Functions II

Return Values

Sometimes we're interested in the output (return value) of a function

- math functions are great examples

If I call the `sqrt()` function, I want it to return something!

- we want `sqrt(25)` to return the number 5

Likewise, we want `acos(-1)` to return the value of **π**

- for functions like these, the return value is important

Side Effects

Sometimes, though, we're just interested in a function's *side effects*

- a *side effect* is something that happens or changes as a result of calling a function
- we say that the *side effect* of using `cout` is that something gets printed to the screen

Think about the stream formatting functions we've used

- when we use `cout.precision(10)`, we're interested in the side effect (setting numeric precision for `cout`)
- we use `cout.setf(ios::fixed)` to set formatting flags (the side effect)
- same with `cout.width(10)`. We want the side effect!

What should functions like these return?

void functions

They don't have to return anything!

```
// this void function doesn't return anything
// it is only useful for its side effect
void sayHi() {
    cout << "Hi, user!" << endl; // output something
}
```

A void function:

- is useful only for its side effects (changing stream settings, outputting something, etc.)
- is just like other functions, but does not return a value
- has the word void as its return type (think of void as an anti-datatype that represents nothingness)

void functions

Assume we have this void function:

```
// this void function doesn't do or return anything
void doNothing() { }
```

Because its return type is void, this is represents an error:

```
// error: void value not ignored as it ought to be
int i = doNothing();
```

There is no return value to use!

- you cannot store a non-existent value in a variable, nor can you use it in any other way
- a variable whose type is void would be completely pointless, so you cannot create void variables.

void functions

void functions can use the return keyword!

```
// this void function doesn't return anything
// but it still uses the return keyword
void printCubeVolume(double sidelength) {
    if (sidelength < 0) { // if invalid value
        return;           // exit the current function
    }
    cout << pow(sidelength, 3) << endl; // display cube volume
}
```

return immediately exits the current function, even in non-void ones

- notice how there is no value after the return statement in a void function, though!

void functions

void functions can use the return keyword!

```
// this void function doesn't do or return anything
void doNothing() { }

// neither does this one, but it uses the return keyword ^_^
void doNothingAgain() {
    return;
}
```

If control reaches the end of a function, it returns automatically.

- this is okay for void functions
- not okay for functions that are *supposed* to return a value, though

void functions

void functions are just like other functions (not at all scary)

- they just don't return a value
- they're useful only for their side effects (output, modifying settings, etc)

Battlecruiser FTW!

Another use for void

void can also be used to indicate that a function takes no arguments

- put void inside of the parentheses (not at all necessary, though)

```
// this void function doesn't take any arguments
void doNothing() { }

// neither does this one, but it uses an extra word to do so!
void doNothingAgain(void) { }

// ooh, no arguments here, either!
double randomNumber(void) {
    return 3.0;
}
```

Argument Passing

There are two different ways to pass arguments to a function

- the difference between them is subtle, but significant

Pass by Value:

- this is what we've been doing
- the values used to call a function are copied into different variables for use by the function

Pass by Reference:

- if a variable is supplied as a pass-by-reference argument, the variable itself is passed to the function (not its value, but the actual variable)
- this means that changes made to a pass-by-reference argument will be reflected in the original variable

Value or Reference?

Pass-by-value or pass-by-reference?

```
// increments @x, which is pass-by-value, by one
void incrementValue(int x) {
    ++x;
}
// increments @x, which is pass-by-reference, by one
void incrementValue(int& x) {  // notice the &
    ++x;
}
```

The ampersand (&) specifies pass-by-reference

- without it, the argument is always pass-by-value

Pass-by-Value

Pass-by-value is the default argument type

- the value of a variable is copied into a new variable
- changes made to the argument inside the function will NOT be reflected in the original variable (whose value was copied)

This function has one pass-by-value argument:

```
// increments @x, which is pass-by-value, by one
void incrementValue(int x) {
    ++x;
}
```

Pass-by-Reference

Pass-by-reference requires the addition of an ampersand (&):

- the variable itself is sent to the function
- changes made to the pass-by-reference argument inside the function <u>also change</u> the original variable!

This function has one pass-by-reference argument:

```
// increments @x, which is pass-by-reference, by one
void incrementValue(int& x) {  // notice the &
    ++x;
}
```

Notice the ampersand before x!

- the ampersand specifies pass-by-reference

Pass-by-Reference

The ampersand (&) just has to be between the type and the identifier:

```
// & adjacent to argument type (works)
void incrementValue(int& x) { ++x; }


// & adjacent to argument identifier (works)
void incrementValue(int &x) { ++x; }


// & smells bad, so type and identifier avoid it (still works)
void incrementValue(int        &        x) { ++x; }
```

This does NOT work:

```
void incrementValue(&int x) { ++x; }
```

Pass-by-Value

Passing num by value to `incrementValue()`:

```
1  #include <iostream>
2  using namespace std;
3
4  // increments @x, which is passed by value, by one
5  void incrementValue(int x) {
6      ++x;
7  }
8
9  int main() {
10     int num = 3;
11
12     cout << "Original value of num: " << num << endl;
13     incrementValue(num);
14     cout << "Final value of num:    " << num << endl;
15
16     return 0;
17 }
18
```

The value of num (declared in main) does not change

- only its value (3) was passed to the function, NOT the variable itself

Pass-by-Reference

Passing num by reference to `incrementValue()`:

```
1  #include <iostream>
2  using namespace std;
3
4  // increments @x, which is passed by reference, by one
5  void incrementValue(int& x) {
6      ++x;
7  }
8
9  int main() {
10     int num = 3;
11
12     cout << "Original value of num: " << num << endl;
13     incrementValue(num);
14     cout << "Final value of num:    " << num << endl;
15
16     return 0;
17 }
18
```

notice the ampersand!

The value of num (declared in main) CHANGES!!!

- num itself is passed to the function, where it is then modified

The Address Operator (&)

The address operator yields the address of a variable

- pass-by-reference (using the &) is providing the address of a variable (the variable itself)
- pass-by-value is providing the value of a variable

```
1  #include <iostream>
2  using namespace std;
3
4  // increments @x, which is passed by value, by one
5  void incrementValue(int x) {
6      cout << "Address of argument:   " << &x   << endl; // unique address
7      ++x;
8  }
9
10 int main() {
11     int num = 3;
12
13     cout << "Original value of num: " <<  num << endl;
14     cout << "Address of num:        " << &num << endl; // unique address
15     incrementValue(num);
16     cout << "Final value of num:    " <<  num << endl;
17
18     return 0;
19 }
20
```

The Address Operator (&)

The address operator yields the address of a variable

- pass-by-reference (using the &) is providing the address of a variable (the variable itself)
- pass-by-value is providing the value of a variable

```
1  #include <iostream>
2  using namespace std;
3
4  // increments @x, which is passed by reference, by one
5  void incrementValue(int& x) {
6      cout << "Address of argument:   " << &x   << endl; // SAME address
7      ++x;
8  }
9
10 int main() {
11     int num = 3;
12
13     cout << "Original value of num: " <<  num << endl;
14     cout << "Address of num:        " << &num << endl; // SAME address
15     incrementValue(num);
16     cout << "Final value of num:    " <<  num << endl;
17
18     return 0;
19 }
20
```

Value or Reference?

If the variable is a primitive (int, double, bool, char) or an object:

- pass-by-value (the default behavior) is like we talked about
- pass-by-reference can be used by adding the address operator (&)

If the variable already holds an address (an array, for example):

- pass-by-value is still passing an address, so it will behave like pass-by-reference!
- using pass-by-reference (with the address operator) on such a variable is seldom useful

We'll talk about using arrays as arguments next week

- don't concern yourself with this tidbit until then =)

Value *and* Reference?

Sometimes you want to return multiple values from a function

- using just a return statement, this isn't possible*
- you can couple a return value with pass-by-reference arguments to simulate multiple return values, though!

See the demo "`returnAndReference.cpp`"

Pass-By-Reference Multi-Modify

Sometimes you want to return multiple values from a function

- using just a return statement, this isn't possible*
- you can use pass-by-reference arguments alone, if you want to

See the demo "`multiModify.cpp`"