**Thread Classes**

The C++11 standard added five multi-threading libraries to the Standard Library Collection.  Its thread and future libraries contain all the templates for simple multi-threaded solutions.  The thread library provides support for creating and managing threads that execute concurrently.  The future library provides support for retrieving the result from a function that has executed in the same or a concurrently executing thread.  These libraries implement the Resource Allocation is Initialization (RAII) idiom.

This chapter describes the thread class template in detail and demonstrates how to launch and synchronize threads.  It also describes how to communicate data between threads using objects generated from the future class, the promise class and the packaged\_task() class templates as well as functions generated from the async() function template.

Thread Class

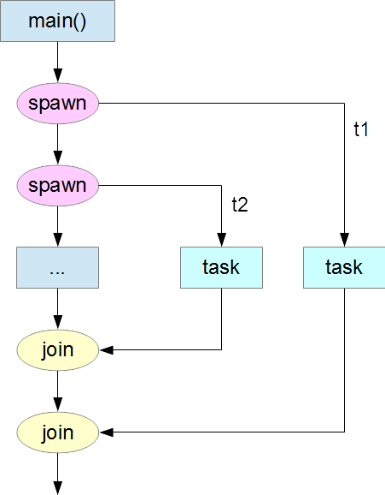
The thread class defines an object that represents a single thread of execution in a team of threads.  The thread class template is defined in the header file:

#include <thread>

A thread object is either *joinable* or *not-joinable*.  A joinable object represents an actual thread of execution with a unique id.  A non-joinable object represents a potential thread of execution.  Operations on a thread object can change its joinable/non-joinable state.

Example

The following program executes three tasks concurrently.  It spawns two child threads from its main thread, executes all three tasks and waits in its main thread for the spawned threads to finish.



Each thread performs the same task (task()):

|  |  |
| --- | --- |
| // Thread Class  // thread.cpp  #include <iostream>  #include <string>  #include <thread>  void task(const std::string& str) {  std::cout << str + " says Hi\n";  }  int main() {  // spawn child thread t1  std::thread t1(task, "t1");  // spawn child thread t2  std::thread t2(task, "t2");  // continue executing main thread  task("main");  // synchronize - IMPORTANT!  t2.join();  t1.join();  } | main says Hi  t1 says Hi  t2 says Hi |

The synchronization step is necessary.  Had we neglected to join() the spawned threads to the main thread, the result would be undefined.  The main thread could, for instance, finish executing its task and return control to the operating system before one or both of the spawned threads had finished executing their tasks.

Member Functions

The member functions of the thread templated class include:

* thread() noexcept - default - creates a not-joinable thread object (a potential thread of execution)
* thread(thread&& t) noexcept - moves the thread handler from thread t to the newly constructed thread object
* ~thread() - destroys the current thread object
* thread& operator=(thread&& t) noexcept - moves the thread handler from thread t to the current not-joinable object
* thread::id get\_id() const - returns the unique identifier of the current thread object
* bool joinable() const noexcept - returns true if the current object is an actual thread of execution
* void join() - returns when the current object has completed executing its task
* void detach() - detaches the current object from its parent object
* void swap(thread& t) - swaps the state of the current object with the state of object t

This template's copy-constructor and the copy-assignment special member functions are deleted.  The thread::id type represents a thread identifier.  The definition of the thread class includes an overload of the insertion operator for a right operand of type thread::id.

The class definition also includes a template for constructing objects that execute functions, function objects or lambda expressions.  This template takes the form

template <*typename Fn, typename... Args*>

explicit thread(*Fn&& f, Args&&... args*);

Fn is the type of the function, function object or lambda expression and Args are the arguments passed to the function call itself.  See the following for examples.

Thread Identifier

The thread identifier of a joinable thread object is accessible from within the function executing the thread's task.  std::this\_thread::get\_id() returns this identifier.

The following program launches 10 thread objects and displays their identifiers on standard output.  The program creates each object through the constructor template with the address of a function as its sole argument:

|  |  |
| --- | --- |
| // Thread Class - Thread Identifiers  // thread\_id.cpp  #include <iostream>  #include <thread>  #include <vector>  const int NT = 10;  void task() {  std::cout << "Thread id = " <<  std::this\_thread::get\_id() << std::endl;  }  int main() {  // create a vector of threads  std::vector<std::thread> threads;  // launch execution of each thread  for (int i = 0; i < NT; i++)  threads.push\_back(std::thread(task));  // synchronize their execution here  for (auto& thread : threads)  thread.join();  } | Thread id = Thread id = 23620  Thread id = 23596  Thread id = 23608  23584  Thread id = 23592  Thread id = 23612  Thread id = 23600  Thread id = 23588  Thread id = 23604  Thread id = 23616 |

Note that a thread does not necessarily pass the entire cascaded insertion expression to standard output as a cohesive unit.  Some executing threads interleve the constituent operations amongst themselves, as shown in the right panel.

The following program uses the templated constructor to launches a task that takes a single argument:

|  |  |
| --- | --- |
| // Thread Class - Function with Arguments  // thread\_id\_arg.cpp  #include <iostream>  #include <thread>  #include <vector>  const int NT = 10;  void task(int i) {  std::cout << i << " Thread id = " <<  std::this\_thread::get\_id() << std::endl;  }  int main() {  // create a vector of not-joinable threads  std::vector<std::thread> threads;  // launch execution of each thread  for (int i = 0; i < NT; i++)  threads.push\_back(std::thread(task, i));  // synchronize their execution here  for (auto& thread : mythreads)  thread.join();  } | 0 Thread id = 23968  3 Thread id = 19484  7 Thread id = 21992  2 Thread id = 21656  8 Thread id = 17844  5 Thread id = 23764  4 Thread id = 24396  6 Thread id = 22580  1 Thread id = 24100  9 Thread id = 23964 |

In this particular run, the output from different threads happens not to interleave.

Function Object

The function object version of the above example is as follows:

|  |  |
| --- | --- |
| // Thread Class - Function Object  // thread\_id\_fo.cpp  #include <iostream>  #include <thread>  #include <vector>  const int NT = 10;  class Task {  public:  Task(){}  void operator()(int i) {  std::cout << i << " Thread id = " <<  std::this\_thread::get\_id() << std::endl;  }  };  int main() {  // create a vector of not-joinable threads  std::vector<std::thread> threads;  // launch execution of each thread  for (int i = 0; i < NT; i++)  threads.push\_back(std::thread(Task(), i));  // synchronize their execution here  for (auto& thread : threads)  thread.join();  } | 023 Thread id = 25096  1 Thread id = 22704  7 Thread id = 24540  Thread id = 25516  6 Thread id = 17152  4 Thread id = 14460  Thread id = 25024  9 Thread id = 24632  8 Thread id = 18060  5 Thread id = 22796 |

In this particular run, the output from different threads happens to interleave.

Lambda Expression

The lambda expression version of the above example is more compact and accesses the index i by value as a non-local variable:

|  |  |
| --- | --- |
| // Thread Class - Lambda Expression  // thread\_id\_lambda.cpp  #include <iostream>  #include <thread>  #include <vector>  const int NT = 10;  int main() {  // create a vector of not-joinable threads  std::vector<std::thread> threads;  // launch the execution of each thread  for (int i = 0; i < NT; i++)  threads.push\_back(std::thread([=]() {  std::cout << i << " Thread id = " <<  std::this\_thread::get\_id() << std::endl;  }));  // synchronize their execution here  for (auto& thread : threads)  thread.join();  } | 0 Thread id = 23624  3 Thread id = 24412  7 Thread id = 20744  2 Thread id = 9580  8 Thread id = 18552  5 Thread id = 23808  4 Thread id = 24560  6 Thread id = 24296  1 Thread id = 21600  9 Thread id = 19552 |

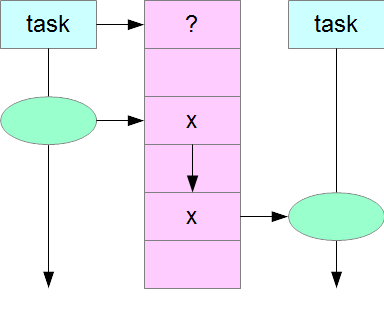
Note that the order of output differs from that for the other versions.

Future Library

The future library of the thread support category facilitates efficient transfer of values between tasks through *shared state*s.  The class and function templates that support communications across a shared state are defined in the header file:

#include <future>

A *future* object retrieves a value that a *provider* has stored in a shared state.  Each provider-future pair establishes a synchronization point for two tasks that are executing concurrently.  The provider creates an empty shared state on initialization.  Once the provider has supplied a value to that shared state, that state is *ready* for access by the future object associated with that provider.  A shared state can survive the lifetime of its provider.



Futures

Instantiation of the templated std::future class creates a future object.  A future object is either *valid* or *not-valid*.  A valid object is associated with a shared state and can retrieve the value of that shared state once it is ready.  Until it is ready, any retrieval request necessarily waits.

Member Functions

The member functions of the templated future class include:

* future() noexcept - default - creates a future object not associated with a shared state
* future(future&& f) noexcept - moves the shared state from future f to the current object
* ~future() - disassociates the shared state from the current object and destroys the state if not associated with any other object
* future& operator=(future&& f) noexcept - acquires the shared state of future f
* T get() - returns the value stored in the current object's shared state, if ready; waits, if not ready
* bool valid() const noexcept - returns true if the current object is associated with a shared state
* void wait() const - waits for the current object's shared state to be ready

This class' copy-constructor and the copy-assignment special member functions are deleted.

Providers

A *provider* object complements a future object.  One of the following templates can instantiate a *provider* object:

* std::promise class template
* std::packaged\_task class template
* std::async() function template

promise objects

A promise object creates or acquires a shared state in which it can store a value.

The templated promise class defines a simple set\_value() counterpart to the get() member function of the templated future class.  The member functions of the templated promise class include:

* promise() - default constructor - a promise object with a new empty shared state
* promise(promise&& p) noexcept - moves the shared state from promise p to the newly created current object
* ~promise() - abandons the shared state and destroys the promise
* promise& operator=(promise&& p) noexcept - moves the shared state from promise p to the current object
* future<T> get\_future() - returns the future object associated with the current object's shared state
* void set\_value(const T&) - stores a value in the current object's shared state, making it ready for retrieval
* void swap(promise& p) - swaps the shared state of the current object with the shared state of object p

This template's copy-constructor and the copy-assignment operations are deleted.

packaged\_task object

A packaged\_task object consists of two components: a stored task and a shared state.

The templated packaged\_task class defines a simple wrapper for passing the result of a task to a future object; that is, for launching a thread that executes a task and capturing the return value, which a future can subsequently retrieve.  The member functions of the templated packaged\_task class include:

* packaged\_task() - default constructor - a packaged\_task object with no shared state and no task
* packaged\_task(packaged\_task&& p) noexcept - moves the shared state and stored task from packaged\_task p to the newly created current object
* ~packaged\_task() - abandons the shared state and destroys the packaged\_task
* packaged\_task& operator=(packaged\_task&& p) noexcept - acquires the shared state and stored task from packaged\_task p
* bool valid() const noexcept - returns true if the current object is associated with a shared state and a stored task
* future<T> get\_future() - returns the future object associated with the current object's shared state
* void operator()(Args...) - forwards the function arguments of type Args to the stored task and initiates its execution
* void swap(packaged\_task& p) - swaps the shared state of the current object with that of object p

This template's copy-constructor and copy-assignment operations are deleted.

async() function

The templated async() function provides an extremely simple pair that spawns a thread to execute a task and creates a future for retrieving the return value from that task.

An async() function

* accepts the address of the function that defines the task
* launches the task
* reverts control to its caller
* returns a future object that can retrieve the value returned on the task's completion

The executing task stores the return value temporarily in a shared state.  The future object can retrieve the value from this shared state.

The template for an async() function has the form

template<class Fn, class... Args>

future<typename result\_of<f(Args...)>::type> async(Fn&& fn, Args&&... args);

fn is a pointer to a function or any kind of move-constructable function object of type Fn and args denotes arguments of type Args to the function, where type is also move-constructable.

Examples

Promise - Future

Fulfilling a promise on a child thread and retrieving the promised value on the main thread involves passing the promise object by reference to the thread.  In the following program task() on thread t fulfils the promise by setting a value.  The main thread retrieves that value.

|  |  |
| --- | --- |
| // Promise - Future  // promise\_future.cpp  #include <iostream>  #include <thread>  #include <future>  void task(std::promise<double>& p) {  p.set\_value(12.34);  }  int main() {  std::promise<double> p;  std::future<double> f = p.get\_future();  std::thread t(task, std::ref(p));  std::cout << "Value = " << f.get()<< std::endl;  t.join();  } | Value = 12.34 |

Note that any return value from the function executed by a child thread can be captured by an std::promise object.  The return value from the function is otherwise ignored.

Packaged Task

The following program packages a task and a shared state.  The get\_future() member function retrieves the future object that will hold the return value generated by execution of the task.  Calling the packaged\_task executes the task, while calling the get() member function on the future object retrieves the return value, which the main function can then display:

|  |  |
| --- | --- |
| // Packaged Task  // packaged\_task.cpp  #include <iostream>  #include <thread>  #include <future>  double task(double x) { return x \* 2; }  int main() {  std::packaged\_task<double(double)> pt(task);  auto f = pt.get\_future();  pt(10);  double r = f.get();  std::cout << "Result = " << r << std::endl;  } | Result = 20 |

async()

The following program launches task() asynchronously and returns the future object associated with the shared state.  The get() member function on the future object retrieves the value of the shared state, which the main function can then display:

|  |  |
| --- | --- |
| // Asynchronous Launch  // async.cpp  #include <iostream>  #include <thread>  #include <future>  double task(double x) { return x \* 2; }  int main() {  std::future<double> f = std::async(task, 10);  double r = f.get();  std::cout << "Result = " << r << std::endl;  } | Result = 20 |

On the three examples, this is the simplest.

The following program demonstrates changes in the validity of future objects as a result of its operations on them.  The program:

* creates 2 future objects - a default one and a valid one
* moves the shared state of the future object created by asychronously launching the execution of get()
* moves the shared state from the valid future object (g) to the not-valid one (f)
* retrieves the value from the shared state by calling the get() member function on the valid future object (f)

|  |  |
| --- | --- |
| // Future Class Template - Explicit Asynchronous Launch  // future\_async.cpp  #include <iostream>  #include <future>  double get() { return 12.34; }  int main() {  std::future<double> f; // default ctor  std::future<double> g = std::async(get); // move-ctor  std::cout << "After Construction" << std::endl;  std::cout << (f.valid() ? "f is valid" :  "f is not valid") << std::endl;  std::cout << (g.valid() ? "g is valid" :  "g is not valid") << std::endl;  f = std::move(g); // move-assignment  std::cout << "After Assignment" << std::endl;  std::cout << (f.valid() ? "f is valid" :  "f is not valid") << std::endl;  std::cout << (g.valid() ? "g is valid" :  "g is not valid") << std::endl;  double a = f.get(); // retrieve shared value  std::cout << "After Retrieval" << std::endl;  std::cout << (f.valid() ? "f is valid" :  "f is not valid") << std::endl;  std::cout << (g.valid() ? "g is valid" :  "g is not valid") << std::endl;  std::cout << "Return Value = " << a << std::endl;  } | After Construction  f is not valid  g is valid  After Assignment  f is valid  g is not valid  After Retrieval  f is not valid  g is not valid  Return Value = 12.34 |

Note that after retrieval of the value of a shared state associated with a future object that object is no longer valid.

Thread Local Storage

The same variable can have a different storage location for each thread in a team of threads.  We identify such variables as having thread\_local storage duration.  This storage duration lasts the lifetime of that thread and is the equivalent of static storage duration for a program variable.

In the following example, k has three separate storage locations: one for the main thread, one for thread t1 and one for thread t2:

|  |  |
| --- | --- |
| // Thread Local Storage Duration  // thread\_local.cpp  #include <iostream>  #include <sstream>  #include <thread>  thread\_local int k = 0;  void task(int i) {  k = i;  std::stringstream s;  s << k << " at " << &k << std::endl;  std::cout << s.str();  }  int main() {  k = 10;  std::thread t1(task, 15);  std::thread t2(task, 20);  t1.join();  t2.join();  task(k);  } | 15 at 00A6730C  20 at 00A67CCC  10 at 00A5644C |

Note that the address of the storage location for k is different for each thread.