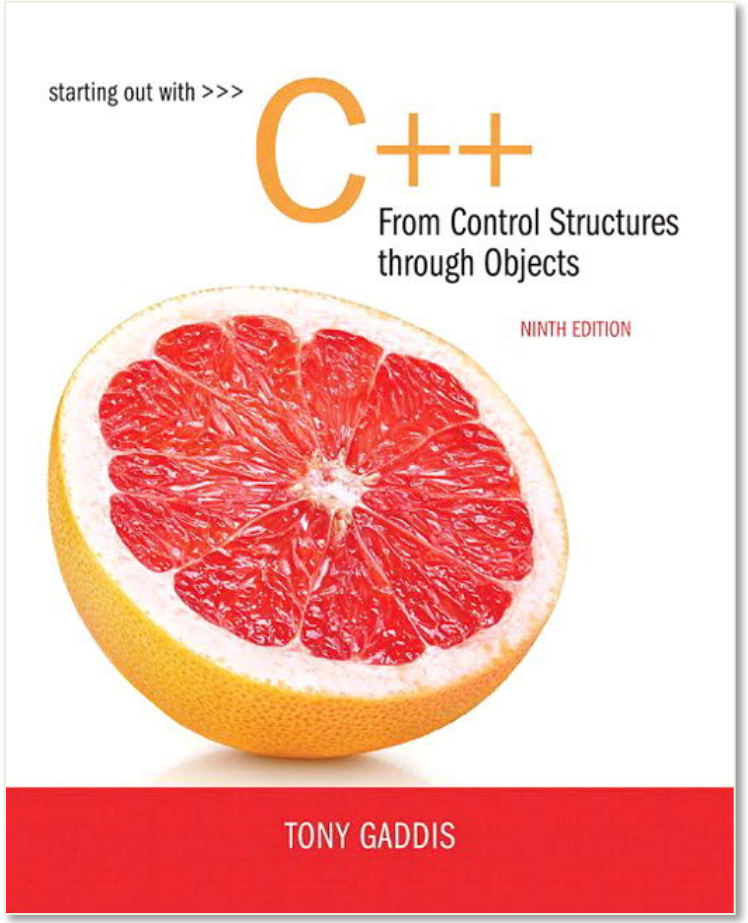
Chapter 6:

Functions

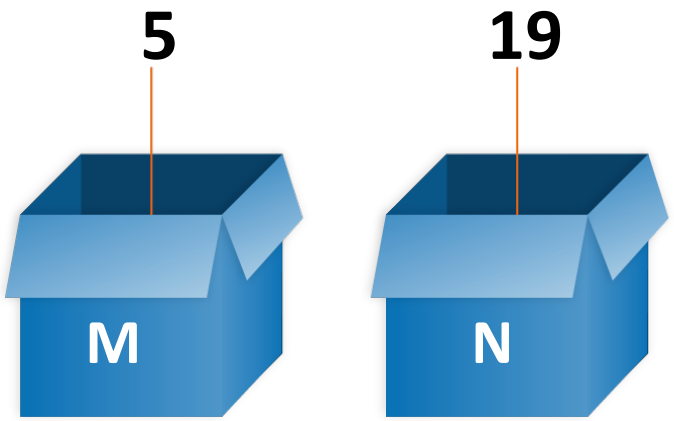
Important



Edited by:
Marwa Hassan

Exercise 1

Write the function doSwap that receives* two variables and swaps their stored values. Example: If the input to the procedure (function) is M = 5 and N = 19, the output will be M = 19 and N = 5



New Concept!

*receives means the function will receive the input in a way that may not involve a user. For example, the input can be received from another function.

doSwap function

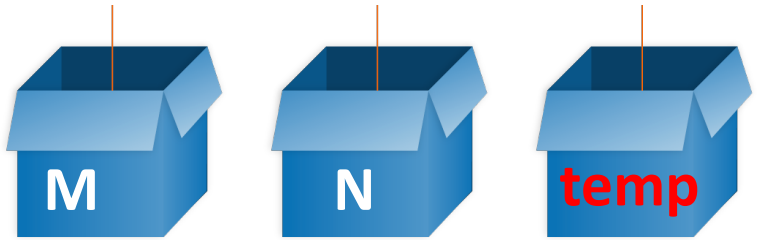
When given two variables, this function will swap their stored values **(physically).**

```
function doSwap(M,N)
1.   temp ← M
2.   M ← N
3.   N ←temp
```

Tracing the logic

	M	N	temp
Initial case	**5**	**19**	X
step 1	5	19	5
step 2	19	19	5
step 3	**19**	**5**	5

How many variables are being used by this function?



Exercise 2

- Given two numbers: **a** and **b**. Sort them in place in ascending order. Use doSwap function that will allow you to swap the values of two variables. The output of the algorithm is that "**a**" always holds the smaller value.

If you assign a piece of work to someone

```
if(a > b)
        doSwap(a,b)
end if
Print a, b
```

When you decide to do it yourself!

```
if(a > b)
        temp ← a
        a ← b
        b ←temp
end if
Print a, b
```

Exercise 3: Sorting 3 Numbers

Write the algorithm for sorting three numbers (a, b & c) in place. i.e. not just display the order, but change values stored in the containers *(variables)*.

```
if (a > b)
        temp ← a
        a ← b
        b ←temp
end if
if (b > c)
        temp ← b
        b ← c
        c ←temp
end if
if (a > b)
        temp ← a
        a ← b
        b ←temp
end if
```

Instead of performing the same steps over and over, use a **function** that performs the task and just **call** it as needed.

Algorithm 2 (sorting 3 numbers)

- **Assume we can use doSwap function that allows us to swap the values of two variables**

```
if (a > b)
        doSwap(a, b)
end if
if (b > c)
        doSwap(b, c)
end if
if (a > b)
        doSwap(a, b)
end if
```

Example:
Assume a=7, b=2, c=5
Table shows memory stored values

	a	b	c
Originally assigned values:	7	2	5
is (a>b)? Yes, swap	2	7	5
is (b>c)? Yes, swap	2	5	7
is (a>b)> No End	2	5	7

Final values stored

Sorting 3 numbers - Pseudocode

```
function main
    if (a > b)
        doSwap(a, b)
    end if
    if (b > c)
        doSwap(b, c)
    end if
    if (a > b)
        doSwap(a, b)
    end if
end main
```

```
function doSwap(x,y)
    temp ← x
    x ← y
    y ←temp
end doSwap
```

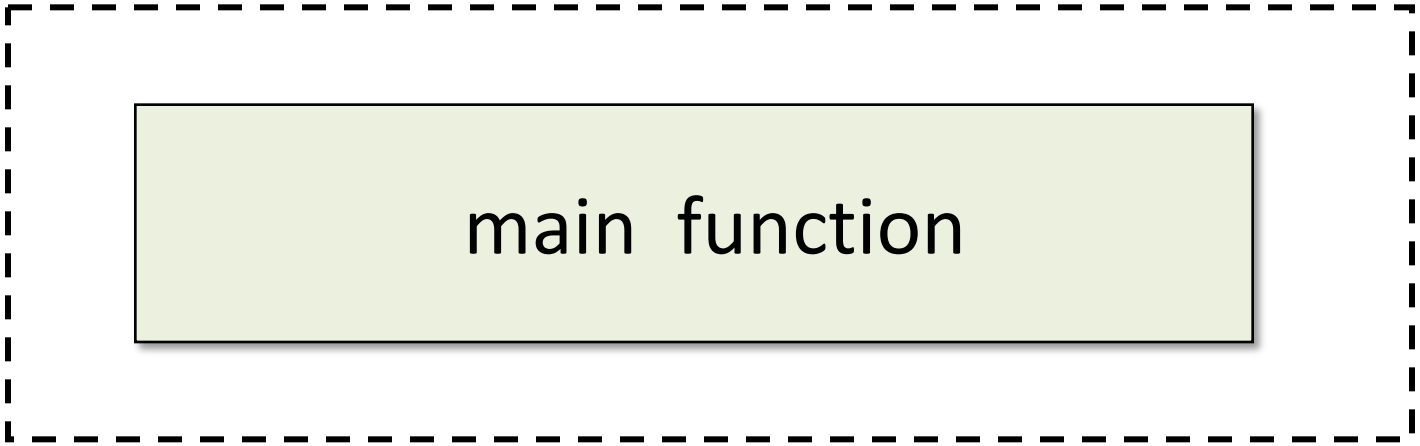
Steps are written one time and used multiple times

Writing functions in pseudocode

A C++ program is a collection of one or more functions

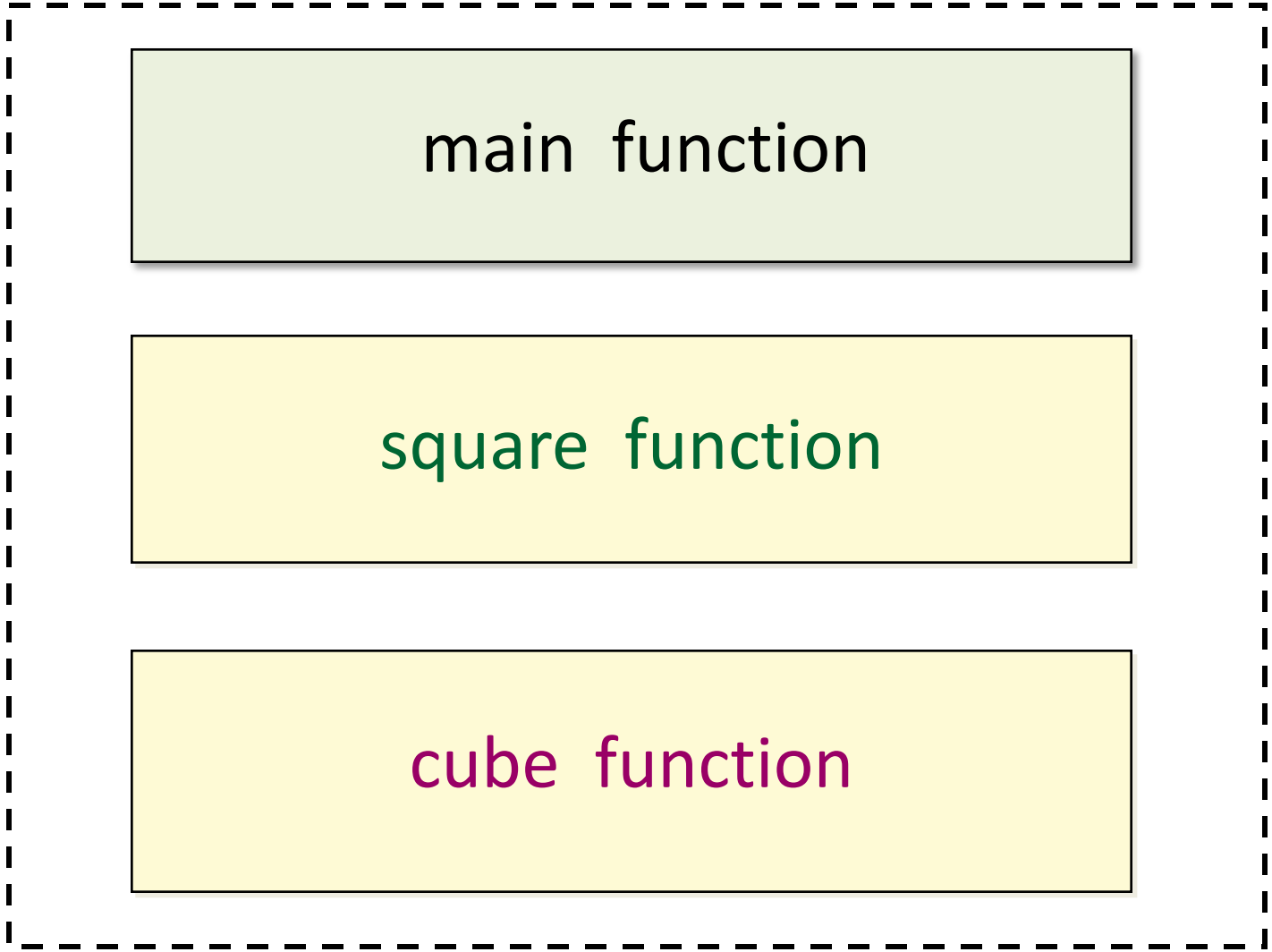
- there must be a function called main( )
- execution always begins with the first statement in function main( )
- any other functions in your program are subprograms and are not executed until they are called

Program With One Function



What we have been doing so far

Program With Several Functions



6.1 Modular Programming

- Modular programming: breaking a program up into smaller, manageable functions or modules
  - Instead of writing one long function, write small functions that each solve a specific part
- Function: a collection of statements to perform a specific task
- Motivation for modular programming:
  - Improves maintainability of programs
  - Simplifies the process of writing programs

Why write and use functions?

- Makes it easier to write, test, and debug
- Simplify modification process and maintenance process
- Avoids repetition of code in a program → reduce code size
  - A function is written once and callled (executed) any number of times (*doSwap*)
- Allow for *code reuse* in different programming projects
- Easier to break up the work for team development

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;
}
```

main function is the program "manager", controls <u>how</u> and <u>when</u> to call other functions

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

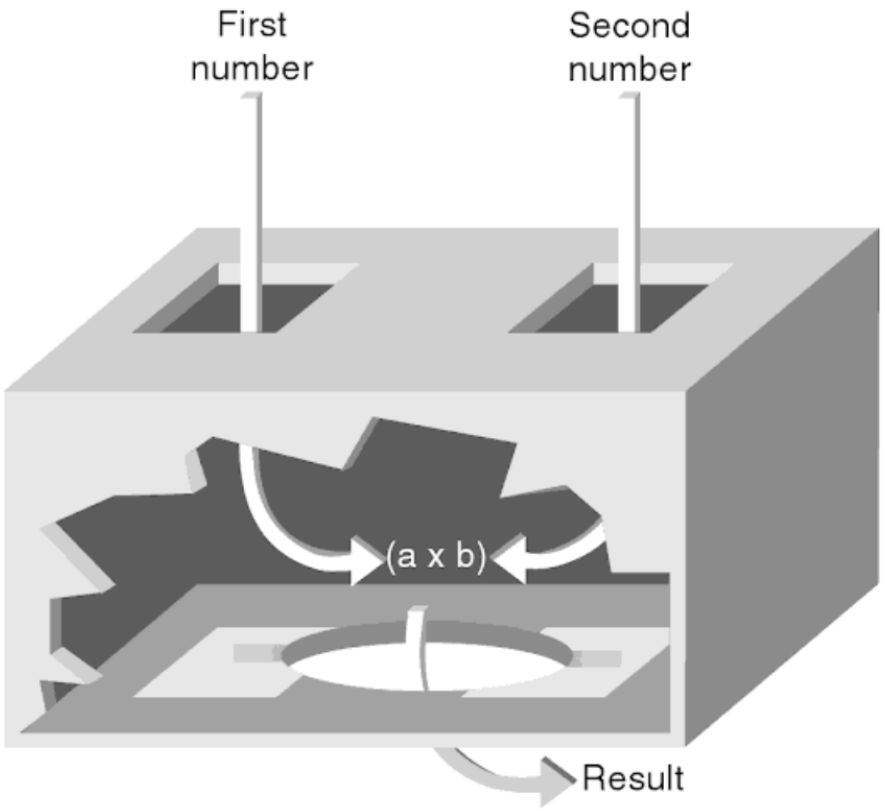
```
void function3()
{
    statement;
    statement;
    statement;
}
```

function 3

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

function 4

A Multiplying Function



<u>Call & Send</u> the **<u>variables</u>** to the function and **<u>receive</u>** the result (black box)

Figure: A First Book of C++ (Introduction to Programming) - ISBN-13: 978-1111531003

The Very First Exercise – Trace!

```
#include <iostream>
using namespace std;

void sayHello()
{
    cout << "Hello!" << endl;
}

int main()
{
    cout << "Hello from main." << endl;
    sayHello();
    cout << "Main waves back!" << endl;
    return 0;
}
```

Start at main

Output:

```
Hello from main.
Hello!
Main waves back!
```

Now take an analytical look

Defining a function – Main Parts

```
float sqrt (int num)
{
    //statements;
}
```

- **Name**
  - Should be a descriptive name (same rules as for variables)
- **Return Type**
  - The data type of the value returned to the part of the program that activated (called) the function
- **Parameter List**
  - A list of variables that hold the values being passed to the function
- **Body**
  - Stmts enclosed by *braces* that perform the function's operations (tasks).

Function Return Type

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

```
int main()
```

`main` function is declared to return an `int` value to the operating system when it finishes executing

```
return 0;
```

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

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

  - <u>The function has no return statement</u>

Function Parts - Example

Return Type

Function Name

Parameter List

Function header

```
int square(int x)
{
    int result = x * x;
    return result;
}
```

Body

Calling a Function

- A function is executed ONLY when it is ***called***
- Function `main` is called automatically when a program starts, but all other functions must be executed by *function call* statements
- To call a function, use the function name followed by `()` and `;`

```
printHeading();
//this function does not take any data
```

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

Calling a Function

- **`main`** is automatically called when the program starts
- **`main`** can call any number of functions
- Functions can call other functions

Calling a Function

Can we switch the order of functions in this code?

```
#include <iostream>
using namespace std;
void displayMessage()
{
    cout << "Hello from function dislplayMessage.\n";
}

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

    return 0;
}
```

Function Header

Function Call

Multiple Function Calls

What is the output?

```
void first()
{
  cout << "In first\n";
}

void second()
{
  cout << "In Second\n";
}
```

```
int main()
{
  cout<<"Start main\n";
  first();
  second();
  cout<<"Done main\n";
  return 0;
}
```

Where to define functions?

- Even though the program starts executing at `main`, in previous examples functions are defined first

The compiler must know the following about a function BEFORE it is called

1. **name**
2. **return type**
3. **number of parameters**
4. **data type of each parameter**

- **One way** is to place the function definition before all calls to that function

Another way

Function Prototypes

Function prototypes eliminate the need to place a function definition before all calls to the function

Prototype: `void printHeading();`

Header: `void printHeading()`

Locate the differences

```
void first();          //function prototype
void second();         //function prototype

int main()
{
   cout << "I am starting in function main.\n";
   first();            // Call function first
   second();           // Call function second
   cout << "Back in function main again.\n";
   return 0;
}
//*************************************
void first()
{
   cout << "I am now inside the function first.\n";
}
//*************************************
void second()
{
   cout << "I am now inside the function second.\n";
}
```

trace

Prototypes - Notes

- Place prototypes near top of program
- Program must include either prototype or full function definition BEFORE any call to the function, otherwise a compiler error occurs
- When using prototypes, function definitions (body) can be placed in any order in the source file.

  Traditionally, `main` is placed first. **DO this please!**

Your programs should look like this

- Format your .cpp files as follows:

```
preprocessor directives

function prototypes

int main()
{

}

function definitions
```

6.4 Sending Data into a Function

- When a function is <u>called</u>, the program may send values into the functions:

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

Function Call

- Values passed to function are called <u>arguments</u>
  - a & b are arguments

- Variables in a function that hold the values passed as arguments are called <u>parameters</u>

```
double pow(double b, double p)
```

Function Header

b and p are parameters with types double and double

A Function with a Parameter Variable

```
void displayValue(int num)
{
    cout << "The value is " << num << endl;
}
```

The integer variable `num` is a parameter.

It accepts any integer value passed to the function.

The whole picture / what is the output?

```
# include <iostream>
using namespace std;
// Function Prototype
void displayValue(int);

int main()
{
   cout << "I am passing 5 to displayValue.\n";
   displayValue(5);    // Call displayValue with argument 5
   cout << "Now I am back in main.\n";
   return 0;
}
// Passed argument is copied into parameter variable.
// Definition of function displayValue
void displayValue(int num)
{
   cout << "The value is " << num << endl;
}
```

```
I am passing 5 to displayValue.
The value is 5
Now I am back in main.
```

How does it work?

argument

```
displayValue(5);
```

parameter variable

```
void displayValue(int num)
{
  cout << "The value is " << num << endl;
}
```

The function call passes the value 5 as an argument to the function, the function will receive it in variable num.

Any argument listed inside the parentheses of a function call is **copied** into the function's parameter variable.