COSC 151: Intro to Programming: C++

Chapter 6
Functions

Chapter 6: Functions

- **Objectives**
  - Understand how to write functions
    - Understand why we would want to
  - Understand how to call a function
  - Understand how functions return values
  - Understand how scope is important in functions
  - Differentiate between function declarations and function definitions
  - Understand the different means to pass values
  - Understand that functions can be **overloaded**
    - And understand why that is important

C6: Writing a Function

- **If we have logic in our program that should be repeated, it makes sense to write that logic once, and use it everywhere necessary**
  - It makes our code more:
    - Readable
    - Testable
    - Scalable ---- allows multiple people to work an a problem
    - Reliable ----- errors will be isolated to the functions where they are introduced
- **We could write a factorial function…**

```
int fact(int val)
{
   int ret = 1;  // local variable to hold the
                 // result as we calculated it
   while(val > 1)
   {
      ret *= val--;  // assign ret*val to ret
                     // and decrement val
   }
   return ret;  // return the calculated value
}
```

C6: Calling a Function

- **To call our factorial function, we have to supply an int. The result is also an int.**

```
int result = fact(5);  // result is 120
```

- **A function call**
  - Initializes the functions parameters from the arguments
  - Transfers control to that function
- **Execution of a function**
  - Begins with definition and initialization of parameters
  - Ends when a return statement is encountered, or the function block ends

C6: Parameters and Arguments

- **Arguments**
  - Are initializers for a functions parameters
  - Argument types must match the parameters
    - Implicit (or explicit) conversions are accepted
  - Number of arguments must match the number of parameters

```
fact(“hello”);    // error, wrong type
fact();           // error too few arguments
fact(42, 10, 0); // error, too many arguments
fact(3.14);       // OK, argument converted to int
```

C6: Function Parameter List

- **A function parameter lits can be empty, but it must be present**

```
void f() { } // function taking no arguments
```

- **Each parameter must have a type, even if two parameters are the same type, it must be repeated**

```
void g(int x, y)      {} // error, type for y unknown
void g(int x, int y) {} // correct!
```

C6: Function Return Type

- **Most types can be used as the return type of a function**
- **A function that does not return a value has a return type of `void`.**

C6: Local Objects

- **Parameters and variables defined inside a function are referred to as local variables**
  - They are "local" to that function, and can hide a name from an outer scope

```
int var = 10;  // var object in this outer scope

void foo()
{
   int var = 20; // local var, hides the other one
   cout << var << endl; // prints 20
}
```

C6: Local Objects and Scope

- **Local function objects allow us to demonstrate the difference between**
  - Scope – where a name has meaning
  - Lifetime – when an object or variable exists

```
int x = 95;

void f(int& r)
{
   // r introduced into local scope here
   r -= 5;
}

// r is no longer in scope
f(x);
// but the object to which r referred (x), is still alive and well
cout << x << endl;  // prints 90
```

C6: Automatic Objects

- **An object that exists only while a block is executing are known as automatic objects**
  - Function parameters are automatic objects

```
int area(int len, int wid) // len and wid are
{                          // automatic objects
   int a = len * wid; // a is an automatic object
   return a;
}
```

C6: Function Declarations

- **Like any other name, a function must be declared before we can use it**
- **Like any other name, a function can be defined only once**
  - So, for functions it's useful to differentiate between declarations and definitions
- **A declaration looks like a function where a ; replaces the body**

```
int fact(int value);  // declaration of our fact function
```

C6: Function Declarations (contd)

- **We put function declaration in header files**
  - So we can #include that function where needed
- **We put function definitions in separate implementation files**
  - So, we need to compile more that just one .cpp file
- **For example, if fact is defined in fact.cpp**

```
g++ -std=c++11 -g main.cpp fact.cpp -o factorial
```

C6: Argument Passing

- **There are two ways to pass arguments to a function**
  - By Value
    - The argument's value is copied to the function
  - By reference
    - The parameters is bound to the argument

C6: Pass by Value

- **Non-Reference parameters are copied**
  - The value of the parameter cannot affect the argument

```
int fact(int val)
{
    int ret = 1;
    while(val > 1)
    {
        ret *= val--;  // val _is_ changed
    }
    return ret;
}
```

  - Although val is changed, that change has no effect on the argument.
    - Calling `fact(i)` **does not change the value of** `i`.

C6: Pass by Value (pointer types)

- **Pointers behave like any other non-reference type**
  - Copying a pointer gives two distinct objects
- **However, pointers give indirect access to the value to which they point.**

```
void reset(int* p)
{
   *p = 0;
   p = nullptr;  // changes only the local p
}

int i = 42;
reset(&i);
cout << i << endl;  // prints 0
```

C6: Pass by Reference

- **Operations on a reference, are operations on the the object to which the reference refers.**
  - Reference parameters work the same way

```
void reset(int& p)
{
   p = 0;
}

int i = 42;
reset(i);
cout << i << endl;  // prints 0
```

C6: Pass by Reference (contd)

- Copying large objects, large class types or large containers can be inefficient
- Passing by Reference allows us to avoid copying

C6: Pass by reference (contd)

- **Prefer const references when you can, it makes the code more flexible**

```
bool is_shorter(std::string& s1, std::string s2)
{
   return s1.size() < s2.size();
}

// the reference parameters in is_shorter avoid copies
// but since they are non-const references, we can't do this:
if(is_shorter(“Hello”, “Long String to Test”)) // ERROR :(
{
}

bool is_shorter(const std::string& s1, const std::string* s2)
{
   return s1.size() < s2.size();
}

// The const-references above allow us to fix that
if(is_shorter(“Foo”, “FooBar”)); // OK!
```

C6: Handling Command Line Arguments

- **The `main` function has another standard accepted form**
  - Up to now, we've used `int main()`
- **The other standard form allows us to handle command line arguments to our programs**

```
./myprogram inputfile.txt outputfile.txt
```

  - To handle the above, main must take the form:

```
int main(int argc, char** argv)
{
   cout << argc << endl;    // prints 3 (for 0, 1, 2)
   cout << argv[0] << endl; // prints ./myprogram
   cout << argv[1] << endl; // prints inputfile.txt
   cout << argv[2] << endl; // prints outputfile.txt
}
```

C6: Return Types and `return` statement

- **A `return` statement terminates the function that is currently executing and returns control to the point from which the function was called.**
  - There are two forms:

```
return; // suitable for functions that have no return value
return expression;
```

C6: Functions with No Return Value

- **A `return` statement with no value may only be used in a function that has a return type of `void`.**

```
void swap(int& v1, int& v2)
{
   if(&v1 == &v2) // same address means same object!
      return;  // stop now, ends this function

   int tmp = v1;
   v1 = v2;
   v2 = tmp;
   // no explicit return necessary, function ends here
}
```

C6: Functions that Return a Value

- **The second form returns the function's result**

```
return expression;
```

- **The type of the expression must be an appropriate (convertible) type to match the function's return type**
  - A function that returns bool must have a return statement with an expression that is convertible to bool

```
return 4; // OK, 4 convertible to bool
return false;  // OK
```

  - A function that returns std::string must have a statement with an expression that is convertible to std::string

```
return s; // if s is of type std::string, OK
return “This is OK too.”;
return 32.12; // ERROR
```

C6: How Values are Returned

- **The return value is used to initialize a temporary object**
  - The temporary object is the result of the function call

```
string concat(const string& s1, const string& s2)
{
  return s1 + s2;
}
```

  - **concat** returns a **temporary** string, the result of adding **s1** and **s2**,
    - THIS is an r-value

C6: List Initializing the Return Value

- **You can use list initialization for a return value**
  - This makes writing code much easier
  - Though sometimes, NOT clearer

```
std::vector<int> some_numbers(int i)
{
    if(i % 2) // i is odd, return 1, 3, 5, 7, 9
    {
        return {1, 3, 5, 7, 9};
    }

    // Otherwise, return 0, 2, 4, 6, 8
    return {0, 2, 4, 6, 8};
}
```

C6: Overloaded Functions

- **Functions with the same name but different parameters lists and appear in the same scope are overloaded**
  - This is a powerful idea – it allows us to create function with behavior based on the types given

C6: Overloaded Functions (contd)

- **Examples**

```
void print(int);
void print(double);
void print(const string&);
void print(const Sales_data&);
```

  - Why did the last two use `const type&`?
- **We can have as many `print()` functions as we like, as long as the parameter lists are different**

Final Thouhgts

- **We write functions to**
  - Isolate complicated logic
  - Make our programs more readable, testable, reliable and scalable
- **Arguments are passed to functions and are used to initialize the function parameters**
- **Functions can return values (but don't have to)**
- **Objects declared locally in functions are limited in scope to that function**
  - This includes the parameters!\
  - Remeber the difference between scope and lifetime

Final Thoughts (contd)

- **Automatic objects are objects that exist only while a block is executing**
- **Think about separating function declarations and function definitions**
  - And why that may be important
- **There are only two ways to pass arguments**
  - By Value
  - By Reference
  - Pointers aren't magic, don't think of them as so

Final Thoughts (contd)

- Command line arguments using

```
int main(int argc, char** argv)
```

- Functions that return a value have to return the correct type
- Non-Reference Values returned from functions are temporary
- We can overload functions if they have different parameter lists