

# OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

## What is object oriented programming

Last year we have explored the structure of a simple program that starts execution at *main()* and enables you to declare local and global variables and constants and branch your execution logic into function modules that can take parameters and return values. All this is very similar to a procedural language like C, which has no object-orientation to it. In other words, you need to learn about managing data and connecting methods to it.

## The Concept of Classes and Objects

Imagine you are writing a program that models a human being, like yourself. This human being needs to have an identity: a name, a date of birth, a place of birth, and a gender. The human can perform certain functions, such as talk and introduce him- or herself, among others. Now, the former information is data about the human being, whereas the latter information comprises functions as illustrated by Figure below.

A broad representation of a human.



Human Being

Data

- Gender
- Date of birth
- Place of birth
- Name






Methods

- IntroduceSelf()
- ...

To model a human, what you now need is a construct that enables you to group within it the attributes that define a human (data) and the activities a human can perform (methods that are similar to functions) using the available attributes. This construct is the **class**.

## Organizing Code with Functions

Functions give you a way to compartmentalize and organize your program's execution logic. They enable you to split the contents of your application into logical blocks that are invoked sequentially. A function is hence a subprogram that optionally takes parameters and returns a value, and it needs to be invoked to perform its task. In this lesson you learn

-  Function prototypes and function definition
-  Passing parameters to functions and returning values from them
-  Reference arguments
-  Overloaded function
-  Inline function

## What Is a Function Prototype?

The function prototype basically tells what a function is called (the name of function), the list of parameters the function accepts (one or more parameters) and the return type of the function.

### Example:

Two Functions That Compute the Area and Circumference of a Circle Given Radius
<pre> 0: #include &lt;iostream&gt; 1: using namespace std; 2: 3: const double Pi = 3.14159; 4: 5: // Function Declarations (Prototypes) 6: double Area(double InputRadius); 7: double Circumference(double InputRadius); 8: 9: int main() 10: { 11:     cout &lt;&lt; "Enter radius: "; 12:     double Radius = 0; 13:     cin &gt;&gt; Radius; 14: 15: // Call function "Area" 16:     cout &lt;&lt; "Area is: " &lt;&lt; Area(Radius) &lt;&lt; endl; 17: 18: // Call function "Circumference" 19:     cout &lt;&lt; "Circumference is: " &lt;&lt; Circumference(Radius) &lt;&lt; endl; 20: 21:     return 0; 22: } 23: 24: // Function definitions (implementations) 25: double Area(double InputRadius) 26: { 27:     return Pi * InputRadius * InputRadius; 28: } 29: 30: double Circumference(double InputRadius) 31: { 32:     return 2 * Pi * InputRadius; 33: }</pre>
<b>Output:</b> Enter radius: 6.5 Area is: 132.732 Circumference is: 40.8407

Let's take a look at the example—Lines 6 and 7 in particular:

`double Area(double InputRadius);`

Return  
value  
type

Function  
Name

Function parameter(s) – optional:  
Parameter list comprised of type and  
optionally name, separated by comma in  
event of multiple parameters

The function prototype is called (the name, *Area*), the list of parameters the function accepts (one parameter, a double called *InputRadius*) and the return type of the function (a double).

Without the function prototype, on reaching Lines 16 and 19 in *main()* the compiler wouldn't know what the terms **Area** and **Circumference** are. The function prototypes tell the compiler that *Area* and *Circumference* are functions; they take one parameter of type double and return a value of type double. The compiler then recognizes these statements as valid and the job of linking the function call to its implementation and ensuring that the program execution actually triggers them is that of the linker.

**Note:** A function can have multiple parameters separated by commas, but it can have only one return type. When programming a function that does not need to return any value, specify the return type as **void**.

## What Is a Function Definition?

The implementation of a function—is what is called the definition. Analyze the definition of function *Area*:

```
25: double Area(double InputRadius)
26: {
27:     return Pi * InputRadius * InputRadius;
28: }
```

A function definition is always comprised of a statement block. A return statement is necessary unless the function is declared with return type void. In this case, *Area* needs to return a value because the return type of the function has not been declared as void.

The statement block contains statements within open and closed braces ({...}) that are executed when the function is called. *Area()* uses the input parameter *InputRadius* that contains the radius as an argument sent by the caller to compute the area of the circle.

## What Is a Function Call, and What Are Arguments?

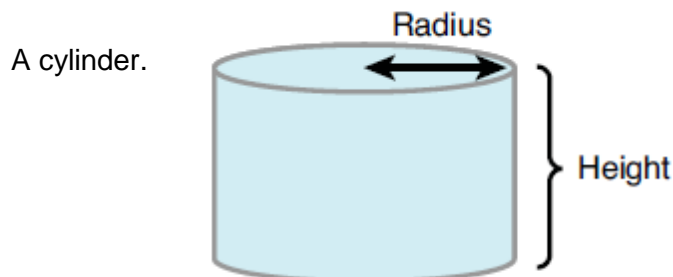
Invoking a function is also called making a function call. When a function is declared as one with parameters, the function call needs to send arguments that are values the function requests within its parameter list. Let's analyze a call to **Area** in the example:

```
16: cout << "Area is: " << Area(Radius) << endl;
```

Here, *Area(Radius)* is the function call, wherein *Radius* is the argument sent to the function *Area*. When invoked, execution jumps to *Area* that uses the radius sent to compute the area of the circle. When function *Area* is done, it returns a double. This return value double is then displayed on the screen via the **cout** statement.

## Programming a Function with Multiple Parameters

Assume you were writing a program that computes the area of a cylinder, as shown in the Figure below:



The formula you use would be the following:

$$\begin{aligned} \text{Area of Cylinder} &= \text{Area of top circle} + \text{Area of bottom circle} + \text{Area of Side} \\ &= \text{Pi} * \text{Radius}^2 + \text{Pi} * \text{Radius}^2 + 2 * \text{Pi} * \text{Radius} * \text{Height} \\ &= 2 * \text{Pi} * \text{Radius}^2 + 2 * \text{Pi} * \text{Radius} * \text{Height} \end{aligned}$$

Thus, you need to work with two variables, the radius and the height, in computing the area of the cylinder. In such cases, when writing a function that computes the surface area of the cylinder, you specify at least two parameters in the parameter list, within the function declaration. You do this by separating individual parameters by a comma as shown in the example below:

### Function That Accepts Two Parameters to Compute the Surface Area of aCylinder

```
0: #include <iostream>
1: using namespace std;
2:
3: const double Pi = 3.14159;
4:
5: // Declaration contains two parameters
6: double SurfaceArea(double Radius, double Height);
7:
8: int main()
9: {
10:     cout << "Enter the radius of the cylinder: ";
11:     double InRadius = 0;
12:     cin >> InRadius;
13:     cout << "Enter the height of the cylinder: ";
14:     double InHeight = 0;
15:     cin >> InHeight;
16:
17:     cout << "Surface Area: " << SurfaceArea(InRadius, InHeight) << endl;
18:
19:     return 0;
20: }
21:
22: double SurfaceArea(double Radius, double Height)
23: {
24:     double Area = 2 * Pi * Radius * Radius + 2 * Pi * Radius * Height;
25:     return Area;
26: }
```

#### Output:

```
Enter the radius of the cylinder: 3
Enter the height of the cylinder: 6.5
Surface Area: 179.071
```

## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

**Note:** Function parameters are like local variables. They are valid within the scope of the function only. So in the example, parameters Radius and Height to function *SurfaceArea* are valid and usable within function *SurfaceArea* only and not outside it.

### Programming Functions with No Parameters or No Return Values

If you delegate the task of saying “Hello World” to a function that does only that and nothing else, you could do it with one that doesn’t need any parameters (as it doesn’t need to do anything apart from say “Hello”), and possibly one that doesn’t return any value (because you don’t expect anything from such a function that would be useful elsewhere). Example below demonstrates one such function.

A Function with No Parameters and No Return Values
<pre>0: #include &lt;iostream&gt; 1: using namespace std; 2: 3: void SayHello(); 4: 5: int main() 6: { 7:     SayHello(); 8:     return 0; 9: } 10: 11: void SayHello() 12: { 13:     cout &lt;&lt; "Hello World" &lt;&lt; endl; 14: }</pre>
<b>Output:</b> Hello World

### Using Functions to Work with Different Forms of Data

Functions don’t restrict you to passing values one at a time; you can pass an array of values to a function. You can create two functions with the same name and return value but different parameters. You can program a function such that its parameters are not created and destroyed within the function call; instead, you use references that are valid even after the function has exited so as to allow you to manipulate more data or parameters in a function call. In this section you learn about passing arrays to functions, function overloading, and passing arguments by reference to functions.

### Overloading Functions

Functions with the same name and return type but with different parameters or set of parameters are said to be overloaded functions. Overloaded functions can be quite useful in applications where a function with a particular name that produces a certain type of output might need to be invoked with different sets of parameters. Say you need to be writing an application that computes the area of a circle and the area of a cylinder. The function that computes the area of a circle needs a parameter—the radius. The other function that computes the area of the cylinder needs the height of the cylinder in addition to the radius of the cylinder. Both functions need to return the data of the same type, containing the area. So, C++ enables you to define two

## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

overloaded functions, both called Area, both returning double, but one that takes only the radius as input and another that takes the height and the radius as input parameters as shown in the example below.

### Using an Overloaded Function to Calculate the Area of a Circle or a Cylinder

```
0: #include <iostream>
1: using namespace std;
2:
3: const double Pi = 3.14159;
4:
5: double Area(double Radius); // for circle
6: double Area(double Radius, double Height); // overloaded for cylinder
7:
8: int main()
9: {
10:     cout << "Enter z for Cylinder, c for Circle: ";
11:     char Choice = 'z';
12:     cin >> Choice;
13:
14:     cout << "Enter radius: ";
15:     double Radius = 0;
16:     cin >> Radius;
17:
18:     if (Choice == 'z')
19:     {
20:         cout << "Enter height: ";
21:         double Height = 0;
22:         cin >> Height;
23:
24:         // Invoke overloaded variant of Area for Cylinder
25:         cout << "Area of cylinder is: " << Area (Radius, Height) << endl;
26:     }
27:     else
28:         cout << "Area of cylinder is: " << Area (Radius) << endl;
29:
30:     return 0;
31: }
32:
33: // for circle
34: double Area(double Radius)
35: {
36:     return Pi * Radius * Radius;
37: }
38:
39: // overloaded for cylinder
40: double Area(double Radius, double Height)
41: {
42:     // reuse the area of circle
43:     return 2 * Area (Radius) + 2 * Pi * Radius * Height;
44: }
```

#### Output:

```
Enter z for Cylinder, c for Circle: z
Enter radius: 2
Enter height: 5
Area of cylinder is: 87.9646
```

#### Next run:

```
Enter z for Cylinder, c for Circle: c
Enter radius: 1
Area of cylinder is: 3.14159
```

# OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

## Passing an Array of Values to a Function

A function that displays an integer can be represented like this:

```
void DisplayInteger(int Number);
```

A function that can display an array of integers has a slightly different prototype:

```
void DisplayIntegers(int [ ] Numbers, int Length);
```

The first parameter tells the function that the data being input is an array, whereas the second parameter supplies the length of the array such that you can use the array without crossing its boundaries. See the following example.

Function That Takes an Array as a Parameter
<pre>0: #include &lt;iostream&gt; 1: using namespace std; 2: 3: void DisplayArray(int Numbers[], int Length) 4: { 5:     for (int Index = 0; Index &lt; Length; ++Index) 6:         cout &lt;&lt; Numbers[Index] &lt;&lt; " "; 7: 8:     cout &lt;&lt; endl; 9: } 10: 11: void DisplayArray(char Characters[], int Length) 12: { 13:     for (int Index = 0; Index &lt; Length; ++Index) 14:         cout &lt;&lt; Characters[Index] &lt;&lt; " "; 15: 16:     cout &lt;&lt; endl; 17: } 18: 19: int main() 20: { 21:     int MyNumbers[4] = {24, 58, -1, 245}; 22:     DisplayArray(MyNumbers, 4); 23: 24:     char MyStatement[7] = {'H', 'e', 'l', 'l', 'o', '!', '\0'}; 25:     DisplayArray(MyStatement, 7); 26: 27:     return 0; 28: }</pre>
<b>Output:</b>  24 58 -1 245 H e l l o !

## Passing Arguments by Reference

Take another look at the function in the following code that computed the area of a circle given the radius:

```
24: // Function definitions (implementations)
25: double Area(double InputRadius)
26: {
27:     return Pi * InputRadius * InputRadius;
28: }
```

Here, the parameter *InputRadius* contains a value that is copied in to it when the function is invoked in *main()*:

```
15: // Call function "Area"
16: cout << "Area is: " << Area(Radius) << endl;
```

This means that the variable *Radius* in *main* is unaffected by the function call, as *Area()* works on a copy of the value *Radius* contains, held in *InputRadius*. There are cases where

## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

you might need a function to work on a variable that modifies a value that is available outside the function, too, say in the calling function. This is when you declare a parameter that takes an argument *by reference*. A form of the function *Area()* that computes and returns the area as a parameter by reference looks like this:

```
// output parameter Result by reference
void Area(double Radius, double& Result)
{
    Result = Pi * Radius * Radius;
}
```

Note how *Area()* in this form takes two parameters. Don't miss the ampersand (&) next to the second parameter Result. This sign indicates to the compiler that the second argument should NOT be copied to the function; instead, it is a reference to the variable being passed. The return type has been changed to void as the function no longer supplies the area computed as a return value, rather as an output parameter by reference. Returning values by references is demonstrated in the following example, which computes the area of a circle.

Fetching the Area of a Circle as a Reference Parameter and Not as a Return Value
<pre>0: #include &lt;iostream&gt; 1: using namespace std; 2: 3: const double Pi = 3.1416; 4: 5: // output parameter Result by reference 6: void Area(double Radius, double&amp; Result) 7: { 8:     Result = Pi * Radius * Radius; 9: } 10: 11: int main() 12: { 13:     cout &lt;&lt; "Enter radius: "; 14:     double Radius = 0; 15:     cin &gt;&gt; Radius; 16: 17:     double AreaFetched = 0; 18:     Area(Radius, AreaFetched); 19: 20:     cout &lt;&lt; "The area is: " &lt;&lt; AreaFetched &lt;&lt; endl; 21:     return 0; 22: }</pre>
<b>Output:</b> Enter radius: 2 The area is: 12.5664

## Inline Functions

A regular function call is translated into a CALL instruction, which results in stack operations and microprocessor execution shift to the function and so on. This might sound like a lot of stuff happening under the hood, but it happens quite quickly—for most of the cases. However, what if your function is a very simple one like the following?

```
double GetPi()
{
    return 3.14159;
}
```



## OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

The overhead of performing an actual function call on this might be quite high for the amount of time spent actually executing *GetPi()*. This is why C++ compilers enable the programmer to declare such functions as inline. Keyword inline is the programmers' request that these functions be expanded inline where called.

```
inline double GetPi()
{
    return 3.14159;
}
```

Similarly, functions that just double a number and perform such simple operations are good candidates for being inlined, too. Example below demonstrates one such case.

### Using an Inline Function That Doubles an Integer

```
0: #include <iostream>
1: using namespace std;
2:
3: // define an inline function that doubles
4: inline long DoubleNum (int InputNum)
5: {
6:     return InputNum * 2;
7: }
8:
9: int main()
10: {
11:     cout << "Enter an integer: ";
12:     int InputNum = 0;
13:     cin >> InputNum;
14:
15:     // Call inline function
16:     cout << "Double is: " << DoubleNum(InputNum) << endl;
17:
18:     return 0;
19: }
```

#### Output::

```
Enter an integer: 35
Double is: 70
```