Introduction to C++11 Threads

In C++11 a wrapper Thread class is available. However, you still need to have a library that implements the thread class. Visual Studio supports C++11 threads, but some other IDE's and C++ compilers don't. In Linux we will need to use the pthread library.

A thread is a separate computation process. In Java and C++, you can have programs with multiple threads. You can think of the threads as computations that execute in parallel. On a computer with enough processors, the threads might indeed execute in parallel. In most normal computing situations, the threads do not really execute in parallel. Instead, the computer switches resources between threads so that each thread in turn does a little bit of computing. To the user this looks like the processes are executing in parallel.

You have already experienced this idea. Consider Visual Studio – as you are typing your code it is able to check for errors and draw a red squiggle if one is detected. This happens "at the same time" while you are entering your program, you don't have to stop typing and select a "check for errors" command. Visual Studio and the operating system is using threads to make this happen. There may or may not be some work being done in parallel depending on your computer and operating system. Most likely the two computation threads are simply sharing computer resources so that they take turns using the computer's resources.

Threads are useful when you want some processing to continue when another part is blocked/stopped (perhaps waiting for input). Threads may also run faster, if some computations can be run on separate CPU's or cores on your system. In GPU programming you can have hundreds of thousands of threads! It is now possible to run what used to be the equivalent of a supercomputer on a GPU-enabled server or workstation.

Let's show how to fire off a function that runs in a separate thread:

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

void func(int a)
{
    cout << "Hello World: " << a << " " << this_thread::get_id()
<< endl;
}

int main()
{
    thread t1(func, 10);
    thread t2(func, 20);
    t1.join();
    t2.join();

    return 0;
}
```

Compile this program in Linux with:

```
 g++ program.cpp -std=c++11 -lpthread
```

This program starts off two threads, each runs the function "func". Each thread is automatically given a unique ID, which we can access if desired from get_id(). It is often useful to pass in an ID number, which we did by passing in the number in variable a.

If you run this, you'll see the two threads run and output Hello World. Once a thread is started we have no control over when it runs – it is now up to the operating system! You can see this by running the threads and seeing the output from each thread overwriting the other. This is because while one thread is in the middle of outputting its message, there is a context switch and we run the second thread, which spits out its text right in the middle of the text from the first thread.

The join() function makes the main function wait for each thread to finish before continuing. This is important to synchronize multiple threads. **If the join is missing, the main thread could exit before the child threads finish, causing an error.**

If we want to avoid the threads overwriting each other, we can add a **mutex**, for mutual exclusion. This locks the thread so only one thread can enter a region of code at a time. This is extremely important for some programs to prevent deadlock or other types of errors (you see more of this in operating systems). The following modification forces other threads to wait so only one at a time can run the code in func:

```
#include <mutex>

using namespace std;

mutex global_lock;

void func(int a)
{
    global_lock.lock();
    cout << "Hello World: " << a << " " << this_thread::get_id()
<< endl;
    global_lock.unlock();
}
```

It is common to want a lot more than one or two threads. In this case, we can make an array of threads. Here is some code in main that makes an array of 10 threads:

```
thread tarr[10];
for (int i =0; i < 10; i++)
    tarr[i] = thread(func, i);
for (int i =0; i < 10; i++)
    tarr[i].join();
```

Notice the unpredictability of which thread runs first!

```
      Hello World: 0 140198342674176
      Hello World: 3 140198311204608
      Hello World: 2 140198321694464
      Hello World: 4 140198300714752
      Hello World: 1 140198332184320
      Hello World: 5 140198290224896
      Hello World: 6 140198279735040
      Hello World: 7 140198269245184
      Hello World: 8 140198258755328
      Hello World: 9 140198248265472
```

It is common to want to run a class in a thread. Here is the template. In this case we called the class “Runnable” so it is similar to Java but it could be whatever name you like:

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

using namespace std;

class Runnable
{
  public:
    Runnable();
    Runnable(int n);
    void operator()();  // Yes, there are two ()()
  private:
    int num;
};
Runnable::Runnable() : num(0)
{ }
Runnable::Runnable(int n) : num(n)
{ }
void Runnable::operator()()
{
        cout << "Hello world, I am number " << num <<  endl;
}


int main()
{
    Runnable r1(10);
    Runnable r2(20);

    thread t1(r1);
    thread t2(r2);

    t1.join();
    t2.join();
    return 0;
}
```

When the thread starts, the class Runnable executes the code in the operator()() method. Any data we want to pass to the thread is generally sent in the constructor.

Here is one more example. In this case, we do something useful! We have three threads, and each one is searching (perhaps in parallel) a portion of an array for the minimum value. The minimum each thread finds for each section is stored in a “results” array, where we have a slot reserved for each thread. The main function has to go through the results to find the overall minimum.

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

using namespace std;

class Runnable
{
 public:
  Runnable();
  Runnable(int *target, int *results, int num, int start, int end);
  void operator()();
 private:
  int *target, *results;
  int num, start, end;
};

Runnable::Runnable()
{
 target=nullptr;
 results=nullptr;
 num=0;
 start=0;
 end=0;
}
Runnable::Runnable(int *target, int *results, int num, int start,
                   int end)
{
  this->target= target;
  this->results = results;
  this->num = num;
  this->start = start;
  this->end = end;
}
void Runnable::operator()()
{
 int min = target[start];
 for (int i=start+1; i<=end; i++)
 {
        if (target[i]<min)
                min = target[i];
 }
 results[num] = min;
}
int main(int, char **)
{
   thread tarr[3];
   int target[] = {31, 66, 41, 8, 92, 47, 22, 87, 45, 92, 4, 14};
   int results[] = {999, 999, 999, 999};
   for (int i = 0; i < 3; i++)
   {
        Runnable r(target, results, i, i*4, i*4+3);
        tarr[i] = thread(r);
   }
    for (int i =0; i < 3; i++)
        tarr[i].join();
    for (int i =0; i < 3; i++)
        cout << results[i] << endl;
    int min = results[0];
    if (min > results[1])
        min = results[1];
    if (min > results[2])
        min = results[2];
    cout << "The minimum from threaded min-search is " << min << endl;
    return 0;
}
```

The output:

```
8
22
4
The minimum from threaded min-search is 4
```

The code sends in the array to search, an array for results, an ID, and bounds for each thread to search. Each thread then searches its portion of the array, finds the minimum, and uses its ID to determine a unique spot to place its result in the results array.

Notice that we passed in an array (a pointer) to each thread to store its results. This is the general way a thread can communicate back to another thread. We can't just store the results in a class variable, because each thread is like a separate running program and has its own class variables.

To demonstrate, consider this code:

```
class Runnable
{
 public:
  Runnable();
  void operator()();
  int num;
};

Runnable::Runnable()
{
 num=1;
 cout << "In constructor " << num << endl;
}
void Runnable::operator()()
{
 num=2;
 cout << "In op " << num << endl;
}
```

```
int main(int, char **)
{
   Runnable r;
   thread t;

   t = thread(r);
   t.join();
   cout << "In main " << r.num << endl;
   return 0;
}
```

The output is:

```
In constructor 1
In op 2
In main 1
```

We aren’t able to make a change in the operator()() function and have that sent back to the caller, because it’s in another thread. But we can send data back if it’s through a pointer:

```
class Runnable
{
 public:
  Runnable(int *n);
  void operator()();
  int *num;
};

Runnable::Runnable(int *n)
{
 num = n; // Save pointer to n, allocated in main
 cout << "In constructor " << *num << endl;
}
void Runnable::operator()()
{
 *num = 2;
 cout << "In op " << *num << endl;
}

int main()
{
   int n=9;
   Runnable r(&n);
   thread t;

   t = thread(r);
   t.join();
   cout << "In main " << n << endl;
   return 0;
}
```

```
In constructor 9
In op 2
In main 2
```

There is a lot more to threads but at least you should have an idea of what they are and where they may be useful from this brief introduction.