

CS441 4 Recitation 4

C++ class and multithreading

09/2025

Alicia Yang, Shouxu Lin

Logistics

- **Session 4: prelim review**

- Time: 7:15 PM – 8:15 PM, Monday (09/22)
- Led by TAs: Nam Anh Dang, Jephthah Kwame Mensah
- Location: TBA

- **Session 2: HW1 part 2 help session**

- Time: 2PM – 5 PM, Sunday (9/21)
- Led by TAs: Briaana Liu, Ruichen Bao
- Location: Gates 122

Overview

- C++ template continue
 - Variadic templates
- Multithreading
 - What is concurrency
 - Threads launching
 - Thread finishing

More fun facts left from last recitation

C++ Class revisit

Copy constructor

- **Create** a new object by **initializing** it with an object of the same class
- Called when
 - Initialization `Rectangle obj2 (obj1);`
 - Function argument passing by value `func(Rectangle obj);`
 - Function return by value `return obj;`

// Note: returning a local created variable by value, will be optimized by c++ 17+, via RVO(copy elision) that could avoid copy. It constructs the return object directly in return's storage. (https://en.cppreference.com/w/cpp/language/copy_elision.html)

Implicitly-defined default copy-constructor

- If no user-defined copy constructor, the compiler declare and define a copy constructor
 - It performs member-wise copy of the object's bases and members to the new object it initializes
 - Default constructor does only **shallow** copy

myIntVector example

```
class myIntVector{
public:
    int* data;
    size_t size;
    size_t capacity;

    myIntVector();
    myIntVector(size_t s);
    ~myIntVector();
    ...
};
```

```
myIntVector::myIntVector(size_t s) {
    size = s;
    capacity = s;
    data = new int[capacity];
    for (size_t i = 0; i < size; ++i) {
        data[i] = 0;
    }
}

myIntVector::~~myIntVector(){
    delete[] data;
}
```


Default copy-constructor

```
myIntVector vect1 = myIntVector(3);
```

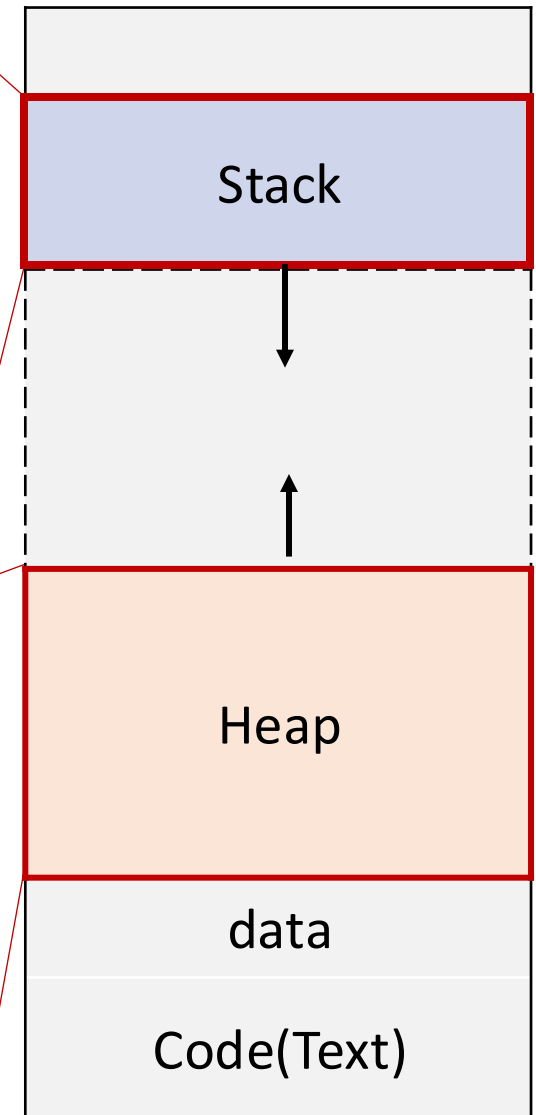
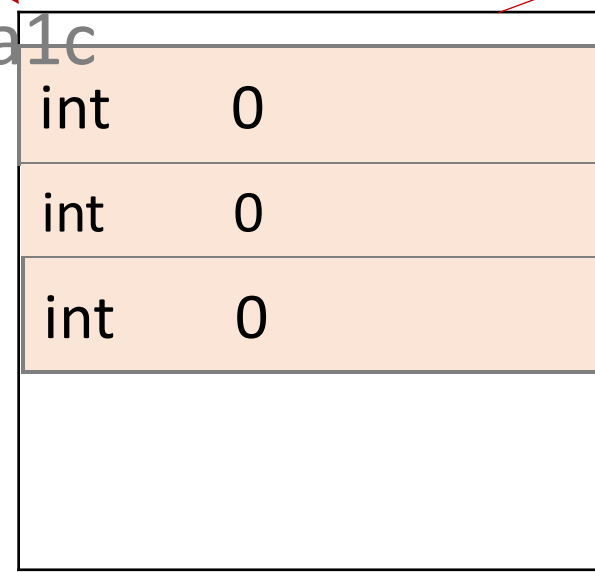
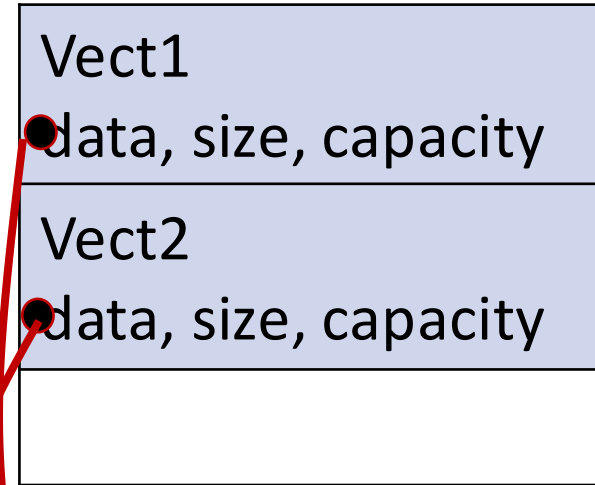
```
myIntVector vect2 = vect1;
```

```
vect1.data = 5;
```



Not ideal, because changing only one changes both of them.
Want two identical independent objects

0x7cd10a1c



Fix: User-defined copy constructor

```
myIntVector(const myIntVector& other) :
```

```
    size(other.size), data(new int[other.size]) {
```

```
        for (size_t i = 0; i < size; ++i) {
```

```
            data[i] = other.data[i];
```

```
        }
```

```
    }
```

Deep copy the object's
members

Move constructor

```
class myIntVector{
```

```
    myIntVector(myIntVector && other);
```

```
}
```

```
// Transfer the ownership of the resources from the  
object, other, to the new object
```

Move constructor

- Transfer the ownership of resources from one object to another, instead of making a copy
- Called when
 - Initialization `Rectangle obj2(std::move(obj1));`
 - Move smart pointers `std::unique_ptr<int> p2 = std::move(p1);`
 - Function return with Return Value Optimization(RVO), or return a named local and want to force a move without(RVO) `return std::move(obj);`

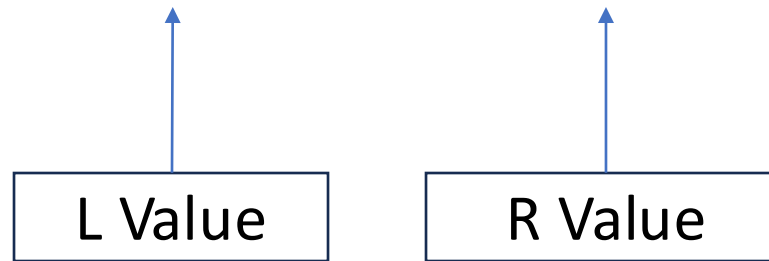
Why use move constructor?

- Improve the performance of the program by avoiding the overhead caused by unnecessary copying.

```
myIntVector(myIntVector&& other) : size(0), data(nullptr) {  
    data = other.data;           // copy the pointer of the memory  
    size = other.size;           address of other.data  
  
    other.data = nullptr;        // Transfer the ownership of other's  
    other.size = 0;              resource to this new object  
  
}
```

lvalue and rvalue

int var = 25;



- L value refers to a **memory location** with a **name** (e.g., a variable).
- It typically has a lifetime longer than a single expression or statement.
- E.g. variable var continue to exist after this line of code.

- R value refers to a **temporary** value that does not have a persistent memory location.
- Only exist within the scope of the expression in which they are used.

Copy assignment

- Defines what happens when an **already existing object** is **assigned** the value of **another object** of the same class, using a **copy** of the source's data
- Called when
 - Assign to an already existing object

```
Rectangle obj2;  
obj2 = obj1;
```

- the move assignment operator is used when you assign an object that is an rvalue (a temporary object or one you explicitly cast with `std::move`) to another existing object.
- Called when
 - Assign to an already existing object

```
std::unique_ptr<int> p1 = std::make_unique<int>(42); // owns 42  
std::unique_ptr<int> p2 = std::make_unique<int>(99); // owns 99  
p2 = p1; // move assignment
```


C++ Template continue

Templated function

template < *parameter-list* > *function-declaration*

```
template <typename T>
T subtract(T a, T b) {
    return a - b;
}
```

```
int main(){
    int x = 10;
    int y = 7;
    std::cout << subtract(x, y) << std::endl;

    double p = 5.5;
    double q = 2.2;
    std::cout << subtract(p, q) << std::endl;
    ...}
```

typename is the type-parameter-key, which could be either **typename** or **class**.

T is the name of **type template parameter**. Tells the compiler *"I will use a placeholder type T in this function."*

Templated class

template < *parameter-list* > *class-declaration*

```
template <typename T>
class Subtracter {
public:
    T subtract(T a, T b) {
        return a - b;
    }
};

int main() {
    Subtracter<double> double_sub;
    std::cout << double_sub.subtract(5.5, 2.2) << std::endl;
    ...}
```

Parameter list

```
template <typename T, int N>
class Array {
    T data[N]; // size N known at compile time
public:
    int size() const { return N; }
};

int main() {
    Array<int, 5> arr1; // array of 5 ints
    std::cout << arr1.size() << std::endl; // 5
    ...}
```

Non-type template parameter

- A variable for a constant (e.g., an int ...)
- Known at compile time

Type template parameter

- A variable for a type
- Known at compile time

Template instantiation

- A function template or a class template **by itself** is **not** a type, or a function, or any other entity.
- **No code is generated** from a source file that contains **only** template definitions.
- For any code to appear, a template must be **instantiated**: the template arguments must be determined so that the compiler can generate an actual function

Template instantiation - explicit instantiation

```
template <typename T>
T subtract(T a, T b) {
    return a - b;
}
```

```
template int subtract<int>(int, int);    // Explicit instantiation declarations
template double subtract<double>(double, double);    // Explicit instantiation declarations

int main(){
    int x = 10, y = 7;
    std::cout << subtract(x, y) << std::endl;    // Use subtract<int>
    double p = 5.5, q = 2.2;
    std::cout << subtract(p, q) << std::endl;    // Use subtract<double>
    ...}
```

Template instantiation - implicit instantiation (default)

```
template <typename T>
T subtract(T a, T b) {
    return a - b;
}
```

- The compiler generates code, in this case **subtract<int>**
- If subtract was called on a double another function subtract (overload) will be generated with T = double

```
int main(){
    int x = 10;
    int y = 7;
    std::cout << subtract(x, y) << std::endl; // Compiler generate subtract<int>
    double p = 5.5, q = 2.2;
    std::cout << subtract(p, q) << std::endl; // Compiler generate subtract<double>
    ...}
```

Quick aside: template .hpp files don't come with associated .cpp files

- A template is a “pattern” that the compiler uses to generate a family of classes or functions
- For the compiler to generate the code, it must see both the template definition and the specific types used to “fill in” the template.
 - For example, if you're trying to use a `subtract<int>`, the compiler must see both the `subtract` template and the fact that you're trying to make a specific `subtract<int>`

How do we use templates when our function has an arbitrary number of parameters?

- Common issue...
 - Solution: Variadic templates
 - Let's code

Variadic templates

```
class car {  
public:  
    int price;  
    car(int price) :price(price) {}  
};
```

```
class pc {  
public:  
    int price;  
    pc(int price) : price(price) {}  
};
```

```
class pen {  
public:  
    int price;  
    pen(int price) : price(price) {}  
};
```

Variadic templates

```
int sum() {  
    return 0;  
}  
template <typename T, typename... Args>  
int sum (T item, Args... rest) {  
    return item.price + sum(rest...);  
}  
  
int main() {  
    car c(100);  
    pc pc(10);  
    pen p(1);  
    std::cout << "The sum is " << sum(c, pc, p);  
}
```

Templates in the perspective of programming

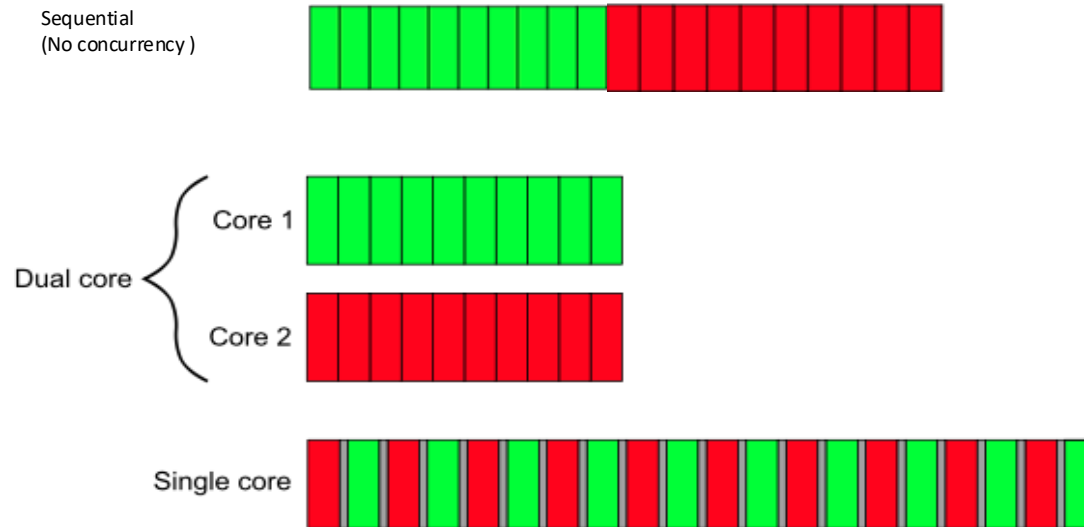
- Avoid code duplication
 - Functions are blocks of organized
 - Reusable code that model a particular action
 - Classes model similar set of objects
 - Libraries provide a consistent set of features
- Performance (compile-time resolution)
 - Templates are expanded by the compiler for the types you use

Multithreading

- What is concurrency
- Threads launching
- Thread finishing
- Threads safety

Concurrency

- What is concurrency?
 - a single system performs multiple independent activities in parallel

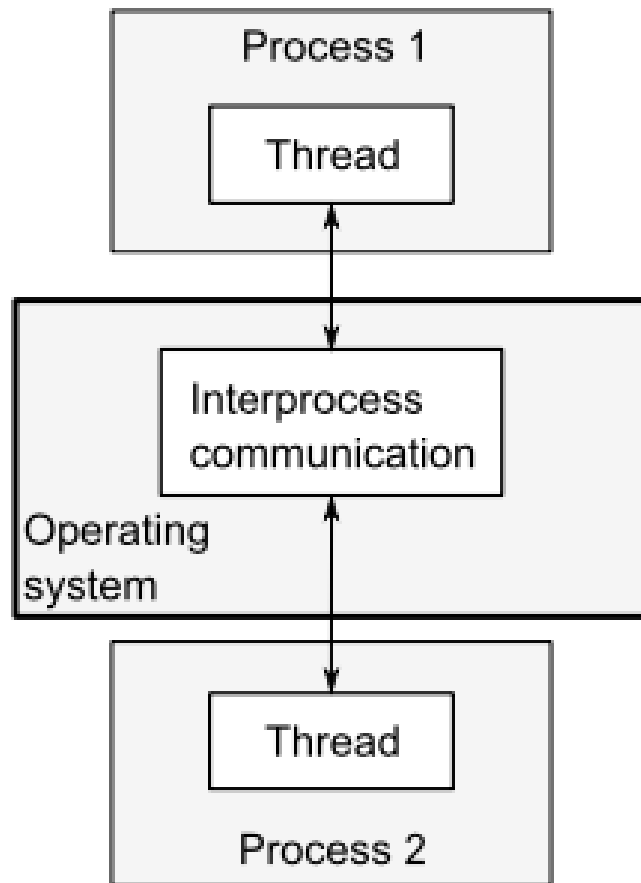


- Why use concurrency?
 - Separation of concerns
 - Performance

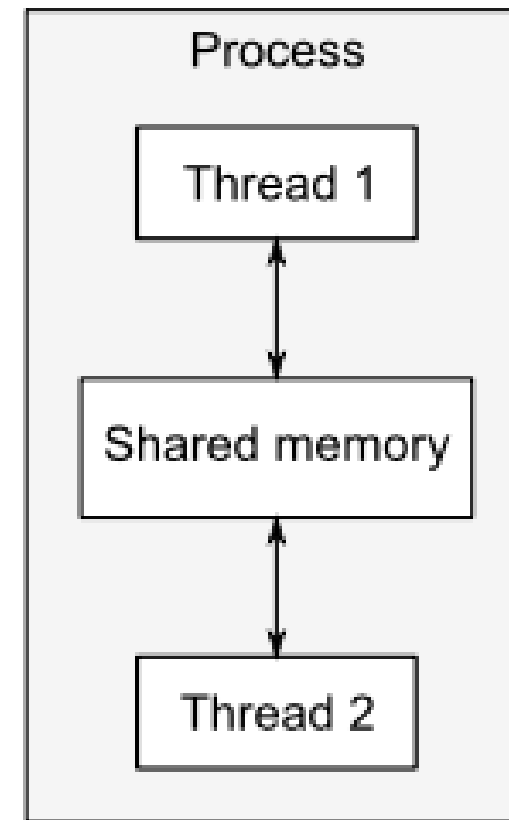


Types of concurrency

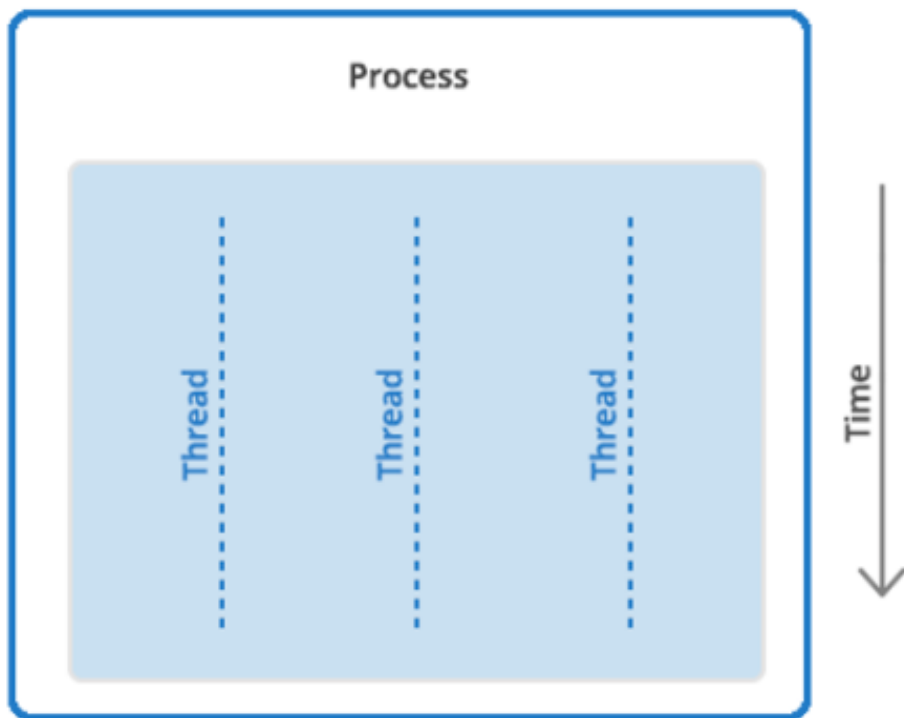
Concurrent Processes



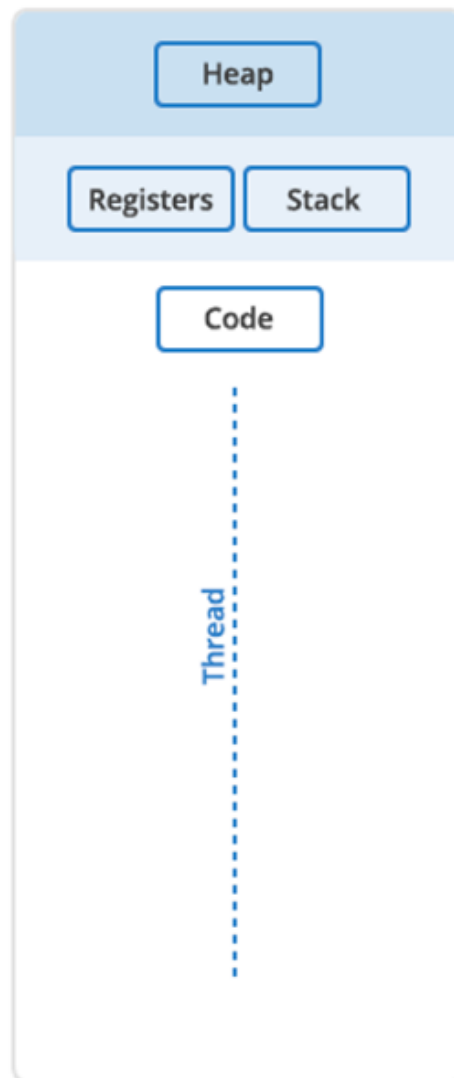
Concurrent Threads



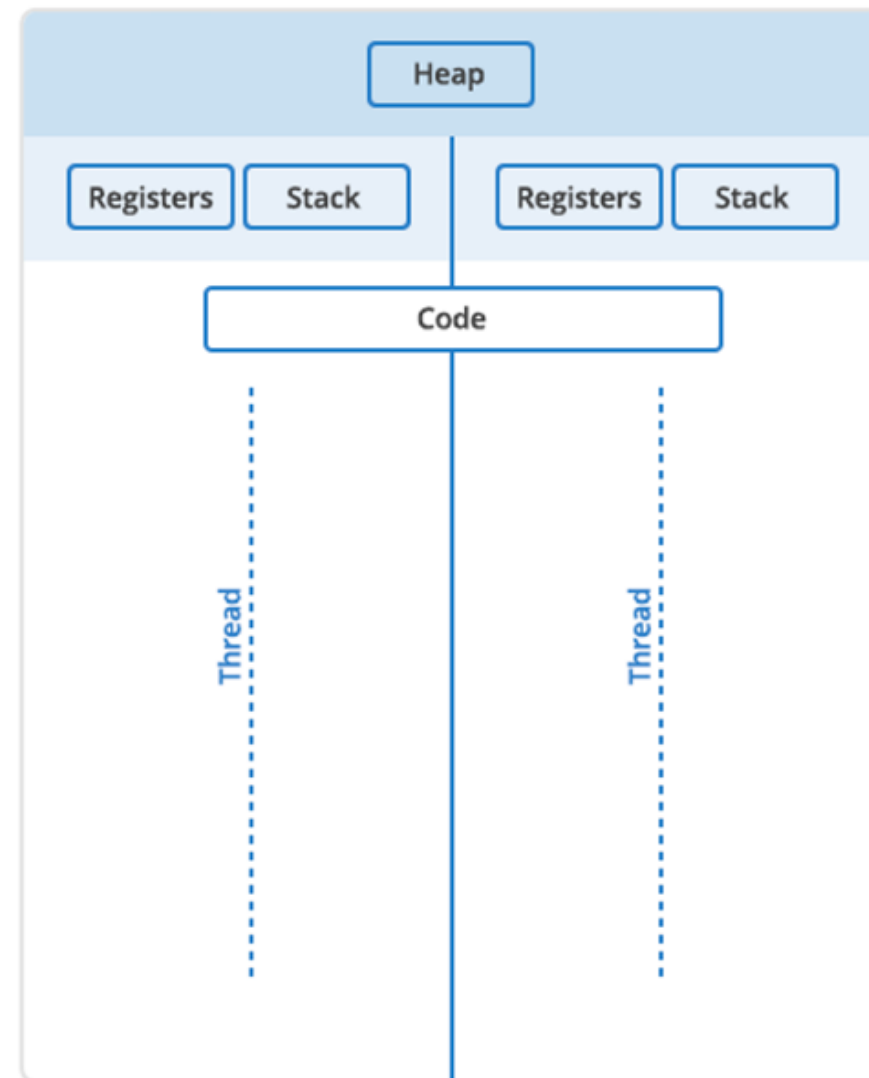
Concurrency



Single Thread



Multi Threaded



Multithreading

- Threads:
 - Threads are lightweight **executions**: each thread runs independently of the others and may run a different sequence of instructions.
 - All threads in a process **share the same address space**, and most of the data can be accessed directly from all threads—global variables remain global, and pointers or references to objects or data can be passed around among threads.

Multithreading

- What is concurrency
- Threads launching
 - `std::thread`
 - (Thread pool)
 - (openmp)
- Thread finishing
- Threads safety

Launching thread (via std::thread)

- Create a new thread object.
- Pass the executing code to be called (i.e, a callable object) into the constructor of the thread object.
- Once the object is created a new thread is launched, it will execute the code specified in callable

```
#include <thread> // part of the C++ Standard Library
```

Launching thread (via `std::thread`)

- A callable types:
 - **A function pointer**
 - **Free function (non-member function)**
 - **Member function**
 - A function object (functor)
 - A lambda expression

Launching thread

--- function pointer

- Launching a thread using **function pointers and function parameters**

```
void func(params)
{
    // Do something
}
```

```
std::thread thread_obj(func, args);
```

Launching thread

--- function pointer

Example 1: function takes one argument

```
#include <thread>
void hello(std::string to)
{
    std::cout << "Hello Concurrent World to " << to << "\n";
}
int main()
{
    std::thread t1( &hello, "alicia");
    std::thread t2( hello, "jonathan");
    t1.join();
    t2.join();
}
```

&(address-of) is **optional**
the function name decays to
function pointer **automatically**,
due to function-to-function-
pointer decay

Launching thread

--- function pointer

Example2: function takes multiple arguments (passing by values, references)

- **std::ref** for reference arguments

```
#include <thread>
void hello_count(std::string to, int &x){
    x++;
    std::cout << "Hello to " << to << x << std::endl;
}
int main(){
    int x = 0;
    std::thread threadObj(hello_count, "alicia", std::ref(x));
    ... // join
}
```

Launching thread (via `std::thread`)

- A callable types:
 - **A function pointer**
 - Free function (non-member function)
 - **Member function**
 - A function object (functor)
 - A lambda expression

How does calling a function on a class object work in C++?

- Suppose I have a class with an attribute `x`, a function `print()` that prints `x`.
- All objects of the class have their own copy of the non-static data members, but they share the class functions.
- When I call `print()` on different objects, why are their behavior different?

```
class myClass{  
public:  
    int x;  
    void print(){  
        std::cout << x << std::endl;  
    }  
};
```

```
int main(){  
    myClass obj;  
    obj.print();  
}
```

Solution to the puzzle:

- All class functions automatically receive a pointer to the class object as their first argument
- For example, `myClass::print()` behaves as if it's written as `myClass::print(myClass* obj_ptr)`
- All references to `x` in the function resolve as `obj_ptr->x`

```
class myClass{  
public:  
    int x;  
    void print(){  
        std::cout << x << std::endl;  
    }  
};
```

```
int main(){  
    myClass obj;  
    obj.print();  
}
```

Launching thread

--- member function pointer

- Launching a thread using (non-static) member function

```
class FunClass {  
    void func(params) {  
        // Do Something  
    }  
};  
FunClass x;  
std::thread thread_object(&FunClass::func, &x, params);
```

Launching thread

--- member function pointer

- Example3: launching thread with (non-static) member function

```
class Hello
{
public:
    void greeting(std::string const &message) const{
        std::cout << message << std::endl;
    }
};

int main(){
    Hello x;
    std::string msg("hello");
    std::thread t(&Hello::greeting, &x, msg);
    ... //join}
```

Multithreading

--- managing thread

- A callable types:
 - A function pointer
 - **A function object (functor)**
 - A lambda expression

Multithreading

--- Launching thread with function object

- Launching a thread using **function object and taking function parameters**

```
class fn_object_class {  
    // Overload () operator  
    void operator()(params) {  
        // Do Something  
    }  
}  
  
std::thread thread_object(fn_object_class(), params)
```

- Example: launching thread with function object

- Create a callable object using the constructor
- The thread calls the function call operator on the object

```
#include <thread>  
#include <iostream>  
  
class Hello{  
public:  
    void operator() (std::string name)  
    {  
        std::cout << "Hello to " << name << std::endl;  
    }  
};  
  
int main(){  
    std::thread t(Hello(), "alicia");  
    t.join();  
}
```

Multithreading

--- managing thread

- A callable types:
 - A function pointer
 - A function object
 - **A lambda expression**

Multithreading

--- Launching thread with lambda function

- Launching a thread using **lambda function**

```
std::thread thread_object([](params) {  
    // Do Something  
}, params);
```

- Example:

```
#include <iostream>  
#include <string>  
#include <thread>  
  
int main()  
{  
    std::thread t([](std::string name){  
        std::cout << "Hello World ! " << name << " \n";  
    }, "Alicia");  
    t.join();  
}
```


Lambda function

- Lambda expression

```
[ capture clause ] (parameters) -> return-type  
{  
    definition of method  
}
```


Lambda function

```
[ capture clause ] (parameters) -> return-type  
{  
    definition of method  
}
```

- Capture variables:
 - [&] : capture all external variables by reference
 - [=] : capture all external variables by value
 - [a, &b] : capture a by value and b by reference

```
std::vector<int> v1 = {3, 1, 7, 9};  
std::vector<int> v2 = {10, 2, 7, 16, 9};  
// access v1 and v2 by reference  
auto pushinto = [&] (int m){  
    v1.push_back(m);  
    v2.push_back(m);  
};  
pushinto(100);
```

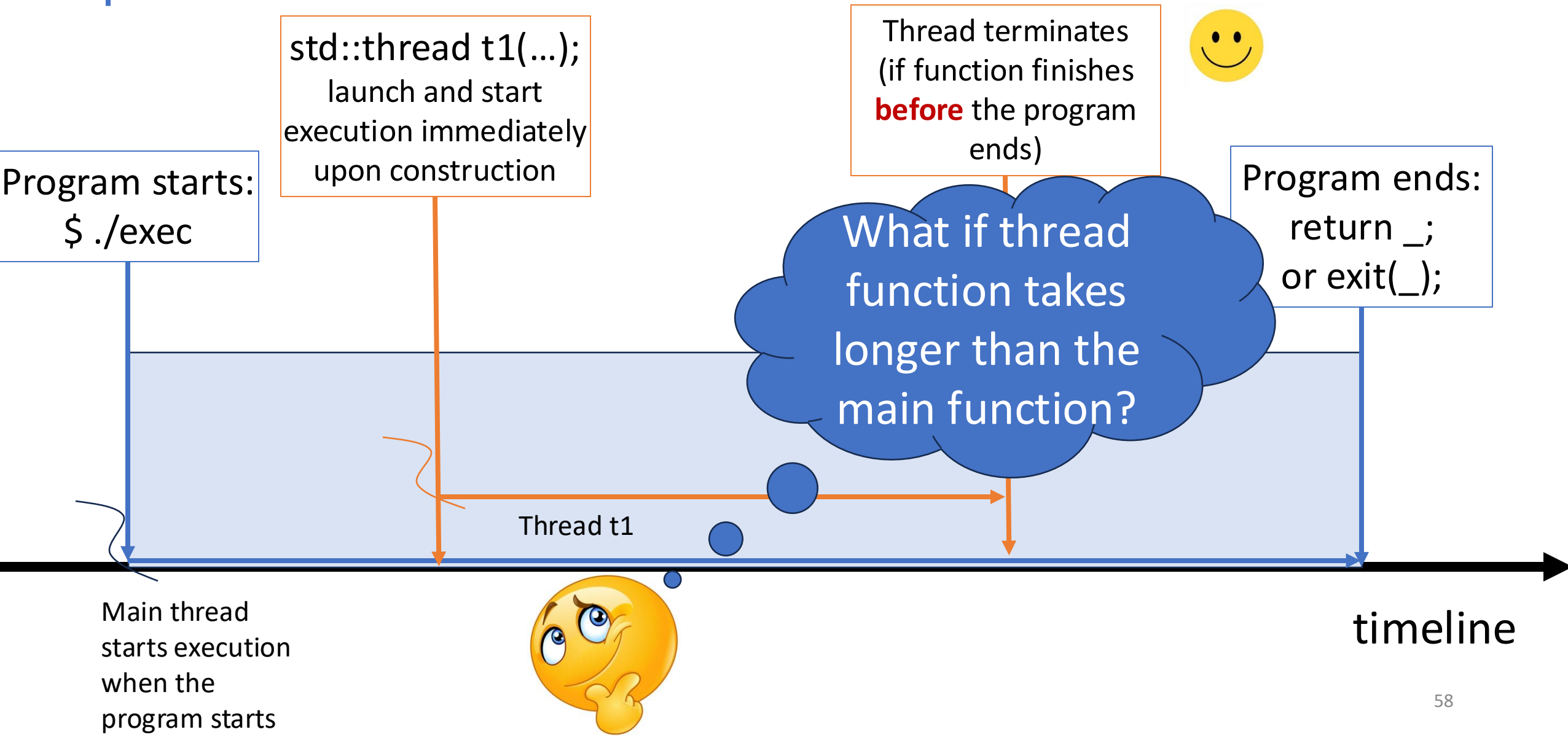
& can access all
the variables that
are in scope.



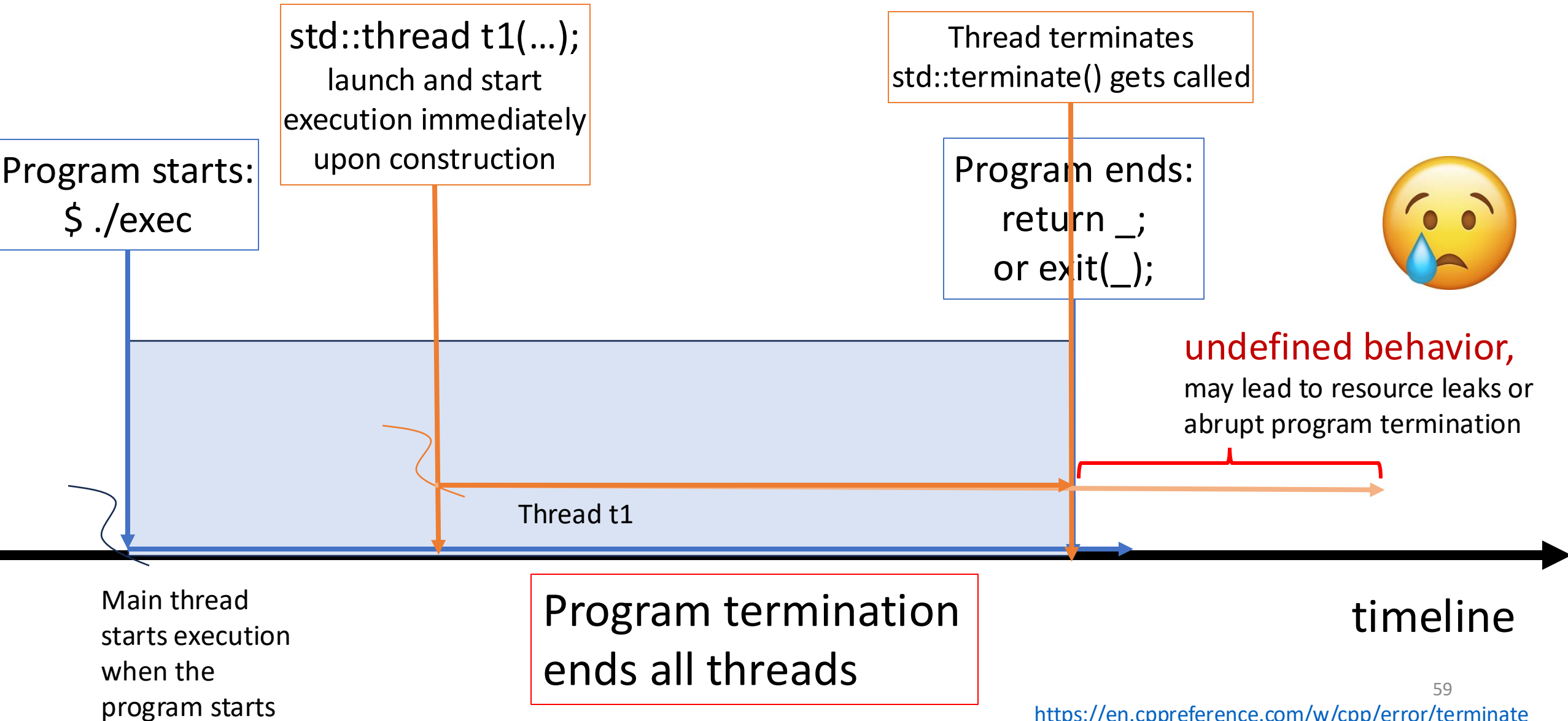
Multithreading

- What is concurrency
- Threads launching
- Thread finishing
 - `join()`
 - `detach()`
- Threads safety

Thread lifecycle and program termination



Thread lifecycle and program termination



Multithreading

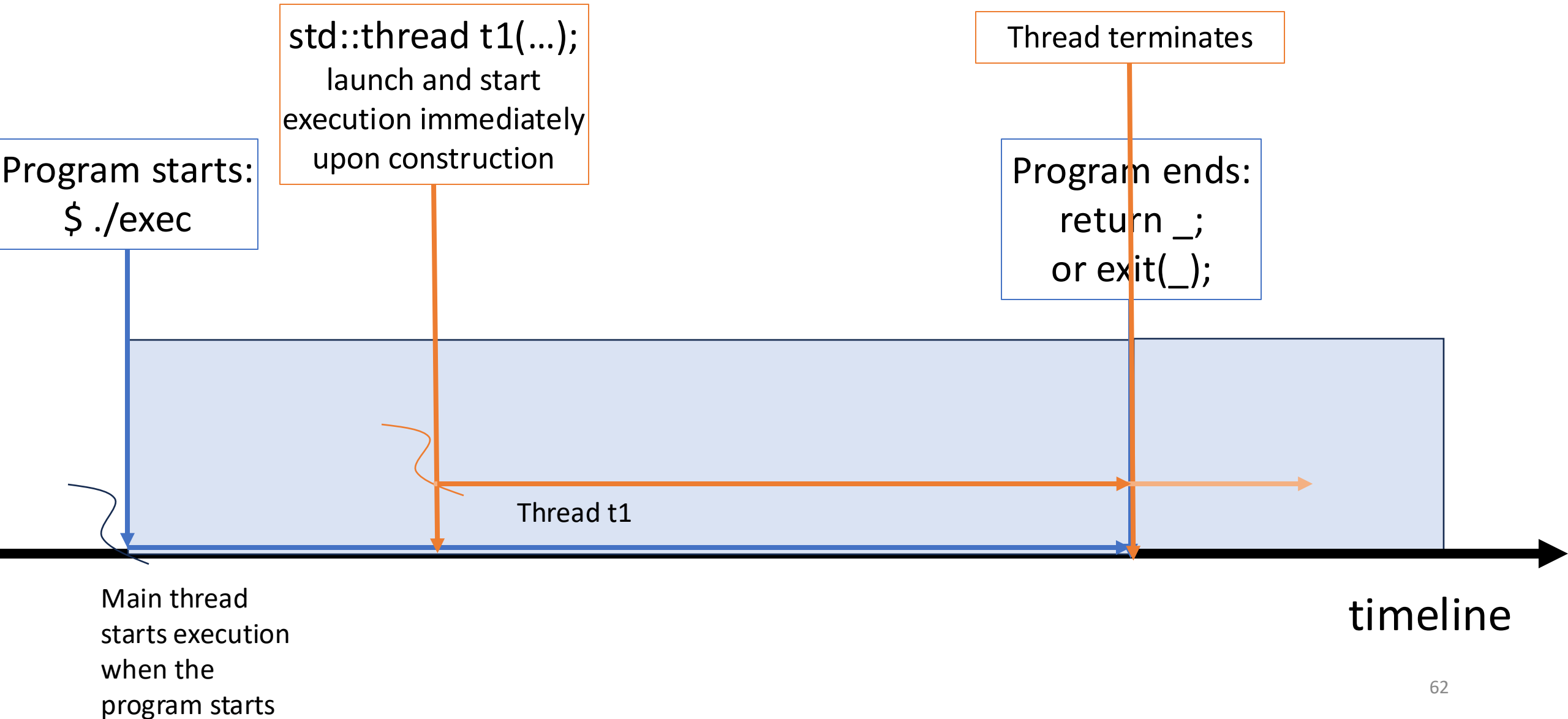
- Launching a thread:
 - Function pointer
 - Function object
 - Lambda function
- Managing threads
 - Join()

Joining threads with `std::thread`

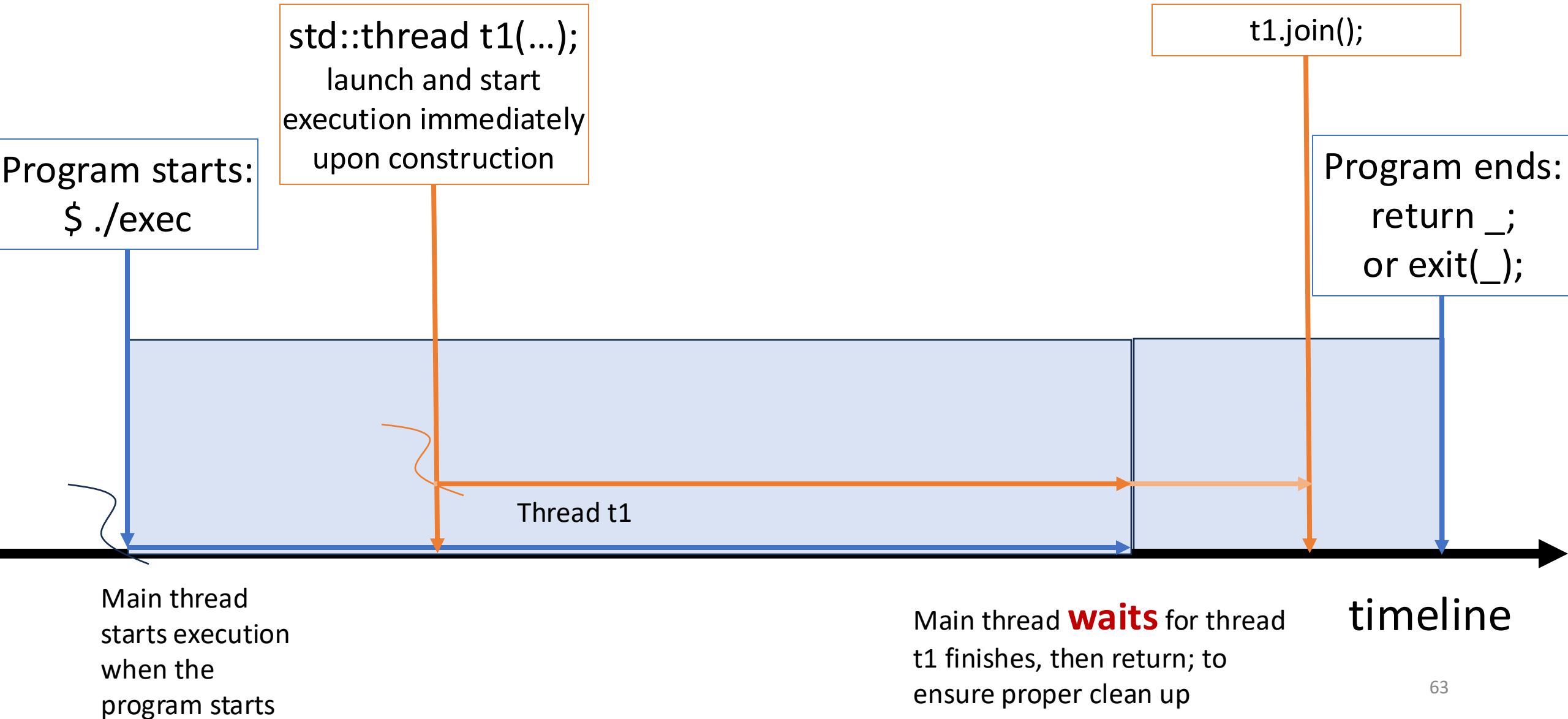
```
std::thread thread_obj(func, params);  
Thread_obj.join();
```

- **Wait** for a thread to complete
- Ensure that the thread was **finished before** the function was **exited**
- **Clean up** any storage associated with the thread
- `join()` can be called only **once for a given thread**

Thread lifecycle and program termination



Thread lifecycle and program termination



Where to find the resources?

- Copy constructor: <https://www.geeksforgeeks.org/copy-constructor-in-cpp/>
- Move semantics: <https://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html>
- Operator overload: <https://www.geeksforgeeks.org/operator-overloading-cpp/>
- Effective C++: 55 specific ways to improve your programs and designs, Scott Meyers, 3rd edition
- A Tour of C++, Bjarne Stroustrup
- Concurrency programming:
 - [Book: C++ Concurrency in Action Practice Multithreading](#)
 - <https://learn.microsoft.com/en-us/archive/blogs/ericlippert/what-is-this-thing-you-call-thread-safe>
 - cppcon thread-safe: https://youtu.be/s5PCh_FaMfM?si=-3h7nszcy_jesQAH
- Notes:
 - <https://thispointer.com/c11-multithreading-part-3-carefully-pass-arguments-to-threads/>