Week 3
Functions & Arrays
Gaddis: Chapters 6 and 7

CS 5301
Spring 2015
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

```c++
// 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)

```c
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.

```c
# 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;
}
```

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

Example: Pass by Value

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:

```cpp
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

```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
```

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;
}
```

```cpp
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";
    }
}
```

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:

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

- 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:

```c
float scores[3] = {86.5, 92.1, 77.5};
```
- creates an array of size 3 containing the specified values.

```c
float scores[10] = {86.5, 92.1, 77.5};
```
- creates an array containing the specified values followed by 7 zeros (partial initialization).

```c
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:

```c
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:** brackets 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 structs:
  - assignment: `array1 = array2;`
  - comparison: `array1 == array2`
  - output: `cout << array1;`
  - input: `cin >> array2;`
  - Must do these element by element, probably using a for loop

Example: Processing arrays

Computing the average of an array of scores:

```c
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

```c
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]; //tracks number of elems in array
    int count = 0; //tracks number of elements
    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);
}
```

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.

Multidimensional arrays

- when using a 2D array as a parameter, you must specify the number of columns:
  ```c
  void myfunction(int vals[][3], int rows) {
    for (int i = 0; i < rows; ++i) {
      for (int j = 0; j < 3; ++j)
        cout << vals[i][j] << " ";
      cout << "\n";
    }
  }
  ```
  ```c
  int a[4][3] = {4,6,3,12,7,15,41,32,81,52,11,9};
  ```
  ```c
  myfunction(a,4);
  ```
Sample Problem 1

- **Celsius Temperature Table:** The formula for converting a temperature from Fahrenheit to Celsius is

\[
C = \frac{5}{9}(F - 32)
\]

where \( F \) is the Fahrenheit temperature and \( C \) is the Celsius temperature.

- Write a function named `celsius` that accepts a Fahrenheit temperature as an argument. The function should return the temperature, converted to Celsius.

- Demonstrate the function by calling it in a loop that displays a table of the Fahrenheit temperatures 0 through 20 and their Celsius equivalents.

Sample Problem 2

- **Larger Than \( n \):** In a program, write a function that accepts three arguments: an array, the size of the array, and a number \( n \). Assume that the array contains integers. The function should display all of the numbers in the array that are greater than the number \( n \).