Function Definitions

- **Function definition pattern:**

  ```
  datatype identifier (parameter1, parameter2, ...) {
  statements . . .
  }
  ```

  Where a parameter is:

  ```
  datatype identifier
  ```

- **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 `expression` becomes the value of the function call.

Example: Function

```
// function example
#include <iostream>
using namespace std;
int addition (int a, int b) {
  int r;
  r=a+b;
  return (r);
}
int main () {
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

- 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 r;
    r=a+b;
    cout << "the answer is: " << r << 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);
    return 0;
}
```

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
```

```
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
```

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

```cpp
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

```c
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:
  ```c
  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.
  ```c
  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:
  ```c
  int showArea (double = 20.0, double);  //NO
  int showArea (double, double = 20.0);  //OK
  ```

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 the definition.
  - Local variable (defined inside of a function): can be accessed inside the block in which it is defined, after the 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.

- Variables defined in one function cannot be seen from another.

```cpp
int x = 10;
if (x > 100) {
    int x = 30;
    cout << x << endl;
}
cout << x << endl;
```

Output:
```
30
10
```

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)

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

Array initialization

- To specify contents of the array in the definition:
  - creates an array of size 3 containing the specified values.
  - creates an array containing the specified values followed by 7 zeros (partial initialization).
  - creates an array of size 3 containing the specified values (size is determined from list).

```cpp
float scores[3] = {86.5, 92.1, 77.5};
float scores[10] = {86.5, 92.1, 77.5};
float scores[] = {86.5, 92.1, 77.5};
```

Array access

- To access the value of any of the elements of the array individually as if it was a normal variable:
  - scores[2] is a variable of type float
  - rules about subscripts:
    - 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.

```cpp
scores[2] = 89.5;
```
Working with arrays and array elements

- An array element:
  - can be used exactly like any variable of the element type.
  - you can assign values to it, use it in arithmetic expressions, pass it as an argument to a function.
- Generally there are NO operations you can perform over entire arrays.
  - you cannot assign one array to another
  - you cannot input into an array
  - you cannot compare one array to another

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 parameter
    ```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.