Week 3

Functions & Arrays

Gaddis: Chapters 6 and 7

CS 5301
Spring 2016

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);
}
```

5

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).

6

Arguments passed by value

- <u>Pass by value</u>: when an argument is passed to a function, its value is *copied* into the parameter.
- It is implemented using variable initialization (in the background):

  ```
  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.

7

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!

8

Parameter passing by Reference

- <u>Pass by reference</u>: 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

9

Example: Pass by Reference

```cpp
#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

10

Example: Boolean functions

```cpp
bool isEven(int number) {
    bool status;
    if (number % 2 == 0)
        status = true;  // number is even if there is no remainder.
    else
        status = false; // Otherwise, the number is odd.
    return status;
}

int main() {
    int val;
    cout << "Enter an integer and I will tell you ";
    cout << "if it is even or odd: ";
    cin >> val;

    if (isEven(val))
        cout << val << " is even.\n";
    else
        cout << val << " is odd.\n";
}
```

Returns a true or false

Function call used as a boolean expression

11

Arrays

- An **array** is:
  - A series of elements of the same type
  - placed in contiguous memory locations
  - that can be individually referenced by adding an index to a unique identifier.
- 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) [12]

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
   - use it anywhere a float variable can be used.
- rules about subscripts:
   - always start at 0, last subscript is size-1
   - must have type int but can be any expression
- watchout: square brackets are used both to declare the array and to access elements.

Arrays: operations

- Valid operations over entire arrays:
   - function call: `myFunc(scores,x);`
- **Invalid** operations over entire arrays:
   - assignment: `array1 = array2;`
   - comparison: `array1 == array2`
   - output: `cout << array1;`
   - input: `cin >> array2;`
   - Must do these element by element, probably using a for loop

Processing arrays

- <u>Assignment</u>: copy one array to another

```
const int SIZE = 4;
int oldValues[SIZE] = {10, 100, 200, 300};
int newValues[SIZE];

for (int count = 0; count < SIZE; count++)
    newValues[count] = oldValues[count];
```

- <u>Output</u>: displaying the contents of an array

```
const int SIZE = 5;
int numbers[SIZE] = {10, 20, 30, 40, 50};

for (int count = 0; count < SIZE; count++)
    cout << numbers[count] << endl;
```

Processing arrays

Summing and averaging 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;
                                                    17
```

Finding highest and lowest values in arrays

- <u>Maximum</u>: Need to track the highest value seen so far. Start with highest = first element.

```
const int SIZE = 5;
int array[SIZE] = {10, 100, 200, 30};

int highest = array[0];
for (int count = 1; count < SIZE; count++)
   if (array[count] > highest)
      highest = array[count];

cout << "The maximum value is " << highest << endl;
```
18

Comparing arrays

- <u>Equality</u>: Are the arrays exactly the same?
  Must examine entire array to determine true
  Only one counter-example proves it is false

```
const int SIZE = 5;
int firstArray[SIZE] = {10, 100, 200, 300};
int secondArray[SIZE] = {10, 100, 201, 300};

bool arraysEqual = true; //assume true, until proven false

for (int count = 0; count < SIZE && arraysEqual; count++)
   if (firstArray[count] != secondArray[count])
      arraysEqual=false;

if (arraysEqual)
   cout << "The arrays are equal" << endl;
else
   cout << "The arrays are not equal" << endl;     19
```

Arrays as parameters

- In the <u>function definition</u>, 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 <u>prototype</u>, empty brackets go after the element datatype.

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

- In the <u>function call</u>, use the variable name for the entire array.

  ```
  showArray(numbers, 5)
  ```
  20

- An array is **always** <u>passed by reference</u>.

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

21

Multidimensional arrays

- multidimensional array: an array that is accessed by more than one index

```
int table[2][5];    // 2 rows, 5 columns
table[0][1] = 10;   // puts 10 in first row,
                    // second column
```

- Initialization:

```
int a[4][3] = {4,6,3,12,7,15,41,32,81,52,11,9};
```

  - First row: 4,6,3
  - Second row: 12, 7, 15
  - etc.

22