C++ Programming

More Basics

More Basics

- Functions
- Arrays
- Preprocessor Directives (#include …)
- Namespaces
- I/O

Functions

- Functions in C++ are similar to functions in C, with a few additions and options that aren't present in C
- Functions must be declared before they can be called, otherwise the compiler won't know what to do with them
- Functions also have to be defined; that is, somewhere the body of the function must be given along side its declaration
  - Note that a function can be declared in one spot (like a header file) and then defined elsewhere (in a source code file)

Function Declarations

- Function declarations in C++ contain several things
  - The name of the function
  - The argument or parameter list for the function (which could be empty)
  - The return type of the function (which could be void if the function is not going to be returning anything)
  - Optionally, one or more modifiers designating special behaviour of the function or modifying how the compiler treats or compiles it (this is where things can get ugly and deviate from C …)

Function Declarations

- For example, consider the following declaration:

```
void swap (int *, int *);
```

- This does the following:
  - Declares a function called `swap`
  - Tells us that this function will take two integer pointers as parameters
  - Tells us that this function will not be returning anything

Function Definitions

- A function definition is a function declaration alongside a presentation of the body of the function
- The function body provides the code to process the parameters given as input to produce the desired return result
- Note that to use parameters in the function body, they must be named in the accompanying declaration (otherwise, how else do you refer to them)
  - Declarations that are not definitions do not need to have their parameters named, as we saw on the previous slide

Function Definitions

- For example, consider the following definition:

```
void swap (int *p, int *q) {
   int t = *p;
   *p = *q;
   *q = t;
}
```

- What does this code do?

Function Calls

- A function is called by its name, identifying parameters to pass in and specifying what to do with the value it returns, if any
- The parameters passed in to the function must match the types given in the function declaration
- Similarly, the operation carried out on what is returned (assignment, condition, etc.) must also match the function declaration by type

Function Calls

- What happens when I call our `swap` function from this code?

```
int main() {
  int a, b;
  a = 1;
  b = 2;
  swap(&a, &b);
}
```

Function Parameters

- As in C, C++ defaults to passing parameters into functions by value
- In other words, the parameter’s value at call-time is copied into the function into a local variable named in the declaration that accompanies the function definition
- Any changes in value within the function stay within the function, only impacting the local variable

Function Parameters

```
void Two (int x) {
   x = 2;
   cout << x << endl;
}

void One () {
  int y = 1;
  Two (y);
  cout << y << endl;
}
```

The output when function One() is called:

2
1

Function Parameters

- What about our `swap` function though?
- Recall that it exchanges the values in the two variables pointed to by its parameters!
- Does this not break the pass by value rules?

Function Parameters

- Recall the function definition for `swap`:

```
void swap (int *p, int *q) {
  int t = *p;
  *p = *q;
  *q = t;
}
```

- The `swap` function doesn't modify its parameters. They are still copied in as pass by value states they should. Because they are pointers, however, we can access what they point at nonetheless …

Function Parameters

- C++ also allows parameters to be passed by reference by designating them with an & in the function declaration
- The variable in the function definition is not a local variable in this case, but rather an alias to the variable outside of the function and passed into it as a parameter
- As a result, changes in value within the function propagate outside the function as well
- Only use pass by reference when this behaviour is necessary, otherwise things can get rather confusing!

Function Parameters

```
void Two (int& x) {
   x = 2;
   cout << x << endl;
}

void One () {
  int y = 1;
  Two (y);
  cout << y << endl;
}
```

The output when function One() is called:

2
2

Function Parameters

- When we pass a parameter, we may not want to change the value of the parameter
- To ensure that we do not inadvertently do that, we can declare the parameter as `const`
- Recall that the keyword `const` is a commitment to not modify something and so the compiler will treat it as a constant and not let us modify it

Function Parameters

- Consider, for example, the following function named `Two`:

```
void Two (int const& x)
{
   x = 2;                // NOT ALLOWED.
   cout << x << endl; // This is OK.
}
```

Returning from a Function

- There are a a few ways that a function can exit:
  - It executes a `return` statement, providing a return value of the appropriate type (or not providing anything in the case of a `void` function)
  - Reaching the end of the function body, which is only allowed in `void` functions or `main()`, in which case this indicates successful completion
  - Calling a system function that does not return (like `exit()`)
- Exception throwing and handling can also cause a function to exit (more on exceptions later)

Function Modifiers

- `inline`
  - The compiler should try to embed the code for the function where it is called from, typically for performance reasons
- `constexpr`
  - The compiler should evaluate the results of calling the function at compile time, making this usable with `constexpr` constants
- `static`
  - The function is not visible outside of its file / translation unit
- Plus many, many others …

Arrays

- An array is a series of elements of the same type stored in contiguous memory locations that can be individually referenced
- Arrays in C++ work mostly the same as they do in C, with a few additional bits thrown in
  - You can use new and delete to manage dynamically allocated arrays instead of only malloc and free
  - Newer C++ standards also allow you to have initializer lists to dynamically allocate an array and initialize its contents in one step

Arrays

- Creating simple static arrays:

```
int foo1[5]; // array "foo1" with five int elements
             // that are not initialized

int foo2[5] = {1, 2, 3, 4, 5};  // array "foo2"
                                // initialized with five
                                // consecutive numbers
```

Arrays

- Using arrays (following the previous declarations):

```
for (int count = 0; count < 5; count++) {
   foo1[count] = foo2[count];
   cout << foo1[count] << endl;
}
```

- Notice that indexing starts at 0 and there is no inherent bounds checking … you can fall off either end of the array and do nasty things, so you need to be careful here!

Arrays

- Arrays and pointers (following the previous declarations):

```
int *p;             // an integer pointer

p = &(foo1[2]);   // have p point to the second element
cout << foo1[2] << end;
cout << *p << endl;
```

- We can have pointers point to array elements, just like we could the original types

Arrays

- Arrays and pointers (following the previous declarations):

```
int *p;    // an integer pointer

p = foo1;  // the array can be treated as a pointer here
cout << foo1[0] << end;
cout << *p << endl;
```

- Another way of looking at this: `foo1 == &(foo1[0])`

Arrays

- As noted above, we can allocate and deallocate arrays in C++ using `new` and `delete`

```
int size = 5;
int *foo;
foo = new int[size];
for (int count = 0; count < size; count++) {
   foo[count] = count;
   cout << foo[count] << endl;
}
delete foo;
```

Arrays

- Arrays can be powerful, but can start getting complex, depending on how you use them
  - Arrays of structures
  - Arrays of pointers
  - Arrays of arrays (multidimensional arrays)
- Their complexity grows when they are dynamically allocated, as you are typically forced to switch between array [] and pointer * notation
- This is why C++ tries to promote other data structures like vectors (more on such things shortly!)

Preprocessor Directives

- In C++ (as in C), before compilation, a preprocessor goes through your code, doing a variety of things
- In particular, the preprocessor looks for directives that give it instructions on things it should do with or to your code
- Preprocessor directives begin with a # and generally take an entire single line of code
- Unlike regular code, they do not end with a ;

Preprocessor Directives

- You have already seen a couple of these so far …
- #include
    - Used to include the contents of another source file into the current file, like:

      ```
      #include <iostream> // C++ standard headers drop the .h
      ```

    - Generally, this is used to include header files containing various declarations, type definitions, other preprocessor directives, and so on
    - That said, you can include code files (and other things!) as well, but doing so is generally frowned upon

Preprocessor Directives

- To prevent multiple inclusion of the same header file (which can have bad consequences like duplicate definitions of various things) we can use preprocessor directives to create include guards:

```
#ifndef MYHEADER_H
#define MYHEADER_H

…

#endif
```

Preprocessor Directives

- #define
  - Used to define constants, replaced during compilation, as discussed earlier
  - Can also be used to define macros that are also processed before compilation, such as:

```
#define square(x)    (x * x)

cout << square(2) << endl;
```

Preprocessor Directives

- There are other directives as well for various purposes
  - Conditional compilation (#ifdef, #if, #elif, #endif, …)
  - Throwing errors (#error)
  - Line control (#line)
  - Various other things (generally under #pragma)

Namespaces

- Namespaces provide a method for explicitly defining scope to the identifiers within it (e.g. types, functions, variables, etc.)
- This allows us to logically organize our code better and avoid name collisions that can occur in large projects with multiple programmers (or when code is used from multiple sources)
  - Only one entity can exist with a particular name in a particular scope; otherwise we have a name conflict or collision
- Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes

Namespaces

- The syntax to declare a namespace is:

```
namespace identifier
{
    named_entities
}
```

Namespaces

- For example:

```
namespace myNamespace
{
    int a, b;
}
```

- The variables can be accessed from within their namespace normally, (as `a` and `b`), but if accessed from outside the `myNamespace` namespace, they have to be properly qualified with the scope operator (::) as `myNamespace::a` and `myNamespace::b`

Namespaces

- As a shorthand, we can declare that we are using a namespace, to introduce direct visibility of all the names of the namespace into the current code file
- Recall that entities (variable, types, constants, and functions) of the standard C++ library are declared within the `std` namespace, and we can avoid explicitly using `std::` each time we reference them by:

```
using namespace std;
```

Namespaces

- It is generally considered poor form to make use of the `using namespace` syntax in a header file
  - This can lead to inadvertently opening up access to namespaces in unanticipated ways, leading to name collisions and other problems
- It is also potentially dangerous to use multiple namespaces at a time with this declaration for similar reasons; if more than one namespace uses the same name, you've got problems
- For these reasons, some purists would suggest avoiding this syntax entirely and always explicitly identify scope when required to do so

Input/Output

- In C++, a standard library (`iostream`) provides streaming output and input objects similar to Java's System.out and System.in
- Two commonly used output stream objects:
  - cout: standard output stream (to the terminal, like C's stdout)
  - cerr: standard error stream (also to the terminal, like C's stderr)
  - Even though they both go to the terminal, that can be treated and redirected separately in different ways; also, cout is typically buffered while cerr is not
- One commonly used input stream object:
  - cin: standard input stream (from the terminal keyboard, like C's stdin)

Input/Output

- As an important note, the standard input/output functions in C are still accessible in C++
- This includes `printf()`, `fprintf()`, and so on
- Generally, programmers are encouraged to use C++'s input/output streams in a C++ program, but it is not uncommon to find C's functions in use regardless

Input/Output

- I/O operations in C++ are performed by operators (not methods or functions, per se)
  - Insertion operator (<<) does output
  - Extraction operator (>>) does input
- The direction of the symbols indicates the data's destination:
  - << inserts data on to an output stream (cout or cerr)
  - >> extracts data from an input stream (cin) into a variable

Insertion

- << is called the output or insertion operator
- General syntax:

```
cout <<  expression;
```

- For example:

```
cout << "You are "  << age  <<  " years old” << endl;
```

Extraction

- >> is called the input or extraction operator
- General syntax:

```
cin >> variable;
```

- The only thing that can go on the right of the extraction operator is a variable, as something is needed to store data extracted from the input stream

Insertion and Extraction

- Consider the following example:

```
int x;  // variable declaration

cout << "Enter a number: "; // print a prompt
cin >> x; // read value from terminal into variable x
cout << "You entered: " << x << endl; // echo
```

Limitations of Extraction

- The extraction operator works fine as long as a program’s user provides the right kind of data – a number when asked for a number, for example
- If erroneous data is entered (a letter instead of a number, for example), the extraction operator isn’t equipped to handle it; instead of reading the data, it puts a premature end to the extraction

Limitations of Extraction

- When using the extraction operator to read input characters into a string variable:
  - The >> operator skips any leading whitespace characters such as blanks and newlines
  - It then reads successive characters into the string, and stops at the first trailing whitespace character (which is not consumed, but remains waiting in the input stream)
  - You can use this to type check your input better, but the whitespace separation of input is still an issue …

Limitations of Extraction

- For example, consider this code:

```
string name;
cout << "Enter your name: ";
cin >> name;
cout << "You entered: " << name << endl;
```

- If you were to enter “John Doe” at the prompt, the variable `name` would only receive “John” and not the full “John Doe”

Limitations of Extraction

- Let’s try this again:

```
string name;
cout << "Enter your name: ";
getline(cin, name);
cout << "You entered: " << name << endl;
```

- If you were to enter “John Doe” at the prompt, the variable `name` now receives the full “John Doe”

Limitations of Extraction

- If we wanted to parse a line and extract and type check data from it, there are various methods in the `string` class to assist us
- Another option would be to use something called a `stringstream` that creates a stream-like interface to a `string`, allowing us to use the extraction operator on it in a smarter way
- Once we see discuss classes more in a bit, we can see how to do this sort of thing a bit better …

File Input/Output

- When reading data from a file, the programmer doesn't need to be concerned with interacting with a user

- This means prompts are unnecessary, and more than one piece of data can be extracted from the input stream and moved around using a single line of code

File Input/Output

- Earlier, we used the `iostream` standard library, which provides cin and cout as mechanisms for reading from/writing to standard input and standard output respectively

- We can also read from/write to files; this requires another standard C++ library called `fstream`

File Input/Output

- The `fstream` library provides the following classes to perform output and input of characters to/from files:

- `ofstream`: Stream class to write on files
- `ifstream`: Stream class to read from files
- `fstream`: Stream class to both read and write from/to files

File Input/Output

- As in C, in C++ you generally do the following to work with files:
  - Declare your variable for handling the file
  - Open the file using this variable, naming the file and (as necessary) the type of operation you will be performing on the file
  - Carry out operations (including >> and <<) and call functions to move data to/from the file
  - Close the file when done

File Input/Output

- Consider this for example. What will this do?

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  string line;
  ifstream file;
  file.open("text.txt");
  while (getline(file, line)) {
     cout <<  line << endl;
  }
}
```