Concurrency

Lesson 1

Introduction to Running Threads

Lesson 2

Passing Data between Threads

Lesson 3

Mutexes and Locks

Lesson 4

Condition Variables and Message Queues

Project

Program a Concurrent Traffic Simulation

**LESSON 1**

**INTRODUCTION TO RUNNING THREADS**

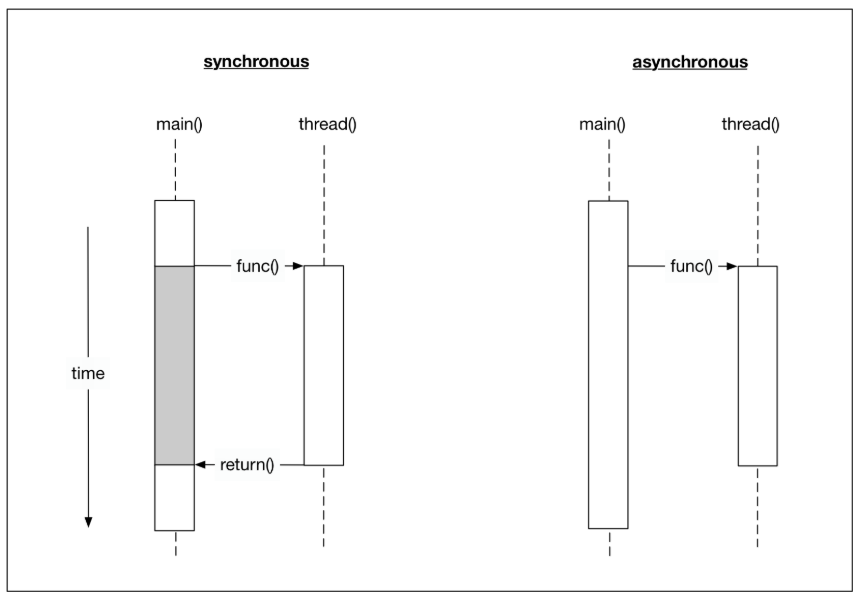
1. **Introduction**

Video: <https://youtu.be/frWbL4aWNI4>

1. **Processes and Threads**

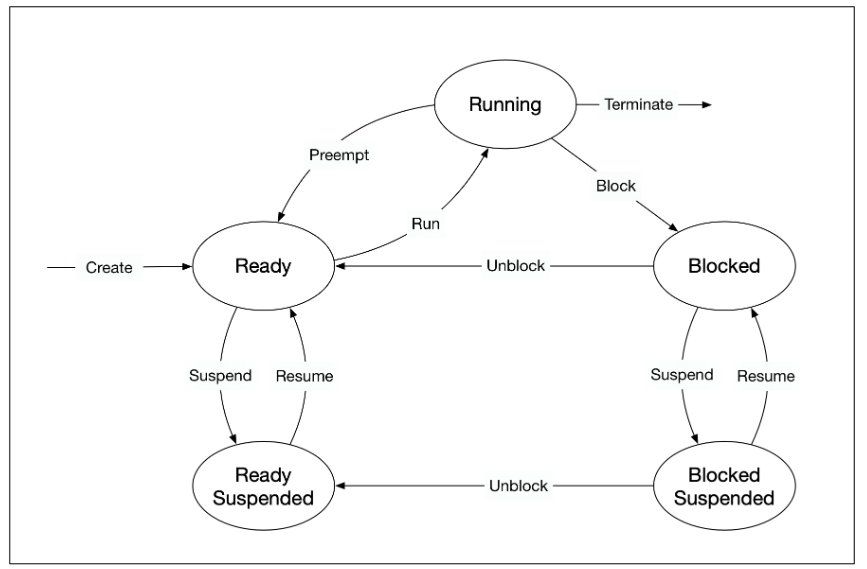
Video: <https://youtu.be/V4YyCCELd4M>

In this lesson, you will learn how to start and manage your first parallel path of execution, which runs concurrently with the main program and is thus asynchronous. In contrast to synchronous programs, the main program can continue with its line of execution without the need to wait for the parallel task to complete. The following figure illustrates this difference.

[](https://classroom.udacity.com/nanodegrees/nd213/parts/7f8f6354-1461-493e-8542-8dd742bd205b/modules/6f1d46b4-0891-4a77-ba27-f6ff46a6dd44/lessons/0ccc7913-cee9-43c7-824a-5148d5fce65a/concepts/8d9fcf44-5a91-4e2e-aa0e-74b7399fdb04)

Before we start writing a first asynchronous program in C++, let us take a look at the differences between two important concepts : processes and threads.

A *process* (also called a task) is a computer program at runtime. It is comprised of the runtime environment provided by the operating system (OS), as well as of the embedded binary code of the program during execution. A process is controlled by the OS through certain actions with which it sets the process into one of several carefully defined states:

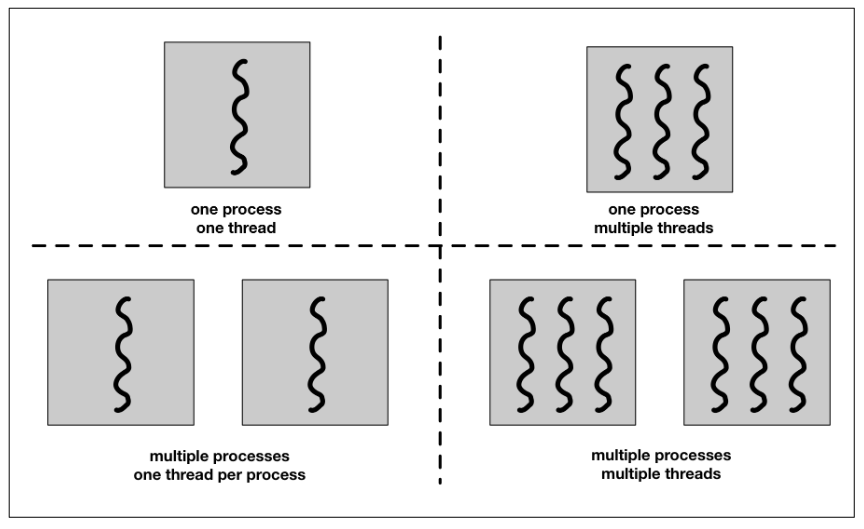
[](https://classroom.udacity.com/nanodegrees/nd213/parts/7f8f6354-1461-493e-8542-8dd742bd205b/modules/6f1d46b4-0891-4a77-ba27-f6ff46a6dd44/lessons/0ccc7913-cee9-43c7-824a-5148d5fce65a/concepts/8d9fcf44-5a91-4e2e-aa0e-74b7399fdb04)

* **Ready** : After its creation, a process enters the ready state and is loaded into main memory. The process now is ready to run and is waiting for CPU time to be executed. Processes that are ready for execution by the CPU are stored in a queue managed by the OS.
* **Running** : The operating system has selected the process for execution and the instructions within the process are executed on one or more of the available CPU cores.
* **Blocked** : A process that is blocked is one that is waiting for an event (such as a system resource becoming available) or the completion of an I/O operation.
* **Terminated** : When a process completes its execution or when it is being explicitly killed, it changes to the "terminated" state. The underlying program is no longer executing, but the process remains in the process table as a "zombie process". When it is finally removed from the process table, its lifetime ends.
* **Ready suspended** : A process that was initially in ready state but has been swapped out of main memory and placed onto external storage is said to be in suspend ready state. The process will transition back to ready state whenever it is moved to main memory again.
* **Blocked suspended** : A process that is blocked may also be swapped out of main memory. It may be swapped back in again under the same conditions as a "ready suspended" process. In such a case, the process will move to the blocked state, and may still be waiting for a resource to become available.

Processes are managed by the *scheduler* of the OS. The scheduler can either let a process run until it ends or blocks (non-interrupting scheduler), or it can ensure that the currently running process is interrupted after a short period of time. The scheduler can switch back and forth between different active processes (interrupting scheduler), alternately assigning them CPU time. The latter is the typical scheduling strategy of any modern operating system.

Since the administration of processes is computationally taxing, operating systems support a more resource-friendly way of realizing concurrent operations: the threads.

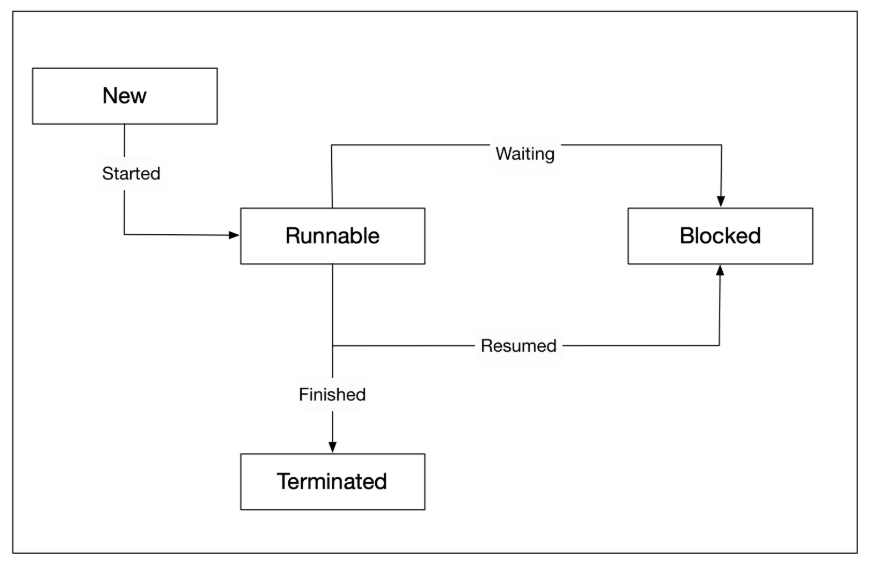
A *thread* represents a concurrent execution unit within a process. In contrast to full-blown processes as described above, threads are characterized as light-weight processes (LWP). These are significantly easier to create and destroy: In many systems the creation of a thread is up to 100 times faster than the creation of a process. This is especially advantageous in situations, when the need for concurrent operations changes dynamically.



Threads exist within processes and share their resources. As illustrated by the figure above, a process can contain several threads or - if no parallel processing is provided for in the program flow - only a single thread.

A major difference between a process and a thread is that each process has its own address space, while a thread does not require a new address space to be created. All the threads in a process can access its shared memory. Threads also share other OS dependent resources such as processors, files, and network connections. As a result, the management overhead for threads is typically less than for processes. Threads, however, are not protected against each other and must carefully synchronize when accessing the shared process resources to avoid conflicts.

Similar to processes, threads exist in different states, which are illustrated in the figure below:

[](https://classroom.udacity.com/nanodegrees/nd213/parts/7f8f6354-1461-493e-8542-8dd742bd205b/modules/6f1d46b4-0891-4a77-ba27-f6ff46a6dd44/lessons/0ccc7913-cee9-43c7-824a-5148d5fce65a/concepts/8d9fcf44-5a91-4e2e-aa0e-74b7399fdb04)

* **New** : A thread is in this state once it has been created. Until it is actually running, it will not take any CPU resources.
* **Runnable** : In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to assign CPU time to the thread.
* **Blocked** : A thread might be in this state, when it is waiting for I/O operations to complete. When blocked, a thread cannot continue its execution any further until it is moved to the runnable state again. It will not consume any CPU time in this state. The thread scheduler is responsible for reactivating the thread.

The System Task Manager

Video: <https://youtu.be/QdXCdZdGBwM>

Outro

Video: <https://youtu.be/9XuSTdkxgOE>

1. **Running a Single Thread**

Video: <https://youtu.be/tQZA7gukZTc>

**Concurrency Support in C++11**

The concurrency support in C++ makes it possible for a program to execute multiple threads in parallel. Concurrency was first introduced into the standard with C++11. Since then, new concurrency features have been added with each new standard update, such as in C++14 and C++17. Before C++11, concurrent behavior had to be implemented using native concurrency support from the OS, using POSIX Threads, or third-party libraries such as BOOST. The standardization of concurrency in C++ now makes it possible to develop cross-platform concurrent programs, which is as significant improvement that saves time and reduces error proneness. Concurrency in C++ is provided by the thread support library, which can be accessed by including the header.

A running program consists of at least one thread. When the main function is executed, we refer to it as the "main thread". Threads are uniquely identified by their thread ID, which can be particularly useful for debugging a program. The code on the right prints the thread identifier of the main thread and outputs it to the console:

#include <iostream>

#include <thread>

int main()

{

std::cout << "Hello concurrent world from main! Thread id = " << std::this\_thread::get\_id() << std::endl;

return 0;

}

These are the results when run:

Hello concurrent world from main! Thread id = 1

You can compile this code from the terminal in the lower right using g++ as follows:

g++ example\_1.cpp

and run it with

./a.out

Note: The actual thread id and process exit message will vary from machine to machine.

Also, it is possible to retrieve the number of available CPU cores of a system. The example on the right prints the number of CPU cores to the console.

#include <iostream>

#include <thread>

int main()

{

unsigned int nCores = std::thread::hardware\_concurrency();

std::cout << "This machine supports concurrency with " << nCores << " cores available" << std::endl;

return 0;

}

These are the results from a local machine at the time of writing:

This machine supports concurrency with 2 cores available

Try running this code to see what results you get!

This machine supports concurrency with 8

**Starting a second thread**

In this section, we will start a second thread in addition to the main thread of our program. To do this, we need to construct a thread object and pass it the function we want to be executed by the thread. Once the thread enters the runnable state, the execution of the associated thread function may start at any point in time.

// create thread

std::thread t(threadFunction);

After the thread object has been constructed, the main thread will continue and execute the remaining instructions until it reaches the end and returns. It is possible that by this point in time, the thread will also have finished. But if this is not the case, the main program will terminate and the resources of the associated process will be freed by the OS. As the thread exists within the process, it can no longer access those resources and thus not finish its execution as intended.

To prevent this from happening and have the main program wait for the thread to finish the execution of the thread function, we need to call join() on the thread object. This call will only return when the thread reaches the end of the thread function and block the main thread until then.

The code on the right shows how to use join() to ensure that main() waits for the thread t to finish its operations before returning. It uses the function sleep\_for(), which pauses the execution of the respective threads for a specified amount of time. The idea is to simulate some work to be done in the respective threads of execution.

#include <iostream>

#include <thread>

void threadFunction()

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(100)); // simulate work

std::cout << "Finished work in thread\n";

}

int main()

{

// create thread

std::thread t(threadFunction);

// do something in main()

std::this\_thread::sleep\_for(std::chrono::milliseconds(50)); // simulate work

std::cout << "Finished work in main\n";

// wait for thread to finish

t.join();

return 0;

}

To compile this code with g++, you will need to use the -pthread flag. pthread adds support for multithreading with the pthreads library, and the option sets flags for both the preprocessor and linker:

g++ example\_3.cpp -pthread

**Note:** If you compile without the -pthread flag, you will see an error of the form: undefined reference to pthread\_create. You will need to use the -pthread flag for all other multithreaded examples in this course going forward.

The code produces the following output:

Finished work in main

Finished work in thread

Not surprisingly, the main function finishes before the thread because the delay inserted into the thread function is much larger than in the main path of execution. The call to join() at the end of the main function ensures that it will not prematurely return. As an experiment, comment out t.join() and execute the program. What do you expect will happen?

**Randomness of events**

One very important trait of concurrent programs is their non-deterministic behavior. It can not be predicted which thread the scheduler will execute at which point in time. In the code on the right, the amount of work to be performed both in the thread function and in main has been split into two separate jobs.

The console output shows that the work packages in both threads have been interleaved with the first package being performed before the second package.



Interestingly, when executed on my local machine, the order of execution has changed. Now, instead of finishing the second work package in the thread first, main gets there first.



Executing the code several times more shows that the two versions of program output interchange in a seemingly random manner. This element of randomness is an important characteristic of concurrent programs and we have to take measures to deal with it in a controlled way that prevent unwanted behavior or even program crashes.

**Reminder:** You will need to use the -pthread flag when compiling this code, just as you did with the previous example. This flag will be needed for all future multithreaded programs in this course as well.

**Using join() as a barrier**

In the previous example, the order of execution is determined by the scheduler. If we wanted to ensure that the thread function completed its work before the main function started its own work (because it might be waiting for a result to be available), we could achieve this by repositioning the call to join.

In the file on the right, the .join() has been moved to before the work in main(). The order of execution now always looks like the following:



In later sections of this course, we will make extended use of the join() function to carefully control the flow of execution in our programs and to ensure that results of thread functions are available and complete where we need them to be.

**Detach**

Let us now take a look at what happens if we don’t join a thread before its destructor is called. When we comment out join in the example above and then run the program again, it aborts with an error. The reason why this is done is that the designers of the C++ standard wanted to make debugging a multi-threaded program easier: Having the program crash forces the programer to remember joining the threads that are created in a proper way. Such a hard error is usually much easier to detect than soft errors that do not show themselves so obviously.

There are some situations however, where it might make sense to not wait for a thread to finish its work. This can be achieved by "detaching" the thread, by which the internal state variable "joinable" is set to "false". This works by calling the detach() method on the thread. The destructor of a detached thread does nothing: It neither blocks nor does it terminate the thread. In the following example, detach is called on the thread object, which causes the main thread to immediately continue until it reaches the end of the program code and returns. Note that a detached thread can not be joined ever again.

#include <iostream>

#include <thread>

void threadFunction()

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(50)); // simulate work

std::cout << "Finished work in thread\n";

}

int main()

{

// create thread

std::thread t(threadFunction);

// detach thread and continue with main

t.detach();

// do something in main()

std::this\_thread::sleep\_for(std::chrono::milliseconds(50)); // simulate work

std::cout << "Finished work in main\n";

return 0;

}

You can run the code above using example\_6.cpp over on the right side of the screen.

Programmers should be very careful though when using the detach()-method. You have to make sure that the thread does not access any data that might get out of scope or be deleted. Also, we do not want our program to terminate with threads still running. Should this happen, such threads will be terminated very harshly without giving them the chance to properly clean up their resources - what would usually happen in the destructor. So a well-designed program usually has a well-designed mechanism for joining all threads before exiting.

**Quiz: Starting your own threads**

In the code on the right, you will find a thread function called threadFunctionEven, which is passed to a thread t. In this example, the thread is immediately detached after creation. To ensure main does not quit before the thread is finished with its work, there is a sleep\_for call at the end of main.

Please create a new function called threadFunctionOdd that outputs the string "Odd threadn". Then write a for-loop that starts 6 threads and immediately detaches them. Based on wether the increment variable is even or odd, you should pass the respective function to the thread.

Sol:

1. **Starting a Thread function with Object**

Video: <https://youtu.be/ytEszL3n_Fw>

**Functions and Callable Objects**

In the previous section, we have created our first thread by passing it a function to execute. We did not discuss this concept in depth at the time, but in this section we will focus on the details of passing functions to other functions, which is one form of a callable object.

In C++, callable objects are object that can appear as the left-hand operand of the call operator. These can be pointers to functions, objects of a class that defines an overloaded function call operator and lambdas (an anonymous inline function), with which function objects can be created in a very simple way. In the context of concurrency, we can use callable objects to attach a function to a thread.

In the last section, we constructed a thread object by passing a function to it without any arguments. If we were limited to this approach, the only way to make data available from within the thread function would be to use global variables - which is definitely not recommendable and also incredibly messy.

In this section, we will therefore look at several ways of passing data to a thread function.

**Starting Threads with Function Objects**

The std::thread constructor can also be called with instances of classes that implement the function-call operator. In the following, we will thus define a class that has an overloaded ()-operator. In preparation for the final project of this course, which will be a traffic simulation with vehicles moving through intersections in a street grid, we will define a (very) early version of the Vehicle class in this example:

#include <iostream>

#include <thread>

class Vehicle

{

public:

void operator()()

{

std::cout << "Vehicle object has been created \n" << std::endl;

}

};

int main()

{

// create thread

std::thread t(Vehicle()); // C++'s most vexing parse

// do something in main()

std::cout << "Finished work in main \n";

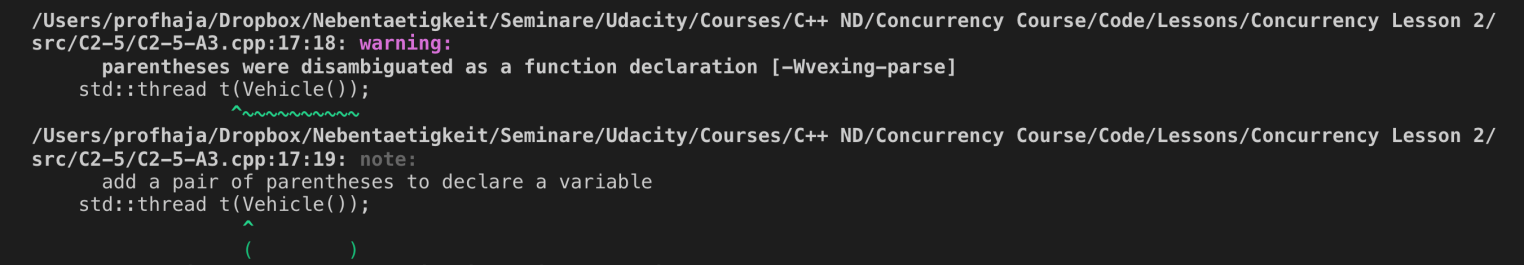
// wait for thread to finish

t.join();

return 0;

}

When executing this code, the clang++ compiler generates a warning, which is followed by an error:



A similar error is shown when compiling with g++:

error: request **for** member ‘join’ **in** ‘t’, which is of non-class type ‘std::thread(Vehicle (\*)())’

t.join();

So you will see an error when you compile example\_1.cpp!

The extra parentheses suggested by the compiler avoid what is known as C++'s "most vexing parse", which is a specific form of syntactic ambiguity resolution in the C++ programming language.

The expression was coined by Scott Meyers in 2001, who talks about it in details in his book "Effective STL". The "most vexing parse" comes from a rule in C++ that says that anything that could be considered as a function declaration, the compiler should parse it as a function declaration - even if it could be interpreted as something else.

In the previous code example, the line

// create thread

std::thread t0(Vehicle());

is seemingly ambiguous, since it could be interpreted either as

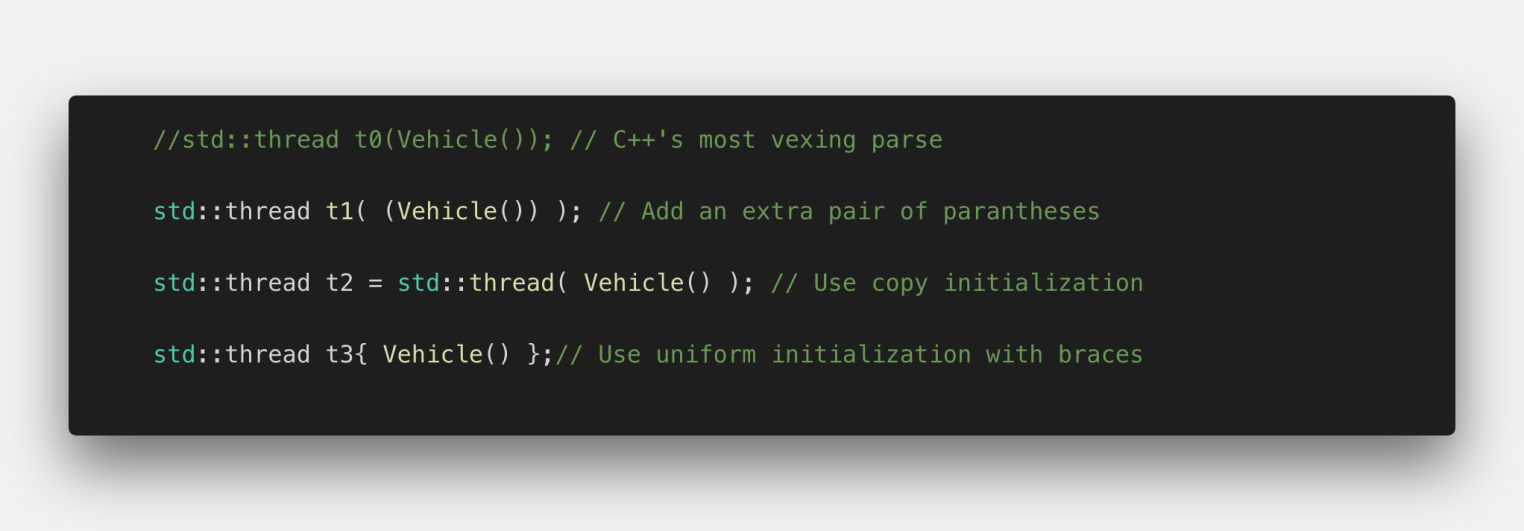
1. a variable definition for variable t of class std::thread, initialized with an anonymous instance of class Vehicle or
2. a function declaration for a function t that returns an object of type std::thread and has a single (unnamed) parameter that is a pointer to function returning an object of type Vehicle

Most programmers would presumable expect the first case to be true, but the C++ standard requires it to be interpreted as the second - hence the compiler warning.

There are three ways of forcing the compiler to consider the line as the first case, which would create the thread object we want:

* Add an extra pair of parentheses
* Use copy initialization
* Use uniform initialization with braces

The following code shows all three variants:



The output of this code sample shows that all three threads are executed and the Vehicle object is properly initialized:

*Vehicle object has been created*

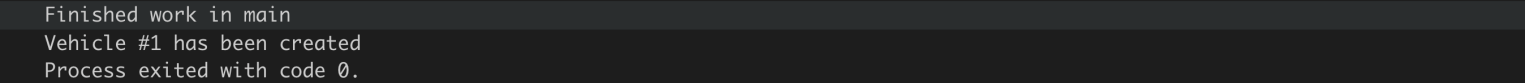
*Vehicle object has been created*

*Vehicle object has been created*

Whichever option we use, the idea is the same: the function object is copied into internal storage accessible to the new thread, and the new thread invokes the operator (). The Vehicle class can of course have data members and other member functions too, and this is one way of passing data to the thread function: pass it in as a constructor argument and store it as a data member:



In the above code example, the class Vehicle has a constructor that takes an integer and it will store it internally in a variable \_id. In the overloaded function call operator, the vehicle id is printed to the console. In main(), we are creating the Vehicle object using copy initialization. The output of the program is given below:



As can easily be seen, the integer ID has been successfully passed into the thread function.

## Lambdas

Another very useful way of starting a thread and passing information to it is by using a lambda expression ("Lambda" for short). With a Lambda you can easily create simple function objects.

The name "Lambda" comes from Lambda Calculus , a mathematical formalism invented by Alonzo Church in the 1930s to investigate questions of logic and computability. Lambda calculus formed the basis of LISP, a functional programming language. Compared to Lambda Calculus and LISP, C ++ - Lambdas have the properties of being unnamed and capturing variables from the surrounding context, but lack the ability to execute and return functions.

A Lambda is often used as an argument for functions that can take a callable object. This can be easier than creating a named function that is used only when passed as an argument. In such cases, Lambdas are generally preferred because they allow the function objects to be defined inline. If Lambdas were not available, we would have to define an extra function somewhere else in our source file - which would work but at the expense of the clarity of the source code.

A Lambda is a function object (a "functor"), so it has a type and can be stored and passed around. Its result object is called a "closure", which can be called using the operator () as we will see shortly.

A lambda formally consists of three parts: a capture list [] , a parameter list () and a main part {}, which contains the code to be executed when the Lambda is called. Note that in principal all parts could be empty.

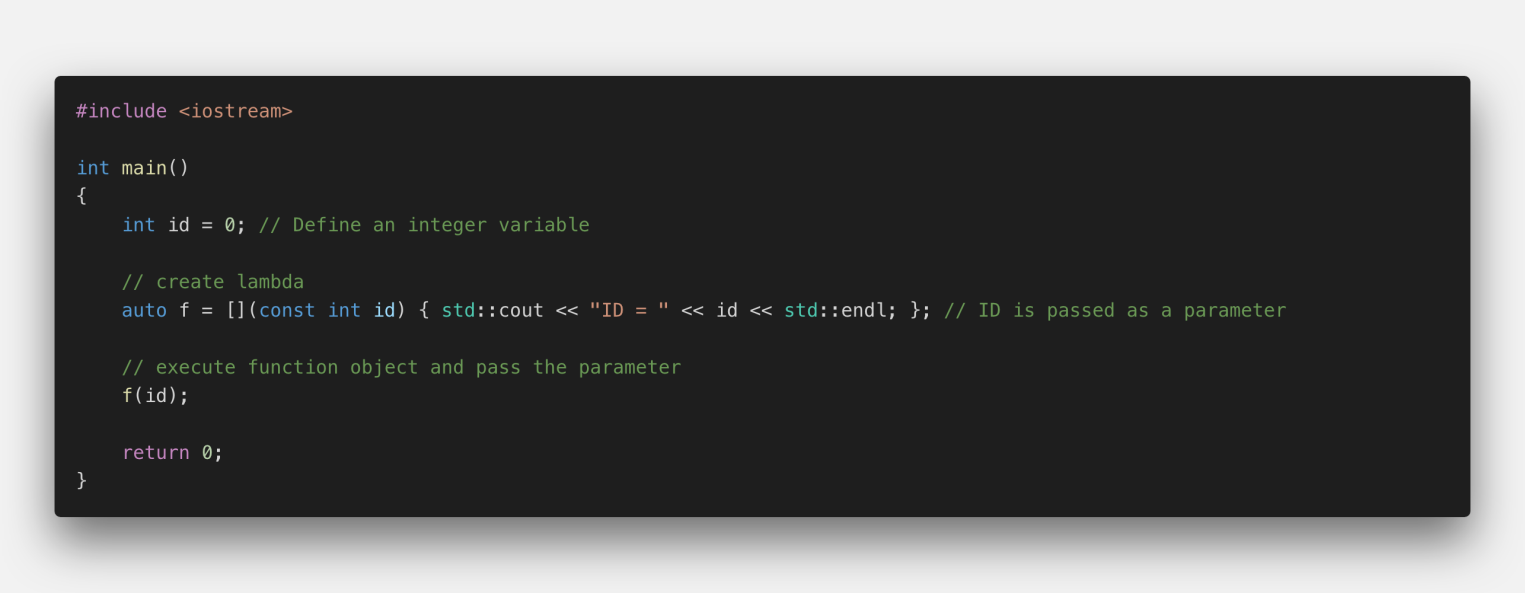
The capture list *[]*: By default, variables outside of the enclosing {} around the main part of the Lambda can not be accessed. By adding a variable to the capture list however, it becomes available within the Lambda either as a copy or as a reference. The captured variables become a part of the Lambda.

By default, variables in the capture block can not be modified within the Lambda. Using the keyword "mutable" allows to modify the parameters captured by copy, and to call their non-const member functions within the body of the Lambda. The following code examples show several ways of making the external variable "id" accessible within a Lambda.



Even though we have been using Lambdas in the above example in various ways, it is important to note that a Lambda does not exist at runtime. The runtime effect of a Lambda is the generation of an object, which is known as closure. The difference between a Lambda and the corresponding closure is similar to the distinction between a class and an instance of the class. A class exists only in the source code while the objects created from it exist at runtime.

The following example illustrates how the function object is first created and then used to pass the parameter id later in the code.



## Quiz

The code below shows how to capture variables by value and by reference, how to pass variables to a Lambda using the parameter list and how to use the closure to execute the Lambda.



Please think about the resulting output for a while. What would you say is the order in which the various strings are printed to the console? Also, what will be the value for ID for each output?

Sol:

The following image shows the order of the printed strings, along with the ID for each output. For a complete explanation of the code and this output, see the solution video after this workspace.



## Starting Threads with Lambdas[¶](https://r845225c859887xjupyterl663lbcad.udacity-student-workspaces.com/lab?#Starting-Threads-with-Lambdas)

A Lambda is, as we’ve seen, just an object and, like other objects it may be copied, passed as a parameter, stored in a container, etc. The Lambda object has its own scope and lifetime which may, in some circumstances, be different to those objects it has ‘captured’. Programers need to take special care when capturing local objects by reference because a Lambda’s lifetime may exceed the lifetime of its capture list: It must be ensured that the object to which the reference points is still in scope when the Lambda is called. This is especially important in multi-threading programs.

So let us start a thread and pass it a Lambda object to execute:



The output of the program looks like this



As you can see, the output in the main thread is generated first, at which point the variable ID has taken the value 1. Then, the call-by-value thread is executed with ID at a value of 0. Then, the call-by-reference thread is executed with ID at a value of 1. This illustrates the effect of passing a value by reference : when the data to which the reference refers changes before the thread is executed, those changes will be visible to the thread. We will see other examples of such behavior later in the course, as this is a primary source of concurrency bugs.

## Exercise Solution

Video: <https://youtu.be/K8GxT0bHosY>

1. **Starting a Thread  Variadic Templates & Member Functions**
2. **Running a Multible thred**