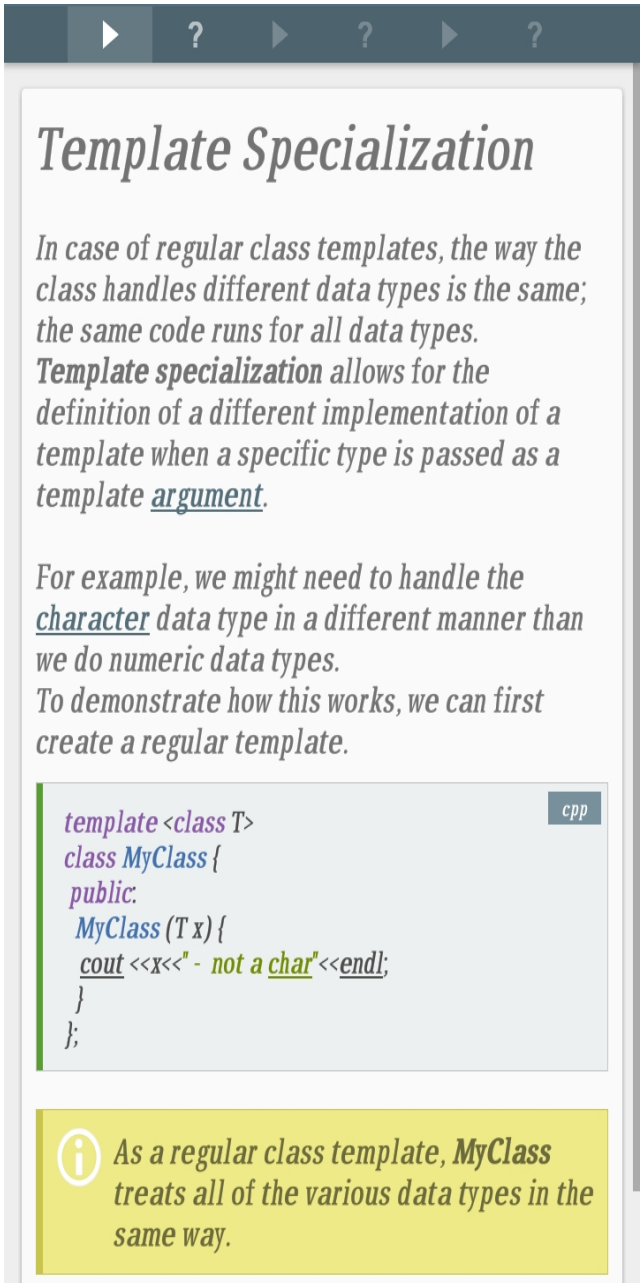
DAILY ASSESSMENT FORMAT

Date:	25th June 2020	**Name:**	Soundarya NA
Course:	C++ programming	**USN:**	4AL16EC077
Topic:	Inheritance and Polymorphism Templates, Exceptions and Files	**Semester & Section:**	8th - B

FORENOON SESSION DETAILS

Image of session



Template Specialization

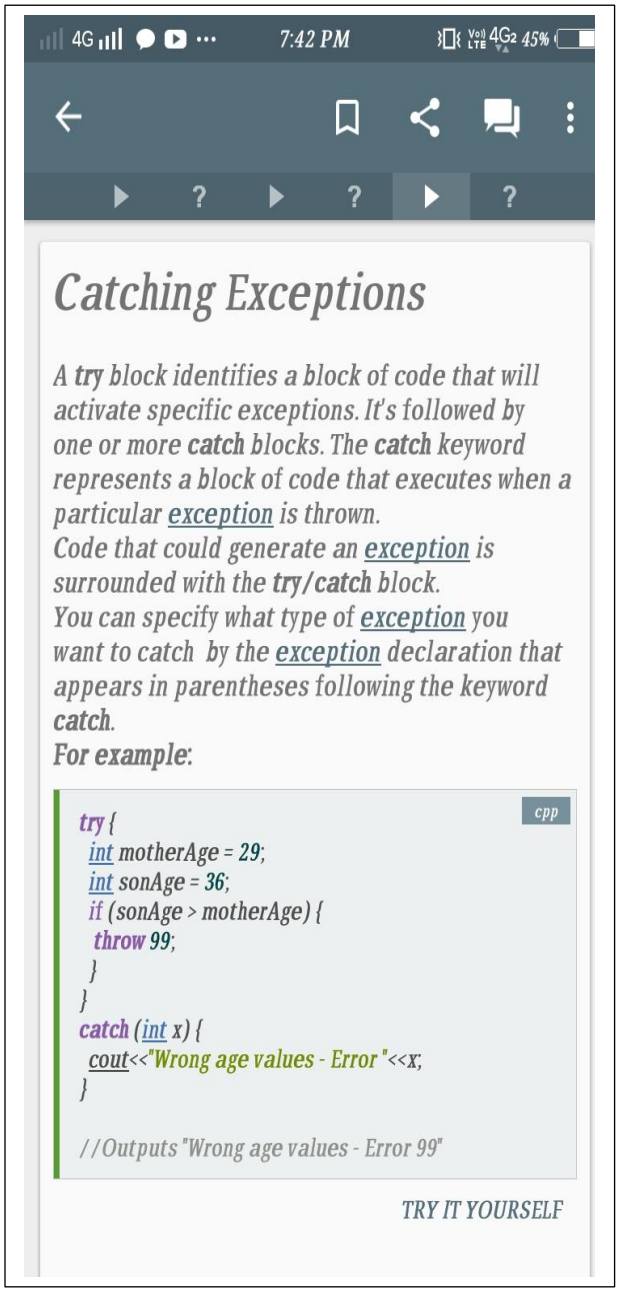
In case of regular class templates, the way the class handles different data types is the same; the same code runs for all data types.
***Template specialization** allows for the definition of a different implementation of a template when a specific type is passed as a template argument.*

For example, we might need to handle the character data type in a different manner than we do numeric data types.
To demonstrate how this works, we can first create a regular template.

cpp

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

*As a regular class template, **MyClass** treats all of the various data types in the same way.*



Catching Exceptions

*A **try** block identifies a block of code that will activate specific exceptions. It's followed by one or more **catch** blocks. The **catch** keyword represents a block of code that executes when a particular exception is thrown.*
*Code that could generate an exception is surrounded with the **try/catch** block.*
*You can specify what type of exception you want to catch by the exception declaration that appears in parentheses following the keyword **catch**.*
For example:

cpp

```
try {
  int motherAge = 29;
  int sonAge = 36;
  if (sonAge > motherAge) {
   throw 99;
  }
}
catch (int x) {
  cout<<"Wrong age values - Error "<<x;
}

//Outputs "Wrong age values - Error 99"
```

TRY IT YOURSELF

Report:

Polymorphism:

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 function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

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

class Shape {
  protected:
    int width, height;

  public:
    Shape( int a = 0, int b = 0){
      width = a;
      height = b;
    }
    int area() {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
```

```
};

class Triangle: public Shape {
  public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};

// Main function for the program
int main() {
  Shape *shape;
  Rectangle rec(10,7);
  Triangle  tri(10,5);

  // store the address of Rectangle
  shape = &rec;

  // call rectangle area.
  shape->area();

  // store the address of Triangle
  shape = &tri;

  // call triangle area.
  shape->area();
```

```
  return 0;
}
```

Output:

```
Parent class area:
Parent class area:
```

Code:

```
class Shape {
  protected:
    int width, height;

  public:
    Shape( int a = 0, int b = 0) {
      width = a;
      height = b;
    }
    virtual int area() {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
```

Output:

```
Rectangle class area
Triangle class area
```

Virtual Function:

A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or late binding.

Pure virtual function:

Code:

```
class Shape {
  protected:
    int width, height;

  public:
    Shape(int a = 0, int b = 0) {
      width = a;
      height = b;
    }

    // pure virtual function
    virtual int area() = 0;
};
```

Inheritance:

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.

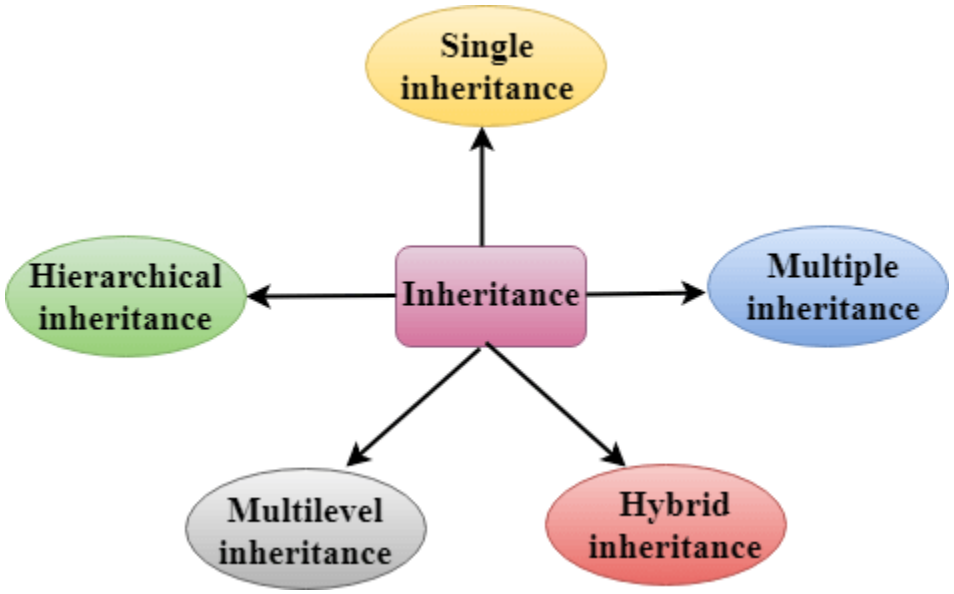
In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class.

Advantages of C++ Inheritance:

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So, less code is required in the class.

Types of Inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



1. **Single Inheritance:**

Code:

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
    Programmer p1;
```

```
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
   return 0;
}
```

Output:

Salary: 60000

Bonus: 5000

Code:

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
      public:
     void bark(){
    cout<<"Barking...";
     }
   };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

Output:

Eating...

Barking...

Code:

```
#include <iostream>
using namespace std;
class A
{
   int a = 4;
   int b = 5;
   public:
   int mul()
   {
      int c = a*b;
      return c;
   }
};

class B : private A
{
   public:
   void display()
   {
      int result = mul();
      std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
   }
};
int main()
{
```

```
  B b;
  b.display();


   return 0;
}
```

Output:

Multiplication of a and b is : 20

Multi-Level Inheritance:

Code:

```
#include <iostream>
using namespace std;
 class Animal {
  public:
 void eat() {
   cout<<"Eating..."<<endl;
 }
  };
  class Dog: public Animal
  {
     public:
    void bark(){
   cout<<"Barking..."<<endl;
    }
  };
  class BabyDog: public Dog
  {
     public:
    void weep() {
   cout<<"Weeping...";
```

```
    }
  };
int main(void) {
   BabyDog d1;
   d1.eat();
   d1.bark();
    d1.weep();
    return 0;
}
```

Output:

```
Eating...
Barking...
Weeping...
```

Multiple Inheritance:

```
#include <iostream>
using namespace std;
class A
{
   protected:
    int a;
   public:
   void get_a(int n)
   {
      a = n;
   }
};

class B
{
```

```
    protected:
    int b;
    public:
    void get_b(int n)
    {
        b = n;
    }
};
class C : public A,public B
{
  public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
```

Addition of a and b is : 30

Hybrid Inheritance:

Code:

```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
     std::cout << "Enter the value of 'a' : " << std::endl;
     cin>>a;
   }
};

class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
      std::cout << "Enter the value of 'b' : " << std::endl;
     cin>>b;
   }
};
class C
```

```
{
   protected:
   int c;
   public:
   void get_c()
   {
      std::cout << "Enter the value of c is : " << std::endl;
      cin>>c;
   }
};

class D : public B, public C
{
   protected:
   int d;
   public:
   void mul()
   {
      get_a();
      get_b();
      get_c();
      std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
   }
};
int main()
{
   D d;
   d.mul();
   return 0;
}
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

Hierarchical Inheritance:

```
#include <iostream>
using namespace std;
class Shape              // Declaration of base class.
{
   public:
   int a;
   int b;
   void get_data(int n,int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape  // inheriting Shape class
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
```

```
    }
};
class Triangle : public Shape    // inheriting Shape class
{
    public:
    int triangle_area()
    {
        float result = 0.5*a*b;
        return result;
    }
};
int main()
{
    Rectangle r;
    Triangle t;
    int length,breadth,base,height;
    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
    cin>>length>>breadth;
    r.get_data(length,breadth);
    int m = r.rect_area();
    std::cout << "Area of the rectangle is : " <<m<< std::endl;
    std::cout << "Enter the base and height of the triangle: " << std::endl;
    cin>>base>>height;
    t.get_data(base,height);
    float n = t.triangle_area();
    std::cout <<"Area of the triangle is : "  << n<<std::endl;
    return 0;
}
```

Output:

```
Enter the length and breadth of a rectangle:
```

23

20

Area of the rectangle is : 460

Enter the base and height of the triangle:

2

5

Area of the triangle is : 5

Exception:

- Some code (e.g. a library module) may detect an error but not know what to do about it; other code (e.g. a user module) may know how to handle it
- C++ provides exceptions to allow an error to be communicated
- In C++ terminology, one portion of code throws an exception; another portion catches it
- If an exception is thrown, the call stack is unwound until a function is found which catches the exception
- If an exception is not caught, the program terminates

Templates:

- Templates support meta-programming, where code can be evaluated at compile-time rather than run-time
- Templates support generic programming by allowing types to be parameters in a program
- Generic programming means we can write one set of algorithms and one set of data structures to work with objects of any type
- We can achieve some of this flexibility in C, by casting everything to void * (e.g. sort routine presented earlier)
- The C++ Standard Template Library (STL) makes extensive use of templates

Files:

Another useful C++ feature is the ability to read and write to files. That requires the standard C++ library called fstream.

Three new data types are defined in fstream:

- **ofstream:** Output file stream that creates and writes information to files.
- **ifstream:** Input file stream that reads information from files.
- **fstream:** General file stream, with both ofstream and ifstream capabilities that allow it to create, read, and write information to files.

To perform file processing in C++, **header files and ** must be included in the C++ source file.

```
#include <iostream>
#include <fstream>
```

Opening a File:

Code:

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

int main() {
  ofstream MyFile;
  MyFile.open("test.txt");

  MyFile << "Some text. \n";
}
```

Closing a File:

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

int main() {
  ofstream MyFile;
  MyFile.open("test.txt");
```

```
  MyFile << "Some text! \n";
  MyFile.close();
}
```

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();
}
```