

Concurrency Support in C++11

Page 1:

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.

```
#include <iostream>
```

```
#include <thread>
```

```
int main()
```

```
{
```

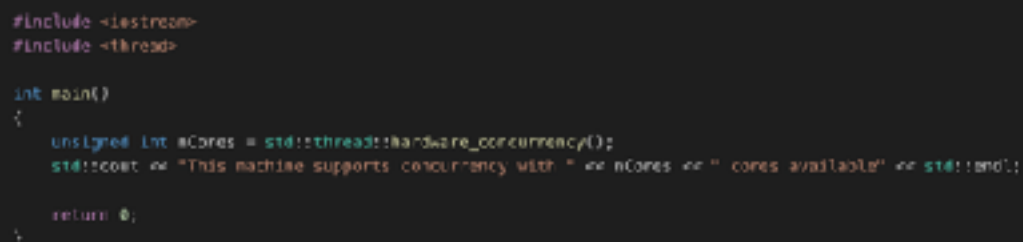
```
    std::cout << "Hello concurrent world from main! Thread id =  
" << std::this_thread::get_id() << std::endl;
```

```
    return 0;
```

```
}
```

Page 2:

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.

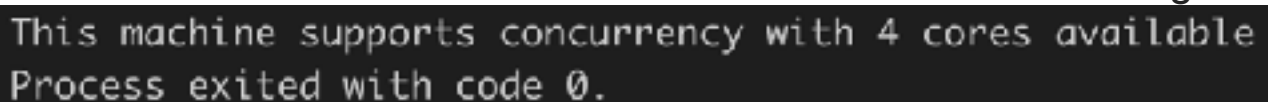


```
#include <iostream>
#include <thread>

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 4 cores available
Process exited with code 0.
```

Try running this code to see what results you get!

```
#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;
}
```

Page 3:

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 `pthread` 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
Process exited with code 0.
```

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?

```
#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;
```

```
}
```

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.

```
Finished work 1 in thread  
Finished work 1 in main  
Finished work 2 in thread  
Finished work 2 in main
```

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.

```
Finished work 1 in thread  
Finished work 1 in main  
Finished work 2 in main  
Finished work 2 in thread
```

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.

```
#include <iostream>
```

```
#include <thread>
```

```
void threadFunction()
```

```
{
```

```

    std::this_thread::sleep_for(std::chrono::milliseconds(50)); //
simulate work
    std::cout << "Finished work 1 in thread\n";

    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    std::cout << "Finished work 2 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 1 in main\n";

    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    std::cout << "Finished work 2 in main\n";

    // wait for thread to finish
    t.join();

    return 0;
}

```

Page 5:

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:

```
Finished work 1 in thread
Finished work 2 in thread
Finished work 1 in main
Finished work 2 in main
```

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.

```
#include <iostream>
```

```
#include <thread>
```

```
void threadFunction()
```

```
{
```

```
    std::this_thread::sleep_for(std::chrono::milliseconds(50)); //
```

```
    simulate work
```

```
    std::cout << "Finished work 1 in thread\n";
```

```
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
```

```
    std::cout << "Finished work 2 in thread\n";
```

```
}
```

```
int main()
```

```
{
```

```
    // create thread
```

```
    std::thread t(threadFunction);
```

```
    // wait for thread to finish
```

```
    t.join();
```

```
    // do something in main()
```

```
    std::this_thread::sleep_for(std::chrono::milliseconds(50)); //
```

```
    simulate work
```

```
    std::cout << "Finished work 1 in main\n";
```



```
std::this_thread::sleep_for(std::chrono::milliseconds(50));  
std::cout << "Finished work 2 in main\n";  
  
return 0;  
}
```

Page 6:

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 programmer 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.

```

#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;
}

```

Page 7:

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 whether the

increment variable is even or odd, you should pass the respective function to the thread.

HIDE SOLUTION

```
#include <iostream>
#include <thread>

void threadFunctionEven()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(1))
    ; // simulate work
    std::cout << "Even thread\n";
}

/* Student Task START */
void threadFunctionOdd()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(1))
    ; // simulate work
    std::cout << "Odd thread\n";
}
/* Student Task END */

int main()
{
    /* Student Task START */
    for (int i = 0; i < 6; ++i)
    {
        if (i % 2 == 0)
        {
            std::thread t(threadFunctionEven);
            t.detach();
        }
        else
        {
            std::thread t(threadFunctionOdd);
            t.detach();
        }
    }
    /* Student Task END */
}
```

```
// ensure that main does not return before the  
threads are finished  
  
std::this_thread::sleep_for(std::chrono::milliseconds(1))  
; // simulate work  
  
std::cout << "End of main is reached" << std::endl;  
return 0;  
}
```

Page 8:

Run the program several times and look the console output. What do you observe? As a second experiment, comment out the `sleep_for` function in the main thread. What happens to the detached threads in this case?

```

#include <iostream>
#include <thread>

void threadFunctionEven()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(1)); // simulate work
    std::cout << "Even thread\n";
}

/* Student Task START */
void threadFunctionOdd()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(1)); // simulate work
    std::cout << "Odd thread\n";
}
/* Student Task END */

int main()
{
    /* Student Task START */
    for (int i = 0; i < 6; ++i)
    {
        if (i % 2 == 0)
        {
            std::thread t(threadFunctionEven);
            t.detach();
        }
        else
        {
            std::thread t(threadFunctionOdd);
            t.detach();
        }
    }
    /* Student Task END */

    // ensure that main does not return before the threads are finished
    std::this_thread::sleep_for(std::chrono::milliseconds(1)); // simulate work

    std::cout << "End of main is reached" << std::endl;
    return 0;
}

```

HIDE SOLUTION

The order in which even and odd threads are executed changes. Also, some threads are executed after the main function reaches its end. When `sleep_for` is removed, threads will not finish before the program terminates.