C++ for High Performance Programming

2017 Compute Ontario Summer School (Waterloo)

Paul Preney, OCT, M.Sc., B.Ed., B.Sc.

preney@sharcnet.ca

School of Computer Science
University of Windsor
Windsor, Ontario, Canada

May 29, 2017

Presentation Overview

Introduction

Concurrency in C++:

- was introduced in C++11 (concurrently with C11) [1, 2]
- defines **memory models** providing a set of **guarantees** for **sequential/concurrent memory accesses**.
- supports **lock-free** programming (e.g., to avoid data races)
- supports `atomic`s
- supports `thread`s complete with locks, `condition_variable`s, and `mutex`es
- supports `future`s, `promise`s, `packaged_task`s, and `async`

Part I

Copying and Moving

Table of Contents

Overview

- C++98 supports object **copy** semantics.
- C++11 added object **move** semantics to the language.
- Moving an object is **potentially an optimized copy:**
  - $O(\text{moving data}) = O(\text{copying data})$
  - $\Omega(\text{moving data}) \neq \varnothing$
  - Often, $\Omega(\text{moving data}) = \mathit{cost}(\text{copying pointers to the data})$

Table of Contents

What is Copying?

Copying an object A to another object B involves:

1. Destroy any data in B.
2. Ensure B is capable of holding A's data.
3. Copy all data in A into B.

ASIDE: If an exception occurs, B's state is invalid.

What is Copying? (con't)

Assuming `swap()` is **`noexcept`**, copying an object A to another object B in an **exception-safe** manner involves:

1. Declare a temporary, T, capable of holding A's data.
2. Copy all data in A into T.
3. If no exception occurred, `swap(B,T);`.
4. Destroy T.

NOTE: This is a **waste** of time and RAM if one **no longer needs / will destroy** A after the copy!

Table of Contents

Copy and Move Elision

Copy/Move Elision

Should the cost of copying or moving ever appear to be zero, then **copy elision** was performed by the compiler **instead of** copying or moving objects multiple times. [2, §12.8, para. 31-32, [class.copy]]

Copy and Move Elision (con't)

From [3, §12.8, para. 31, [intro.memory]]:

- "When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the" constructor and destructor to be invoked "have side effects."
- When this is done, the extra invocations of constructor-destructor pairs are omitted (i.e., optimized away).

Benefit: Eliminates costly copying/moving of objects when returning from functions or throwing exceptions.

Copy Elision Example

To help programs fit better on slides, the following header file will be used:

output.hxx

```
#ifndef OUTPUT_HXX_
#define OUTPUT_HXX_
  #define OUT(S) std::cout << S << this << ' ' << '\n'
  #define OUT2(S,A) std::cout << S << this << ' ' << A << '\n'
#endif // #ifndef OUTPUT_HXX_
```

Copy Elision Example (con't)

copy-elision.cxx

```
#include <iostream>
#include "output.hxx"
class A {
  public:
    A() { OUT("A()␣"); }
    A(A const& b) { OUT2("A(copy)␣",&b); }
    ~A() { OUT("~A()␣"); }
};

A a_function() {
  A a; // default construct A
  return a; // return copy of A
}

A value = a_function(); // Copy elision possible here.

int main() { }
```

Copy Elision Example (con't)

copy-elision-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./copy-elision.exe
A() 0x601171
~A() 0x601171
$
```

Copy/Move Elision Criteria

Elision can be performed by the compiler in these circumstances:

- in a function's `return` statement with a class return type when "the expression is the name of a non-volatile automatic object (other than a function or catch-clause parameter) with the same cv-unqualified type as the function return type"
- "when a temporary class object that has not been bound to a reference (12.2) would be copied/moved to a class object with the same cv-unqualified type"
- in certain *throw-expression*, and,
- in certain *exception-declarations* of an exception handler.

[3, §12.8, para. 31-32, [intro.memory]]

Table of Contents

Object Copy Semantics

C++98 supports user-defined object copy semantics by:

- defining a **copy constructor**, and,
- defining a **copy assignment operator**.

Object Copy Semantics (con't)

The copy constructor and copy assignment operator are both **required** to have a **single argument** that is a **const lvalue reference** to the object type.

e.g., Given the type `Foo`, the argument type must be `Foo const&`.

NOTE: Function arguments that are `Foo const&` in C++ accept both lvalue and rvalue `Foo` objects.

Copy Semantics Example 1

copy.cxx

```
#include <iostream>
#include "output.hxx"
class A {
  public:
    A() { OUT("A() "); }
    A(A const& b) { OUT2("A(copy) ",&b); }
    A& operator =(A const& b) { OUT2("a=b;(copy) ",&b); return *this; }
    ~A() { OUT("~A() "); }
};

int main() {
  A a; // default constructed
  A b = a; // copy constructed
  a = b; // copy assignment
}
```

Copy Semantics Example 1 (con't)

copy-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./copy.exe
A() 0x7ffd28106bd0
A(copy) 0x7ffd28106bd1 0x7ffd28106bd0
a=b;(copy) 0x7ffd28106bd0 0x7ffd28106bd1
~A() 0x7ffd28106bd1
~A() 0x7ffd28106bd0
$
```

Copy Semantics Example 2

copy-large.cxx

```
#include <iostream>
#include <algorithm>
using namespace std;
class A
{
  double *ptr;
  public:
    A() : ptr(new double[100]) { cout << "A() " << this << '\n'; }
    A(A const& b) : ptr (new double[100]) {
      cout << "A(copy) " << this << '\n';
      std::copy(b.ptr, b.ptr+100, ptr);
    }
    A& operator =(A const& b) {
      cout << "a=b;(copy) " << this << '\n';
      std::copy(b.ptr, b.ptr+100, ptr);
      return *this;
    }
```

Copy Semantics Example 2 (con’t)

```
    ~A() {
      cout << "~A()␣" << this << '\n';
      delete[] ptr;
    }
    A operator +(A const& b) const {
      cout << "a+b;␣" << this << '\n';
      A retval(*this); // make copy every invocation
      for (int i{}; i != 100; ++i)
        retval.ptr[i] += b.ptr[i];
      return retval;
    }
};

int main() {
  A a, b, c;
  c = a + b + b;
}
```

Copy Semantics Example 2 (con't)

copy-large-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./copy-large.exe
A() 0x7ffec337aa20
A() 0x7ffec337aa30
A() 0x7ffec337aa40
a+b; 0x7ffec337aa20
A(copy) 0x7ffec337aa50
a+b; 0x7ffec337aa50
A(copy) 0x7ffec337aa60
a=b;(copy) 0x7ffec337aa40
~A() 0x7ffec337aa60
~A() 0x7ffec337aa50
~A() 0x7ffec337aa40
~A() 0x7ffec337aa30
~A() 0x7ffec337aa20
$
```

Table of Contents

Object Move Semantics

C++11 introduced the ability to move user-defined objects by:

- defining a **move constructor**, and,
- defining a **move assignment operator**.

Object Move Semantics (con't)

The move constructor and move assignment operator are both **required** to have a **single argument** that is an **(non-const) rvalue reference** to the object type.

e.g., Given the type `Foo`, the argument type must be `Foo&&`.

NOTE: Function arguments that are `Foo&&` in C++ can only accept rvalue `Foo` objects.

What is Moving?

Briefly:

- Copying A to B **copies the data** in A into B.
- Moving A to B **moves the data** in A into B.

What is Moving? (con't)

Clearly:

- All fundamental types (e.g., `int`, `double`) will always be copied.
- While pointer types (e.g., `int*`, `Foo*`) can only have their pointer values copied, what they point to does not necessarily need to be copied!

What is Moving? (con't)

References are conceptually equivalent to **`const`** pointers:

Pointer Declaration	Reference Declaration
`T*`	n/a
`T const*`	n/a
`T * const`	`T&`
`T const * const`	`T const&`

What is Moving? (con't)

Additionally, since the C++ standard states, "It is unspecified whether or not a reference requires storage (3.7)." [3, §8.3.2, para. 4, [idcl.ref]] the following also holds:

Type	Reference To Type
`T`	`T&&`
`T const`	`T const&&`

What is Moving? (con't)

So reference types **referring to lvalues**:

- will need to be copied if `const` or otherwise immutable
- could be moved if an **rvalue**, or, **cast** to an **rvalue** using `std::move()` or `std::forward<T>()`.

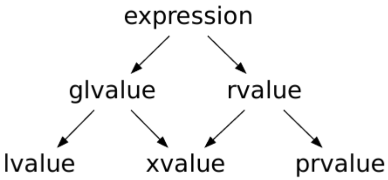
Lvalues and Rvalues

Excluding when explicit casts are used, in practice:

- an **lvalue** is a value/variable that **has a user-defined name**
  - `int i = 5;`
  - `int& lvalue_ref = i;`
  - `int&& rvalue_ref = 5;` is an lvalue that refers to an rvalue
- an **rvalue** is a value/variable that **does not have a user-defined name**
  - `5` is a literal value
  - `int(5)` is a variable with no name

Lvalues and Rvalues (con't)

More formally, C++ defines a taxonomy in [3, §3.10, Figure 1, [basic.lval]]:



Lvalues and Rvalues (con't)

From [3, §3.10, [basic.lval]]:

- "An *lvalue* […] designates a function or an object."
- "An *xvalue* (an 'eXpiring' value) also refers to an object, usually near the of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references (8.3.2)."
- "An *glvalue* ('generalized' lvalue) is an lvalue or an xvalue."
- "An *rvalue* […] is an xvalue, a temporary object (12.2) or subobject thereof, or a value that is not associated with an object."
- "A *prvalue* ('pure' rvalue) is an rvalue that is not an xvalue."
  - e.g., the value of a literal such as `8.23e-1`

Object Move Semantics

Moving an object *A* into another object *B* involves:

1. If *A* isn't going to be immediately destroyed, or, if data leakage and/or security issues are a concern, then reset/zero/null out *B*'s data.
2. `swap(A,B);`

NOTE: If this involves pointers to data structures, moving is much simpler than copying since it involves only copying and nulling-out pointers!

Move Semantics Example 1

move.cxx

```
#include <iostream>
#include "output.hxx"
class A {
public:
  A() { OUT("A()␣"); }
  A(A const& b) { OUT2("A(copy)␣",&b); }
  A(A&& b) { OUT2("A(move)␣",&b); }
  A& operator =(A const& b) { OUT2("a=b;(copy)␣",&b); return *this; }
  A& operator =(A&& b) { OUT2("a=b;(move)␣",&b); return *this; }
  ~A() { OUT("~A()␣"); }
};
int main() {
  A a, b; // default constructions
  A c = A{}, d{std::move(a)}, e = std::move(b); // move constructions
  a = b; // copy assignment
  a = std::move(b); a = A{}; // move assignment
}
```

Move Semantics Example 1 (con't)

move-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./move.exe
A() 0x7ffd6b677460
A() 0x7ffd6b677461
A() 0x7ffd6b677462
A(move) 0x7ffd6b677463 0x7ffd6b677460
A(move) 0x7ffd6b677464 0x7ffd6b677461
a=b;(copy) 0x7ffd6b677460 0x7ffd6b677461
a=b;(move) 0x7ffd6b677460 0x7ffd6b677461
A() 0x7ffd6b677465
a=b;(move) 0x7ffd6b677460 0x7ffd6b677465
~A() 0x7ffd6b677465
~A() 0x7ffd6b677464
~A() 0x7ffd6b677463
~A() 0x7ffd6b677462
~A() 0x7ffd6b677461
~A() 0x7ffd6b677460
$
```

Move Semantics Example 2

In this example:

- `class A` stores a `double` array of length size.
- If `operator +(A,A)` is invoked the sum of the two arrays is computed.
- If `A(length)` is constructed, the array size is set to length with all initial values set to zero.
- If `A(A,length)` is constructed, the array is copied and length is set to the maximum of the array size or length.
- Note the use of lvalues and rvalues with `operator +()`.

Move Semantics Example 2 (con't)

move-large.cxx

```cpp
#include <iostream>
#include <algorithm>
#include <utility>
#include <initializer_list>
class A
{
  std::size_t size_;
  double *ptr_;

  void size_adjust(A const& b)
  {
    if (size_ < b.size_)
    { // Extend *this to be as large as b...
      std::cout << "size_adjust() " << this << '\n';
      A tmp{b, b.size_}; swap(tmp);
    }
    // else *this is same or larger than b...
  }
```

Move Semantics Example 2 (con't)

```
public:
  void swap(A& b) noexcept
  {
    std::swap(ptr_, b.ptr_);
    std::swap(size_, b.size_);
  }

  std::size_t size() const noexcept
  {
    return size_;
  }

  double& operator [](std::size_t i) const noexcept
  {
    return ptr_[i];
  }
```

Move Semantics Example 2 (con't)

```
A() :
  size_{},
  ptr_(nullptr)
{
  std::cout << "A()␣" << this << '\n';
}

A(std::size_t sz) :
  size_{sz},
  ptr_{new double[sz]}
{
  std::cout << "A(" << sz << ")␣" << this << '\n';
}

~A()
{
  std::cout << "~A()␣" << this << '\n';
  delete[] ptr_;
}
```

Move Semantics Example 2 (con’t)

```
A(std::initializer_list<double> il) :
  size_{il.size()},
  ptr_{new double[il.size()]}
{
  std::cout << "A(init_list:" << size_ << ")␣" << this << '\n';
  std::copy(il.begin(), il.end(), ptr_);
}

A(A const& b, std::size_t sz) :
  size_{std::max(b.size_,sz)},
  ptr_(new double[size_])
{
  std::cout << "A(copy," << sz << ")␣" << this << '␣' << &b << '\n';

  std::copy(b.ptr_, b.ptr_+b.size_, ptr_); // copy data
  std::fill_n(ptr_, size_-b.size_, double{}); // zero data
}
```

Move Semantics Example 2 (con’t)

```
A(A const& b) :
  size_{b.size_}, ptr_(new double[b.size_])
{
  std::cout << "A(copy)␣" << this << '␣' << &b << '\n';
  std::copy(b.ptr_, b.ptr_+b.size_, ptr_); // copy data
}

A& operator =(A const& b) {
  std::cout << "a=b;(copy)␣" << this << '␣' << &b << '\n';
  if (size_ != b.size_) {
    A tmp{b}; swap(tmp); // copy and swap
  } else {
    size_adjust(b); // adjust size if needed
    std::copy(b.ptr_, b.ptr_+b.size_, ptr_); // copy data
    size_ = b.size_;
  }
  return *this;
}
```

Move Semantics Example 2 (con't)

```
A(A&& b) :
  size_{std::move(b.size_)}, // move size
  ptr_{std::move(b.ptr_)} // move pointer
{
  std::cout << "A(move)␣" << this << '␣' << &b << '\n';

  b.size_ = {}; // zero out original size
  b.ptr_ = nullptr; // null out original pointer
}

A& operator =(A&& b)
{
  std::cout << "a=b;(move)␣" << this << '␣' << &b << '\n';
  swap(b); // swap
  return *this;
}
```

Move Semantics Example 2 (con't)

```
  A operator +(A const& b) const
  {
    std::cout << "a+b;(const)␣" << this << '␣' << &b << '\n';

    A retval{*this, std::max(size_, b.size_)}; // copy *this
    for (std::size_t i{}; i != b.size_; ++i)
      retval.ptr_[i] += b.ptr_[i];
    return retval;
  }

  A operator +(A&& b) const {
    std::cout << "a+b;(rvalue1)␣" << this << '␣' << &b << '\n';
    b.size_adjust(*this); // adjust size if needed

    for (std::size_t i{}; i != size_; ++i)
      b.ptr_[i] += ptr_[i]; // Add *this.ptr to b.ptr!
    return std::move(b); // Move b into return value
  }
};
```

Move Semantics Example 2 (con't)

```
A operator +(A&& a, A const& b) {
  std::cout << "a+b;(rvalue2) " << &a << ' ' << &b << '\n';
  return b+std::move(a);
}

A operator +(A&& a, A&& b) {
  std::cout << "a+b;(rvalue3) " << &a << ' ' << &b << '\n';
  return a+std::move(b);
}

int main() {
  A result = A(5) + A{0.0, 1.1, 2.2, 3.3, 4.4};
}
```

Move Semantics Example 2 (con't)

move-large-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./move-large.exe
A(init_list:5) 0x7ffd955da0b0
A(5) 0x7ffd955da0a0
a+b;(rvalue3) 0x7ffd955da0a0 0x7ffd955da0b0
a+b;(rvalue1) 0x7ffd955da0a0 0x7ffd955da0b0
A(move) 0x7ffd955da090 0x7ffd955da0b0
~A() 0x7ffd955da0a0
~A() 0x7ffd955da0b0
~A() 0x7ffd955da090
$
```

When To Use std::move() and std::forward<T>()

Use `std::move()` when non-**`const`** lvalue needs to be moved.

Use `std::forward<T>(value)` when `value`'s type can be an lvalue or rvalue reference when `value`'s is a template parameter.

`std::forward<T>(value)` when used with templates and the special function argument pattern `T&&` where `T` is a template parameter enables the **perfect forwarding** of function arguments.

std::forward<T>() Example

std-forward.cxx

```cpp
#include <cmath>
#include <iostream>
#include "output.hxx"
using namespace std;

struct Foo
{
  double d_;

  // Permit initialization of Foo instance with a double...
  explicit Foo(double d) : d_{d} {
    cout << "Foo(" << d_ << ") " << this << '\n';
  }

  // Permit implicit cast to double...
  operator double() const { return d_; }
```

std::forward<T>() Example (con't)

```
  // Constructors, assignment operators, and destructors w/outputs...
  Foo() : d_{} { OUT("Foo()␣"); }
  Foo(Foo const& f) : d_{f.d_} { OUT2("Foo(copy)␣",&f); }
  Foo(Foo&& f) : d_{f.d_} { OUT2("Foo(move)␣",&f); }
  Foo& operator =(Foo const& f) {
    OUT2("f=g;(copy)␣",&f);
    d_ = f.d_; return *this;
  }
  Foo& operator =(Foo&& f) {
    OUT2("f=g;(move)␣",&f);
    d_ = f.d_; return *this;
  }
  ~Foo() { OUT("~Foo()␣"); }
};
```

std::forward<T>() Example (con't)

```
template <typename Op, typename Arg>
constexpr auto simple_proxy(Op&& op, Arg&& arg) {
  return op(std::forward<Arg>(arg));
}
template <typename Op, typename... Args>
constexpr auto fancy_proxy(Op&& op, Args&&... args) {
  return op(std::forward<Args>(args)...);
}
constexpr double my_func(double a, double b) { return a+b; }

int main() {
  cout << "sp:\n";
  cout << simple_proxy<double(double)>(std::sin, Foo{0.0}) << "\n";
  cout << "\nfp:\n";
  cout << fancy_proxy(my_func, Foo{1.1}, Foo{2.2}) << '\n';
}
```

std::forward<T>() Example (con't)

std-forward-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./std-forward.exe
sp:
Foo(0) 0x7ffd31f9a180
0
~Foo() 0x7ffd31f9a180

fp:
Foo(2.2) 0x7ffd31f9a180
Foo(1.1) 0x7ffd31f9a170
3.3
~Foo() 0x7ffd31f9a170
~Foo() 0x7ffd31f9a180
$
```

Part II

Using Concurrency Constructs

Table of Contents

Overview

- Every C++ program has at least one thread
  - i.e., the thread that runs `main()`
- Additional threads can be created.
  - You must provide a function / function object entry point.
  - When such has finished executing the thread exits.
- Nearly the entire C++ Standard Library is not thread-safe.
  - Essentially only concurrency features are safe.
  - e.g., use a lock with `std::mutex` to perform output with `std::cout`.

Table of Contents

std::thread

`std::thread` is a low-level construct:

- It enables one to create a thread of execution through construction.
- After creation, instances **must** either invoke `detach()` or `join()`.
  - `join()` requires the caller to wait for the thread to end
  - `detach()` runs the thread and the caller must never wait to end (via `join()`)
- Its interface has no facilities to process the thread's result.
- If an uncaught exception occurs in the thread, the program immediately aborts.
- If a (non-detached) thread variable goes out of scope without `join()` being called, an exception is thrown.
- When `main()` ends, all **detach**ed threads are abruptly aborted.

std::thread (con't)

- Move operations and `swap()` are permitted; copy semantics are not allowed.
- One can obtain a thread's ID via `get_id()`.
- One can obtain the number of concurrent threads the hardware supports via `hardware_concurrency()`.

A First Thread Example

thread-hello.cxx

```
#include <iostream>
#include <random>
#include <chrono>
#include <thread>
#include <vector>
#include <mutex>
```

A First Thread Example (con't)

```
std::mutex cout_mutex;

void hello(std::random_device::result_type seed, int i)
{
  { std::lock_guard<std::mutex> guard(cout_mutex);
    std::cout << "Hello " << i << ", id=" << std::this_thread::get_id() << '\n';
  }

  std::default_random_engine dre(seed);
  std::uniform_int_distribution<std::size_t> ud(500,1000);
  std::this_thread::sleep_for(std::chrono::milliseconds(ud(dre)));

  { std::lock_guard<std::mutex> guard(cout_mutex);
    std::cout << "Bye " << i << ", id=" << std::this_thread::get_id() << '\n';
  }
}
```

A First Thread Example (con't)

```
int main()
{
  std::vector<std::thread> v;

  std::random_device rd;
  for (unsigned int i{}; i != std::thread::hardware_concurrency(); ++i)
    v.push_back(std::thread(hello,rd(),i));

  for (auto& e : v)
    e.join();
}
```

A First Thread Example (con't)

thread-hello-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./thread-hello.exe
Hello 0, id=139652946458368
Hello 2, id=139652929672960
Hello 1, id=139652938065664
Hello 3, id=139652921280256
Bye 2, id=139652929672960
Bye 1, id=139652938065664
Bye 3, id=139652921280256
Bye 0, id=139652946458368
$
```

Ugh!

Unless you are writing a library, `std::thread` is very likely too low-level to be of practical use!

Ideally one wants to be able to data to, from, and amongst threads and also to be able to handle exceptions should they occur.

Although you can pass arguments to the thread function via `std::thread`'s constructor arguments, `std::promise` is a high-level construct that does this and is also capable of handling exceptions.

std::promise

`std::promise<T>` allows one to **store a value of type T or an exception** that will later be acquired (asynchronously) by a `std::future` object (created by the `std::promise`).

- Move operations and `swap()` are permitted; copy semantics are not allowed.
- One can obtain the `std::future` value associated with the `std::promise` promised result via `get_future()`.
- One can set the promised result via one of these calls:
  - `set_value()` (sets value)
  - `set_value_at_thread_exit()` (sets value when thread exits)
  - `set_exception()` (sets exception)
  - `set_exception_at_thread_exit()` (sets exception when thread exits)

A std::promise Example

thread-promise.cxx

```
#include <mutex>
#include <future>
#include <thread>
#include <exception>
#include <stdexcept>
#include <string>
#include <iostream>

std::mutex io_mutex;
```

A std::promise Example (con't)

```
void read_data(std::promise<std::string>& p) {
  try {
    char c; std::string retval;
    { std::lock_guard<std::mutex> guard(io_mutex);
      std::cout << "enter␣char␣or␣'e'␣for␣exception:␣";
      c = std::cin.get();
    }
    if (std::cin) {
      if (c != 'e') retval += c;
      else throw std::runtime_error(std::string("read_data␣exception!"));
    }
    retval = std::string("read␣char␣") + c;
    p.set_value(std::move(retval)); // Set promise!
  }
  catch (...) {
    p.set_exception(std::current_exception()); // Set promise!
  }
}
```

A std::promise Example (con't)

```
int main()
{
  try {
    std::promise<std::string> p;
    std::thread t(read_data,std::ref(p));
    t.detach(); // okay since we will wait for the promise below

    std::future<std::string> f(p.get_future()); // Ask for future value/exception
    std::cout << "result: " << f.get() << '\n'; // Retrieve future value/exception
  }
  catch (const std::exception& e) {
    std::cerr << "EXCEPTION: " << e.what() << '\n';
  }
  catch (...) {
    std::cerr << "EXCEPTION\n";
  }
}
```

A std::promise Example (con't)

thread-promise-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./thread-promise.exe
enter char or 'e' for exception: result: read char c
$ ./thread-promise.exe
enter char or 'e' for exception: EXCEPTION: read_data exception!
$
```

std::future and std::shared_future

`std::future<T>` and `std::shared_future<T>` allows one to have a variable to a **future** value of type T or an exception.

- `std::future` only allows the value/exception to be retrieved (e.g., using `get()`) exactly once.
- `std::shared_future` allows the value/exception multiple times.
- Important member functions include:
  - `valid()` returns true if the future has a valid value/exception
  - `get()` blocks until the operation is done, then returns the value (or re-throws the exception) and if `std::future` it invalidates its state
  - `wait()` blocks until the operation is done
  - `wait_for(t)` blocks for time t or until the operation is done
  - `wait_until(t)` blocks until timepoint t or until the operation is done
  - `share()` returns a `std::shared_future` with the current state and invalids this `std::future` (`std::future` only)

Table of Contents

std::async

A much easier way to start a new thread is to use `std::async()`.

The `std::async(f,args...)` function:

- Allows one to execute the function call `f(args...)` according to a launch policy
    - The default policy is if `async()` didn't start the thread immediately, it will defer the call until the outcome is requested (e.g., `get()`).
    - The launch policy can be explicitly specified by passing in the launch policy first, e.g., `std::async(std::launch::deferred,f,args...)`.
    - With `std::launch::deferred`, the task is executed on the **calling thread** when its result (e.g., via `std::future::get()`) is requested.
    - With `std::launch::async`, a new thread is launched asynchronously or `std::system_error` is thrown.
- Returns a `std::future<T>` where `T` is same type that the call `f(args...)` returns.

A std::async Example 1

thread-async.cxx

```
#include <iostream>
#include <thread>
#include <future>

double func(double a, double b) {
  return a + b;
}

int main()
{
  std::future<double> fut = std::async(func,1.1,2.2);
  std::cout << fut.get() << '\n'; // get the result
}
```

A std::async Example 1 (con't)

thread-async-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./thread-async.exe
3.3
$
```

A std::async Example 2

thread-async2.cxx

```
#include <iostream>
#include <random>
#include <chrono>
#include <thread>
#include <future>

void slow(std::random_device::result_type seed) {
  std::default_random_engine dre(seed);
  std::uniform_int_distribution<std::size_t> ud(500,1000);
  std::this_thread::sleep_for(std::chrono::milliseconds(ud(dre)));
  return;
}
```

A std::async Example 2 (con't)

```
int func1(std::random_device::result_type seed, int i) {
  slow(seed);
  return i*2;
}

double func2(std::random_device::result_type seed, int i) {
  slow(seed);
  return i*3.2;
}

int main()
{
  std::random_device rd;
  std::future<int> one = std::async(std::launch::async,func1,rd(),1);
  std::future<double> two = std::async(std::launch::async,func2,rd(),2);
  std::cout << one.get() << ',' << two.get() << '\n';
}
```

A std::async Example 2 (con't)

thread-async2-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./thread-async2.exe
2,6.4
$
```

Table of Contents

std::packaged_task< >

With `std::thread` and `std::async()` you are committing the arguments to the function-to-be-executed being specified when the `std::thread` object is created or when the `std::async()` call is made!

What if you have a function that you want to pass arguments to at some later point in time?

This is what the `std::packaged_task<T>` object is for!

A std::package_task< > Example

thread-package-task.cxx

```
#include <iostream>
#include <thread>
#include <future>

double func(double a, double b) {
  return a + b;
}

int main()
{
  std::packaged_task< double(double, double) > task(func);
  std::future<double> fut = task.get_future(); // get the future
  // ... later ...
  task(1.1, 2.2); // invoke the task
  // ... later ...
  std::cout << fut.get() << '\n'; // get the result
}
```

A std::package_task< > Example (con't)

thread-package-task-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./thread-package-task.exe
3.3
$
```

Table of Contents

Mutexes and Locks

Instances of various types of mutexes are used by various "lock" classes to enable various types of **concurrent mutual exclusion** behaviours.

C++ supports the following mutex types:

- `std::mutex`: can be locked once, only by one thread at a time
- `std::recursive_mutex`: allows multiple locks at the same time by the same thead
  - e.g., very useful for recursive functions
- `std::time_mutex`: like `std::mutex` but also permits one to pass a duration or timepoint for how long it attempts to acquire a lock
- `std::recursive_timed_mutex`: like `std::recursive_mutex` but also permits one to pass a duration or timepoint for how long it attempts to acquire a lock

SHARCNET™

Mutexes and Locks (con't)

The mutex types have the following operations (if applicable):

- Default construction. Creates an unlocked mutex.
- Destruction. Destroys the mutex if unlocked, otherwise, behaviour is undefined.
- `lock()` blocks for the lock and then locks the mutex
- `try_lock()` attempts to lock the mutex (returning **`true`** if successful)
- `try_lock_for(t)` attempts to lock the mutex for duration t (returning lstinlinetrue if successful)
- `try_lock_for(t)` attempts to lock the mutex until timepoint t (returning **`true`** if successful)
- `unlock()` unlocks the mutex —undefined behaviour if not locked.

NOTE: Typically one does not use these mutex types directly. Instead one uses one of the lock classes to ensure a locked mutex is always freed.

Mutexes and Locks (con't)

C++ has three types of lock classes: `std::lock_guard` `std::unique_lock`, and `std::shared_lock`.

`std::lock_guard` is often sufficient. It has the following members:

- `lock_guard var(some_mutex)` creates a `lock_guard` for `some_mutex` and locks it
- `lock_guard var(some_mutex,adopt_lock)` creates a `lock_guard` for `some_mutex` which is already locked
- Its destructor will ensure the mutex is unlocked.

Mutexes and Locks (con't)

C++ has a special variadic `std::lock()` function that will lock **all** of the provided objects using a deadlock avoidance algorithm, or, none of them won't be locked.

A N-mutex std::lock() Example

thread-nlocks.cxx

```
#include <iostream>
#include <thread>
#include <mutex>
#include <algorithm>

struct entity {
  std::mutex m_; // for locking
  double d; // data
};

void swap_entities(entity& a, entity& b) {
  std::lock(a.m_, b.m_); // Acquire both locks
  std::lock_guard<std::mutex> lga(a.m_, std::adopt_lock);
  std::lock_guard<std::mutex> lgb(b.m_, std::adopt_lock);
  std::swap(a.d, b.d); // do processing
}

```

A N-mutex std::lock() Example (con't)

```cpp
int main() {
  entity a, b, c; a.d = 1.1; b.d = 2.2; c.d = 3.3;
  std::thread t1(swap_entities, std::ref(a), std::ref(b));
  std::thread t2(swap_entities, std::ref(a), std::ref(c));
  t1.join(); t2.join();
  std::cout << a.d << ' ' << b.d << ' ' << c.d << '\n';
}
```

A N-mutex std::lock() Example (con't)

thread-nlocks-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./thread-nlocks.exe
2.2 3.3 1.1
$
```

A std::call_once Example

thread-call-once.cxx

```
#include <iostream>
#include <thread>
#include <mutex>

std::once_flag one_time;
std::mutex cout_mutex;

void msg(const char* str) {
  std::lock_guard<std::mutex> guard(cout_mutex);
  std::cout << std::this_thread::get_id() << ":␣" << str << '\n';
}

void do_only_once() {
  std::call_once(one_time, [](){ msg("Do␣this␣only␣once!"); });
  msg("Do␣this␣multiple␣times.");
}

```

A std::call_once Example (con't)

```
int main() {
  std::thread t1(do_only_once), t2(do_only_once), t3(do_only_once);
  t1.join(); t2.join(); t3.join();
}
```

A std::call_once Example (con't)

thread-call-once-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./thread-call-once.exe
139644086630144: Do this only once!
139644086630144: Do this multiple times.
139644078237440: Do this multiple times.
139644069844736: Do this multiple times.
$
```

A std::condition_variable Example

thread-condvar.cxx

```
#include <iostream>
#include <future>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;

bool can_consume = false;
bool can_process = false;
double data;
```

A std::condition_variable Example (con't)

```
void func() {
  std::unique_lock<std::mutex> ul(m);
  cv.wait(ul, []{ return can_consume; });

  data *= 2; // lock obtained, process the data

  can_process = true;
  ul.unlock();
  cv.notify_one();
}
```

A std::condition_variable Example (con't)

```
int main() {
  std::thread t(func);
  data = 4.2;
  { std::lock_guard<std::mutex> lg(m);
    can_consume = true;
    std::cout << "can_consume!\n";
  }
  cv.notify_one();
  { std::unique_lock<std::mutex> ul(m);
    cv.wait(ul, []{ return can_process; });
    std::cout << "can_process!\n";
  }
  std::cout << "data␣=␣" << data << '\n';
  t.join();
}
```

A std::condition_variable Example (con't)

thread-condvar-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./thread-condvar.exe
can_consume!
can_process!
data = 8.4
$
```

Table of Contents

Atomics

Condition variable code using boolean “ready” flag variables still need to make use of a mutex!

Why?

- In general, reading and writing of data is **not** atomic.
- Compiler-generated code can **change the order** operations occur.
- Modern chips can also perform certain types of operation reordering involving the machine opcodes.

NOTE: Your code is likely not doing things exactly the way you think! Mutexes and locks are a solution to these issues —except they have significant overheads.

`std::atomic` provides a type of low-overhead locking.

Earlier Condition Variable Example Using std::atomic

atomic-condvar.cxx

```
#include <iostream>
#include <future>
#include <mutex>
#include <condition_variable>

std::atomic<bool> can_consume{false};
std::atomic<bool> can_process{false};
double data;
```

Earlier Condition Variable Example Using std::atomic (con't)

```
void func() {
  while (!can_consume.load()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(150));
  }

  data *= 2; // lock obtained, process the data

  can_process.store(true);
}
```

Earlier Condition Variable Example Using std::atomic (con't)

```
int main() {
  std::thread t(func);
  data = 4.2;
  can_consume.store(true);
  std::cout << "can_consume!\n";
  while (!can_process.load()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(150));
  }
  std::cout << "can_process!\n";
  std::cout << "data = " << data << '\n';
  t.join();
}
```

Earlier Condition Variable Example Using std::atomic (con't)

atomic-condvar-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./atomic-condvar.exe
can_consume!
can_process!
data = 8.4
$
```

Part III

Exercises

Table of Contents

Download And Compile Code Presented In This Lecture

Download, compile, and run the codes presented in this lecture!

- `https://preney.ca/sharcnet/ss2017/cxx-for-hpc.zip`

Important:

- If using your own computer, it is okay to use `std::thread::hardware_concurrency()` threads.
- If using SHARCNET / Compute Canada facilities, hard-code the number of threads or pass in such from the command line to match your job submission parameters.

Table of Contents

Computing The Definite Integral Of f(x)

Compute the definite integral of some $f(x)$ in parallel threads.

Serial code for (inefficiently) computing π is provided.

Important:

- If using your own computer, it is okay to use `std::thread::hardware_concurrency()` threads.
- If using SHARCNET / Compute Canada facilities, hard-code the number of threads or pass in such from the command line to match your job submission parameters.

Definite Integral Serial Code

exercise-serial-defint.cxx

```
#include <cmath>
#include <limits>
#include <iostream>
#include <iomanip>

using UInt = unsigned int;
using Real = double;
```

Definite Integral Serial Code (con't)

```
template <typename Op>
Real definite_integral(Real a, Real b, UInt n, Op op)
{
  Real width = b - a; // Width of the entire interval
  Real delta_x = width / n; // Width of each subdivision's rectangle
  Real sum = 0.0; // Start the sum at 0.
  for (UInt i=0; i<n; ++i) // Iterate from [0,n)
  {
    Real x = a + (i+0.5) * delta_x; // Compute the midpoint of current
         rectangle
    Real area = delta_x * op(x); // Apply op(x)
    sum += area; // And accumulate the area.
  }
  return sum;
}
```

Definite Integral Serial Code (con't)

```
int main()
{
  using namespace std;

  Real pi =
    definite_integral(
      0.0, 1.0, 100000000,
      [](Real x) -> Real { return 1.0 / (1.0 + x*x); }
    ) * 4;
```

Definite Integral Serial Code (con't)

```
  // Ensure numbers are not written in scientific notation...
  cout.unsetf(ios_base::floatfield);
  cout.setf(ios_base::fixed, ios_base::floatfield);

  cout
    << "pi = "
    << setw(numeric_limits<Real>::max_digits10)
    << setprecision(numeric_limits<Real>::max_digits10)
    << pi
    << " (error = "
    << setw(numeric_limits<Real>::max_digits10)
    << setprecision(numeric_limits<Real>::max_digits10)
    << abs(pi - 3.14159265358979323846264338)
    << ")\n"
  ;
}
```

Definite Integral Serial Code (con't)

exercise-serial-defint-output.txt

```
$ g++-6.3.0 -std=c++14 -O3 -Wall -Wextra -Wpedantic -pthread
$ ./exercise-serial-defint.exe
pi = 3.14159265358936191 (error = 0.00000000000043121)
$
```

Part IV

References

References

[1] ISO/IEC. *Information technology – Programming languages – C.* ISO/IEC 9899-2011, IDT. Geneva, Switzerland, 2012 (cit. on p. 3).

[2] ISO/IEC. *Information technology – Programming languages – C++.* ISO/IEC 14882-2011, IDT. Geneva, Switzerland, 2012 (cit. on pp. 3, 11).

[3] ISO/IEC. *Information technology – Programming languages – C++.* ISO/IEC 14882-2014, IDT. Geneva, Switzerland, 2014 (cit. on pp. 12, 16, 31, 34, 35).