CIS 22A – Lecture 15

Manish Goel

Functions

- Functions – a collection of statements to perform some task

- Modular Programming – divide a program into smaller, manageable modules or functions
  - Eliminates duplication of code by combining it into function calls
  - Simplifies writing programs by minimizing changes
  - Improves maintainability by localizing bugs

- *"Divide and Conquer"* a large problem into smaller chunks of work that can be *"chained"* together

- `main` is the *"main"* and first function in any program– it can call other functions

Functions - pictorially

This program has one long, complex function containing all of the statements necessary to solve a problem.

```
int main()
{
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
}
```

In this program the problem has been divided into smaller problems, each of which is handled by a separate function.

```
int main()
{
   statement;
   statement;
   statement;
}
```

main function

```
void function2()
{
   statement;
   statement;
   statement;
}
```

function 2

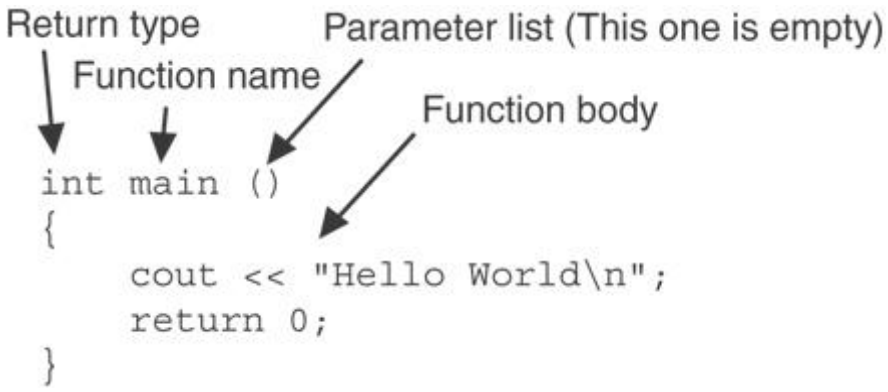
```
void function3()
{
   statement;
   statement;
   statement;
}
```

function 3

```
void function4()
{
   statement;
   statement;
   statement;
}
```

function 4

Defining Functions



- Definition includes:
  - return type: data type of the value that function returns to the part of the program that called it
  - name: name of the function. Function names follow same rules as variables
  - parameter list: variables containing values passed to the function
  - body: statements that perform the function's task, enclosed in { }

Return Types

- If a function returns a value, the type of the value must be indicated:

```
int main()
```

- If a function does not return a value, its return type is `void`:

```
void printHeading()
{
      cout << "Monthly Sales\n";
}
```

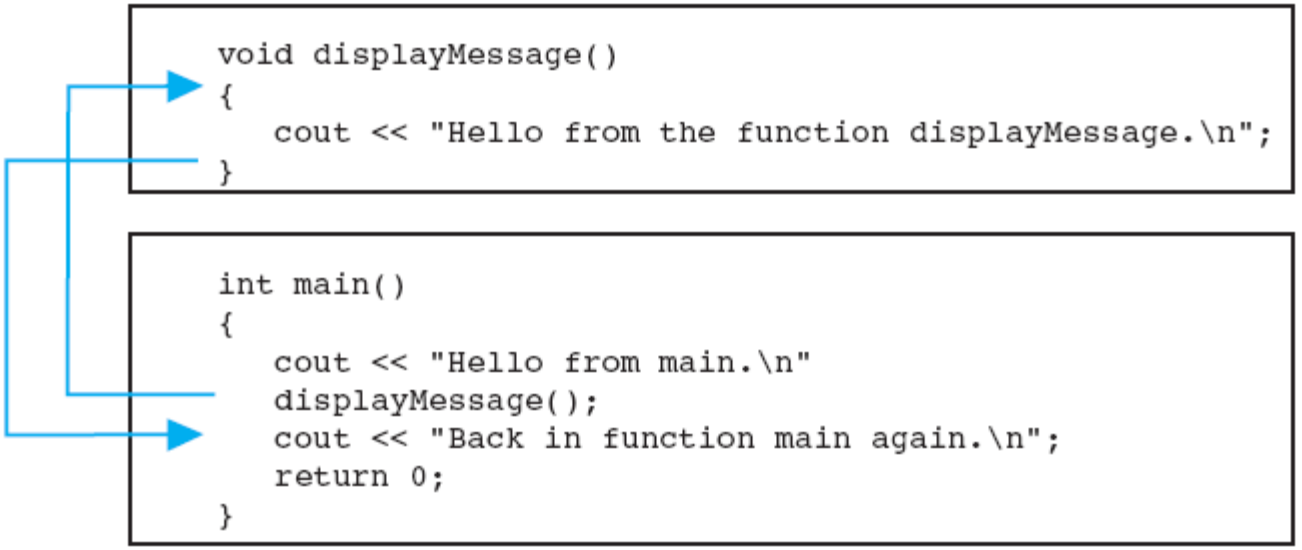
- Functions can also return boolean values also – true or false

Calling Functions

- To call a function, use the function name followed by `()` and `;`

```
printHeading();
```

- When called, program executes the body of the called function
- After the function terminates, execution resumes in the calling function at point of call.



```
void displayMessage()
{
   cout << "Hello from the function displayMessage.\n";
}

int main()
{
   cout << "Hello from main.\n"
   displayMessage();
   cout << "Back in function main again.\n";
   return 0;
}
```

Function Prototypes

- `main` can call any number of functions
- Functions can call other functions

- Compiler needs to know the following about a function before it is called:
  - name
  - return type
  - number of parameters
  - data type of each parameter

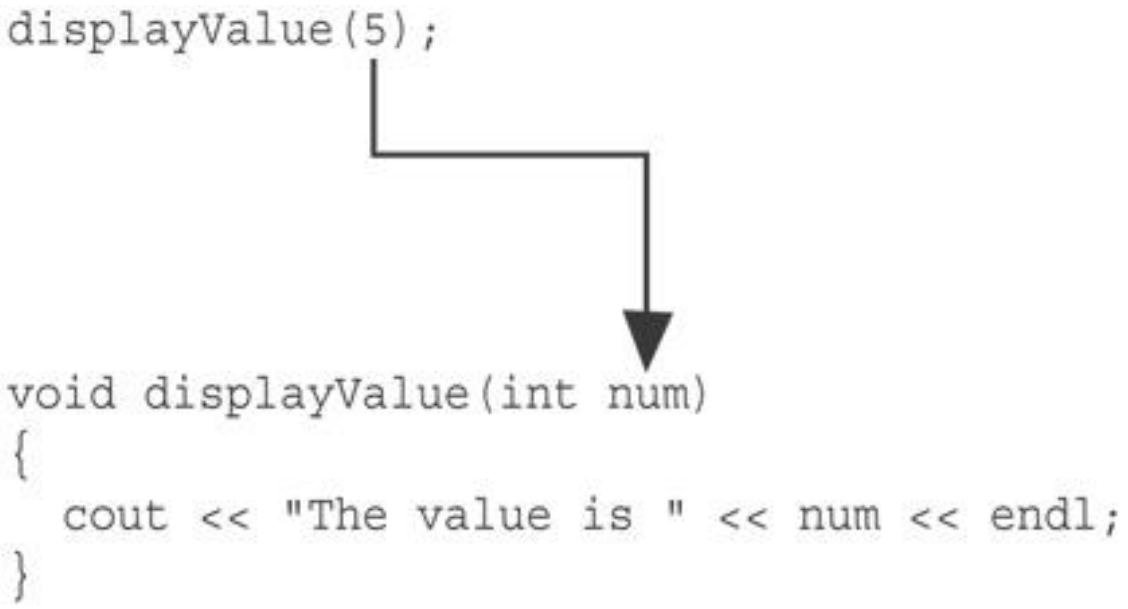
- To notify the compiler about a function before it is called
  - Place the function definition before it is called – usually placed after the pre-processor directives and globals and before `main`
  - Use a function prototype (function declaration) – like the function definition without the body – now function definition can be placed anywhere in the code
    - Header: `void printHeading()`
    - Prototype: `void printHeading();`

Sending data to functions

- Can pass values into a function at time of call:

```
c = pow(a, b);
```

- Values passed to function are arguments
- Variables in a function that hold the values passed as arguments are parameters



Parameters, Prototypes and Headers

- A parameter can also be called a formal parameter or a formal argument

- An argument can also be called an actual parameter or an actual argument

- For each function argument,
  - the prototype must include the data type of each parameter inside its parentheses
  - the header must include a declaration for each parameter in its ()

```
void evenOrOdd(int);  //prototype
void evenOrOdd(int num) //header
evenOrOdd(val);       //call
```

Function Call Notes

- Value of argument is copied into parameter when the function is called
- A parameter's scope is the function which uses it
- Function can have multiple parameters
- There must be a data type listed in the prototype（）and an argument declaration in the function header（）for each parameter
- Arguments will be promoted/demoted as necessary to match parameters
- When calling a function and passing multiple arguments:
  - the number of arguments in the call must match the prototype and definition
  - the first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.

Pass-by-Value

- Pass by value: when an argument is passed to a function, its value is copied into the parameter.

- Changes to the parameter in the function do not affect the value of the argument

- Example: `int val=5;`
  `evenOrOdd(val);`

- `evenOrOdd` can change variable `num`, but it will have no effect on variable `val`

Function Return

- `return` used to end execution of a function
- Can be placed anywhere in a function
  - Statements that follow the `return` statement will not be executed
- Can be used to prevent abnormal termination of program
- In a `void` function without a `return` statement, the function ends at its last `}`
- A value-returning function, `return` statement returns a value back to the statement that called the function

```
int sum(int num1, int num2)
{
  double result;
  result = num1 + num2;
  return result;
}
```

Local Variables

- Variables defined inside a function are *local* to that function. They are hidden from the statements in other functions, which normally cannot access them.
- Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.
- A function's local variables exist only while the function is executing. This is known as the *lifetime* of a local variable.
- Local variables are not automatically initialized. They must be initialized by programmer.
- When the function begins, its local variables and its parameter variables are created in memory, and when the function ends, the local variables and parameter variables are destroyed.
- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

Global Variables and Constants

- A global variable is any variable defined outside all the functions in a program.
- The scope of a global variable is the portion of the program from the variable definition to the end.
- This means that a global variable can be accessed by *all* functions that are defined after the global variable is defined.
- You should avoid using global variables because they make programs difficult to debug.
- Any global that you create should be *global constants*.
- Global variables (not constants) are automatically initialized to `0` (numeric) or `NULL` (character) when the variable is defined.

Static Local Variables

- Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.

- `static` local variables retain their contents between function calls or invocations.

- `static` local variables are defined and initialized only the first time the function is executed. `0` is the default initialization value.

- `static` local variables can also be used for running totals or counters