Functions, Arrays & Structs

Unit 1

Chapters 6-7, 11

CS 2308
Spring 2024

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.

Overloaded Functions

- <u>Overloaded functions</u> 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 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
```

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.

Arrays: operations

- Valid operations over entire arrays:
  - function call: `myFunc(scores,x);`
- **<u>Invalid</u>** 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

- Assignment: 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];
```

- Output: 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;
```

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;
```

Finding highest and lowest values in arrays

- Maximum: 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;
```

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 array.

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

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

Two-Dimensional Arrays

- Like a table in a spreadsheet: rows and columns
- Declaration requires two size declarators:

```
int table [5][3]; // 5 rows, 3 columns
```

- Rows are always first
- 2D arrays can be initialized:

```
int table [2][3] =
    { {1, 2, 3},
      {4, 5, 6} };
```

1	2	3
4	5	6

Two-Dimensional Array processing

- Access an element of the array using two indices:

```
int table [2][3] =
    { {1, 2, 3},
      {4, 5, 6} };
cout << table[0][2];
```

Output: 3

- Two dimensional arrays can be passed to functions.
- The number of **columns** is required in the parameter declaration:

```
void showTable (int array[][3], int rows) {
…
}
```

Two-Dimensional Array functions

- 2D array processing usually requires nested for loops:

```
void showTable (int array[][3], int rows) {
   for (int x=0; x<rows; x++) {
      for (int y=0; y<3; y++)
         cout << setw(4) << array[x][y] << " ";
      cout << endl;
   }
}
```

- How showTable is called:

```
int table [2][3] =
   { {1, 2, 3},
     {4, 5, 6} };

showTable(table,2);
```

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);`

```
void myFunc(Student, int); //prototype
```

- **<u>Invalid</u>** 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;
```

Arrays of Structures: initialization

- To initialize an array of structs:

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

int main()
{
    Student roster[] = {
        {123456,"Ann Page",22,"Math"},
        {111222,"Jack Spade",18,"Physics"}
    };

}
```

Arrays of Structures

- Arrays of structures processed in loops:

```
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;
}
```

Arrays of Structures as function arguments

- Arrays of structure may be passed as arguments to functions.

```
double avgAge(Student arr[], int size) {
   int total = 0;
   for (int i=0; i<size; i++)
      total = total + arr[i].age;
   return static_cast<double>(total)/size;
}

int main() {
  Student roster[250];  // array of 250 student structures

  //input information about students here (see slide 13)

  cout << "Average age is: " << avgAge(roster,250) << end;
}
```

Note: Student declaration must be global!!

Note: works for an array of <u>any</u> (provided) size