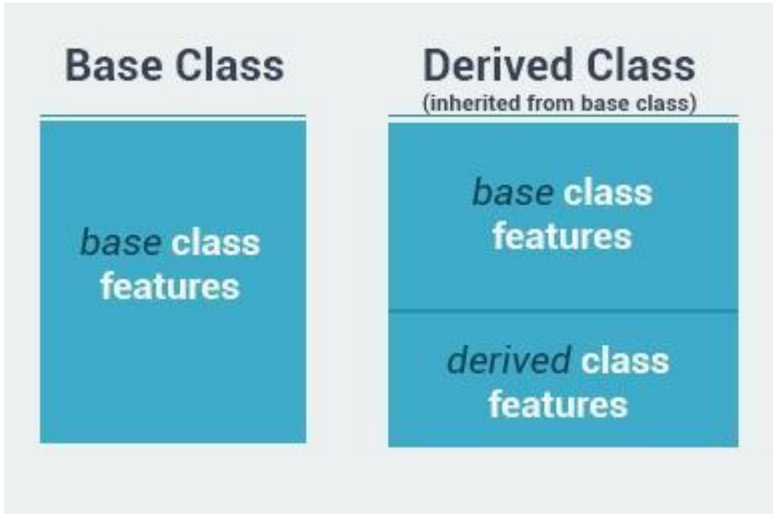
REPORT JUNE 26

Date:	26th JUNE 2020	Name:	Sushant poojary
Course:	Solo Learning	USN:	4AL18EC400
Topic:	Inheritance,Templates,Exceptions and Files	Semester & Section:	6th SEM B
Github Repository :	Sushant7026		

FORENOON SESSION DETAILS
Image of session 
Report – The class whose properties are inherited by another class is called the Base class. The class which inherits the properties is called the Derived class. For example, the Daughter class (derived) can be inherited from the Mother class (base). The derived class inherits all feature from the base class, and can have its own additional features.



To further explain this behavior, let's create a sample class that includes a constructor and a destructor:

```
class Mother {
public:
Mother()
{
 cout <<"Mother ctor"<<endl;
}
~Mother()
{
 cout <<"Mother dtor"<<endl;
}
};
```

Inheritance

Next, let's create a Daughter class, with its own constructor and destructor, and make it a derived class of the Mother:

```
class Daughter: public Mother {
public:
Daughter()
{
 cout <<"Daughter ctor"<<endl;
}
~Daughter()
{
 cout <<"Daughter dtor"<<endl;
}
};
```

Polymorphism

The word polymorphism means "having many forms".

Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different implementation to be executed depending on the type of object that invokes the function.

Polymorphism can be demonstrated more clearly using an example:
Suppose you want to make a simple game, which includes different enemies: monsters, ninjas, etc. All enemies have one function in common: an attack function. However, they each attack in a different way. In this situation, polymorphism allows for calling the same attack function on different objects, but resulting in different behaviors.

The first step is to create the Enemy class.

```
class Enemy {
protected:
 int attackPower;
public:
 void setAttackPower(int a){
  attackPower = a;
 }
};
```

Polymorphism

Our second step is to create classes for two different types of enemies, Ninjas and Monsters. Both of these new classes inherit from the Enemy class, so each has an attack power. At the same time, each has a specific attack function.

```
class Ninja: public Enemy {
public:
 void attack() {
  cout << "Ninja! - "<<attackPower<<endl;
 }
};

class Monster: public Enemy {
public:
 void attack() {
  cout << "Monster! - "<<attackPower<<endl;
 }
};


class Enemy {
public:
 virtual void attack() {
 }
};
class Ninja: public Enemy {
public:
 void attack() {
```

```
  cout << "Ninja!"<<endl;
 }
};
class Monster: public Enemy {
public:
 void attack() {
  cout << "Monster!"<<endl;
}
};
```

Pure Virtual Functions

The pure virtual function in the Enemy class must be overridden in its derived classes.

```
class Enemy {
public:
 virtual void attack() = 0;
};

class Ninja: public Enemy {
public:
 void attack() {
  cout << "Ninja!"<<endl;
 }
};
class Monster: public Enemy {
public:
 void attack() {
  cout << "Monster!"<<endl;
 }
};
```

Function Templates

Functions and classes help to make programs easier to write, safer, and more maintainable. However, while functions and classes do have all of those advantages, in certain cases they can also be somewhat limited by C++'s requirement that you specify types for all of your parameters.

For example, you might want to write a function that calculates the sum of two numbers, similar to this:

```
int sum(int a, int b) {
 return a+b;
}
```

Function Templates

We can now call the function for two integers in our main.

```
int sum(int a, int b) {
 return a+b;
}
int main () {
 int x=7, y=15;
```

```
 cout << sum(x, y) << endl;
}
// Outputs 22
```

It becomes necessary to write a new function for each new type, such as doubles.

```
double sum(double a, double b) {

 return a+b;

}
```

Wouldn't it be much more efficient to be able to write one version of sum() to work with parameters of any type?
Function templates give us the ability to do that!
With function templates, the basic idea is to avoid the necessity of specifying an exact type for each variable. Instead, C++ provides us with the capability of defining functions using placeholder types, called template type parameters.

To define a function template, use the keyword template, followed by the template type definition:

```
template <class T>
```

Now we can use our generic data type T in the function:

```
template <class T>
T sum(T a, T b) {
 return a+b;
}

int main () {
  int x=7, y=15;
  cout << sum(x, y) << endl;
}
```

Template functions can save a lot of time, because they are written only once, and work with different types.

Template functions reduce code maintenance, because duplicate code is reduced significantly.

In our main, we can use the function for different data types:

```
template <class T, class U>
T smaller(T a, U b) {
 return (a < b ? a : b);
}
int main () {
 int x=72;
 double y=15.34;
```

```
 cout << smaller(x, y) << endl;
}
```

Class Templates

Just as we can define function templates, we can also define class templates, allowing classes to have members that use template parameters as types.

The same syntax is used to define the class template:

```
template <class T>

class MyClass {

};
```

As an example, let's create a class Pair, that will be holding a pair of values of a generic type.

```
template <class T>
class Pair {
private:
 T first, second;
public:
 Pair (T a, T b):
  first(a), second(b) {
 }
};
```

A specific syntax is required in case you define your member functions outside of your class - for example in a separate source file.

You need to specify the generic type in angle brackets after the class name.

For example, to have a member function bigger() defined outside of the class, the following syntax is used:

```
template <class T>
class Pair {
private:
 T first, second;
public:
 Pair (T a, T b):
  first(a), second(b){
 }
 T bigger();
};
template <class T>
T Pair<T>::bigger() {
```

```
    // some code
    }
    template <class T>
    class MyClass {
    public:
     MyClass (T x) {
      cout <<x<<" -  not a char"<<endl;
     }
    };
template <class T>
class MyClass {

public:

 MyClass (T x) {

  cout <<x<<" -  not a char"<<endl;

 }

};

template < >

class MyClass<char> {

public:

 MyClass (char x) {

  cout <<x<<" is a char!"<<endl;

 }

};
```

Exceptions

Problems that occur during program execution are called exceptions.

In C++ exceptions are responses to anomalies that arise while the program is running, such as an attempt to divide by zero.

Working with Files

To perform file processing in C++, header files <iostream> and <fstream> must be included in the C++ source file.

```
#include <iostream>

#include <fstream>
```

File Opening Modes

An optional second parameter of the open function defines the mode in which the file is opened. This list shows the supported modes.

Mode Parameter	Meaning
ios::**app**	append to end of file
ios::**ate**	go to end of file on opening
ios::**binary**	file open in binary mode
ios::**in**	open file for reading only
ios::**out**	open file for writing only
ios::**trunc**	delete the contents of the file if it exists

All these flags can be combined using the bitwise operator OR (|).

For example, to open a file in write mode and truncate it, in case it already exists, use the following syntax:

```
ofstream outfile;

outfile.open("file.dat", ios::out | ios::trunc );
```

Reading from a File

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

int main () {
```

```
string line;
ifstream MyFile("test.txt");
while ( getline (MyFile, line) ) {
 cout << line << '\n';
}
MyFile.close();
}
```