Week 3
Functions, Arrays & Structures
Gaddis: Chapters 6, 7, 11
CS 5301
Spring 2014
Jill Seaman

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
```

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: 37.195  
Pay is: 721.5  
Enter annual salary: 75000  
Pay is: 1442.31

Default Arguments

- A default argument 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
**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 is the type of the elements
  - identifier is the name of the array
  - size is the number of elements (constant)

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

**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:

```cpp
class Example: Processing arrays

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 [ ]
  ```cpp
  void showArray(int values[], int size)
  ```
- In the prototype, empty brackets go after the element datatype.
  ```cpp
  void showArray(int[], int)
  ```
- In the function call, use the variable name for the array.
  ```cpp
  showArray(numbers, 5)
  ```
- An array is always passed by reference.

Example: Partially filled arrays

```cpp
class 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);
}
```

Multidimensional arrays

- multidimensional array: an array that is accessed by more than one index
  ```cpp
  int table[2][5];   // 2 rows, 5 columns
  table[0][1] = 10;  // puts 10 in first row, second column
  ```
- Initialization:
  ```cpp
  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:

```cpp
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";
    }
}
```

```cpp
int main() {
    int a[4][3] = {4,6,3,12,7,15,41,32,81,52,11,9};
    ...
    myfunction(a,4);
    ...
}
```

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.

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

```cpp
Student student1, student2, student3 = {123456,"Ann Page",22,"Math"};
```

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.
  ```cpp
  Student roster[40]; // holds 40 Student structs
  ```
- Each student is accessible via the subscript notation.
  ```cpp
  roster[0] = student1;
  ```
- Members of structure accessible via dot notation
  ```cpp
  cout << roster[0].name << endl;
  ```
Arrays of Structures

- Arrays of structures processed in loops:

```cpp
Student roster[40];

// input
for (int i=0; i<40; i++) {
    cout << "Enter the name, age, idNumber and " <<
         "major of the next student: \n";
    cin >> roster[i].name >> roster[i].age
        >> roster[i].idNumber >> roster[i].major;
}

// output all the id numbers and names
for (int i=0; i<40; i++) {
    cout << roster[i].idNumber << endl;
    cout << roster[i].name << endl;
}
```

Passing structures to functions

- Structure variables may be passed as arguments to functions:

```cpp
void getStudent(Student &s) { // pass by reference
    cout << "Enter the name, age, idNumber and " <<
         "major of the student: \n";
    cin >> s.name >> s.age >> s.idNumber >> s.major;
}

void showStudent(Student x) {
    cout << x.idNumber << endl;
    cout << x.name << endl;
    cout << x.age << endl;
    cout << x.major << endl;
}

// in main:
Student student1;
getStudent(student1);
showStudent(student1);
```