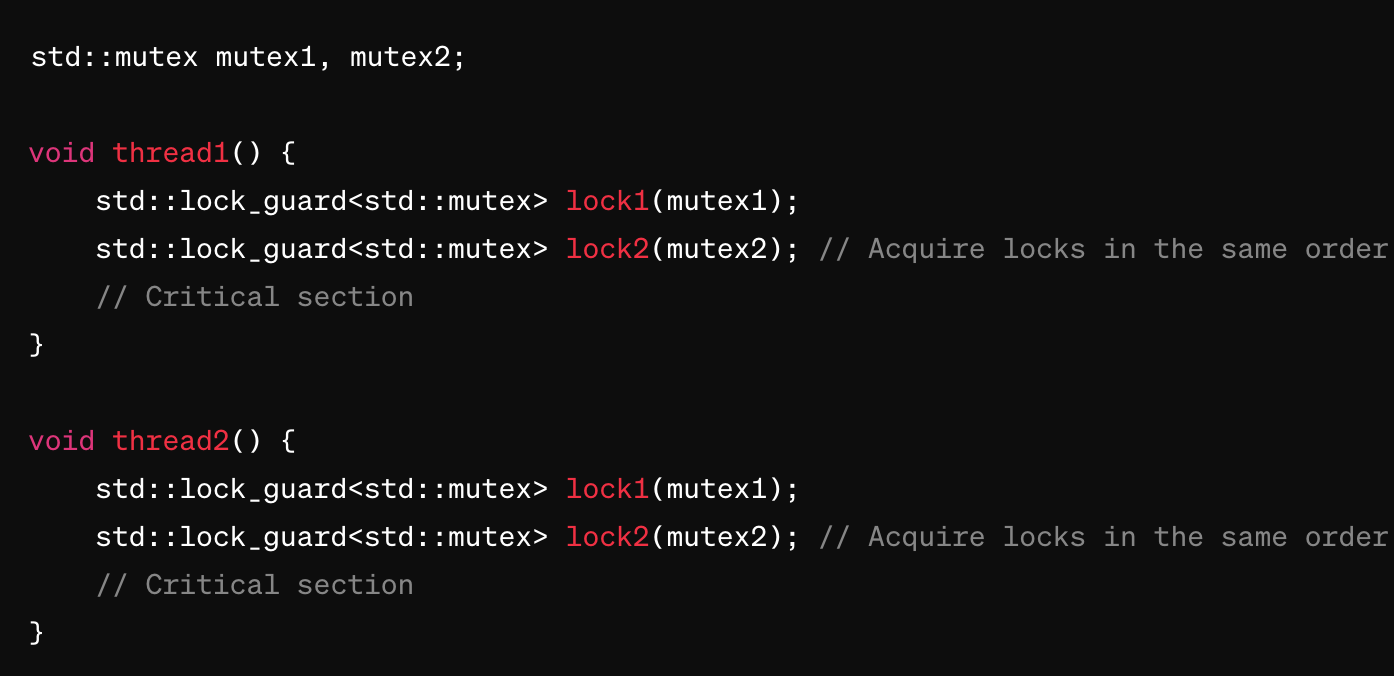
+ nên sử dụng 1 khoá mutex duy nhất cho 1 resource nào . Bởi vì deadlock chủ yếu xảy ra bởi việc sử dụng nhiều khoá mutex cùng lúc. Thay vì dùng chung 2 khoá mutex cho cùng resource như trên, mình có thể tách ra riêng lẻ như sau:



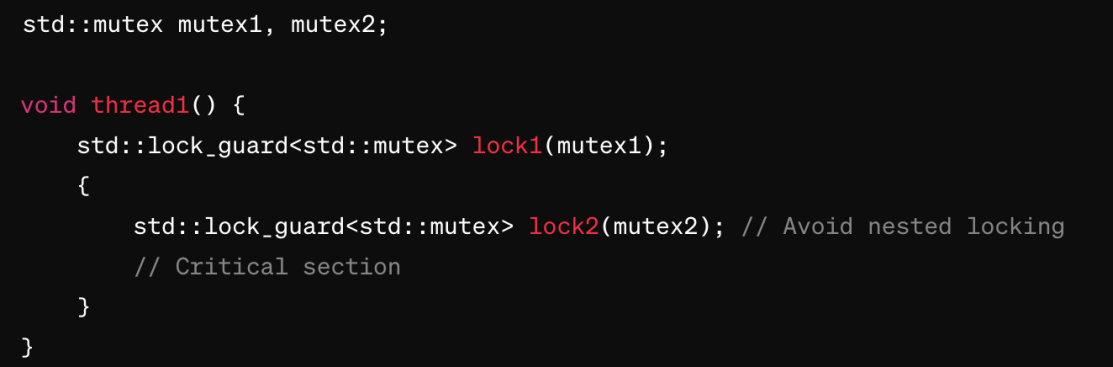
+ Sử dụng std::lock() khi muốn khoá nhiều mutex. std::lock sử dụng cơ chế sẵn có để giúp ta tránh deadlock. Khi kết thúc phạm vi sử dụng, mutex sẽ được unlock (mở khoá) một cách chính xác.



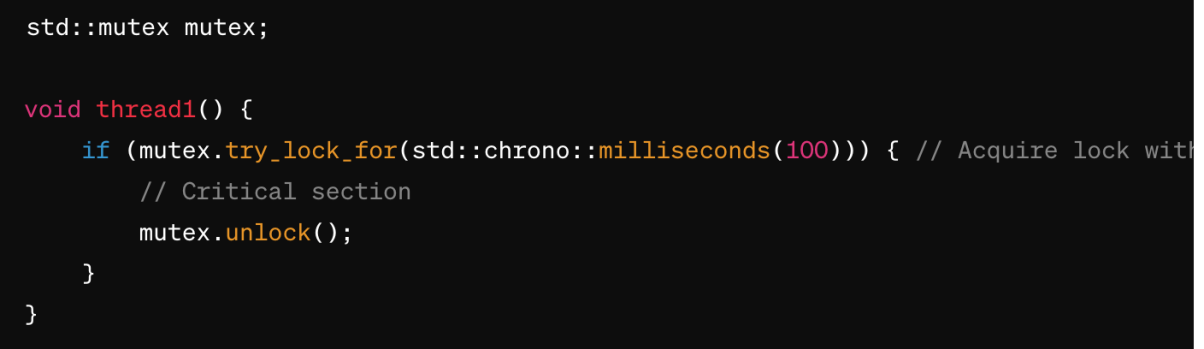
+ All threads have to lock in same order.



+ Avoid nested locking: (tránh khoá lồng nhau):



+ using timeout to prevent thread from being blocked:



1. **Singleton pattern:**

- Đảm bảo chỉ tạo ra 1 instance của object.

- Lợi ích:

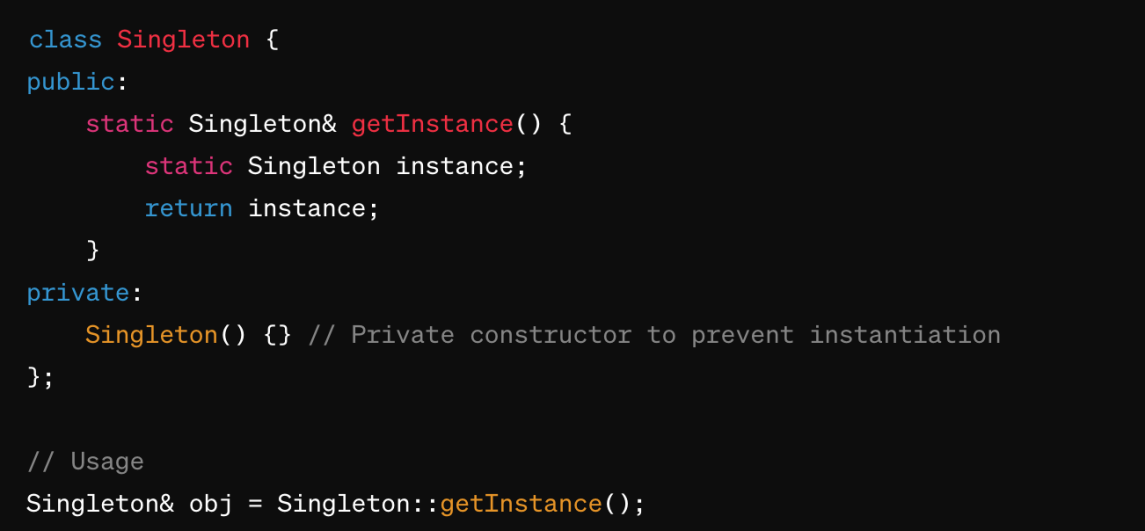
+ không cần tạo ra nhiều object -> memory dùng cho app sẽ giảm đi.

+ trong multithread: không bị xung đột khi có nhiều object của cùng 1 class cùng hoạt động.

- Dùng singleton pattern khi:

+ muốn hạn chế sử dụng tài nguyên.

+ muốn có 1 object chứa data chung cho cả project.

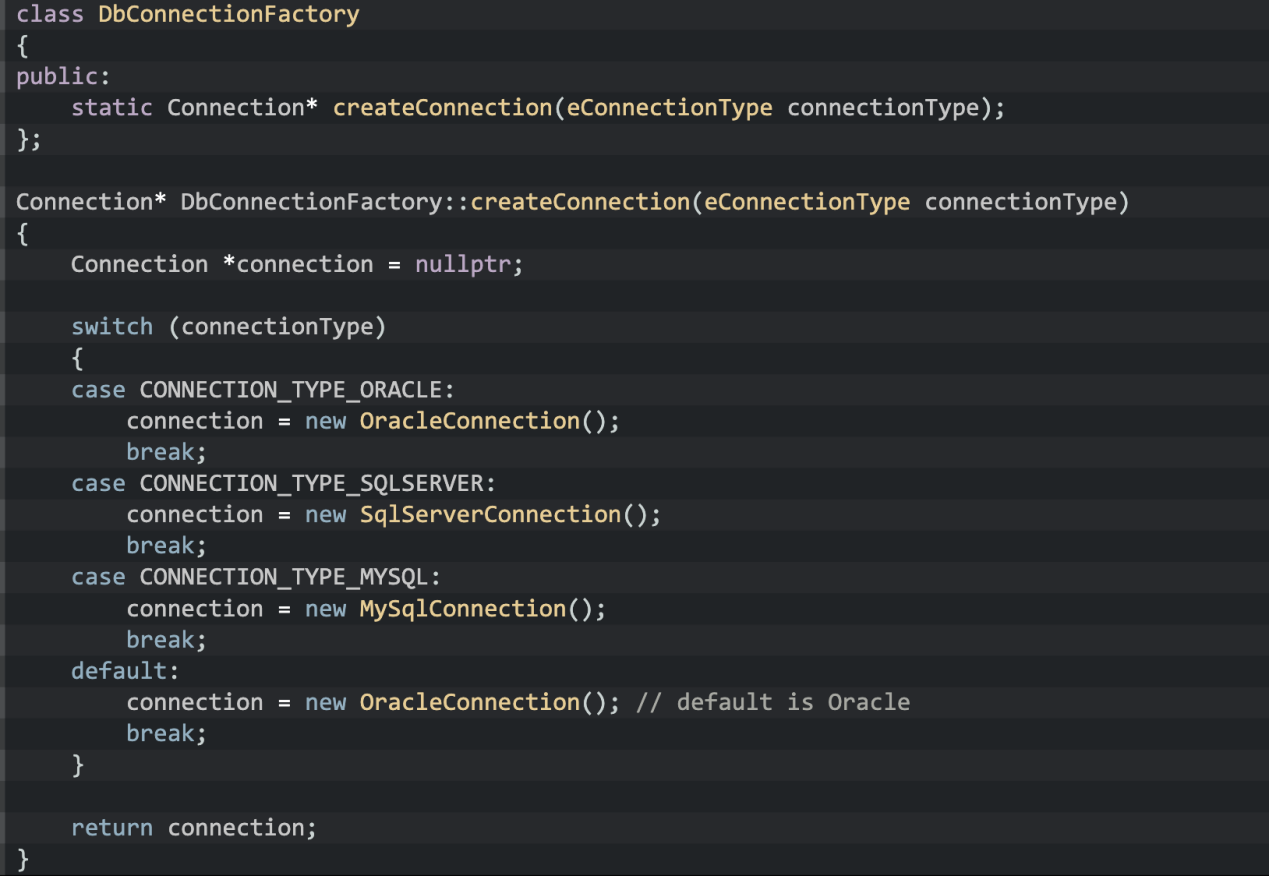


1. **Factory Pattern:**

- Factory pattern includes 2 patterns: factory method and abstract factory pattern.

1. **Factory method pattern:**

- cung cấp 1 interface để tạo các instance của các class khác dựa vào tham số truyền vào.



- Trong ví dụ trên:

+ có 3 class OracleConnection, SqlServerConnection, MySqlConnection kế thừa class Connection.

+ interface DBConnectionFactory và function createConnection() chính là factory method.

- Ưu điểm:

+ tách biệt được công việc tạo object/instance ra khỏi xử lý của client code (client code là phần code sử dụng object / instance để thực hiện hoàn thành công việc của nó). Client code sẽ không cần phải quan tâm instance được tạo ra như thế nào, nó là object của class nào, nhờ đó giảm thiểu dependency của client code.

+ Dễ maintain: thêm, ­sửa.

1. **Abstract factory pattern:**

<https://cppdeveloper.com/design-patterns/design-patterns-3-factory-pattern-phan-2-abstract-factory/>

- Giống đa hình.

- cung cấp 1 interface để tạo các object độc lập mà không cần chỉ ra class cụ thể.

- ví dụ bài toán cần giải quyết:

+ Giả sử chúng ta đang xây dựng một ứng dụng có UI, và ứng dụng hỗ trợ hiển thị theo nhiều style (trong bài này gọi là themes). Các theme khác nhau sẽ có hình dáng và màu sắc của các đối tượng đồ họa (trong bài này gọi là widgets) như: scroll bar, button, ... khác nhau. Nếu trong code chúng ta tạo các instance của widget một cách trực tiếp thì code của chúng ta sẽ bị trùng lặp rất nhiều và sẽ phải sử dụng rất nhiều các câu lệnh điều kiện để xử lý việc chuyển đổi theme. Việc này dẫn đến code khó bảo trì, khó reuse.

+ Giả sử chúng ta có 2 themes là Dark và Light, 2 widgets là ScrollBar và Button.

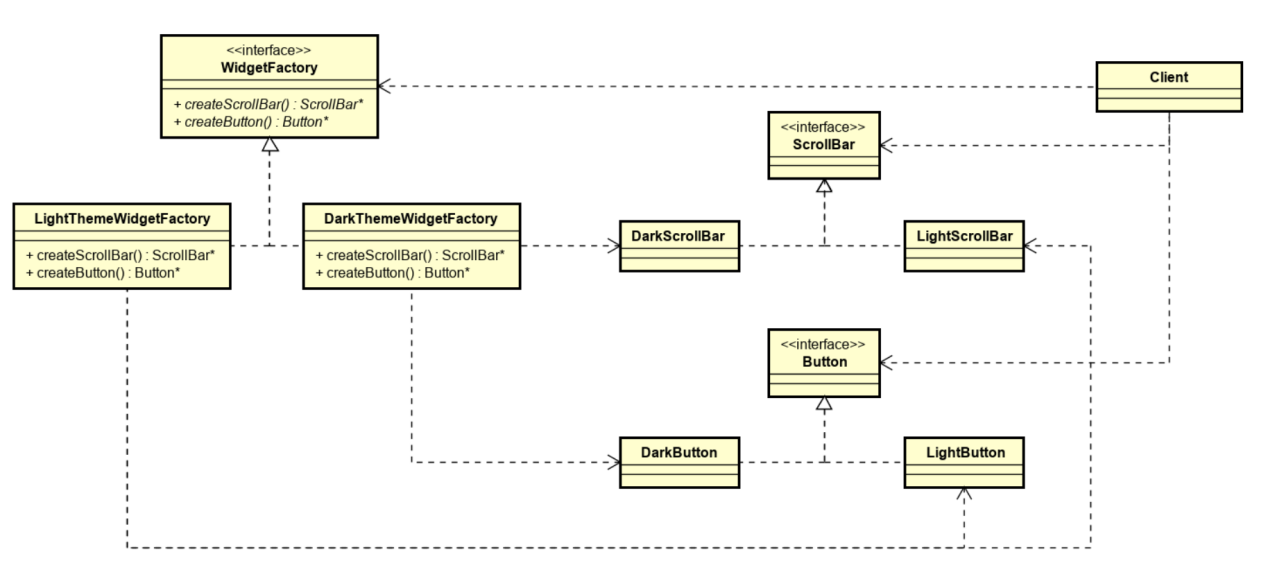
- Solution:

+ tạo 1 interface WidgetFactory: khai báo các methods để tạo instance cho mỗi loại widget.

+ client sẽ call đến các function được định nghĩa bởi widget để lấy được widget instance nhưng nó không biết (và cũng không cần biết) class widget cụ thể nào được trả về. Chính vì vậy Client code sẽ độc lập và tách biệt khỏi implementation cụ thể của các widget, cũng hoàn toàn độc lập với việc các widget instance được tạo ra như thế nào.

+ Có 2 class con cụ thể của WidgetFactory cho mỗi theme đó là LightThemeWidgetFactory và DarkThemeWidgetFactory. Các class con sẽ implement các method để tạo widget instance tương ứng với mỗi theme.

+ Client code tạo các widget instance chỉ thông qua interfaces được khai báo bởi WidgetFactory mà không quan tâm đến các class cụ thể của widget.



|  |
| --- |
| class Button  {  // declare methods  };  class ScrollBar  {  // declare methods  };  class LightButton : public Button  {  // LightButton need implement methods declared by Button  };  class DarkButton : public Button  {  // DarkButton need implement methods declared by Button  };  class LightScrollBar : public ScrollBar  {  // LightScrollBar need implement methods declared by ScrollBar  };  class DarkScrollBar : public ScrollBar  {  // DarkScrollBar need implement methods declared by ScrollBar  }; |

+ tạo interface WidgetFactory

|  |
| --- |
| class WidgetFactory  {  public:  virtual ScrollBar\* createScrollBar()=0;  virtual Button\* createButton()=0;  }; |

+ implement interface WidgetFactory

|  |
| --- |
| class LightThemeWidgetFactory : public WidgetFactory  {  public:  ScrollBar\* createScrollBar() {  return new LightScrollBar();  }  Button\* createButton() {  return new LightButton();  }  };  class DarkThemeWidgetFactory : public WidgetFactory  {  public:  ScrollBar\* createScrollBar() {  return new DarkScrollBar();  }  Button\* createButton() {  return new DarkButton();  }  }; |

+ Client code sử dụng Factory để tạo instance của Widget

|  |
| --- |
| // use light theme  LightThemeWidgetFactory widgetFactory;  WidgetFactory \*factoryPtr = &widgetFactory;  ScrollBar \*scrollbar = factoryPtr->createScrollBar();  Button \*button = factoryPtr->createButton();  // use dark theme  DarkThemeWidgetFactory widgetFactory;  WidgetFactory \*factoryPtr = &widgetFactory;  ScrollBar \*scrollbar = factoryPtr->createScrollBar();  Button \*button = factoryPtr->createButton(); |

-Abstract factory được sử dụng khi:

+ Muốn tách biệt độc lập phần code sử dụng các products với việc implement và tạo instance của chúng, nâng cao tính reusability và mantainability của hệ thống.

+ dùng khi hệ thống có nhiều loại khác nhau của 1 sản phẩm.

1. **Adapter pattern:**

<https://cppdeveloper.com/design-patterns/design-patterns-4-adapter-pattern/>

- Problem:

+ giả sử ứng dụng bao gồm 2 phần chính là UI (User Interface) và Back-end

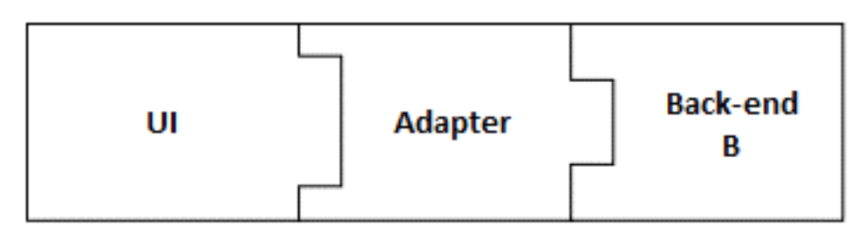
+ Phần Back-end chúng ta đang sử dụng được phát triển bởi một công ty khác, tạm gọi là công ty A đi.

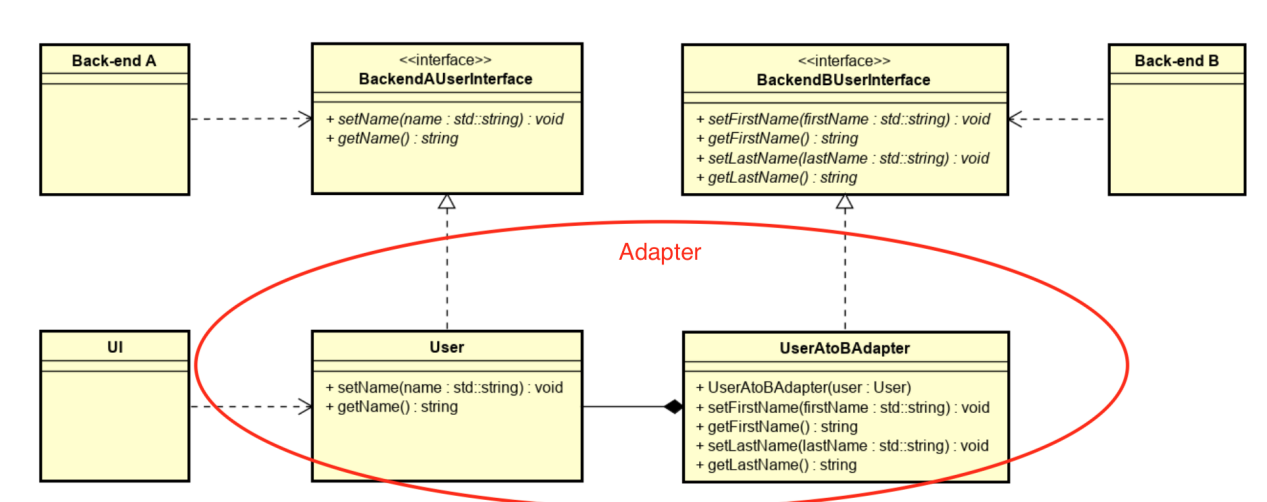
+ Bây giờ chúng ta phải chuyển sang dùng Back-end của công ty B. Và có một vấn đề xảy ra, đó là các Interfaces của Back-end A và Back-end B khác nhau (khác nhau về function name, về danh sách parameters, kiểu dữ liệu của paramaters) do đó nếu muốn dùng Back-end B thì chúng ta đang đứng trước nguy cơ phải sửa rất nhiều code.

- Solution:

+ dùng adapter pattern

+ cần tạo một adapter ở giữa để giúp cho UI có thể làm việc được với Back-end B. Adapter này cũng tương tự như adpater chuyển từ ổ cắm tròn sang ổ cắm dẹt, giúp cho phích cắm đầu tròn có thể nối được vào ổ cắm dẹt



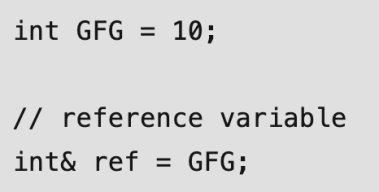


1. **Compare process and thread:**

|  |  |
| --- | --- |
| Process | Thread |
| Is a running program. It has memory space, resources, state execution | Is a smallest unit of execution in process. It shares same memory, resources with process. |
| Process has its own memory, resources. Communication between processes requires mechanisms: socket, pipe, shared memory | Threads in a process share same memory, resources. Them can communicate directly through shared variable or synchronization. |
| Creating and managing process is more overhead than thread | Thread is lightweight in comparing with process. |
| If a process is blocked, no other process can execute until first process is unblocked | If a thread is blocked and waited, other thread can run. |

1. **What is reference?**

- Khi 1 variable đc khai báo là reference, reference variable trở thành 1 tên khác của variable có sẵn đó. Trong ví dụ dưới. Khi ref thay đổi giá trị -> GFG cg thay đổi giá trị và ngược lại.



1. **What is the difference between C and C++:**

|  |  |
| --- | --- |
| C | C++ |
| It doesn’t support object, class | It supports object, class |
| It doesn’t support OOP concept | OOP concept |
| No function overload and operator overload | function overload and operator overload |
| Is a function-driven language (ngôn ngữ hướng chức năng) | Is a object-driven language (ngôn ngữ hướng đối tượng) |

1. **Different between Struct and Class:**

|  |  |
| --- | --- |
| Struct | Class |
| Members of struct are always default public mode | Members can be private, protected, public mode. |
| It is value type. It only holds value in memory. | It’s reference type. It holds reference of object in memory. |
| Memory in struct is stored as stack | Memory is stored as heap. |

1. **Differences between reference variable and pointer variable:**

<https://www.geeksforgeeks.org/pointers-vs-references-cpp/>

|  |  |
| --- | --- |
| Reference | Pointer |
| Declare and initialize in one step  int &p = a; | Can declare, then initialize  int \*p;  p = &a; |
| It cannot be reassigned | It can be reassign |
| It never holds null value | It can be null value |
| Cannot work with array | Work with array |
| It shares the same memory address with the original variable. It takes up no space on the stack | has its own memory address and size on the stack. Pointer contains address of variable. |
| Modifying reference -> original variable will be modified |  |

1. **What do you know about friend class and friend function?**
2. friend class:

- có thể truy cập đến biến private và protected của class làm bạn với nó.

1. Friend function:

- là function có thể truy cập đến các biến private và protected của class có friend function đó.

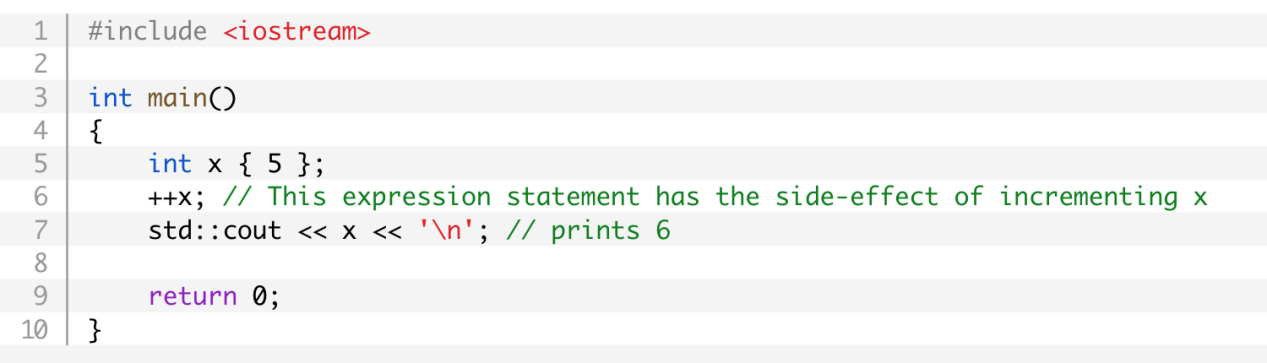
1. **Value category:**

- value category là 1 phần trong biểu thức (expression)

- there are two types of value category: (prior C++11)

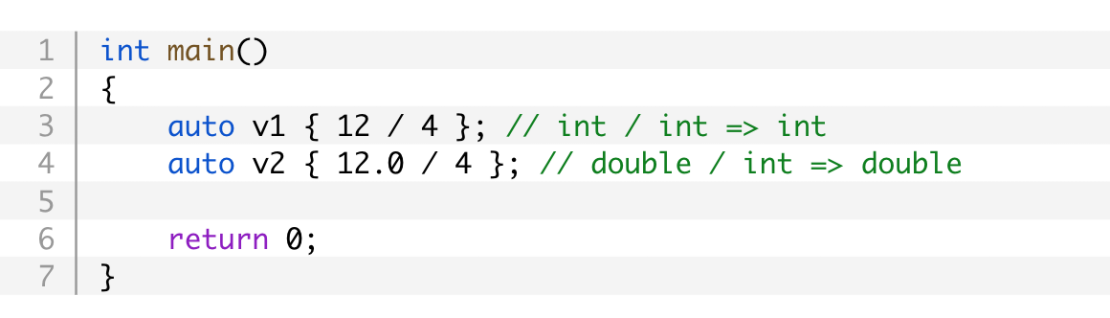
+ lvalue

+ rValue



- in the figure:

+ line 5 is an expression: int is type of expression, 5 is value category of expression.

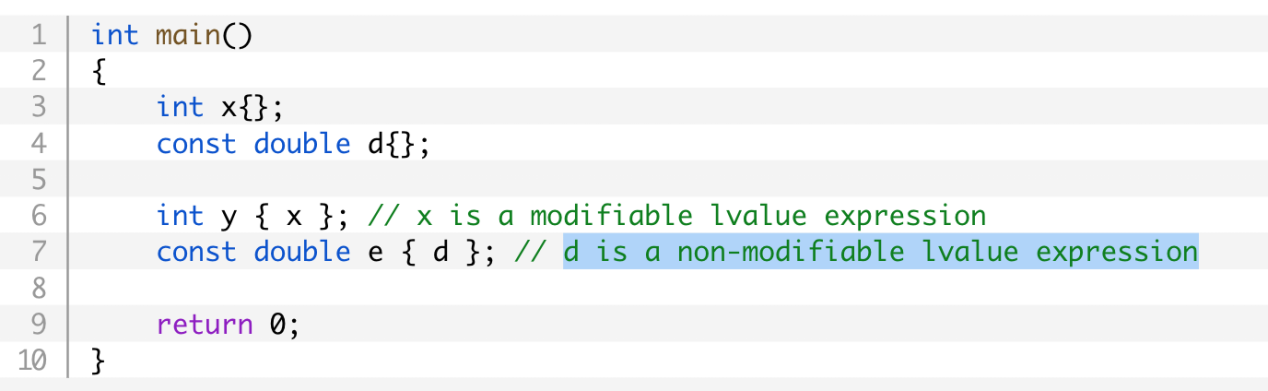


1. LValue:

- is an expression which evaluate to identifiable objects.

- identifiable object is object which can be accessed via identifier, reference, pointer. It has location in memory, address in memory.

- lvalue là các biến.

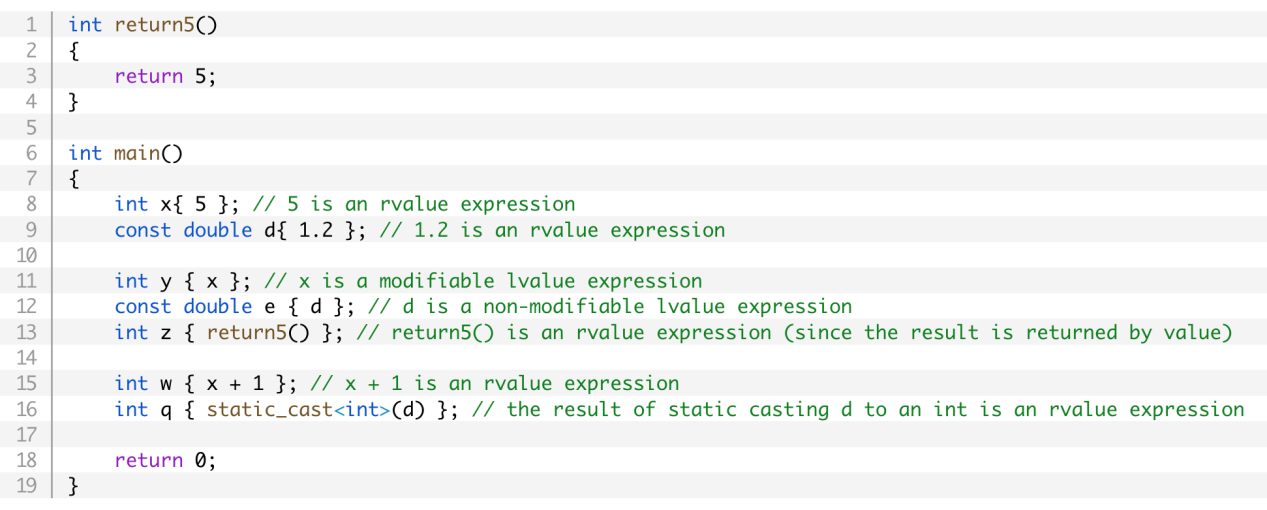


1. Rvalue:

- is expressions which create temporary values. The temporary values are not identifiable objects.

- It has not an address.

- rvalue là các giá trị tạm thời.



- In figure:

+ (x+1) is a temporary value (value = 6) -> nó không có address -> là rvalue.

1. **Ways to access private property of a class from other classes:**

- using getter method: get()

- using friend class.

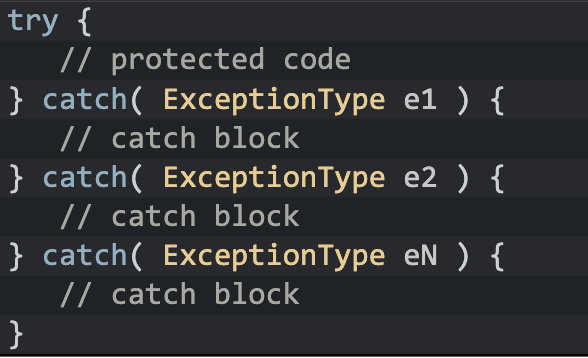
1. **Exception handling:**

- Dùng để bắt các lỗi xảy ra

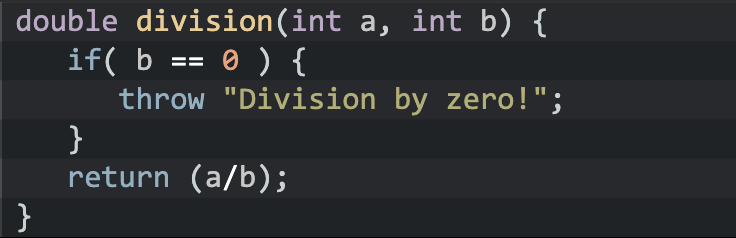
- Question: Can an exception be thrown in constructor and destructor?

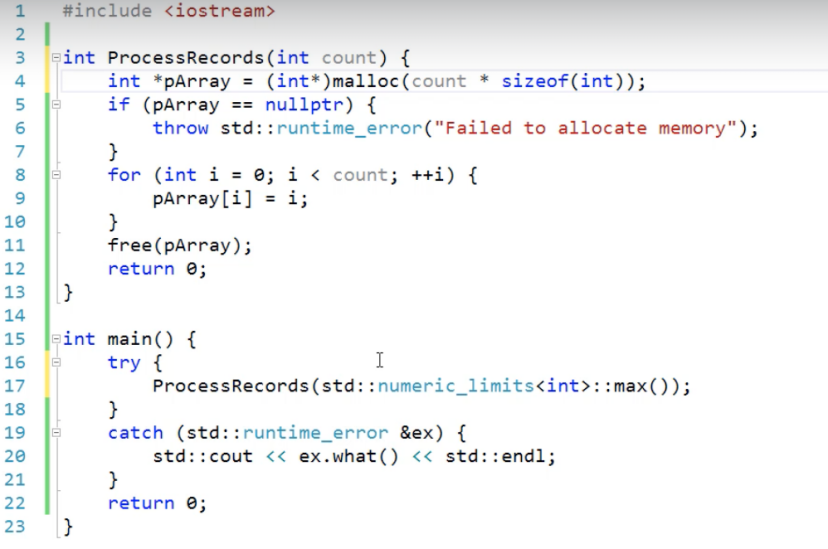
+ exception can be thrown in constructor. It’s not in destructor.

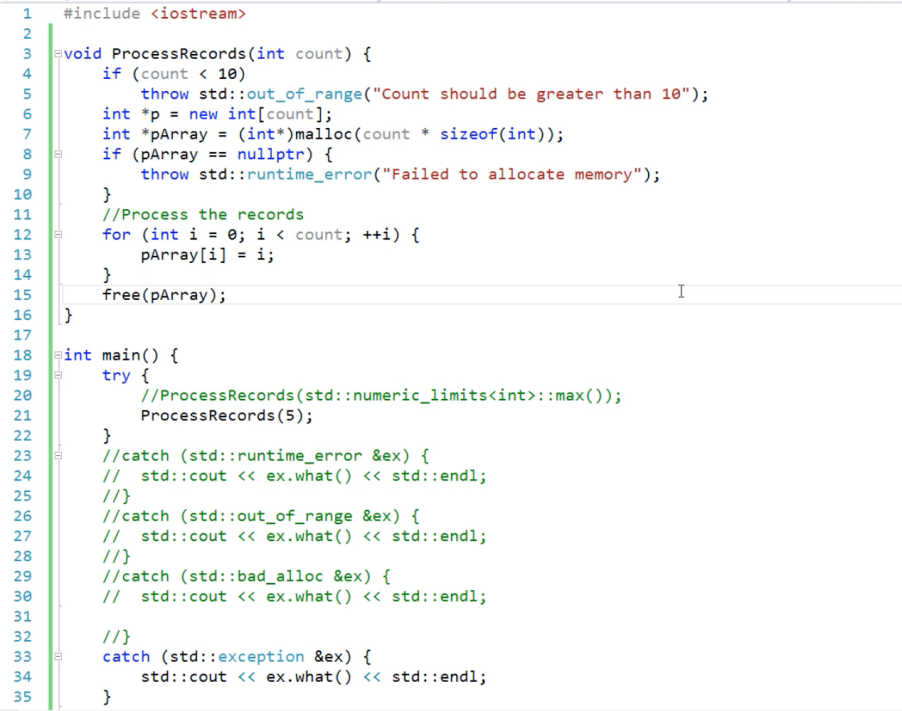
1. Try catch:



1. Throw:







1. **Prevent object from copying:**

- Copying object can make :

+ resource leaks, double deletion

+ Object có thể có thuộc tính ko được thay đổi. Nhưng nếu copy có thể khiến những object này bị thay đổi.

+ trong quản lý multi-thread, cho phép copy object có thể gây lên race condition hoặc các vấn đề về đồng bộ.

- Các biện pháp để prevent copying objects:

+ delete the copy constructor and copy assignment operator using the “= delete”



1. **Algorithms:**