Week 10: Functions 2

Gaddis: 6.5,7-10,13

CS 1428
Fall 2014

Jill Seaman

6.7 The `return` statement

- It is used to stop the execution of a function
- It can be placed anywhere in a function
  - the function immediately transfers control back to the statement that called it.
- Statements that follow the return statement will not be executed
  - unless the return is in an if-branch
- In a void function with no return statement, the compiler adds an implicit return statement before the last }

The `return` statement: example

```
void someFunc (int x) {
   if (x < 0)
      cout << “x must not be negative.” << endl;
   else {
      // Continue with lots of statements, indented
      // ...
      // so many it’s hard to keep track of matching {}
   }
}
```

This is equivalent, easier to read

```
void someFunc (int x) {
   if (x < 0) {
      cout << “x must not be negative.” << endl;
      return;
   }
   // Continue with lots of statements, less indentation,
   // no brackets to try to match ...
}
```

6.8 Returning a value from a function

- You can use the return statement in a function to send a value back to the function call:

```
return expr;
```

- The value of the `expr` will be sent back.
- The data type of the value that the function is returning must be placed in the function header:

Return type:

```
int doubleIt(int x) {
   return x*2;
}
```

Value being returned

Calling a function that returns a value

- If the function returns void, the function call is a statement:

  `pluses(4);`

- If the function returns a value, the function call is an expression:

  `int y = doubleIt(4);`

- The value of the function call is the value of the expr returned from the function, and you should do something with it.

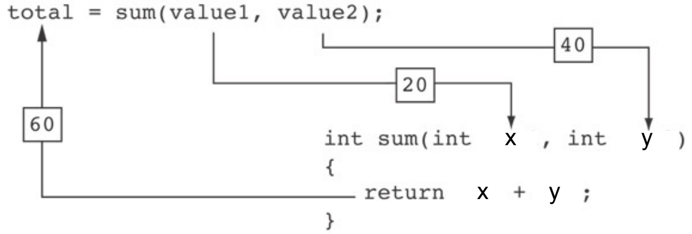
Returning the sum of two ints

```
#include <iostream>
using namespace std;

int sum(int,int);

int main() {
   int value1;
   int value2;
   int total;
   cout << "Enter 2 numbers: " << endl;
   cin >> value1 >> value2;
   total = sum(value1, value2);
   cout << "The sum is " << total << endl;
}

int sum(int x, int y) {
   return x + y;
}
```

Output:

```
Enter 2 numbers:
20  40
The sum is 60
```

Data transfer



- The function call from main: `sum(value1, value2)` passes the values stored in `value1` and `value2` (`20` and `40`) to the function, assigning them to `x` and `y`.
- The result, `x+y (60)`, is returned to the call and stored in `total`.

Function call expression

- When a function call calls a function that returns a value, it is an *expression*.
- The function call can occur in any context where an expression is allowed:
  - assign to variable (or array element) `total = sum(x,y);`
  - output via cout `cout << sum(x,y);`
  - use in a more complicated expression `cout << sum(x,y)*.1;`
  - pass as an argument to another function `z = pow(sum(x,y),2);`
- The value of the function call is determined by the value of the expression returned from the function.

6.9 Returning a boolean value

```
bool isValid(int number)
{
   bool status;
   if (number >=1 && number <= 100)
      status = true;
   else
      status = false;
   return status;
}
```

- the above function is equivalent to this one:

```
bool isValid (int number) {
   return (number >=1 && number <= 100);
}
```

Returning a boolean value

- You can call the function in an if or while:

```
bool isValid(int);

int main() {
   int val;
   cout << “Enter a value between 1 and 100: ”
   cin >> val;

   while (!isValid(val)) {
      cout << “That value was not in range.\n”;
      cout << “Enter a value between 1 and 100: ”
      cin >> val;
   }
   // . . .
```

6.5 Passing Data by Value

(review)

- Pass by value: when an argument is passed to a function, its value is copied into the parameter.
- Parameter passing is implemented using variable initialization (behind the scenes):

```
int param = argument;
```

- Changes to the parameter in the function definition do not affect the value of the argument in the call

---

Example: Pass by Value

```
#include <iostream>
using namespace std;

void changeMe(int);

int main() {
   int number = 12;
   cout << "number is " << number << endl;
   changeMe(number);
   cout << "Back in main, number is " << number << endl;
   return 0;
}

void changeMe(int myValue) {
   myValue = 200;
   cout << "myValue is " << myValue << endl;
}
```

Output:

```
number is 12
myValue is 200
Back in main, number is 12
```

```
int myValue = number;
```

changeMe failed!

Pass by Value notes

When the argument is a variable:

- The parameter is initialized to a *copy* of the argument's value.
- Even if the body of the function changes the parameter, the argument in the function call is unchanged.
- The parameter and the argument are stored in separate variables, separate locations in memory.

6.13 Passing Data by Reference

- Pass by reference: when an argument is passed to a function, the function has direct access to the original argument.
- Pass by reference in C++ is implemented using a reference parameter, which has an ampersand (&) in front of it:

```
void changeMe (int &myValue);
```

- A reference parameter acts as an *alias* to its argument.
- Changes to the parameter in the function **DO** affect the value of the argument

---

Example: Pass by Reference

```
#include <iostream>
using namespace std;

void changeMe(int &);

int main() {
   int number = 12;
   cout << "number is " << number << endl;
   changeMe(number);
   cout << "Back in main, number is " << number << endl;
   return 0;
}

void changeMe(int &myValue) {
   myValue = 200;
   cout << "myValue is " << myValue << endl;
}
```

Output:

```
number is 12
myValue is 200
Back in main, number is 200
```

myValue is an *alias* for number, only one shared variable

Using Pass by Reference for input

```
double square(double) {
   return number * number;
}

void getRadius(double &rad) {
   cout << "Enter the radius of the circle: ";
   cin >> rad;
}

int main() {
   const double PI = 3.14159;
   double radius;
   double area;
   cout << fixed << setprecision(2);
   getRadius(radius);
   area = PI * square(radius);
   cout << "The area is " << area << endl;
   return 0;
}
```

During the function execution, rad is an alias to radius in the main program.

Pass by Reference notes

- Changes made to a reference parameter are actually made to its argument
- The & must be in the function header AND the function prototype.
- The argument passed to a reference parameter must be a <u>variable</u> – it cannot be a constant or contain an operator (like +)
- Use when appropriate – don't use when:
  - the argument should not be changed by function (!)
  - the function returns only 1 value: use return stmt!

6.10 Local and Global Variables

- Variables defined inside a function are local to that function.
  - They are hidden from the statements in other functions, which cannot access them.
- Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.
  - This is not bad style. These are easy to keep straight
- Parameters are also local to the function in which they are defined.

Local variables are hidden from other functions

```
#include <iostream>
using namespace std;

void anotherFunction();

int main() {
   int num = 1;
   cout << "In main, num is " << num << endl;
   anotherFunction();
   cout << "Back in main, num is " << num << endl;
   return 0;
}

void anotherFunction() {
   int num = 20;
   cout << "In anotherFunction, num is " << num << endl;
}
```

Output:

```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is 1
```

This `num` variable is visible only in `main`

This `num` variable is visible only in `anotherFunction`

Local Variable Lifetime

- A function's local variables and parameters exist only while the function is executing.
- When the function begins, its parameters and local variables (as their definitions are encountered) are created in memory, and when the function ends, the parameters and local variables are destroyed.
- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

Global Variables

- A global variable is any variable defined outside **all** the functions in a program.
- The scope of a global variable is the portion of the program starting from the variable definition to the end of the file
- This means that a global variable can be accessed by all functions that are defined after the global variable is defined

---

Global Variables: example

```
#include <iostream>
using namespace std;

void anotherFunction();
int num = 2;

int main() {
   cout << "In main, num is " << num << endl;
   anotherFunction();
   cout << "Back in main, num is " << num << endl;
   return 0;
}

void anotherFunction() {
   cout << "In anotherFunction, num is " << num << endl;
   num = 50;
   cout << "But now it is changed to " << num << endl;
}
```

Output:

```
In main, num is 2
In anotherFunction, num is 2
But now it is changed to 50
Back in main, num is 50
```

Global Variables/Constants

Do not use global variables!!! Because:

- They make programs difficult to debug.
  - If the wrong value is stored in a global var, you must scan the entire program to see where the variable is changed
- Functions that access globals are not self-contained
  - cannot easily reuse the function in another program.
  - cannot understand the function without understanding how the global is used everywhere

It is ok (and good) to use global **constants** because their values do **not** change.

Global Constants: example

```
const double PI = 3.14159;

double getArea(double number) {
   return PI * number * number;
}

double getPerimeter(double number) {
   return PI * 2 * number;
}

int main() {
   double radius;
   cout << fixed << setprecision(2);
   cout << "Enter the radius of the circle: ";
   cin >> radius;

   cout << "The area is " << getArea(radius) << endl;
   cout << "The perimeter is " << getPerimeter(radius) << endl;
}
```

Output:

```
Enter the radius of the circle: 2.2
The area is 15.21
The perimeter is 13.82
```