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

```cpp
// 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:

```cpp
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)

```cpp
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:

```cpp
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):

```cpp
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

```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 12
- 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
```

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.

```cpp
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 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)

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

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: 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
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
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;
student1.name = "John Smith";
Student 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

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

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; //copy student1 into 1st position`
- Members of structure accessible via dot notation
  `cout << roster[0].name << endl;`
Arrays of Structures: initialization

• To initialize an array of structs:

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

```cpp
int main()
{
    Student roster[] = {
        {111222,"Jack Spade",18,"Physics"}
    };
}
```

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: 
";
    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: 
";
    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);