Template: It is a technique that enables us to define generic functions and classes so that one function or class can handle many different data types. With the help of templates, we can design a single function or class that operates on data of many types, instead of having to create a separate function or class for each type. When templates are used with function, they are called function templates. And when they are used with class, they are called class templates.

For example, if add( ) function is defined as a function template, then it can be used for adding various types of data such as int, float, double etc.

If a class A is defined as a class template, then it can be used to operate on various data types, both for data members and member functions.

Advantages/Benefits of templates:

- Templates support generic programming. That means, we can create one generic version of our function or class that can operate on various types of information.
- They can be used in situations that result in duplication of same code for multiple data types.
- They deliver fast, efficient and robust code.
- They are type-safe, easier to write and understand.

Types of templates:

A. Function template
B. Class template

A. Function template: When template is used with function, it is called function template. It can be used to create a family of functions that support various argument types. A single function template can work with different data types. It overcomes the limitation of a normal function that can operate only on a single data type.

General format/syntax:

```
template <class (type) >
return_type function_name (arguments of (type))
{
    //body of function
}
```

By: Manoj Sapkota, U.T.C

Example:

```
template <class T >
void add(T num1, T num2 )
{
   //body of function
}
```

Let us consider a following program example. *[Example program- 63]*

Example program- 63:

```
template< class T >
void add(T num1,T num2)
{
    T sum;
    sum=num1+num2;
    cout<<"Sum= "<<sum<<endl;
}
int main()
{
    int a1=2, a2=4;
    float b1=4.2f, b2=1.5;
    cout<<"Calculation for integer values: "<<endl;
    add(a1,a2);
    cout<<"Calculation for floating point values: "<<endl;
    add(b1,b2);
    return 0;
}
```

Output: Calculation for integer values:
Sum= 6
Calculation for floating point values:
Sum= 5.7

Explanation of the program: In the above program, the function add( ) is defined as a function template. When integer values are passed as argument, the compiler automatically replaces the type 'T' in the function add( ) as integer type, and when floating point values are passed, the compiler replaces them with floating point type. Thus, the function can handle various data types.

Example program- 64, Write a program to define a function template to return maximum of two integers, floats or characters.

```
template< class T >
T maximum(T num1,T num2)
{
  T max= (num1>num2)? num1 : num2;
  return max;
}

int main()
{
    int a1=4, a2=7;
    float b1=4.2, b2=7.3;
    char c1='a', c2='c';
    cout<<"Maximum integer value: "<<maximum(a1,a2)<<endl;
    cout<<"Maximum floating point value: "<<maximum(b1,b2)<<endl;
    cout<<"Maximum character based on ASCII value: "<<maximum(c1,c2)<<endl;
    return 0;
}
```

Output: Maximum integer value: 7
Maximum floating point value: 7.3
Maximum character based on ASCII value: c

Example program- 65, Write a program to define a function template that interchanges the values of two arguments sent to it (the arguments should be char, int and float types).

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

```
int main()
{
    int a1=4, a2=7;
    float b1=4.2, b2=7.3;
    char c1='a', c2='c';
    cout<<"Before swapping.. "<<endl;
    cout<<"a1="<<a1<<" "<<"a2="<<a2<<endl;
    cout<<"b1="<<b1<<" "<<"b2="<<b2<<endl;
    cout<<"c1="<<c1<<" "<<"c2="<<c2<<endl;
    swapValue(a1,a2);
    swapValue(b1,b2);
    swapValue(c1,c2);
    cout<<"After swapping.. "<<endl;
    cout<<"a1="<<a1<<" "<<"a2="<<a2<<endl;
    cout<<"b1="<<b1<<" "<<"b2="<<b2<<endl;
    cout<<"c1="<<c1<<" "<<"c2="<<c2<<endl;
    return 0;
}
```

Function template with multiple parameters: We can define a function to handle various data types within a single call. For such operation, we can use more than one generic data type in template statement using a comma separated list as shown below.

General format/ syntax:

```
template <class (type1), class (type2), … >
return_type function_name (arguments of (type1), (type2), ….)
{
    //body of function
}
```

Example:

```
template <class T1, class T2 >
void add (T1 num1, T2 num2)
{
  cout<<num1+num2;
}
```

An example program to demonstrate the use of function template with multiple parameters is shown below. *[Example program- 66]*

By: Manoj Sapkota, U-TEC

Example program- 66:

```
template <class T1,class T2>
void display(T1 x,T2 y)
{
    cout<<x<<" & "<<y<<endl;
}
int main()
{
    cout<<"Passing integer and String type parameters"<<endl;
    display(9,"Chitwan");
    cout<<"Passing floating point and integer type parameters"<<endl;
    display(2.5,7);

    return 0;
}
```

Output: Passing integer and String type parameters
9 & Chitwan
Passing floating point and integer type parameters
2.5 & 7

B. Class templates: When template is used with class, it is called class template. When a class is defined as a class template, then it can be used to operate on different data types.

Sometimes, we might need a class implementation that is same for various classes, which differ only by the data types used. We would need to create a different class for each data type or create a different data member and functions within a single class. This leads to code redundancy and difficult to maintain, as a change in one class or a function should be performed on all class or functions. However, class templates make it easy to reuse the same code for all data types.

General format/ syntax:

```
template <class (type)>
class class_name
{
//class member specification
//with (type) whenever appropriate
};
```

By: Manoj Sapkota,

Example:

```
template <class T>
class class_name
{
  T a, b;
  public:
    void getData( T x, T y);
};
```

While creating an object of a class, it is necessary to mention the type of data to be used in that object.

Syntax for creating object:

Class_name <data_type> objectname;

For example:

For integer data,

ABC <int> a1; where, a1 is object of class 'ABC'.

For float data,

ABC <float> b1; where, b1 is object of class 'ABC'.

For integer data,

ABC <char> c1; where, c1 is object of class 'ABC'.

Let us consider a following program example. *[Example program- 67]*

Example program- 67:

```
template <class T>
class Addition
{
    T x, y;
public:
    void setData(T a, T b)
    {
        x=a;
        y=b;
    }
    void add()
    {
        cout<<"Sum: "<<x+y<<endl;
    }
};
```

Manoj Sapkota, U-TEC

```
int main()
{
    Addition <int> objA;
    objA.setData(2,5);
    objA.add();
    Addition <float> objB;
    objB.setData(3.2, 5.3);
    objB.add();
    return 0;
}
```

Output: Sum: 7
Sum: 8.5

Explanation of the program: In the above program, the class Addition is defined as a class template. It has data members as generic type T, and member function operating on these types. When the object is created and initialized with integer values in its data members, then the type 'int' should be mentioned while creating it, so that the compiler replaces the type T as integer type during initialization. Similarly, when the data members are initialized with floating point values, then the type 'float' should be mentioned while creating object as shown in above program. All the 'type T' used in the member functions are also replaced by the particular data-types. Thus, the class Addition performs addition for both integer and floating point values.

Class templates with multiple parameters: As similar to function-templates, we can use more than one generic data types in a class template. They are declared as a comma-separated list within the template as shown below.

General format:

```
template <class (type1), class (type2), … >
class class_name {
//class member specification
//with (type1) and (type2) whenever appropriate
};
```

Example:

```
template <class T1, class T2>
class Abc {
      T1 a;
      T2 b;
      public:
        void getData( T1 x, T2 y);
};
```

By: Manoj Sapkota, U.TEC

General format of creating object:
Class_name <data_type1, data_type2, … > objectname;

Example:
Abc <int, float > obj1;
Abc <int, char> obj2;

An example program to demonstrate the use of class template with multiple parameters is shown below. *[Example program- 68]*

Example program- 68:

```
template <class T1, class T2>
class Addition
{
    T1 x;
    T2 y;
public:
    void setData(T1 a, T2 b)
    {
        x=a;
        y=b;
    }
    void display()
    {
        cout<<x<<" & "<<y<<endl;
    }
};
int main()
{
    Addition <int,string> objA;
    objA.setData(9,"Chitwan");

    Addition <float,int> objB;
    objB.setData(2.5, 7);

    objA.display();
    objB.display();

    return 0;
}
```

Output: 9 & Chitwan
2.5 & 7

Differentiate between class template and function template:

Function Template	Class Template
1. When a template is used with function, it is called function template.	1. When a template is used with class, it is called class template.
2. It can be used to create a function that supports various argument types.	2. It can be used to create a class that can be used to operate on various data types.
3. Function template cannot have default template parameters.	3. Class template can have default template parameters.

4. Example:
template <class T>
void add(T num1, T num2){
//body of function
}
	4. Example:
	template <class T>
	class Class_name
	{
	T a,b;
	public:
	void getData(T x, T y);
	};

5. Compiler automatically replaces the template-type based on the type of parameters passed.	5. While creating an object of a class, it is necessary to mention the type of data to be used in that object.
6. A basic program example that illustrates the concept of function template. *[Example program- 63]*	6. A basic program example that illustrates the concept of class template. *[Example program- 67]*

Exception handling: Exception handling is a mechanism to detect and report an exceptional circumstance so that appropriate action can be taken. These exceptional circumstances are the runtime anomalies (abnormalities) or unusual condition that a program may encounter while executing. These exceptions might include conditions such as division by zero, not being able to open a file, running out of memory space or disk space etc.

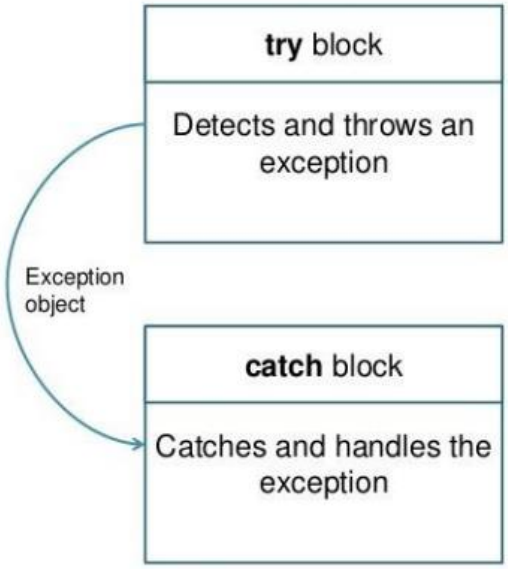
The exception handling mechanism suggests a separate error handling code that performs the following tasks:

1. Find the problem (Hit the exception)
2. Inform the error has occurred (Throw the exception)

By: Manoj Sapkota, U-TEC

3. Receive the error information (Catch the exception)
4. Take the corrective action (Handle the exception)

The error handling code mainly consists of two segments, one to detect error and throw exceptions and other to catch the exceptions and to take appropriate actions.



The mechanism is built upon **three keywords** namely **try, throw** and **catch.** The keyword try is used to preface a block of statements (surrounded by braces) which may generate exception. This block of statement is known as **try block.** When an exception is detected, it is thrown using a **throw statement** in the try block. A **catch block** defined by the keyword catch. This block catches the exception thrown by the throw statement in the try block and handles it appropriately.

General format:

```
try {
  //block of statements which detects and throw an exception
throw exception;
}
catch(type arg)  //catch the exception
{
  // Block of statements that handles the exceptions
}
```

Control flow mechanism:

- When the try block throws an exception, the program control leaves the try block and enters the catch statement of the catch block.
- If the type of object thrown matches the argument (arg) type in the catch statement, then the catch block is executed for handling the exception.
- If they do not match, the program is aborted with the help of abort( ) function which is executed implicitly by the compiler.
- When no exception is detected and thrown, the control goes to the statement immediately after the catch blocks. That is, catch block is skipped.

An example program that demonstrates the exception handling mechanism is shown below. *[Example program- 69]*

Example program- 69:

```
int main()
{
    int a,b,c,d;
    cout<<"Enter values of a and b: "<<endl;
    cin >>a>>b;
    c=a+b;
    d=a-b;
    try
    {
        if(d!=0){
            cout << "Result (c/d)= "<< c/d <<endl;
        }
        else{
            throw(d); //throws an object
        }
    }
    catch(int x) //catches an exception
    {
        cout<< "Exception caught: DIVIDE BY ZERO"<<endl;
    }
    cout << "END";
    return 0;
}
```

Output: First Run

```
Enter values of a and b:
4
2
Result (c/d)= 3
END
```

Second Run

```
Enter values of a and b:
4
4
Exception caught: DIVIDE BY ZERO
END
```

Multiple Catch Statements: A program segment can have more than one condition to throw an exception. In such cases, more than one catch statement can be associated with a try block as shown below.

```
try {
  //try block
}
catch(type1 arg)
{
  //catch block1
}
catch(type2 arg)
{
  //catch block1
}
catch(typeN arg)
{
  //catch blockN
}
```

Control flow Mechanism: When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last catch block for that try. When no match is found, the program is terminated. 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 type is executed.

An example program that demonstrates the exception handling with multiple catch statements is shown below. *[Example program- 70]*

By: Manoj Sapkota, U-TEC

Example program- 70:

```
int main()
{
    int num;
    cout<<"Enter a number: "<<endl;
    cin>>num;
    try
    {
        if(num == 0) throw num;
        else if(num==1) throw float(num);
        else if(num == 2) throw char(num);
        else cout<<"Number: "<<num<<endl;

    }
    catch(int i)
    {
        cout<<"Exception detected: Caught an integer"<<endl;
    }
    catch(float f)
    {
        cout<<"Exception detected: Caught a float"<<endl;
    }
    catch(char c)
    {
        cout<<"Exception detected: Caught a character"<<endl;
    }
    cout<<"End of try catch"<<endl;
    return 0;
}
```

Output: First Run

Enter a number:
5 (entered by user)
Number: 5
End of try catch

Second Run

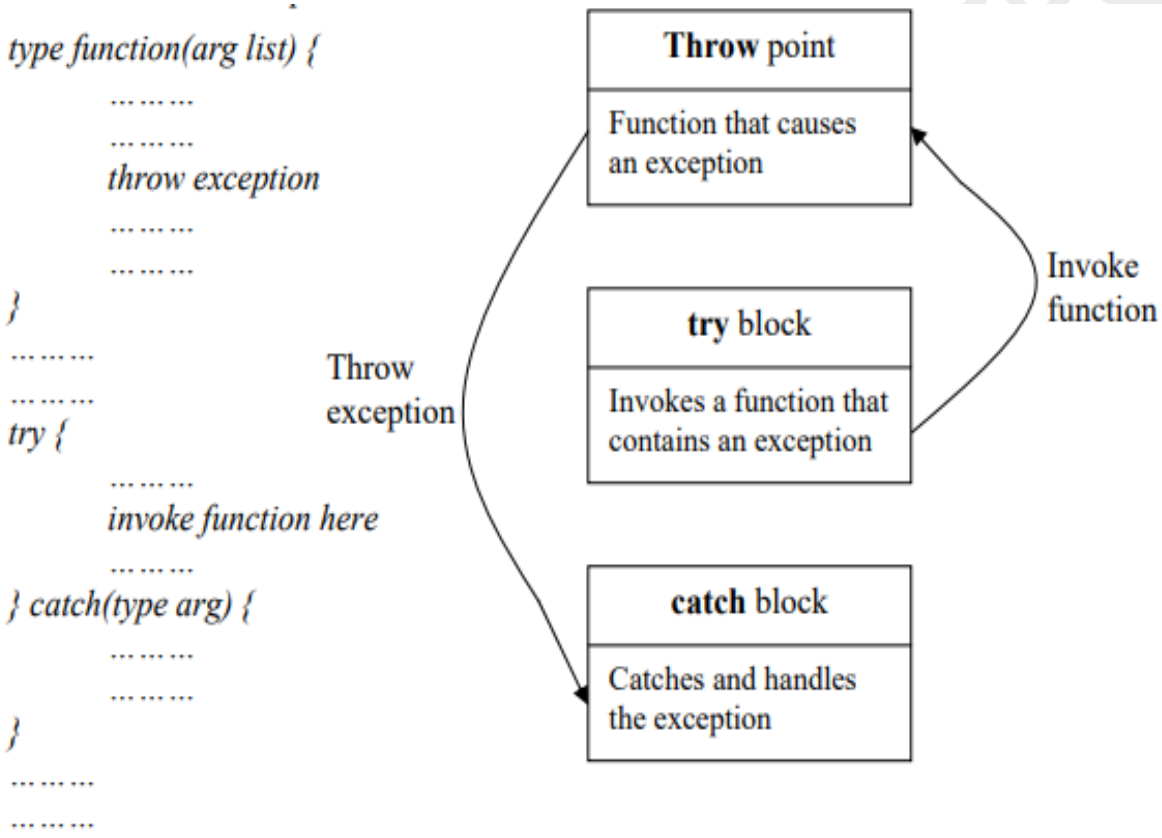
Enter a number:
0 (entered by user)
Exception detected: Caught an integer
End of try catch

Third Run

Enter a number:
2 (entered by user)
Exception detected: Caught a character
End of try catch

Function throwing an exception: Generally, exceptions are thrown by the functions that are invoked from within the try blocks. The point at which the throw is executed is called the throw point. The control cannot return to the throw point once an exception is thrown to the catch block. The catch block must be straight after the try block without any code between them.

The general format and the relationship of this kind of exception handling are shown below.



An example program is shown below that demonstrates the exception handling in which the exception is generated (thrown) by the function that is invoked from within the try block. *[Example program- 71]*

Example program- 71:

```
void divide(int x, int y){
  if(y==0){
    throw y; //throw point
  }
  else{
   float z= (float)x/y;
   cout<<"Result= "<<z<<endl;
  }
}
int main()
{
    int num1, num2;
    cout<<"Enter two numbers: "<<endl;
    cin>>num1>>num2;
    try{
       divide(num1, num2);
    }catch(int a){
      cout<<"Exceptions caught: DIVIDE BY ZERO."<<endl;
    }
    return 0;
}
```

Output:

First Run

Enter two numbers:
5 (entered by user)
2 (entered by user)
Result= 2.5

Second Run

Enter two numbers:
5 (entered by user)
0 (entered by user)
Exceptions caught: DIVIDE BY ZERO.

Catch statement that can catch any type of exception: There is a special catch block called 'catch all' that can be used to catch all types of exceptions. This block handles any type of exception that is thrown in a try block. It is specified by putting '…' in the parenthesis of the catch statement as shown below.

```
try {
   //code
} catch(...) {
  // handles any exception
}
```

Let us consider a following code example. *[Example program- 72]*

Example program- 72:

```
int main()
{
    try
    {
        throw 1.0;
    }
    catch (char c)
    {
        cout << "Caught " <<c;
    }
    catch (...)
    {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output: Default Exception

Explanation of the program: In the above program, a floating point value is thrown as an exception, but there is no catch block for this type. However, catch all block 'catch (...)' catches the thrown object and handles it since it can catch any type of exceptions.

Advantages/Benefits of using exception handling:

- It helps to detect and handle the unusual conditions (runtime anomalies) that a program may encounter while executing.
- It provides a type safe, integrated approach, for coping with the unusual predictable problems that arise while executing a program.

By: Manoj Sapkota, U-TEC

Difference between function overloading and function overriding:

Function overloading	Function overriding
1. It is a feature that allows us to have more than one function having the same name but with different parameter list.	1. It is a feature that allows us to use a function in the derived class that is already present in its base class.
2. The compiler decides which function to call based the parameters passed in the called function.	2. If we call the overridden function using the object of the derived class, the function of the derived class is executed.
3. It occurs within a single program or a single class, so inheritance is not required for overloading.	3. It occurs within a base class and derived class, so inheritance is must for overriding.
4. Overloading is typically resolved at compile time. (early binding)	4. Overriding is resolved at run time. (late binding).
5. It does not create any ambiguity in program.	5. It might create ambiguity in program, and so it should be handled with care while using the overrided function.

6. Program example:
`void sum( int a)`
`{`
`  cout<<a+5<<endl;`
`}`
`void sum(int a, int b){`
`  cout<<a+b<<endl;`
`}`
 	
`int main(){`
`  sum(2);`
`  sum(2,3);`
`  return 0;`
`}`
Output: 7
5
	6. Program example:
	`class A{`
	`  public:`
	`  void fun(){`
	`  cout<<"Hello!"<<endl;`
	`  }`
	`};`
	`class B: public A{`
	`  public:`
	`  void fun(){`
	`  cout<<"HI!"<<endl;`
	`  }`
	`};`
	
	`int main(){`
	`  B obj;`
	`  obj.fun();`
	`  return 0;`
	`}`
	Output: HI!

Difference between Structure and Class:

One of the main differences between structure and class in C++ is hiding the implementation details. A class hides all its implementation details from the outside world by default. However, a structure will not hide its implementation details from the outside world by default. So, the class strictly follows the principle of encapsulation and data hiding while structure does not.

Class	Structure
1. It is a collection of related variables and functions contained within a single construct.	1. It is a grouping of variables of various data types referenced by the same name.
2. The members of the class are private by default.	2. The members of the structure are public by default.
3. It is defined by the keyword 'class'.	3. It is defined by the keyword 'struct'.
4. It may have all the types of constructors and destructors.	4. It may have only parameterized constructor.
5. It supports all the object oriented programming features.	5. It doesn't support all the object oriented programming features.
6. It is used for huge amount of data.	6. It is used for smaller amount of data.

7. Basic example:
class A{
int a;
public:
void getData(int x){
a=x;
}
void display( ){
cout<<a<<endl;
}
};
int main( )
{
A obj;
obj.getData(5);
obj.display( );
return 0;
}
	7. Basic example:
	struct area{
	int b, h;
	float a;
	};
	int main( )
	{
	Area s1;
	s1.b=2;
	s1.h=3;
	s1.a=0.5*s1.b*s1.h;
	cout<<"Area: "<<s1.a<<endl;
	return 0;
	}

By: Manoj Sapkota, UTC

Creating array of objects: If we want to initialize data members with multiple objects, then we can create an array of objects.

Example program- 73:

```
class Random{
  int num;
  public:
       void getData( ){
         cout<<"Enter the number: "<<endl;
         cin>>num;
       }
       void display(){
        cout<<"Value of num: "<<num<<endl;
       }
};
int main(){
  Random obj[5]; /** Creating an array to store 5 objects **/

  for (int i=0; i<5; i++){
     obj[i].getData();  /** Initializing data member for each object **/
  }

  for (int i=0; i<5; i++){
    obj[i].display(); /** Displaying the information of each object **/
  }
  return 0;
}
```

Output:

Enter the number:
4 (entered by user)
Enter the number:
3 (entered by user)
Enter the number:
2 (entered by user)
Enter the number:
6 (entered by user)
Enter the number:
1 (entered by user)
Value of num: 4
Value of num: 3
Value of num: 2
Value of num: 6
Value of num: 1

Dynamic initialization of object: Initializing the object at run-time, i.e. providing initial values to an object during run time is known as dynamic initialization of object.

This means, initializing the data members of the class while creating an object. So, dynamic initialization of objects is used when we want to provide initial or default values to the data members while creating an object. **It can be achieved by using parameterized constructors**.

Why is the need of dynamic objects ?

- It utilizes memory efficiently.
- It is useful when there are multiple constructors with different inputs of the same class.
- It provides the flexibility of using different formats of data at run-time.

Example program- 74:

```
class Rectangle{
  int length,breadth,area;
  public:
       Rectangle(){

       }
       Rectangle(int l, int b){
        length=l;
        breadth=b;
       }
       void display(){
        area=l*b;
        cout<<"Area: "<<area<<endl;
       }
};
int main(){
  int l,b;
  cout<<"Enter the value of length:"<<endl;
  cin>>l;
  cout<<"Enter the value of breadth:"<<endl;
  cin>>b;
  Rectangle r(l, b); /**Dynamic Initialization of object**/
  r.display();
  return 0;
}
```

Dynamic constructor: The constructor that is used for allocating the memory at runtime is known as dynamic constructor. The memory is allocated at runtime using a new operator. A class should have a pointer variable as a data member and this pointer is used to allocate memory dynamically at run-time inside the dynamic constructor.

Example program- 75:

```
class Utech
{
    int *ptr;
    public:
    Utech( ) /** Dynamic Constructor **/
    {
        ptr= new int;
        cout<<"Memory allocated dynamically."<<endl;
    }
    ~Utech(){
        delete ptr;
        cout<<"Memory released"<<endl;
    }
};

int main()
{
    Utech obj;
    return 0;
}
```

Output: Memory allocated dynamically
Memory released

Containership/Composition:
When a class contains object of another class as its member data, it is termed as containership. The class which contains the object is called **container class**. Containership is also termed as "class within class". It is also known as composition. In containership/composition, a class has another class so it exhibits a 'has-a' relationship.

Containership is a type of association that implies ownership. In this type, a parent class owns child entity so the child entity cannot exist without parent entity. The child entity cannot be independently accessed.

For example, school is made of classrooms. If we represent school and classroom as classes, then it is suitable to include the classroom class inside the school class.

General format/ syntax:

```
class Classroom {
 …
};
class School {
 …
 Classroom obj1;
 …
};
```

Here, class 'School' contains object of class 'Classroom'. So, in this case, 'School' is the container class.

Example program- 76:

```
class A{
 int x;
 public:
     void getData(){
       cout<<"Enter the value of x: "<<endl;
       cin>>x;
     }
     void display(){
      cout<<"Value of x: "<<x<<endl;
     }
};

class B{
  int y;
  A obj1;
  public:
      void getData(){
       obj1.getData();
       cout<<"Enter the value of y:"<<endl;
       cin>>y;
      }
      void display(){
       obj1.display();
       cout<<"The value of y: "<<y<<endl;
      }
};

int main()
{
   B obj2;
   obj2.getData();
   obj2.display();
   return 0;
}
```

Output:
Enter the value of x:
3 (entered by user)
Enter the value of y:
5 (entered by user)
Value of x: 3
The value of y: 5

Difference between inheritance and container class:

Inheritance	**Containership/Container class**
1. In inheritance, the child classes inherit the properties (data members, methods) from their parent classes.	1. A container class contains the object of another class as its data member.
2. It exhibits a 'is-a' relationship.	2. It exhibits a 'has-a' relationship.
3. It allows polymorphism.	3. It doesn't allow polymorphism.
4. For e.g. Student is a person, where student class can be inherited from the person class.	4. For e.g. Car has an engine, where car class contains the engine class.

5. Program Format/Syntax:
class Person{
 	
};
class Student : public Person{
 	
};
	5. Program Format/Syntax:
	class Engine{
	
	};
	class Car{
	Engine e;
	};

6. Inheritance leads tight coupling between superclass and subclass. So, it is harder to maintain in future.	6. In composition, relationship between classes can be decoupled easily. So, it can be easily maintained in the future.
7. Program example: *[Example program- 77]*	7. Program example: *[Example program- 76]*

Aggregation: It is similar to containership in which a class contains object of another class. In aggregation, parent and child entity maintain a 'has-a' relationship but both entities can exist independently. Any modification in the parent class will not impact the child class or vice-versa.

For example, classroom has students. If we represent classroom and student as classes, then the student class can be included inside the class classroom. Then, a classroom and student will be linked with 'has-a' relationship. But here, both the classroom and student class can be made to exist independently.

Example program- 77: (Basic Inheritance Example)

```
class A{
 int a;
   public:
        void getA(int x){
         a=x;
        }
        void display(){
          cout<<"Value of a: "<<a<<endl;
        }

};
class B: public A{
 int b;

 public:
     void getB(int x, int y){
       getA(x);
       b=y;
     }
     void display(){
       A::display();
       cout<<"Value of b: "<<b<<endl;
     }
};

int main(){
   B obj;
   obj.getB(5,7);
   obj.display();
   return 0;
}
```

Output: Value of a: 5
Value of b: 7

Differences between is-a relationship and has-a relationship:

Is-a relationship	Has-a relationship
1. 'Is a' relationship refers to inheritance.	1. 'Has-a' relationship refers to aggregation or composition.
2. In a 'Is-a' relationship', the instance of the child class is a more specialized form of the parent class, and can inherit the properties from them.	2. In a 'Has-a' relationship, a class contains the object of another class as its data member.
3. A real-world example of it is: Student is a person. Here, Student class can be derived from Person class, and holds a 'is-a' relationship.	3. A real-world example of it is: Library has books. Here, Library class can contain the object of Book class, and holds a 'has-a' relationship.

4. Program format/syntax:
class Person{
 	
};
class Student : public Person{
 	
};
	4. Program format/syntax:
	class Book{
	
	};
	class Library{
	Book b;
	};

5. It allows polymorphism.	5. It doesn't allow polymorphism.
6. Program example: *[Example program- 77]*	6. Program example: *[Example program- 76]*

By: Manoj Sapkota, UTEC