Chapter 2: Managing threads

Basic thread management

Launching a Thread

```
struct background_task {
  void operator()()
  {
    std::cout << "Executing background task";
  }
};

background_task f;
std::thread t(f);
```

After the thread has started you must chose to either detach it, or to join it. If you don’t do this, and the thread object get’s destroyed, std::terminate will be called.

```
thread1.join() // join
thread.detach() // detatch
```

You should be carefull with “C++ most vexing parsing”, it might deduce a temp object creation as a function declaration.

For instance in the following code IAmAFunctionDeclaration is parsed as function declaration:

```
std::thread IAmAFunctionDeclaration(background_task());
```

This can be avoided using the “{}”-brackets:

```
std::thread IAmNotAFunctionDeclaration{ background_task() };
std::thread IAmAlsoNotAFunctionDeclaration( background_task{} );
```

Waiting for a thread to complete

```
thread.join()
```

- You should only call join once, if called a second time it will return false.
- Brute force technique
- Detatch or Join must be called before the thread object is gone.
- Careful with exceptions that might circumvent the join

```
try {
  ... some code
}
catch(...){
  t.join(); // <- make sure join is called even with exception thrown
  throw; // rethrow the exception
}
```

One way to guarantee that join will be called is by the use of a guard.

```
class thread_guard {
  std::thread& t;
public:
  explicit thread_guard(std::thread& t_):
    t(t_){}

  thread_guard(const thread_guard&) = delete;

  thread_guard operator=(const thread_guard&) = delete;

  ~thread_guard()
  {
    if(t.joinable())
```

```
    {
      t.join();
    }
  }
};
```

```
std::thread t2(background_task{});
  thread_guard tg(t2);
```

When the guard goes out of scope, join is called, this works even with the exception.

Passing arguments to thread function

When detached is called, the ownership of the thread is handed over to the operating systen. It’s not possible to join it after detatch was called.

```
void f(int i, const std::string& s)
{
  std::stringstream ss;
  ss << i << ": " << s[20] << std::endl;
  const auto msg = ss.str();

    std::osyncstream sync_cout(std::cout);
  sync_cout << msg;
}
```

You can pass parameters to the tread function. If we just pass it in like this, it will only work if we don't terminate before the thread is at the end.

```
char buffer[1024];
buffer[20] = 'A';
std::thread t1(f, 1, buffer);
t1.detach();
```

This copies the char array into a string object the arguments for the function are variadic, and in this case resolved to pass by value. So no ownershipe problem here

```
std::thread t2(f, 2, std::string(buffer));
t2.detach();
```

If we wanted to pass by reference on purpose this becomes problematic. For instance:

```
struct NotCopyableObject {
  NotCopyableObject () {}
  NotCopyableObject (const NotCopyableObject &) = delete;
};
void f_with_not_movable(const NotCopyableObject& obj)
{}
```

This won't compile as you can't do a pass by reference it's alway's copied. (which kinda makes sense here).

```
std::thread t3(f_with_not_movable, obj);
```

Wrap it in a std::ref, and then it will work, there is however the lifetime problem again, so this time we join and we won’t detach.

```
std::thread t3(f_with_not_movable, std::ref(obj));
t3.join();
```

Transfering ownership of a thread

- std::thread objects are movable but no copyable
- If you move a thread inside a thread object that already has a thread associated → std::terminate

```
std::thread t; // <- thread object without thread
t = std::move(other_t);
```

The lifetime of a thread can be

```
class scoped_thread {
  std::thread t;
public:
  explicit scoped_thread(std::thread&& t_) :t(std::move(t_)) {
    if(!t.joinable())
    {
      throw std::logic_error("No thread");
```

```
    }
  }

  ~scoped_thread()
  {
    // We have ownership, so alway's join.
    t.join();
  }

  // no copy or assign
  scoped_thread(scoped_thread const&) = delete;
  scoped_thread& operator=(scoped_thread const&) = delete;

};
```

The C++20 standard defines the jthread class that works just like the scoped_tread, unless you detach it. The book shows an example of how to implement this, but left out as it’s part of the standard now. When detach is called upon, the jthread won’t join when leaving the scope.

```
// This thread will join by itself,
// unless you call the detach.
// C++ 20 feature
std::jthread t_joining(f_transfer);
// t_joining.detach(); // if you want to detach-> won't join when leaving scope.
```

Choosing the number of threads at runtime

The Standard library can find out how many hardware threads there are, this avoids **oversubscription** .When you use more software threads then there are hardware threads, putting extra load on the system.

```
std::thread::hardware_concurrency()
```

Identifying threads

Each thread is uniqueliy identified by it’s id, the type of the id is std::thread::id, it supports comparisons and std::hash.

There are 2 way’s of getting the thread id:

1. From the thread object

```
std::jthread t1(WaitAndPrintThreadId);

std::stringstream ss;
ss << "The id of the thread from the outside is: " << t1.get_id() << std::endl;
auto msg = ss.str();

std::osyncstream sync_cout(std::cout);
sync_cout << msg;
```

2. From within the thread by calling std::this_thread::get_id()

```
void WaitAndPrintThreadId()
{
  f_wait();
  std::stringstream ss;
  ss << "The id of the thread from the inside is: " << std::this_thread::get_id() << std::endl;
  auto msg = ss.str();

  std::osyncstream sync_cout(std::cout);
  sync_cout << msg;
}
```