Multithreading support was introduced in C+11. Prior to C++11, we had to use POSIX threads or p threads library in C. While this library did the job the lack of any standard language provided feature-set caused serious portability issues. C++ 11 did away with all that and gave us **std::thread**. The thread classes and related functions are defined in the **thread** header file.

**std::thread** is the thread class that represents a single thread in C++. To start a thread we simply need to create a new thread object and pass the executing code to be called (i.e, a callable object) into the constructor of the object. Once the object is created a new thread is launched which will execute the code specified in callable.

A callable can be either of the three

* A function pointer
* A function object
* A lambda expression

After defining callable, pass it to the constructor.

|  |
| --- |
| Ex: import<thread>  std::thread thread\_object(callable) |

**Launching thread using function pointer**  
The following code snippet demonstrates how this is done

|  |
| --- |
| void foo(param)  {      // Do something  }    // The parameters to the function are put after the comma  std::thread thread\_obj(foo, params); |

**Launching thread using lambda expression**

The following code snippet demonstrates how this is done

|  |
| --- |
| // Define a lamda expression  auto f = [](params) {      // Do Something  };    // Pass f and its parameters to thread  // object constructor as  std::thread thread\_object(f, params); |

We can also pass lambda functions directly to the constructor.

|  |
| --- |
| std::thread thread\_object([](params) {      // Do Something  };, params); |

**Launching threads using function objects**

The following code snippet demonstrates how this is done

|  |
| --- |
| // Define the class of function object  class fn\_object\_class {      // Overload () operator      void operator()(params)      {          // Do Something      }  }    // Create thread object  std::thread thread\_object(fn\_class\_object(), params) |

**Waiting for threads to finish**

Once a thread has started we may need to wait for the thread to finish before we can take some action. For instance, if we allocate the task of initializing the GUI of an application to a thread, we need to wait for the thread to finish to ensure that the GUI has loaded properly. To wait for a thread use the **std::thread::join()** function. This function makes the current thread wait until the thread identified by **\*this** has finished executing.  
For instance, to block the main thread until thread t1 has finished we would do

|  |
| --- |
| int main()  {      // Start thread t1      std::thread t1(callable);      // Wait for t1 to finish      t1.join();      // t1 has finished do other stuff        ...  } |

**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 for multithreading using three different callables.  #include <thread>  using namespace std;  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      th1.join();      th2.join();      th3.join();        return 0;  } |

Output (Machine Dependent)

Threads 1 and 2 and 3 operating independently

Thread using function pointer as callable

Thread using lambda expression as callable

Thread using function pointer as callable

Thread using lambda expression as callable

Thread using function object as callable

Thread using lambda expression as callable

Thread using function pointer as callable

Thread using function object as callable

Thread using function object as callable

Note:  
To compile programs with std::thread support use

g++ -std=c++11 –pthread

**Function object without params:**

#include <iostream>

#include <thread>

class DisplayThread

{

public:

void operator()()

{

for(int i = 0; i < 10000; i++)

std::cout<<"Display Thread Executing"<<std::endl;

}

};

int main()

{

std::thread threadObj( (DisplayThread()) );

for(int i = 0; i < 10000; i++)

std::cout<<"Display From Main Thread "<<std::endl;

std::cout<<"Waiting For Thread to complete"<<std::endl;

threadObj.join();

std::cout<<"Exiting from Main Thread"<<std::endl;

return 0;

}

**Lambda Expression Ex2:**

#include <iostream>

#include <thread>

int main()

{

int x = 9;

std::thread threadObj([]{

for(int i = 0; i < 5; i++)

std::cout<<"Display Thread Executing"<<std::endl;

});

for(int i = 0; i < 5; i++)

std::cout<<"Display From Main Thread"<<std::endl;

threadObj.join();

std::cout<<"Exiting from Main Thread"<<std::endl;

return 0;

}

**Differentiating between threads**

Each of the std::thread object has an associated ID and we can fetch using,

Member function, gives id of associated thread object i.e.

|  |  |
| --- | --- |
|  | std::thread::get\_id() |

To get the identifier for the current thread use,

|  |  |
| --- | --- |
|  | std::this\_thread::get\_id() |

If std::thread object does not have an associated thread then get\_id() will return a default constructed std::thread::id object i.e. “not any thread.”

std::thread::id is a Object, it can be compared and printed on console too. Let’s look at an example,

|  |  |
| --- | --- |
|  | #include <iostream>  #include <thread>  void thread\_function()  {      std::cout<<"Inside Thread :: ID  = "<<std::this\_thread::get\_id()<<std::endl;  }  int main()  {      std::thread threadObj1(thread\_function);      std::thread threadObj2(thread\_function);        if(threadObj1.get\_id() != threadObj2.get\_id())          std::cout<<"Both Threads have different IDs"<<std::endl;        std::cout<<"From Main Thread :: ID of Thread 1 = "<<threadObj1.get\_id()<<std::endl;      std::cout<<"From Main Thread :: ID of Thread 2 = "<<threadObj2.get\_id()<<std::endl;        threadObj1.join();      threadObj2.join();      return 0;  } |

## ****Joining Threads with std::thread::join()****

Once a thread is started then another thread can wait for this new thread to finish. For this another need to call join() function on the std::thread object i.e.

|  |  |
| --- | --- |
|  | std::thread th(funcPtr);   // Some Code   th.join(); |

Let’s see an example ,

Suppose Main Thread has to start 10 Worker Threads and after starting all these threads, main function will wait for them to finish. After joining all the threads main function will continue,

|  |  |
| --- | --- |
|  | #include <iostream>  #include <thread>  #include <algorithm>  class WorkerThread  {  public:      void operator()()      {          std::cout<<"Worker Thread "<<std::this\_thread::get\_id()<<" is Executing"<<std::endl;      }  };  int main()  {      std::vector<std::thread> threadList;      for(int i = 0; i < 10; i++)      {          threadList.push\_back( std::thread( WorkerThread() ) );      }      // Now wait for all the worker thread to finish i.e.      // Call join() function on each of the std::thread object      std::cout<<"wait for all the worker thread to finish"<<std::endl;      std::for\_each(threadList.begin(),threadList.end(), std::mem\_fn(&std::thread::join));      std::cout<<"Exiting from Main Thread"<<std::endl;      return 0;  } |