C++ Programming

Functions and Arrays

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;
}
```

Swap two numbers in memory.

- 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);
}
```

swap the value of a and 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 …

what we change is the pointed values.

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] << endl;
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] << endl;
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!)