UNIT IV

TEMPLATES AND EXCEPTION HANDLING

UNIT IV - TEMPLATES AND EXCEPTION HANDLING

- **Topics to be discussed,**
  - Function Template and Class Template
  - Namespaces
  - Casting
  - Exception Handling

UNIT IV - TEMPLATES AND EXCEPTION HANDLING

- **Topics to be discussed,**
  - **Function Template and Class Template**
  - Namespaces
  - Casting
  - Exception Handling

- Need same functionality but it should work for different data types, what concept you use?
  - **Function overloading**

Function Overloading

```
#include<iostream>
using namespace std;
int add(int x,int y)
{
    return (x+y);
}
float add(float x,float y)
{
    return (x+y);
}
int main()
{
    int int_sum;
    float float_sum;
    int_sum=add(12,78);
    cout<<"Integer sum:"<<int_sum<<endl;
    float_sum=add(34.789f,89.456f);
    cout<<"Float sum:"<<float_sum<<endl;

}
```

Output:

```
Integer sum:90
Float sum:124.245
```

Generics in C++

- Generic Programming enables the programmer to write a general algorithm which will work with all data types
- It eliminates the need to create different algorithms for different data types such as integer, string or a character.
- The advantages of Generic Programming are
  - Code Reusability
  - Avoid Function Overloading
  - Once written it can be used for multiple times and cases.
- **Generics can be implemented in C++ using Templates**.

Function Template and Class Template

- **Templates** are the **foundation of generic programming**
- **Templates** are powerful features of C++ which allows us to write **generic** programs.
- It involves writing code in a way that is independent of any particular type.
- A template is a **blueprint** or **formula** for creating a **generic class** or a **function**.
- In simple terms, we can create a single function or a class to work with different data types using templates.
  - For example, if we are writing a function to sort a sequence of data but we don't know what types of data will be passed to that function so we may need to write different functions for different data types, but rather than writing the same for function multiple times for different types we can use the Templates in C++ to make our sort function generic which would sort data of any type.

Function Template and Class Template – Cont'd

- Templates are often used in larger **codebase** for the purpose of code **reusability** and **flexibility** of the programs.
- The concept of templates can be used in two different ways:
  - **Function Templates**
  - **Class Templates**
- For using the templates tool in C++ we must know two keywords: **'template'** and **'typename'**, the typename keyword can be replaced with the keyword **'class'**.

Function Template and Class Template – Cont'd

C++ Function Template

- **We can create a single function to work with different data types by using a template.**
- **Defining a Function Template:**
  - A function template starts with the keyword template followed by template parameter(s) inside <> which is followed by the function definition.

```
template <typename T>
T functionName(T parameter1, T parameter2, ...)
{
    // code
}
```

  - In the above code, **T is a template argument** that accepts different data types (int, float, etc.), and **typename is a keyword**.
  - When an argument of a data type is passed to functionName(), the compiler generates a new version of functionName() for the given data type.

Function Template and Class Template – Cont'd

C++ Function Template

- **Calling a Function Template**
  - Once we've declared and defined a function template, we can call it in other functions or templates (such as the main() function) with the following syntax

```
functionName<dataType>(parameter1, parameter2,...);
```

C++ Function Template – Example 1

```
#include<iostream>
using namespace std;
template<typename T>
T add(T x,T y)
{
    return (x+y);
}
int main()
{
    int int_sum;
    float float_sum;
    int_sum=add<int>(12,78); // calling with int parameters
    cout<<"Integer sum:"<<int_sum<<endl;
    float_sum=add<float>(34.789,89.456);// calling with float parameters
    cout<<"Float sum:"<<float_sum<<endl;

}
```

```
int add(int x,int y)
{
    return (x+y);
}
float add(float x,float y)
{
    return (x+y);
}
```

Output:

```
Integer sum:90
Float sum:124.245
```

Function Template and Class Template – Cont'd

- **How Do Templates Work?**
  - Templates are a type of static (compile time) polymorphism
  - They expand at the compilation time.
  - This is similar to macros, the only difference is that the compiler does type-checking before the expansion of the template.
  - In simple words, the source code contains only a function/class but compiled code may contain multiple copies of the same function/class.

```
#include<iostream>

template<typename T>
T add(T num1, T num2) {
  return (num1 + num2);
}

int main() {
  ... ... ...

  result1 = add<int>(12,78);
  ... ... ...

  result2 = add< float > (34.789,89.456);
  ... ... ...

}
```

Compiler internally generates and adds below code

```
int add(int num1, int num2) {
    return (num1 + num2);
}
```

Compiler internally generates and adds below code

```
float add( float num1, float num2) {
  return (num1 + num2);
}
```

- The process of creating functions (with specific types) from function templates (with template types) is called **function template instantiation (or instantiation)**
- When a function is instantiated due to a function call, it's called **implicit instantiation**.
- A function that is instantiated from a template is technically called a **specialization**, but in common language is often called a **function instance**.
- The template from which a specialization is produced is called a **primary template**.

A function call without using angled brackets for same type of data

```
#include<iostream>
using namespace std;
template<typename T>
T add(T x,T y)
{
    return (x+y);
}
int main()
{
    int int_sum;
    float float_sum;
    int_sum=add(12,78); // will instantiate add(int, int)
    cout<<"Integer sum:"<<int_sum<<endl;
    float_sum=add(34.789,89.456);  // will instantiate add(double, double)
    cout<<"Float sum:"<<float_sum<<endl;

}
```

Output:

```
Integer sum:90
Float sum:124.245
```

A function call without using angled brackets for different types of data

```
18  #include<iostream>
19  using namespace std;
20  template<typename T>
21  T add(T x,T y)
22  {
23      return (x+y);
24  }
25  int main()
26  {
27      int int_sum;
28      float float_sum;
29      int_sum=add(12,7.8);
30      cout<<"Integer sum:"<<int_sum<<endl;
31      float_sum=add(34.789,89);
32      cout<<"Float sum:"<<float_sum<<endl;
33
34  }
35
```

ers

::Blocks | Search results | Cccc | Build log | Build messages | CppCheck/Vera++

	Line	Message
		=== Build file: "no target" in "no project" (compiler: unknown) ===
4\CS320...		In function 'int main()':
4\CS320...	29	error: no matching function for call to 'add(int, double)'
4\CS320...	21	note: candidate: 'template<class T> T add(T, T)'
4\CS320...	21	note: template argument deduction/substitution failed:
4\CS320...	29	note: deduced conflicting types for parameter 'T' ('int' and 'double')
4\CS320...	31	error: no matching function for call to 'add(double, int)'
4\CS320...	21	note: candidate: 'template<class T> T add(T, T)'
4\CS320...	21	note: template argument deduction/substitution failed:
4\CS320...	31	note: deduced conflicting types for parameter 'T' ('double' and 'int')
		=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

A function call without using angled brackets for different types of data

- In our function call add(12, 7.8), we're passing arguments of two different types: one int and one double.
- Because we're making a function call without using angled brackets to specify an actual type, the compiler will first look to see if there is a non-template match for max(int, double).
- It won't find one.
- Next, the compiler will see if it can find a function template match
- However, this will also fail, for a simple reason: T can only represent a single type.
- There is no type for T that would allow the compiler to instantiate function template add<T>(T, T) into a function with two different parameter types.
- Put another way, because both parameters in the function template are of type T, they must resolve to the same actual type.

A function call without using angled brackets for different types of data

```
#include<iostream>
using namespace std;
template<typename T>
T add(T x,T y)
{
    return (x+y);
}
int main()
{
    int int_sum;
    float float_sum;
    int_sum=add(12,static_cast<int>(7.8));
    cout<<"Integer sum:"<<int_sum<<endl;
    float_sum=add(34.789,static_cast<double>(89));
    cout<<"Float sum:"<<float_sum<<endl;

}
```

Output:

```
Integer sum:19
Float sum:123.789
```

A function call without using angled brackets for different types of data

```
#include<iostream>
using namespace std;
template<typename T,typename U>
T add(T x,U y)
{
    return (x+y);
}
int main()
{
    cout<<"sum1:"<<add(12,7.8)<<endl;
    cout<<"sum2:"<<add(7.8,12)<<endl;
    cout<<"sum3:"<<add(12.0,7.8)<<endl;
}
```

Output:

```
sum1:19
sum2:19.8
sum3:19.8
```

A function call without using angled brackets for different types of data

```
#include<iostream>
using namespace std;
template<typename T,typename U>
U add(T x,U y)
{
    return (x+y);
}
int main()
{
    cout<<"sum1:"<<add(12,7.8)<<endl;
    cout<<"sum2:"<<add(7.8,12)<<endl;
    cout<<"sum3:"<<add(12.0,7.8)<<endl;
}
```

Output:

```
sum1:19.8
sum2:19
sum3:19.8
```

A function call without using angled brackets for different types of data

```
#include<iostream>
using namespace std;
template<typename T,typename U>
auto add(T x,U y)
{
    return (x+y);
}
int main()
{
    cout<<"sum1:"<<add(12,7.8)<<endl;
    cout<<"sum2:"<<add(7.8,12)<<endl;
    cout<<"sum3:"<<add(12.0,7.8)<<endl;
}
```

auto return type -- we'll let the compiler deduce what the return type should be from the return statement

Output:

```
sum1:19.8
sum2:19.8
sum3:19.8
```

Abbreviated function templates C++20

- C++20 introduces a new use of the auto keyword: When the auto keyword is used as a parameter type in a normal function, the compiler will automatically convert the function into a function template with each auto parameter becoming an independent template type parameter. This method for creating a function template is called an abbreviated function template.

```
#include<iostream>
using namespace std;
auto add(auto x, auto y)
{
    return (x+y);
}
int main()
{
    cout<<"sum1:"<<add(12,7.8)<<endl;
    cout<<"sum2:"<<add(7.8,12)<<endl;
    cout<<"sum3:"<<add(12.0,7.8)<<endl;
}
```

Output:

```
sum1:19.8
sum2:19.8
sum3:19.8
```

is shorthand in C++20 for the following:

```
template <typename T, typename U>
auto max(T x, U y)
{
    return (x+y)
}
```

which is the same as the add function template we wrote before.

C++ Function Template – Example 2

```
#include<iostream>
using namespace std;
template<typename T>
void sortData(T a[],int n)
{
    int i,j;
    T t;
    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(a[j]<a[i])
            {
                t=a[i];
                a[i]=a[j];
                a[j]=t;
            }
        }
    }
}
```

```
int main()
{
    int n;
    cout<<"\nEnter n:";
    cin>>n;
    int *a=new int[n];
    cout<<"\nEnter integer elements:";
    for(int i=0;i<n;i++)
        cin>>a[i];
    cout<<"\nGiven data:";
    for(int i=0;i<n;i++)
        cout<<a[i]<<" ";
    sortData<int>(a,n);
    cout<<"\nSorted data:";
    for(int i=0;i<n;i++)
        cout<<a[i]<<" ";
    cout<<"\nEnter n:";
    cin>>n;
    float *b=new float[n];
    cout<<"\nEnter floating point data:";
    for(int i=0;i<n;i++)
        cin>>b[i];
    cout<<"\nGiven data:";
    for(int i=0;i<n;i++)
        cout<<b[i]<<" ";
    sortData<float>(b,n);
    cout<<"\nSorted data:";
    for(int i=0;i<n;i++)
        cout<<b[i]<<" ";
}
```

Output:

```
Enter n:4

Enter integer elements:34
56
12
8

Given data:34 56 12 8
Sorted data:8 12 34 56
Enter n:5

Enter floating point data:12.7
67.45
23.78
11.345
55.89

Given data:12.7 67.45 23.78 11.345 55.89
Sorted data:11.345 12.7 23.78 55.89 67.45
```

Create the C++ Function Template named swapNum so that it has two parameters of the same type. A Template Function created from swapNum will exchange the values of these two parameters. (Test for two different types of data)

```
#include<iostream>
using namespace std;
template<typename T>
void swapNum(T &x,T &y)
{
   T t;
   t=x;
   x=y;
   y=t;
}
int main()
{
   int n1,n2;
   cout<<"\nEnter two integers:";
   cin>>n1>>n2;
   cout<<"\nBefore swap:";
   cout<<"\nn1="<<n1<<"\nn2="<<n2;
   swapNum<int>(n1,n2);
   cout<<"\nAfter swap:";
   cout<<"\nn1="<<n1<<"\nn2="<<n2;
   float f1,f2;
   cout<<"\nEnter float values:";
   cin>>f1>>f2;
   cout<<"\nBefore swap:";
   cout<<"\nf1="<<f1<<"\nf2="<<f2;
   swapNum<float>(f1,f2);
   cout<<"\nAfter swap:";
   cout<<"\nf1="<<f1<<"\nf2="<<f2;
}
```

```
Enter two integers:45 78

Before swap:
n1=45
n2=78
After swap:
n1=78
n2=45
Enter float values:89.98 34.786

Before swap:
f1=89.98
f2=34.786
After swap:
f1=34.786
f2=89.98
```

Function Template and Class Template – Cont'd

- **Class Templates**
  - Similar to function templates, we can use class templates **to create a single class to work with different data types**.
  - Class templates come in handy as they can make our code shorter and more manageable.
- **Class Template Declaration**
  - A class template starts with the keyword template followed by template parameter(s) inside <> which is followed by the class declaration.

```
template <class T>
class className
{
  private:
    T var;
    ... .. ...
  public:
    T functionName(T arg);
    ... .. ...
};
```

Function Template and Class Template – Cont'd

- **Creating a Class Template Object**
  - Once we've declared and defined a class template, we can create its objects in other classes or functions (such as the main() function) with the following syntax:

    className<dataType> classObject;

- For example,

  className<int> classObject;

  className<float> classObject;

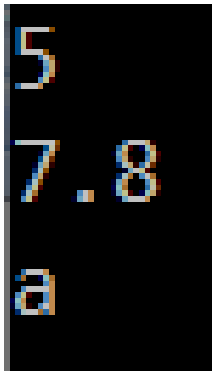
  className<string> classObject;

Class Template - Example

```
#include<iostream>
using namespace std;
template <class T>
class Number
{
   private:
   T n;
   public:
   Number(T x)
   {
      n=x;
   }
   T getVar()
   {
      return n;
   }
};
```

```
int main()
{
   Number <int> n1(5);
   cout<<n1.getVar()<<endl;
   Number <float> n2(7.8);
   cout<<n2.getVar()<<endl;
   Number <char> n3('a');
   cout<<n3.getVar()<<endl;
}
```

Output:



Function Template and Class Template – Cont'd

- **Defining a Class Member Outside the Class Template**
  - Suppose we need to define a function outside of the class template. We can do this with the following code:

```
template <class T>
class ClassName
{
   ... .. ...
   // Function prototype
   returnType functionName();
};
// Function definition
template <class T>
returnType ClassName<T>::functionName()
{
   // code
}
```

Notice that the code template <class T> is repeated while defining the function outside of the class. This is necessary and is part of the syntax.

```
#include <iostream>
using namespace std;
template <class T>
class Calculator
{
  private:
   T num1, num2;
   public:
   Calculator(T n1, T n2)
   {
      num1 = n1;
      num2 = n2;
   }
   void displayResult();
   T add() { return num1 + num2; }
   T subtract() { return num1 - num2; }
   T multiply() { return num1 * num2; }
   T divide() { return num1 / num2; }
};
```

```
template <class T>
 void Calculator<T>:: displayResult()
 {
      cout << "Numbers: " << num1 << " and " << num2 << "." << endl;
      cout << num1 << " + " << num2 << " = " << add() << endl;
      cout << num1 << " - " << num2 << " = " << subtract() << endl;
      cout << num1 << " * " << num2 << " = " << multiply() << endl;
      cout << num1 << " / " << num2 << " = " << divide() << endl;
   }

int main()
{
   Calculator<int> intCalc(6,7);
   Calculator<float> floatCalc(5.6,7.8);
   cout << "Int results:" << endl;
   intCalc.displayResult();
   cout << endl;
   cout<< "Float results:" << endl;
   floatCalc.displayResult();
   return 0;
}
```

Output:

```
Int results:
Numbers: 6 and 7.
6 + 7 = 13
6 - 7 = -1
6 * 7 = 42
6 / 7 = 0

Float results:
Numbers: 5.6 and 7.8.
5.6 + 7.8 = 13.4
5.6 - 7.8 = -2.2
5.6 * 7.8 = 43.68
5.6 / 7.8 = 0.717949
```

Function Template and Class Template – Cont'd

- **C++ Class Templates With Multiple Parameters**
  - In C++, we can use multiple template parameters and even use default arguments for those parameters.
  - For example,

```
template <class T, class U, class V = int>
class ClassName
{
 private:
   T member1;
   U member2;
   V member3;
   ... .. ...
 public:
   ... .. ...
};
```

Class Templates With Multiple Parameters - Example

```
#include <iostream>
using namespace std;
// Class template with multiple and default parameters
template <class T, class U, class V = char>
class ClassTemplate
{
  private:
   T var1;
   U var2;
   V var3;
  public:
   ClassTemplate(T v1, U v2, V v3)
   {
      var1=v1;
      var2=v2;
      var3=v3;
   }
   void printVar()
   {
      cout << "var1 = " << var1 << endl;
      cout << "var2 = " << var2 << endl;
      cout << "var3 = " << var3 << endl;
   }
};
```

```
int main()
{
   // create object with int, double and char types
   ClassTemplate<int, double> obj1(7, 7.7, 'c');
   cout << "obj1 values: " << endl;
   obj1.printVar();

   // create object with int, double and bool types
   ClassTemplate<double, char, bool> obj2(8.8, 'a', false);
   cout << "\nobj2 values: " << endl;
   obj2.printVar();
   return 0;
}
```

Output:

```
obj1 values:
var1 = 7
var2 = 7.7
var3 = c

obj2 values:
var1 = 8.8
var2 = a
var3 = 0
```