C++ Training Course

Advance features

Lesson Objectives

- Understand about smart pointer
- Understand about multi threading
- Understand about compiling C++ program
- Understand about static vs dynamic library

các vấn đề gặp phải khi sử dụng con trỏ thường:
1. Rò rỉ bộ nhớ (Memory Leaks)
Mô tả: Bộ nhớ được cấp phát nhưng không được giải phóng sau khi không còn sử dụng.
Nguyên nhân: Không gọi delete sau khi sử dụng new hoặc không giải phóng bộ nhớ khi thoát khỏi phạm vi sử dụng.
2. Dangling Pointers
Mô tả: Con trỏ vẫn trỏ đến một vùng bộ nhớ đã được giải phóng.
Nguyên nhân: Sử dụng con trỏ sau khi gọi delete hoặc sau khi đối tượng ra khỏi phạm vi của nó.
3. Double Deletion
Mô tả: Giải phóng cùng một vùng bộ nhớ nhiều lần.
Nguyên nhân: Gọi delete hai lần trên cùng một con trỏ hoặc có nhiều con trỏ trỏ tới cùng một đối tượng và không kiểm soát việc giải phóng bộ nhớ.
4. Sử dụng bộ nhớ chưa được cấp phát (Use of Uninitialized Memory)
Mô tả: Truy cập vào vùng bộ nhớ chưa được cấp phát hoặc chưa được khởi tạo.
Nguyên nhân: Truy cập vào con trỏ chưa được khởi tạo (ví dụ: con trỏ null hoặc con trỏ ngẫu nhiên).
5. Truy cập ngoài phạm vi (Out-of-Bounds Access)
Mô tả: Truy cập vào vùng bộ nhớ nằm ngoài phạm vi của mảng hoặc vùng bộ nhớ cấp phát.
Nguyên nhân: Lỗi chỉ số mảng hoặc tính toán sai kích thước vùng bộ nhớ.
6. Undefined Behavior
Mô tả: Hành vi không xác định do các lỗi về quản lý bộ nhớ hoặc sử dụng con trỏ sai.
Nguyên nhân: Lỗi logic trong mã nguồn, truy cập bộ nhớ không hợp lệ, hoặc bất kỳ tình huống nào không được định nghĩa rõ ràng bởi ngôn ngữ.

Section 1

Smart pointer

CHÚ Ý: bản chất các smart pointer là các class template nên việc truy cập phải tuân thủ nguyên tắc của template.
sau khi khai báo 1 smart pointer có thể sử dụng như class (VD ptr.get())

Smart pointer. Agenda

- Introduction to smart pointer
- Types of smart pointer

7. Lỗi ghi đè bộ nhớ (Memory Corruption)
Mô tả: Viết đè lên vùng bộ nhớ không thuộc quyền sở hữu của đối tượng.
Nguyên nhân: Truy cập vượt quá giới hạn của vùng bộ nhớ cấp phát hoặc viết đè lên bộ nhớ đã giải phóng.
8. Race Conditions (Trong lập trình đa luồng)
Mô tả: Hai hoặc nhiều luồng cùng truy cập và thay đổi cùng một vùng bộ nhớ mà không có sự đồng bộ hóa hợp lý.
Nguyên nhân: Thiếu khóa (locks) hoặc các cơ chế đồng bộ hóa khác khi truy cập tài nguyên chia sẻ.
9. Lỗi quản lý tài nguyên (Resource Management Errors)
Mô tả: Quên giải phóng tài nguyên khác ngoài bộ nhớ như file handles, mutexes, sockets, vv.
Nguyên nhân: Không quản lý tài nguyên đúng cách, đặc biệt khi sử dụng con trỏ để quản lý các tài nguyên này.
10. Pointer Arithmetic Errors
Mô tả: Các lỗi xảy ra khi thực hiện phép tính trên con trỏ dẫn đến việc trỏ đến vùng bộ nhớ không hợp lệ.
Nguyên nhân: Tính toán sai địa chỉ bộ nhớ hoặc vượt quá phạm vi mảng.
11. Aliasing Issues
Mô tả: Một vùng bộ nhớ được trỏ đến bởi nhiều con trỏ, dẫn đến khó khăn trong việc quản lý quyền sở hữu và vòng đời của đối tượng.
Nguyên nhân: Nhiều con trỏ trỏ tới cùng một vùng bộ nhớ mà không có quy tắc rõ ràng về quyền sở hữu.
12. Complexity in Managing Lifetime and Ownership
Mô tả: Việc quản lý vòng đời và quyền sở hữu của đối tượng trở nên phức tạp, dễ dẫn đến lỗi.
Nguyên nhân: Thiếu các quy tắc và mô hình rõ ràng để quản lý con trỏ và bộ nhớ.

Introduction to smart pointer

- Problem with normal pointer ▪ Vấn đề với con trỏ thông thường
  - ✓ C++ doesn't have automatic garbage collection C++ không có cơ chế thu gom rác tự động
  - ✓ App is responsible for returning all acquired resources to operating system
    chịu trách nhiệm hệ điều hành
    Ứng dụng phải tự chịu trách nhiệm trả lại tất cả các tài nguyên đã thu nhận về hệ điều hành
  - ✓ Failure to release an unused resource is called a leak Việc không giải phóng một tài nguyên không sử dụng được gọi là rò rỉ (leak)
    trả lại
  - ✓ Leaked resources are unavailable to other programs until the process exits
    không thể được sử dụng
    Tài nguyên bị rò rỉ không thể được sử dụng bởi các chương trình khác cho đến khi quá trình kết thúc
  - ✓ Memory leaks in particular are a common cause of bugs in C-style programming
    phổ biến
    Rò rỉ bộ nhớ đặc biệt là nguyên nhân phổ biến gây ra lỗi trong lập trình kiểu C

=> Difficult to ensure that programs are free of memory and resource leaks and are exception-safe

chắc chắn

=> Khó đảm bảo rằng các chương trình không bị rò rỉ bộ nhớ và tài nguyên, cũng như xử lý ngoại lệ.

Introduction to smart pointer

- Smart pointer
  - ✓ A wrapper class over a pointer with an operator like * and -> overloaded
    Một class bao bọc con trỏ với các toán tử như * và -> được nạp chồng
  - ✓ Declared on stack and initialized by using a raw pointer that points to a heap-allocated object
    con trỏ bình thường
    Được khai báo trên stack và khởi tạo bằng cách sử dụng một con trỏ thô trỏ đến một đối tượng được cấp phát trên heap
  - ✓ Smart pointer look like a pointer but can do many things that a normal pointer can't like automatic destruction, reference counting and more…
    Con trỏ thông minh trông giống như con trỏ nhưng có thể làm nhiều thứ mà con trỏ thông thường không thể, như tự động hủy, đếm tham chiếu và nhiều hơn nữa
  - ✓ When smart pointer object out of scope, destructor is automatically called. So, the dynamically allocated memory would automatically be deleted (or reference count can be decremented…)
    Khi đối tượng con trỏ thông minh ra khỏi phạm vi, hàm hủy sẽ được gọi tự động. Do đó, bộ nhớ được cấp phát động sẽ tự động bị xóa (hoặc số lượng tham chiếu có thể bị giảm...)

Introduction to smart pointer

```
class SmartPtr
{
private:
   int* m_ptr; // actual pointer

public:
   explicit SmartPtr(int* p = NULL)
   {
      m_ptr = p;
   }
   ~SmartPtr()
   {
      if (m_ptr != NULL)
      {
         delete (m_ptr);
      }
   }

   // overloading dereferencing operator
   int& operator*()
   {
      return *m_ptr;
   }
};
```

từ khóa explicit ngăn chặn tất cả các chuyển đổi ngầm định có thể xảy ra
- Chuyển đổi ngầm định là quá trình mà trình biên dịch tự động chuyển đổi một kiểu dữ liệu này sang kiểu dữ liệu khác mà không cần sự can thiệp rõ ràng từ lập trình viên
-explicit đặc biệt thường sử dụng với constructor

```
int main()
{
   SmartPtr ptr(new int());
   *ptr = 20;
   cout << *ptr;

   // We don't need to call delete ptr when the object
   // ptr goes out of scope, the destructor for it is automatically
   // called and destructor does delete ptr

   return 0;
}
```

Types of smart pointer

- There are 3 types of smart pointer:
  - ✓ Unique pointer (unique_ptr)
  - ✓ Shared pointer (shared_ptr)
  - ✓ Weak pointer (weak_ptr)

move semantic
move constructor

r value, l value

Types of smart pointer

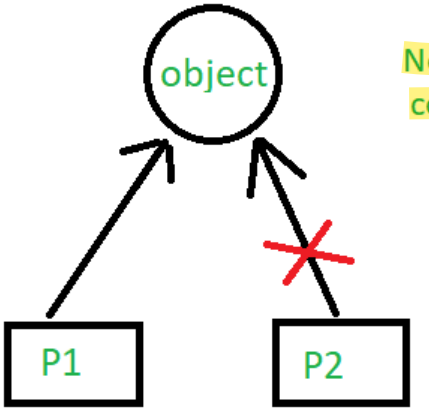
- **Unique pointer (unique_ptr)** con trỏ duy nhất
  - ✓ Allows exactly one owner of the underlying pointer. With one object is created, only one pointer can point to this object at one time

con trỏ gốc

Cho phép chỉ có duy nhất một chủ sở hữu của con trỏ gốc. Khi một đối tượng được tạo ra, chỉ có một con trỏ có thể trỏ đến đối tượng này tại một thời điểm.

  - ✓ Can't share with another pointer but can transfer the control to another pointer by removing old pointer (using std::move)

Không thể chia sẻ với con trỏ khác nhưng có thể chuyển quyền kiểm soát sang con trỏ khác bằng cách loại bỏ con trỏ cũ (sử dụng std::move).

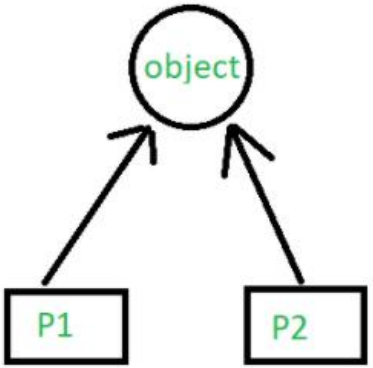


Note: But we can transfer the control to P2 by removing P1

```
int main()
{
    unique_ptr<Rectangle> P1(new Rectangle(10, 5));
    cout << P1->area() << endl; // This'll print 50

    // unique_ptr<Rectangle> P2(P1); => don't allow

    unique_ptr<Rectangle> P2;
    P2 = move(P1);

    cout << P2->area() << endl; // This'll print 50
    //cout << P1->area() << endl;  => crash, need check NULL

    return 0;
}
```

Types of smart pointer

- Shared pointer (shared_ptr)
  - ✓ Allow more than one pointer can point to one object at one time
    Cho phép nhiều con trỏ có thể trỏ đến một đối tượng tại cùng một thời điểm.
  - ✓ Object is not deleted until all pointers out of scope or given up ownership
    Đối tượng sẽ không bị xóa cho đến khi tất cả các con trỏ ra khỏi phạm vi hoặc từ bỏ quyền sở hữu.
  - ✓ Shared pointer has two pointers, one for object and one for shared control block that contains reference count
    Con trỏ chia sẻ có hai con trỏ: một cho đối tượng và một cho khối điều khiển chia sẻ chứa số lượng tham chiếu.



Note:It maintains a Reference Counter by using use_count() method

Here Reference Counter is 2

```
int main()
{
    shared_ptr<Rectangle> P1(new Rectangle(10, 5));
    cout << P1->area() << endl; // This'll print 50

    shared_ptr<Rectangle> P2;
    P2 = P1;

    cout << P2->area() << endl;   // This'll print 50
    cout << P1->area() << endl;   // This'll now not give an error
    cout << P1.use_count() << endl;  // This'll print 2 as reference count

    return 0;
}
```

Types of smart pointer

- Shared pointer (shared_ptr)
  - ✓ Circular reference => cannot delete objects

```
class ClassA
{
public:
   shared_ptr<ClassB> m_data;

   ClassA() { cout << "A constructor" << endl; }
   ~ClassA() { cout << "A destructor" << endl; }
};

class ClassB
{
public:
   shared_ptr<ClassA> m_data;

   ClassB() { cout << "B constructor" << endl; }
   ~ClassB() { cout << "B destructor" << endl; }
};
```

```
int main()
{
   shared_ptr<ClassA> sharedA(new ClassA());
   shared_ptr<ClassB> sharedB(new ClassB());

   cout << "ref count A: " << sharedA.use_count() << endl;
   cout << "ref count B: " << sharedB.use_count() << endl;

   sharedA->m_data = sharedB;
   sharedB->m_data = sharedA;

   cout << "ref count A: " << sharedA.use_count() << endl;
   cout << "ref count B: " << sharedB.use_count() << endl;

   return 0;
}
```

Types of smart pointer

- Weak pointer (weak_ptr)

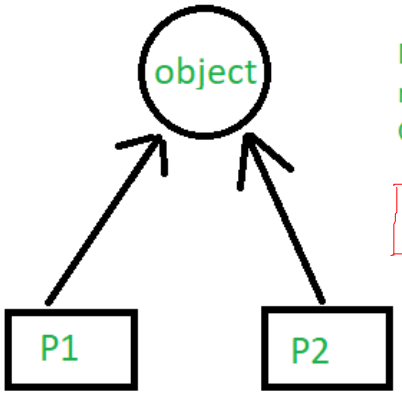
Cung cấp quyền truy cập đến một đối tượng được sở hữu bởi một hoặc nhiều đối tượng shared_ptr, nhưng không tham gia vào việc đếm số lượng tham chiếu.

  - ✓ Provide access to an object that is owned by one or more shared_ptr instances, but does not participate in reference counting
  - ✓ Use when you want to observe an object, but do not require it to remain alive

Sử dụng khi bạn muốn quan sát một đối tượng, nhưng không yêu cầu đối tượng đó phải tồn tại.(không ngăn cản quá trình giải phóng đối tượn từ share_ptr)

  - ✓ Often use to break circular reference between shared pointer instances

Thường được sử dụng để phá vỡ vòng tham chiếu giữa các đối tượng shared_ptr.



Note:It does not maintain a Reference Counter

quan sát đối tượng tức là không có operator * và ->; nên cần gọi thêm lock() để sử dụng

```
int main()
{
   shared_ptr<Rectangle> sp(make_shared<Rectangle>(10, 5));
   cout << sp->area() << endl; // This'll print 50

   weak_ptr<Rectangle> wp;
   wp = sp;

   cout << wp.lock()->area() << endl;  // This'll print 50
   cout << sp.use_count() << endl;     // This'll print 1 as reference count

   return 0;
}
```

Section 2

Multi threading

Multi threading. Agenda

- Program, Process and Thread
- Multi-threading in C++
- Multi-threading Problem

1. Chương trình (Program):
Định nghĩa: Chương trình là một tập hợp các hướng dẫn được viết bằng ngôn ngữ lập trình để thực hiện một nhiệm vụ cụ thể trên máy tính.
Chương trình được lưu trữ trong các tệp tin thực thi (executable files) và có thể được thực thi để thực hiện các công việc như tính toán, quản lý dữ liệu, giao tiếp với người dùng, và nhiều tác vụ khác.
Ví dụ: Một chương trình đơn giản có thể là một ứng dụng soạn thảo văn bản như Microsoft Word, một trò chơi như Minecraft, hay một trình duyệt web như Google Chrome.

2. Tiến trình (Process):
Định nghĩa: Tiến trình là một chương trình đang được thực thi trên hệ điều hành. Mỗi tiến trình có một không gian bộ nhớ riêng, thực thi các hướng dẫn, và có thể có nhiều luồng chạy song song trong nó.
Đặc điểm:
Mỗi tiến trình có thể có một hoặc nhiều luồng.
Tiến trình có các tài nguyên riêng như bộ nhớ, bảng định danh, tài nguyên hệ thống.Được quản lý bởi hệ điều hành và có thể được tạo, sắp xếp, và hủy bỏ bởi hệ điều hành.
Ví dụ: Khi bạn mở một ứng dụng như Photoshop trên máy tính của bạn, hệ điều hành sẽ tạo một tiến trình Photoshop để thực thi các lệnh từ chương trình Photoshop. Mỗi bản sao của Photoshop đang chạy trên máy tính của bạn là một tiến trình riêng biệt.

3. Luồng (Thread):
Định nghĩa: Luồng là một con đường thực thi độc lập trong một tiến trình. Mỗi tiến trình có ít nhất một luồng thực thi, gọi là luồng chính (main thread), và có thể có nhiều luồng khác chạy song song.

Đặc điểm:
Các luồng trong cùng một tiến trình chia sẻ không gian bộ nhớ và các tài nguyên khác.
Các luồng cho phép thực hiện đồng thời các tác vụ, tăng hiệu suất của ứng dụng.Luồng được quản lý bởi hệ điều hành và có thể được tạo, kết thúc, và lập lịch thực thi bởi hệ điều hành.

Ví dụ: Trong một trò chơi đa luồng, có thể có một luồng chịu trách nhiệm cho việc vẽ hình ảnh trò chơi (rendering), một luồng khác quản lý đầu vào từ bàn phím và chuột của người dùng, và một luồng khác để xử lý âm thanh. Tất cả các luồng này chạy đồng thời trong cùng một tiến trình game.

Program, Process and Thread

- Program

```
#include <iostream>

int main(void) {
   std::cout << “hello world” << endl;
   return 0;
}
```

Program, Process and Thread

quá trình thực thi 1 chương trình trên hệ điều hành (chung cho tất cả các loại phần mềm như game, ứng dụng, IDE,...)

- **Program Execution**
  - ✓ Running a program from shell:

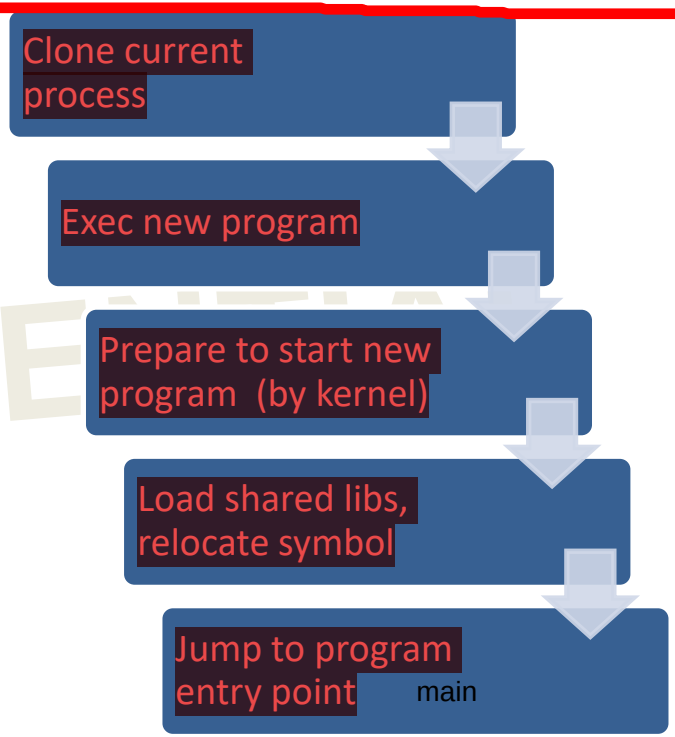
$ ./helloworld

Clone current process:
Quá trình bắt đầu khi hệ điều hành sao chép (clone) quá trình hiện tại. Điều này bao gồm việc sao chép bộ nhớ, trạng thái và các tài nguyên cần thiết từ quá trình gốc để tạo ra một bản sao gọi là quá trình con. Quá trình con này thường được sử dụng khi hệ điều hành cần tạo ra nhiều bản thể của cùng một chương trình hoặc khi cần tạo ra các tiến trình con để thực hiện các tác vụ cụ thể.

Exec new program:
Sau khi quá trình con được sao chép, hệ điều hành thực hiện hàm exec (execute) để thay thế nội dung của quá trình con bằng một chương trình mới cần thực thi.Bước này là quá trình load chương trình mới vào bộ nhớ và chuẩn bị các tài nguyên cần thiết cho việc thực thi của chương trình mới.

Prepare to start new program (by kernel):
Kernel (nhân của hệ điều hành) chuẩn bị môi trường thực thi cho chương trình mới. Điều này bao gồm thiết lập các thông số môi trường, thực hiện các phân quyền cần thiết (ví dụ như quyền truy cập vào tệp tin, quyền thực thi, quyền ghi vào bộ nhớ,...).

Load shared libraries, relocate symbols:
Trước khi chương trình mới có thể thực thi, hệ điều hành cần phải tải các thư viện chia sẻ (shared libraries) cần thiết vào bộ nhớ. Đây là các thư viện đã được biên dịch trước và được sử dụng chung bởi nhiều chương trình khác nhau.Khi các thư viện chia sẻ được tải vào bộ nhớ, các biểu tượng (symbols) trong các thư viện này cần được điều chỉnh (relocate) để phù hợp với vị trí bộ nhớ cụ thể mà chúng được tải vào.
Jump to program entry point:
Cuối cùng, sau khi đã chuẩn bị xong môi trường và các thư viện chia sẻ, hệ điều hành thực hiện việc nhảy (jump) đến điểm nhập chương trình (program entry point). Điểm này thường là hàm main() trong chương trình C/C++.Tại điểm này, quá trình thực thi chính thức bắt đầu và chương trình mới bắt đầu thực thi các lệnh được định nghĩa trong hàm main().



Program, Process and Thread

- Process tiến trình
  - ✓ Code segment
  - ✓ Data segment
  - ✓ BSS segment biến static chưa được gán giá trị
  - ✓ Stack segment
  - ✓ Heap segment

Program, Process and Thread

- Thread
  - ✓ A sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing a same address space

    Một chuỗi các lệnh có thể được thực thi đồng thời với các chuỗi lệnh khác trong môi trường đa luồng, trong khi chia sẻ cùng một không gian địa chỉ.
  - ✓ A thread is a path of execution within a process. A process can contain multiple threads

    Một luồng là một đường lối thực thi trong một tiến trình. Một tiến trình có thể chứa nhiều luồng.

Khi Sử Dụng Nhiều Hơn Số Lượng Luồng Được Cấp Phát
Nếu một chương trình C++ sử dụng nhiều hơn số lượng luồng phần cứng có sẵn, hệ điều hành sẽ quản lý việc chuyển đổi giữa các luồng này. Điều này có thể dẫn đến việc chuyển đổi luồng (context switching), điều mà RTOS (Real-Time Operating System) cũng thực hiện. Tuy nhiên, trong các hệ điều hành không phải RTOS, việc chuyển đổi này không được ưu tiên theo thời gian thực. Hệ điều hành chuyển sang chạy so le:Khi số luồng vượt quá số luồng phần cứng, hệ điều hành sẽ chia sẻ thời gian CPU giữa các luồng này bằng cách chuyển đổi luồng. Điều này có thể dẫn đến overhead từ việc chuyển đổi luồng, nhưng đảm bảo rằng tất cả các luồng đều có cơ hội để chạy.

Program, Process and Thread

- Process vs Thread

  ✓ Threads within the same process run in a shared memory space, while processes run in separate memory spaces

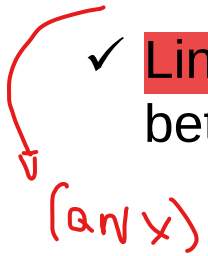
Các luồng trong cùng một tiến trình chạy trong không gian bộ nhớ chung, trong khi các tiến trình chạy trong các không gian bộ nhớ riêng biệt.

  ✓ Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space

Các luồng không độc lập với nhau như các tiến trình, do đó các luồng chia sẻ với các luồng khác phần mã, phần dữ liệu và tài nguyên hệ điều hành (như các tệp mở và tín hiệu). Nhưng giống như tiến trình, một luồng có bộ đếm chương trình (PC), tập thanh ghi và không gian ngăn xếp riêng của nó.

Program, Process and Thread

- Types of thread
  - ✓ POSIX thread: usually referred to as pthread, user level thread
    - ✓ POSIX thread: Thường được gọi là pthread, là luồng ở mức người dùng (user level thread).
  - ✓ Linux kernel thread: low level thread, using task (not different between thread and process)
    - ✓ Linux kernel thread: Luồng ở mức thấp (low level thread), sử dụng task (không có sự khác biệt giữa luồng và tiến trình).



Program, Process and Thread

- Advantages of Thread over Process
  - ✓ Responsiveness
  - ✓ Faster context switch
  - ✓ Effective utilization of multiprocessor system
  - ✓ Resource sharing
  - ✓ Communication
  - ✓ Enhanced throughput of the system

Ưu Điểm của Luồng So Với Tiến Trình

✓ **Độ nhạy**: Luồng có thể tăng cường độ phản hồi của ứng dụng.

✓ **Chuyển ngữ cảnh nhanh hơn**: Chuyển đổi ngữ cảnh giữa các luồng nhanh hơn so với giữa các tiến trình.

✓ **Sử dụng hiệu quả hệ thống đa bộ xử lý**: Luồng giúp tận dụng tối đa khả năng của hệ thống đa bộ xử lý.

✓ **Chia sẻ tài nguyên**: Các luồng trong cùng một tiến trình có thể chia sẻ tài nguyên với nhau.

✓ **Giao tiếp**: Giao tiếp giữa các luồng dễ dàng và hiệu quả hơn so với giữa các tiến trình.

✓ **Tăng thông lượng của hệ thống**: Luồng giúp nâng cao thông lượng và hiệu suất của hệ thống.

Multi-threading in C++

- Using pthread API

thư viện do hệ điều hành cung cấp, phổ biến hơn

```
struct thread_data {
   int id;
   string msg;
};

void *testFunc(void *threadarg)
{
   thread_data *data = (thread_data *)threadarg;
   cout << "id: " << data->id << ", msg: " << data->msg << endl;
}

int main()
{
   pthread_t threadIds[10];
   thread_data threadData[10];
   for(int i = 0; i < 10; i++)
   {
      cout << "create thread: " << i << endl;
      threadData[i].id = i;
      threadData[i].msg = "Hello " + std::to_string(i);
      pthread_create(&threadIds[i], NULL, testFunc, (void *)&threadData[i]);
   }
   pthread_exit(NULL);
   cout << "exit program"<< endl;
   return 0;
}
```

Multi-threading in C++

- Using std::thread chỉ dùng trong C++

```
void testFunc(int id, string msg)
{
   cout << "id: " << id << ", msg: " << msg << endl;
}

int main()
{
   for(int i = 0; i < 10; i++)
   {
      cout << "create thread: " << i << endl;
      thread t(&testFunc, i, "Hello " + std::to_string(i));
      t.detach();
   }

   cout << "exit program"<< endl;
   return 0;
}
```

Multi-threading Problem

- Data race
- Deadlock
- Memory ordering

Data race

```
// test_datarace.cpp
#include <iostream>
#include <thread>

int count = 0;

void increaseTask()
{
   int i= 10000000;
   while(--i >= 0) {
      ++count;
   }
}
```

```
int main() {
   std::thread t1(increaseTask);
   std::thread t2(increaseTask);

   t1.join();
   t2.join();

   std::cout  << count << std::endl;
   return 0;
}
```

Data race

```
// count++;

ldr r3, [r2]
add r3, r3, #1
str r3, [r2]
```

```
// thread 1
LOAD (0)
ADD (0 -> 1)
STORE (1)



LOAD(2)


ADD(2->3)
STORE(3)
```

```
// thread 2


LOAD(1)
ADD (1 -> 2)
STORE (2)

LOAD(2)
ADD(2->3)
STORE(3)
```

Data Race Detection

- Review code
- Tool: valgrind, sanitizer

Data Race Prevention

- Mutex
- Atomic
- Change implementation

- smaphore

Mutex

```
// test_thread.cpp
#include <iostream>
#include <thread>
#include <mutex>

int count = 0;
std::mutex countMutex;

void increaseTask()
{
   int i= 10000000;
   while(--i >= 0) {
      countMutex.lock();
      ++count;
      countMutex.unlock();
   }
}
```

```
int main() {
   std::thread t1(increaseTask);
   std::thread t2(increaseTask);

   t1.join();
   t2.join();

   std::cout  << count << std::endl;
   return 0;
}
```

Mutex Using Cost

- Using cost
  - ✓ Locking overhead
  - ✓ Locking contention
- Tool: valgrind, perf, Intel VTune

Does read need to be protected?

```
// test_thread.cpp
#include <iostream>
#include <thread>
#include <mutex>

int count = 0;
std::mutex  countMutex;

void increaseTask() {
   while(true) {
      countMutex.lock();
      ++count;
      countMutex.unlock();
   }
}
void readTask() {
   while(true) {
      std::cout << count << std::endl;
   }
}
```

```
int main() {
   std::thread t1(increaseTask);
   std::thread t2(increaseTask);
   std::thread t3(readTask);

   t1.join();
   t2.join();
   t3.join();
   return 0;
}
```

Deadlock

2 mutex đợi nhau

```
// test_deadlock.cpp
#include <iostream>
#include <thread>
#include <mutex>

int count1 = 0;
Int count2 = 0;
std::mutex count1Mutex;
std::mutex count2Mutex;

void increaseTask1() {
   while(true) {
      count1Mutex.lock();
      count2Mutex.lock();
      ++count1;
      ++count2;
      count2Mutex.unlock();
      count1Mutex.unlock();
   }
}
```

```
void increaseTask2() {
   while(true) {
      count2Mutex.lock();
      count1Mutex.lock();
      ++count1;
      ++count2;
      count1Mutex.unlock();
      count2Mutex.unlock();
   }
}
int main() {
   std::thread t1(increaseTask1);
   std::thread t2(increaseTask2);
   t1.join();
   t2.join();
   return 0;
}
```

Deadlock Stack

```
(gdb) thread apply all bt
Thread 3 (Thread 0x7fa150328700 (LWP 5611)):
#0  __lll_lock_wait () at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1  0x00007fa150a05dbd in __GI___pthread_mutex_lock (mutex=0x605340 <count2Mutex>) at
../nptl/pthread_mutex_lock.c:80
#2  0x0000000000401007 in __gthread_mutex_lock(pthread_mutex_t*) ()
#3  0x0000000000401460 in std::mutex::lock() ()
#4  0x00000000004010de in increaseTask1() ()

Thread 2 (Thread 0x7fa14fb27700 (LWP 5612)):
#0  __lll_lock_wait () at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1  0x00007fa150a05dbd in __GI___pthread_mutex_lock (mutex=0x605300 <count1Mutex>) at
../nptl/pthread_mutex_lock.c:80
#2  0x0000000000401007 in __gthread_mutex_lock(pthread_mutex_t*) ()
#3  0x0000000000401460 in std::mutex::lock() ()
#4  0x000000000040112a in increaseTask2() ()
```

Deadlock Prevention

- Using cost
  - ✓ Locking overhead
  - ✓ Locking contention
- Tool: valgrind, perf, Intel VTune

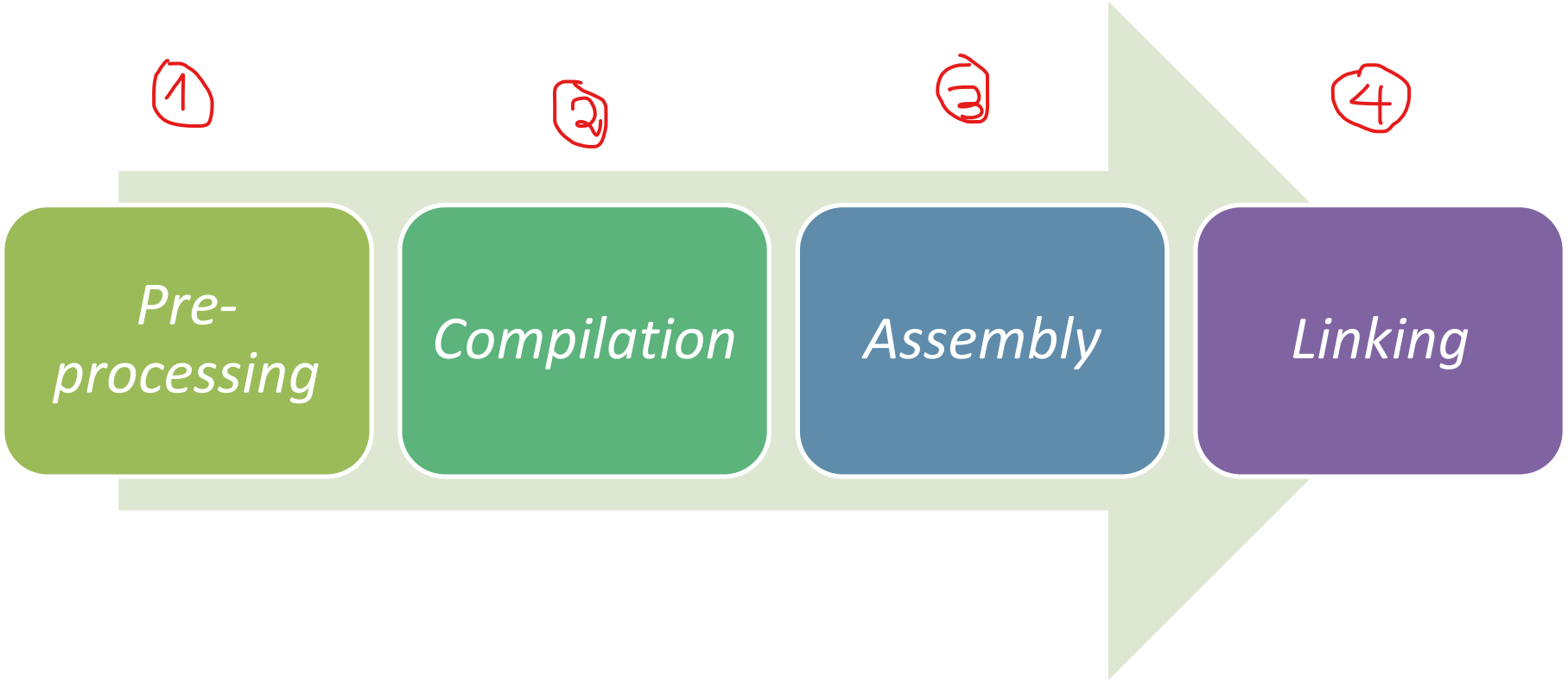
Section 3

Compiling C++ program

Compiling C++ program. Agenda

- Compilation process
- Macro vs inline function vs function

Compilation process



Pre-processing

- Removal of Comments
- Expansion of Macros
- Expansion of the included files
- Conditional compilation

Compilation

- Compile and produce an intermediate compiled output file
- This file contains assembly level instructions

Assembly

- Convert intermediate compiled output file into machine language
- This file contains machine level instructions

Linking

phát hiện ra lỗi ở đây

- This is the final phase in which all the linking of function calls with their definitions are done
- Linker knows where all these functions are implemented
- Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends

Macro vs inline function vs function

```
#define ADD_MACRO(a, b) a + b

int add_func(int a, int b)
{
   return a + b;
}

int main()
{
   cout << 5 * add_func(2, 3) << endl;
   cout << 5 * ADD_MACRO(2, 3) << endl;

   return 0;
}t
```

trình biên dịch tự động tạo các hàm rất ngắn thành hàm inline

phân biệt macro, inline function và function

Section 4

Static vs dynamic library

Static vs dynamic library. Agenda

- What is library?
- Static library vs dynamic library

What is library?

- A package of code that is reused by many programs
- Library contains two pieces
  - ✓ Header file that defines the functionality the library is exposing (offering) to the programs using it
  - ✓ Precompiled binary that contains the implementation of that functionality pre-compiled into machine language

Static library vs dynamic library

Static library	Dynamic library
Consists of routines that are compiled and linked directly into program flie .exe đã chứa thư viện tĩnh sau khi biên dịch	Consists of routines that are loaded into application at run time khi chạy thì thư viện động mới được add
Become part of executable => only have to distribute executable for users to run program	Do not become part of executable => have to distribute executable and libraries for users to run program
Ensure right version of library is always used with program	Do not ensure right version of library is used with program => must check version
Do not share between programs => waste memory	Share between programs => save memory có thể chia sẻ giữa các thư viện khác nhau
Can not be upgraded easy	Can be upgraded to newer version without replacing all of executables

References

- https://www.geeksforgeeks.org/smart-pointers-cpp/
- https://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm
- https://www.geeksforgeeks.org/thread-in-operating-system/
- https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/
- http://www.learncpp.com/cpp-tutorial/a1-static-and-dynamic-libraries/

Lesson Summary

- Smart pointer
- Multi threading
- Compiling C++ program
- Static vs dynamic library

Thank you