Introduction To C++

References

- https://isocpp.org/tour
- http://www.modernescpp.com/index.php?start=0
- https://www.bfilipek.com/
- https://www.cppstories.com/

Preface

These lecture notes assume a basic comfort level in programming in another language, such as Java or C#. Concepts such as conditionals, loops, functions, objects/classes, generics, and exceptions are assumed to already be known. These topics are only addressed insofar as to introduce them with respect to how C++ provides the capabilities, not the concepts themselves.

Language Overview

C++ is a massive language, more than any one person comprehensively knows or knows well (when I speak of C++, I mean the language and its standard library). Over time is has become a multi-paradigm language, taking on aspects/concepts from various programming models including imperative, object oriented, generic, and functional. It is intended to enable high-performance code, with the concept of "Leave no room for a lower-level language below C++ (except assembler)". An overview of its features include:

- Cross-platform, compiled to native OS binary
- Conditional flow-control
- Looping structures
- Dynamic memory
- Objects
- Polymorphism
- Operator overloading
- Exception handling
- Generic programming
- Large standard library

A Compiled Language

C++ is a compiled language…

Several steps:

- Preprocessor
- Compiler

- Linker

Whenever you see a statement that begins with #, it is a compiler (preprocessor) directive, instructing the compiler to do something as part of the process in constructing the final compiled program. A few preprocessor directives are covered throughout these lecture notes as needed.

The preprocessor modifies the C++ code according to the placement and use of the directives in the C++ code. This is called the first-pass over the C++ code and the result is C++ code modified according to the directives, prepared for use by the compiler.

The compiler translates the C++ code into mostly executable code, with some placeholders that it is unable to resolve. It is the job of the linker to resolve the placholders based on information from other compiled code (usually external libraries) and create the final executable, ready for execution.

A first C++ program

```
#include <iostream>

//
// This is the main entry point of a C++ program.
//
int main(int argc, char* argv[])
{

       std::cout << "Hello World!" << std::endl;

       return 0;
}
```

There are 7 elements of this program to discuss:

1. The `#include` directive
2. Comments
3. Entry point of a C++ program, including command line parameters
4. Primitive data types & arrays
5. Function declaration, scope, and return type
6. Statements
7. Streaming output to the console

The #include Directive

The statement above is a compiler directive, it is not a statement that compiles to logic in the executable program. The `#include` directive instructs the preprocessor to insert the specified file into the directive location in the file currently being processed. The name of the file to include can be specified in two different ways, enclosed using angle brackets, `<>`, or double quotes, `""`. The different enclosing methods instruct the compiler's search for the location of the file.

When angle brackets are used it tells the preprocessor to search for the file within the compiler's own distribution folder(s). Files included using the angle brackets are from the C++ standard library, for things like console output, file manipulation, or containers. Conventionally, no filename extensions are used when including C++ vendor supplied standard files. It is left to the compiler vendor the naming, including the extension, of these files. One vendor might name a file `vector.hpp` while another might name it vector.cc, each vendor having their reasons for doing so. Therefore, the standard is to provide the name of the library, then let the compiler decide the actual name to locate and include.

The use of double quotes when using the #include directive indicates to the preprocessor to first search the C++ project defined folders, and if the file is found there, use it for replacement. If the file is not located within the project defined folders, then the preprocessor searches for the file in its set of folders.

For the Hello World program above, #include <iostream> instructs the preprocessor to find the C++ library for streaming output (to the console).

It is tempting to think that import from Java or using from C# are analogous to #include in C++, they aren't! The similarities exist only at a cursory and naive glance, they are very different things. The keywords import/using from Java/C# instruct the compiler on how to search for, or resolve, names of things it finds in code. The #include directive in C++ instructs the compiler (preprocessor) to include a file in that location. While the effect of both approaches may feel similar, they are very different things and shouldn't be confused with each other. In other words, don't think of #include being C++'s way to do what Java's import does.

Reference the upcoming section on Object Oriented C++ and the discussion of translation units.

Comments

Comments in C++ may be placed on a single line using \\. Multi-line comments are enclosed using /* */.

```
// This is a single line comment

/* This is a comment
   which spans multiple lines
*/
```

Program Entry Point

The starting point of a C++ program is the function called main. The main function can be defined to have an int or void return type, and also to have no parameters or the two parameters of data types int and char *[], respectively. In the above example, main is defined with an int return type and the two function parameters that contains the values from the command line when executed (if any).

Primitive Data Types & Arrays

Two (of the many) primitive data types are shown in this example, int and char. C++ includes primitives that you expect such as int, long, float, double, and boolean. Additionally, unlike Java for example, it provides support for unsigned (integral) data types (only positive values) through the use of the unsigned keyword. For example, unsigned int or unsigned long.

Another relevant difference between C++ and Java is that there are no corresponding class, or reference, types for the primitives. In Java, these types are necessary for a number of reasons, including the fact that Generics (in Java) only work with reference types. In C++, templates (C++ generic programming) may be based on primitive types.

Both variable and parameter declaration follow the pattern of defining the data type first, followed by the identifier name for the value. The following are example primitive data type declarations.

```
int age;
double lowerBound = 200.24;       // also double lowerBound{200.24}
unsigned int total = 0;           // also unsigned int total{0};
bool finished = false;            // also finished{false};
```

Variables and parameters may be declared with or without an initial value. In the case of a parameter declaration, this is known as a *default value*; more detailed discussion in the section on functions.

C++ supports compiler deduction of variable types through the use of the `auto` and `decltype` keywords, these will be covered in detail later.

The second parameter of the `main` function is an array type (it also shows a pointer type, but we'll save that discussion for later). In C++, the array type declaration goes on the right-hand side of the identifier name, unlike Java where (conventionally) the array type is on the left-hand side of the identifier, combined with the type of the array elements. There is another C++ array type that will be covered later, because of that, I will call the kind of array in this sample code a *raw array*. The following are example raw array declarations:

```
int ages[10];
float results[4];
float sizes[10][10];
```

When declaring a raw array, the size of the array is indicated between the brackets. In Java the declaration of an array doesn't create the array, instead a dynamic memory allocation is necessary. In C++, the declaration of an array does result in the elements of the array being created; on the stack. Accordingly, when the scope for the array is exited, the array no longer exists; removed from the stack. This is a very important distinction to get used to between the way arrays work in C++ versus what you are used to with Java. In Java, arrays are heap allocated, in C++ arrays (as shown above) are stack allocated. Arrays in C++ can be heap allocated, this will be discussed and demonstrated later.

In the case of the `main` function, no size is indicated within the array brackets. This is because a raw array parameter may be of any size. In C++, unlike Java, raw arrays do not know their size; this is why the first parameter exists, it is the number of elements in the array.

Function declaration, scope, and return type

The entry point to the program, the `main` function, also demonstrates the structure of all functions in C++. Functions have the following general form…

```
<return type> name(<parameters>)
{
        ...statements...

        return <value>;
}
```

The `<return type>` is any valid data type, a primitive or any other library or user defined data type. Functions may return one or no values. If no return value, the `<return type>` must be `void`. A function that returns a value, may use the return statement multiple times throughout the function body; although I am a believer in a single point of return at the end of functions, with only a few exceptions.

The name of the function is any valid C++ identifier.

A function may have zero to any number of parameters. Parameters may be defined with a default value.

The body of a function begins with an opening brace `{` and ends with a closing brace `}`. Generally speaking, braces define scope in C++; there are other scoping rules, but scope as defined by braces dominates. A new local scope, within an existing scope, can be opened in C++ through the opening brace, and closed with the closing brace.

There are also file and class scope: https://codescracker.com/cpp/cpp-scope-rules.htm

Statements

A semicolon `;` is used to terminate a statement in C++. Within the function body any number of statements may be written. The rules for semicolon placement are nearly the same as Java, with only a few minor differences. For example, when defining a class in Java, the closing brace is not followed by a semicolon, whereas in C++, the closing brace on the declaration of a C++ class is followed by a semicolon.

Remember, preprocessor directives are not program statements, therefore, they don't require semicolons at the end. Having a semicolon after a preprocessor directive won't lead to a compilation error, because a semicolon by itself is a statement. However the compiler may issue a warning if it doesn't recognize a good reason for the semicolon to be needed.

Streaming output to the console

The first line of code in the program tells the preprocessor to include the C++ input/output (console) streaming library code. The `std::cout << "Hello World!" << std::endl;` statement is used to send the string `Hello World!` to the console.

`std::cout` is a static object where output is streamed. The prefix `std::` is a namespace, and will be discussed in more detail later. For now, just understand that it is necessary. The `<<` and `>>` symbols are the streaming operators. The `<<` is called the *insertion* operator and `>>` is called the *extraction* operator (more on these later). In the above statement, we are inserting the `Hello World!` string to the `std::cout` object. Anything inserted into the `std::cout` object, by default, will appear on the console.

Language Basics

With an introduction to the C++ language out of the way, let's take a more detailed look at some basic elements of the language.

Primitive Data Types

The following are the predefined data types provided in C++

- `char` – Used for storing characters (ASCII). Size is 8-bits.
- `int` – Integer data type. Size depends on the system, but typically 32-bits.
- `float` – Single precision floating point value. Size depends on the system, but typically 32-bits.
- `double` – Double precision floating point value. Size depends on the system, but typically 64 bits.
- `bool` – Boolean data type, with values of true and false. Size depends on the system, but typically 8 bits.
- `void` – Indicates no value, usually seen as the return type for a function. There is no size, because there is no value to store.

In addition to these primitives, there are modifiers that can be used to change the size, and therefore, numeric range of the primitives. These modifiers are:

- `signed` – Modifies the type to have a negative and positive range. Have of the values are negative, while the other half are positive (where 0 is considered a positive number). By default, the primitives are signed.
- `unsigned` – Modifies the type to have only a positive range.
- `short` – Reduces the storage of the type by half, and therefore the range of the type.
- `long` – Increases the storage of the type by two, and therefore, the range of the type.

The `sizeof` operator can be used to determine (at compile time) the number of bytes for a type. `std::numeric_limits<>`, found in the `<limits>` library, is a C++ template that allows a program to determine (at compile time) various numeric details of primitive types in C++; for example, the numeric ranges of a type. The `sizeof` operator and `std::numeric_limits<>` template are used in the following code segment to report the size and range of selected primitive data types.

```
std::cout << "--- Primitive type sizes ---" << std::endl;
std::cout << "# of bytes for a char            : " << sizeof(char) << std::endl;
std::cout << "# of bytes for an int             : " << sizeof(int) << std::endl;
std::cout << "# of bytes for a float            : " << sizeof(float) << std::endl;
std::cout << "# of bytes for a double           : " << sizeof(double) << std::endl;

std::cout << std::endl;
std::cout << "--- Primitive numeric ranges ---" << std::endl;
// This static_cast is necessary for the char type to show the output as numbers
std::cout << "Numeric range for char   : [" <<
       static_cast<int>(std::numeric_limits<char>::min()) << ", " <<
       static_cast<int>(std::numeric_limits<char>::max()) << "]" << std::endl;

std::cout << "Numeric range for int    : [" <<
       std::numeric_limits<int>::min() << ", " <<
       std::numeric_limits<int>::max() << "]" << std::endl;

std::cout << "Numeric range for float  : [" <<
       std::numeric_limits<float>::min() << ", " <<
       std::numeric_limits<float>::max() << "]" << std::endl;
```

```
std::cout << "Numeric range for double : [" <<
        std::numeric_limits<double>::min() << ", " <<
        std::numeric_limits<double>::max() << "]" << std::endl;
```

Why the `static_cast<int>` when reporting the numeric range of a `char`? The `char` type is a numeric value, but is interpreted for output as an ASCII character, even in the case of reporting the `min()` and `max()` values of the numeric range. `static_cast<>` is a type conversion operator that converts to the type specified in the angle brackets `<>.` In order to get a numeric value reported to the console, `int` is used as the conversion type.

Default values

The C++ language does not define the default value for uninitialized variables. Therefore, you cannot count on the value of an uninitialized variable to be a known value, such as 0.

C++ has something called uniform initialization through the use of `{}`. When used to initialize a type, using empty brackets, the default value for the data type is set.

Additional Integer Types

The C++ standard library provides a header file `<cstdint>` that contains a set of pre-defined integral data types that have specific bit-level definitions. Depending on the computer architecture, an `int` could be 8 bits, 16 bits, 32 bits, or 64 bits. If you require a specific number of bits for an integral data type, the `<cstdint>` library is what you need. The following are some of the types defined in the library (both signed and unsigned):

- `std::intmax_t`, `std::uintmax_t` – Max supported `int` type by the platform.
- `std::int8_t`, `std::uint8_t` – Exactly 8 bit integer.
- `std::int16_t`, `std::uint16_t` – Exactly 16 bit integer.
- `std::int32_t`, `std::uint32_t` – Exactly 32 bit integer.
- `std::int64_t`, `std::uint64_t` – Exactly 64 bit integer.
- `std::int_least8_t`, `std::uint_least8_t` – Integer type with at least 8 bits
- `std::int_least16_t`, `std::uint_least16_t` – Integer type with at least 16 bits
- `std::int_least32_t`, `std::uint_least32_t` – Integer type with at least 32 bits
- `std::int_least64_t`, `std::uint_least64_t` – Integer type with at least 64 bits

If any of these types are not supported by the platform, they will not be defined, therefore portability across all possible platforms is not guaranteed.

Literals & Constants

Literals and constants are related, but mean different things in the C++ language. Both are fixed values that, once specified, can not be changed either during compile or run-time. A literal is a value, numeric, character, boolean, or string located anywhere in the code. A constant is defined using the `const` keyword (`constexpr` coming later).

The following are examples of literals:

```
int a{2};
double b{2.34};
bool condition{true};
char setting{'x'};
std::cout << "Hello World!" << std::endl;
std::string message = "Hello from C++"s; // using namespace std::string_literals;
```

The `2`, `2.34`, `true`, `'x'`, and `"Hello World!"` are all literals. Literals have a type. An integral literal is an `int`, whereas a literal numeric with a decimal is a `double`. To define a `long` integral literal, place an `l` after the number, e.g., `1234l.` To define a `float` point literal, place an `f` after the number, e.g., `1234.56f`. An `std::string` literal is defined by placing an `s` after the double quoted string.

The literal `"Hello World!"` is an array of `char`, or more specifically, a pointer to `char`; `char*`. For the most part, `char` arrays and pointers to char (`char*`) are not used in these notes, instead the focus is on the `std::string` type.

The following are examples of constants:

```
const double PI{3.14159};
const unsigned int MAX_CAPACITY{100};
```

All constants have a type, just as literals. In the case of a constant the type is specified right after the keyword `const`.

While not yet detailed functions (or methods), it is relevant to note that parameters to functions and methods can have constant parameters. When a parameter is marked as constant, it means the value of that parameter cannot be changed for the function scope.

Type Conversions

A type conversion occurs where a value of one type is copied into the value of another type.

There is a lot to say on the subject of casting between types in C++. For now, an introduction to type conversions with respect to *implicit* conversions and explicit conversions, or *type casting*. Throughout the remainder of these notes, as appropriate for the context, additional detail on type conversions is provided.

Implicit Conversions

An implicit conversion is one that is recognized by the compiler and automatically performed without requiring any additional code. Consider the following code:

```
short a{1000};
long b{a};
```

The first statement assigns the (`short`) value of 1000 to the variable `a`, which is of type `short`. The second statement copies the `short` value stored in `a` to the variable `b` whose type is `long`. Because the types are different, a type conversion takes place. In this case, because the C++ compiler considers the type conversion from `short` to `long` to be safe, because a `long` can represent all `short` values, an implicit conversion takes place.

Another example of an implicit conversion is when converting from a float to a double.

```
float pi1{3.14159f};
double pi2{pi1};
```

Both of these examples are also known as *promotions*. A smaller storage type is being promoted to a larger storage type, which is a perfectly safe type conversion.

Type Casting

What if we want to do a type conversion from a `long` to a `short`, or from a `double` to a `float`. As the developer of the code, we may know and have other protections in place to ensure the conversion is appropriate for our application, but also want to eliminate the warnings associated with the implicit conversions. Consider the next code segment.

```
long c{1000};
short d{c};
```

This code will compile, but report a warning on the conversion from `c` to `d` (depending upon compiler warning options). The compiler is warning the developer that it can't guarantee any loss of data in the conversion. In order eliminate this warning, an explicit type cast is necessary, as demonstrated in the next code sample.

```
long c{1000};
short d{static_cast<short>(c)};
```

The `static_cast<type>(var)` will work if there is an implicit conversion sequence from the value/expression being converted to the new type. Additional detail on the `static_cast<type>(var)` operator is discussed in later sections, particularly when discussing the object oriented features of the language.

There are several other casting operators that are presented later on. These include `const_cast<type>`, `dynamic_cast<type>`, `reinterpret_cast<type>`, and `static_cast<type>`.

You might also see type casting done in two other ways in C++, these are older style approaches and not recommended.

```
long c{1000};
short d = short(c);
short e = (short)c;
```

The reason it isn't recommended is that a (short) casting attempts all of these in C++ (in this order), and you probably don't want all of these as possibilities in your code when you likely meant just to perform a `static_cast`:

```
const_cast
static_cast
reinterpret_cast
```

There is also a `dynamic_cast` in C++, used to inspect the type of an object.

Strings

There are two types of strings available within C++. The first comes from the original C language and is a one-dimensional array of characters (`char`) terminated by the null character; the null character is represented by the character `\0`. This is called a *null terminated string* or *c-string*. The second type is the `std::string` class that is part of the standard library which provides high-level string manipulation.

I am not going to discuss c-strings other than to acknowledge they exist and you probably need to be familiar with them for working with legacy code, and a few exceptional cases. Otherwise, it is the author's recommendation to work with the `std::string` class, unless a c-string is required for some reason. The `std::string` class is a lot more like the strings you are used to in Java, but with some differences.

The `std::string` class is available in the `<string>` library header. It provides the functionality you expect from a string type. This includes capabilities for constructing, copying, manipulation, along with many other features. The following code demonstrates a few examples of creating a string.

```
std::string greeting = "Good Morning";
std::string part1 = "This is the first part ";
std::string part2 = "of a two-part message.";
std::string full = part1 + part2;
```

The `std::string` class is part of the `std` namespace, therefore the `std::` prefix is placed in front of the class type.

The string literals on the right-hand side of the declarations are c-strings. These c-strings are accepted by an overloaded `std::string` constructor that accepts c-strings, then copies and manages them internally.

A new `std::string` instance is created when two `std::string` instances are added, as demonstrated by the full variable declaration.

Unlike Java strings, C++ strings are mutable. They can be modified without resulting in the creation of a new instance. This is demonstrated by the following code example.

```
std::string incomplete = "This is an incomplete ";
incomplete.append("sentence");  // Method to append

std::string incomplete2 = "This is an incomplete ";
incomplete2 += "sentence";      // Operator to append
```

The first two statements show the use of the `.append` method to concatenate to an existing string. The second two statements show the use of the `+=` operator to concatenate to an existing string. Both sets of statements are equivalent.

A string knows its own length through the `.size` method. Additionally, the individual characters in a string may be accessed through the use of the array `[]` operator; a 0-based index.

```
unsigned int firstAndLast = greeting[0] + greeting[greeting.size() - 1];
std::cout << "First and last characters sum: " << firstAndLast << std::endl;
```

String Views

Let’s say we have a large string, but want to break it into multiple small strings, maybe even each word. Doing this with the `std::string` class results in making a lot of copies, which usually isn’t necessary (the copies). For cases where copies of a string are not necessary, C++ provides the `std::string_view,` which is a read-only view into an existing `std::string`. Consider the following code.

```
std::string part1 = "This is the first part ";
std::string part2 = "of a two-part message.";
std::string full = part1 + part2;

std::string_view view = full;
std::string_view view1 = view.substr(0, part1.size());
std::string_view view2 = view.substr(part1.size(), part2.size());
std::cout << view1 << std::endl;
std::cout << view2 << std::endl;

full[0] = 'Z';
std::cout << view << std::endl;
std::cout << view1 << std::endl;
```

The variable `full` contains an instance of an `std::string`, while the variable `view` is an instance of an `std::string_view`, which is a window, or view, into the `std::string` contained by `full`. When `view` is initialized, it refers to the complete string contained in `full`. The variable `view1`, however, is initialized by taking a sub-string of `view`, similarly `view2` is a sub-string of `view`. Because `view`, `view1`, and `view2` are all `std::string_view` types, no copies of the `std::string` in `full` are made, instead windows into the `full` variable are made. This is proved by the code where the first character in the `full` string is modified and then the views in `view` and `view1` of the string are displayed, reflecting the change made to the source `full` string.

Automatic Type Inference

C++ has the ability to infer, or deduce, a type from its initializer, return statement, or function/method argument usage. This section only covers type inference from initializers, inference from return statements and function/method arguments are in other/later sections.

In place of the type, the keyword `auto` may be used, requiring the compiler to infer the type from the initializer or right-hand side of the statement or expression. Consider the following declarations:

```
auto zero(0);
auto one{1};
auto twoptwo{2.2};
auto place{"Yellowstone"};
auto anotherPlace{std::string("Everglades")};
auto thatPlace{"Capital Reef"s}; // using namespace std::string_literals
const auto deduced{2.46f};
```

- The type for `zero` is `int`, because the `0` literal used for initialization is an `int`.
- The type for `one` is `int`, because the `1` literal is an `int`.
- The type for `twoptwo` is `double`, because the literal `2.2` is a `double`.
- The type for `place` is `char*`, because that is the type for the literal `"Yellowstone"`. In order to force the inferred type to be a `std::string`, you must construct an `std::string` on the right-hand side. Alternatively, can use a string literal, but need to bring in the `std::string_literals` namespace.
- The type for `deduced` is `float`, because the literal `2.46f` is `float`, due to the trailing `f` on the number. Additionally, notice this is a constant, types for constants can also be inferred.

Inferring arrays requires some discussion. Consider the following statement:

```
auto primes = { 2, 3, 5, 7 };
```

Your expectation might be that `primes` is some type of array. Instead, rather than `primes` being an array, the inferred type is an `std::initializer_list<int>`. If you try to access the elements of `primes` using the array brackets, e.g., `primes[0]`, a compiler error is given, because it isn't an array.

What is an `std::initializer_list`? An `std::initializer_list` is an array-like structure that provides access to a list of objects of the type specified in the template parameter. This type is accepted as a parameter type in many of the standard library container constructors. The constructors accept the list, then iterate over the objects and add them to their internal storage. The following code shows how an (inferred) `std::initializer_list` can be used to initialize an `std::vector`:

```
auto primes = { 2, 3, 5, 7 };
std::vector primes1 = primes;
```

The variable `primes1` uses the `primes` `std::initializer_list` to set the initial values of the container.

An `std::array`, unlike `std::vector`, cannot be initialized from an `std::initializer_list` in the same way. However, both `std::vector` and `std::array` can be inferred from a partial declaration like the following:

```
std::vector primes2{ 2, 3, 5, 7 };
std::array primes3{ 2, 3, 5, 7 };
```

Dependent Type Inference

It is possible to set the type of a variable based on the type of another using the `decltype` keyword. It is as simple as this:

```
int a{4};
decltype(a) b{a};
```

Consider the following code:

```
int a{4};
decltype(a) b{a};
decltype(auto) c{a};
decltype(auto) d{(a)};

std::cout << a << " " << b << " " << c << " " << d << std::endl;
c = 6;
std::cout << a << " " << b << " " << c << " " << d << std::endl;
d = 8;
std::cout << a << " " << b << " " << c << " " << d << std::endl;
```

The first `decltype(auto)` deduces the type from `a`, which is an `int`. The second `decltype(auto)` deduces the type from `a`, but makes it a reference type (and I don't know why, that's just how this syntax works).

Now, if the type of `a` is changed (e.g. to an `unsigned int`), the compiler will infer the other types accordingly.

Side Trip – Reporting C++ Types

Here is some code from Stack Overflow that we'll use to report the type of a value:

```
//
// Reference: https://stackoverflow.com/questions/81870/is-it-possible-to-print-a-
variables-type-in-standard-c/20170989#20170989
//
template <typename T>
constexpr auto type_name()
{
    std::string_view name, prefix, suffix;
#ifdef __clang__
    name = __PRETTY_FUNCTION__;
    prefix = "auto type_name() [T = ";
    suffix = "]";
#elif defined(__GNUC__)
    name = __PRETTY_FUNCTION__;
    prefix = "constexpr auto type_name() [with T = ";
    suffix = "]";
#elif defined(_MSC_VER)
    name = __FUNCSIG__;
    prefix = "auto __cdecl type_name<";
    suffix = ">(void)";
#endif
    name.remove_prefix(prefix.size());
    name.remove_suffix(suffix.size());
    return name;
}
```

We can use this with the following code to report on the inferred types above:

```
std::cout << type_name<decltype(a)>() << std::endl;
std::cout << type_name<decltype(b)>() << std::endl;
std::cout << type_name<decltype(c)>() << std::endl;
std::cout << type_name<decltype(d)>() << std::endl;
```

Arrays

There are three arrays types to know in C++. The first comes from the C language and is what I will call a *raw array*. The second is is an array type called `std::array` which was added to the standard library as part of the C++11 standard. The third is an array type called `std::vector` that has long been available in the standard library.

This section provides an initial glance at arrays, but more detail on them is covered in later sections, particularly when looping and the standard library are discussed.

Raw Array

Raw arrays can be of any dimension, single or multi dimensional. The declaration of a single-dimensioned raw array uses the following general form:

```
type name[size];
```

Where type is any valid data type, including user defined data types, name is any valid identifier, and size is the number of elements. This is different from the typical Java declaration where the array brackets go with the type, on the left side of the identifier, also not needing to specify the number of elements. An array declaration like this creates the array on the stack, as opposed to the heap. In Java, all arrays are dynamically allocated on the heap. In C++ arrays can be stack or heap allocated (or combination stack and heap allocated!), based on how they are declared and constructed. *Heap allocated arrays are discussed in the section on dynamic memory allocation.*

Upon declaration (or allocation), raw arrays cannot change their size. Raw arrays do not know their size (unlike Java arrays). In fact, raw arrays have no methods on them, they are not a class type, they are a fundamental type, part of the C++ language itself. It is the responsibility of the programmer to separately track the size of the array as necessary.

The following code illustrates the declaration and initialization of single-dimensioned raw arrays.

```
int primes1[4];

primes1[0] = 2;
primes1[1] = 3;
primes1[2] = 5;
primes1[3] = 7;

int primes2[] = { 2, 3, 5, 7 };
```

The first statement declares and creates room for 4 `int` values in the array. The next four statements assign values to those array elements. The last statement shows the declaration and initialization of an array from a set of values. Note the array declaration doesn't include a size, it is optional. In this statement, the size of the array is determined by the number of values on the right-hand side of the equals sign. Both arrays are equivalent.

Let's say we have a custom data type named `Person` and want to create an array of these objects. The declaration looks like:

```
Person employees[10];
```

Similar to the previous example using the `int` primitive, this creates an array of 10 elements and the space for the 10 employees; *all on the stack*. This is different from Java, where you have to first create the array, which only creates an array of `null` references, and then separately dynamically allocate room for each object. It is possible to make such an array, on the heap, in C++, but that discussion waits until dynamic memory is discussed.

Multi-dimensional arrays are declared like the following:

```
int table[3][3];
```

```
table[0][0] = 0;
table[0][1] = 1;
table[0][2] = 2;
table[1][0] = 3;

int table2[3][3] = { { 0, 1, 2 }, { 3, 4, 5 }, { 6, 7, 8 } };
```

The first statement declares a 3 x 3 two-dimensional array; each dimension may be of a different size, this example happens to be a square array. The last statement declares the same size and type of array, but initializes the values at the time of declaration. Note that with multi-dimensional array declaration and initialization in a single statement, the sizes of the dimensions must be specified, the compiler will not determine them based on the right-hand side initializer.

Raw arrays of additional dimensions are declared and accessed simply by adding more array brackets following the same pattern.

With the addition of `std::array` with the C++11 standard, and also the long standing `std::vector` type, I don't recommend the use of raw arrays unless necessary for interfacing with legacy or third party code.

std::array

The `std::array` is a fixed size container (array) data type that is part of the standard library. Unlike raw arrays, this array knows its size, through the `.size` method. This array is a template data type, found in the `<array>` header, where the type and number of elements are specified as part of the declaration. We haven't yet discussed C++ templates, but at a cursory level (for now) they can be considered as quite similar to Java Generics. The following code demonstrates different `std::array` declarations and initializations.

```
std::array<int, 4> primes3;

primes3[0] = 2;
primes3[1] = 3;
primes3[2] = 5;
primes3[3] = 7;

std::array<int, 4> primes4 = { 2, 3, 5, 7 };
```

The first statement declares and creates room for 4 `int` values in the array; same as the raw array declaration. The first template parameter is `int`, specifying the array element type, the second template parameter is the number of elements in the array. It is worth reminding, in C++, the elements of the array are instances of the type, regardless of the type being a primitive or a class-based type. In Java, if the element type is a primitive, the elements are values, whereas if the element type is class-based, the elements are references to the types, with the initial value of the references being null.

The last statement shows the declaration and initialization of an array from a set of values. In this case, unlike raw arrays, the second template for the array size is required.

Using the same `Person` example from the raw arrays discussion, the array declaration looks like:

```
std::array<Person,10> employees;
```

This creates an array of of `Person` values (instances) of size 10.

Using the built in `std::string` type, an array of strings can be declared and initialized like:

```
std::array<std::string, 4> cities { "New York", "Chicago", "Denver", "Los Angeles" };
```

The `std::array` type only provides for single-dimensional arrays. However, multi-dimensional arrays are possible, by specifying the type for an `std::array` to be another `std::array`, like the following example:

```
std::array<std::array<int, 3>, 3> table3
       { { { 0, 1, 2 }, { 3, 4, 5 }, { 6, 7, 8 } } };
```

In all of these array cases, the values can be read/written by indexing with the array bracket operators `[]`.

std::vector

The `std::vector` is a dynamically sized container (array) data type that is part of the standard library. Similar to the `std::array` type, this is a template data type, but found in the `<vector>` header, where only the type of the array elements is specified. The following code demonstrates different `std::vector` declarations and initializations.

```
std::vector<int> primes5;

primes5.push_back(2);
primes5.push_back(3);
primes5.push_back(5);
primes5.push_back(7);

std::vector<int> primes6(4);

primes6[0] = 2;
primes6[1] = 3;
primes6[2] = 5;
primes6[3] = 7;

std::vector<int> primes7 { 2, 3, 5, 7 };
```

The first declaration creates a vector type named `primes5`, however. Because no initial storage is specified, the next 4 statements show how to add a new item. The method `.push_back` adds a new item to the vector. If the array brackets `[]` had been used to (attempt to) assign values, an exception would be thrown because no elements currently exist in the array.

The second declaration shows how to declare a vector with an initial number of elements, using one of the `std::vector` constructors, in this case 4. Because an initial size is specified, the next four statements are able to use the array brackets to assign (new) values to the elements.

The third declaration shows an example of declaring a vector and initializing it with values, in the same pattern as the other vector types.

In addition to being able to add new elements to the end, new elements can be inserted or removed at any location using the `.insert` and `.erase` methods.

The next two examples demonstrate creating an `std::vector` of strings and a multi-dimensional `std::vector` of integers. Notice the similarity between the `std::array` and `std::vector`.

```
std::vector<std::string> cities2 { "New York", "Chicago", "Denver", "Los Angeles" };

std::vector<std::vector<int>> table4
       { { { 0, 1, 2 }, { 3, 4, 5 }, { 6, 7, 8 } } };
```

Because a vector's size can change during run-time, in particular it may grow, the internal implementation may reserve space for future elements. It may be the case when a new element is added, enough reserved space already exists to immediately place the value. In another case, there may not be reserved space available, and new space has to be made at the time of insert. Because of this, the actual time to insert elements can vary.

Conditionals

Decision making, or conditionals, are the same, syntactically and semantically, as what you are already used to in Java, with only a few important differences. All of the following conditionals are available in C++

- `if`
- `if else`
- `switch`
- Ternary operator `? :`

Any nesting of these statements are also possible.

In Java the condition in the expression must evaluate to a `boolean` (`true` or `false`). In C++ the condition can evaluate to either a boolean or a numeric value. If the expression results in a numeric value, a value of non-zero is considered `true`, whereas a zero value is considered as `false`. The following examples illustrate:

```
if (0)        // resolves to false
if (0.00)     // resolves to false
if (1)        // resolves to true
if (2)        // resolves to true
if (4.44)     // resolves to true
```

The reason for this is due to C++'s roots in the C language. At the time C++ was developed, C did not have a boolean type, instead integers were used to represent booleans. In order to maintain reasonable compatibility with C, C++ continued with the practice of interpreting numeric expressions in boolean conditions.

Initializers

Both `if` and `switch` statements allow for an initialization statement. The next code sample demonstrates this with an `if` statement.

```
if (std::string message = getMessage(); message.size() > 0)
{
	std::cout << message << std::endl;
}
```

The first part of the `if` condition declares `message` and makes a call to the function `getMessage()` to set the value. The right-hand side of the condition, the variable `message`, is used in the test for true/false.

Switch Statements

The `switch` statement works as it does in Java, with one important difference. C++ does not allow strings (`std::string`) to be a type for the switch condition. The `switch` condition must be an integral or enumeration type.

As mentioned earlier, the C++ `switch` also allows for an initialization statement, alongside the condition. The following code demonstrates this:

```
switch (int input = getUserInput(); input)
{
	case 1:
		std::cout << "1 selected" << std::endl;
		break;
	case 2:
		std::cout << "2 selected" << std::endl;
		break;
	default:
		std::cout << "something else selected" << std::endl;
```

```
        }
```

The first part of the switch condition declares `input` and makes a call to the function `getUserInput()` to set the value. The right-hand side of the condition, the variable `input`, is used for testing against the different cases.

C++ has the concept of attributes, introduced in the C++11 standard. Attributes provide a standard syntax for different vendors to provide vendor-defined language extensions. There are, however, some C++ standard attributes. One of these attributes can be used in a switch statement to tell the compiler that the fall-through of case statements is expected, therefore, to not generate a compiler warning. The next code example demonstrates this:

```
        switch (int input = getUserInput(); input)
        {
                case 1:
                        std::cout << "1 was selected" << std::endl;
                        [[fallthrough]];
                case 2:
                        std::cout << "1 or 2 selected" << std::endl;
                        break;
                case 3:
                        std::cout << "3 was selected" << std::endl;
                case 4:
                        std::cout << "3 or 4 selected" << std::endl;
                        break;
                default:
                        std::cout << "something else selected" << std::endl;
        }
```

Notice the `[[fallthrough]]` attribute after the last statement for `case 1` and before `case 2`. This tells the compiler the fall-through is expected. Between `case 3` and `case 4` no attribute is placed, therefore, the compiler might generate a warning.

Also, interesting to note that a semi-colon is required after the `[[fallthrough]]` attribute. Most other attribute uses do not necessitate a semi-colon after their use.

C++ switch on string (finally with C++11, using constexpr: https://dev.krzaq.cc/post/switch-on-strings-with-c11/, http://www.rioki.org/2016/03/31/cpp-switch-string.html) : Show this later in the semester, putting the note here for now.

Comparing Strings

Comparing strings works differently in C++ than it does in Java. In Java when comparing two strings (or any objects) using the equality operator, `==`, the comparison is with respect to the value of the references, not the content of the strings. To compare strings in Java, the `.equals` method is used to compare content of one string with another. In C++, when comparing `std::string` values, the `==` compares the contents of the strings. The reason this works, is because the `std::string` class defines the behavior of the `==` operator (details on this to come later)! The following code demonstrates this…

```
        std::string message1 = "Hello World!";
        std::string message2 = "Hello ";
        message2 += "World!";

        if (message1 == message2)  // Resolves to true
```

The purpose of building `message2` in two statements is to prove that no common string optimization is being performed, causing both `message1` and `message2` to refer to the exact same string in memory; it wouldn't be the case or matter anyway, because `message1` and `message2` are not reference types, they are values on the stack.

Loops

Loops are quite similar, syntactically and semantically, as what you are already used to in Java, but there are some important differences. The following loop types are available in C++

- `while`
- `do {} while`
- `for`
    - of the form *`for (init statement; loop condition; iteration expression) {}`* Referred to as a counted for loop.
    - of the form `for (range declaration : range expression) {}` Referred to as a range-based for loop.

The `while`, `do {} while`, and counted `for` loops are all the same as Java, with the note that the condition may resolve to either a boolean or a numeric value; interpreted the same as described for conditional statements. It is the range-based for loop that requires further discussion.

Java provides what is called a *for each* loop, where each item in the container is iterated over during the loop. For example, each item in an array is touched, as the following Java code illustrates:

```
int[] primes = { 2, 3, 5, 7 };
int sumOfPrimes = 0;
for (int prime : primes) {
        sumOfPrimes += prime;
}
```

A loop of this form doesn't require the declaration or update of an array index. It presents all items in the array in the loop body.

The range-based for loop in C++ is a *for each* loop, and follows the same syntactical pattern as Java. There are some requirements for what is called the *range expression* (the sequence to iterate over). It can be a raw array, or any object that provides compatible `.begin` and `.end` member methods, or compatible `begin` and `end` free functions. The topic of `.begin` and `.end` member/free methods is going to be pushed until much later when iterators are discussed, but for now, it is enough to know that all containers in the standard library (e.g., `std::array` and `std::vector`) provide them.

The following code segment demonstrates a range-based for loop over a `std::array`:

```
std::array primes{ 2, 3, 5, 7 };
int sumOfPrimes = 0;
for (int prime : primes)
{
        sumOfPrimes += prime;
}
```

Type inference may be used for the *range declaration*, as shown in the following code segment. Much more about type inference and range-based for loops is to come.

```
for (auto prime : primes)
{
        sumOfPrimes += prime;
}
```

The code for other compatible sequences, such as raw arrays and `std::vector`, is exactly the same, including type inference.

Functions

I use the term *function* (and sometimes *free function*) to mean a function that is not part of a class. I use the term *method* to refer to a *function* that is part of a class. Java has only methods, whereas C++ allows for both functions and methods.

From the *Hello World* program, the syntax of a function has already been detailed. Some additional discussion is worthwhile, however.

Before a function can be called, the C++ compiler must already know of its existence. For code contained in a single C++ file, this means that a function must be defined before any calls to it are made. The code below demonstrates this:

```
std::array<int, 4> byTwo(std::array<int, 4> values)
{
    for (std::uint8_t i = 0; i < values.size(); i++)
    {
        values[i] *= 2;
    }

    return values;
}

int main(int argc, char* argv[])
{
    std::array primes{ 2, 3, 5, 7 };

    primes = byTwo(primes);
    for (auto prime : primes)
    {
        std::cout << prime << std::endl;
    }

    return 0;
}
```

The definition of the `byTwo` function comes before it is used in the `main` function. If the `byTwo` function is placed below the `main` function, the compiler would report an error, indicating the `byTwo` identifier was not found.

An alternative to defining the function before `main` is to provide a separate declaration and definition of the function. A simple declaration, or prototype, of a function may be first defined, then at a later location in the file (or in another file!) the definition, or implementation, of the function is provided. The next code sample demonstrates this:

```
std::array<int, 4> byTwo(std::array<int, 4> values);

int main(int argc, char* argv[])
{
    std::array primes{ 2, 3, 5, 7 };

    primes = byTwo(primes);
    for (auto prime : primes)
    {
        std::cout << prime << std::endl;
    }

    return 0;
}

std::array<int, 4> byTwo(std::array<int, 4> values)
{
    for (std::uint8_t i = 0; i < values.size(); i++)
    {
        values[i] *= 2;
    }
```

```
        return values;
    }
```

The first line in this sample is the function prototype. A function prototype includes the function return type, name, and parameters. This informs the compiler to expect a full function definition at a later time, and enabling other code to call to the function without it having yet been fully implemented. The function implementation can come anytime after the prototype.

Function Parameters – Pass-by-Value

By default, parameters are *pass-by-value*, which is the same as Java; meaning the value is copied. Be aware that C++ provides an additional way to pass parameters knows as *pass-by-reference*; detailed in the next section on *Reference Types*. In the examples above, the parameter named `values` is copied. When an `std::array` is copied, the entire array is copied. Therefore, `values` is not a reference to the array at the call site, instead, it is a copy of that array. This is a tricky subject in C++ because the programmer must be aware of how each object type handles itself when a copy is made. More on this to come as pointers and reference data types are discussed.

Function Parameters – Default Values

Parameters may specify a default value; Java does not have this capability. Default values are specified by declaring the parameter, followed by an equals sign and then an appropriate initializer. The following function prototype illustrates:

```
std::array<int, 4> byN(std::array<int, 4> values, int n = 2);
```

The `n` parameter has a default value of 2. When calling a function that has parameters with default values, arguments for those parameters may be omitted, allowing the defaults to be used. Therefore, this function may be called by omitting the second argument or by providing a value, as shown in the next two statements.

```
auto result = byN(primes);
auto result = byN(primes, 4);
```

In the first case, `n` takes on the default value of 2, in the second case, `n` takes on the value of 4.

If a function has separate prototype and implementation, it is recommended to specify the default values at the prototype. Either place, but not both, is allowed, but recommended at the prototype to ensure the compiler is aware of the defaults as it compiles code that make calls to the function.

More than one parameter may have a default. If any parameter has a default, all other parameters to its right must also have default values. When calling a function with multiple parameters with default values, the function call must fill arguments with default parameters from left to right, there is no ability to omit an argument and then specify the next one. The following code sample illustrates:

```
std::vector<int> createArray(int size = 10, int initialValue = 0);

...

auto array1 = createArray();            // Use both defaults
auto array2 = createArray(20);          // Specify size
auto array3 = createArray(20, 1);       // Specify both size and initialValue
```

(yes, I do know that std::vector provides a constructor to do this)

The prototype provides defaults for both parameters. Therefore, this function may be called in three different ways, with zerio, one, or two arguments. It isn't possible to call the function in a way that accepts the default value for `size`, while specifying a value for `initialValue`.

Enumerations

There are two types of enumerations in C++, with similar concept as enumerated types in Java. In both cases, an enumeration is a user defined type where the individual values are specified.

Unscoped Enumeration

The first type of enumeration is known as an unscoped enumeration. The following demonstrates the definition and use:

```
enum City
{
    NewYork,
    Chicago,
    Denver,
    LosAngeles
};

int main(int argc, char* argv[])
{
    City thisCity = Chicago;

    switch (thisCity)
    {
        case NewYork:    std::cout << "New York" << std::endl; break;
        case Chicago:    std::cout << "Chicago" << std::endl; break;
        case Denver:     std::cout << "Denver" << std::endl; break;
        case LosAngeles: std::cout << "Los Angeles" << std::endl; break;
    }

    return 0;
}
```

The `enum` declaration specifies the name (any valid identifier) and the set of values. Note in the `switch` statement the distinct values for the enumeration are used, no scoping to the `City` type is necessary (but is allowed). In this example, only the names for the values are declared, however, specific integer values may be set at declaration. The next example demonstrates this:

```
enum City
{
    NewYork         = 0,
    Chicago         = 1,
    Denver          = 2,
    LosAngeles      = 3
};
```

In both cases, the underlying type of `City` is an integral type whose size is large enough to accommodate the range of user defined values. Additionally, only some of the `enum` values may have initializers, any uninitialized values are set by the compiler.

Scoped Enumeration

A scoped enumeration requires the type to be specified along with the value. Scoped enumerations are also known as *strongly typed* enumerations. This is different from the unscoped enumeration where the type is not necessary when referring to the value. Scoped enumerations solve the problem of two different enumerations having the same names for their values, but different integers associated with them.

The following code shows the use of scoped enumerations:

```
enum class State
{
    Alabama,
    Arizona,
    Hawaii,
    Kansas
};

int main(int argc, char* argv[])
{
    State thisState = State::Alabama;

    switch (thisState)
    {
        case State::Alabama:    std::cout << "Alabama" << std::endl; break;
        case State::Arizona:    std::cout << "Arizona" << std::endl; break;
        case State::Hawaii:     std::cout << "Hawaii" << std::endl; break;
        case State::Kansas:     std::cout << "Kansas" << std::endl; break;
    }


    return 0;
}
```

When defining a scoped enumeration, the keyword `class` is used immediately after the `enum` keyword. Notice also, that when referring to the values of the enumeration, they must be scoped with the type. In C++, the `::` is known as the scope resolution operator.

The underlying type for a scoped enumeration can be specified by placing it after the `enum` identifier name. This is shown in the next code segment; assigning values to the enumerations is not required, only shown as something you can do.

```
enum class State : unsigned char
{
    Alabama   = 0,
    Arizona   = 1,
    Hawaii    = 2,
    Kansas    = 3
};
```

Reference Types

A reference is an alias to a variable, a variable that already exists. Any use, read or write, to the reference is performed against the variable on which the reference is aliased. Let's look at an example.

```
double goldenRatio = 1.6180339887;
double pi = 3.14159;
double& alias = goldenRatio;

std::cout << "The golden ratio is " << alias << std::endl;
alias /= 2;
std::cout << "The 1/2 of the golden ratio is " << goldenRatio << std::endl;
```

The `&` symbol is used in combination with any data type to declare a reference type, this is then followed by the name of the reference (alias). *The reference must be initialized at the time of declaration*, as shown above; references cannot be uninitialized. Once initialized, it cannot be changed to bind to another object; a compiler might allow this, but not all of them.

Notice the division by `2` on `alias` in the code above results in the original `goldenRatio` value being divided by 2; because it is a reference.

It is easy to be confused by the term reference in C++ and the same term, reference, in Java. They are not the same, don't let yourself try to think of a C++ reference as similar to a Java reference. Instead, a Java reference is more like a C++ pointer, which is discussed in the next major topic.

Function Parameters – Pass-by-Reference

Let's look at something that is trivially possible, without tricks in C++, but not in Java...a swap function!

```
void swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}

...

std::cout << "Before swap: a = " << a << ", b = " << b << std::endl;
swap(a, b);
std::cout << "After swap:  a = " << a << ", b = " << b << std::endl;
```

The `swap` function defines the `x` and `y` parameters to be references to `int`s. Therefore, any changes made to either the `x` or `y` values during the `swap` function are actually happening to the arguments to the function. The parameter `x` becomes an alias for the argument `a`, the parameter `y` becomes an alias for the argument `b`.

Illegal Use of References

A reference can not refer to a constant or a literal; references must refer to mutable values. The following two statements are not valid uses of references:

```
swap(a, 6);

const double PI = 3.14159;
double& z = PI;
```

The call to `swap` is invalid because it isn't possible to form a reference to the literal `6`. Similarly, the initialization of `z` to the constant `PI` isn't allowed because z is not a constant; could do: `const double& z = PI;`

Another example of an incorrect attempt to use a reference is the following:

```
int& doubleValue(int n)
{
    return 2 * n;
}
```

This isn't allowed because the result of the `2 * n` expression is not mutable. More technically, it isn't a non-const lvalue. *Stay tuned, we are going talk about how to get a reference to that expression. I promise to bend your mind!*

The following code compiles, but does not result in valid code, it is easy to be tricked into thinking it should work.

```
int& twiceValue(int n)
{
    int twice = 2 * n;

    return twice;
}


int& numberDouble = twiceValue(10);
std::cout << "Twice the value is " << numberDouble << std::endl;
```

When this code is run, on my computer, the output shows "`Twice the value is 32762`", which is definitely incorrect. What is going on here?

It is legal to return a reference to a value from a function, that isn't the mistake. The mistake is that the local variable `twice` is allocated on the stack. When the `twiceValue` function exists, the `twice` variable goes out of scope and is no longer valid. However, at the time the return statement was executed, it was a valid lvalue (left-value), but once the function scope ended, the source of the alias is no longer valid and any operation on the alias is undefined.

While it is possible to return references to values from a function, it isn't all that common, but does offer great benefits in some contexts. Reference return values are typically to heap allocated values. Because we haven't yet covered dynamic memory, returning valid references from functions will wait until that subject is covered.

Multiple Files

A single C++ program may be composed of any number of implementation files. In Java, if a class or (static) method is used in a file different from where it is implemented, the compiler is able to resolve the existence or location automatically. In C++ it isn't as simple.

Let's say we want to create a file with a bunch of common utility functions and some constants. In order to expose these functions and constants to other code, we'll place the function prototypes and the constant declarations into a *header* file. In a separate (*implementation*) file, we'll place the function implementations.

Soon we will be covering classes where you see that C++ has separate class declaration and implementations.

Side note: There are many different extension naming standards for header and implementation files. I use the extension `.hpp` for C++ header files and `.cpp` for C++ implementation files.

For this discussion, the next two sections of code show the header and implementation code for our utilities.

--- utilities.hpp ---

```
#pragma once

#include <cmath>

const double PI = 355.0 / 113.0;
const double GOLDEN_RATIO = (1.0 + std::sqrt(5.0)) / 2.0;

double sin(double angle);
void swap(int& x, int& y);
```

--- utilities.cpp ---

```
#include "utilities.hpp"

double sin(double angle)
{
    return (4 * angle * (180 - angle)) / (40500 - angle * (180 - angle));
}

void swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

This code shows placing the constants and function prototypes in a header file, followed by a separate implementation file for the function definition. A few details of note about this code:

- The `#pragma once` directive in `utilties.hpp` instructs the compiler to only process the code in this file one time, regardless of how many times the file is included. It is expected the `utilities.hpp` file will be included in many different places. Without the `#pragma once` directive, the compiler would attempt to redefine the constants and prototypes, resulting in compiler errors.
- The `<cmath>` library is included because `std::sqrt` is used in the `GOLDEN_RATIO` constant definition.
- The `utilities.cpp` file includes the `utilities.hpp` file. This is necessary to ensure the function definitions for the prototypes are matched up.
- The file `utilities.cpp` uses double quotes `""` for the include, instructing the compiler to first search the project folders for the file.

With the utilities code prepared, it can now be used by other parts of the project. The following sample shows how the utilities code is used in the `main.cpp` file:

--- main.cpp ---

```
#include <iostream>

#include "utilities.hpp"

int main(int argc, char* argv[])
{

    std::cout << "Pi estimate: " << PI << std::endl;
    std::cout << "Golden ratio estimate: " << GOLDEN_RATIO << std::endl;

    for (int angle = 0; angle <= 180; angle++)
    {
        std::cout << "sin(" << angle << ") = " << sin(angle) << std::endl;
    }

    return 0;
}
```

In order for the compiler to understand the `PI` and `GOLDEN_RATIO` constants, along with the `sin` function, the `utilities.hpp` file is included. The compiler will insert the `utilities.hpp` file into the `main.cpp` file, ensuring the code has been processed and stored in its memory while compiling the code in `main.cpp`.

Namespaces

Let's say we want to compare the results of the sin function we wrote in `utilities.cpp` with the standard library `sin` function. How do we make calls to two different sin functions? Namespaces to the rescue…

Namespaces are a way to organize code in a way that prevents conflicts in naming of things. We've already been using namespaces throughout this introduction. Every time the prefix `std::` is used, that is referring to the standard library *namespace*. All of the standard library code is contained within the `std` namespace.

Namespaces are somewhat similar to Java packages in terms of organizing code within identifiers. In Java, however, packages come with specific folder organization requirements, whereas C++ doesn't. The name of a namespace doesn't say anything about where the code must be located in the filesystem.

The following illustrates how to place the code in a namespace:

--- utilities.hpp ---

```
#pragma once

#include <cmath>

namespace mymath
{
	const double PI = 355.0 / 113.0;
	const double GOLDEN_RATIO = (1.0 + std::sqrt(5.0)) / 2.0;

	double sin(double angle);
	void swap(int& x, int& y);
}
```

The above code places all of the utilities constants and prototypes into the namespace `mymath`. A similar namespace declaration is necessary for the implementation file.

--- utilities.cpp ---

```
#include "utilities.hpp"

namespace mymath
{
    double sin(double angle)
    {
        return (4 * angle * (180 - angle)) / (40500 - angle * (180 - angle));
    }

    void swap(int& x, int& y)
    {
        int temp = x;
        x = y;
        y = temp;
    }

}
```

With the utilities code all placed into a namespace, there are two ways to refer to any of the constants or functions.

The first is to prefix the code with the namespace `mymath`, like `mymodule::PI`. The `::` symbol is known as the *scope resolution operator*. In this case, the scope is `mymath`, and is necessary in order to resolve to the `PI` constant in the `mymath` namespace.

The second way is to have a `using namespace` clause that resolves the namespace for the current scope. For example:

```
using namespace mymath;

int main(int argc, char* argv[])
{
    for (int angle = 0; angle <= 180; angle++)
    {
        double result = std::sin((angle * PI) / 180.0);

        std::cout << "std::sin(" << angle << ") = " << result << std::endl;
    }
       return 0;
}
```

The statement `using namespace mymath;` brings the `mymath` namespace into the current scope, automatically resolving any code in the namespace without requiring the use of the `mymath::` namespace prefix. In the code above, the `PI` constant can stand along, not needing the `mymath::` prefix.

Many developers, including me, recommend against the use of a `using namespace` at the file scope. The reason for this is to avoid inadvertent naming collisions for when the same identifier is used in different namespaces. I recommend, if reasonable, to go ahead and prefix code with the namespace, as you seen throughout these notes. You might find it appropriate to have a `using namespace` declaration at a function scope to reduce some unwieldy namespace prefixes, this is generally acceptable. As with many things in software development, this is by no means a settled issue, there are many different opinions on the use of namespaces in C++ code.

Some additional notes about namespaces:

- Namespaces may be nested inside other namespaces.
- Code for the same namespace may be declared/defined in multiple files. In other words, the code in multiple files can all have the same namespace declaration resulting in all of the code belonging to the same namespace.
- Code can be added to an anonymous namespace by omitting the namespace identifier.

Namespace Aliases

A namespace alias allows you to compress a long nested namespace. Consider the following code:

```
namespace outer
{
    namespace inner
    {
        namespace goodstuff
        {
            std::array<int, 4> getPrimes()
            {
                return { 2, 3, 5, 7 };
            }
        }
    }
}
```

Calling this function looks like: `auto primes = outer::inner::goodstuff::getPrimes();`

If a lot of functions inside the `goodstuff` namespace are being used, prepending the full set of namespaces begins to get laborious. Instead, a using can be used to give an alias to the namespace, like the following:

```
namespace gs = outer::inner::goodstuff;
auto primes = gs::getPrimes();
```