**How to Stop or Terminate a Thread**

C++11 does not provides a direct method to stop a running thread and that’s because that thread might have some resources to release or close before exit i.e.

* What if a thread has acquired a lock and we kill that thread suddenly, then who’s gonna release that lock ?
* What if a thread has opened a file to write the text and we stopped that thread, then who’s gonna close that file ?
* What if thread has allocated memory on heap and before it could delete that memory, we stopped the thread. Then who’s gonna prevent that memory leak.

Therefore there is no direct function to close the thread. But we can notify the thread to exit and we can implement out thread in a such a way that after some interval or at some checkpoints it should check, if I am requested to exit or not. If yes then it should exit gracefully , by releasing all the resources.

## Stopping a Thread using std::future<>

We can pass a **std::future<void>** object to thread and thread should exit when value in future is available. As, we want to only signal the thread and not actually passing any value in that signal, so we will can use future object of type void.

Let’s create a promise object of type void in main function i.e.



|  |  |
| --- | --- |
| 1  2 | // Create a std::promise object  std::promise<void> exitSignal; |

Now, fetch the associated future object from this promise in main function i.e.



|  |  |
| --- | --- |
| 1  2 | //Fetch std::future object associated with promise  std::future<void> futureObj = exitSignal.get\_future(); |

Now in main function while creating the thread, pass the future object to the thread function i.e.



|  |  |
| --- | --- |
| 1  2 | // Starting Thread & move the future object in lambda function by reference  std::thread th(&threadFunction, std::move(futureObj)); |

Inside the thread, we are doing some Work and keep on checking that if thread has been requested to exit i.e. value in future is available or not.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | void threadFunction(std::future<void> futureObj)  {  std::cout << "Thread Start" << std::endl;  while (futureObj.wait\_for(std::chrono::milliseconds(1)) == std::future\_status::timeout)  {  std::cout << "Doing Some Work" << std::endl;  std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));    }  std::cout << "Thread End" << std::endl;    } |

As soon as we set value in promise object from main function, value in future object will available in thread function i.e.



|  |  |
| --- | --- |
| 1  2 | //Set the value in promise  exitSignal.set\_value(); |

Complete example is as follows,



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | #include <thread>  #include <iostream>  #include <assert.h>  #include <chrono>  #include <future>    void threadFunction(std::future<void> futureObj)  {  std::cout << "Thread Start" << std::endl;  while (futureObj.wait\_for(std::chrono::milliseconds(1)) == std::future\_status::timeout)  {  std::cout << "Doing Some Work" << std::endl;  std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));    }  std::cout << "Thread End" << std::endl;    }  int main()  {    // Create a std::promise object  std::promise<void> exitSignal;    //Fetch std::future object associated with promise  std::future<void> futureObj = exitSignal.get\_future();    // Starting Thread & move the future object in lambda function by reference  std::thread th(&threadFunction, std::move(futureObj));    //Wait for 10 sec  std::this\_thread::sleep\_for(std::chrono::seconds(10));    std::cout << "Asking Thread to Stop" << std::endl;    //Set the value in promise  exitSignal.set\_value();    //Wait for thread to join  th.join();    std::cout << "Exiting Main Function" << std::endl;  return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Thread Start  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Asking Thread to Stop  Thread End  Exiting Main Function |

## Creating a Stoppable Task

Creating promise and future object every time, when we want to create task that can be stopped, is a repeating task.  
Let’s avoid that by creating a Stoppable class that provide all this functionality in object oriented manner.

**Stoppable class that encapsulate the promise and future object**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | /\*  \* Class that encapsulates promise and future object and  \* provides API to set exit signal for the thread  \*/  class Stoppable  {  std::promise<void> exitSignal;  std::future<void> futureObj;  public:  Stoppable() :  futureObj(exitSignal.get\_future())  {    }  Stoppable(Stoppable && obj) : exitSignal(std::move(obj.exitSignal)), futureObj(std::move(obj.futureObj))  {  std::cout << "Move Constructor is called" << std::endl;  }  Stoppable & operator=(Stoppable && obj)  {  std::cout << "Move Assignment is called" << std::endl;  exitSignal = std::move(obj.exitSignal);  futureObj = std::move(obj.futureObj);  return \*this;  }    // Task need to provide defination  for this function  // It will be called by thread function  virtual void run() = 0;      // Thread function to be executed by thread  void operator()()  {  run();  }    //Checks if thread is requested to stop  bool stopRequested()  {  // checks if value in future object is available  if (futureObj.wait\_for(std::chrono::milliseconds(0)) == std::future\_status::timeout)  return false;  return true;  }  // Request the thread to stop by setting value in promise object  void stop()  {  exitSignal.set\_value();  }  }; |

We just need to extend this class in our task class and provide definition for run() function, which will like our thread function i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | /\*  \* A Task class that extends the Stoppable Task  \*/  class MyTask: public Stoppable  {  public:  // Function to be executed by thread function  void run()  {  std::cout << "Task Start" << std::endl;    // Check if thread is requested to stop ?  while (stopRequested() == false)  {  std::cout << "Doing Some Work" << std::endl;  std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));    }  std::cout << "Task End" << std::endl;  }  }; |

Now lets see how to use this Stoppable task in main function i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // Creating our Task  MyTask task;    //Creating a thread to execute our task  std::thread th([&]()  {  task.run();  });    std::this\_thread::sleep\_for(std::chrono::seconds(10));    std::cout << "Asking Task to Stop" << std::endl;  // Stop the Task  task.stop();    //Waiting for thread to join  th.join(); |

Complete example is as follows,



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104 | #include <thread>  #include <iostream>  #include <assert.h>  #include <chrono>  #include <future>    /\*  \* Class that encapsulates promise and future object and  \* provides API to set exit signal for the thread  \*/  class Stoppable  {  std::promise<void> exitSignal;  std::future<void> futureObj;  public:  Stoppable() :  futureObj(exitSignal.get\_future())  {    }  Stoppable(Stoppable && obj) : exitSignal(std::move(obj.exitSignal)), futureObj(std::move(obj.futureObj))  {  std::cout << "Move Constructor is called" << std::endl;  }    Stoppable & operator=(Stoppable && obj)  {  std::cout << "Move Assignment is called" << std::endl;  exitSignal = std::move(obj.exitSignal);  futureObj = std::move(obj.futureObj);  return \*this;  }    // Task need to provide defination  for this function  // It will be called by thread function  virtual void run() = 0;      // Thread function to be executed by thread  void operator()()  {  run();  }    //Checks if thread is requested to stop  bool stopRequested()  {  // checks if value in future object is available  if (futureObj.wait\_for(std::chrono::milliseconds(0)) == std::future\_status::timeout)  return false;  return true;  }  // Request the thread to stop by setting value in promise object  void stop()  {  exitSignal.set\_value();  }  };    /\*  \* A Task class that extends the Stoppable Task  \*/  class MyTask: public Stoppable  {  public:  // Function to be executed by thread function  void run()  {  std::cout << "Task Start" << std::endl;    // Check if thread is requested to stop ?  while (stopRequested() == false)  {  std::cout << "Doing Some Work" << std::endl;  std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));    }  std::cout << "Task End" << std::endl;  }  };    int main()  {  // Creating our Task  MyTask task;    //Creating a thread to execute our task  std::thread th([&]()  {  task.run();  });    std::this\_thread::sleep\_for(std::chrono::seconds(10));    std::cout << "Asking Task to Stop" << std::endl;  // Stop the Task  task.stop();    //Waiting for thread to join  th.join();  std::cout << "Thread Joined" << std::endl;  std::cout << "Exiting Main Function" << std::endl;  return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Task Start  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Doing Some Work  Asking Task to Stop  Task End  Thread Joined  Exiting Main Function |

# Start thread by member function with arguments

## Starting thread with non static member function

Suppose we have a class Task, which has non static member function execute() i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5 | class Task  {  public:  void execute(std::string command);  }; |

Now we want to start a thread which uses execute() function of the class Task as thread function.

As execute() is a non static function of class Task, so first of all we need a object to call this function. Let’s create an object of class Task i.e.



|  |  |
| --- | --- |
| 1 | Task \* taskPtr = new Task(); |

Now let’s create a Thread that will use this member function execute() as thread function through its object i.e.



|  |  |
| --- | --- |
| 1  2 | // Create a thread using member function  std::thread th(&Task::execute, taskPtr, "Sample Task"); |

Here in std::thread constructor we passed 3 arguments i.e.

**1.) Pointer to member function execute of class Task**  
When std::thread will internally create a new thread, it will use this passed member function as thread function. But to call a member function, we need a object.

**2.) Pointer to the object of class Task**  
As a second argument we passed a pointer to the object of class Task, with which above member function will be called. In every non static member function, first argument is always the pointer to the object of its own class. So, thread class will pass this pointer as first argument while calling the passed member function.

**3.) String value**  
This will be passed as second argument to member function i.e. after Task \*

Checkout complete example as follows,



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | #include <iostream>  #include <thread>    class Task  {  public:  void execute(std::string command)  {  for(int i = 0; i < 5; i++)  {  std::cout<<command<<" :: "<<i<<std::endl;  }  }    };    int main()  {  Task \* taskPtr = new Task();    // Create a thread using member function  std::thread th(&Task::execute, taskPtr, "Sample Task");    th.join();    delete taskPtr;  return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1  2  3  4  5 | Sample Task :: 0  Sample Task :: 1  Sample Task :: 2  Sample Task :: 3  Sample Task :: 4 |

## Starting thread with static member function

As static functions are not associated with any object of class. So, we can directly pass the static member function of class as thread function without passing any pointer to object i.e



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #include <iostream>  #include <thread>    class Task  {  public:  static void test(std::string command)  {  for(int i = 0; i < 5; i++)  {  std::cout<<command<<" :: "<<i<<std::endl;  }  }    };    int main()  {  // Create a thread using static member function  std::thread th(&Task::test, "Task");    th.join();  return 0;  } |

**Output**



|  |  |
| --- | --- |
| 1  2  3  4  5 | Task :: 0  Task :: 1  Task :: 2  Task :: 3  Task :: 4 |

# How to get a Thread ID?

Every thread has an unique Id associated with it. c++11 provides a type to store this id i.e.



|  |  |
| --- | --- |
| 1 | std::thread::id |

Objects of std::thread::id is comparable, copy-able and default implementation of std::hash() is also provided by the standard. Therefore, std::thread::id objects can be used as keys in both map and unordered\_map.

## std::thread::get\_id()

std::thread provides a member function get\_id() i.e.



|  |  |
| --- | --- |
| 1 | std::thread::id get\_id() const noexcept; |

It returns the thread id of associated object.

Let’s use this function to fetch the thread id i.e.

## Get thread ID from a Join-able Thread Object

Let’s create a thread i.e.



|  |  |
| --- | --- |
| 1  2 | // Starting Thread  std::thread th(threadFunction); |

Now get it’s thread id from thread object.



|  |  |
| --- | --- |
| 1  2 | // Fetching thread ID from Thread Object using get\_id() member function  std::thread::id threadID = th.get\_id(); |

## Get thread ID from a Detached Thread object

Let’s create a thread i.e.



|  |  |
| --- | --- |
| 1  2 | // Starting Thread  std::thread dThObj(threadFunction); |

Detach the thread from thread object i.e.



|  |  |
| --- | --- |
| 1  2 | // Detached the thread  dThObj.detach(); |

Now thread object has no associated thread with it’s id. Therefore, get\_id() on detached thread object will return default constructed value i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5 | // Fetching thread ID from Thread Object using get\_id() member function  std::thread::id dThreadID = dThObj.get\_id();    // Detached thread's get\_id() function will return default constructed thread::id only  assert(dThreadID == std::thread::id()); |

## Get current thread ID inside thread function

Inside a function, which is currently executed by some thread, we can access the current thread object through,



|  |  |
| --- | --- |
| 1 | std::this\_thread |

So, to get the current thread ID inside thread function, we can call get\_id() with this\_thread i.e.



|  |  |
| --- | --- |
| 1  2 | // Fetch the thread ID of the thread which is executing this function  std::thread::id threadID = std::this\_thread::get\_id(); |

Complete example is as follows,



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | #include <thread>  #include <iostream>  #include <assert.h>  #include <chrono>    using namespace std::chrono\_literals;    void threadFunction()  {  std::cout << "Func Start" << std::endl;    // Fetch the thread ID of the thread which is executing this function  std::thread::id threadID = std::this\_thread::get\_id();    std::cout << "Inside Thread :: Thread ID : " << threadID << "\n";  std::cout << "Func End" << std::endl;  }    int main()  {    // Starting Thread  std::thread th(threadFunction);    // Fetching thread ID from Thread Object using get\_id() member function  std::thread::id threadID = th.get\_id();    // Join the Thread if its Joinable  if (th.joinable())  th.join();    std::cout << "Thread from Main : " << threadID << std::endl;    /\*\* Fetching Thread ID from Detached Thread \*\*\*\*/    // Starting Thread  std::thread dThObj(threadFunction);    // Detached the thread  dThObj.detach();    // Fetching thread ID from Thread Object using get\_id() member function  std::thread::id dThreadID = dThObj.get\_id();    // Detached thread's get\_id() function will return default constructed thread::id only  assert(dThreadID == std::thread::id());    std::this\_thread::sleep\_for(2s);    std::cout << "Thread from Main : " << dThreadID << std::endl;  return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Func Start  Inside Thread :: Thread ID : 139924979255040  Func End  Thread from Main : 139924979255040  Func Start  Inside Thread :: Thread ID : 139924979255040  Func End  Thread from Main : thread::id of a non-executing thread |

To Compile the above example on linux use following command,

***g++ –std=c++14 example.cpp -lpthread***

# How to create Vector of Thread Objects?

## Creating & Using vector of std::thread

Let’s Create a vector of std::thread objects i.e.



|  |  |
| --- | --- |
| 1  2 | // Create a vector of threads  std::vector<std::thread> vecOfThreads; |

Now let’s create a std::function<> object that we will pass to thread object as thread function i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // Create a function object  std::function<void()> func = []() {  //Do Some Important Work  // .....  //Print Thread ID  std::cout << "From Thread ID : "<<std::this\_thread::get\_id() << "\n";  }; |

Now let’s create 2 thread objects using this std::function objects i.e.



|  |  |
| --- | --- |
| 1  2 | std::thread th1(func);  std::thread th2(func); |

Now, as std::thread objects are move only i.e. we can not copy them, only move them. Therefore, we need to move these 2 thread objects in vector i.e.



|  |  |
| --- | --- |
| 1  2  3 | // Move thread objects to vector  vecOfThreads.push\_back(std::move(th1));  vecOfThreads.push\_back(std::move(th2)); |

We can also push std::thread without specifically specifying std::move(), if we pass them as rvalue i.e.



|  |  |
| --- | --- |
| 1  2 | // Add a Thread object to vector  vecOfThreads.push\_back(std::thread(func)); |

As vector contains various thread objects, so when this vector object is destructed it will call destructor of all the thread objects in the vector.  
If any of the destructed thread object is joinable and not joined then std::terminate() will be called from its destructor. Therefore its necessary to join all the joinable threads in vector before vector is destructed i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // Iterate over the thread vector  for (std::thread & th : vecOfThreads)  {  // If thread Object is Joinable then Join that thread.  if (th.joinable())  th.join();  } |

Complete example is as follows,



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | #include <thread>  #include <vector>  #include <iostream>      int main()  {  // Create a vector of threads  std::vector<std::thread> vecOfThreads;    // Create a function object  std::function<void()> func = []() {  //Do Some Important Work  // .....  //Print Thread ID  std::cout << "From Thread ID : "<<std::this\_thread::get\_id() << "\n";  };    // Add a Thread object to vector  vecOfThreads.push\_back(std::thread(func));      // Create 3 differet thread objects  std::thread th1(func);  std::thread th2(func);  std::thread th3(func);    // Move all three thread objects to vector  vecOfThreads.push\_back(std::move(th1));  vecOfThreads.push\_back(std::move(th2));  vecOfThreads.push\_back(std::move(th3));    // Do some important work in main thread.    /\*\* Wait for all the threads in vector to join \*\*/    // Iterate over the thread vector  for (std::thread & th : vecOfThreads)  {  // If thread Object is Joinable then Join that thread.  if (th.joinable())  th.join();  }    return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1  2  3  4 | From Thread ID : 140261435352832  From Thread ID : 140261452138240  From Thread ID : 140261426960128  From Thread ID : 140261443745536 |

To compile the above example in linux use,

***g++ –std=C++11 example.cpp -lpthread***

## Use vector<std::thread> cautiously



|  |  |
| --- | --- |
| 1 | std::vector<std::thread> vecOfThreads; |

### **Move only vector of thread**

As thread objects are move only objects, therefore we can not copy vector of thread objects to an another of vector of thread i.e.



|  |  |
| --- | --- |
| 1  2 | // Can not copy vector of thread , COMPILE TIME ERROR  std::vector<std::thread> newVecThreads = vecOfThreads; |

It will not compile

Therefore, we can only move vector of thread to an another vector thread i.e.



|  |  |
| --- | --- |
| 1  2 | // Can Only move vector of thread  std::vector<std::thread> newVecThreads = std::move(vecOfThreads); |

### **Changing contents of vector of thread**

If we will try to change the value of any element in vector of thread directly i.e.



|  |  |
| --- | --- |
| 1  2 | //Destructor of already existing thread object will call terminate  vecOfThreads[1] = std::move(th4); |

It will crash our application, because on replacing a thread object inside the vector, destructor of existing thread object will be called and we haven’t joined that object yet. So, it call terminate in its destructor. So, to replace a thread object in vector, we first need to join the existing object and then replace it with new one i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // First join the existing object  if(vecOfThreads[1].joinable())  vecOfThreads[1].join();    // Replace the joined thread object  vecOfThreads[1] = std::move(th4); |

Complete example is as follows,



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | #include <thread>  #include <mutex>  #include <vector>  #include <iostream>  #include <chrono>  int main()  {  // Create a vector of threads  std::vector<std::thread> vecOfThreads;    // Create a function object  std::function < void() > func = []() {  //Do Some Important Work  // Sleep for 1 second  std::this\_thread::sleep\_for (std::chrono::seconds(1));  //Print Thread ID  std::cout << "From Thread ID : "<<std::this\_thread::get\_id() << "\n";  };    // Add a Thread object to vector  vecOfThreads.push\_back(std::thread(func));    // Create 3 differet thread objects  std::thread th1(func);  std::thread th2(func);  std::thread th3(func);    // Move all three thread objects to vector  vecOfThreads.push\_back(std::move(th1));  vecOfThreads.push\_back(std::move(th2));  vecOfThreads.push\_back(std::move(th3));    std::thread th4(func);    //Destructor of already existing thread object will call terminate  //vecOfThreads[1] = std::move(th4);    // First join the existing object  if (vecOfThreads[1].joinable())  vecOfThreads[1].join();    // Replace the joined thread object  vecOfThreads[1] = std::move(th4);    // Can not copy vector of thread , COMPILE TIME ERROR  //std::vector<std::thread> newVecThreads = vecOfThreads;    // Can Only move vector of thread  std::vector<std::thread> newVecThreads = std::move(vecOfThreads);    /\*\* Wait for all the threads in vector to join \*\*/    // Iterate over the thread vector  for (std::thread & th : newVecThreads)  {  // If thread Object is Joinable then Join that thread.  if (th.joinable())  th.join();  }    return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1  2  3  4  5 | From Thread ID : 140642994353920  From Thread ID : 140643002746624  From Thread ID : 140642985961216  From Thread ID : 140643011139328  From Thread ID : 140642977568512 |

To compile the above example in linux use,

***g++ –std=C++11 example.cpp -lpthread***

# How to use std::thread as a member variable in class?

In this article we will discuss how to use std::thread object as member variable inside class and its benefits.

As std::thread objects are move only, therefore while designing a class that use std::thread as member variable, we need to take care that objects of this class should also be move only.

## Creating Move-only class with std::thread as member variable

Let’s create a **ThreadWrapper** class that has **std::thread**as member variable and make it move-able by,

* Deleting its copy constructor and assignment operator.
* Defining Move constructor and Move assignment operator.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | /\*  \* A class that has thread object as member variable  \*/  class ThreadWrapper  {  // std::thread object  std::thread  threadHandler;    public:  //Delete the copy constructor  ThreadWrapper(const ThreadWrapper&) = delete;    //Delete the Assignment opeartor  ThreadWrapper& operator=(const ThreadWrapper&) = delete;    // Parameterized Constructor  ThreadWrapper(std::function<void()> func);    // Move Constructor  ThreadWrapper(ThreadWrapper && obj);    //Move Assignment Operator  ThreadWrapper & operator=(ThreadWrapper && obj);    //Destructor  ~ThreadWrapper();  }; |

Its parameterized constructor will accept callback / function pointer / funcion object as argument that will be used as thread function by internal thread object i.e.



|  |  |
| --- | --- |
| 1  2  3 | // Parameterized Constructor  ThreadWrapper::ThreadWrapper(std::function<void()> func) : threadHandler(func)  {} |

### Move Constructor & Assignment Operator

In Move constructor and assignment operator we need to move the thread object i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // Move Constructor  ThreadWrapper::ThreadWrapper(ThreadWrapper && obj) : threadHandler(std::move(obj.threadHandler))  {  std::cout << "Move Constructor is called" << std::endl;  }    //Move Assignment Operator  ThreadWrapper & ThreadWrapper::operator=(ThreadWrapper && obj)  {  std::cout << "Move Assignment is called" << std::endl;  if (threadHandler.joinable())  threadHandler.join();  threadHandler = std::move(obj.threadHandler);  return \*this;  } |

In Move assignment operator we first need to join the current thread  object if its joinable, before replacing it with new thread object.

In the destructor of ThreadWrapper we need to join the thread object. It is necessary because if thread object is destructed without joining, then it will terminate the application i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Destructor : Join the thread object  ThreadWrapper::~ThreadWrapper()  {  if (threadHandler.joinable())  threadHandler.join();  } |

Now Let’s create a ThreadWrapper object, now when this object will be destructed, internal thread will be joined in destructor i.e



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // Creating a std::function object  std::function<void()> func = []() {  // Sleep for 1 second  std::this\_thread::sleep\_for (std::chrono::seconds(1));  // Print thread ID  std::cout << "From Thread ID : " << std::this\_thread::get\_id() << "\n";  };    {  // Create a ThreadWrapper object  // It will internally start the thread  ThreadWrapper wrapper(func);    //When wrapper will go out of scope, its destructor will be called  // Which will internally join the member thread object  } |

Also, we can create a vector of ThreadWraper i.e.



|  |  |
| --- | --- |
| 1  2 | // Create a vector of ThreadWrapper objects  std::vector<ThreadWrapper> vecOfThreads; |



|  |  |
| --- | --- |
| 1  2  3  4  5 | // Add ThreadWrapper objects in thread  ThreadWrapper thwp1(func);  ThreadWrapper thwp2(func);  vecOfThreads.push\_back(std::move(thwp1));  vecOfThreads.push\_back(std::move(thwp2)); |

We don’t even need to join the threads separately , it will be automatically joined when vector is destructed. We can also change the content of vector i.e.



|  |  |
| --- | --- |
| 1  2  3  4 | ThreadWrapper thwp3(func);    // Change the content of vector  vecOfThreads[1] = std::move(thwp3); |

Complete example is as follows,



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103 | #include <thread>  #include <mutex>  #include <vector>  #include <iostream>  #include <assert.h>  #include <chrono>    /\*  \* A class that has thread object as member variable  \*/  class ThreadWrapper  {  // std::thread object  std::thread  threadHandler;    public:  //Delete the copy constructor  ThreadWrapper(const ThreadWrapper&) = delete;    //Delete the Assignment opeartor  ThreadWrapper& operator=(const ThreadWrapper&) = delete;    // Parameterized Constructor  ThreadWrapper(std::function<void()> func);    // Move Constructor  ThreadWrapper(ThreadWrapper && obj);    //Move Assignment Operator  ThreadWrapper & operator=(ThreadWrapper && obj);    //Destructor  ~ThreadWrapper();  };    // Parameterized Constructor  ThreadWrapper::ThreadWrapper(std::function<void()> func) : threadHandler(func)  {}    // Move Constructor  ThreadWrapper::ThreadWrapper(ThreadWrapper && obj) : threadHandler(std::move(obj.threadHandler))  {  std::cout << "Move Constructor is called" << std::endl;  }    //Move Assignment Operator  ThreadWrapper & ThreadWrapper::operator=(ThreadWrapper && obj)  {  std::cout << "Move Assignment is called" << std::endl;  if (threadHandler.joinable())  threadHandler.join();  threadHandler = std::move(obj.threadHandler);  return \*this;  }    // Destructor  ThreadWrapper::~ThreadWrapper()  {  if (threadHandler.joinable())  threadHandler.join();  }    int main()  {  // Creating a std::function object  std::function<void()> func = []() {  // Sleep for 1 second  std::this\_thread::sleep\_for (std::chrono::seconds(1));  // Print thread ID  std::cout << "From Thread ID : " << std::this\_thread::get\_id() << "\n";  };    {  // Create a ThreadWrapper object  // It will internally start the thread  ThreadWrapper wrapper(func);    //When wrapper will go out of scope, its destructor will be called  // Which will internally join the member thread object  }    // Create a vector of ThreadWrapper objects  std::vector<ThreadWrapper> vecOfThreads;      // Add ThreadWrapper objects in thread  ThreadWrapper thwp1(func);  ThreadWrapper thwp2(func);  vecOfThreads.push\_back(std::move(thwp1));  vecOfThreads.push\_back(std::move(thwp2));    ThreadWrapper thwp3(func);    // Change the content of vector  vecOfThreads[1] = std::move(thwp3);      //When vector will go out of scope, its destructor will be called, which will  // internally call the destructor all ThreadWrapper objects , which in turn  // joins the member thread object.    return 0;  } |

**Output:**

Vim



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | From Thread ID : 140646533629696  Move Constructor is called  Move Constructor is called  Move Constructor is called  Move Assignment is called  From Thread ID : 140646533629696  From Thread ID : 140646525236992  From Thread ID : 140646516844288 |

# How to put a thread to sleep in c++11 ? | sleep\_for | sleep\_until

In this article we will discuss how to put a c++11 thread to sleep.

c++11 provides 2 functions for putting a thread to sleep i.e.



|  |  |
| --- | --- |
| 1  2 | std::this\_thread::sleep\_for  std::this\_thread::sleep\_untill |

## Sleep for a Duration

C++11 provides a function std::this\_thread::sleep\_for to block the current thread for specified duration i.e.



|  |  |
| --- | --- |
| 1  2 | template <class Rep, class Period>  void sleep\_for (const chrono::duration<Rep,Period>& rel\_time); |

This function accepts a duration as an argument and make the calling thread to sleep for that particular duration.

This duration can be from nanoseconds to hours i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | std::chrono::nanoseconds  std::chrono::microseconds  std::chrono::milliseconds  std::chrono::seconds  std::chrono::minutes  std::chrono::hours |

Lets see some examples,

**Sleeping a Thread for MilliSeconds:**

To sleep a thread for 200 Milliseconds call sleep\_for with following argument i.e.



|  |  |
| --- | --- |
| 1 | std::this\_thread::sleep\_for(std::chrono::milliseconds(200)); |

**Sleeping a Thread for Minutes:**

To sleep a thread for 1 Minute call sleep\_for with following argument i.e.



|  |  |
| --- | --- |
| 1 | std::this\_thread::sleep\_for(std::chrono::minutes(1)); |

Checkout complete example as follows,



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | #include <iostream>  #include <thread>  #include <chrono>    void threadFunc()  {  int i = 0;  while (i < 10)  {  // Print Thread ID and Counter i  std::cout<<std::this\_thread::get\_id()<<" :: "<<i++<<std::endl;    // Sleep this thread for 200 MilliSeconds  std::this\_thread::sleep\_for(std::chrono::milliseconds(200));  }  }    int main()  {  std::thread th(&threadFunc);  th.join();  return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 140484807997184 :: 0  140484807997184 :: 1  140484807997184 :: 2  140484807997184 :: 3  140484807997184 :: 4  140484807997184 :: 5  140484807997184 :: 6  140484807997184 :: 7  140484807997184 :: 8  140484807997184 :: 9 |

## Sleep Until a TimePoint

Many times we want the thread to sleep untill a time point in future. That can be acieved using sleep\_untill() i.e.



|  |  |
| --- | --- |
| 1  2 | template< class Clock, class Duration >  void sleep\_until( const std::chrono::time\_point<Clock,Duration>& sleepTime ); |

It accepts a time point as an argument and blocks the current thread till this time point is achieved.

Checkout the complete example, here we will put a thread to sleep until a time point in future i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #include <iostream>  #include <thread>  #include <chrono>    // Print Current Time  void print\_time\_point(std::chrono::system\_clock::time\_point timePoint)  {  std::time\_t timeStamp = std::chrono::system\_clock::to\_time\_t(timePoint);  std::cout << std::ctime(&timeStamp) << std::endl;    }    void threadFunc()  {    std::cout<<"Current Time :: ";  // Print Current Time  print\_time\_point(std::chrono::system\_clock::now());    // create a time point pointing to 10 second in future  std::chrono::system\_clock::time\_point timePoint =  std::chrono::system\_clock::now() + std::chrono::seconds(10);    std::cout << "Going to Sleep Until :: "; print\_time\_point(timePoint);      // Sleep Till specified time point  // Accepts std::chrono::system\_clock::time\_point as argument  std::this\_thread::sleep\_until(timePoint);    std::cout<<"Current Time :: ";  // Print Current Time  print\_time\_point(std::chrono::system\_clock::now());  }    int main()  {  std::thread th(&threadFunc);  th.join();  return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1  2  3 | Current Time :: Sat Feb 25 16:44:40 2017  Going to Sleep Until :: Sat Feb 25 16:44:50 2017  Current Time :: Sat Feb 25 16:44:50 2017 |