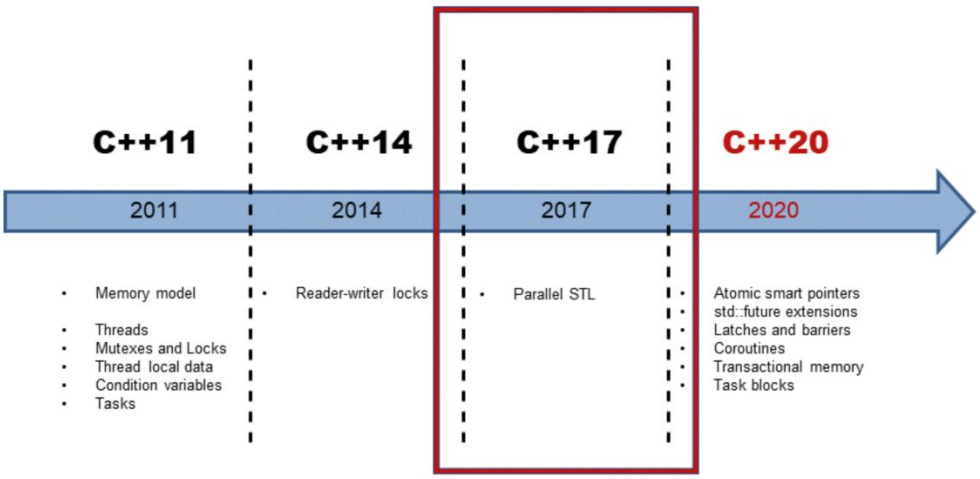
C++ Threads

What is *concurrency*?

Concurrency occurs when multiple copies of a program run simultaneously while communicating with each other. Simply put, concurrency is when two tasks are overlapped. A simple concurrent application will use a single machine to store the program's instruction, but that process is executed by multiple, different threads. This setup creates a kind of control flow, where each thread executes its instruction before passing to the next one. The threads act independently and to make decisions based on the previous thread as well. However, some issues can arise in concurrency that make it tricky to implement.

History of C++ concurrency.

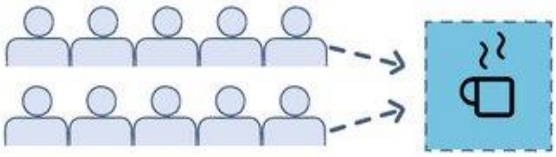
C++ 11 was the first *C++* standard to introduce concurrency, including threads, the C++ memory model, conditional variables, mutex, and more. The *C++11* standard changes drastically with *C++17*. The addition of parallel algorithms in the Standard Template Library (STL) greatly improved concurrent code.



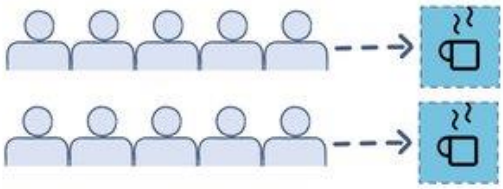
Concurrency vs. parallelism

Concurrency and *parallelism* often get mixed up, but it's important to understand the difference. In parallelism, we run multiple copies of the same program simultaneously, but they are executed on different data.

Concurrent: *Two Queues* & a *Single* Espresso machine.

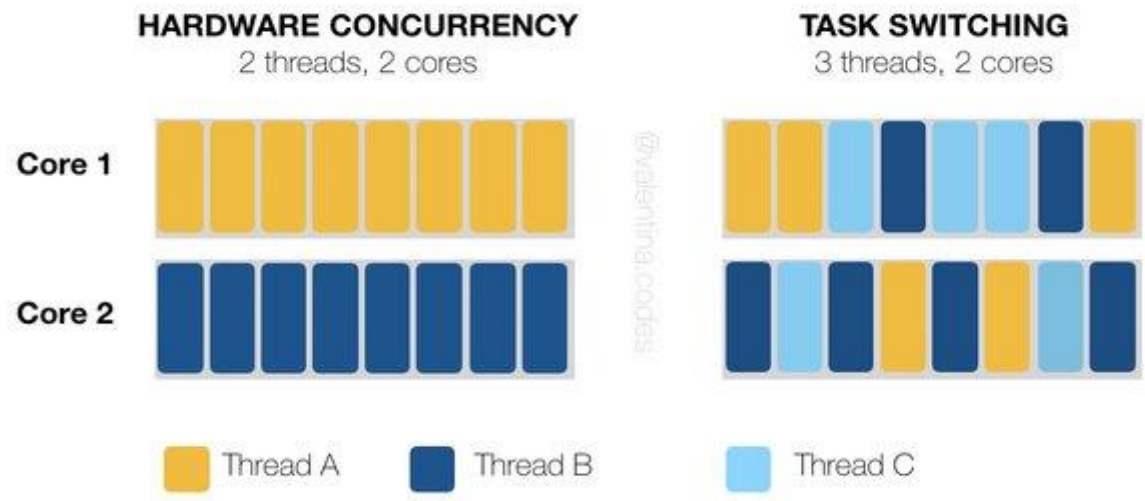


Parallel: *Two Queues* & *Two* Espresso machines.



C++ Multithreading

C++ multithreading involves creating and using thread objects, seen as *std::thread* in code, to carry out delegated sub-tasks independently. Upon creation, threads are passed a function to complete, and optionally some parameters for that function.



While each individual thread can complete only one function at a time, thread pools allow us to recycle and reuse thread objects to give programs the illusion of unlimited multitasking.

Initializing thread with a function

```
#include <iostream>
#include <thread>
using namespace std;
void threadFunc()
{
        cout << "Welcome to Multithreading" << endl;
}
int main()
{
        //pass a function to thread
        thread funcTest1(threadFunc);
}
```

Try to compile and run this program. It compiles without any errors but you will get a runtime error:

```
//pass a function to thread
thread funcTest1(threadFunc);
//main is blocked until funcTest1 is not finished
funcTest1.join();
```

Joinable and not Joinable threads

After *join()* returns, thread becomes not joinable. A joinable thread is a thread that represents a thread of execution which has not yet been joined. A thread is not joinable when it is default constructed or is moved/assigned to another thread or *join()* or *detach()* member function is called. Not joinable thread can be destroyed safely. You can check if a thread is joinable by using *joinable()* member function:

```
bool joinable()
```

This function returns true if the thread is joinable and false otherwise. It's better to check if the thread is joinable before *join()* function is called:

```
//pass a function to thread
thread funcTest1(threadFunc);
//check if thread is joinable
if (funcTest1.joinable())
{
        //main is blocked until funcTest1 is not finished
        funcTest1.join();
}
```

Detaching thread

As we mentioned above, thread becomes not joinable after *detach()* member function is called: *void detach()*. This function detaches a thread from the parent thread. It allows parent and child threads to be executed independently from each other. After the call of

detach() function, the threads are not synchronized in any way:

```
//detach funcTest1 from main thread
funcTest1.detach();
if (funcTest1.joinable())
{
	//main is blocked until funcTest1 is not finished
	funcTest1.join();
}
else
{
	cout << "functTest1 is detached" << endl;
}
```

You will notice that main thread is not waiting for the termination of its child thread.

Passing arguments to thread

In the previous examples, we used only functions and objects without passing any arguments to these functions and objects. We can use a function with parameters for thread initialization. Create new function for testing this possibility:

```
void printSomeValues(int val, char* str, double dval)
{
	cout << val << " " << str <<" " << dval << endl;
}
```

As you can see, this function takes three arguments. If you want to initialize a thread with this function, first you have to pass a pointer to this function, then pass the arguments to the function in the same order as they are in the parameter list of the function:

```
char* str = "Hello";
//5, str and 3.2 are passed to printSomeValues function
thread paramPass(printSomeValues, 5, str, 3.2);
if (paramPass.joinable())
paramPass.join();
```

When you want to initialize a thread with an object with parameters, we have to add corresponding parameter list to the overloading version of operator ():

```
class myFunctorParam
{
public:
	void operator()(int* arr, int length)
	{
		cout << "An array of length " << length << "is passed to thread" << endl;
		for (int i = 0; i != length; ++i)
			cout << arr[i] << " " << endl;
		cout << endl;
	}
};
```

As you can see, operator () takes two parameters:

void operator()(int arr, int length)*

The initialization of the thread with an object in this case is similar to using a function with parameters:

```
//these parameters will be passed to thread
int arr[5] = { 1, 3, 5, 7, 9 };
myFunctorParam objParamPass;
thread test(objParamPass, arr, 5);
if (test.joinable())
	test.join();
```

It is possible to use a member function of a class to pass parameters to thread. Add new public function to *myFunctorParam* class:

```
void changeSign(int* arr, int length)
{
	cout << "An arrray of length " << length << "is passed to thread" << endl;
	for (int i = 0; i != length; ++i)
		cout << arr[i] << " ";
	cout << "Changing sign of all elements of initial array" << endl;
	for (int i = 0; i != length; ++i)
	{
		arr[i] *= -1;
		cout << arr[i] << " ";
	}
}
```

Passing arguments to member function:

```
int arr2[5] = { -1, 3, 5, -7, 0 };
//initialize thread with member function
thread test2(&myFunctorParam::changeSign, &objParamPass, arr2, 5);
if (test2.joinable())
	test2.join();
```

When you pass arguments to the member function of a class, you have to specify arguments in the same order as they are listed in the parameter list of the function. It is done after the second parameter of the thread constructor:

thread test2(&myFunctorParam::changeSign, &objParamPass, arr2, 5);

Thread ID

Every thread has its unique identifier. Class thread has public member function that returns the *ID* of the thread:

id get_id()

The returned value is of type id that is specified in thread class. Look on the following example:

```
//create 3 different threads
thread t1(showMessage);
thread t2(showMessage);
thread t3(showMessage);
//get id of all the threads
thread::id id1 = t1.get_id();
thread::id id2 = t2.get_id();
thread::id id3 = t3.get_id();
//join all the threads
if (t1.joinable())
{
        t1.join();
        cout << "Thread with id " << id1 << " is terminated" << endl;
}
if (t2.joinable())
{
        t2.join();
        cout << "Thread with id " << id2 << " is terminated" << endl;
}
if (t3.joinable())
{
        t3.join();
        cout << "Thread with id " << id3 << " is terminated" << endl;
}
```

Every thread prints its unique identifier after it finishes its execution.

A Complete C++ Program

A *C++* program is given below. It launches three thread from the main function. Each thread is called using one of the callable objects specified above.

```
// CPP program to demonstrate multithreading using three different callables.

#include <iostream>
#include <thread>
using namespace std;

// A dummy function
void foo(int Z)
{
   for (int i = 0; i < Z; i++) {
      cout << "Thread using function"
           " pointer as callable\n";
   }
}
```

```
// A callable object
class thread_obj {
public:
    void operator()(int x)
    {
        for (int i = 0; i < x; i++)
            cout << "Thread using function"
                 " object as  callable\n";
    }
};

int main()
{
    cout << "Threads 1 and 2 and 3 "
         "operating independently" << endl;

    // This thread is launched by using
    // function pointer as callable
    thread th1(foo, 3);

    // This thread is launched by using
    // function object as callable
    thread th2(thread_obj(), 3);

    // Define a Lambda Expression
    auto f = [](int x) {
        for (int i = 0; i < x; i++)
            cout << "Thread using lambda"
             " expression as callable\n";  };
    // This thread is launched by using
    // lamda expression as callable
    thread th3(f, 3);

    // Wait for the threads to finish
    // Wait for thread t1 to finish
    th1.join();

    // Wait for thread t2 to finish
    th2.join();

    // Wait for thread t3 to finish
    th3.join();

    return 0;
}
```

References:

https://www.educative.io/blog/modern-multithreading-and-concurrency-in-cpp
https://www.tutorialcup.com/cplusplus/multithreading.htm#Initializing_thread_with_a_function
https://www.geeksforgeeks.org/multithreading-in-cpp/