UNIT-4

TEMPLATE

- It is one of the features of C++
- It is a new concept which enable us to define generic classes and functions
- Template provide support for generic programming.
- Generic programming is an approach where generic types are used as parameters.
- A template can be used to create a family of classes or functions.

Template

Class T function T

class Template

* It is a technique where a prototype of class is created.

Eg: class template for an array class would enable us to create arrays of various data types.

* A template can be considered as a kind of macro.

Function Template

It is a technique of creating a prototype of function which is not specific to any data type.

Function template can be used to create a family of functions with different arguments

Function template syntax is similar to that of a class template except we are defining functions instead of class.

Syntax :

Class Template

```
Template < class T>
class class_name
{

    // class member specifed
    // with type

}
```

Function Template

```
Template < class T>
return Type function_name (arg with Type)
{

}
```

<Template T>

```
template <class T>
T large (Tn1, Tn2)
{
    return (n1>n2) ? n1 : n2;
}
int main ()
{
    int x,y;
    float a,b;
    char p,q;
cout << "Enter 2 integer no";
 cin >> x >> y;
cout << "The large :";
large (x,y);
cout << "float"
cin >> a >> b;
cout << large (a,b);
cin >> p >> q
cout << "char;";
cout << large (p,q);
 getch();
}
```

19/8/19.

FUNCTION OVERLOADING

```
#include <iostream.h>
#include <conio.h>
class abc
{
   int a;
  void enter (void)
  {  int rollno;
     cout << " Enter the details ";
     cin >> rollno;
  }

  void enter (int b, int c)
  {
     cout << " Enter the id and marks ";
     cin >> b >> c;
  }

};
void main ()
{
    abc a1;
  a1. enter ();
  a1. enter (5, 60);
  getch();
}
```

```
#include < iostream.h>
#include < conio.h>
class xyz
{
  public :
int func ( int x , int y)
{
  return (x-y);
}
  int func ( float a, float b)
                    float c
{
  return ( a+b+c);
}
  void main
  {
    xyz s;
cout << " The result of x and y is " << s.func(4,3);
cout << " The result of a and b is " << s.func(4.0, 3.0)
                                                  7.0.
  getch();
}
```


Usee of function overloading

1. It is done for code reusability, to save efforts and save memory.
2. eliminates the use of different function names for the same operation
3. overloaded methods give programmers the flexibility to call a similar method for different types of data ....

Page No.
Date

INHERITANCE

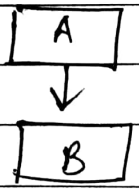
Inheritance is a feature of OOPs (Object oriented programming system) which allows the creation of a new class with derivation of its (base) own features and allows child class to have its own unique features

This is basically done by creating new classes and reusing the properties of the existing one.

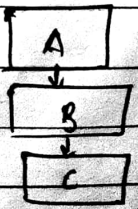
The mechanism of deriving a new class from the old one is known as Inheritance.

Types of Inheritance

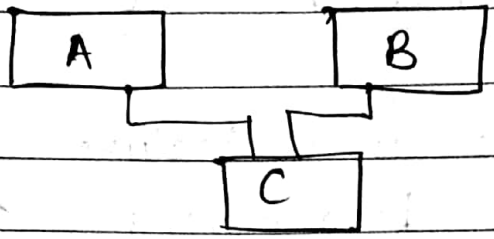
1) Single level Inheritance



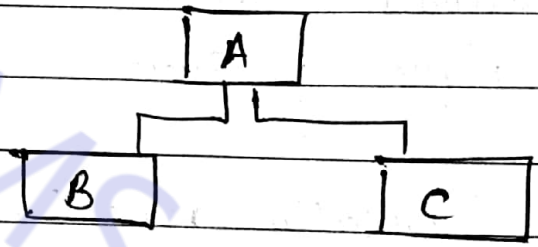
2) Multi-level Inheritance



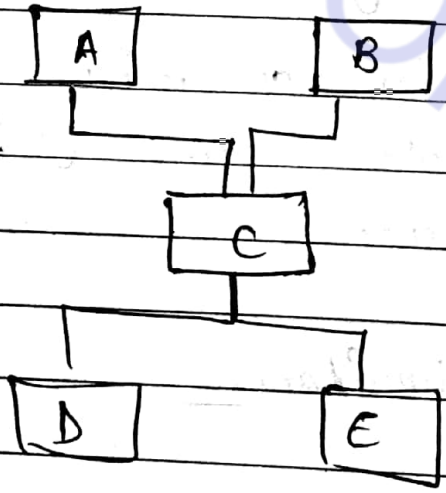
3) Multiple Inheritance



(4) Hierarchical Inheritance



(5) Hybrid Inheritance

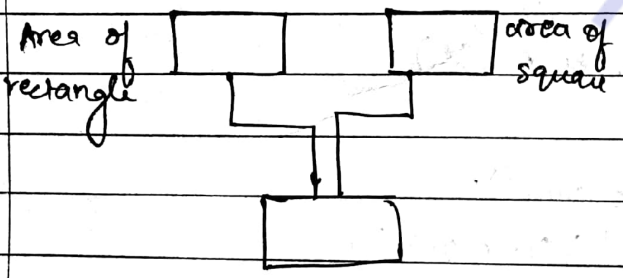


MULTI LEVEL INHERITANCE PROGRAM EXAMPLE

```
class A
{
  public:
  int roll;
  char name;
void get_name(void)
{
  cout << "Enter name";
  cin >> name;
  cout << "Enter rollno";
  cin >> rollno;
}
};
class B : public A
{
  public:
int m1,m2,m3;
void marks(void)
{
  get_name();
  cout << "Enter marks";
  cin >> m1 >> m2 >> m3;
}
};
class C : public B
{
  public:
  int sum;
void display(void
{
```

```
    marks();
    sum = (m1 + m2 + m3);
    cout << "The name is" << name;
    cout << "The rollno is" << rollno;
    cout << "The marks are" << m1 << m2 << m3;
    cout << "The sum is" << sum;
    }
};
  int main()
{
    C obj;
    obj.display();
    getch();
    return 0;
}
```

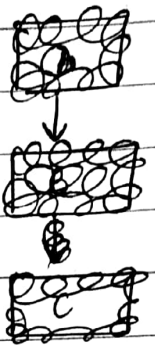
MULTIPLE INHERITANCE

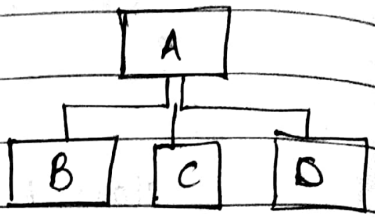



```
class A
{
    public:
    int length;
    int breadth
void get_data(void)
{
    cout << "Enter length";
```

```
    cin >> length;
    cout << " Enter breadth "
    cin >> breadth ;
  }
};
  class B {
  {
    public :
  int side ;
  void area(void)
  {
   cout << "Enter the side of square ";
   cin >> side;
  }
};
  class C : public b, public A
  {
    public :
   int result;
   void display (void)
  {
     area();
     get_data();                                  length * breadth;
  cout << " The area of rectangle is " << result;
  cout << " The area of square is " << side* side;
  }
};
   int main()
  {
     C obj;
   obj. display();
     getch();
} return 0;
```

HIERARCHICAL INHERITANCE






```
class A
{
   public :
    int l, b, side, base, height;

void get_data (void)
{
   cout << "Enter the values";
   cin >> l;
   cin >> b;
   cin >> side;
   cin >> base;
   cin >> height;
}
};

class B : public A
{
 void calculate (void)
 {    get_data ();
      int area;

   area = l * b;
```

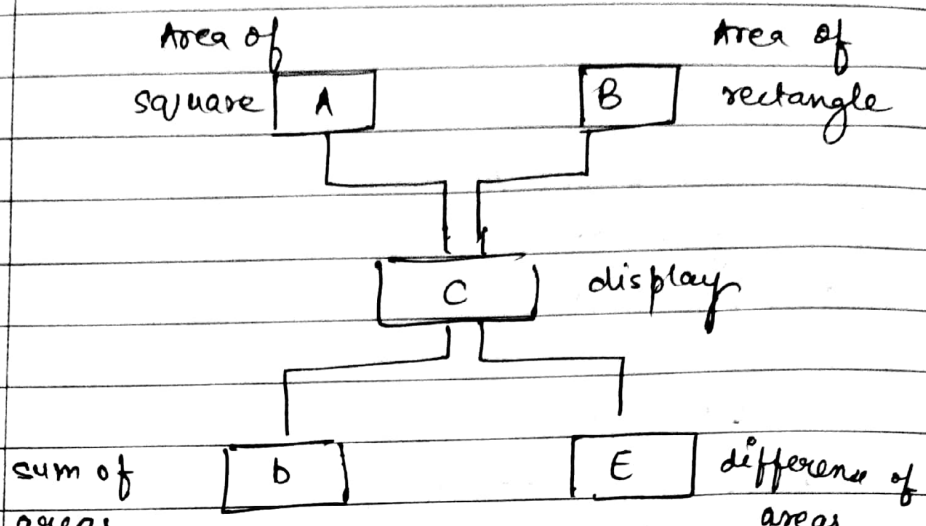
```
    cout << " The area of rectangle is :" << area1;
  }
};

class C : public A
{     get_data();
  void area (void)
  {
     int area2;
  area2 = side * side;
  cout << " The area of square is :" << area2;
  }
};
class D : public A.
{

  void triarea (void)
  {
     int area3;
  area3 = 1/2 * base * height;
 cout << " The area of triangle is :" << area3;
 }
};
 int main ()
{

     D obj1;
     C obj2;
     B obj3;
     obj1. triarea ();
     obj2. area ();
     obj3. calculate ();
     return 0;
}
```

5. # HYBRID INHERITANCE




```
class A
{   public:
    int length;
    int breadth;

void calculate (void)
{
    int area1;
cout << " Enter the value of length and breath.";
cin >> length >> breadth;
area1 = length * breadth;
}
};

class B
{   public:
    int side;
void area (void)
{   int area2;
```

Calculate ()

```
cout << " Enter the side of square ";
cin >> side;
area2 = side * side;
}
};

class C : public A, public B
{ public:
void display(void)
{ calculate();
  area();
cout << "The area of rectangle" << area1;
cout << " The area of squared " << area2;
}
};

class D : public C
{ public:
  void sum(void)
  { int sum;
    display();
  sum = area1 + area2;
  cout << " The sum of area is " << sum;
  }
};
class E : public C
{ public:
  void diff(void)
  { int diff;
    display();
diff = area1 - area2;
```

```
cout << " The difference is " << diff ;
 }
};
int main ()
{
      E obj;
      D obj;
      C obj;
    E . diff ();
   D . sum ();
   C . display ();
  return 0;
}
```

UNIT-IV

Generic function

Template function, function name overloading, Overriding inheritance methods, Run time polymorphism, Multiple Inheritance.

Templates (Functions)

Templates ?

- **Templates** are a feature of the C++ programming language that allow functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

Types of templates ?

- C++ provides two kinds of templates:
  - Class templates and
  - Function templates.

Function Template?

- Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type. In C++ this can be achieved using template parameters.

What is template parameter ?

- A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass values and also types to a function.

Template Instantiation

- When the compiler generates a class, function or static data members from a template, it is referred to as template instantiation.
  - A function generated from a function template is called a generated function.

From Compiler's point of view…

- Templates are not normal functions or classes. At that moment, when an instantiation is required, the compiler generates a function specifically for those arguments from the template.

```
template <class myType>
myType GetMax (myType a, myType b)
{
  return (a>b?a:b);
}
```

Template function with two arguments of same type.

```
template <class T, class U>
 T GetMin (T a, U b)
{
   return (a<b?a:b);
}
```

Template function with two arguments of different type or same type. It depends on the argument passed.

More…

We can also overload a Function Template as well as Override a Function Template.

Overloading and Overriding can be achieved through Functions as well as Template Functions.

Example-Overloading

```
#include<iostream.h>
#include<conio.h>
template <class t>
void max(t a,t b)
{
   if(a>b)
         cout<<a;
   else
         cout<<b;
}
```

```
template <class t>
void max(t a,t b,t c)
{
   if(a>b&&a>c)
         cout<<a;
   else if(b>a&&b>c)
         cout<<b;
   else
         cout<<c;
}
```

```
void main()
{
   clrscr();
   max(1,2);
   max(3,2,1);
   getch();
}
```

Example-Overriding

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
template <class T>
void sorting(T a[],int n)
{
    T temp;
    int i,j;
    for(i=0;i<n;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(a[i]>a[j])
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }
    }
}
```

```
void sorting(char a[10][10],int n)
{
    char temp[10];
    int i,j;
    for(i=0;i<n;i++)
    {
            for(j=i+1;j<n;j++)
            {
                    if(strcmp(a[i],a[j])>0)
                    {
                            strcpy(temp,a[i]);
                            strcpy(a[i],a[j]);
                            strcpy(a[j],temp);
                    }
            }
    }
}
```

```
template <class T>
void print(T a[],int n)
{
    int i;
    cout<<"\nSorted List\n";
    for(i=0;i<n;i++)
    {
            cout<<a[i]<<"\n";
    }
}
void print(char a[10][10],int n)
{
    int i;
    cout<<"\nSorted List\n";
    for(i=0;i<n;i++)
    {
            cout<<a[i]<<"\n";
    }
}
```

```
template <class T>
void get(T a[],int n)
{
    int i;
    for(i=0;i<n;i++)
    {
            cin>>a[i];
    }
}
void get(char a[10][10],int n)
{
    int i;
    for(i=0;i<n;i++)
    {
            cin>>a[i];
    }
}
```

```
void main()
{
   clrscr();
   int i=1,a[20],size;
   float b[10];
   char c[10],d[10][10];
   while(i!=5)
   {
   cout<<"\nChoose that you want to sort..
   \n1.Integer\n2.Float\n3.Character\n4.String\n5.Exit\n"
   ;
         cin>>i;
```

```
if(i==1)
{
        cout<<"\nEnter the size of List\n";
        cin>>size;
        get(a,size);
        sorting(a,size);
        print(a,size);
}
else if(i==2)
{
        cout<<"\nEnter the size of List\n";
        cin>>size;
        get(b,size);
        sorting(b,size);
        print(b,size);
}
```

```
            else if(i==3)
            {
                        cout<<"\nEnter the size of List\n";
                        cin>>size;
                        get(c,size);
                        sorting(c,size);
                        print(c,size);
            }
            else if(i==4)
            {
                        cout<<"\nEnter the size of List\n";
                        cin>>size;
                        get(d,size);
                        sorting(d,size);
                        print(d,size);
            }
    }
}
```

Thank You.

C++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

- If derived class defines same function as defined in the base class, it is known as function overriding.
- If you create an object of the child class and call the member function which exists in both the classes, then the member function of the child class invoked and the function of base class is ignored.
- It enables you to provide specific implementation of the function which is already provided by the base class.


```
Class base

{

Public:

Void display()

{

Cout<< “show”;

}

};

Class Child: public base

{

Public:

Void display()

{

Cout<< “show”;

}

};
```

```
Int main()
{
Child c;
c.display();
}
```