Module M58

Partha Pratim Das

Objectives & Outlines

thread Programming in C++

std::thread

std::bind

Race Condition & Data Race

Race Condition Example

Solution by Mutex

Solution by Atomic

Module Summary

Programming in Modern C++

Module M58: C++11 and beyond: Concurrency: Part 1

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Discussed various policies of smart pointer
  - Ownership Policies
  - Implicit Conversion policy
  - Null test policy
- Familiarized with Resource Management using Smart Pointers from Standard Library
  - `unique_ptr`
  - `shared_ptr`
  - `weak_ptr`
  - `auto_ptr`

Module Objectives

- To introduce the notion of concurrent programming in C++11 using thread support
- To explore library support through std::thread and std::bind
- To expose to the bugs in thread programming - race condition and data race
- To discuss examples of thread programs with bugs and their solution

Module Outline

thread Programming in C++

Sources:

- C++11 - the new ISO C++ standard: Threads, Stroustrup, 2016
- C++11 Standard Library Extensions — Concurrency: Threads, isocpp
- std::thread, cppreference
- Concurrency memory model, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- A tutorial on modern multithreading and concurrency in C++, 2020
- **C++11 Multi-threading Tutorials**: Parts 1-8, thisPointer
  - C++11 Multithreading – Part 1 : Three Different ways to Create Threads
- **C++20 Concurrency**: Parts 1-3, Gajendra Gulgulia, 2021
  - C++20 Concurrency: Part 1: synchronized output stream
  - C++20 Concurrency: Part 2: jthreads
  - C++20 Concurrency: Part 3: request_stop and stop_token for std::jthread

thread Programming in C++

Multi-threading in C / C++ Post-Recording

- Unlike built-in multi-threading support for Java, C / C++ needs libraries for multi-threading
- Typically features of a Multi-threading Library include:
  - Thread Management: Create, Join threads etc.
  - Synchronization: Mutex, Lock, Condition variables, Barrier, Future etc.
  - Thread Local Storage
- Third-Party Library
  - POSIX Threads (aka pthreads), is an execution model that exists independently from a language, as well as a parallel execution model
  - Boost C++ Threads
  - Multithreading with C and Win32, Multithreading with C++ and MFC by Microsoft
  - Others: OpenMP, OpenThreads, Qt QThread, Parallel Pattern Library, oneAPI Threading Building Blocks (oneTBB), IPP, etc.
- Language Provided Library
  - Concurrency Support Library for C: C11 / C17
  - Concurrency Support Library for C++: C++11 / C++14 / C++17 / C++20
    - ▷ We will discuss here

Spawn Thread

- A **thread** is a representation of an *execution/computation in a program*
- In C++11, a thread *usually shares an address space with other threads* – it differs from a process, which generally does not directly share data with other processes
- C++ has had a host of threads implementations for a variety of hardware and operating systems in the past, *what is new is a standard-library threads library*
- A thread is launched by constructing a `std::thread` with a function / a function object / a λ:

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

void f() { cout << "In f()" << endl; };
struct F { void operator()() { cout << "In F()()" << endl; }; };
int main() {
    std::thread t1{f};      // f() executes in separate thread
    std::thread t2{F()};    // F()() executes in separate thread
}
// terminate called without an active exception
```

- This program is unlikely to give any useful results – whatever `f()` and `F()` might do
- The program may terminate before or after `t1` executes `f()` and before or after `t2` executes `F()`

Join Thread

- We need to wait for the two tasks to complete:

```
#include <iostream>
#include <thread>
using namespace std;
void f() { cout << "In f()" << endl; };
struct F { void operator()() { cout << "In F()()" << endl; }; };
int main() {
    std::thread t1{f};      // f() executes in separate thread
    std::thread t2{F()};    // F()() executes in separate thread

    t1.join();  // wait for t1
    t2.join();  // wait for t2
}
In f() // 10 out of 15 attempts outputs this
In F()()
                                              //  Non-deterministic behavior on display
In F()() // 5 out of 15 attempts outputs this
In f()
```

- The `join()`s ensure that we don't terminate until the threads have completed. *To **join** means to wait for the thread to terminate*

Thread with Parameters

- Typically, we would like to pass some arguments to the task to be executed. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v) { cout << "In f()" << ": ";
    for(auto x : v) { cout << ' ' << x; } cout << endl;
};
struct F { vector<int>& v;
    F(vector<int>& vv) :v{vv} { }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; } cout << endl;
    };
};
int main() {
    vector<int> my_vec {2, 3, 5, 7, 11 }; // Init vector
    std::thread t1{std::bind(f, my_vec)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec)};            // F(my_vec)() executes in separate thread
    t1.join(); t2.join();
}
In F()():  2 3 5 7 11 // In f():  2 3 5 7 11In F()()
In f():  2 3 5 7 11   // :  2 3 5 7 11
```

- Basically, the standard library function std::bind makes a function object of its arguments

Thread with Output

- In general, we would also like to get a result back from an executed task
- With plain tasks, there is no notion of a return value; std::future is the correct default choice for that
- Alternatively, we can pass an argument to a task telling it where to put its result. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v, int* res) { cout << "In f()" << ": "; *res = 0; // Function with output
    for(auto x : v) { cout << ' ' << x; *res += x; } cout << endl; // Accumulate sum
};
struct F { vector<int>& v; int* res; // Functor with output
    F(vector<int>& v, int* res) : v{v}, res{res} { *res = 0; }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; *res += x; } cout << endl; // Accumulate sum
    };
};
int main() { vector<int> my_vec {2, 3, 5, 7, 11 }; int res1, res2;
    std::thread t1{std::bind(f, my_vec, &res1)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec, &res2)};            // F(my_vec)() executes in separate thread
    t1.join(); t2.join(); std::cout << res1 << ' ' << res2 << '\n';
}
```

```
In f():  2 3 5 7 11
In F()():  2 3 5 7 11
28 28
```

thread Programming in C++: std::thread

Sources:

- C++11 Standard Library Extensions — Concurrency: Threads, isocpp
- std::thread, cppreference
- A tutorial on modern multithreading and concurrency in C++, 2020
- C++11 Multithreading – Part 1 : Three Different ways to Create Threads

std::thread

`class thread;`

- Defined in `<thread>`
- The `class thread` represents a *single thread of execution*. Threads allow *multiple functions to execute concurrently*
- Threads *begin execution immediately upon construction of the associated thread object* (pending any OS scheduling delays), *starting at the top-level function* provided as a constructor argument
- The *return value of the top-level function is ignored* and if it terminates by throwing an exception, `std::terminate` is called
- The top-level function may communicate its return value or an exception to the caller via `std::promise` or by modifying shared variables (which may require synchronization, see `std::mutex` and `std::atomic` later)
- `std::thread` objects may also be in the state that does not represent any thread (after default construction, move from, `detach`, or `join`), and a thread of execution may not be associated with any thread objects (after `detach`)
- No two `std::thread` objects may represent the same thread of execution:
  - `std::thread` is not CopyConstructible or CopyAssignable
  - `std::thread` is MoveConstructible and MoveAssignable

std::thread::id and std::thread::thread

- id: represents the id of a thread
- Member Functions:
  - std::thread::thread: Constructor: Constructs the thread object

```
// A new thread object which does not represent a thread
 thread() noexcept;
// Move constructor. Constructs the thread object to represent the thread of
// execution that was represented by other. After this call other no longer
// represents a thread of execution
 thread(thread&& other) noexcept;
// Creates new std::thread object and associates it with a thread of execution
// The new execution starts with (std::move(f_copy), std::move(args_copy)...)
 template<class Function, class... Args>
 explicit thread(Function&& f, Args&&... args);
// The copy constructor is deleted; threads are not copyable. No two  std::thread
// objects may represent the same thread of execution
 thread(const thread&) = delete;
```

std::thread::~thread and std::thread::operator=

- Member Functions:
  - `std::thread::~thread()`: Destructor: Destroys the thread object
    - ▷ If `*this` has an associated thread (`joinable() == true`), `std::terminate()` is called
    - ▷ A thread object does not have an associated thread (and is safe to destroy) after
      - it was default-constructed
      - it was moved from
      - `join()` has been called
      - `detach()` has been called
  - `std::thread::operator=`

    `thread& operator=(thread&& other) noexcept;`
    - ▷ If `*this` still has an associated running thread (that is, `joinable() == true`), call `std::terminate()`
    - ▷ Otherwise, assigns the state of other to `*this` and sets other to a default constructed state
    - ▷ After this call, `this->get_id()` is equal to the value of `other.get_id()` prior to the call, and other no longer represents a thread of execution

std::thread: Observers and Operations

- Member Functions:
  - Observers:
    - ▷ `joinable`: checks whether the thread running in parallel context is joinable. Typically:
      - `joinable` is `false` before it starts execution or after it has joined
      - `joinable` is `true` while excuting
    - ▷ `get_id`: returns the id of the thread
    - ▷ `native_handle` returns the underlying implementation-defined thread handle
    - ▷ `hardware_concurrency` [static]: returns the number of concurrent threads supported by the implementation
  - Operations:
    - ▷ `join`: waits for the thread to finish its execution
    - ▷ `detach`: permits the thread to execute independently from the thread handle
    - ▷ `swap`: swaps two thread objects

std::thread: Example

```
#include <iostream>
#include <utility> // std::ref, std::move
#include <thread>  // std::this_thread::sleep_for
#include <chrono>  // for sleep time: std::chrono::milliseconds(10)
void f1(int n) { for(int i=0; i<5; ++i) { std::cout << "Thread 1 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
}
void f2(int& n) { for(int i=0; i<5; ++i) { std::cout << "Thread 2 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
}
class foo { public: int n = 0;
    void bar() { for(int i=0; i<5; ++i) { std::cout << "Thread 3 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
    }
};
class baz { public: int n = 0;
    void operator()() { for(int i=0; i<5; ++i) { std::cout << "Thread 4 executing\n";
                          ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
};
// this_thread::sleep_for: Blocks the execution of the thread for sleep_duration or more
```

std::thread: Example


```
using namespace std;
int main() { int n = 0; foo f; baz b;
    // t1 is not a thread
    thread t1; cout << "t1 = " << t1.get_id() << '\n';
    // pass by value
    thread t2(f1, n + 1); cout << "t2 = " << t2.get_id() << '\n';
    // pass by reference
    thread t3(f2, ref(n)); cout << "t3 = " << t3.get_id() << '\n';
    // t4 is now running f2(). t3 is no longer a thread
    thread t4(move(t3)); cout << "t4 = " << t4.get_id() << '\n';
    // t5 runs foo::bar() on object f
    thread t5(&foo::bar, &f); cout << "t5 = " << t5.get_id() << '\n';
    // t6 runs baz::operator() on a copy of object b
    thread t6(b); cout << "t6 = " << t6.get_id() << '\n';
    t2.join(); t4.join(); t5.join(); t6.join();
    cout << "Final n = " << n << '\n';
    cout << "Final f.n (foo::n) = " << f.n << '\n';
    cout << "Final b.n (baz::n) = " << b.n << '\n';
}
```

```
t1 = thread::id of a non-executing thread
t2 = 140285525227264
t3 = 140285516834560
t4 = 140285516834560 // same as t3
t5 = 140285508441856
t6 = 140285500049152
```

```
Thread 4 executing
Thread 3 executing
Thread 2 executing
Thread 1 executing
Thread 4 executing
Thread 3 executing
Thread 2 executing
Thread 1 executing
Thread 4 executing
Thread 2 executing
Thread 3 executing
Thread 1 executing
Thread 4 executing
Thread 2 executing
Thread 1 executing
Thread 3 executing
Thread 4 executing
Thread 2 executing
Thread 1 executing
Thread 3 executing
Final n = 5
Final f.n (foo::n) = 5
Final b.n (baz::n) = 0
```

thread Programming in C++: std::bind

Sources:

- C++11 - the new ISO C++ standard: std::function and std::bind, Stroustrup, 2016
- std::bind, cppreference
- std::bind, cplusplus
- std::function and std::bind: what are they, and when should they be used?, stackoverflow
- std::bind – Tutorial and Examples, thispointer
- Bind function and placeholders in C++, geeksforgeeks

std::bind

```
template<class F, class... Args>                  (since C++11)
/*unspecified*/ bind(F&& f, Args&&... args); (until C++20)
template<class F, class... Args>                  (since C++20)
constexpr /*unspecified*/ bind(F&& f, Args&&... args);

template<class R, class F, class... Args>         (since C++11)
/*unspecified*/ bind(F&& f, Args&&... args); (until C++20)
template<class R, class F, class... Args>         (since C++20)
constexpr /*unspecified*/ bind(F&& f, Args&&... args);
```

- Defined in `<functional>`
- The function template bind generates a forwarding call wrapper for f. Calling this wrapper is equivalent to invoking `f` with some of its arguments bound to `args`. Parameters are:
  - `f`: Callable object (function object, pointer to function, reference to function, pointer to member function, or pointer to data member) that will be bound to some arguments
  - `args`: list of arguments to bind, with the unbound arguments replaced by the placeholders _1, _2, _3... of namespace `std::placeholders`

std::bind

- Suppose we have a callable object `f` with 3 parameters: `f(a, b, c);`
- We want a new function object `g` with only 2 parameters: `g(a, b) = f(a, 4, b);`
- Using `std::bind` to set: `auto g = std::bind(f, _1, 4, _2);` where `_1` (or `_2`) refers to the first (or second) param in `g`. `g` becomes a partial function of `f` with the middle param preset
- `std::bind` is useful in various contexts including:
  - Partial functions (or currying)
  - Reordering parameters / Defining default parameters
  - Generalized function pointer for callback and Passing functors to STL algorithms

```
#include <iostream>
#include <functional>
class MyClass { typedef std::function<void (float result)> TCallback; // shorthand to avoid long typing
    void longRunningFunction(TCallback callback) { double result = 2.7; // this function takes long time
        // do some long running task ...
        callback(result); // callback to return result
    }
    void callback(float result) { std::cout << result; } // called by longRunningFunction after its done
public: void longRunningFunctionAsync() { auto callback = // create callback as a safe function pointer
            std::bind(&MyClass::callback, this, std::placeholders::_1); // Mem fn, object, future param
        longRunningFunction(callback); // normally starts on separate thread, simple call for demo
    }
};
int main() { MyClass().longRunningFunctionAsync(); } // 2.7
```

std::bind: Example

```
#include <iostream>
#include <utility>    // ref, move, mem_fn
#include <memory>     // make_shared, make_unique
#include <functional> // std::placeholders
void f(int n1, int n2, int n3, const int& n4, int n5) {
    std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4 << ' ' << n5 << '\n';
}
int g(int n1) { return n1; }
struct Foo { int data = 10; void print_sum(int n1, int n2) { std::cout << n1+n2 << '\n'; } };
int main() { using namespace std::placeholders;  // for _1, _2, _3...
    std::cout << "1) argument reordering and pass-by-reference: "; int n = 7;
    // _1 and _2 are from std::placeholders, and represent future arguments that will be passed to f1
    auto f1 = std::bind(f, _2, 42, _1, std::cref(n), n); n = 10;
    f1(1, 2, 1001); // 1 is bound by _1, 2 is bound by _2, 1001 is unused: call to f(2, 42, 1, n, 7)
    // 2 42 1 10 7

    std::cout << "2) achieving the same effect using a lambda: "; n = 7;
    auto lambda = [ncref=std::cref(n), n=n](auto a, auto b, auto /*unused*/) { f(b, 42, a, ncref, n); };
    n = 10; lambda(1, 2, 1001); // same as a call to f1(1, 2, 1001)
    // 2 42 1 10 7

    std::cout << "3) nested bind subexpressions share the placeholders: ";
    auto f2 = std::bind(f, _3, std::bind(g, _3), _3, 4, 5);
    f2(10, 11, 12); // makes a call to f(12, g(12), 12, 4, 5);
    // 12 12 12 4 5
```

std::bind: Example


```
// int g(int n1) { return n1; }
// struct Foo { int data = 10; void print_sum(int n1, int n2) { std::cout << n1+n2 << '\n'; } };
// int main() { using namespace std::placeholders;  // for _1, _2, _3...
    std::cout << "4) bind to a pointer to member function: ";
    Foo foo; auto f3 = std::bind(&Foo::print_sum, &foo, 95, _1);
    f3(5); // 100

    std::cout << "5) bind to a mem_fn that is a pointer to member function: ";
    auto ptr_to_print_sum = std::mem_fn(&Foo::print_sum); // std::mem_fn generates wrapper objects
    auto f4 = std::bind(ptr_to_print_sum, &foo, 95, _1);  // for pointers to members
    f4(5); // 100

    std::cout << "6) bind to a pointer to data member: ";
    auto f5 = std::bind(&Foo::data, _1);
    std::cout << f5(foo) << '\n'; // 10

    std::cout << "7) bind to a mem_fn that is a pointer to data member: ";
    auto ptr_to_data = std::mem_fn(&Foo::data);
    auto f6 = std::bind(ptr_to_data, _1);
    std::cout << f6(foo) << '\n'; // 10

    std::cout << "8) use smart pointers to call members of the referenced objects: ";
    std::cout << f6(std::make_shared<Foo>(foo)) << ' ' << f6(std::make_unique<Foo>(foo)) << '\n';
    // 10 10
}
```

Race Condition & Data Race

Sources:

- Concurrency in C++11, University of Chicago
- Race Conditions versus Data Races, modernescpp, 2017
- Malicious Race Conditions and Data Races, modernescpp, 2017
- C++11 Multithreading – Part 4: Data Sharing and Race Conditions
- C++11 Multithreading – Part 5: Using mutex to fix Race Conditions

Race Condition & Data Race

- We often talk about bugs in multi-threading:
  - *Race Condition*
  - *Data Race*
- Are they same?
  - No, they are not
  - They are not a subset of one another
  - They are also neither the necessary, nor the sufficient condition for one another
- *Race Condition*: A race condition is a semantic error
  - A race condition is a situation, in which the result of an operation depends on the interleaving of certain individual operations
  - Many race conditions can be caused by data races, but this is not necessary
- *Data Race*: A data race occurs when 2 instructions from different threads access the same memory location without synchronization
  - A data race is a situation, in which at least two threads access a shared variable at the same time. At least one thread tries to modify the variable.
  - The discovery of data race can be automated
- We take examples to illustrate both

Race Condition & Data Race: Race Condition Example

Example 1: Race Condition

- Let us write a simple program to compute:

$$\sum_{i=1}^{20} i^2 = \frac{20 \times (20+1) \times (2 \times 20 + 1)}{6} = 2870$$

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

int accum = 0; // init accumulator
void square(int x) { accum += x * x; } // compute and accumulate product
int main() {
    for (int i = 1; i <= 20; i++) { square(i); }
    cout << " accum = " << accum << endl; // print the result
}
```

- Assuming that `x*x` is a heavy computation (fake it!) let us write a simple multi-threaded program for the above:
  - Spawn 20 threads
  - Each thread computes `square` for a distinct value
  - The accumulated result is available after the threads join

Example 1: Race Condition


```
#include <iostream>
#include <vector>
#include <thread>  // thread
using namespace std;

int accum = 0; // init accumulator
void square(int x) { // called in different threads - one each for 1 .. 20
    accum += x * x; // compute and accumulate product
}
int main() {
    vector<thread> ths; // vector of threads
    for (int i = 1; i <= 20; i++) {
        ths.push_back(thread(&square, i)); // 20 threads spawned
    }
    for (auto& th : ths) {
        th.join(); // join 20 threads
    }
    cout << " accum = " << accum << endl; // print the result
}
```

Example 1: Race Condition: Random Delay

- As we execute the multi-threaded program, it seems to correctly give the result 2870
- Does it? Always? Can we be sure?
- We need to validate our assumption that `x*x` is indeed a heavy computation, and on different threads it may take different quanta of time
- To increase the heaviness of computation of `x*x`, we insert a delay between computation of `x*x` and its accumulation
- To simulate varying situations between threads, we randomize the delay

```
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product

    // random number between 0 and 100 where std::rand() is from <cstdlib>
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); //

    // random delay: 0ms .. 100ms where std::milliseconds() is from <chrono>
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));

    accum += p; // accumulate product
}
```

- We try again!

Example 1: Race Condition: Random Delay


```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;

int accum = 0; // init accumulator
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));    // random delay: 0ms .. 100ms
    accum += p; // accumulate product
}
int main() {
    vector<thread> ths; // vector of threads
    for (int i = 1; i <= 20; i++) {
        ths.push_back(thread(&square, i)); // 20 threads spawned
    }
    for (auto& th : ths) {
        th.join(); // join 20 threads
    }
    cout << " accum = " << accum << endl; // print the result
}
```

Example 1: Race Condition: Random Delay + Repeat

- As we execute the modified multi-threaded program with delays, it seems still to correctly give the result 2870
- We keep trying. Running it over and over again to be convinced of the correctness (someone told that thread programming is tricky)
- When we are almost certain of the correctness, suddenly on the 37^{th} run, we get 2845!
- Was it a computer error, false observation? We try another 100+ times and always get 2870!
- We decide we need to automate the runs:
  - We run (trial) in a in an infinite loop
  - We break the loop if the trail fails to produce correct result

```
int main() {
    int trial_count = 0; // counting trials before failure
    do {
        ++trial_count; // increment trial counter
        accum = 0; // reset to start a trial
        // codes for vector of threads, 20 threads spawned, join 20 threads
    } while (accum == 2870); // 1^2+2^2+...20^2 = 2870: infinite loop!!!
}
```

- Correct program will loop forever! But Murphy says: *If anything can go wrong, it will*

Example 1: Race Condition: Random Delay + Repeat

```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;

int accum = 0; // init accumulator
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));    // random delay: 0ms .. 100ms
    accum += p; // accumulate product
}
int main() { int trial_count = 0; // counting trials before failure
    do { ++trial_count; // increment trial counter
    if (0 == trial_count % 100) // message after every 100 trials - that the process is alive
        cout << "trials = " << trial_count << endl;
        accum = 0; // reset to start a trial
        vector<thread> ths; // vector of threads
        for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
        for (auto& th : ths) { th.join(); } // join 20 threads
    } while (accum == 2870); // 1^2+2^2+...20^2 = 2870: infinite loop!!!
    cout << "trials = " << trial_count << " accum = " << accum << endl; // print if there is bad result
}
```

Example 1: Race Condition: Random Delay + Repeat: Results

- Murphy is correct![1]. Every time we run, the loop breaks after some trials (not very large in fact)
- Here are the first 20 runs (of multiple trials). Every time there is some trial which **falls short of 2870**

```
trials =   56 accum = 2845
trials =    1 accum = 2806
trials =  221 accum = 2470
trials =  825 accum = 2806
trials = 1502 accum = 2861
trials =  487 accum = 2861
trials =  113 accum = 2470
trials =  156 accum = 2861
trials = 1120 accum = 2581
trials =  914 accum = 2645
trials = 1279 accum = 2726
trials =  932 accum = 2806
trials = 1120 accum = 2581
trials =  174 accum = 2845
trials =  190 accum = 2645
trials =  229 accum = 2546
trials =  802 accum = 2821
trials =   67 accum = 2614
trials =  784 accum = 2869
trials =  295 accum = 2854
```

[1] *which is a logical contradiction by Murphy's law*

Example 1: Race Condition: Analysis

- So what is going wrong? We have hit a *Race Condition*
- When the compiler processes `accum += x * x;`, reading the current value of `accum` and setting the updated value is not an *atomic* (meaning *indivisible*) event. Let us re-write `square` to capture this:

```
int t1 = x * x;
int t2 = accum;
accum = t2 + t1;
```

- Now, let us assume that we working with only 2 threads (for 1 & 2). The threads are interleaved over time and a possible sequence is:

```
// Thread 1 (Th1)          // Thread 2 (Th2)
                           int t1 = 2 * 2;          // Th2.t1 = 4
int t1 = 1 * 1;                                     // Th1.t1 = 1
                           t2 = accum;              // Th2.t2 = 0
t2 = accum;                                         // Th1.t2 = 0
accum = t2 + t1;                                    // accum = 1
                           accum = t2 + t1;         // accum = 4
```

- We end up with `accum` as 4, instead of the correct 5
- It also makes clear why a wrong result will always be less
- Let us now provide two solutions to the race condition problem using
  - Mutex
  - Atomic

Race Condition & Data Race: Race Condition Example: Solution by Mutex

Example 1: Race Condition: Solution by Mutex

- A mutex (mutual exlusion) allows us to encapsulate blocks of code that should only be executed in one thread at a time. Keeping the main function the same:

```
int accum = 0;
mutex accum_mutex; // mutex variable

void square(int x) {
    int temp = x * x;
    accum_mutex.lock();   // gets the lock on accum_mutex
    accum += temp;
    accum_mutex.unlock(); // release the lock on accum_mutex
}
```

- We try running the program repeatedly again and the problem should now be fixed
- The first thread that calls `lock()` gets the `lock`
- During this time, all other threads that call `lock()`, will wait at that line for the mutex to be unlocked. Creates a *Critical Section*
- It is important to introduce the variable `temp`, since we want the `x * x` calculations to be outside the lock-unlock block, otherwise we would be hogging the lock while we are running our heavy calculations

Example 1: Race Condition: Solution by Mutex

```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <mutex>   // mutex
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
int accum = 0; // init accumulator
mutex accum_mutex; // mutex variable
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));    // random delay: 0ms .. 100ms
    accum_mutex.lock();   // gets the lock on accum_mutex
    accum += p;           // accumulate product
    accum_mutex.unlock(); // release the lock on accum_mutex
}
int main() { int trial_count = 0; // counting trials before failure
    do { ++trial_count; // increment trial counter
        accum = 0; // reset to start a trial
        vector<thread> ths; // vector of threads
        for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
        for (auto& th : ths) { th.join(); } // join 20 threads
    } while (accum == 2870); // 1^2+2^2+...20^2 = 2870: infinite loop!!!
    cout << "trials = " << trial_count << " accum = " << accum << endl; // print if there is bad result
}
```

Race Condition & Data Race: Race Condition Example: Solution by Atomic

Example 1: Race Condition: Solution by Atomic

- With Mutex the problem gets fixed. The program does not produce a wrong result even after 6000+ trials
- Interestingly, C++11 offers even nicer abstractions to solve this problem. For instance, the `atomic` container:

```
#include <atomic>

atomic<int> accum(0); // makes accum and initializes to 0

void square(int x) {
    accum += x * x;
}
```

- We do not need to introduce temp here, since `x * x` will be evaluated before handed off to `accum`, so it will be outside the atomic event
- However, we will continue to show the solution using the temporary

Example 1: Race Condition: Solution by Atomic


```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <atomic>  // atomic
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
atomic<int> accum(0); // makes accum and initializes to 0
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));    // random delay: 0ms .. 100ms
    accum += p;           // accumulate product
}
int main() { int trial_count = 0; // counting trials before failure
    do { ++trial_count; // increment trial counter
        accum = 0; // reset to start a trial
        vector<thread> ths; // vector of threads
        for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
        for (auto& th : ths) { th.join(); } // join 20 threads
    } while (accum == 2870); // 1^2+2^2+...20^2 = 2870: infinite loop!!!
    cout << "trials = " << trial_count << " accum = " << accum << endl; // print if there is bad result
}
```

- Works fine. Does not produce a wrong result even after 5000+ trials

Module Summary

- Introduced the notion of concurrent programming in C++11 using thread support
- Explored library support through std::thread and std::bind
- Exposed to the bugs in thread programming - race condition and data race
- Discussed examples of thread programs with bugs and their solution