

The logo for Aryabhata Knowledge University consists of several overlapping circles in blue, black, and yellow.

Aryabhata Knowledge University (AKU)

Information Technology (IT)

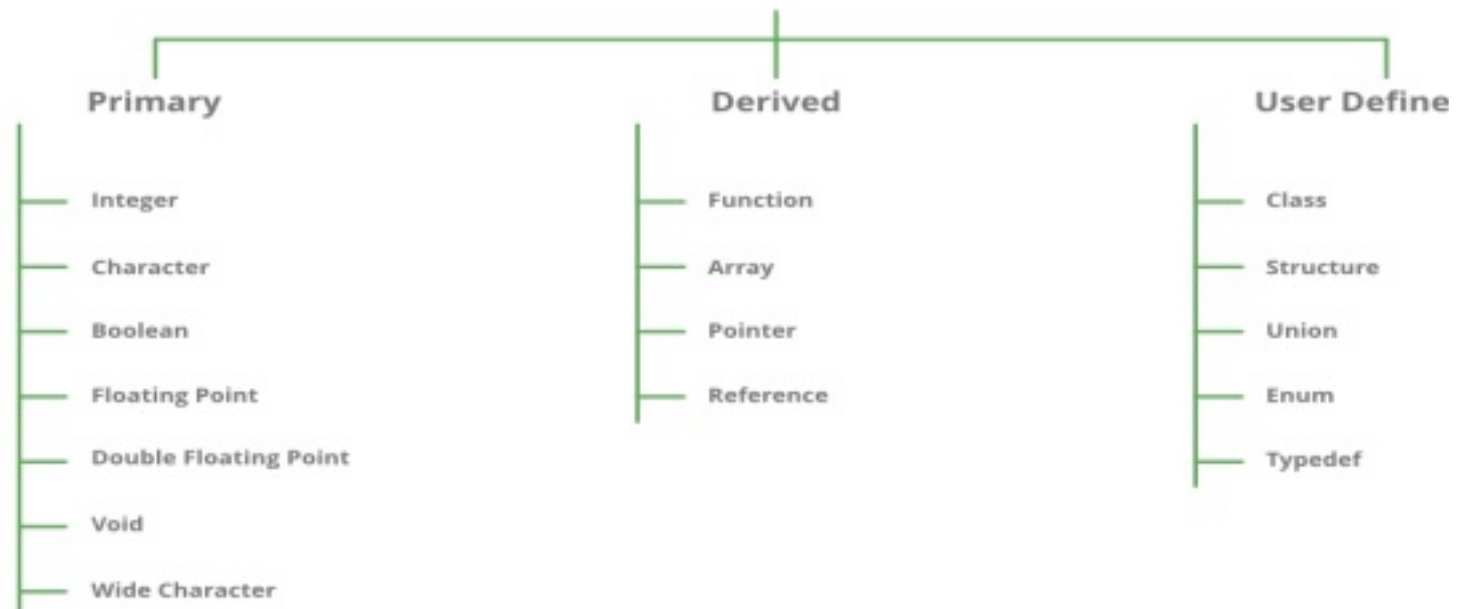
Object oriented programming

Solved Exam Paper 2019

Question. What are the different data types in C++? Explain that C++ is an object-oriented language?

Answer: All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the datatype with which it is declared. Every data type requires a different amount of memory.

DataTypes in C / C++



Data types in C++ is mainly divided into three types:

- Primitive Data Types:** These data types are built-in or predefined data types and can be used directly by the user to declare variables. Example: int, char, float, bool etc. Primitive data types available in C++ are:
 - Integer
 - Character
 - Boolean
 - Floating Point
 - Double Floating Point
 - Valueless or Void
 - Wide Character
- Derived Data Types:** The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:
 - Function
 - Array
 - Pointer
 - Reference
- Abstract or User-Defined Data Types:** These data types are defined by user itself. Like, defining a class in C++ or a structure.

C++ provides the following user-defined datatypes:

- Class
- Structure
- Union
- Enumeration
- Typedef defined Datatype
- **Integer**: Keyword used for integer data types is **int**. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.
- **Character**: Character data type is used for storing characters. Keyword used for character data type is **char**. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.
- **Boolean**: Boolean data type is used for storing Boolean or logical values. A Boolean variable can store either *true* or *false*. Keyword used for Boolean data type is **bool**.
- **Floating Point**: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is **float**. Float variables typically requires 4 byte of memory space.
- **Double Floating Point**: Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is **double**. Double variables typically requires 8 byte of memory space.
- **Void**: Void means without any value. Void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.
- **Wide Character**: Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by **wchar_t**. It is generally 2 or 4 bytes long.

Here are the reasons C++ is called partial or semi Object Oriented Language:

1) Main function is outside the class: C++ supports object-oriented programming, but OO is not intrinsic to the language. You can write a valid, well-coded, excellently-styled C++ program without using an object even once.

In C++, main function is mandatory, which executes first but it resides outside the class and from there we create objects. So, here creation of class becomes optional and we can write code without using class.

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
int main()
```

```
{  
  
    cout << "Hello World";  
  
    return 0;  
}
```

While in JAVA, main function is executed first and it reside in the class which is mandatory. So, we can't do anything without making Class. For doing the same thing as above, we need to make a class as:

```
class hello  
{  
  
    public static void main (String args[])  
  
    {  
  
        System.out.println("Hello World");  
  
    }  
  
}
```

2) Concept of Global variable: In C++, we can declare a variable globally, which can be accessible from anywhere and hence, it does not provides complete privacy to the data as no one can be restricted

to access and modify those data and so, it provides encapsulation partially whereas In JAVA, we can declare variable inside class only and we can provide access specifier to it.

```
#include <iostream>
```

```
Using namespace std;
```

```
// Global variable declaration:
```

```
int g = 50;
```

```
int main ()
```

```
{
```

```
    // global variable g
```

```
    cout << g;
```

```
    // Local variable g
```

```
    g = 20;
```

```
    cout << g;
```

```
    return 0;
```

```
}
```

Output:

So, in JAVA, basically every data is asked explicitly by user if it should be accessible or not.

3) Availability of Friend function: Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class.

Therefore, again the Object oriented features can be violated by C++.

Question. What do you understand by object oriented programming? What are the advantages of programming using object-oriented approach?

Answer: OOP (object-oriented programming) is a programming paradigm that is completely based on 'objects'. A general explanation of 'object' for better understanding - Mr. A is going to build a POT with the use of BLOCKS. Blocks are a kind of measurement units like height, radius, and shape by default. These properties are there by default, which means if you use a block it has some dimensions associated with it. Now there are some other crucial properties that are not yet assigned like - color, material, and price. So, Objects are nothing but POTS. We build an object by assigning values to the properties when we need them. BLOCK is nothing but the templates of the object. There we write how the object should appear (means look like) and how the actions will take place. In Java, we call it a class.

Advantages of OOP

Moving to the advantages of OOP, we would like to say that there are many as this is one of the core development approaches which is

widely accepted.

1. Re-usability

It means reusing some facilities rather than building it again and again. This is done with the use of a class. We can use it 'n' number of times as per our need.

2. Data Redundancy

This is a condition created at the place of data storage (you can say Databases) where the same piece of data is held in two separate places. So the data redundancy is one of the greatest advantages of OOP. If a user wants a similar functionality in multiple classes he/she can go ahead by writing common class definitions for the similar functionalities and inherit them.

3. Code Maintenance

This feature is more of a necessity for any programming languages, it helps users from doing re-work in many ways. It is always easy and time-saving to maintain and modify the existing codes with incorporating new changes into it.

4. Security

With the use of data hiding and abstraction mechanism, we are filtering out limited data to exposure which means we are maintaining security and providing necessary data to view.

5. Design Benefits

If you are practicing on OOPs the design benefit a user will get is in terms of designing and fixing things easily and eliminating the risks (if any). Here the Object Oriented Programs forces the designers to have a longer and extensive design phase, which results in better designs and fewer flaws. After a time when the program has reached some critical limits, it is easier to program all the non-OOP's one

separately.

6. Better productivity

With the above-mentioned facts of using the application definitely enhances its users overall productivity. This leads to more work done, finish a better program, having more inbuilt features and easier to read, write and maintain. An OOP programmer can stitch new software objects to make completely new programs. A good number of libraries with useful functions in abundance make it possible.

7. Easy troubleshooting

let's witness some common issues or problems any developers face in their work.

Is this the problem in the widget file?

Is the problem is in the Whale lumper?

Will I have to trudge through that 'sewage.c' file?

Commenting on all these issues related to code.

So, many a time it happens that something has gone wrong which later becomes so brainstorming for the developers to look where the error is. Relax! Working with OOP language you will know where to look for. This is the advantage of using encapsulation in OOP; all the objects are self-constrained. With this modality behavior, the IT teams get a lot of work benefits as they are now capable to work on multiple projects simultaneously with an advantage that there is no possibility of code duplicity.

8. Polymorphism Flexibility

You behave in a different way if the place or surrounding gets change. A person will behave like a customer if he is in a market, the same person will behave like a student if he is in a school and as a

son/daughter if put in a house. Here we can see that the same person showing different behavior every time the surroundings are changed. This means polymorphism is flexibility and helps developers in a number of ways.

Its simplicity Extensibility

9. Problems solving

Decomposing a complex problem into smaller chunks or discrete components is a good practice. OOP is specialized in this behavior, as it breaks down your software code into bite-sized - one object at a time. In doing this the broken components can be reused in solutions to different other problems (both less and more complex) or either they can be replaced by the future modules which relate to the same interface with implementations details.

A general relatable real-time scenario - at a high level a car can be decomposed into wheels, engine, a chassis soon and each of those components can be further broken down into even smaller atomic components like screws and bolts. The engine's design doesn't need to know anything about the size of the tires in order to deliver a certain amount of power (as output) has little to do with each other.

Question. What is a Class? Also write an example (syntax) to define a class in C++. Differentiate between a class and an object?

Answer: Class: A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share

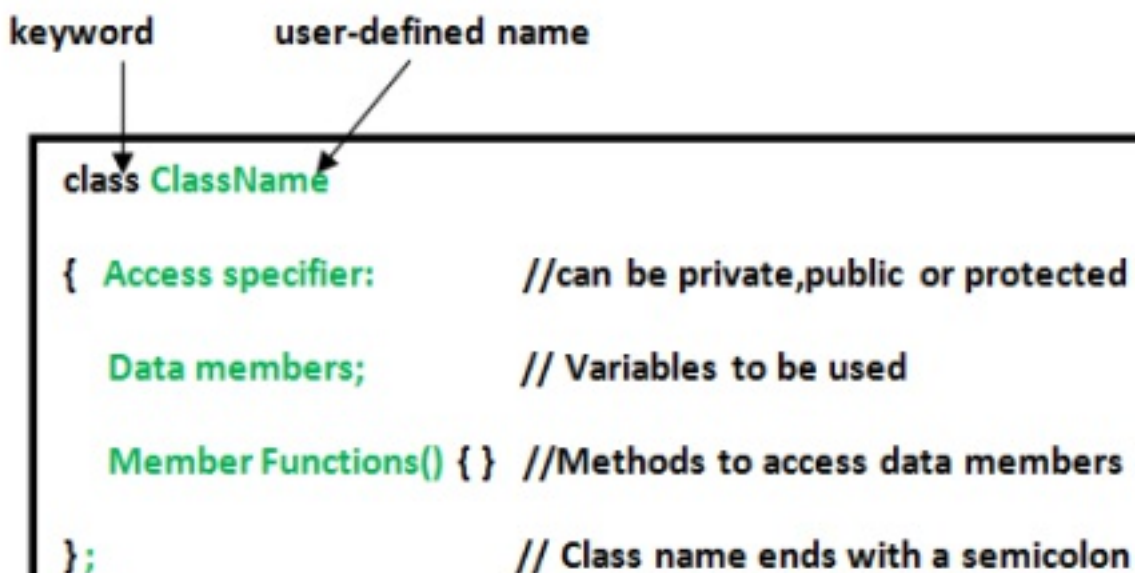
some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit, mileage* etc. and member functions can be *apply brakes, increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



```
keyword      user-defined name
↓            ↙
class ClassName
{ Access specifier:      //can be private,public or protected
  Data members;        // Variables to be used
  Member Functions() { } //Methods to access data members
};                // Class name ends with a semicolon
```

Comparison Chart

BASIS FOR COMPARISON	OBJECT	CLASS
Definition	An instance of a class is known as Object.	A template or blueprint which objects are created known as Class.
Type of entity	Physical	Logical
Creation	Object is invoked by new keyword.	Class is declared by using keyword.
Memory allocation	Creation of object consumes memory.	The formation of a doesn't allocate memory.

Question. What is Inheritance? What are the different types of inheritance in C++?

Answer: The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Types of Inheritance in C++

- 1) Single inheritance
- 2) Multilevel inheritance
- 3) Multiple inheritance
- 4) Hierarchical inheritance
- 5) Hybrid inheritance

Single inheritance

In Single inheritance one class inherits one class exactly.

For example: Lets say we have class A and B

B inherits A

Example of Single inheritance:

```
#include <iostream>

using namespace std;

class A {

public:

    A(){

        cout<<"Constructor of A class"<<endl;

    }

}
```

```
};  
  
class B: public A {  
  
public:  
  
    B(){  
  
        cout<<"Constructor of B class";  
  
    }  
  
};  
  
int main() {  
  
    //Creating object of class B  
  
    B obj;  
  
    return 0;  
  
}
```

Output:

Constructor of A class

Constructor of B class

2)Multilevel Inheritance

In this type of inheritance one class inherits another child class.

C inherits B and B inherits A

Example of Multilevel inheritance:

```
#include <iostream>
```

```
using namespace std;
```

```
class A {
```

```
public:
```

```
    A(){
```

```
        cout<<"Constructor of A class"<<endl;
```

```
    }
```

```
};
```

```
class B: public A {
```

```
public:
```

```
    B(){
```

```
        cout<<"Constructor of B class"<<endl;
```

```
    }
```

```
};
```

```
class C: public B {
```

```
public:
```

```
    C(){
```

```
        cout<<"Constructor of C class"<<endl;
```

```
    }
```

```
};
```

```
int main() {  
  
    //Creating object of class C  
  
    C obj;  
  
    return 0;  
  
}
```

Output:

Constructor of A class

Constructor of B class

Constructor of C class

Multiple Inheritance

In multiple inheritance, a class can inherit more than one class. This means that in this type of inheritance a single child class can have multiple parent classes.

For example:

C inherits A and B both

Example of Multiple Inheritance:

```
#include <iostream>  
  
using namespace std;  
  
class A {
```

```
public:

A(){

    cout<<"Constructor of A class"<<endl;

}

};

class B {

public:

B(){

    cout<<"Constructor of B class"<<endl;

}

};

class C: public A, public B {

public:

C(){

    cout<<"Constructor of C class"<<endl;

}

};

int main() {

    //Creating object of class C

    C obj;

    return 0;

}
```



```
}
```

Constructor of A class

Constructor of B class

Constructor of C class

4) Hierarchical Inheritance

In this type of inheritance, one parent class has more than one child class. For example:

Class B and C inherits class A

Example of Hierarchical inheritance:

```
#include <iostream>
```

```
using namespace std;
```

```
class A {
```

```
public:
```

```
    A(){
```

```
        cout<<"Constructor of A class"<<endl;
```

```
    }
```

```
};
```

```
class B: public A {
```

```
public:
```

```
B(){
```

```
    cout<<"Constructor of B class"<<endl;
```

```
}
```

```
};
```

```
class C: public A{
```

```
public:
```

```
    C(){
```

```
        cout<<"Constructor of C class"<<endl;
```

```
    }
```

```
};
```

```
int main() {
```

```
    //Creating object of class C
```

```
    C obj;
```

```
    return 0;
```

```
}
```

Output:

Constructor of A class

Constructor of C class

5) Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of

inheritance. For example, a child and parent class relationship that follows multiple and hierarchical inheritance both can be called hybrid inheritance.

Question. What do you mean by polymorphism? What are the Static and dynamic polymorphism techniques?

Answer: The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possess different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.



In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

Static and dynamic polymorphism techniques:

The static polymorphism is often referred to as compile-time or early binding polymorphism whereas, the dynamic polymorphism is referred to as run-time or late binding polymorphism. The static polymorphism is achieved using method overloading and operator overloading, whereas the dynamic polymorphism is achieved using method overriding. Also, in the subsequent section of this post, you will find the implementation of operator overloading and the list of operators that can and can't be overloaded.

Static Polymorphism (Method Overloading)

This type of polymorphism is also referred to as compile-time or early binding polymorphism because the decision about which method will

be called is made at the compile time. Now let's see one example of method overloading where we will give two methods the same name but different signatures (number and type of parameters) to achieve static polymorphism.

Filename: Program.cs

```
using System;

namespace Studytonight
{
    public class Interest
    {
        // interest for 1 year of tenure
        public double TrueBank(double amount, double rate)
        {
            return amount + (amount * rate);
        }

        public double TrueBank(double amount, double rate, string
holdertype)
        {
            return amount + (amount * rate) + 2000;
        }
    }
}
```

```
}  
  
public class Program  
  
{  
  
    public static void Main(string[] args)  
  
    {  
  
        Interest i = new Interest();  
  
        double finalamount = i.TrueBank(5000.00, 0.1);  
  
        Console.WriteLine("Normal interest for a holder " +  
finalamount);  
  
        finalamount = i.TrueBank(5000.00, 0.1, "prime");  
  
        Console.WriteLine("Prime interest for a holder " +  
finalamount);  
  
    }  
  
}  
  
}
```

Output:

Normal interest for a holder 5500

Prime interest for a holder 7500

In the code example above, we have created a class named Interest, and in that, we have given the same method name that is TrueBank to two different methods having different signature (number of parameters) and method definition. We have then created the object of the Interest class and provided the parameter list. If you closely observe the above parameter list, you will come to know where the difference lies. In the first one, we have given two parameters, and in the second call, three parameters are provided. At compile-time, the respective method automatically gets called with the help of signatures of the methods. And hence, we have achieved static polymorphism.

2. Dynamic Polymorphism (Method Overriding)

This type of polymorphism also referred to as run-time or late binding polymorphism because of the decision about which method is to be called is made at run time. In dynamic polymorphism, we override the base class method in derived class using inheritance, and this can be achieved using override and virtual keywords. Now we will see the example of method overriding where we will give the same method name and signature (same number of parameters and type but with different definitions) too in parent and child class.

Filename: Program.cs

```
using System;
```

```
namespace Studytonight
```

```
{
```

```
    public class Interest
```

```
    {
```

```
        public virtual double TrueBank(double amount, double rate)
```

```
{  
    return amount + (amount * rate);  
}  
  
}  
  
// first child class  
  
public class SimpleInterest: Interest  
{  
    public override double TrueBank(double amount, double rate)  
    {  
        return amount + (amount * rate) + 1000;  
    }  
}  
  
// second child class  
  
public class FixedInterest: Interest  
{  
    public override double TrueBank(double amount, double rate)  
    {  
        return amount + (amount * rate) + 1500;  
    }  
}  
  
public class Program
```

```
{  
  
public static void Main(string[] args)  
  
{  
  
    Interest i = new Interest();  
  
    double finalamount = i.TrueBank(5000.00,0.1);  
  
        Console.WriteLine("Normal interest for a holder  
"+finalamount);  
  
    i = new SimpleInterest();  
    finalamount = i.TrueBank(5000.00,0.1);  
        Console.WriteLine("Simple interest for a holder  
"+finalamount);  
  
    i = new FixedInterest();  
    finalamount = i.TrueBank(5000.00,0.1);  
        Console.WriteLine("Fixed interest for a holder  
"+finalamount);  
  
    }  
  
}
```

Output:

Normal interest for a holder 5500

Simple interest for a holder 6500

Fixed interest for a holder 7000

In the above example, we have created a base class named Interest, and two derived classes that is SimpleInterest and FixedInterest. In the base class, we used the virtual keyword with the method so that it can be overridden in the derived class using the override keyword. Here, we have given the same method name that is TrueBank and the same signature (number and type parameters) but different method definitions in the derived/child classes.

We created the object of the Interest class and provided the parameter list. If you closely observe the above parameter list, you will find the same parameters have been provided for each method call. Here the compiler only requires TrueBank() method to compile successfully and at the run-time desired methods get called respectively, based on which class's object is calling it.

Question. Define Exception handling. What are the uses of keywords TRY Throw and CATCH? Explain with an example writing a C++ program?

Answer: An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

- Throw – A program throws an exception when a problem shows up. This is done using a throw keyword.
- Catch – A program catches an exception with an exception

handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

- Try – A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows –

```
try {  
  
    // protected code  
  
} catch( ExceptionName e1 ) {  
  
    // catch block  
  
} catch( ExceptionName e2 ) {  
  
    // catch block  
  
} catch( ExceptionName eN ) {  
  
    // catch block  
  
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using throw statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of

exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs –

```
double division(int a, int b) {  
  
    if( b == 0 ) {  
  
        throw "Division by zero condition!";  
  
    }  
  
    return (a/b);  
  
}
```

Catching Exceptions

The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {  
  
    // protected code  
  
} catch( ExceptionName e ) {  
  
    // code to handle ExceptionName exception  
  
}
```

Above code will catch an exception of Exception Name type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows –

```
try {
```

```
// protected code
} catch(...) {

// code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

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

double division(int a, int b) {
    if( b == 0 ) {
        throw "Division by zero condition!";
    }
    return (a/b);
}

int main () {
    int x = 50;
    int y = 0;
    double z = 0;
```

```
try {  
  
    z = division(x, y);  
  
    cout << z << endl;  
  
} catch (const char* msg) {  
  
    cerr << msg << endl;  
  
}  
  
return 0;  
  
}
```

Because we are raising an exception of type `const char*`, so while catching this exception, we have to use `const char*` in catch block. If we compile and run above code, this would produce the following result – Division by zero condition!

Question. What is Access modifier in C++? Define each type and also differentiate between these.

Answer: The access modifiers of C++ are public, private, and protected.

One of the main features of object-oriented programming languages such as C++ is data hiding.

Data hiding refers to restricting access to data members of a class. This is to prevent other functions and classes from tampering with the class data.

However, it is also important to make some member functions and member data accessible so that the hidden data can be manipulated

indirectly.

The access modifiers of C++ allows us to determine which class members are accessible to other classes and functions, and which are not.

For example,

```
class Patient {  
  
private:  
  
    int patientNumber;  
    string diagnosis;  
  
public:  
  
    void billing() {  
        // code  
    }  
  
    void makeAppointment() {  
        // code  
    }  
};
```

Here, the variables `patientNumber` and `diagnosis` of the `Patient` class are hidden using the `private` keyword, while the member functions are made accessible using the `public` keyword.

Types of C++ Access Modifiers

In C++, there are 3 access modifiers:

- `public`
 - `private`
 - `protected`
-

public Access Modifier

- The `public` keyword is used to create public members (data and functions).
 - The public members are accessible from any part of the program.
-

Example 1: C++ public Access Modifier

```
#include <iostream>
```

```
using namespace std;
```

```
// define a class
```

```
class Sample {
```

```
    // public elements
```

```
public:
```

```
    int age;
```

```
    void displayAge() {
```

```
        cout << "Age = " << age << endl;
    }
};

int main() {

    // declare a class object
    Sample obj1;

    cout << "Enter your age: ";

    // store input in age of the obj1 object
    cin >> obj1.age;

    // call class function
    obj1.displayAge();

    return 0;
}
```

Output:

Enter your age: 20

Age = 20

In this program, we have created a class named `Sample`, which contains a `public` variable `age` and a `public` function `displayAge()`. In `main()`, we have created an object of the `Sample` class named `obj1`. We then access the public elements directly by using the codes `obj.age` and `obj.displayAge()`.

private Access Modifier

- The `private` keyword is used to create private members (data and functions).
- The private members can be accessed only from within the class.
- However, friend classes and friend functions can access private members.

Example 2: C++ private Access Specifier

```
#include <iostream>
```

```
using namespace std;
```

```
// define a class
```

```
class Sample {
```

```
    // private elements
```

```
private:
```

```
    int age;
```

```
// public elements

public:

void displayAge(int a) {

    age = a;

    cout << "Age = " << age << endl;

}

};

int main() {

    int ageInput;

    // declare an object

    Sample obj1;

    cout << "Enter your age: ";

    cin >> ageInput;

    // call function and pass ageInput as argument

    obj1.displayAge(ageInput);
```

```
return 0;
```

```
}
```

Output:

```
Enter your age: 20
```

```
Age = 20
```

In `main()`, the object `obj1` cannot directly access the class variable `age`.

```
// error
```

```
cin >> obj1.age;
```

We can only indirectly manipulate `age` through the public function `displayAge()`, since this function assigns `age` to the argument passed into it i.e. the function parameter `int a`.

protected Access Modifier

Before we learn about the `protected` access specifier, make sure you know about inheritance in C++.

- The `protected` keyword is used to create protected members (data and function).
- The protected members can be accessed within the class and from the derived class.

Example 3: C++ protected Access Specifier

```
#include <iostream>
```

```
using namespace std;
```

```
// declare parent class
```

```
class Sample {
```

```
    // protected elements
```

```
protected:
```

```
    int age;
```

```
};
```

```
// declare child class
```

```
class SampleChild : public Sample {
```

```
public:
```

```
    void displayAge(int a) {
```

```
        age = a;
```

```
        cout << "Age = " << age << endl;
```

```
    }
```

```
};
```

```
int main() {
```

```
    int ageInput;
```

```
// declare object of child class
```

```
SampleChild child;
```

```
cout << "Enter your age: ";
```

```
cin >> ageInput;
```

```
// call child class function
```

```
// pass ageInput as argument
```

```
child.displayAge(ageInput);
```

```
return 0;
```

```
}
```

Output:

```
Enter your age: 20
```

```
Age = 20
```

Here, `ChildSample` is an inherited class that is derived from `Sample`. The variable `age` is declared in `Sample` with the `protected` keyword. This means that `ChildSample` can access `age` since `Sample` is its parent class.

We see this as we have assigned the value

of `age` in `ChildSample` even though `age` is declared in the `Sample` class.

`public` elements can be accessed by all other classes and functions.

- `private` elements cannot be accessed outside the class in which they are declared, except by `friend` classes and functions.
- `protected` elements are just like the `private`, except they can be accessed by derived classes.

Specifiers	Same Class	Derived Class
<code>public</code>	Yes	Yes
<code>private</code>	Yes	No
<code>protected</code>	Yes	Yes

Question. Write short notes on the examples on each of the following with respect to C++?

a) Data Abstraction: 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.

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.

b) Overriding: C++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

1. #include <iostream>
2. using namespace std;
3. class Animal {
4. public:
5. void eat(){
6. cout<<"Eating...";
7. }
8. };
9. class Dog: public Animal

```
10. {
11. public:
12. void eat()
13. {
14.     cout<<"Eating bread...";
15. }
16. };
17. int main(void) {
18.     Dog d = Dog();
19.     d.eat();
20.     return 0;
21. }
```

Output:

```
Eating bread...
```

C) Encapsulation: 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.

Designing Strategy

Most of us have learnt to make class members private by default unless we really need to expose them. That's just good encapsulation. This is applied most frequently to data members, but it applies equally to all members, including virtual functions.

d) Virtual Functioning: A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is overridden. This especially applies to cases where a pointer of base class points to an object of a derived class.

For example, consider the code below:

```
class Base {  
    public:  
    void print() {  
        // code  
    }  
};  
  
class Derived : public Base {  
    public:  
    void print() {  
        // code  
    }  
};
```

Later, if we create a pointer of `Base` type to point to an object of `Derived` class and call the `print()` function, it calls the `print()` function of the `Base` class.

In other words, the member function of `Base` is not overridden.

```
int main() {  
  
    Derived derived1;  
  
    Base* base1 = &derived1;  
  
    // calls function of Base class  
  
    base1->print();  
  
    return 0;  
}
```

In order to avoid this, we declare the `print()` function of the `Base` class as virtual by using the `virtual` keyword.

```
class Base {  
  
    public:  
  
    virtual void print() {  
  
        // code  
  
    }  
  
};
```

Virtual functions are an integral part of polymorphism in C++. To learn more, check our tutorial on C++ Polymorphism.

Example 1: C++ virtual Function

```
#include <iostream>

using namespace std;

class Base {

public:

virtual void print() {

    cout << "Base Function" << endl;

}

};

class Derived : public Base {

public:

void print() {

    cout << "Derived Function" << endl;

}

};
```



```
int main() {  
  
    Derived derived1;  
  
    // pointer of Base type that points to derived1  
    Base* base1 = &derived1;  
  
    // calls member function of Derived class  
    base1->print();  
  
    return 0;  
}
```

Output

Derived Function

Here, we have declared the `print()` function of `Base` as `virtual`. So, this function is overridden even when we use a pointer of `Base` type that points to the `Derived` object `derived1`.

e) Constructor and destructor: Constructors and Destructors in C++

Constructor

A Constructor is a member function of a class. It is mainly used to initialize the objects of the class. It has the same name as the class. When an object is created, the constructor is automatically called. It

is a special kind of member function of a class.

Difference Between Constructor and Other Member Functions:

1. The Constructor has the same name as the class name.
2. The Constructor is called when an object of the class is created.
3. A Constructor does not have a return type.
4. When a constructor is not specified, the compiler generates a default constructor which does nothing.
5. There are 3 types of constructors:
 - Default Constructor
 - Parameterized Constructor
 - Copy constructor

A constructor can also be defined in the private section of a class.

Moving on with this article on Constructor and Destructor in C++

Default Constructor


A Default constructor is a type of constructor which doesn't take any argument and has no parameters.

Here is an Example Code

```
1    #include <iostream>
2
3    using namespace std;
4
5    class test {
6
7    public:
8
9    int y, z;
10   test()
11   {
12   y = 7;
13   z = 13;
14   }
```

```
11     };
12     int main()
13     {
14     test a;
15     cout <<"the sum is: "<< a.y+a.z;
16     return 1;
17     }
```

Output:



```
the sum is: 20
```

Explanation:

The above program is a basic demo of a constructor in c++. We have a class test, with two data members of type int called y and z. Then we have a default constructor, which assigns 7 and 13 to the variables y and z respectively.

The main function has an object of class test called a. When this object is created the constructor is called and the variables y and z are given values. The main function has a cout statement or a print statement. In this statement, the sum is printed. With respect to the object of the class, we access the public members of the class, that is, a.y gives the value of y and the same for z. We display the sum of y and z.

The default constructor works this way. When we do not provide a default constructor, the compiler generates a default constructor which does not operate.

Next, let us take a look at parameterized constructor.



Visit www.goseeko.com to access free study material as per your university syllabus