Modern C++ Exercises

The starter code for this course's exercises is in the **$\Exercises\Exercises.sln** solution (some exercises do not have starter code, in which case you will have to create a new project). Each exercise is in a separate project.

Make sure you use the most recent version of Visual Studio (Visual Studio 2015 at the time of writing) to complete the exercises. If you do not have access to a new version of Visual Studio, you can try using an online modern C++ compiler, such as Ideone (http://ideone.com). Alternatively, you can use the Code Blocks IDE (http://www.codeblocks.org), which is cross-platform, free, and supports the gcc and clang compilers on Windows, OS X, and Linux.

C++ 11/14 Productivity Features

Exercise 1 – `auto` and Range-Based `for` Loop

Open the **Exercise1** project and follow the TODO instructions in the code. You will need to use a couple of STL algorithms and replace iterator loops with `auto` and range-based `for`.

Exercise 2 – Lambda Functions

Task 1

Open the **Exercise2** project and follow the TODO instructions in the code. You will need to use lambda functions for various tasks, including capturing external local variables.

Task 2

Can you write a lambda that calls itself recursively? Try it.

C++ 11 Move Semantics

Exercise 3 – Move Semantics

Task 1

Open the **Exercise3** project and follow the TODO instructions in the code. You will need to add move semantics to a number of classes and use `std::move`.

Task 2

Which of the following code examples illustrate a valid, reasonable use case for `std::move`? Explain why or why not.

Example 1

```
void replace_impl(std::unique_ptr<impl> ptr)
{
  impl_ = std::move(ptr);
}
```

Example 2

```
matrix operator+(matrix const& a, matrix const& b)
{
  matrix temp(a);
  temp.add(b);
  return std::move(temp);
}
```

Example 3

```
template <typename T, typename U>
T get_first(std::pair<T, U>& pair)
{
  return std::move(pair.first);
}
```

Example 4

```
class button : public control
{
public:
  button(button&& rhs) : control(std::move(rhs)) {}
  …more stuff…
};
```

STL in C++11

Exercise 4 – Smart Pointers

Open the **Exercise4** project and follow the TODO instructions in the code. You will need to identify several situations in which smart pointers are used or not used, and determine what the correct function signatures should be.

Microsoft Concurrency Runtime

Exercise 5 – Parallel Loops

Open the **Exercise5** project and follow the TODO instructions in the code. Parallelize the loop that goes over the files. Use `parallel_for_each` and a parallel container, such as `concurrent_vector`. Parallelize the code that sorts the results container.

Bonus: modify the code so it recursively enters sub-directories.

Exercise 6 – Asynchronous Tasks

Open the **Exercise6** project. The `get_weather_for_city` function returns a `task<http_response>` that represents an ongoing HTTP request to a weather server. Write code that uses the `when_any` convenience method to print out the weather in several cities. Obtain the response body using the `http_response::extract_json` method (which is also asynchronous) and then use the `print_weather` function to parse the JSON body.

Bonus: use Visual C++ 2015 resumable function support (with the `__await` keyword) instead of task continuations.

Advanced Templates

Exercise 7 – Experimenting with `enable_if` and Trait Detection

Extend the `print` framework presented in class so that it doesn't treat printable types as containers (you can find it in the **Exercise7** starter project). That is, if the type has an `operator<<` that can output it to an `ostream`, it isn't a container and should be printed directly. This is a useful feature because the `is_container` template presented in class considers string types to be containers, which means they are printed character-by-character.

Hint: implement an `is_printable<T>` template that determines whether a type T has an `operator<<` that outputs to an `ostream`. To do so, you can use the `decltype`-based approach presented in class. Modify the `is_container` template so that it rejects types that are printable.

Exercise 8 – Variadic Templates

Overload `operator<<` for tuples. You should print the tuple's elements member-wise, assuming of course that the elements have an overloaded `operator<<`.

One option is to use the `safe_printf` function developed in class, which is a variadic function template that prints any number of elements. What you will need then is a tool that takes a tuple and converts it to a list of its elements. You can build such a tool using the `std::make_index_sequence` template, as follows.

In the code below, the `std::make_index_sequence<N>` helper returns `std::index_sequence<0, 1, …, N-1>`. The indices 0, 1, …, N-1 can be used as a non-type parameter pack, and when passed to the `std::get` method, can extract the tuple's elements for you. For example, the following `apply` function calls a specified function (such as `safe_printf`) with all the tuple's elements.

```
template <typename Tuple, typename Function, size_t... Indices>
void apply(Tuple const& tup, Function fn, std::index_sequence<Indices...>)
{
  fn(std::get<Indices>(tup)...);
}

template <typename Tuple, typename Function>
void apply(Tuple const& tup, Function fn)
{
  apply(tup, fn, std::make_index_sequence<std::tuple_size<Tuple>::value>>{});
}
```

Another option is to build the necessary machinery from scratch. You'll need a variadic function template that "recurses" down on the index of the tuple element. You can express the index using the `std::integral_constant` class. Here's the skeleton structure of the variadic function template you need:

```
// Base case: do nothing
template <typename Tuple>
void print(std::ostream&, Tuple const&, std::integral_constant<int, -1>) {}

// "Recursive" case
template <typename Tuple, int I>
void print(std::ostream& os, Tuple const& tup,
           std::integral_constant<int, I>)
{
  …you fill this in…
}
```

You then need to invoke `print` with `std::integral_constant<int, N>` where N is the tuple size (you can use `std::tuple_size` to obtain it).

Exercise 9 – Variadic Templates and Template Meta-Programming

C++ 14 introduces another useful function for working with tuples: `get<T>`. Here's an example of how it works. Note that if the type doesn't appear exactly once in the tuple, the result should be a compilation error (not a runtime exception).

```
auto tup = std::make_tuple(4.0, "hello", 52);
std::get<double>(tup) = 3.0;
std::cout << std::get<int>(tup) << "\n";
```

If you already have a standard library that supports C++ 14, good for you, but we're still going to implement this facility ourselves. Here is the general sketch of a possible solution:

1. Build a class template `count` that you can invoke as follows: `count<T, T1, T2, …, Tn>::value` which returns the number of times T appears in the list of types that follow. This can be done by a simple specialization for `count<T>` and `count<T, Head, Tail...>`.
2. Build a class template `find` that you can invoke as follows: `find<T, T1, T2, …, Tn>::value` which returns the first index at which the type T appears in the list of types that follow. This can be done in a very similar way to `count`.

Now, the `get<T>` function can use these helpers to call the standard `get<I>` function:

```
template <typename T, typename... Ts>
T& get_by_type(std::tuple<Ts...>& tup)
{
  static_assert(count<T, Ts...>::value == 1, "T must appear exactly once");
  return get<find<T, Ts...>::value>(tup);
}
```

Exception Safety

Exercise 10 – Analyzing Exception Safety

Determine the exception safety level (none, basic, strong, no-throw) of each of the following functions, based on the provided assumptions regarding the other functions used in the code.

Function 1

```
A& A::operator=(A const& rhs)
{
  if (this != &rhs)
  {
    x_ = rhs.x_;  // x_ is of type std::shared_ptr<T>
  }
  return *this;
}
```

Function 2

```
// T's copy constructor is noexcept
template <typename T>
void commit_twice(set<T>& s, std::function<T(void)> factory)
{
  auto val = factory();
  s.insert(val);
  s.insert(val);
}
```

After determining this function's exception safety level, how would you improve it to obtain strong exception safety?

Function 3

```
template <typename T>
struct node
{
  T data;
  node *next, *prev;
  // a bunch of functions, but no explicitly specified destructor
};
void linked_list<T>::remove(node<T>* where)
{
  auto prev(where->prev), next(where->next);
  delete where;
  prev->next = next;
  next->prev = prev;
}
```

Function 4

In an attempt to cleverly share code between the copy constructor and the assignment operator, someone wrote the following assignment operator. After analyzing its exception safety, suggest how it can be improved (and, as a bonus, discuss additional problems with this kind of code).

```
SomeClass& SomeClass::operator=(SomeClass const& rhs)
{
  if (this != &rhs)
  {
    this->~SomeClass();        // destroy our object in-place
    new (this) SomeClass(rhs); // call copy constructor using "placement new"
  }
  return *this;
}
```

Bonus: Modernizing C++ Code

Exercise 11 – Modernizing C++ Code

In the **$\Exercises\Modernizing** folder, you will find a solution with a number of legacy C++ code segments gathered from various open source projects and other sources. Inspect them and try to apply modern C++ and STL features to modernize these samples, while adding type safety, readability, runtime safety, and other desirable characteristics.