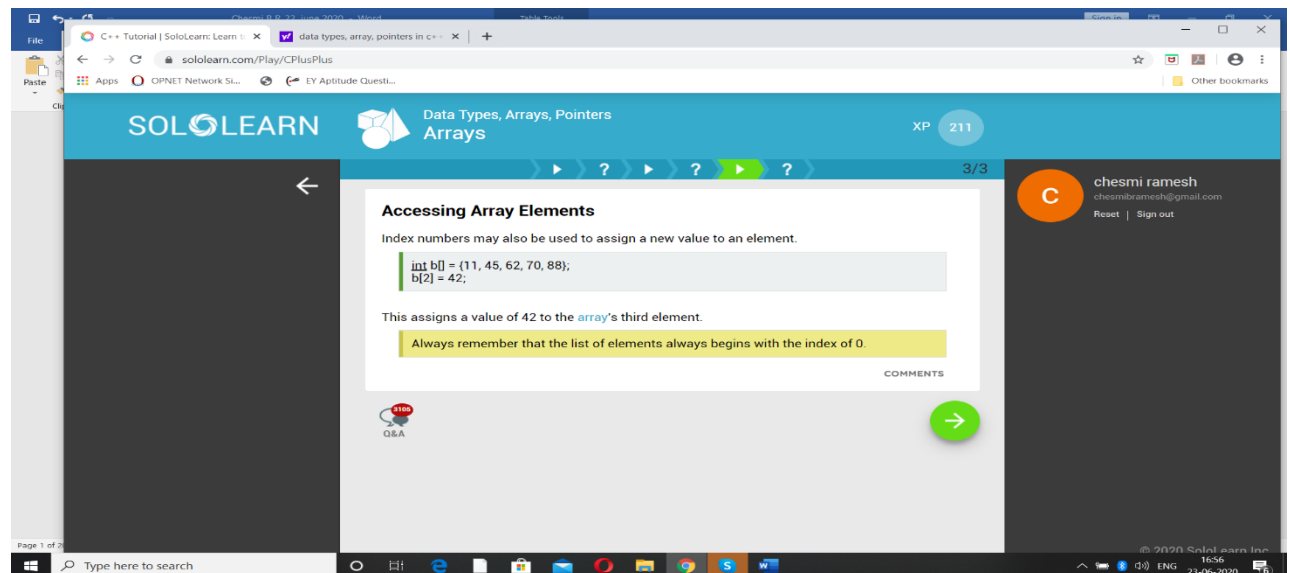


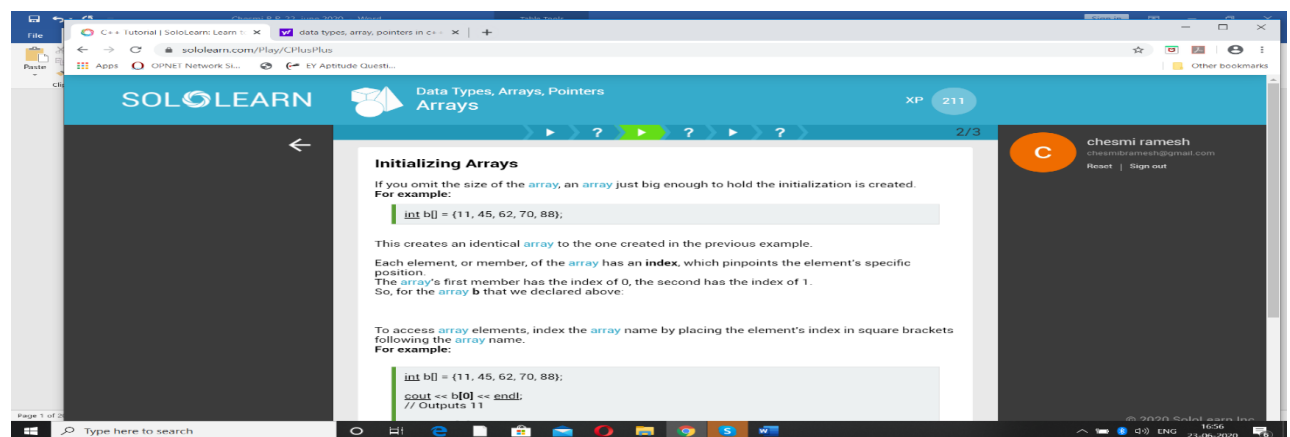
|                    |  |                     |                                 |
|--------------------|--|---------------------|---------------------------------|
| Date:              | 24/06/2020   | Name:               | Chesmi B R                      |
| Course:            | C++  | USN:                | 4AL16EC100                      |
| Topic:             | Module 5: classes and objects<br>Module 6: More on classes | Semester & Section: | 8 <sup>TH</sup> SEM & A Section |
| Github Repository: | chesmibr   |                     |                                 |

## DAILY ASSESSMENT

### FORENOON SESSION DETAILS



The screenshot shows the SoloLearn C++ tutorial interface. The title is "Data Types, Arrays, Pointers Arrays". The user's profile "chesmi ramesh" is visible on the right. The current lesson is "Accessing Array Elements". The text explains that index numbers can be used to assign new values to array elements. A code example shows: `int b[] = {11, 45, 62, 70, 88};` followed by `b[2] = 42;`. A yellow highlight states: "This assigns a value of 42 to the array's third element." Another yellow highlight says: "Always remember that the list of elements always begins with the index of 0." Navigation buttons for back, forward, and search are visible.



The screenshot shows the SoloLearn C++ tutorial interface for the lesson "Initializing Arrays". The user's profile "chesmi ramesh" is visible on the right. The text explains that if the size of an array is omitted, it will be initialized with a size just big enough to hold the initialization. A code example shows: `int b[] = {11, 45, 62, 70, 88};`. It then states: "This creates an identical array to the one created in the previous example. Each element, or member, of the array has an index, which pinpoints the element's specific position. The array's first member has the index of 0, the second has the index of 1. So, for the array b that we declared above." It then explains how to access array elements by indexing the array name with square brackets. A code example shows: `int b[] = {11, 45, 62, 70, 88};` followed by `cout << b[0] << endl;` and `// Outputs 11`. Navigation buttons for back, forward, and search are visible.

## **Report:**

### **C++ Classes and Objects**

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

### **C++ Class Definitions**

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows –

```
class Box {  
    public:  
        double length; // Length of a box  
        double breadth; // Breadth of a box  
        double height; // Height of a box  
};
```

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within

the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

### Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box –

```
Box Box1;      // Declare Box1 of type Box
```

```
Box Box2;      // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

### Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear

```
#include <iostream>
using namespace std;
class Box {
    public:
        double length; // Length of a box
        double breadth; // Breadth of a box
        double height; // Height of a box
};
int main() {
    Box Box1;      // Declare Box1 of type Box
    Box Box2;      // Declare Box2 of type Box
    double volume = 0.0; // Store the volume of a box here
```

```
// box 1 specification
Box1.height = 5.0;
Box1.length = 6.0;
Box1.breadth = 7.0;
// box 2 specification
Box2.height = 10.0;
Box2.length = 12.0;
Box2.breadth = 13.0;
// volume of box 1
volume = Box1.height * Box1.length * Box1.breadth;
cout << "Volume of Box1 : " << volume << endl;
// volume of box 2
volume = Box2.height * Box2.length * Box2.breadth;
cout << "Volume of Box2 : " << volume << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Volume of Box1 : 210

Volume of Box2 : 1560

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

## **Classes and Objects in Detail**

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below –

| Sr.No | Concept & Description  |  |
|-------|--|--|
| 1     | <p><u>Class Member Functions</u></p> <p>A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.</p>   |  |
| 2     | <p><u>Class Access Modifiers</u></p> <p>A class member can be defined as public, private or protected. By default member would be assumed as private.</p>  |  |
| 3     | <p><u>Constructor &amp; Destructor</u></p> <p>A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.</p> |  |
| 4     | <p><u>Copy Constructor</u></p> <p>The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.</p>  |  |
| 5     | <p><u>Friend Functions</u></p> <p>A <b>friend</b> function is permitted full access to private and protected members of a class.</p>   |  |
| 6     | <p><u>Inline Functions</u></p> <p>With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.</p>   |  |

|   |   |  |
|---|---|--|
| 7 | <u>this Pointer</u><br><br>Every object has a special pointer <b>this</b> which points to the object itself.  |  |
| 8 | <u>Pointer to C++ Classes</u><br><br>A pointer to a class is done exactly the same way a pointer to a structure is. In fact, a class is really just a structure with functions in it. |  |
| 9 | <u>Static Members of a Class</u><br><br>Both data members and function members of a class can be declared as static.  |  |

### **Data Abstraction in C++**

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and

to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this –

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello C++" << endl;
    return 0;
}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of 'cout' is free to change.

## Benefits of Data Abstraction

Data abstraction provides two important advantages –

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

### Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example –

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

class Adder {
public:
    // constructor
    Adder(int i = 0) {
        total = i;
    }

    // interface to outside world
    void addNum(int number) {
        total += number;
    }

    // interface to outside world
    int getTotal() {
        return total;
    };
};
```



```
private:
    // hidden data from outside world
    int total;
};
int main() {
    Adder a;
    a.addNum(10);
    a.addNum(20);
    a.addNum(30);
    cout << "Total " << a.getTotal() << endl;
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Total 60

Above class adds numbers together, and returns the sum. The public members - **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

### Data Encapsulation in C++

All C++ programs are composed of the following two fundamental elements –

- **Program statements (code)** – This is the part of a program that performs actions and they are called functions.
- **Program data** – The data is the information of the program which gets affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside

interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

**Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private**, **protected** and **public** members. By default, all items defined in a class are private. For example –

```
class Box {  
    public:  
        double getVolume(void) {  
            return length * breadth * height;  
        }  
  
    private:  
        double length;    // Length of a box  
        double breadth;   // Breadth of a box  
        double height;    // Height of a box  
};
```

The variables `length`, `breadth`, and `height` are **private**. This means that they can be accessed only by other members of the `Box` class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

### Data Encapsulation Example

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example

```
#include <iostream>
using namespace std;
class Adder {
public:
    // constructor
    Adder(int i = 0) {
        total = i;
    }
    // interface to outside world
    void addNum(int number) {
        total += number;
    }
    // interface to outside world
    int getTotal() {
        return total;
    };
private:
    // hidden data from outside world
    int total;
};
```

```
int main() {  
    Adder a;  
    a.addNum(10);  
    a.addNum(20);  
    a.addNum(30);  
    cout << "Total " << a.getTotal() << endl;  
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

Total 60

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

### **C++ Overloading (Operator and Function)**

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

## Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types

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

class printData {
public:
    void print(int i) {
        cout << "Printing int: " << i << endl;
    }
    void print(double f) {
        cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
        cout << "Printing character: " << c << endl;
    }
};

int main(void) {
    printData pd;

    // Call print to print integer
```

```
pd.print(5);

// Call print to print float
pd.print(500.263);

// Call print to print character
pd.print("Hello C++");

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

## Operators Overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows –

```
Box operator+(const Box&, const Box&);
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below –

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

class Box {
public:
    double getVolume(void) {
        return length * breadth * height;
    }
    void setLength( double len ) {
        length = len;
    }
    void setBreadth( double bre ) {
        breadth = bre;
    }
    void setHeight( double hei ) {
        height = hei;
    }

    // Overload + operator to add two Box objects.
    Box operator+(const Box& b) {
        Box box;
        box.length = this->length + b.length;
        box.breadth = this->breadth + b.breadth;
```

```
        box.height = this->height + b.height;
        return box;
    }

private:
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
};

// Main function for the program
int main() {
    Box Box1;        // Declare Box1 of type Box
    Box Box2;        // Declare Box2 of type Box
    Box Box3;        // Declare Box3 of type Box
    double volume = 0.0; // Store the volume of a box here

    // box 1 specification
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);

    // box 2 specification
    Box2.setLength(12.0);
    Box2.setBreadth(13.0);
    Box2.setHeight(10.0);

    // volume of box 1
```



```
volume = Box1.getVolume();
cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2
volume = Box2.getVolume();
cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:
Box3 = Box1 + Box2;

// volume of box 3
volume = Box3.getVolume();
cout << "Volume of Box3 : " << volume <<endl;

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

