<https://thispointer.com/>

<http://www.bogotobogo.com/cplusplus/multithreaded4_cplusplus11.php>

<http://www.bogotobogo.com/cplusplus/multithreaded4_cplusplus11B.php>

<http://www.bogotobogo.com/cplusplus/multithreadedDebugging.php>

# C++11 Multithreading – Part 1 : Three Different ways to Create Threads

**Introduction to C++11 Thread Library**

Original C++ Standard supported only single thread programming. The new C++ Standard (referred to as C++11 or C++0x) was published in 2011. In C++11 a new thread library is introduced.

**Compilers Required:**  
**Linux:** gcc 4.8.1 (Complete Concurrency support)  
**Windows:** Visual Studio 2012 and MingW

**How to compile on Linux:** g++ –std=c++11 sample.cpp -lpthread

**Thread Creation in C++11**

In every C++ application there is one default main thread i.e. main() function. In C++ 11 we can create additional threads by creating objects of std::thread class.  
Each of the std::thread object can be associated with a thread.

Header Required : <thread>

What this std::thread object will execute ?

We can attach a callback with the std::thread object, that will be executed when this new thread starts. These callbacks can be,

1.) Function Pointer  
2.) Function Objects  
3.) Lambda functions

Thread objects can be created like this,

*std::thread thObj(<CALLBACK>);*

New Thread will start just after the creation of new object and will execute the passed callback in parallel to thread that has started it.  
Moreover, any thread can wait for another to exit by calling join() function on that thread’s object.

Lets look at an example where main thread will create a separate thread. After creating this new thread, main thread will print some data on console and then wait for newly created thread to exit.

Lets implement above using three different callback mechanism,

**Creating a thread using Function Pointer,**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <iostream>  #include <thread>    void thread\_function()  {      for(int i = 0; i < 10000; i++);          std::cout<<"thread function Executing"<<std::endl;  }    int main()  {        std::thread threadObj(thread\_function);      for(int i = 0; i < 10000; i++);          std::cout<<"Display From MainThread"<<std::endl;      threadObj.join();      std::cout<<"Exit of Main function"<<std::endl;      return 0;  } |

**Creating a thread using Function Objects,**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #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;  } |

**Creating a thread using Lambda functions,**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <iostream>  #include <thread>  int main()  {      int x = 9;      std::thread threadObj([]{              for(int i = 0; i < 10000; i++)                  std::cout<<"Display Thread Executing"<<std::endl;              });        for(int i = 0; i < 10000; 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,  
*std::thread::get\_id()*  
Gives the identifier for the current thread,  
*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.  
Lets look at an example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #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 and Detaching Threads

**Joining Threads:**

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

*std::thread th(funcPtr);*  
*th.join();*

Lets 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,

|  |  |
| --- | --- |
| 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 | #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;  } |

**Detaching Threads:**

Detached threads are also called daemon / Background threads.  To detach a thread we need to call std::detach() function on std::thread object i.e.

|  |  |
| --- | --- |
| 1  2 | std::thread th(funcPtr);  th.detach(); |

After calling detach(), std::thread object is no longer associated with the actual thread of execution.

**Be careful with calling detach() and join() on Thread Handles**

**Case 1: *Never call join() or detach() on std::thread object with no associated executing thread***

|  |  |
| --- | --- |
| 1  2  3 | std::thread threadObj( (WorkerThread()) );      threadObj.join();      threadObj.join(); // It will cause Program to Terminate |

When a join() function is called on an thread object, then when this join(0 returns then that std::thread object has no associated thread with it. In case again join() function is called on such object then it will cause the program to Terminate.

Similarly calling detach() makes the std::thread object not linked with any thread function. In that case calling detach(0 function twice on an std::thread object will cause the program to terminate.

|  |  |
| --- | --- |
| 1  2  3 | std::thread threadObj( (WorkerThread()) );      threadObj.detach();      threadObj.detach(); // It will cause Program to Terminate |

Therefore, before calling join() or detach() we should check if thread is join-able every time 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 | std::thread threadObj( (WorkerThread()) );      if(threadObj.joinable())      {          std::cout<<"Detaching Thread "<<std::endl;          threadObj.detach();      }      if(threadObj.joinable())      {          std::cout<<"Detaching Thread "<<std::endl;          threadObj.detach();      }        std::thread threadObj2( (WorkerThread()) );      if(threadObj2.joinable())      {          std::cout<<"Joining Thread "<<std::endl;          threadObj2.join();      }      if(threadObj2.joinable())      {          std::cout<<"Joining Thread "<<std::endl;          threadObj2.join();      } |

**Case 2*: Never forget to call either join or detach on a std::thread object with associated executing thread***  
  
If  neither join or detach is called with a std::thread object that has associated executing thread then during that object’s destruct-or it will terminate the program.  
Because inside the destruct-or it checks if Thread is Still Join-able then Terminate the program i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #include <iostream>  #include <thread>  #include <algorithm>  class WorkerThread  {  public:      void operator()()      {          std::cout<<"Worker Thread "<<std::endl;      }  };  int main()  {      std::thread threadObj( (WorkerThread()) );      // Program will terminate as we have't called either join or detach with the std::thread object.      // Hence std::thread's object destructor will terminate the program      return 0;  } |

Similarly we should not forget call either join() or detach() in case of exceptions. To prevents with we should use RESOURCE ACQUISITION IS INITIALIZATION (RAII) 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 | #include <iostream>  #include <thread>  class ThreadRAII  {      std::thread & m\_thread;      public:          ThreadRAII(std::thread  & threadObj) : m\_thread(threadObj)          {            }          ~ThreadRAII()          {              // Check if thread is joinable then detach the thread              if(m\_thread.joinable())              {                  m\_thread.detach();              }          }  };  void thread\_function()  {      for(int i = 0; i < 10000; i++);          std::cout<<"thread\_function Executing"<<std::endl;  }    int main()  {      std::thread threadObj(thread\_function);        // If we comment this Line, then program will crash      ThreadRAII wrapperObj(threadObj);      return 0;  } |

Resource Acquisition Is Initialization (RAII)

**Resource Acquisition Is Initialization (RAII)** provides a class that does the **join()** in its destructor, as shown in the following code.

#include <iostream>

#include <thread>

using namespace std;

void foo(int i)

{

cout << "foo(" << i << ")\n";

}

struct task\_struct

{

int& i;

task\_struct(int& ii):i(ii){ "task\_struct constructor\n";}

void operator()()

{

cout << "task\_struct::operator()\n";

for(unsigned j = 0; j < 10; ++j)

{

foo(i);

}

}

};

class thread\_RAII

{

thread& t;

public:

thread\_RAII(thread& th):t(th)

{

cout << "thread\_RAII constructor\n";

}

~thread\_RAII()

{

if(t.joinable())

{

cout << "if joinable(), then t.join()\n";

t.join();

}

cout << "thread\_RAII destructor\n";

}

private:

// copy constructor

thread\_RAII(const thread\_RAII& thr) ;

// copy-assignment operator

thread\_RAII& operator=(const thread\_RAII& thr);

};

void current\_thread\_task()

{

cout << "do something in current\_thread\_task()\n";

};

void A\_function\_creating\_a\_thread\_within()

{

int state = 99;

task\_struct task(state);

cout << "launching a thread\n";

thread t(task);

cout << "make an instance of thead\_RAII\n";

thread\_RAII raii(t);

cout << "call current\_thread\_task()\n";

current\_thread\_task();

}

int main()

{

A\_function\_creating\_a\_thread\_within();

return 0;

}

The output can be random:

launching a thread

task\_struct::operator()

foo(make an instance of thead\_RAII

thread\_RAII constructor

call current\_thread\_task()

do something in current\_thread\_task()

if joinable(), then t.join()

99)

foo(99)

foo(99)

foo(99)

foo(99)

foo(99)

foo(99)

foo(99)

foo(99)

foo(99)

thread\_RAII destructor

When the execution of the current thread reaches the end of **A\_function\_creating\_a\_thread\_within()** , the local objects are destroyed in reverse order of construction. In other words, the **thread\_RAII** object **raiiObj** is destroyed first. The next one to be destroyed is **t**.

The destructor of **thread\_RAII** first tests to see if the **std::thread** object is **joinable()** before calling **join()**. This is important, because **join()** can be called only once for a given thread of execution, so it would therefore be a mistake to do so if the thread had already been joined.

# Carefully Pass Arguments to Threads

To Pass arguments to thread’s associated callable object or function just pass additional arguments to the std::thread constructor.  
By default all arguments are copied into the internal storage of new thread.

Lets look at an example

**Passing simple arguments to a thread**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include <iostream>  #include <string>  #include <thread>  void threadCallback(int x, std::string str)  {      std::cout<<"Passed Number = "<<x<<std::endl;      std::cout<<"Passed String = "<<str<<std::endl;  }  int main()  {      int x = 10;      std::string str = "Sample String";      std::thread threadObj(threadCallback, x, str);      threadObj.join();      return 0;  } |

**How not to pass arguments to threads:**

Don’t pass addresses of variables from local stack to thread’s callback function. Because it might be possible that local variable in Thread 1 goes out of scope but Thread 2 is still trying to access it through it’s address.  
In such scenario accessing invalid address can cause unexpected behaviour.  
For example,

|  |  |
| --- | --- |
| 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>  void newThreadCallback(int \* p)  {      std::cout<<"Inside Thread :  "" : p = "<<p<<std::endl;      std::chrono::milliseconds dura( 1000 );      std::this\_thread::sleep\_for( dura );      \*p = 19;  }  void startNewThread()  {      int i = 10;      std::cout<<"Inside Main Thread :  "" : i = "<<i<<std::endl;      std::thread t(newThreadCallback,&i);      t.detach();      std::cout<<"Inside Main Thread :  "" : i = "<<i<<std::endl;  }  int main()  {      startNewThread();      std::chrono::milliseconds dura( 2000 );      std::this\_thread::sleep\_for( dura );      return 0;  } |

Similarly be careful while passing pointer to memory located on heap to thread. Because it might be possible that some thread deletes that memory before new thread tries to access it.  
In such scenario accessing invalid address can cause unexpected behaviour.  
For example,

|  |  |
| --- | --- |
| 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 | #include <iostream>  #include <thread>  void newThreadCallback(int \* p)  {      std::cout<<"Inside Thread :  "" : p = "<<p<<std::endl;      std::chrono::milliseconds dura( 1000 );      std::this\_thread::sleep\_for( dura );      \*p = 19;  }  void startNewThread()  {      int \* p = new int();      \*p = 10;      std::cout<<"Inside Main Thread :  "" : \*p = "<<\*p<<std::endl;      std::thread t(newThreadCallback,p);      t.detach();      delete p;      p = NULL;  }  int main()  {      startNewThread();      std::chrono::milliseconds dura( 2000 );      std::this\_thread::sleep\_for( dura );      return 0;  } |

**How to pass references to thread**:

As arguments are copied to new threads stack so, if you need to pass references in common way i.e.

Check this,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <iostream>  #include <thread>  void threadCallback(int const & x)  {      int & y = const\_cast<int &>(x);      y++;      std::cout<<"Inside Thread x = "<<x<<std::endl;  }  int main()  {      int x = 9;      std::cout<<"In Main Thread : Before Thread Start x = "<<x<<std::endl;      std::thread threadObj(threadCallback, x);      threadObj.join();      std::cout<<"In Main Thread : After Thread Joins x = "<<x<<std::endl;      return 0;  } |

Its output is

*In Main Thread : Before Thread Start x = 9*  
*Inside Thread x = 10*  
*In Main Thread : After Thread Joins x = 9*

Even if threadCallback accepts arguments as reference but still changes done it are not visible outside the thread.  
Its because x in the thread function threadCallback is reference to the temporary value copied at the new thread’s stack.

How to fix this ?

Using std::ref() i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <iostream>  #include <thread>  void threadCallback(int const & x)  {      int & y = const\_cast<int &>(x);      y++;      std::cout<<"Inside Thread x = "<<x<<std::endl;  }  int main()  {      int x = 9;      std::cout<<"In Main Thread : Before Thread Start x = "<<x<<std::endl;      std::thread threadObj(threadCallback,std::ref(x));      threadObj.join();      std::cout<<"In Main Thread : After Thread Joins x = "<<x<<std::endl;      return 0;  } |

Its output is

*In Main Thread : Before Thread Start x = 9*  
*Inside Thread x = 10*  
*In Main Thread : After Thread Joins x = 10*  
  
**Assigning pointer to member function of a class as thread function:**

Pass the pointer to member function as callback function and pass pointer to Object as second argument.

For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #include <iostream>  #include <thread>  class DummyClass {  public:      DummyClass()      {}      DummyClass(const DummyClass & obj)      {}      void sampleMemberFunction(int x)      {          std::cout<<"Inside sampleMemberFunction "<<x<<std::endl;      }  };  int main() {        DummyClass dummyObj;      int x = 10;      std::thread threadObj(&DummyClass::sampleMemberFunction,&dummyObj, x);      threadObj.join();      return 0;  } |

# Data Sharing and Race Conditions

In multithreading environment data sharing between threads is very easy. But this easy sharing of data can cause problems in application. One such problem is *Race Condition*.

**What is a Race Condition?**

Race condition is a kind of a bug that occurs in multithreaded applications.

When two or more threads perform a set of operations in parallel, that access the same memory location.  Also, one or more thread out of them modifies the data in that memory location, then this can lead to an unexpected results some times.

This is called race condition.

Race conditions are usually hard to find and reproduce because they don’t occur every time. They will occur only when relative order of execution of operations by two or more threads leads to an unexpected result. Let’s understand by an example,

**A Practical example of Race Condition:**

Let’s Create a Wallet class that internally maintains money and provides a service/function i.e. addMoney(). This member function increments the internal money of wallet object by specified count.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Wallet  {      int mMoney;  public:      Wallet() :mMoney(0){}      int getMoney() { return mMoney; }      void addMoney(int money)      {         for(int i = 0; i < money; ++i)         {            mMoney++;         }      }  }; |

Now Let’s create 5 threads and all these threads will share a same object of class Wallet and add 1000 to internal money using it’s addMoney() member function in parallel.

So, if initially money in wallet is 0. Then after completion of all thread’s execution money in Wallet should be 5000.

But, as all threads are modifying the shared data at same time, it might be possible that in some scenarios money in wallet at end will be much lesser than 5000.

Let’s test this,

|  |  |
| --- | --- |
| 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 | int testMultithreadedWallet()  {     Wallet walletObject;     std::vector<std::thread> threads;     for(int i = 0; i < 5; ++i){          threads.push\_back(std::thread(&Wallet::addMoney, &walletObject, 1000));     }       for(int i = 0; i < threads.size() ; i++)     {         threads.at(i).join();     }     return walletObject.getMoney();  }    int main()  {      int val = 0;    for(int k = 0; k < 1000; k++)    {       if((val = testMultithreadedWallet()) != 5000)       {         std::cout << "Error at count = "<<k<<" Money in Wallet = "<<val << std::endl;       }    }    return 0;  } |

As addMoney() member function of same Wallet class object is executed 5 times hence it’s internal money is expected to be 5000. But as addMoney() member function is executed in parallel hence in some scenarios mMoney will be much lesser than 5000 i.e.

**Output is,**

***Error at count = 971  Money in Wallet = 4568                                                                                                               Error at count = 971  Money in Wallet = 4568                                                                                                               Error at count = 972  Money in Wallet = 4260                                                                                                               Error at count = 972  Money in Wallet = 4260                                                                                                               Error at count = 973  Money in Wallet = 4976                                                                                                               Error at count = 973  Money in Wallet = 4976***

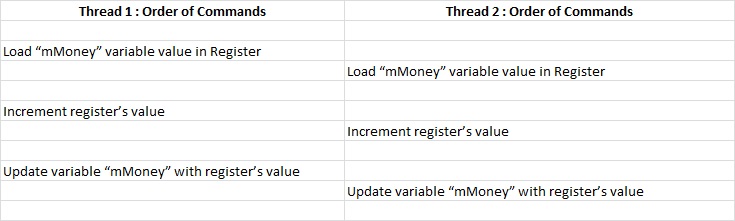
This is a race condition, as here two or more threads were trying to modify the same memory location at same time and lead to unexpected result.

**Why this happened?**

Each thread increments the same “mMoney” member variable in parallel. Although it seems a single line but this “mMoney++” is actually converted into three machine commands,

* Load “mMoney” variable value in Register
* Increment register’s value
* Update variable “mMoney” with register’s value

Now suppose in a special scenario, order of execution of above these commands is as follows,

[](https://thispointer.com/wp-content/uploads/2015/02/thread_order1.jpg)Order of Executions Of Commands

In this scenario one increment will get neglected because instead of incrementing the “mMoney” variable twice, different registers got incremented and “mMoney” variable’s value was overwritten.

Suppose prior to this scenario mMoney was 46 and as seen in above image it is incremented 2 times, so expected result is 48. But due to race condition in above scenario final value of mMoney will be 47 only.

This is called a Race Condition.

**How to fix Race Conditions?**

To fix this problem we need to use Lock mechanism i.e. each thread need to acquire a lock before modifying or reading the shared data and after modifying the data each thread should unlock the Lock.

# Using mutex to fix Race Conditions

Here we will discuss how to use mutex locks to protect shared data in multithreaded environment and avoid race conditions.

To fix race conditions in multi-threaded environment we need mutex i.e. each thread needs to lock a mutex before modifying or reading the shared data and after modifying the data each thread should unlock the mutex.

In the C++11 threading library, the mutexes are in the <mutex> header file. The class representing a mutex is the std::mutex class.

There are two important methods of mutex:  
1.) lock()  
2.) unlock()

Here we will see how to use mutex to fix the race condition in that multithreaded wallet.

As, Wallet provides a service to add money in Wallet and same Wallet object is used between different threads, so we need to add Lock in addMoney() method of the Wallet i.e.  
Acquire lock before incrementing the money of Wallet and release lock before leaving that function. Let’s see the code,

Wallet class that internally maintains money and provides a service/function i.e. addMoney().  
This member function, first acquires a lock then increments the internal money of wallet object by specified count and then releases the lock.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #include<iostream>  #include<thread>  #include<vector>  #include<mutex>    class Wallet  {  int mMoney;  std::mutex mutex;  public:  Wallet() :mMoney(0){}      int getMoney()   { return mMoney; }      void addMoney(int money)      {  mutex.lock();       for(int i = 0; i < money; ++i)  {  mMoney++;  }  mutex.unlock();      }  }; |

Now Let’s create 5 threads and all these threads will share a same object of class Wallet and add 1000 to internal money using it’s addMoney() member function in parallel.

So, if initially money in wallet is 0. Then after completion of all thread’s execution money in Wallet should be 5000.

And this mutex lock guarantees that Money in the Wallet will be 5000 at end.

Let’s test this,

|  |  |
| --- | --- |
| 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 | int testMultithreadedWallet()  {      Wallet walletObject;      std::vector<std::thread> threads;      for(int i = 0; i < 5; ++i){          threads.push\_back(std::thread(&Wallet::addMoney, &walletObject, 1000));      }        for(int i = 0; i < threads.size() ; i++)      {          threads.at(i).join();      }      return walletObject.getMoney();  }    int main()  {    int val = 0;  for(int k = 0; k < 1000; k++)  {  if((val = testMultithreadedWallet()) != 5000)  {  std::cout << "Error at count = "<<k<<"  Money in Wallet = "<<val << std::endl;  //break;  }  }  return 0;  } |

It’s guaranteed that it will not found a single scenario where money in wallet is less than 5000.  
Because mutex lock in addMoney makes sure that once one thread finishes the modification of money then only any other thread modifies the money in Wallet.

But what if we forgot to unlock the mutex at the end of function. In such scenario, one thread will exit without releasing the lock and other threads will remain in waiting.  
This kind of scenario can happen in case some **exception** came after locking the mutex. To avoid such scenarios we should use std::lock\_guard.

**std::lock\_guard :**

std::lock\_guard is a class template, which implements the RAII for mutex.  
It wraps the mutex inside it’s object and locks the attached mutex in its constructor. When it’s destructor is called it releases the mutex.

Let’s see the code,

|  |  |
| --- | --- |
| 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 | class Wallet  {  int mMoney;  std::mutex mutex;  public:      Wallet() :mMoney(0){}      int getMoney()   { return mMoney; }      void addMoney(int money)      {  std::lock\_guard<std::mutex> lockGuard(mutex);  // In constructor it locks the mutex         for(int i = 0; i < money; ++i)  {  // If some exception occurs at this  // poin then destructor of lockGuard  // will be called due to stack unwinding.  //  mMoney++;  }  // Once function exits, then destructor  // of lockGuard Object will be called.  // In destructor it unlocks the mutex.      }  }; |

# Need of Event Handling

In this article we will discuss the need of Event Handling in Multi-threading.

Sometimes a thread needs to wait for an event to happen like a condition to become true or a task to be completed by another thread.

For example,

Suppose we are building a network based application. This application does following tasks,

1. Perform some handshaking with server
2. Load Data from XML files.
3. Do processing on data loaded from XML.

As we can see that Task 1 is not dependent on any other Tasks but Task 3 is dependent on Task 2. So, it means Task 1 and Task 2 can be run in parallel by different Threads to improve the performance of application.

So, let’s break this into a multi-threaded application,

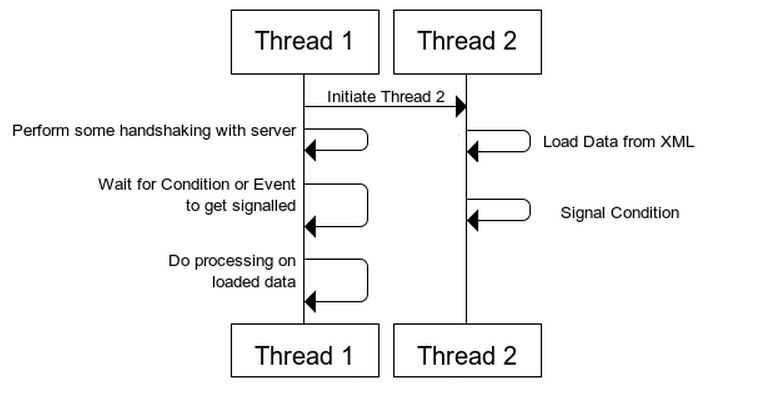
Now, it includes two threads,

Responsibilities of Thread 1 are,

* Perform some handshaking with server.
* Wait for data to be loaded from XML by Thread 2
* Do processing on data loaded from XML.

Responsibilities of Thread 2 are,

* Load data from XML
* Notify another Thread i.e. waiting for the message.

[](https://thispointer.com/wp-content/uploads/2015/06/first.png)

In above, Thread 1 performs some operations and then waits for an event/condition to happen. The event or condition here is,

*Is Data loaded successfully.*

Once Thread 1 receives that Event then it performs some processing on the data.

Thread 2, loads the data in parallel when Thread 1 was busy in doing Hand Shake Mechanism.

When Thread 2 successfully loads the data from XML, it then notifies the Thread 1 by signaling that Event.

Now When Event or Condition is signaled then Thread 1 will continue the processing of Data.

What’s the benefit of making it multi-threaded?

When Thread 1 is busy in some handshaking mechanism then Thread 2 will load the data parallel from XML. So, it will increase the performance of application.

Now, how to achieve this,

**Option 1:**

Make a Boolean global variable with default value false. Set its value to true in Thread 2 and Thread 1 will keep on checking its value in loop and as soon as it becomes true Thread 1 will continue with processing of data. But as it’s a global variable shared by both of the Threads it needs synchronization with mutex. Let’s see its code,

|  |  |
| --- | --- |
| 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 | #include<iostream>  #include<thread>  #include<mutex>    class Application  {  std::mutex m\_mutex;  bool m\_bDataLoaded;  public:  Application()  {  m\_bDataLoaded = false;  }  void loadData()  {  // Make This Thread sleep for 1 Second  std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));  std::cout<<"Loading Data from XML"<<std::endl;    // Lock The Data structure  std::lock\_guard<std::mutex> guard(m\_mutex);    // Set the flag to true, means data is loaded  m\_bDataLoaded = true;    }  void mainTask()  {  std::cout<<"Do Some Handshaking"<<std::endl;    // Acquire the Lock  m\_mutex.lock();  // Check if flag is set to true or not  while(m\_bDataLoaded != true)  {    // Release the lock    m\_mutex.unlock();    //sleep for 100 milli seconds    std::this\_thread::sleep\_for(std::chrono::milliseconds(100));    // Acquire the lock    m\_mutex.lock();    }    // Release the lock    m\_mutex.unlock();    //Doc processing on loaded Data    std::cout<<"Do Processing On loaded Data"<<std::endl;  }  };    int main()  {    Application app;      std::thread thread\_1(&Application::mainTask, &app);    std::thread thread\_2(&Application::loadData, &app);      thread\_2.join();    thread\_1.join();    return 0;  } |

Output is as follows,

[Output](https://thispointer.com/wp-content/uploads/2015/06/second_output.png)

**It has following disadvantages,**

Thread will keep on acquiring the lock and release it just to check the value, therefore it will consume CPU cycles and will also make Thread 1 slow, because it needs to acquire same lock to update the bool flag.

So obviously we need a better mechanism to achieve this, like if somehow Thread 1 could just block by waiting for an Event to get signaled and another Thread could signal that Event and make Thread 1 continue. It would have save many CPU cycles and gave better performance. But the question is how to achieve this?

We will see the answer in Option 2.

**Option 2:**

We can achieve this using *Condition Variables*.

*Condition Variable* is a kind Event used for signaling between 2 threads. One thread can wait for it to get signaled, while other thread can signal this.

Check out next article for detailed explanation of Conditional Variable in this Multi-threading series and solution to this problem using conditional variable.

# Condition Variables Explained

**Condition Variables**

Condition Variable is a kind of Event used for signaling between two or more threads. One or more thread can wait on it to get signaled, while an another thread can signal this.

Header file required for condition Variable in C++11 is

|  |  |
| --- | --- |
| 1 | #include <condition\_variable> |

A mutex is required along with condition variable.

**How things actually work with condition variable,**

* Thread 1 calls the wait on condition variable, which internally acquires the mutex and check if required condition is met or not.
* If not then it releases the lock and waits for Condition Variable to get signaled ( thread gets blocked). Condition Variable’s wait() function provides both these operations in atomic manner.
* Another Thread i.e. like Thread 2 signals the Condition Variable when condition is met
* Once Conditional Variable get signaled the the Thread 1 which was waiting for it resumes. It then acquires the mutex lock again and checks if the condition associated with Condition Variable is actually met or if it is superiors call. If more than one thread was waiting then notify\_one will unblock only one thread.
* If it was a superiors call then it again calls the wait() function.

**Main member functions for condition variable are,**

**Wait()**

It makes the current thread to block until the condition variable get signaled or a spurious wake up happens.

It atomically releases the attached mutex, blocks the current thread, and adds it to the list of threads waiting on the current condition variable object. The thread will be unblocked when some thread calls notify\_one() or notify\_all() on same condition variable object. It may also be unblocked spuriously, therefore after every unblocking it needs to check condition again.

A callback is passed as an parameter to this function, which will be called to check if it is a spurious call or actually condition is met.

When threads get unlocked,

wait() function reacquires the mutex lock and checks that actually condition is met or not. If condition is not met then again it atomically releases the attached mutex, blocks the current thread, and adds it to the list of threads waiting on the current condition variable object.

**notify\_one()**

If any threads are waiting on same conditional variable object then  notify\_one unblocks one of the waiting threads.

**notify\_all()**

If any threads are waiting on same conditional variable object then  notify\_all unblocks all of the waiting threads.

Let’s see how we can handle previously discussed multi threaded scenario with condition variable i.e.

[**Need of Event Handling in Multi-threading**](https://thispointer.com/c11-multithreading-part-6-need-of-event-handling/)**(**In this Article we handled the problem without Event Handling)

**Problem Scenario**

Suppose we are building a network based application. This application does following tasks,

1. Perform some handshaking with server
2. Load Data from XML files.
3. Do processing on data loaded from XML.

As we can see that Task 1 is not dependent on any other Tasks but Task 3 is dependent on Task 2. So, it means Task 1 and Task 2 can be run in parallel by different Threads to improve the performance of application. So, let’s break this into a multi threaded application,

Responsibilities of Thread 1 are,

* Perform some handshaking with server.
* Wait for data to be loaded from XML by Thread 2
* Do processing on data loaded from XML.

Responsibilities of Thread 2 are,

* Load data from XML
* Notify another Thread i.e. waiting for the message.

Code to achieve this with condition variable 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 | #include <iostream>  #include <thread>  #include <functional>  #include <mutex>  #include <condition\_variable>  using namespace std::placeholders;  class Application  {    std::mutex m\_mutex;    std::condition\_variable m\_condVar;    bool m\_bDataLoaded;  public:    Application()    {      m\_bDataLoaded = false;    }    void loadData()    {     // Make This Thread sleep for 1 Second     std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));     std::cout<<"Loading Data from XML"<<std::endl;     // Lock The Data structure     std::lock\_guard<std::mutex> guard(m\_mutex);     // Set the flag to true, means data is loaded     m\_bDataLoaded = true;     // Notify the condition variable     m\_condVar.notify\_one();    }    bool isDataLoaded()    {      return m\_bDataLoaded;    }    void mainTask()    {      std::cout<<"Do Some Handshaking"<<std::endl;      // Acquire the lock      std::unique\_lock<std::mutex> mlock(m\_mutex);      // Start waiting for the Condition Variable to get signaled      // Wait() will internally release the lock and make the thread to block      // As soon as condition variable get signaled, resume the thread and      // again acquire the lock. Then check if condition is met or not      // If condition is met then continue else again go in wait.      m\_condVar.wait(mlock, std::bind(&Application::isDataLoaded, this));      std::cout<<"Do Processing On loaded Data"<<std::endl;    }  };  int main()  {     Application app;     std::thread thread\_1(&Application::mainTask, &app);     std::thread thread\_2(&Application::loadData, &app);     thread\_2.join();     thread\_1.join();     return 0;  } |

# std::future , std::promise and Returning values from Thread

A std::future object can be used with asych, std::packaged\_task and std::promise. In this article will mainly focus on using std::future with std::promise object.

Many times we encounter a situation where we want a thread to return a result.

Now question is how to do this?

Lets take an example,

Suppose in our application we created a thread that will compress a given folder and we want this thread to return the new zip file name and its size in result.

Now to do this we have 2 ways,

**1.) Old Way : Share data among threads using pointer**

Pass a pointer to the new thread and this thread will set the data in it. Till then in main thread keep on waiting using a condition variable. When new thread sets the data and signals the condition variable, then main thread will wake up and fetch the data from that pointer.

To do a simple thing we used a condition variable, a mutex and a pointer i.e. 3 items to catch a returned value.  
Now suppose we want this thread to return 3 different values at different point of time then problem will become more complex. Could there be a simple solution for returning the value from threads.

The answer is yes using std::future, checkout next solution for it.

**2.) C++11 Way : Using std::future and std::promise**

**std::future** is a class template and its object stores the future value.  
Now what the hell is this future value.

Actually a **std::future** object internally stores a value that will be assigned in future and it also provides a mechanism to access that value i.e. using get() member function. But if somebody tries to access this associated value of future through get() function before it is available, then get() function will block till value is not available.

**std::promise** is also a class template and its object promises to set the value in future. Each std::promise object has an associated std::future object that will give the value once set by the std::promise object.

A **std::promise** object shares data with its associated **std::future** object.

Lets see step by step,

Create a std::promise object in Thread1.



|  |  |
| --- | --- |
| 1 | std::promise<int> promiseObj; |

As of now this promise object doesn’t have any associated value. But it gives a promise that somebody will surely set the value in it and  
once its set then you can get that value through associated std::future object.

But now suppose Thread 1 created this promise object and passed it to Thread 2 object. Now how Thread 1 can know that when Thread 2 is going to set the value in this promise object?

**The answer is using std::future object.**

Every std::promise object has an associated std::future object, through which others can fetch the value set by promise.

So, Thread 1 will create the std::promise object and then fetch the std::future object from it before passing the std””promise object to thread 2 i.e.



|  |  |
| --- | --- |
| 1 | std::future<int> futureObj = promiseObj.get\_future(); |

Now Thread 1 will pass the promiseObj to Thread 2.

Then Thread 1 will fetch the value set by Thread 2 in std::promise through std::future’s get function,



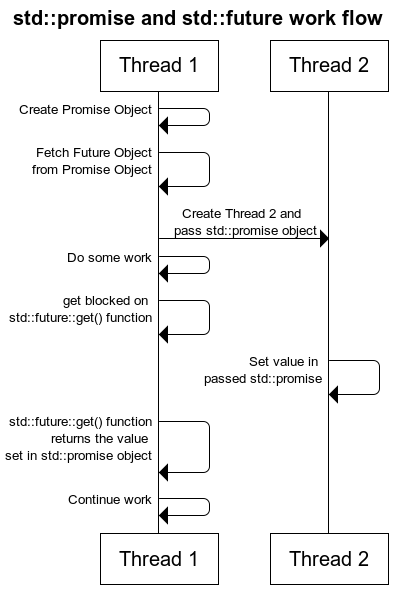
|  |  |
| --- | --- |
| 1 | int val = futureObj.get(); |

But if value is not yet set by thread 2 then this call will get blocked until thread 2 sets the value in promise object i.e.



|  |  |
| --- | --- |
| 1 | promiseObj.set\_value(45); |

Check out complete flow in following Diagram,

[](https://thispointer.com/wp-content/uploads/2015/06/promise.png)

Lets see a complete std::future and std::promise example,



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #include <iostream>  #include <thread>  #include <future>    void initiazer(std::promise<int> \* promObj)  {      std::cout<<"Inside Thread"<<std::endl;     promObj->set\_value(35);  }    int main()  {      std::promise<int> promiseObj;      std::future<int> futureObj = promiseObj.get\_future();      std::thread th(initiazer, &promiseObj);      std::cout<<futureObj.get()<<std::endl;      th.join();      return 0;  } |

If std::promise object is destroyed before setting the value the calling get() function on associated std::future object will throw exception.  
A part from this, if you want your thread to return multiple values at different point of time then just pass multiple std::promise objects in thread and fetch multiple return values from thier associated multiple std::future objects.

# std::async Tutorial & Example

std::async is introduced in c++11.

## what is std::async()

**std::async()** is a function template that accepts a callback(i.e. function or function object) as an argument and potentially executes them asynchronously.



|  |  |
| --- | --- |
| 1  2 | template <class Fn, class... Args>  future<typename result\_of<Fn(Args...)>::type> async (launch policy, Fn&& fn, Args&&... args); |

**std::async** returns a **std::future<T>,** that stores the value returned by function object executed by **std::async()**. Arguments expected by function can be passed to std::async() as arguments after the function pointer argument.

First argument in std::async is launch policy, it control the asynchronous behaviour of std::async. We can create std::async with 3 different launch policies i.e.

* **std::launch::async**
  + It guarantees the asynchronous behaviour i.e. passed function will be executed in seperate thread.
* **std::launch::deferred**
  + Non asynchronous behaviour i.e. Function will be called when other thread will call get() on future to access the shared state.
* **std::launch::async | std::launch::deferred**
  + Its the default behaviour. With this launch policy it can run asynchronously or not depending on the load on system. But we have no control over it.

If we do not specify an launch policy. Its behaviour will be similar to **std::launch::async | std::launch::deferred**.

We are going to use std::launch::async launch policy in this article.

We can pass any callback in std::async i.e.

* Function Pointer
* Function Object
* Lambda Function

Let’s understand the need of std::async by an example,

## Need of std::async()

Suppose we have to fetch some data (string) from DB and some from files in file-system. Then I need to merge both the strings and print.

In a single thread we will do like this,



|  |  |
| --- | --- |
| 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 | #include <iostream>  #include <string>  #include <chrono>  #include <thread>    using namespace std::chrono;    std::string fetchDataFromDB(std::string recvdData)  {  // Make sure that function takes 5 seconds to complete  std::this\_thread::sleep\_for(seconds(5));    //Do stuff like creating DB Connection and fetching Data  return "DB\_" + recvdData;  }    std::string fetchDataFromFile(std::string recvdData)  {  // Make sure that function takes 5 seconds to complete  std::this\_thread::sleep\_for(seconds(5));    //Do stuff like fetching Data File  return "File\_" + recvdData;  }    int main()  {  // Get Start Time  system\_clock::time\_point start = system\_clock::now();    //Fetch Data from DB  std::string dbData = fetchDataFromDB("Data");    //Fetch Data from File  std::string fileData = fetchDataFromFile("Data");    // Get End Time  auto end = system\_clock::now();    auto diff = duration\_cast < std::chrono::seconds > (end - start).count();  std::cout << "Total Time Taken = " << diff << " Seconds" << std::endl;    //Combine The Data  std::string data = dbData + " :: " + fileData;    //Printing the combined Data  std::cout << "Data = " << data << std::endl;    return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1  2 | Total Time Taken = 10 Seconds  Data = DB\_Data :: File\_Data |

As both the functions **fetchDataFromDB()** & **fetchDataFromFile()** takes 5 seconds each and are running in a single thread so, total time consumed will be 10 seconds.

Now as fetching data from DB and file are independent of each other and also time consuming. So, we can run them in parallel.  
One way to do is create a new thread pass a promise as an argument to thread function and fetch data from associated std::future object in calling thread.

The other easy way is using std::async.

## Calling std::async with function pointer as callback

Now let’s modify the above code and call fetchDataFromDB() asynchronously using std::async() i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | std::future<std::string> resultFromDB = std::async(std::launch::async, fetchDataFromDB, "Data");    // Do Some Stuff    //Fetch Data from DB  // Will block till data is available in future<std::string> object.  std::string dbData = resultFromDB.get(); |

**std::async() does following things,**

* It automatically creates a thread (Or picks from internal thread pool) and a promise object for us.
* Then passes the std::promise object to thread function and returns the associated std::future object.
* When our passed argument function exits then its value will be set in this promise object, so eventually return value will be available in std::future object.

Now change the above example and use std::async to read data from DB asyncronously i.e.

Checkout the compete 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  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 | #include <iostream>  #include <string>  #include <chrono>  #include <thread>  #include <future>    using namespace std::chrono;    std::string fetchDataFromDB(std::string recvdData)  {  // Make sure that function takes 5 seconds to complete  std::this\_thread::sleep\_for(seconds(5));    //Do stuff like creating DB Connection and fetching Data  return "DB\_" + recvdData;  }    std::string fetchDataFromFile(std::string recvdData)  {  // Make sure that function takes 5 seconds to complete  std::this\_thread::sleep\_for(seconds(5));    //Do stuff like fetching Data File  return "File\_" + recvdData;  }    int main()  {  // Get Start Time  system\_clock::time\_point start = system\_clock::now();    std::future<std::string> resultFromDB = std::async(std::launch::async, fetchDataFromDB, "Data");    //Fetch Data from File  std::string fileData = fetchDataFromFile("Data");    //Fetch Data from DB  // Will block till data is available in future<std::string> object.  std::string dbData = resultFromDB.get();    // Get End Time  auto end = system\_clock::now();    auto diff = duration\_cast < std::chrono::seconds > (end - start).count();  std::cout << "Total Time Taken = " << diff << " Seconds" << std::endl;    //Combine The Data  std::string data = dbData + " :: " + fileData;    //Printing the combined Data  std::cout << "Data = " << data << std::endl;    return 0;  } |

Now it will take 5 seconds only.

**Output:**



|  |  |
| --- | --- |
| 1  2 | Total Time Taken = 5 Seconds  Data = DB\_Data :: File\_Data |

## Calling std::async with Function Object as callback



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | /\*  \* Function Object  \*/  struct DataFetcher  {  std::string operator()(std::string recvdData)  {  // Make sure that function takes 5 seconds to complete  std::this\_thread::sleep\_for (seconds(5));  //Do stuff like fetching Data File  return "File\_" + recvdData;    }  };    //Calling std::async with function object  std::future<std::string> fileResult = std::async(DataFetcher(), "Data"); |

## Calling std::async with Lambda function as callback

C++



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //Calling std::async with lambda function  std::future<std::string> resultFromDB = std::async([](std::string recvdData){    std::this\_thread::sleep\_for (seconds(5));  //Do stuff like creating DB Connection and fetching Data  return "DB\_" + recvdData;    }, "Data"); |

# packaged\_task<> Example and Tutorial

## std::packaged\_task<>

**std::packaged\_task<>** is a class template and represents a asynchronous task. It encapsulates,

1. A callable entity i.e either function, lambda function or function object.
2. A shared state that stores the value returned or thrown exception by associated callback.

## Need of std::packaged\_task<>

Suppose we have an existing function that fetches the data from DB and return i.e.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // Fetch some data from DB  std::string getDataFromDB( std::string token)  {  // Do some stuff to fetch the data  std::string data = "Data fetched from DB by Filter :: " + token;  return data;  } |

Now we want to execute this function in a separate thread. But how we will fetch the result or exception back in main thread after other thread is finished ?

One way is to change the declaration of function and pass a std::promise<> in the function. Before passing the std::promise<> object in thread function, fetch the associated std::future<> out of it and keep that in main thread. Now, before thread function returns its value, it should set that in passed std::promise<> argument, so that it can be available in associated std::future<> object in main thread. Check below tutorial for this approach i.e,

But creating this std::promise<> and changing function code can be prevented if we use std::packaged\_task<>.

## Using packaged\_task<> with function to create Asynchronous tasks

**std::packaged\_task<>** can wrap around a normal function and make it applicable to run as asynchronous function.

When **std::packaged\_task<>** is called in a separate thread, it calls the associated callback and stores the return  value/exception in its internal shared state. This value can be accessed in other thread or main function through **std::future<>** object.

Let’s create a std::packaged\_task<> from above mentioned function, execute in separate thread and fetch result from its future<> object.

#### Creating std::packaged\_task<> object

std::package\_task<> is a class template, therefore we need to pass template parameter to packaged\_task<> i.e. type of callable function



|  |  |
| --- | --- |
| 1  2 | // Create a packaged\_task<> that encapsulated the callback i.e. a function  std::packaged\_task<std::string (std::string)> task(getDataFromDB); |

#### Fetch the future object from it,



|  |  |
| --- | --- |
| 1  2 | // Fetch the associated future<> from packaged\_task<>  std::future<std::string> result = task.get\_future(); |

#### Passing packaged\_task<> to a thread,

std::packaged\_task<> is movable but not copy-able, so we need to move it to thread i.e.



|  |  |
| --- | --- |
| 1  2 | // Pass the packaged\_task to thread to run asynchronously  std::thread th(std::move(task), "Arg"); |

As packaged\_task was only movable and not copy-able, therefore we fetched the **std::future<>** object from it before moving it to thread.

Thread will execute this task, which internally calls associated callable entity i.e. our function getDataFromDB().

Now when this function returns the value, std::packaged\_task<> sets it to associated shared state and result or exception returned by getDataFromDB() will eventually be available in associated future object.

In main function, fetch result from future<> object i.e.



|  |  |
| --- | --- |
| 1  2 | // Fetch the result of packaged\_task<> i.e. value returned by getDataFromDB()  std::string data =  result.get(); |

get() function will block the calling thread until the callable entity returns and std::packaged\_task<> set the data in its shareable state.

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 | #include <iostream>  #include <thread>  #include <future>  #include <string>    // Fetch some data from DB  std::string getDataFromDB( std::string token)  {  // Do some stuff to fetch the data  std::string data = "Data fetched from DB by Filter :: " + token;  return data;  }    int main()  {    // Create a packaged\_task<> that encapsulated the callback i.e. a function  std::packaged\_task<std::string (std::string)> task(getDataFromDB);    // Fetch the associated future<> from packaged\_task<>  std::future<std::string> result = task.get\_future();    // Pass the packaged\_task to thread to run asynchronously  std::thread th(std::move(task), "Arg");    // Join the thread. Its blocking and returns when thread is finished.  th.join();    // Fetch the result of packaged\_task<> i.e. value returned by getDataFromDB()  std::string data =  result.get();    std::cout <<  data << std::endl;    return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1 | Data fetched from DB by Filter :: Arg |

On similar lines we can create a packaged\_task with lambda function and function objects too i.e.

## Creating packaged\_task with Lambda Function



|  |  |
| --- | --- |
| 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 | #include <iostream>  #include <thread>  #include <future>  #include <string>    int main()  {    // Create a packaged\_task<> that encapsulated a lambda function  std::packaged\_task<std::string (std::string)> task([](std::string token){  // Do some stuff to fetch the data  std::string data = "Data From " + token;  return data;  });    // Fetch the associated future<> from packaged\_task<>  std::future<std::string> result = task.get\_future();    // Pass the packaged\_task to thread to run asynchronously  std::thread th(std::move(task), "Arg");    // Join the thread. Its blocking and returns when thread is finished.  th.join();    // Fetch the result of packaged\_task<> i.e. value returned by getDataFromDB()  std::string data =  result.get();    std::cout <<  data << std::endl;    return 0;  } |

**Output:**



|  |  |
| --- | --- |
| 1 | Data fetched from DB by Filter :: Arg |

## Creating packaged\_task with Function 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 | #include <iostream>  #include <thread>  #include <future>  #include <string>    /\*  \* Function Object to Fetch Data from DB  \*/  struct DBDataFetcher  {  std::string operator()(std::string token)  {  // Do some stuff to fetch the data  std::string data = "Data From " + token;  return data;  }  };    int main()  {    // Create a packaged\_task<> that encapsulated a lambda function  std::packaged\_task<std::string (std::string)> task(std::move(DBDataFetcher()));    // Fetch the associated future<> from packaged\_task<>  std::future<std::string> result = task.get\_future();    // Pass the packaged\_task to thread to run asynchronously  std::thread th(std::move(task), "Arg");    // Join the thread. Its blocking and returns when thread is finished.  th.join();    // Fetch the result of packaged\_task<> i.e. value returned by getDataFromDB()  std::string data =  result.get();    std::cout <<  data << std::endl;    return 0;  } |