Polymorphism

It is a one of the crucial features of OOP means 'one name multiple forms'. It is implemented using the overloaded function and operators. The overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to compiler at the compile time and compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early binding or static binding or static linking. Also known as compile time polymorphism.

Early binding simply means that an object is bound to its function call at compile time.

```
class A
{
    int x;
    public:
        void show(){ ... }
};
class B: public A
{
    public:
        void show(){ ... }
};
```

show() is not overloaded.

It would be nice if appropriate member function could be selected while the program is running. This is known as run time polymorphism It is achieved by Virtual function

At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as late binding. It is also known as dynamic binding because the selection of appropriate function is done dynamically at run time.

Dynamic binding is one of the powerful features of C++; requires the use of pointer to Object.

Pointers

Pointer is a derived data type that refers to another data variable by storing the variable memory address rather than data.

A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

They provide an alternative approach to access other data objects.

Syntax for declaration.

```
data-type *pointer-variable;
```

A pointer variable can point to any type of data available in C++.

```
int *ptr = a;
ptr = &a;
```

We can dereference a pointer, to allow us to get the content of memory location that the pointer points to. by using indirection operator *. We can also change the content of memory location.

Pointer Expression and its Arithmatic

- A pointer can be incremented or decremented.
  ptr++ or ++ptr
  ptr-- or --ptr
- Any integer can be added to or subtracted from a pointer.
- one pointer can be subtracted from another.

Array Pointer

```
int arr[10];
int *ptr;
ptr = &arr[0];
      or
ptr = arr[];
ptr = arr;
```

↓

Points to index 0 of array.

Array of pointer:

datatype *arr-name[size];

E.g; int *arptr[10];

or

int *arptr[10] = {1, 2, 3, 4, 5};

Pointer and Strings

Same as C —

Pointers to Functions

The pointer to function is known as callback function. Function pointers is used to refer a function. These can't be dereferenced.
In C++, we are allowed to select a function at runtime.
We can also pass a function as an argument to another function. Here, the function is passed as a pointer.
C++ allows us to compare two function pointer.

C++ provides two types of function pointers

- pointer to static member function
- pointer to non-static member function (requires hidden argument).

Syntax.

```
data-type (*fun-name)( );

int (*add(int x));

int (*fptr)(int, int);
```

Pointers to Objects

Let us declare an item variable x and pointer ptr to x as

```
item x;
item *ptr = &x;
```

```
x.get(100, 75.50);
x.show();
```

is equivalent to

```
ptr->get(100, 75.50);
ptr->show();
```

Since *ptr is an alias of x.

```
(*ptr).show();
```

The parenthesis are necessary because dot operator has high precedance than indirection operator.

We can also create array of object using pointer.

```
item *ptr = new item[10];
```

this Pointer

C++ uses a unique keyword called this to represent an object that invokes a member function. this is a pointer that points to the object for which this function was called.

The starting address is the same as the address of first variable in class structure.

```
Class ABC
{
    int a;
    ...
    ...
};
```

This unique pointer is automatically passed to a member function when it is called. 'this' acts as an implicit argument to all the member functions.

a = 123; ≈ this → a = 123;

The argument is implicitly passed using the pointer this.

~~retl~~ return *this;

return the object that invoked the function.

Virtual Function

Polymorphism is the ability to refer to objects without any regard to their classes.

We use the pointer to base class to refer to all the derived objects. ~~We~~ Base pointer is made to contain the address of a derived class, always executes the function in base class.

The compiler simply ignores the contents of the pointer and chooses the function that matches the type of pointer. Here the polymorphism is achieved by virtual function.

When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration.

When a function is made virtual, C++ determines which function to use at runtime based on the type of object pointed to by the base pointer, rather than type of the pointer.

Thus, by making the base pointer to point to different version of virtual function.

```
class Base
{
    public: display() { ... };
    virtual void show() { ... };
};

Class Derived : public Base
{
    public:
        void display() { ... };
        void show() { ... };
};
```

```
int main()
{
    Base B;
    Derived D;
    Base *bptr;

    bptr = &B;
    bptr -> display();   // calls base version
    bptr -> show();      // calls base version.

    bptr = &D;
    bptr -> display();   // calls base version.
    bptr -> show();      // calls Derived version

// This is because that display is not
    declared virtual.

    return 0;
}
```

* Note —

We can use the dot operator to access member function ~~to call the virtual~~ with Objects. But Run time polymorphism is achieved only when a virtual function is accessed through a pointer to a base class.

Rules for Virtual Functions

1. The virtual functions must be members of some class.
2. They cannot be static member.
3. They are access by using object pointers.
4. A virtual function can be a friend of another class.
5. ~~A virtual function can be a friend of another class.~~
5. A virtual function in a base class must be defined, even though it may not be used.
6. We cannot have virtual constructor but can have virtual.
7. We can't use a pointer to derived class to access an object of base type but reverse is true.
   (~~bas~~ (pointer to Derived class).

8. The prototype of base class version of a virtual function and all the derived class versions must be identical.
If it happens, C++ considers them as overloaded functions and the virtual function mechanism is ignored.

9. If a virtual function is defined in base class, it need not necessarily redefined in the derived class.
In such cases, calls will invoke the base function.

Pure Virtual Function

A virtual function, equated to zero is called a pure virtual function.
It is a function declared in a base class that has no definition relative to the base class.
A class containing such pure virtual function is called an abstract class or abstract base class.

```
virtual void display() = 0;
```

It can be redefine in the derived class.

Templates

Templates is one of the features added to C++. It enable us to define generic class and function and thus provides support for generic programming. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

A template can be used to create a family of classes or functions.

A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type.

Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the template are called parameterized class or function. template

Class Template.

Template allow us to define generic classes. It is a simple process to create a generic class using a template with an anonymous type.
The general format of a class template is:

```
template <class T>
class classname
{
    // ... class member specification ...
    // with anonymous type T
};
```

The class template definition is very similar to an ordinary class definition except the prefix template <class T> and use of type T. This prefix tells the compiler that we are going to declare a template and T as a type name in the declaration.

A class created from a class template is called a template class.
Syntax for defining an object of a template:-

```
classname <type> objectname (arg.list);
```

The process of creating a specific class from class template is called intantiation.

Class Templates with Multiple Paramdi

```
template <class T1, class T2, ...>
Class classname
{
   ...
};
```

Function Template

They are used to create a family of functions with different argument types. The general format of function template.

```
template <class T>
return type  funcname(arg of type T)
{
   ...
   ...
}
```

E.g;

```
template <class T>
Void swap(T &x, T &y)
{
    T temp = x;
    x = y;
    y = temp;
}.
```

For Multiple parameters

```
template <class T1, class T2,>.
{
    ret
returntype func-name(arg. of T1, T2.. type)
{
    ...
    ...
    ...
}
```

Overloading of Template Function.

A template function may be overloaded either by template functions or ordinary functions of its name. In such cases, the overloading resolution is accomplished as follows:

1. Call an ordinary function that has an exact match
2. Call a template function that could be created with an exact match.

3. Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found.
Note that no automatic conversions are applied to arguments on the template function.

Member Function Template.

We could define the member function of the class outside as well. But remember that the member functions of the template classes themselves are parameterized by the type argument. and therefore these must be defined by function template.

```
Template <class T>
returntype classname <T> :: funcname(arg).
{
   ...
   ...
   ...
};
```

~~Non-Type Template or~~

Member functions of a class template must be defined as function templates using the parameters of class template.

Non-Type Template argument

** Multiple argument

It is also possible to use non-type template arguments.

```
template <class T, int size>
Class array
{
    T a[size];
    //...
};
```

This template supplies the size of array as an argument. This implies that the size of the array is known to the compiler at compile time itself.

```
array <int, 10> a1;
```

array of 10 integers.

The size is given as an argument to the template class.

Exception Handling

Error/bugs

- Logic error (due to poor understanding of algorithm).
- Syntactic errors (due to poor knowledge of language)

Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing.

Exception handling was not part of original C++. It is a new feature added to ANSI C++.

This provides a type-safe, integrated approach, for coping with the unusal predictable problems that arose while executing a ~~prog~~ program.

Basics of Exception handling.

Exceptions

- Synchronous exceptions { out of range } index
- asynchronous exceptions. { beyond the control of Program }

The Purpose of exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken.

1. Hit the exception
2. Throw the exception
3. Catch the exception.
4. Take corrective action (Handle the exception)

Mechanism

C++ exception handling mechanism is basically built upon three keywords try, throw, catch.

The keyword try is used to preface a block of statements which may generate exceptions.

When an exception is detected, it is thrown using a throw statement in try block.

A catch block defined by the keyword 'catch' catches the exception thrown by the throw statement in the try block and handles it appropriately.

The catch block that catches an exception must immediately follow the try block that throws the exception.

The general form of these two blocks are as:

```
...
...
try
{
    ...
    throw exception;
    ...
    ...
}

catch (type arg)
{
    ...
    ...
    ...
}
...
...
```

Exceptions are objects used to transmit information about a problem.

If the type of Object thrown matches the arg. type in catch statement, then catch block is executed for handling the exception

If they do not match, the program is aborted with the help of abort().

When no exception is detected and thrown, the control goes to the statement immediately after Catch-block (skipped Catch)

Exception are thrown by functions that are invoked from try. The point at which throw is executed is called the throw point.

Once exception is thrown to the catch block, control can't return to the throw point. This kind of relationship is shown in code -

```
type func(arg list)    // func. with exception.
{
    ...
    ...
    throw(object);
    ...
} ...
try
{
    ...
    ...        Invoke function here.
}.
Catch (type arg).
{
    ...        handles exception.
}
```

The try block is immediately followed by the catch block, irrespective of the location of throw point.

Throwing Exception

When an exception that is desired to be handled is detected, it is thrown using the throw statement in one of the following forms:

```
throw (exception);
throw exception;
throw;                    // used for rethrowing an exception
```

The operand object may be of any type, including constants. It is also possible to throw objects not intended for error handling.

When an exception is thrown, it will be caught by the catch statement associated with the try block. The control exits the current try block, and is transferred to the catch block after that try block.

Throw point can be in a deeply nested scope within a try block or in a deeply nested function call. In any case, control is transferred to the catch statement

Catching Mechanism

A catch block looks like a function definition and is of the form

```
catch (type arg)
{
    // Statements for
    // managing exceptions
}
```

type indicates the type of exception that catch block handles.

The catch statement catches an exception whose type matches with the type of catch argument. When it is caught, block is executed.

If the parameter in the catch statement is named, then the parameter can be used in exception-handling code.

Multiple Catch Statements

It is possible that a program segment has more than one condition to throw an exception. In such cases, we can use multiple catch with a try.

```
try
{
  ...
}
catch (type1 arg)
{
    block1
}
catch (type2 arg)
{
    block2
}
...
...
catch (typeN ag)
{
    blockN
}.
```

It is possible that arguments of several catch statements match the type of an exception. In such cases, the first handler that matches the exception.

When no match is found, the program is terminated.

When the try block does not throw any exceptions and it completes normal execution, control passes to the first statement after the last catch handler associated with that try block

We can force a catch statement to catch all exceptions instead of a certain type alone.

```
Catch (...)
{
        // statements for
    processing
        // all exceptions
}
```

Catching Class Types as Exceptions

It is always a Good practice to create custom objects corresponding to different types of errors. The objects can then be suitably handled by exception handlers to take necessary action.

Rethrowing an Exception

A handler may decide to rethrow the exception caught without processing. To invoke throw without any arguments as shown:

```
throw;
```

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

When an exception is rethrown, it will not be caught by same catch statement or any other catch in that group. Catch handler itself may detect and throw an exception.

Specifying Exceptions

It is possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to the function definition.

General form of using an exception specification is:

```
type function(arg-list) throw(type-list)
{
    ...
    ... function body
    ...
}
```

The type-list specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination.

```
throw();    // Empty list
```

The restriction applies only when throwing an exception out of the function.

Manipulating Strings

ANSI Standard C++ now provides a new class called string. This class improves on the conventional C-strings in several ways.

The string objects may be used like any other built-in type data. It is not a part of STL, String is treated as another container class by C++.

<cstring> or <string.h>

The string class is very large and includes many constructors, member functions and operators. We use these to achieve the following

Commonly used String Constructors —

String(); → For creating an empty string

String(const char *str); → For creating a string object from a null-terminated string

String(const string &str); → For creating a string object

Operators for String Objects.

=, +, +=, ==, !=, <, <=, >, >=

[] → Subscription

<< → output

>> → Input

Using cin and >> operator we can read only one word of a String while the getline() function permits us to read a line of text containing embedded blanks

Namespace Scope

ANSI C++ standard has added a new keyword namespace to define a scope that could hold global identifiers.

The using namespace statement specifies that the members defined in Std namespace will be used frequently throughout the program.

Defining a namespace —

```
namespace  namespace_name
{
    // declaration of variables, functions, class, etc
}
```

~~We can use~~

```
using namespace name-space-name;
    // using directive.
using namespace-name::membername;
            // using declaration.
```

```
namespace NS1
{
    ...
    ...
    namespace NS2
    {
        int m=100;
    }
    ...
    ...
}
```

```
cout<< NS1::NS2::m;
```

New data type → bool, wchar_t

wchar_t → holds 16 bit wide characters

Class Implementation

Explicit keyword → It is used to declare class constructors to be "explicit" constructors.

mutable keyword → we can make a data item modifiable by declaring it as mutable.

```
mutable int m;
```

Stack Unwinding

In C++, stack unwinding refers to the process of systematically removing entries from the function call stack at runtime.

This usually ~~refers~~ happens in two main scenarios:

1. Exception Handling
   When an exception is thrown inside a function, the stack unwinds from the throw point back to the caller function. This involves:
   - Calling destructors in reverse order
   - transferring control to catch block.

2. Function Exiting Normally:
When a function finishes its execution and returns normally, its stack frame also undergoes unwinding. This involves calling the destructors for the automatic objects in the function as it "pops" off the stack.

Stack unwinding ensures orderly resource clean up and exception handling in C++.

Key points

- only automatic objects are unwound.
- Nested exceptions
- performance considerations.