Chapter 5: Functions for all subtasks

Instructor: Mark Edmonds

edmonds_mark@smc.edu

Header vs. Implementation files

- Typically, we'll write our function declarations in *header* and our function implementations in an *implementation* file.
- Header files have the extension `.h` and implementation files have the extension `.cpp`
- We separate the declaration from the implementation due to how C++ is compiled, but more details on that later.
  - The gist of this is that the header file can be included in as many files as you want with statements like `#include` but the implementation file can only be compiled once. If you include an implementation file in multiple files, you'll get a compiler error saying something about "multiple definitions"
- Let's look at an example of how to separate the function declaration from the implementation and how to use a function in a separate file. Remember this example from the last lecture notes? We'll use it as a reference:

```
1  #include <iostream>
2
3  using namespace std;
4
5  void print_squares(void);  // function declaration, to be moved to a
       header file
6
7  int main(void)
8  {
9    print_squares();
10   // amazing things
11   print_squares();
12   return 0;
13 }
14
15 // function implementation/definition, to be moved to an implementation
       file
16 void print_squares(void)
17 {
18   int i;
19   for(i=1; i <=5; i++)
```

```
20    {
21      cout << i*i << endl;
22    }
23  }
```

- Next, I'll show how to split these definitions into separate files

print_squares.h

```
1  #include <iostream>
2
3  using namespace std;
4
5  // declare the function. This will make any file that includes
       print_squares.h aware that a print_squares() function exists
6  void print_squares(void);
```

print_squares.cpp

```
1  // need to include the header so we have access to std::cout
2  // we should ONLY include the print_squares.h header. Any other
       includes should be directly placed inside print_squares.h (like <
       iostream>)
3  #include "print_squares.h"
4
5  void print_squares(void)
6  {
7    int i;
8    for(i=1; i <=5; i++)
9    {
10     cout << i*i << endl;
11   }
12 }
```

main.cpp

```
1  #include <iostream>
2
3  // need to include print_squares so we have access to the print_squares
       () function
```

```
4  #include "print_squares.h"
5
6  using namespace std;
7
8  int main(void)
9  {
10   print_squares();
11   // amazing things
12   print_squares();
13   return 0;
14 }
```

Call-by-reference parameters

- So far, when we've passed arguments to a function, those arguments were copied inside of the scope of the function, meaning any modification inside of the function will not affect the caller's variables.
- To illustrate this, consider the following:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int dummy_func(int param){
6    // this modification doesn't affect the variable that was passed into
         the function
7    param++;
8    return param;
9  }
10
11 int main(){
12   int a = 5;
13   int b = dummy_func(a); // a is copied to dummy_func
14   // since a was copied (and then the copied value was modified in
        dummy_func, then returned), the value of a in main does not change
15   cout << "a: " << a << ", b: " << b << endl;
16 }
```

- For call-by-reference, the basic idea is instead of copying arguments to a function, use the same underlying memory location to pass values into a function (i.e. instead of duplicating the caller's box when calling a function, use the same box).

  - Call-by-reference is also called “pass-by-reference”
- Call-by-reference prevents the value from being copied and instead tells the function to directly modify the variable stored in the caller’s scope
  - This is clearly useful!
  - So far, we’ve only been able to return a single data type, but if we can modify parameters in the caller’s scope, we have a way to “return” multiple values by telling the parameters “not to copy” into the function’s scope.
- To pass a variable by reference, we modify its type to include a & to indicate we wish to use a reference to the type:

```cpp
#include <iostream>

using namespace std;

int dummy_func(int& param){ // the int& indicates we want a reference to an int as the argument
  // this modification WILL affect param in the caller's scope
  param++;
  return param;
}

int main(){
  int a = 5;
  int b = dummy_func(a); // a is not copied to dummy_func, but instead is passed by reference
  // since a was NOT copied and instead passed by reference, the value of a in main DOES change
  cout << "a: " << a << ", b: " << b << endl;
}
```

Call-by-reference use cases

- But when would you use this? Why would you use this? There are two main cases:

1. “Returning” multiple values from a function
   - Previously, if we wanted to return a value from a function, we could return at most one value of the type specified by the `rtype` in the function signature
   - Pass-by-reference allows us to “return” values. For instance, `dummy_func` could be made a `void function`. We can then extend this to “return” multiple things from a function, by passing every “return” variable by reference:

```
void dummy_func(int& param){ // the int& indicates we want a reference to an int as the argument
  // this modification WILL affect param in the caller's scope
  param++;
  // we don't really need to return in this case, as param has already been modified in the caller's scope
}
```

2. Memory efficiency
    - Since there is no copy that occurs when we pass-by-reference, we can save time by preventing the computer from copying a large amount a data from the caller to callee (this will be relevant later once we learn about classes and structs)
    - To do this, you would typically pass by *constant* reference, so the callee cannot modify the data. For the function signature of `dummy_func` above could be: `int dummy_func (const int& param)` (note the `param++` line would then fail to compile, as we aren't allowed to modify a constant `int`)

reference.cpp

- This program showcases uses cases of pass-by-reference

```
// This program demonstrates how you can use reference parameters

#include <iostream>          // for std::cout and std::cin
using namespace std;         // supports cout

void get_values( int& input1, int& input2 );
void swap_values( int& var1, int& var2 );
void show_values( int value1, int value2 );

int main( )
{
  int first = 0, second = 0;

  get_values( first, second );
  show_values( first, second );
  swap_values( first, second );
  show_values( first, second );
  return 0;
}

```

```
21 // this effectively implements a "multiple return" as our two
       parameters are modified by the function
22 void get_values( int& input1, int& input2 )
23 {
24   cout << "Please enter two values: ";
25   cin  >> input1 >> input2;
26 }
27
28 // this also implments a "multiple return"
29 void swap_values( int& var1, int& var2 )
30 {
31   int temp = var2;
32   var2 = var1;
33   var1 = temp;
34 }
35
36 void show_values( int value1, int value2 )
37 {
38   cout << "value1 = " << value1 << " and value2 = " << value2 << endl;
39 }
```

Debugging & Testing

- It is very difficult to write correctly the first time. Even for the most experienced programmers
- Writing *tests* and *debugging* your code is a great way to ensure you find problems with your code *before* it causes a problem for your employer, client, or customer.
- In fact, this is the basis of the automated grading system used in this course.
  - When you submit your code, a series of tests are launched to check the output of your code against known correct values
  - Getting in the habit of writing test cases for your code is a very good idea and will make life less painful later on

Debugging Techniques

- Debugging is a bit of an art. One thing to keep in mind is to keep an open mind. It's extremely common to be quite certain you know the error is in one part of the code, when it's actually in another.
  - Implementing sanity checks as you code can help alleviate this; if you KNOW your program works up to point, you have a much better idea where the problem is. You need to have

actually written code to verify that the program works up to a certain point; don't just think it works up to that point. Prove it with a check.

- You can use the debugger in Visual Studio or XCode (or any IDE) to help step through and inspect variable values as your program executes.
  - Visual studio debugging guide
  - XCode debugging guide
  - All debuggers rely on a few concepts. They control where to stop executing and pause using *breakpoints*. Once the program is paused at a breakpoint, you can:
    * *continue* (continue running the program until the next breakpoint is hit or the program terminates)
    * *step over* (go to the next line of code, jumping over any function calls)
    * *step in* (if the current line is a function call, go inside of the function and pause)
    * *step out* (execute the current function to completion and pause at the next line of the caller's function)
    * If the current line is not a function call, *step over* and *step in* are equivalent
- The `assert` statement is extremely helpful to writing tests
  - Assert statements can be used to check an expected *pre-condition* or *post-condition* of a function
  - A *pre-condition* of a function is a condition that the function expects to be met before the function can correctly execute. For instance, a pre-condition may be that one of the arguments is greater than 0
  - A *post-condition* of a function is a condition that should be true after the function has executed. For instance, a post-condition may be the the return value is greater than an input argument
  - Include the `<cassert>` library to use the `assert` macro
  - As an example of how to use an assert, consider the following test for our original `dummy_func`:

```
#include <iostream>
#include <cassert>

using namespace std;

int dummy_func(int param){
  param++;
  return param;
}

int main(){
```

```
12    int a = 5;
13    int b = dummy_func(a);
14    assert(b == 6); // b should be incremented
15    assert(a == 5); // a should not have been modified
16    cout << "a: " << a << ", b: " << b << endl;
17  }
```

Exercises

1. Write a function to swap the values stored in two variables

```
1  // function definition to swap the values.
2  void swap(int &x, int &y) {
3    int temp;
4    temp = x; /* save the value at address x */
5    x = y;    /* put y into x */
6    y = temp; /* put x into y */
7    return;
8  }
```

2. Write a test case to verify that the `swap` function works correctly

```
1  #include <iostream>
2  #include <cassert>
3
4  using namespace std;
5
6  // function definition to swap the values.
7  void swap(int &x, int &y) {
8    int temp;
9    temp = x; /* save the value at address x */
10   x = y;    /* put y into x */
11   y = temp; /* put x into y */
12   return;
13 }
14
15 int main(){
16   int test1 = 2, test2 = 7;
17   swap(test1, test2);
18   assert(test1 == 7); // test1 should be swapped with original value of
         test2
```

```
19    assert(test2 == 2); // test2 should be swapped with original value of
          test1
20  }
```