Programming with Native Languages in the 21st Century

Track: Modern C++

Overview

In this track, you will learn about modern C++; specifically how to create safe and fast programs and easy-to-understand code by utilizing many new language and library features. You do not need to be an expert in "old" C++ to complete this track, but some up-front familiarity with C++ will certainly grease the machinery.

Project

We will be working with an implementation for the board game Go. Players alternate their turns laying down their stones (black or white) in an attempt to establish areas of control (territory) and to capture groups (chains) of the opponent's stones by completely surrounding them. If you haven't already, please take a moment to review the rules of the game on Wikipedia® which are available at:
http://en.wikipedia.org/wiki/Go_(game)#Rules

Preparation

1. You must have access to a compiler that complies with (most of) the C++11 standard (e.g. GCC 4.7 or Clang 3.1) and an implementation of the C++11 standard library (e.g. libstdc++ 4.7 or a recent version of libc++). ***IMPORTANT: The Microsoft Visual C++ 11 (Visual Studio 2012) compiler and standard library are not complete enough to use in this track. If you are limited to that set of tools, see the C++/CX and WinRT track.***

2. Download the workshop code at:
https://github.com/michaelbprice/21st_century_native_programming

Exercises

Exercise 1 - Getting Familiar With the Architecture

The implementation of Go has a fairly traditional object-oriented, model-view-controller architecture.

- **`Stone`** - Represents an individual stone, which a player places at a location on the board. A **`Stone`** has an assigned color (see the **`StoneColor`** enum).

- **`Point`** - Represents a location on the board at which a **`Stone`** can be (or has been) placed. A **`Point`** may have one or zero **`Stones`** on it.

- **`PointCoords`** - Represents a set of coordinates for a point.

- **`Board`** - Represents a statically-sized, two dimensional collection of **`Points`**. There is only one **`Board`** per game. Since the **`Board`** has nearly perfect knowledge of the state of the game, many calculations are done by or with assistance from the **`Board`**.

- **`Chain`** - Represents a collection of adjacent points where every **`Point`** in the **`Chain`** has an identical state with respect to their **`Stones`** (or lack thereof). A chain of points without stones is called an "empty" chain.

- **`Player`** and **`IPlayer`** - Represents a player in the game. There are exactly two **`Players`** per game. A **`Player`** collects (or counts) captured **`Stones`** throughout the game and scores points based on that number as well as the number of "surrounded" territory at the end of the game. The **`Player`** also has a user interface (which is a template parameter).

- **`ConsoleUI`** - A user interface implementation for a **`Player`** that displays game state via standard-output and receives **`Player`** input via standard-input. All game interaction with the user passes through **`Player`** and **`ConsoleUI`**.

- **`GameController`** - Manages the flow of the game. The **`GameController`** is responsible for setting up the game, allowing the players to make alternating

moves, and comparing scores and deciding a winner at the end of the game.

Please take a few minutes to look at the code for Exercise 1 and familiarize yourself with the "surface" of these objects, as the following exercises rely on a solid understanding of how these components work together to implement the game.

After examining the code, compile the application and run it.

Exercise 2 - Using Forward Declarations

In this exercise, we are going to make a few changes to structurally improve the code.

When a type is being declared, other type names will often appear either as bases or members of the declared type or as parameters or return types of methods in the declared type. For some of these reference types, a C++ compiler must have a complete declaration available, but in many cases, the compiler only needs to know that the identifier is a type. For those in the second case, we can replace the inclusion of the header file for that type with a forward declaration.

In fact, when you have two types that reference each other, you must use forward declarations in order to resolve the circular header dependency.

Another benefit of using a forward declaration is that if the "shape" of the type changes (such as the addition of extra methods), the change does not trickle through our source code, causing needless recompilation (presuming you are using a build system that is smart enough to know not to build unchanged files). Forward declarations can also speed up the compilation process because the pre-processor has less **`#include`** statements to process.

Here are a list of things that do not require a complete type definition (and therefore can be forward declared)

- Variable declarations that are pointers (**`*`**) or references (**`&`** and **`&&`**)
- Parameters and return values for function declarations
- Parameters for function definitions where the parameter is a pointer (**`*`**) or a reference (**`&`** and **`&&`**), unless the pointer or reference is dereferenced (**`.`** or **`->`**)
- Types that satisfy the above requirements in the instantiation of a template

In addition, you may not forward declare a nested type or a type alias (the alias can

however be supplied again, as long as all aliases match).

Step 1

Examine header files for types that can be forward declared and replace the **`#include`** for their header file with forward declarations. *NOTE: It is tempting to forward declare standard library types, and for many it is possible, but it is typically not advised due to their complex nature. See the header file* ***`<iosfwd>`*** *for more interesting discussion.*

Step 2

Often, you'll come across "clusters" of type declarations whose full declarations share the same header file. As a convenience, you can create a forward declaration header file that contains all of the forward declarations for the types in the normal header file. Find places in the source code where you can leverage this technique.

Exercise 3 - Alleviating the Tediousness of Namespaces

You should always strive to place your code inside an appropriately named namespace. This allows you more flexibility in the identifiers you choose for your code and reduces the possibility that your code will conflict with code that is outside of your control. Consider the following

```
// SomeThirdParty.h

// This will display a message and purposefully
// crash the process
void crash (const char *);

// YourCode.h

// This will cause the sprite identified by the
// parameter to crash and lose a life
void crash (const char *);
```

Now, if you have included both of these header files, you will likely get a linker error complaining about multiple definitions for the same function. If you include only one header (perhaps indirectly), then you could be in for a big surprise. However if you use namespaces, it is much harder to have unexpected behavior and they also make it possible to resolve the multiple definition or ambiguity problems.

```
// SomeThirdParty.h
namespace EvilCompany {

// This will display a message and purposefully
// crash the process
void crash (const char *);

} // namespace EvilCompany


// YourCode.h

namespace CompanyXYZ {

// This will cause the sprite identified by the
// parameter to crash and lose a life
void crash (const char *);

} // namespace CompanyXYZ
```

As soon as you have a nested hierarchy of namespaces, you encounter a second problem; verbosity. But C++ has several ways of stripping away that verbosity when it is truly not needed. When you are inside a namespace, there is no need to namespace qualify sibling identifiers. For other cases there is the **`using`** keyword

```
namespace CompanyXYZ {
    namespace LibraryABC {
        void launchGame ();
    } // namespace LibraryABC
} // namespace CompanyXYZ

// This pulls identifiers in CompanyXYZ into current scope
// (for this translation unit)
using namespace CompanyXYZ;

// This creates a namespace alias (shortens a long namespace)
using namespace ABC = CompanyXYZ::LibraryABC;

// Pulls only the given identifier into current scope
```

```
// Also used to resolve identifier ambiguities
using CompanyXYZ::LibraryABC::launchGame;
```

There is however an important caveat to the **`using`** keyword. Creating aliases is usually safe, but changing the scope of an identifier to be available in the global scope in a header file should be avoided, particularly if your header file might be consumed by an innocent bystander. Generally speaking, the **`using`** keyword should be restricted to the smallest scope that is necessary to decrease keystrokes and improve readability.

Step 1

Look through the source code for this exercise to see how the **`using`** keyword is being used. Play around with using namespace aliases if you wish.

Step 2

Pulling a namespace into scope within a type declaration is not allowed. Can you think of any alternative solutions to shortening names in those cases?

Exercise 4 - Public Fields v. Accessors & Member Methods v. Free Functions

One of the most misunderstood concepts of object-oriented programming is encapsulation. A common idiom is to make every member field private (or protected) and to provide accessors to set and get the value of those those members. Often, those accessors simply return a copy or a reference to the underlying field. Well, isn't that what public accessibility is for?!? Similarly, there are times when you are defining a function that perform some operation on an object, and the natural inclination is to place the function as a member function of that type, even though all of the operations are available via the type's public interface.

There are definitely times when you want to use the accessor pattern, but if you ever see something like this

```
class Data
{
public:
    int getInteger () { return i; } // Getter
    void setInteger (int v) { i = v; } // Setter
```

```
    double getDouble () { return d; } // Getter
    void setDouble (double v) { d = v; } // Setter

    string & getString () { return s; } // Getter & Setter!!!

private:
    int i;
    double d;
    string s;
};
```

Then you should rewrite that type declaration as

```
struct Data
{
    int i;
    double d;
    string s;
};
```

Valid reasons for hiding the member fields (and defining getter and setter methods) include: providing read-only access (**`const & T get () const`**) and value validation (**`bool set (const T &)`** or **`void set (const T &)`** that throws an exception on invalid data).

```
class Data
{
public:
    const & string getString () const { return s; }

    bool set (const string & v)
    {
        if (v.length() > 10) return false;
        s = v; return true;
    }

private:
    string s;
};
```

Similarly, functions that operate purely on the public interfaces of objects should remain outside a type (although, placing them in a namespace is recommended). Even if access to private data is required, you should strongly consider making the function a free-standing function and declaring it as a friend of the intruded-upon type. The reason for this is much clearer if the function must access the private data of another type. If you can be reasonably certain that the function will need access to the private data of a type and **only** that type, making it a member function is appropriate. This conundrum is most often seen with implementing the logical operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) and the stream insertion (`<<`) and extraction (`>>`) operators.

Step 1

Evaluate the member fields of each type and decide whether to use an accessor pattern or to simply make the fields public.

Step 2

Implement the stream insertion operator (`<<`) for some types. This will be useful for logging purposes.

Step 3 (BONUS)

Implement the stream extraction operator (`>>`) for the `PointCoords` type, and use it to eliminate the sentinel values used to determine a pass.

Exercise 5 - Using Standard Containers Effectively (and the New Alias Syntax)

The standard library offers a diverse set of containers for use in your applications. These include the memory-contiguous and dynamically sized `vector` and the doubly-linked `list`, as well as many others. The latest standard introduced hash-based sets and maps as well as a statically sized `array` and the singly-linked `forward_list`.

There have also been several improvements in the usability of standard containers; one of the most overlooked is a brand new syntax for declaring type aliases. Consider the creation of a two-dimensional array of integers using C-style arrays versus C++ standard `array`

```
int a[2][3];
array<array<int, 3>, 2> a;
```

Yuck! It is significantly longer of a name, but worse, the indices are out of order in the declaration. One might try to simplify this by using a **typedef** alias

```
typedef array<array<int, 3>, 2> IntMatrix_2_3;
IntMatrix_2_3 a;
```

But again, there are some serious drawbacks. It is not extensible and would require a **typedef** statement for every used combination of ranks. Fortunately, we have a new type alias syntax that utilizes the existing **using** keyword

```
template <typename T, size_t First, size_t Second>
using matrix = array<array<T, Second>, First>;
matrix<int, 2, 3> a;
```

You are not required to use template parameters with the new alias syntax, so you could have written the previous **typedef** example as

```
using IntMatrix_2_3 = array<array<int, 3>, 2>;
IntMatrix_2_3 a;
```

I vastly prefer using the new alias syntax over the old **typedef** syntax as it feels more in line with the other usages of the **using** keyword in the language.

Step 1

Replace any usages of **typedef** with the new alias syntax with the **using** keyword.

Step 2

Create a partial template alias called **matrix** to replace the type for **Board::m_points**

Step 3 (BONUS)

Try to create a true N-dimensional array type (see Exercise 12 on variadic templates) that has a similar interface to **array**.

Exercise 6 - Smart Pointers Are Your Friends

C++ is a language that affords the programmer the power to control nearly every aspect of the execution of the program, and foremost of this control is memory management.

Memory management is so complex, that many other modern languages have decided to take that power away in order to simplify (conceptually) the view of the system that the programmer must understand. C++ finds a middle ground that increases safety but leaves room for the power: smart pointers.

One of the most difficult aspects of memory management arises from C++'s complex flow control possibilities. It is sometimes difficult to know whether a particular function call could throw an exception, causing the current function to exit scope and potentially leak heap memory. Smart pointers use a feature and pattern in C++ known as *deterministic finalization* and *resource acquisition is initialization (RAII)*, respectively.

Deterministic finalization means that as soon as an object (not references or pointers) becomes falls out of scope, it's destructor is guaranteed to execute at that point in control flow, regardless of what caused the object to fall out of scope. RAII is a principle that says management of resources (such as memory, file handles, socket connections, etc...) should be handled in the constructor and destructor of a class. Looked at from the other direction, once an object is constructed, it should be ready for operation, and once it is no longer able to be referenced in code, it should release any of the resources it was managing.

To manage the memory for heap-allocated objects in C++, you should use smart pointers such as **`unique_ptr<T>`**, **`shared_ptr<T>`**, and **`weak_ptr<T>`**. The **`unique_ptr`** class should be used to manage an object that has only a single owner (although the owner can change). When the **`unique_ptr`** object falls out of scope, the heap memory for the underlying object is freed. The **`shared_ptr`** and **`weak_ptr`** classes are used when an object's lifetime is controlled by more than a single object (and often across threads). When the last **`shared_ptr`** that manages the underlying object is destroyed, the memory is freed. When using **`shared_ptr`**, you can sometimes find yourself with circular references in which a set of objects are all keeping each other alive. In those situations, you may use **`weak_ptr`** to break the lifetime cycle. There is enough material on **`shared_ptr`** alone to cover an entire day-long workshop, but we will not spend any more time on it in this workshop.

Step 1

Replace dynamic objects managed with **`new`**/**`delete`** with the appropriate smart pointers (they should all be **`unique_ptr`** for this application). You will need to use the **`std::move`** function in order to pass **`unique_ptr`**s to methods.

Exercise 7 - Simplifying Your Declarations Part 1 (In-class Initialization)

Uninitialized data has historically been one of the weak points in C and C++, particularly when it comes to pointers or types that contain pointers. And in the past, you had to be very careful to ensure that every constructor for a type properly initialized each member, quite often to the same value for each constructor. Now, you can provide a default initialization value for each member at the declaration site instead of in the constructor.

```
class SomeObject
{
public:
    SomeObject (int length) : m_name("Default")
                            , m_length(length)
    { }

    SomeObject (int length, const char * name) : m_name(name)
                                               , m_length(length)
    { }

    Some Object (const char * name) : m_name(name)
    { }

    SomeObject(bool hasName) : m_name("Default")
                             , m_hasName(hasName)
    { }

private:
    int m_length;
    bool m_hasName;
    string m_name;
};
```

In the above scenario, we initialize **m_name** with the **const char *** value **"Default"** several times, we do not initialize **m_hasName** or **m_length** in every case. Using initialization in the class declaration we would have instead

```
class SomeObject
{
```

```
public:
    SomeObject (int length) : m_length(length)
    { }

    SomeObject (int length, const char * name) : m_name(name)
                                                , m_length(length)
    { }

    Some Object (const char * name) : m_name(name)
    { }

    SomeObject(bool hasName) : m_hasName(hasName)
    { }

private:
    int m_length = 0;
    bool m_hasName = true;
    string m_name = "Default";
};
```

Not only have we reduced the cost of adding new constructors or more class members, but we have made the code much safer.

Step 1

Initialize all member fields in the class declaration (unless default construction is suitable).

Step 2 (BONUS)

Under what situations might you ***NOT*** want to initialize members in the type declaration?

Exercise 8 - Simplifying Your Declarations Part 2 (Override Controls)

One of the most nefarious and difficult to track bugs that occurs with C++ code is triggered by the following scenario

```
class Base {
public
    virtual Base~ ();
    virtual add (int increment)
```

```
    { m_value += increment; }
    int getValue () { return m_value; }
protected:
    int m_value = 0;
};

class Derived : public Base {
public
    virtual Derived~ ();
    virtual add (int increment)
    { m_value += 2*increment; }
};

void f () {
    Derived d;
    d.add(2.6); d.add(2.6);
    d.getValue(); // should return 8 (0 + 2*2 + 2*2)
}
```

Eventually someone realizes that **`m_value`** should be a **`double`** and decides to change **`Base`** but fails to update **`Derived`**

```
class Base {
public
    virtual Base~ ();
    virtual add (double increment)
    { m_value += increment; }
    double getValue () { return m_value; }
protected:
    double m_value = 0;
};

class Derived : public Base {
public
    virtual Derived~ ();
    virtual add (int increment)
    { m_value += 2*increment; }
};
```

```
void f () {
    Derived d;
    d.add(2.6); d.add(2.6); // Calls Base::add(double)!!!
    d.getValue(); // should return 10.4 (0 + 2*2.6 + 2*2.6)
                  // instead returns 5.2 (0 + 2.6 + 2.6)
}
```

This can now be avoided by using the **override** keyword

```
class Base {
public
    virtual Base~ ();
    virtual add (double increment)
    { m_value += increment; }
    double getValue () { return m_value; }
protected:
    double m_value = 0;
};

class Derived : public Base {
public
    virtual Derived~ ();
    virtual add (int increment) override // NOW FAILS TO COMPILE
    { m_value += 2*increment; }
};
```

There is also a **final** keyword that prevents a method from being overridden or prevents a type from being derived from (useful if you design a type that should never be a base, perhaps because of a non-virtual destructor).

Step 1

There is only a single inheritance hierarchy in our implementation, **IPlayer** and the class template **Player**. Apply the **override** keyword to the Player methods that are implementing **IPlayer**.

Step 2

For safety, mark all the classes that are not designed to be derived from with the **final** keyword.

Exercise 9 - Simplifying Your Declarations Part 3 (Special Member Functions)

In C++, there are several special member functions that can be implicitly generated by the compiler under certain conditions. These include

```
SomeObject (); // Default constructor
~SomeObject (); // Default destructor
SomeObject (const SomeObject &); // Copy constructor
SomeObject & operator= (const SomeObject &); // Copy assignment
// NEW IN C++11!
SomeObject (SomeObject &&); // Move constructor
SomeObject & operator= (SomeObject &&); // Move assignment
```

If in your class, you declare a constructor (that is not the copy- or move- constructor), the implicit generation of the default constructor is suppressed. If you desire a constructor with no parameters, you must then define one yourself. If your implementation is identical to the one the generated one would have made, you can now use the keyword **`default`** to instruct the compiler to always generate the method

```
class SomeObject {
public:
    SomeObject (const char * name);
    SomeObject () = default;
private:
    std::string m_name = "Default";
};
```

Similarly, if you define any of a destructor, copy-constructor or move-constructor, you *should* explicitly define all of those functions. You can use the **`default`** keyword, or if you truly wish to suppress those functions you may use the **`delete`** keyword

```
// Safely-derivable, movable, and non-copyable
class SomeObject {
public:
    virtual ~SomeObject = default;
    SomeObject (const SomeObject &) = delete;
    SomeObject & operator= (const SomeObject &) = delete;
    SomeObject (SomeObject &&) = default;
```

```
    SomeObject & operator= (SomeObject &&) = default;
};
```

Step 1

Locate any types that have manually implemented the default for the special member functions and replace them with the **`default`** keyword.

Step 2 (BONUS)

Identify any types that should be non-copyable or non-movable, and use the **`delete`** keyword to remove them.

Exercise 10 - Uniform Initialization (Do My Eyes Deceive? More Curly Braces?)

C++ has many different ways to construct and initialize a new object.

```
SomeObject o1; // Default construction
SomeObject o2("Hello"); // User-defined construction
SomeObject o3 = o2; // Copy construction
SomeObject o4 = getObject(); // Copy/Move construction
SomeObject o5 = "Hello"; // User-defined+Copy/Move construction
```

And for arrays you can initialize like so

```
int a[] = {0, 1, 2, 3, 4};
```

But not for container types

```
vector<int> v = {0, 1, 2, 3, 4}; // Nope!
vector<int> v;
v.push_back(0);
...
v.push_back(4); // Whew, finally done.
```

And even further, there is the well-known *Most Vexing Parse*

```
// Not a declaration of 'o' as a SomeObject that is initialized
// with a default-constructed OtherObject, but rather
// a function declaration 'o' that returns a SomeObject
```

```
// and takes a function that returns an OtherObject and
// takes no parameters.
SomeObject o(OtherObject());

SomeObject o2(); // Similar problem here
```

So in order to solve these discrepancies, the standards committee (in their infinite wisdom) decided to add yet another way to initialize objects, the *uniform initialization* syntax. Uniform initialization uses **{** and **}** instead of **(** and **)**, and practically any place where you could have used **(** and **)** to initialize an object before, you can use **{** and **}** instead (although there are a few gotchas).

```
// These now do what we would expect them to do
SomeObject o{OtherObject{}};
SomeObject o2{};
```

But we still have a problem with container initialization. To resolve this, there is now a **std::initializer_list<T>** object, defined in **<initializer_list>** header, that you can utilize in a constructor (most often for containers), and **initializer_list** is directly constructible from the uniform initialization syntax. So now, our container example looks like

```
int a[] = {0, 1, 2, 3, 4};
vector<int> v = {0, 1, 2, 3, 4}; // Thank goodness!
```

Step 1

Use uniform initialization to initialize as many objects as you can find. The containers are the tricky ones!

Step 2 (BONUS)

Think about how you could use uniform initialization to start a game with a board in a pre-configured state.

Exercise 11 - Automatic Type Deduction & New Function Declarations

Automatic type deduction is a deceivingly useful new feature of the C++ language. It allows the programmer to concentrate on the problem at hand instead of the syntactics of the language. Templates in C++ have always provided some level of type deduction, but it

was of course limited to creating and using generic types or algorithms. There are two new type deduction facilities in C++11, the **auto** keyword and the **decltype** keyword.

The **auto** keyword can be used to reduce the amount of typing needed to declare a variable

```
auto i = 42; // instead of int i = 42;
```

The previous example is a bit simplistic and doesn't really demonstrate the power of **auto**. Here is a better example

```
vector<int> container = vector<int> {0, 1, 2, 3, 4,
                                     5, 6, 7, 8, 9};
for (vector<int>::const_iterator it = begin(container);
                                 it != end(container); ++it)
{ // Do something with *it }
```

Now if we wanted to switch out **vector<int>** for **array<int, 10>** we'd have to make that refactor in at least three distinct source locations. If however we had used **auto**

```
auto container = vector<int> {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (auto it = begin(container); it != end(container); ++it)
{ // Do something with *it }
```

the number of source locations we need to change is reduced to a single location. Furthermore, there are some types that the programmer cannot know, and the only way to capture instances of those types are with the **auto** keyword (see Exercise 13).

The **decltype** keyword is a bit more flexible than the **auto** keyword, but it should only be used in a relatively few circumstances. Consider the following function template

```
template <typename L, typename R>
int multiply (const L & l, const R & r)
{ return l * r; }
```

As long at **L** and **R** are both **int**, this function works as one would expect. However, if one of the parameters is a **double**, you will get truncation (**double → int**). Furthermore, if result of the expression **l * r** is a type that does not implicitly cast to **int**, this code will not compile. Obviously we intend that the return type should be the type of the

expression **`l * r.`** You can obtain said type using the **`decltype`** keyword.

```
template <typename L, typename R>
delctype(l * r) multiply (const L & l, const R & r)
{ return l * r; }
```

However, when the C++ parser sees this code, it will not yet know what types **`l`** and **`r`** are, so we use a new function declaration syntax

```
template <typename L, typename R>
auto multiply (const L & l, const R & r) -> decltype(l * r)
{ return l * r; }
```

Step 1

Replace as many explicit type declarations with **`auto`** (preferred) or **`decltype`**.

Step 2

Experiment with the new function declaration syntax. It works anywhere!

Step 3 (BONUS)

Explore the differences between template type deduction, **`auto`**, and **`decltype`**. The functions in **`<type_traits>`** are quite helpful.

Exercise 12 - Implementing a Logger With Variadic Templates

In this exercise, you'll implement a logging utility that utilizes variadic templates. You will also leverage RAII to utilize the logging utility to log function calls.

Variadic templates are a wonderfully complicated addition to the language, that is primarily intended for library designers. It allows the creation of a class or function template that can take an unlimited (i.e. implementation defined) number of template parameters. The prime example of the usefulness of this facility the the std::tuple class. Tuples are essentially unnamed types that contain other types and are very useful for packing multiple types for a very limited usage (e.g. returning multiple types from a single method).

Variadic templates rely on parameter packs and the pack expansion operator (**`...`**). Strangely enough, a parameter pack is defined using the pack expansion operator. You

can define two different types of parameter packs, template parameter packs and function argument parameter packs.

```
// Examples of template parameter packs
template <typename ... Args>
template <size_t ... Args>
```

Function argument parameter packs are declared in a similar fashion, and utilize template parameter packs.

```
// Examples of function argument parameter packs
template <typename ... Args> void func (Args ... args)
```

Eventually, you will need to expand all the parameter packs that have been defined (and sometimes more than once). You use the ... operator in those cases too. The **...** operator will expand any expressions to the left of the operator that contain at least one unexpanded parameter pack. If more than one unexpanded parameter pack is found, the parameter packs must have the same number of items and they are expanded in conjunction with each other. Examples are useful.

```
template <typename ... Args>
class MyVariadic : public Args... { };
```

when instantiated as **MyVariadic<Base1, Base2, MyVariadic<>>** expands to (with filled in names as only the compiler knows the real name)

```
class MyVariadic_EMPTY { };
class MyVariadic_B1B2EMPTY : public Base1, public Base2
                           , public MyVariadic_EMPTY { };
```

Step 1

The current **Logger** method will only log a single parameter per call, leading to code like this

```
gLogger.log(LogLevel::kFirehose, “Skipping”);
gLogger.log(LogLevel::kFirehose, point);
```

Make the **Logger** class a variadic template so that the previous example can be written as

```
gLogger.log(LogLevel::kFirehose, "Skipping ", point);
```

Be sure to use the **std::forward** function in order to preserve the types of the parameters.

Step 2

Now, we'll use our **Logger** and RAII (and a macro) to log the entry and exit of function calls.

In **Logger.hpp**, we'll make a new class, **FunctionLogger**, whose constructor will take a **LogLevel** and **const char *** (for the method name) parameters. The constructor will call the static logger object to log the entry of the method. The destructor will log the exit of the method.

You may also decide to track a nesting level so that you can more easily visualize nested function calls.

Then you should make two macros **LOG_FUNCTION** and **LOG_BUSY_FUNCTION** that will instantiate a **FunctionLogger** on the stack. **LOG_FUNCTION** will pass in a **LogLevel** of **kHigh** and **LOG_BUSY_FUNCTION** will pass in a **LogLevel** of **kFirehose**.

Go into a couple of your .cpp files and add the **LOG_FUNCTION** or **LOG_BUSY_FUNCTION** as the first line of some of the methods. Change the logging level in **Logger.cpp** to **kHigh**, rebuild the application, and run it.

Step 3 (BONUS)

Use variadic templates to modify our **FunctionLogger** to also print out the parameters to the function. Variadic macros will help.

Exercise 13 - The Wonderful World of Lambdas

A lambda is basically shorthand for creating an object (possibly stateful) that can be invoked as if it were a method. The closest equivalent of a lambda in the C++98/03 standard was a pattern known as a *functor*.

Lambda are one of the hallmark features of the C++11 standard. In the short time since the approval of the standard, lambda have already begun to drastically change the way that modern C++ code is written (see C++/CX as prime evidence).

There are three fundamental parts of a lambda:

1. **The Capture Block**
   Also known as the the lambda introducer; the capture block is the means by which a lambda is able to capture state from the enclosing scope. A capture block is denoted by a **`[`**, a list of captured objects (possibly empty), and terminated by **`]`**. Objects can be captured by value (the default) or by reference (when prefixed with **`&`**). All objects available from the enclosing scope can be captured as well (**`[=]`** for by-value, **`[&]`** for by-reference). You can also capture everything using one of the passing styles, but choose to capture particular items in the other passing style

2. **The Signature**
   The signature of a lambda function uses a syntax similar to the new function declaration syntax. The main differences is the missing **`auto`** keyword (the capture block is sufficient) and that the trailing return type is optional in most simple cases.

3. **The Body**
   The body of a lambda function is just like the body of a named function. Since lambdas are typically used as a component of an enclosing expression or statement, there is typically some context-dependent tokens that will appear after a lambda body (e.g. **`;`** or **`)`**)

So putting it all together, the following is the simplest possible lambda expression. It defines a lambda that captures nothing, takes no parameters, has a void return type, and does nothing.

```
[](){}
```

Of course, that isn't a very useful lambda, so here is a much more useful one that captures nothing, takes two **`int`** parameters, and does a less-than comparison.

```
[] (int i, int j) { return i < j; }
```

Step 1

There are two places in our implementation of Go where we expose the state of an object to some other object so that we can iterate over some set of things. Take a look at the **`Chain::doChainCalculation`** method. In order to calculate a chain of stones, we must

evaluate the neighboring points and determine their state (and potentially their neighbors as well). Currently, the Board class is providing accessor methods to access individual neighboring points, which are then tested, classified, and potentially recursed upon. Let's replace this "leaky" encapsulation with a lambda instead.

In `Board.hpp`, create a public method with the following signature

```
void visitNeighboringPoints(const Point & point, const PointVisitorFn & visitorFn) const;
```

Obviously, we haven't defined the type **`PointVisiterFn`** yet, so let's go ahead and do that. In `Point.fwd.hpp` create the following type alias

```
using PointVisitorFn = std::function<void(const Point &)>;
```

We haven't talked much about the **`std::function`** object, but for the current exercise, just consider it a way to talk about the type of an invokable expression.

In `Board.cpp`, implement **`visitNeighboringPoints`** by calling **`visitorFn`** for each neighboring point (you may use the existing **`getPoint*`** methods, which can now be made private).

Now, back in the **`Chain::doChainCalculation`** method, call the **`visitNeighboringPoints`** method and pass in a lambda that captures the everything by reference (**`[&]`**) and has a signature (minus the second parameter) and body identical to the **`Chain::doTest`** method. You can then remove the **`Chain::doTest`** method.

Step 2

There is also a some state of the **`Chain`** object that is exposed to the **`Board`** object in the methods **`Board::isValidMove`** and **`Board::removeCapturedStones`**. Instead of exposing the internal state of Chain, let's apply the same principles from *Step 1* and create the following **`Chain`** methods

```
void forEachPoint (const PointVisitorFn & visitorFn) const;
void forEachSurroundingPoint (StoneColor color, const PointVisitorFn & visitorFn) const;
```

The method **`forEachPoint`** should visit each point in the chain (they all have the same stone color), and **`forEachSurroundingPoint`** should visit each point in the chain AND neighboring points that have stones that are of the color specified by **`color`**. **`forEachPoint`** can be implemented in terms of **`forEachSurroundingPoint`**.

Wrapping Up

There are so many exciting features of the newest C++ standard, that it would quite literally take several semesters-worth of college courses to cover it all. I hope that you continue on the path of learning as much of it as you can. It is truly an exciting time to write applications with C++, and I'm looking forward to coming years and all of the great code that will be written utilizing all of these great advancement forward.