## ****Detaching Threads using std::thread::detach()****

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.

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

Separates the thread of execution from the thread object, allowing execution to continue independently. Any allocated resources will be freed once the thread exits.

After calling detach \*this no longer owns any thread.

Ex:

#include <iostream>

#include <chrono>

#include <thread>

void independentThread()

{

std::cout << "Starting concurrent thread.\n";

std::this\_thread::sleep\_for(std::chrono::seconds(2));

std::cout << "Exiting concurrent thread.\n";

}

void threadCaller()

{

std::cout << "Starting thread caller.\n";

std::thread t(independentThread);

t.detach();

std::this\_thread::sleep\_for(std::chrono::seconds(1));

std::cout << "Exiting thread caller.\n";

}

int main()

{

threadCaller();

std::this\_thread::sleep\_for(std::chrono::seconds(5));

}

Output: Starting thread caller.

Starting concurrent thread.

Exiting thread caller.

Exiting concurrent thread.

#### ****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**

|  |  |
| --- | --- |
|  | 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.

|  |  |
| --- | --- |
|  | 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.

|  |  |
| --- | --- |
|  | 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.

|  |  |
| --- | --- |
|  | #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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | #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;  }  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.   |  |  | | --- | --- | |  | 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,   |  |  | | --- | --- | |  | 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   In this article 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.  **std::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()  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 increment 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.   |  |  | | --- | --- | |  | #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,   |  |  | | --- | --- | |  | 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.   |  |  | | --- | --- | |  | 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.      }  }; |   Check List: Member functions  |  |  | | --- | --- | | [(constructor)](https://en.cppreference.com/w/cpp/thread/thread/thread) | constructs new thread object  (public member function) | | [(destructor)](https://en.cppreference.com/w/cpp/thread/thread/%7Ethread) | destructs the thread object, underlying thread must be joined or detached  (public member function) | | [operator=](https://en.cppreference.com/w/cpp/thread/thread/operator%3D) | moves the thread object  (public member function) | | Observers | | | [joinable](https://en.cppreference.com/w/cpp/thread/thread/joinable) | checks whether the thread is joinable, i.e. potentially running in parallel context  (public member function) | | [get\_id](https://en.cppreference.com/w/cpp/thread/thread/get_id) | returns the *id* of the thread  (public member function) | | Operations | | | [join](https://en.cppreference.com/w/cpp/thread/thread/join) | waits for a thread to finish its execution  (public member function) | | [detach](https://en.cppreference.com/w/cpp/thread/thread/detach) | permits the thread to execute independently from the thread handle  (public member function) | | [swap](https://en.cppreference.com/w/cpp/thread/thread/swap) | swaps two thread objects  (public member function) | |

Now 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,

|  |  |
| --- | --- |
|  | #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:**

|  |  |
| --- | --- |
|  | 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.

|  |  |
| --- | --- |
|  | #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 |