Threads 2

pass args to thread func

In calling a thread, everything is passed to the thread as a copy.

- this is a good thing, as we want copies local in a thread to avoid any conflict
- if you want to pass by reference, you need to do std::ref

2.1

```
#include<iostream>
using std::cout; using std::endl;
#include<thread>
using std::thread;
#include<string>
using std::string; using std::to_string;

void thread_fun(long start, long stop, string &outs){
  int even=0, odd=0;
  for(int i=start; i<stop; i++)
    if (i%2 == 0)
      even++;
    else
      odd++;
  outs = "Even:" + to_string(even) + ", Odd:" + to_string(odd);
}

int main (){
  string s;
  thread t(thread_fun, 100, 1000, std::ref(s));
  t.join();
  cout << s << endl;
}
```

string by ref

need to tell thread
this is a reference

move semantics

You cannot copy a thread object (much like you cannot copy an ostream).

However, threads do allow for move semantics

- you can move one thread of execution to another thread object
- you can move a thread out of a function.

2.2

```
void thread_fun(long l, double d, string s, long &num){
  long val;2.2
  val = (all_of(s.begin(), s.end(), isdigit)) ? stol(s) : 0;
  num = l + d + val;
}

thread construct_thread(long l, double d, string s, long &num){
  thread t(thread_fun, l, d, s, ref(num));
  return t;
}

int main () {
  long l;
  thread t1(thread_fun, 1, 3.14, "1234", ref(l));
  thread t2 = std::move(t1);
  t2.join();
  cout << "Long:"<<l<<endl;

  l = 0;
  thread t3 = construct_thread(10, 35.678, "5678", ref(l));
  t3.join();
  cout << "Long:"<<l<<endl;
}
```

construct a thread and return it

move thread to another thread obj

thread return is a move to another thread object

timing

You can use time (C++11 time) at various places in the thread libraries and their support:

- *_for takes a duration (how long)
- *_until takes a time point (wait until some timepoint in the future

this_thread

`this_thread` is a namespace that identifies the presently running thread

- `sleep_for(duration)`
- `sleep_until(timepoint)`
- `get_id(unique thread it)`
- `yield()`

Operation	Effect
`this_thread::get_id()`	Yields the ID of the current thread
`this_thread::sleep_for(`*dur*`)`	Blocks the thread for duration *dur*
`this_thread::sleep_until(`*tp*`)`	Blocks the thread until timepoint *tp*
`this_thread::yield()`	Hint to reschedule to the next thread

Table 18.5. Thread-Specific Operations of Namespace `std::this_thread`

2.3

```
#include<thread>
using std::thread;
#include<chrono>
using std::chrono::duration;
using std::chrono::seconds;
#include<iostream>
using std::cout; using std::endl;

bool stop = false;

void thread_fun(){
  cout << "I am:"<<std::this_thread::get_id()<<endl;
  auto wait_for = seconds(2);
  while (!stop){
    cout << "**Inside thread...\n";
    std::this_thread::sleep_for(wait_for);
  }
}

int main() {
  thread  t = thread(thread_fun);
  getchar();  // wait for user to press enter:
  stop = true;  // stop thread:
  t.join();    // wait for thread to finish
}
```

thread unique id

timeout value

checking a global variable

Notice that in the previous code, there is a global variable stop that all the functions can "watch"

The thread is allowed to run until the user enters a character. At that point the global changes and the thread will notice (after it's timeout) and quit.

Note where the `.join()` is

2.4

```
…
#include<thread>
using std::this_thread::sleep_for;
using std::this_thread::get_id;

bool stop=false;
void thread_fun(int seed){
  default_random_engine dre(seed);
  uniform_int_distribution<int> dist(500,2000);
  int rand_val;
  auto wait_for = milliseconds(1);
  while (!stop){
    rand_val = dist(dre);
    wait_for = milliseconds(rand_val);
    cout << "Thread:"<<get_id()<<" slept for:"
        << dec << wait_for.count()<<endl;
    sleep_for(wait_for);
  }
}
```

shortcut the this_thread namespace

global var

```
int main () {
  thread t1(thread_fun, 1);
  thread t2(thread_fun, 2);
  thread t3(thread_fun, 3);
  thread t4(thread_fun, 4);
  getchar();    // wait
  stop = true;  // stop thread:
  t1.join();
  t2.join();
  t3.join();
  t4.join();
}
```

all threads stop when global is set

Operation	Effect
thread *t*	Default constructor; creates a *nonjoinable* thread object
thread *t*(*f*,...)	Creates a thread object, representing *f* started as thread (with additional args), or throws `std::system_error`
thread *t*(*rv*)	Move constructor; creates a new thread object, which gets the state of *rv*, and makes *rv nonjoinable*
t.~***thread***()	Destroys `*this`; calls `std::terminate()` if the object is *joinable*
t = *rv*	Move assignment; move assigns the state of *rv* to *t* or calls `std::terminate()` if *t* is *joinable*
t.`joinable()`	Yields `true` if *t* has an associated thread (is *joinable*)
t.`join()`	Waits for the associated thread to finish (throws `std::system_error` if the thread is not *joinable*) and makes the object *nonjoinable*
t.`detach()`	Releases the association of *t* to its thread while the thread continues (throws `std::system_error` if the thread is not *joinable*) and makes the object *nonjoinable*
t.`get_id()`	Returns a unique `std::thread::id` if *joinable* or `std::thread::id()` if not
t.`native_handle()`	Returns a platform-specific type `native_handle_type` for nonportable extensions

Table 18.4. Operations of Objects of Class `thread`