7COM1025
Programming for Software Engineers
Lecture 29

Dr. Renato C. de Amorim

1 THREAD WITH FUNCTION POINTER

```
#include <iostream>
#include <thread>
using namespace std;
void counter(int id, int numIterations){
        for (int i=0; i<numIterations;i++) {
            cout<<"Counter "<<id << " has value ";
            cout<< i <<endl;
       }
}
int main() {
  thread t1(counter, 1, 6);
  t1.join();
  cout<<"end"<<endl;
  return 0;
}
```

MULTITHREADED PROGRAMMING

C++11 includes a standard threading library.

The constructor of thread is a variadic template, it accepts any number of arguments.

After launching the threads main() calls join(). This is to make sure the main thread keeps running until the other thread is finished.

A call to t1.join() blocks until the thread t1 is finished. With no join() call, main() would finish immediately after launching the new thread. This would make the application end, causing all children threads to be terminated as well.
(whether they are finished or not).

Join() calls are necessary in small examples. In real-world applications you should avoid using it because it causes the thread calling join() to block.

2 THREADS WITH FUNCTION POINTER

What is the output?

```
#include <iostream>
#include <thread>
using namespace std;
void counter(int id, int numIterations)
{
        for (int i=0; i<numIterations;i++) {
            cout<<"Counter "<<id << " has value "<< i<< endl;
        }
}
int main() {
  thread t1(counter, 1, 6);
  thread t2(counter, 2, 6);
  t1.join();
  t2.join();
  cout<<"end"<<endl;
  return 0;
}
```

EXERCISE 29.1

Write a function: bool is_prime(unsigned int x)

Now, write the function: void print_all_primes(unsigned int max)
The function above should print all the prime numbers between 2 and max.
In order to speed things up, your application should use threads.

You may want to write more functions to accomplish the functionality above.

THREAD WITH FUNCTION OBJECT

```
#include <iostream>
#include <thread>
using namespace std;
class Counter {
    public:
        Counter(int id, int numIterations): mId(id), mNumIterations(numIterations){}
        void operator() () const {
            for (int i =0; i<mNumIterations; ++i){
                cout<<"Counter "<<mId<< " has value "<<i<<endl;
        }}
    private:
    int mId, mNumIterations;};
int main() {
  thread t1{Counter{1,20}};//Uniform intialisation
  Counter c(2,20); //using a named variable;
  thread t2(c);
  thread t3(Counter(3,10)); //using temp
  t1.join();  t2.join();  t3.join();
  return 0;
}
```

THREAD LOCAL STORAGE

You can declare a variable so that each thread has its own unique copy of the variable. What is the output of the below?

```
#include <iostream>
#include <string>
#include <thread>
using namespace std;
thread_local unsigned int counter = 1;
void increase_counter(const string& thread_name) {
    ++counter; //this is a thread-local variable
    std::cout << "Counter for " << thread_name << ": " << counter << '\n';
}
int main() {
    std::thread t1(increase_counter, "Thread 1"), t2(increase_counter, "Thread 2");
    std::cout << "Counter for main: " << counter << '\n';
    t1.join(); t2.join();
    return 0;
}
```

RETURNING VALUES

There are many ways you can return values from a thread.

Possibly the easiest one is to pass a pointer or a reference as a parameter. This parameter may be set by the thread.

RETURNING VALUES

Identify in your assignment a place in which it may benefit from using threads.
Make all necessary changes.