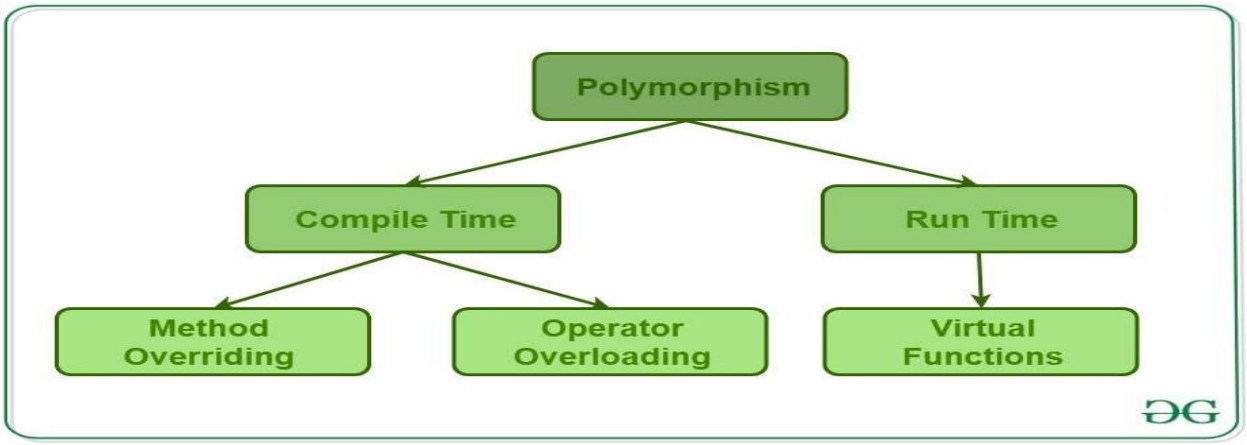
Polymorphism in C++

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. Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses 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

1. **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.

    - **Function Overloading**: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

```
// C++ program for function overloading
#include <bits/stdc++.h>

using namespace std;
class Geeks
{
   public:

   // function with 1 int parameter
   void func(int x)
   {
      cout << "value of x is " << x << endl;
   }
```

```
    // function with same name but 1 double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }

    // function with same name and 2 int parameters
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y << endl;
    }
};

int main() {

    Geeks obj1;

    // Which function is called will depend on the parameters passed
    // The first 'func' is called
    obj1.func(7);

    // The second 'func' is called
    obj1.func(9.132);

    // The third 'func' is called
    obj1.func(85,64);
    return 0;
}
```

Output:

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

In the above example, a single function named *func* acts differently in three different situations which is the property of polymorphism.

- **Operator Overloading**: C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.

Example:

```
// CPP program to illustrate
// Operator Overloading
#include<iostream>
using namespace std;

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
```

```
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

Output:

```
12 + i9
```

In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers. To learn operator overloading in details visit this link.

2. **Runtime polymorphism**: This type of polymorphism is achieved by Function Overriding.
   - **Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
// C++ program for function overriding

#include <bits/stdc++.h>
using namespace std;

class base
{
public:
    virtual void print ()
    { cout<< "print base class" <<endl; }

    void show ()
    { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
    void print () //print () is already virtual function in derived class,
                  //we could also declared as virtual void print () explicitly
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};

//main function
int main()
{
```

```
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, binded at runtime (Runtime polymorphism)
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();

    return 0;
}
```

Output:

```
print derived class

show base class
```

Virtual Function in C++

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions

Consider the following simple program showing run-time behavior of virtual functions.

```
// CPP program to illustrate
// concept of Virtual Functions

#include <iostream>
using namespace std;

class base {
public:
   virtual void print()
```

```
    {
        cout << "print base class" << endl;
    }

    void show()
    {
        cout << "show base class" << endl;
    }
};

class derived : public base {
public:
    void print()
    {
        cout << "print derived class" << endl;
    }

    void show()
    {
        cout << "show derived class" << endl;
    }
};

int main()
{
    base* bptr;
    derived d;
    bptr = &d;

    // virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Output:

```
print derived class
show base class
```

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.
Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is *print derived class* as pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time(output is *show base class* as pointer is of base type).

NOTE: If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

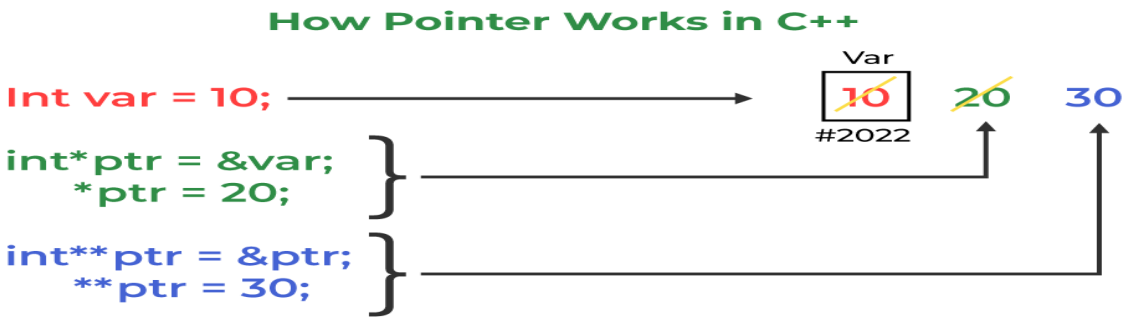
C++ Pointers

Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Iterating over elements in arrays or other data structures is one of the main use of pointers.

The address of the variable you're working with is assigned to the pointer variable that points to the same data type (such as an int or string).

Syntax:

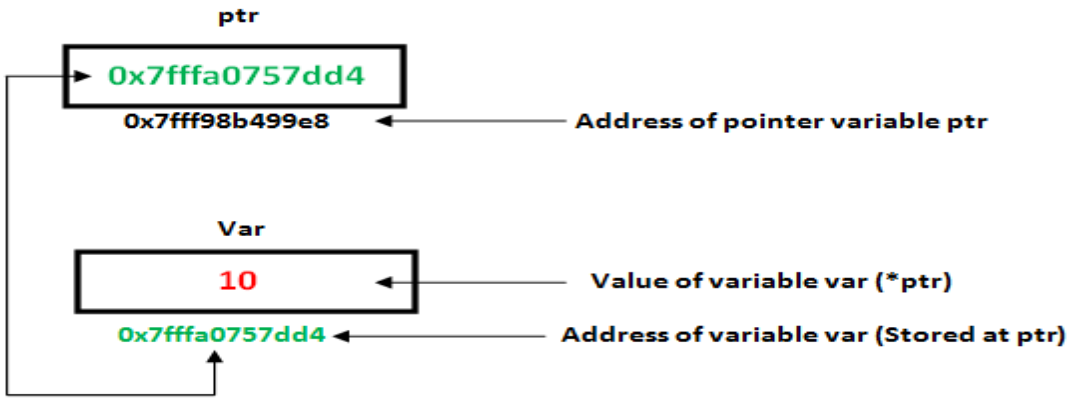
```
datatype *var_name;

int *ptr;   // ptr can point to an address which holds int data
```



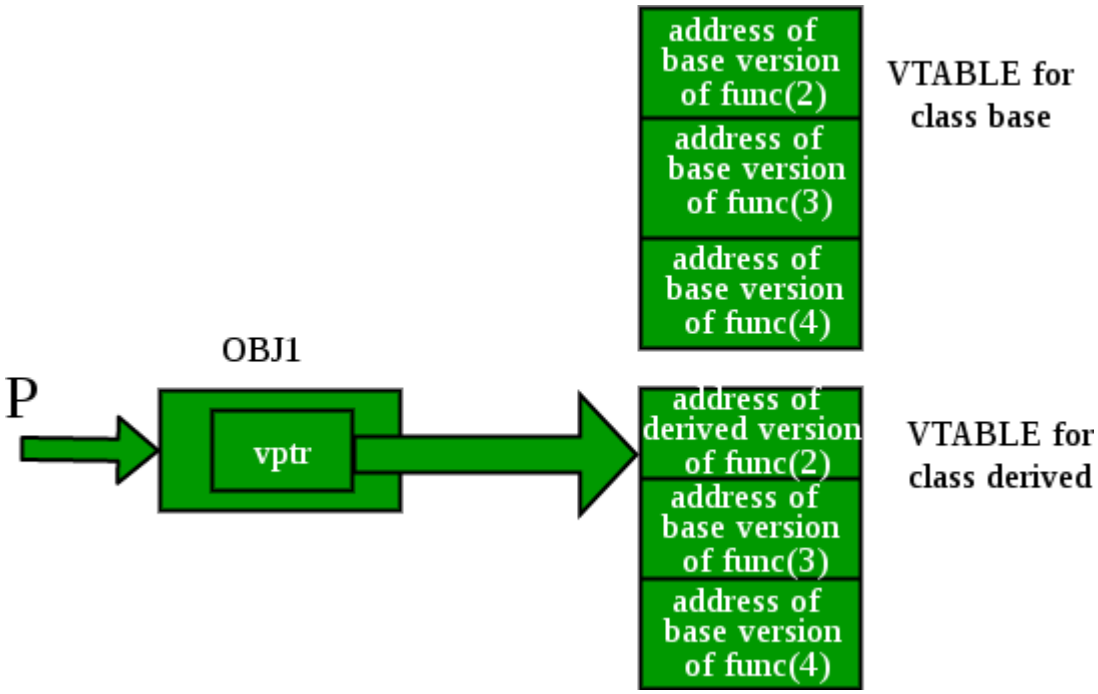
How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using the unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

The reason we associate data type with a pointer is **that it knows how many bytes the data is stored in**. When we increment a pointer, we increase the pointer by the size of the data type to which it points.




```
// C++ program to illustrate Pointers
#include <bits/stdc++.h>
using namespace std;
void geeks()
{
        int var = 20;
        // declare pointer variable
        int* ptr;
```




```
    // note that data type of ptr and var must be same
    ptr = &var;

    // assign the address of a variable to a pointer
    cout << "Value at ptr = " << ptr << "\n";
    cout << "Value at var = " << var << "\n";
    cout << "Value at *ptr = " << *ptr << "\n";
}
// Driver program
int main()
{
geeks();
return 0;
}
```

Output

```
Value at ptr = 0x7ffe454c08cc
Value at var = 20
Value at *ptr = 20
```

Working of virtual functions(concept of VTABLE and VPTR)

As discussed here, If a class contains a virtual function then compiler itself does two things:

1. If object of that class is created then a **virtual pointer(VPTR)** is inserted as a data member of the class to point to VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
2. Irrespective of object is created or not, **a static array of function pointer called VTABLE** where each cell contains the address of each virtual function contained in that class.

Consider the example below:

```
// CPP program to illustrate
// working of Virtual Functions
#include <iostream>
using namespace std;

class base {
public:
   void fun_1() { cout << "base-1\n"; }
   virtual void fun_2() { cout << "base-2\n"; }
   virtual void fun_3() { cout << "base-3\n"; }
   virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
   void fun_1() { cout << "derived-1\n"; }
   void fun_2() { cout << "derived-2\n"; }
   void fun_4(int x) { cout << "derived-4\n"; }
};

int main()
{
   base* p;
   derived obj1;
   p = &obj1;

   // Early binding because fun1() is non-virtual
   // in base
   p->fun_1();

   // Late binding (RTP)
   p->fun_2();

   // Late binding (RTP)
   p->fun_3();

   // Late binding (RTP)
   p->fun_4();

   // Early binding but this function call is
   // illegal(produces error) becasue pointer
   // is of base type and function is of
   // derived class
   // p->fun_4(5);
}
```

Output:

```
base-1
derived-2
base-3
base-4
```

Explanation: Initially, we create a pointer of type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.

Similar concept of **Late and Early Binding** is used as in above example. For fun_1() function call, base class version of function is called, fun_2() is overridden in derived class so derived class version is called, fun_3() is not overridden in derived class and is virtual function so base class version is called, similarly fun_4() is not overridden so base class version is called.

NOTE: fun_4(int) in derived class is different from virtual function fun_4() in base class as prototype of both the function is different.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
// An abstract class
class Test
{
   // Data members of class
public:
   // Pure Virtual Function
   virtual void show() = 0;

  /* Other members */
};
```

'this' pointer in C++

To understand 'this' pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?
The compiler supplies an implicit pointer along with the names of the functions as 'this'.
The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).
For a class X, the type of this pointer is 'X* '. Also, if a member function of X is declared as const, then the type of this pointer is 'const X *'.

Following are the situations where 'this' pointer is used:

1.When local variable's name is same as member's name
2.**To return reference to the calling object**

Pure Virtual Functions and Abstract Classes in C++ (Abstract Class & Function)

Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an **abstract class**. For example, let Shape be a base class. We cannot provide the implementation of function draw() in Shape, but we know every derived class must have an implementation of draw(). Similarly, an Animal class doesn't have the implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.
A **pure virtual function** (or abstract function) in C++ is a virtual function for which we can have an implementation, But we must override that function in the derived class, otherwise, the derived class will also become an abstract class. A pure virtual function is declared by assigning 0 in the declaration.

Example of Pure Virtual Functions

```
// An abstract class
class Test {
        // Data members of class
public:
        // Pure Virtual Function
        virtual void show() = 0;

        /* Other members */
};
```

A complete example:

A pure virtual function is implemented by classes which are derived from a Abstract class. Following is a simple example to demonstrate the same

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

class Base
{
   int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base
{
    int y;
public:
    void fun() { cout << "fun() called"; }
};

int main(void)
{
    Derived d;
    d.fun();
    return 0;
}
```

Output:

```
fun() called
```

Some Interesting Facts/Rules

1. A class is abstract if it has at least one pure virtual function.

2. We can have pointers and references of abstract class type.

3. If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.

4. An abstract class can have constructors.

5. An abstract class in C++ can also be defined using struct keyword.

Exception Handling in C++

In C++, exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. The process of handling these exceptions is called exception handling. Using the exception handling mechanism, the control from one part of the program where the exception occurred can be transferred to another part of the code.
So basically using exception handling in C++, we can handle the exceptions so that our program keeps running.

What is a C++ Exception?

An exception is an unexpected problem that arises during the execution of a program our program terminates suddenly with some errors/issues. Exception occurs during the running of the program (runtime).
Types of C++ Exception

1. **Synchronous:** Exceptions that happen when something goes wrong because of a mistake in the input data or when the program is not equipped to handle the current type of data it's working with, such as dividing a number by zero.
2. **Asynchronous**: Exceptions that are beyond the program's control, such as disc failure, keyboard interrupts, etc.

C++ try and catch

C++ provides an inbuilt feature for Exception Handling. It can be done using the following specialized keywords: try, catch, and throw with each having a different purpose.
Syntax of try-catch in C++

```
try {
    // Code that might throw an exception
    throw SomeExceptionType("Error message");
 }
catch( ExceptionName e1 )  {
    // catch block catches the exception that is thrown from try block
 }
```

1. try in C++

The try keyword represents a block of code that may throw an exception placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, try block throws that exception.
2. catch in C++

The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.
3. throw in C++

An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.
Note: *Multiple catch statements can be used to catch different type of exceptions thrown by try block.*
The try and catch keywords come in pairs: We use the try block to test some code and If the code throws an exception we will handle it in our catch block.

Why do we need Exception Handling in C++?

The following are the main advantages of exception handling over traditional error handling:

1. **Separation of Error Handling Code from Normal Code***:* There are always if-else conditions to handle errors in traditional error handling codes. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.

2. **Functions/Methods can handle only the exceptions they choose***:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller.
   In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

3. **Grouping of Error Types***:* In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, and categorize them according to their types.

Examples of Exception Handling in C++

The following examples demonstrate how to use a try-catch block to handle exceptions in C++.

Example 1

The below example demonstrates throw exceptions in C++.

```
// C++ program to demonstate the use of try,catch and throw
// in exception handling.

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

int main()
{

	// try block
	try {
		int numerator = 10;
		int denominator = 0;
		int res;

		// check if denominator is 0 then throw runtime
		// error.
		if (denominator == 0) {
			throw runtime_error(
				"Division by zero not allowed!");
		}

		// calculate result if no exception occurs
		res = numerator / denominator;
		//[printing result after division
		cout << "Result after division: " << res << endl;
	}
	// catch block to catch the thrown exception
	catch (const exception& e) {
		// print the exception
		cout << "Exception " << e.what() << endl;
	}

	return 0;
}
```

Output

```
Exception Division by zero not allowed!
```

Example 2

The following is a simple example to show exception handling in C++. The output of the program explains the flow of execution of try/catch blocks.

- CPP

```
// C++ program to demonstate the use of try,catch and throw
// in exception handling.

#include <iostream>
using namespace std;

int main()
{
   int x = -1;

   // Some code
   cout << "Before try \n";

   // try block
   try {
      cout << "Inside try \n";
      if (x < 0) {
         // throwing an exception
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }

   // catch block
   catch (int x) {
      cout << "Exception Caught \n";
   }

   cout << "After catch (Will be executed) \n";
   return 0;
}
```

Output

```
Before try Inside try

Exception Caught

After catch (Will be executed)
```

Properties of Exception Handling in C++

Property 1

There is a special catch block called the 'catch-all' block, written as catch(…), that can be used to catch all types of exceptions.

Property 2

Implicit type conversion doesn't happen for primitive types.

Property 3

If an exception is thrown and not caught anywhere, the program terminates abnormally.

Property 4

Unlike Java, in C++, all exceptions are unchecked, i.e., the compiler doesn't check whether an exception is caught or not (See this for details). So, it is not necessary to specify all uncaught exceptions in a function declaration. However,exception-handling it's a recommended practice to do so.

Property 5

In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using "throw; ".

Property 6

When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.

Limitations of Exception Handling in C++

The exception handling in C++ also have few limitations:

- Exceptions may break the structure or flow of the code as multiple invisible exit points are created in the code which makes the code hard to read and debug.
- If exception handling is not done properly can lead to resource leaks as well.
- It's hard to learn how to write Exception code that is safe.
- There is no C++ standard on how to use exception handling, hence many variations in exception-handling practices exist.

Conclusion

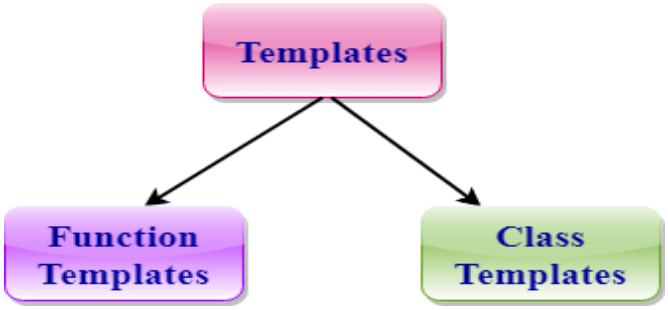
Exception handling in C++ is used to handle unexpected happening using "try" and "catch" blocks to manage the problem efficiently. This exception handling makes our programs more reliable as errors at runtime can be handled separately and it also helps prevent the program from crashing and abrupt termination of the program when error is encountered.

C++ Templates

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates



Function Templates:

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

```
1. template < class Ttype> ret_type func_name(parameter_list)
2. {
3.     // body of function.
4. }
```

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Let's see a simple example of a function template:

```
1.  #include <iostream>
2. using namespace std;
3. template<class T> T add(T &a,T &b)
4. {
5.     T result = a+b;
6.     return result;
7.
8. }
9. int main()
10. {
11.   int i =2;
12.   int j =3;
13.   float m = 2.3;
```

```
14.  float n = 1.2;
15.  cout<<"Addition of i and j is :"<<add(i,j);
16.  cout<<'\n';
17.  cout<<"Addition of m and n is :"<<add(m,n);
18.  return 0;
19. }
```

Output:

```
Addition of i and j is :5
Addition of m and n is :3.5
```

In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
1.  template<class T1, class T2,.....>
2.  return_type function_name (arguments of type T1, T2....)
3.  {
4.      // body of function.
5.  }
```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Let's see a simple example:

```
1.  #include <iostream>
2.  using namespace std;
3.  template<class X,class Y> void fun(X a,Y b)
4.  {
5.      std::cout << "Value of a is : " <<a<< std::endl;
6.      std::cout << "Value of b is : " <<b<< std::endl;
7.  }
8.
9.  int main()
10. {
11.    fun(15,12.3);
12.
13.    return 0;
```

```
}
```

Output:

```
Value of a is : 15
Value of b is : 12.3
```

In the above example, we use two generic types in the template function, i.e., X and Y.

Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

Let's understand this through a simple example:

```
#include <iostream>
using namespace std;
template<class X> void fun(X a)
{
    std::cout << "Value of a is : " <<a<< std::endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
    std::cout << "Value of b is : " <<b<< std::endl;
    std::cout << "Value of c is : " <<c<< std::endl;
}
int main()
{
   fun(10);
   fun(20,30.5);
   return 0;
}
```

Output:

```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
```

In the above example, template of fun() function is overloaded.

Restrictions of Generic Functions

Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

Let's understand this through a simple example:

```
1. #include <iostream>
2. using namespace std;
3. void fun(double a)
4. {
5.     cout<<"value of a is : "<<a<<'\n';
6. }
7.
8. void fun(int b)
9. {
10.    if(b%2==0)
11.    {
12.        cout<<"Number is even";
13.    }
14.    else
15.    {
16.        cout<<"Number is odd";
17.    }
18.
19. }
20.
21. int main()
22. {
23.    fun(4.6);
24.    fun(6);
25.    return 0;
26. }
```

Output:

```
value of a is : 4.6
Number is even
```

In the above example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities. First one is displaying the value and the second one determines whether the number is even or not.

CLASS TEMPLATE

Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
  .
  .
}
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

```
class_name<type> ob;
```

where class_name: It is the name of the class.

type: It is the type of the data that the class is operating on.

ob: It is the name of the object.

Let's see a simple example:

```
#include <iostream>
using namespace std;
template<class T>
class A
{
    public:
    T num1 = 5;
    T num2 = 6;
    void add()
    {
        std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
    }

};

int main()
{
    A<int> d;
    d.add();
    return 0;
}
```

Output:

```
Addition of num1 and num2 : 11
```

In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
1. template<class T1, class T2, .....>
2. class class_name
3. {
4.    // Body of the class.
5. }
```

Let's see a simple example when class template contains two generic data types.

```
1. #include <iostream>
2.      using namespace std;
3.      template<class T1, class T2>
4.     class A
5.     {
6.         T1 a;
7.          T2 b;
8.         public:
9.         A(T1 x,T2 y)
10.       {
11.          a = x;
12.         b = y;
13.       }
14.         void display()
15.         {
16.              std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
17.          }
18.     };
19.
20.     int main()
21.     {
22.         A<int,float> d(5,6.5);
23.          d.display();
```

```
24.         return 0;
25.     }
```

Output:

```
Values of a and b are : 5,6.5
```

Nontype Template Arguments

The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types. **Let' s see the following example:**

```
1. template<class T, int size>
2. class array
3. {
4.         T arr[size];          // automatic array initialization.
5. };
```

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
1. array<int, 15> t1;                    // array of 15 integers.
2. array<float, 10> t2;                  // array of 10 floats.
3. array<char, 4> t3;                    // array of 4 chars.
```

Let's see a simple example of nontype template arguments.

```
1. #include <iostream>
2. using namespace std;
3. template<class T, int size>
4. class A
5. {
6.     public:
7.     T arr[size];
8.     void insert()
9.     {
10.        int i =1;
11.        for (int j=0;j<size;j++)
12.        {
13.            arr[j] = i;
14.            i++;
```

```
        }
    }

    void display()
    {
        for(int i=0;i<size;i++)
        {
            std::cout << arr[i] << " ";
        }
    }
};
int main()
{
    A<int,10> t1;
    t1.insert();
    t1.display();
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

In the above example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.