CS1428 Review

Part II: Chapters 6-7 (+11)

CS 2308
Spring 2014

Jill Seaman

Function Definitions

- Function definition pattern:

```
datatype identifier (parameter1, parameter2, ...) {
  statements . . .
}
```

Where a parameter is:

```
datatype identifier
```

  - *datatype*: the type of data returned by the function.
  - *identifier*: the name by which it is possible to call the function.
  - *parameters*: Like a regular variable declaration, act within the function as a regular local variable. Allow passing arguments to the function when it is called.
  - *statements*: the function's body, executed when called.

Function Call, Return Statement

- **Function call** expression

```
identifier ( expression1, . . . )
```

  - Causes control flow to enter body of function named identifier.
  - parameter1 is initialized to the value of expression1, and so on for each parameter
  - expression1 is called an **argument**.
- **Return statement:** `return expression;`
  - inside a function, causes function to stop, return control to caller.
- The value of the return *expression* becomes the value of the function call

Example: Function

```
// function example
#include <iostream>
using namespace std;
int addition (int a, int b) {
   int result;
   result=a+b;
   return result;
}
int main () {
   int z;
   z = addition (5,3);
   cout << "The result is " << z <<endl;
}
```

- What are the parameters? arguments?
- What is the value of: `addition (5,3)`?
- What is the output?

Void function

- A function that returns no value:

```
void printAddition (int a, int b) {
   int result;
   result=a+b;
   cout << "the answer is: " << result << endl;
}
```

  - use void as the return type.
- the function call is now a statement (it does not have a value)

```
int main () {
   printAddition (5,3);
}
```


Prototypes

- In a program, function definitions must occur before any calls to that function
- To override this requirement, place a prototype of the function before the call.
- The pattern for a prototype:

```
datatype identifier (type1, type2, ...);
```

  - the function header without the body (parameter names are optional).

Arguments passed by value

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

```
int param = argument;
```

- Changes to the parameter in the function body do **not** affect the value of the argument in the call
- The parameter and the argument are stored in separate variables; separate locations in memory.

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 to change the argument!

Parameter passing by Reference

- Pass by reference: when an argument is passed to a function, the function has direct access to the original argument (no copying).
- 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, it is NOT a separate storage location.
- 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

Scope of variables

- For a given variable definition, in which part of the program can it be accessed?
  - **Global variable** (defined outside of all functions): can be accessed anywhere, after its definition.
  - **Local variable** (defined inside of a function): can be accessed inside the block in which it is defined, after its definition.
  - **Parameter**: can be accessed anywhere inside of its function body.
- Variables are destroyed at the end of their scope.

More scope rules

- Variables in the same exact scope cannot have the same name
  - Parameters and local function variables cannot have the same name
  - Variable defined in inner block can hide a variable with the same name in an outer block.

```
int x = 10;
if (x < 100) {
   int x = 30;
   cout << x << endl;
}
cout << x << endl;
```

Output:

```
30
10
```

- Variables defined in one function cannot be seen from another.

Arrays

- An **array** is:
  - A series of elements of the same type
  - placed in contiguous memory locations
  - that can be individually referenced by using an index along with the array name.
- To declare an array:

```
datatype identifier [size];
```

```
int numbers[5];
```

  - datatype is the type of the elements
  - identifier is the name of the array
  - size is the number of elements (constant)

Array initialization

- To specify contents of the array in the definition:

```
float scores[3] = {86.5, 92.1, 77.5};
```

  - creates an array of size 3 containing the specified values.

```
float scores[10] = {86.5, 92.1, 77.5};
```

  - creates an array containing the specified values followed by 7 zeros (partial initialization).

```
float scores[] = {86.5, 92.1, 77.5};
```

  - creates an array of size 3 containing the specified values (size is determined from list).

Array access

- to access the value of any of the elements of the array individually, as if it was a normal variable:

```
scores[2] = 89.5;
```

  - scores[2] is a variable of type float
- rules about subscripts (aka indexes):
  - they always start at 0, last subscript is size-1
  - the subscript must have type int
  - they can be any expression
- watchout: brackets used both to declare the array and to access elements.

Working with arrays and array elements

- An array element:
  - can be used exactly like any variable of the element type.
  - you can assign values to it, use it in arithmetic expressions, pass it as an argument to a function.
- Generally there are NO C++ operations you can perform over entire arrays.
  - you cannot assign one array to another
  - you cannot input into an array
  - you cannot compare one array to another

Example: Processing arrays

Computing the average of an array of scores:

```
const int NUM_SCORES = 8;
int scores[NUM_SCORES];
cout << "Enter the " << NUM_SCORES
     << " programming assignment scores: " << endl;

for (int i=0; i < NUM_SCORES; i++) {
   cin >> scores[i];
}

int total = 0;  //initialize accumulator
for (int i=0; i < NUM_SCORES; i++) {
   total = total + scores[i];
}
double average =
       static_cast<double>(total) / NUM_SCORES;
```


Arrays as parameters

- In the function definition, the parameter type is a variable name with an empty set of brackets: [ ]
  - Do NOT give a size for the array inside [ ]

```
void showArray(int values[], int size)
```

- In the prototype, empty brackets go after the element datatype.

```
void showArray(int[], int)
```

- In the function call, use the variable name for the array.

```
showArray(numbers, 5)
```

- An array is **always** passed by reference.

Example: Partially filled arrays

```
int sumList (int list[], int size) {//sums elements in list array
   int total = 0;
   for (int i=0; i < size; i++) {
      total = total + list[i];
   return total;
}
const int CAPACITY = 100;
int main() {
   int scores[CAPACITY];
   int count = 0;               //tracks number of elems in array
   cout << "Enter the programming assignment scores:" << endl;
   cout << "Enter -1 when finished" << endl;
   int score;
   cin >> score;
   while (score != -1 && count < CAPACITY) {
      scores[count] = score;
      count++;
      cin >> score;
   }
   int sum = sumList(scores,count);
}
```

sums from position 0 to size-1, even if the array is bigger.

pass count, not CAPACITY

The string class

- string literals: represent sequences of chars:

```
cout << "Hello";
```

- To define string variables:

```
string firstName, lastName;
```

- Operations include:
  - = for assignment
  - .size() member function for length
  - ==, <, ... relational operators (alphabetical order)
  - [n] to access one character

```
string name = "George";
for (int i=0; i<name.size(); i++)
   cout << name[i] << " ";
```

Structures

- A structure stores a collection of objects of **various** types
- Each element in the structure is a member, and is accessed using the dot member operator.

```
struct Student {
   int idNumber;
   string name;
   int age;
   string major;
};
```

Defines a new data type

```
Student student1, student2;
student1.name = "John Smith";
Student student3 = {123456,"Ann Page",22,"Math"};
```

Defines new variables

Structures: operations

- Valid operations over entire structs:
  - assignment: `student1 = student2;`
  - function call: `myFunc(gradStudent,x);`
- Invalid operations over structs:
  - comparison: `student1 == student2`
  - output: `cout << student1;`
  - input: `cin >> student2;`
  - Must do these member by member

Arrays of Structures

- You can store values of structure types in arrays.

```
Student roster[40];  //holds 40 Student structs
```

- Each student is accessible via the subscript notation.

```
roster[0] = student1;
```

- Members of structure accessible via dot notation

```
cout << roster[0].name << endl;
```


Overloaded Functions

- Overloaded functions have the same name but different parameter lists.
- The parameter lists of each overloaded function must have different types and/or number of parameters.
- Compiler will determine which version of the function to call by matching arguments to parameter lists

Example: Overloaded functions

```
double calcWeeklyPay (int hours, double payRate) {
   return hours * payRate;
}
double calcWeeklyPay (double annSalary) {
   return annSalary / 52;
}


int main () {
   int h;
   double r;
   cout << "Enter hours worked and pay rate: ";
   cin >> h >> r;
   cout << "Pay is: " << calcWeeklyPay(h,r) << endl;
   cout << "Enter annual salary: ";
   cin >> r;
   cout << "Pay is: " << calcWeeklyPay(r) << endl;
   return 0;
}
```

Output:
Enter hours worked and pay rate: 37 19.5
Pay is: 721.5
Enter annual salary: 75000
Pay is: 1442.31

Default Arguments

- A <u>default argument</u> for a parameter is a value assigned to the parameter when an argument is not provided for it in the function call.
- The default argument patterns:
  - in the prototype:

```
datatype identifier (type1 = c1, type2 = c2, ...);
```

  - OR in the function header:

```
datatype identifier (type1 p1 = c1, type2 p2 = c2, ...) {
 ...
}
```


- c1, c2 are constants (named or literals)

Example: Default Arguments

```
void showArea (double length = 20.0, double width = 10.0)
{
   double area = length * width;
   cout << "The area is " << area << endl;
}
```

- This function can be called as follows:

```
showArea();  ==> uses 20.0 and 10.0
The area is 200

showArea(5.5,2.0);  ==> uses 5.5 and 2.0
The area is 11

showArea(12.0);  ==> uses 12.0 and 10.0
The area is 120
```


Default Arguments: rules

- When an argument is left out of a **function call**, all arguments that come after it must be left out as well.

```
showArea(5.5);    // uses 5.5 and 10.0
showArea( ,7.1);  // NO, won't work, invalid syntax
```

- If not all parameters to a function have default values, the parameters with defaults must come last:

```
int showArea (double = 20.0, double);  //NO
int showArea (double, double = 20.0);  //OK
```