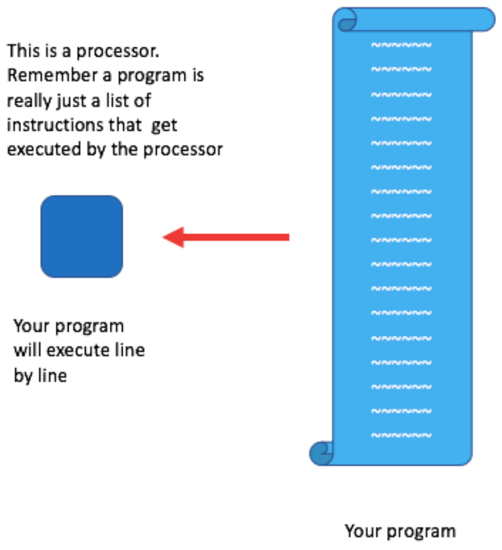
Additional Topics (multithreading, streams pt 2)

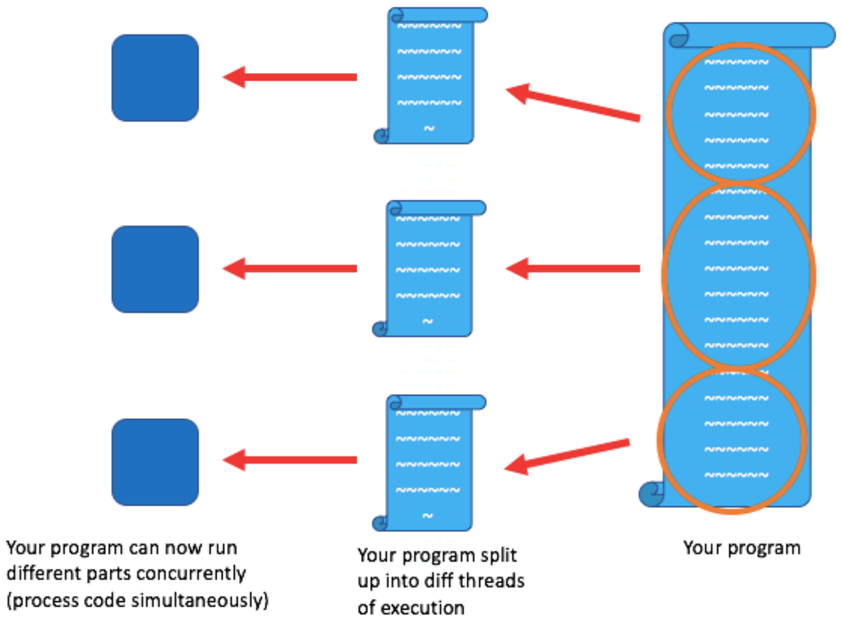
Multithreading

<u>What's actually happening:</u>


This is a processor. Remember a program is really just a list of instructions that get executed by the processor
Your program will execute line by line
Your program

*Note: there are more complicated aspects not mentioned here, like translating your programming language into something the computer understands

You can imagine that if you had a very long list of instructions, the processor would take a long time to actually execute these instructions. It would be faster to split this list up and process each smaller list separately and simultaneously (like hiring 3 people to do a task at once instead of just having 1 person do it):


Your program can now run different parts concurrently (process code simultaneously)
Your program split up into diff threads of execution
Your program

How C++ handles it:

- C++ has a class called *thread* that you include in your program
- You can create threads of execution:

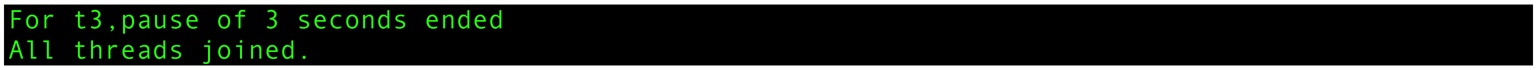
```
computer$ g++ -std=c++11 thread_runTogether.cpp
computer$ ./a.out
Joining threads...:
For t1,pause of For For 3t3t2 seconds ended
,pause of ,pause of 33 seconds ended
 seconds ended
All threads joined.
```

(Notice the thread executions "run into" each other-all trying to do the same thing at once)

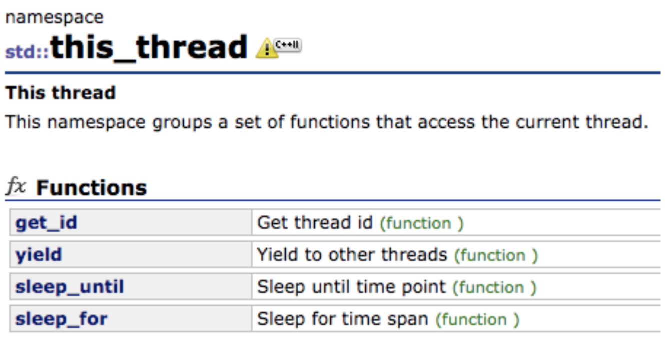
```
#include <iostream>      // std::cout
#include <thread>        // std::thread, std::this_thread::sleep_for
#include <chrono>        // std::chrono::seconds

void pause_thread(int n, std::string th)
{
  std::this_thread::sleep_for (std::chrono::seconds(n)); //sleep means the execution of the thread pauses for the
given time
  std::cout << "For "<<th<<",pause of " << n << " seconds ended\n";
}

int main()
{
  //create threads-each of them has the list of instructions given by pause_thread
  std::thread t1 (pause_thread,3,"t1");
  std::thread t2 (pause_thread,3,"t2");
  std::thread t3 (pause_thread,3,"t3");
  std::cout << "Joining threads...:\n";

  t1.join(); //thread executes-function returns when thread is done executing
  t2.join();
  t3.join();
  std::cout << "All threads joined.\n";

  return 0;
}
```

We can "lock" our execution down so we don't have threads "running" into each other (one executes at at a time-for example, using std::cout one at a time) using something called a mutex *(a mutual exclusion object that prevents two properly written threads from concurrently accessing a critical resource)*

```
computer$ g++ -std=c++11 thread_mutex.cpp
computer$ ./a.out
Joining threads!
For t1,pause of 3 seconds ended
For t2,pause of 3 seconds ended
```



```
For t3,pause of 3 seconds ended
All threads joined.
```



namespace
std::**this_thread**

This thread
This namespace groups a set of functions that access the current thread.

fx **Functions**

get_id	Get thread id (function)
yield	Yield to other threads (function)
sleep_until	Sleep until time point (function)
sleep_for	Sleep for time span (function)

```
#include <iostream>       // std::cout
#include <thread>        // std::thread, std::this_thread::sleep_for
#include <chrono>         // std::chrono::seconds
#include <mutex>

void pause_thread(int n, std::string th, std::mutex &mtx)
{
 std::lock_guard<std::mutex> lock(mtx);
          std::this_thread::sleep_for (std::chrono::seconds(n)); //this thread is a namespace
          std::cout << "For "<<th<<",pause of " << n << " seconds ended\n";
}

//You can also manually lock and unlock the mutex (lock_guard handles that for you):
//void pause_thread(int n, std::string th, std::mutex &mtx)
//{
//        mtx.lock();
 //       std::this_thread::sleep_for (std::chrono::seconds(n));
 //       std::cout << "For "<<th<<",pause of " << n << " seconds ended\n";
//        mtx.unlock();
//}

int main()
{
  std::mutex mtx;
 std::thread t1 (pause_thread,3,"t1", ref(mtx));
 std::thread t2 (pause_thread,3,"t2", ref(mtx));
 std::thread t3 (pause_thread,3,"t3", ref(mtx));
 std::cout << "Joining threads!\n";
 t1.join();
 t2.join();
 t3.join();
 std::cout << "All threads joined.\n";

 return 0;
}
```

(show horse race example-show all steps printed to screen)

```
C++ computer$ cd 23/
23 computer$ cd horserace/
horserace computer$ ./a.out
```

Go over (on board): **Process vs thread**

The following is NOT a multithreaded program (only one thread is running-the main thread):

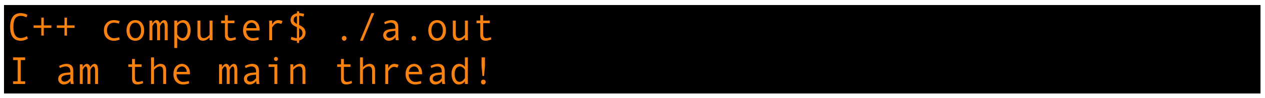
```
#include <iostream>
#include <thread>
#include <chrono>

void sample_function()
{
 std::cout<<"Hello from the the program!"<<std::endl;
}

int main(int argc, char **argv)
{
 sample_function();
}
```

The following is a multithreaded program:

```
C++ computer$ ./a.out
I am the main thread!
Hello from the the program!
```

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

void sample_function()
{
 std::cout<<"Hello from the the program!"<<std::endl;
}

int main(int argc, char **argv)
{
  std::cout<<"I am the main thread!"<<std::endl; //main executes
 std::thread t1(sample_function); //child thread created
 t1.join(); // child thread joins main and executes finish

}
```

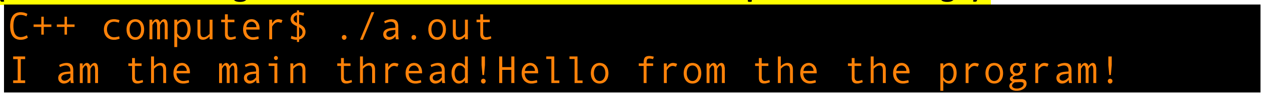
Notice what happens if detach my child thread (only the main statement executes):



```
C++ computer$ ./a.out
I am the main thread!
```

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

void sample_function()
{
  std::cout<<"Hello from the the program!"<<std::endl;
}

int main(int argc, char **argv)
{
  std::cout<<"I am the main thread!"<<std::endl;
  std::thread t1(sample_function); //child thread
  t1.detach(); //this detaches the child thread from main-it will run by itself.  Since the main thread will
execute faster (it will take time to create the thread and run it), the program finishes before it gets a
chance to execute).
}
```

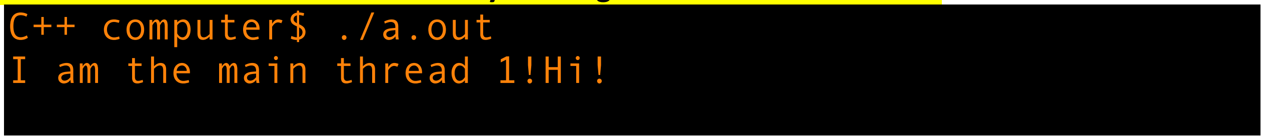
Notice if I even create the child thread first, the main thread still prints out faster than the child thread (but there is enough time for the child thread to at least print its message):



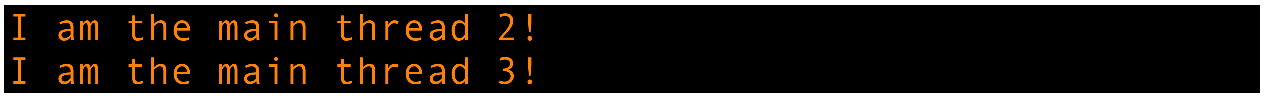
```
C++ computer$ ./a.out
I am the main thread!Hello from the the program!
```

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

void sample_function()
{
  std::cout<<"Hello from the the program!"<<std::endl;
}

int main(int argc, char **argv)
{
  std::thread t1(sample_function); //child thread
  t1.detach(); //this detaches the child thread from main-it will run by itself

  std::cout<<"I am the main thread!"<<std::endl;
}
```

You can see the detached child thread clearly running at the same time below:



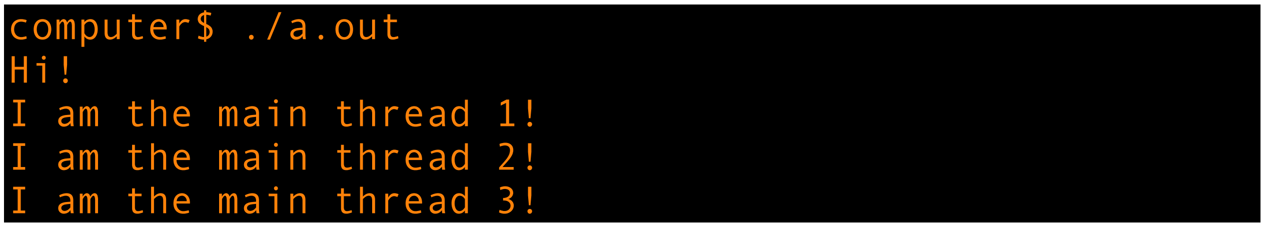
```
C++ computer$ ./a.out
I am the main thread 1!Hi!
```



```
I am the main thread 2!
I am the main thread 3!
```

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

void sample_function()
{
 std::cout<<"Hi!"<<std::endl;
}

int main(int argc, char **argv)
{
 std::thread t1(sample_function); //child thread
 t1.detach(); //this detaches the child thread from main-it will run by itself

 std::cout<<"I am the main thread 1!"<<std::endl;
 std::cout<<"I am the main thread 2!"<<std::endl;
 std::cout<<"I am the main thread 3!"<<std::endl;
}
```

With join, that would not happen:

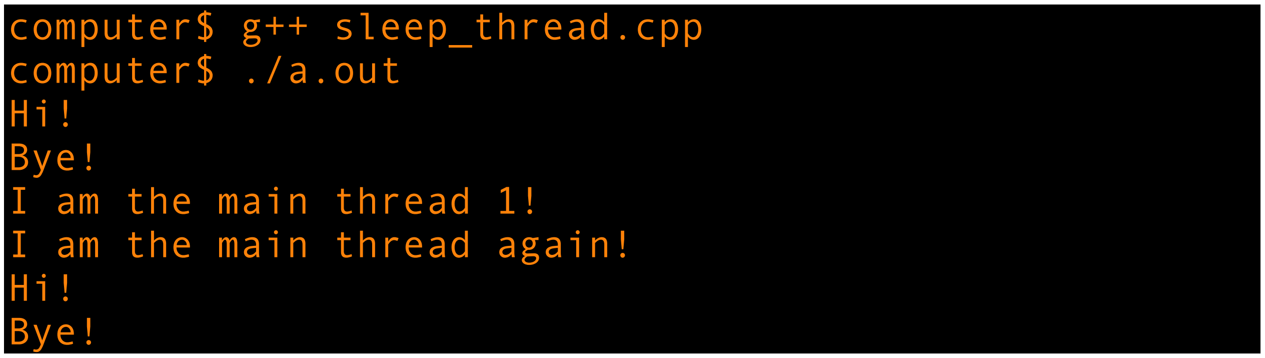


```
computer$ ./a.out
Hi!
I am the main thread 1!
I am the main thread 2!
I am the main thread 3!
```

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

void sample_function()
{
 std::cout<<"Hi!"<<std::endl;
}

int main(int argc, char **argv)
{
 std::thread t1(sample_function); //child thread
 t1.join();

 std::cout<<"I am the main thread 1!"<<std::endl;
 std::cout<<"I am the main thread 2!"<<std::endl;
 std::cout<<"I am the main thread 3!"<<std::endl;
```

}

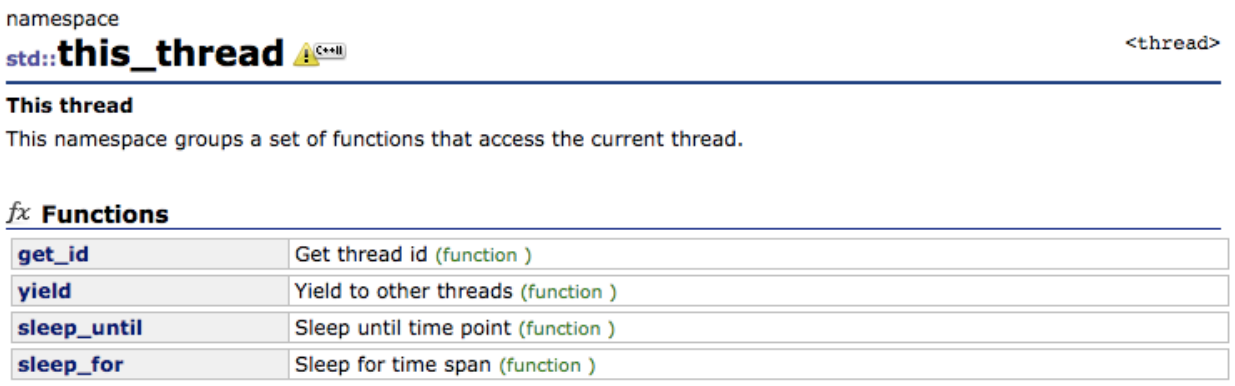
Sleeping a thread (using this_thread):



```
computer$ g++ sleep_thread.cpp
computer$ ./a.out
Hi!
Bye!
I am the main thread 1!
I am the main thread again!
Hi!
Bye!
```

Notice each thread sleeps for a second-both children and main thread:

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

void sample_function()
{
 std::cout<<"Hi!"<<std::endl;
 std::this_thread::sleep_for(std::chrono::seconds(1));
 std::cout<<"Bye!"<<std::endl;
}

int main(int argc, char **argv)
{
 std::thread t1(sample_function); //child thread created
 t1.join();

 std::cout<<"I am the main thread 1!"<<std::endl;
 std::this_thread::sleep_for(std::chrono::seconds(1));
 std::cout<<"I am the main thread again!"<<std::endl;

 std::thread t2(sample_function); //child thread created
 t2.join();
}
```



namespace
std::**this_thread** C++11 <thread>

This thread
This namespace groups a set of functions that access the current thread.

fx **Functions**

get_id	Get thread id (function)
yield	Yield to other threads (function)
sleep_until	Sleep until time point (function)
sleep_for	Sleep for time span (function)

Additional topics (not covered in this class, but you can look them up):
Functors
Lambda expressions

Streams, PT 2:

- The sequence of bytes "flowing" to and from your program
- In C++, we create stream objects to help us with this process.
- For example:
  - From the keyboard to your program
  - From your program to the screen

We have four main stream objects we deal with in the iostream header:

- cin
  - standard input stream (defaulted to the keyboard)
- cout
  - standard output stream (defaulted to the console)
- cerr
  - standard error stream (defaulted to the console)
  - unbuffered (meaning each character is flushed as you write it)
- clog
  - standard log stream (defaulted to the console)
  - buffered

Note: As stated previously, we can also have streams between a program and file.

A main difference between the standard output stream (*cout*) and the error stream (*cerr, clog*) is if redirection occurs. *cout* will not print to screen (it will go to a file) but the error stream will.

Both *cerr* and *clog* are part of the error stream. A main difference is that *cerr* is unbuffered (written to the device immediately) and *clog* is buffered (*cout* is also buffered). We can see the difference between buffered and non-buffered below:

cout (buffered) vs cerr (unbuffered)

The following will print to screen all at once at the end (not written to screen immediately):

```
computer$ g++ buffer_example.cpp
computer$ ./a.out
1 2 3 4 5
```

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

```
int main(int argc, char **argv)
{
  for (int i = 1; i <= 5; ++i)
  {
    cout << i << " ";
    this_thread::sleep_for(chrono::seconds(1)); //slowing down the program so you can see what
happens
  }
  return 0;
}
```

The following will print to screen with each iteration (since cerr is unbuffered, the output is written to screen immediately):

```
computer$ g++ unbuffer_example.cpp
computer$ ./a.out
1 2 3
```

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
int main(int argc, char **argv)
{
  for (int i = 1; i <= 5; ++i)
  {
    cerr << i << " ";
    this_thread::sleep_for(chrono::seconds(1));
  }
  return 0;
}
```

We can use the flush manipulator (remember a manipulator manipulates a stream-check previous lecture about streams for more examples) to force a buffered stream to output to screen immediately:

```
computer$ g++ buffer_flush.cpp
computer$ ./a.out
1 2 3
```

Notice here that cout is acting like cerr before (because we are flushing the buffer with each iteration)

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
int main(int argc, char **argv)
{
  for (int i = 1; i <= 5; ++i)
  {
    cout << i << " "<<flush;
    this_thread::sleep_for(chrono::seconds(1));
  }
  return 0;
}
```

The endl manipulator is simply a flush with a newline character added:

```
computer$ g++ buffer_endl.cpp
computer$ ./a.out
1
2
3
```

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
int main(int argc, char **argv)
{
  for (int i = 1; i <= 5; ++i)
  {
    cout << i << " "<<endl;
    this_thread::sleep_for(chrono::seconds(1));
  }
  return 0;
}
```

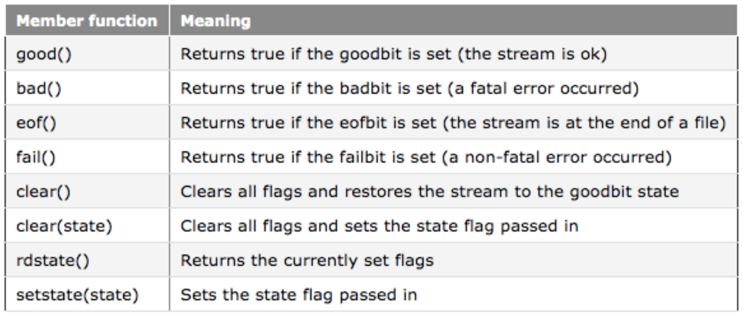
Stream states:

When using streams, we can use a flag to indicate the current state of a stream (a flag is a bit or sequence of bits typically meant to “communicate” something):

Flags used for streams include:

goodbit
badbit
eofbit
failbit

You can access these states using functions:



Member function	Meaning
good()	Returns true if the goodbit is set (the stream is ok)
bad()	Returns true if the badbit is set (a fatal error occurred)
eof()	Returns true if the eofbit is set (the stream is at the end of a file)
fail()	Returns true if the failbit is set (a non-fatal error occurred)
clear()	Clears all flags and restores the stream to the goodbit state
clear(state)	Clears all flags and sets the state flag passed in
rdstate()	Returns the currently set flags
setstate(state)	Sets the state flag passed in

(GO OVER STUDY REVIEW)