



Dr. D. Y. Patil Pratishthan's
**Institute for Advanced Computing
and Software Development**



Sub-c++
Day8

Templates

- Blueprint or formula for creating a generic class or a function.
- We can create a single function or single class to work with different data types using templates.
- it gets expanded at compilation time, just like macros and allows a function or class to work on different data types without being rewritten.
- Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

- Templates are powerful features of C++ which allows us to write generic programs.
- There are two ways we can implement templates:
 - Function Templates
 - Class Templates
- Similar to function templates,
- class templates used 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.

Function Templates

- A function template starts with the keyword `template` followed by template parameter(s) inside `<>` which is followed by the function definition.

```
template <class 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 `class` 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.

Calling function

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

For example, let us consider a template that adds two numbers:

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

We can then call it in the main() function to add int and double numbers.

```
int main() {  
    int result1;  
  
    double result2;  
  
    // calling with int parameters  
    result1 = add<int>(2, 3);  
    cout << result1 << endl;  
  
    // calling with double parameters  
    result2 = add<double>(2.2, 3.3);  
    cout << result2 << endl;  
  
    return 0;  
}
```

```
#include<iostream>
```

```
template<typename T>
```

```
T add(T num1, T num2) {
```

```
    return (num1 + num2);
```

```
}
```

```
int main() {
```

```
    ... ..
```

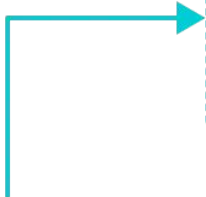
```
    result1 = add<int>(2,3);
```

```
    ... ..
```


```
    result2 = add<double>(2.2,3.3);
```

```
    ... ..
```

```
}
```



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



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

Class Template

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);

        ... ..

};
```


In the above declaration, `T` is the template argument which is a placeholder for the data type used, and `class` is a keyword.

Inside the class body, a member variable `var` and a member function `functionName()` are both of type `T`.

Creating a Class Template Object

syntax

```
className<dataType> classObject;
```

For example,

```
className<int> classObject;
```

```
className<float> classObject;
```

```
className<string> classObject;
```

Defining a Class Member Outside the Class Template

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

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

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:
```

```
        ... .. }
```

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.

Manipulators in c++

Manipulators are helping functions that can modify the [input/output](#) stream. It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.

Manipulators are special functions that can be included in the I/O statement to alter the format parameters of a stream.

Manipulators are operators that are used to format the data display.

To access manipulators, the file `iomanip.h` should be included in the program.

Manipulators without arguments: The most important manipulators defined by the **IOStream library** are provided below.

1.endl: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.

ws: It is defined in istream and is used to ignore the whitespaces in the string sequence.

ends: It is also defined in ostream and it inserts a null character into the output stream. It typically works with std::ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.

flush: It is also defined in ostream and it flushes the output stream, i.e. it forces all the output written on the screen or in the file.

Without flush, the output would be the same, but may not appear in real-time.

Manipulators with Arguments: Some of the manipulators are used with the argument like `setw (20)`, `setfill ('*')`, and many more. These all are defined in the header file. If we want to use these manipulators then we must include this header file in our program. For Example, you can use following manipulators to set minimum width and fill the empty space with any character you want: `std::cout << std::setw (6) << std::setfill ('*');` **Some important manipulators in `<iomanip>` are:**

setw (val): It is used to set the field width in output operations.

setfill (c): It is used to fill the character 'c' on output stream.

setprecision (val): It sets val as the new value for the precision of floating-point values.

C++ Exception Handling

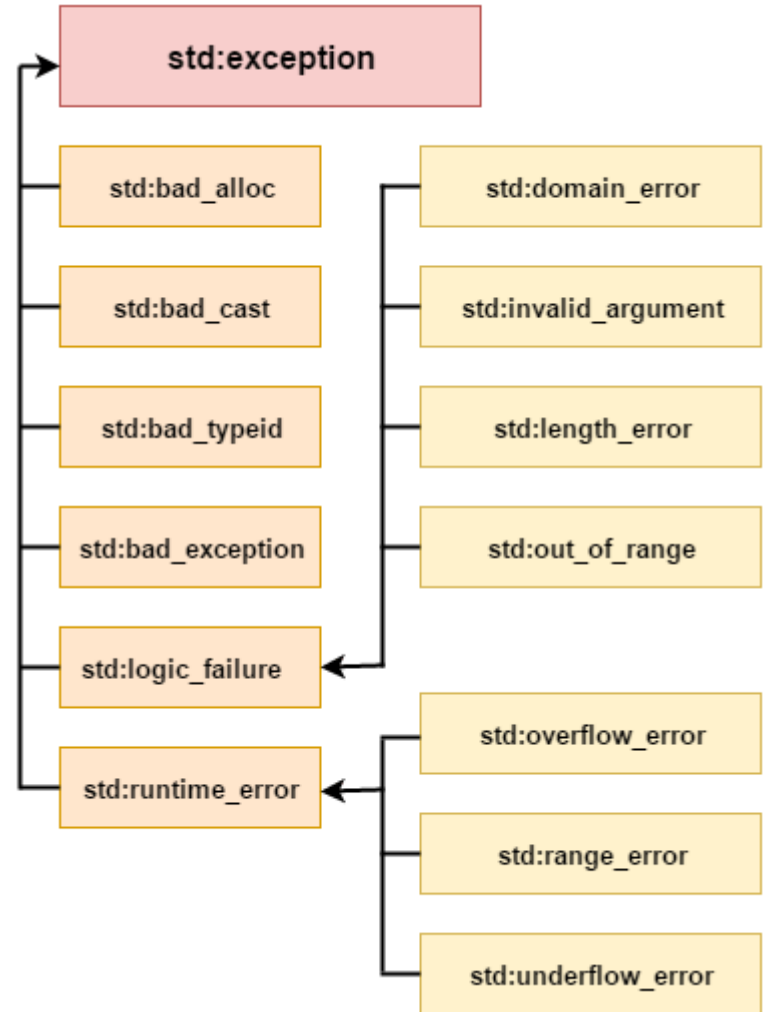
- Exception Handling in C++ is a process to handle runtime errors.
- We perform exception handling so the normal flow of the application can be maintained even after runtime errors.
- In C++, exception is an event or object which is thrown at runtime.
- All exceptions are derived from `std::exception` class.
- It is a runtime error which can be handled.
- If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It maintains the normal flow of the application.

In such case, rest of the code is executed even after exception.

C++ Exception class-



Exception Description

1. `std::exception` -It is an exception and parent class of all standard C++ exceptions.
2. `std::logic_failure`-It is an exception that can be detected by reading a code.
3. `std::runtime_error`-It is an exception that cannot be detected by reading a code.
4. `std::bad_exception`-It is used to handle the unexpected exceptions in a c++ program.
5. `std::bad_cast`-This exception is generally be thrown by **dynamic_cast**.
6. `std::bad_typeid`-This exception is generally be thrown by **typeid**.
7. `std::bad_alloc`-This exception is generally be thrown by **new**.

C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

try

catch

throw

```
//program without try catch
#include <iostream>
using namespace std;
float division(int x, int y) {
    return (x/y);
}
int main () {
    int i = 50;
    int j = 0;
    float k = 0;
    k = division(i, j);
    cout << k << endl;
    return 0;
}
```

Output:

Floating point exception (core dumped)

```
Include<iostream>
using namespace std;
float division(int x, int y) {
    if( y == 0 ) {
        throw "Attempted to divide by zero!";
    }
    return (x/y);
}
```

```
int main () {
    int i = 25;
    int j = 0;
    float k = 0;
    try {
        k = division(i, j);
        cout << k << endl;
    } catch (const char* e) {
        cerr << e << endl;
    }
    return 0;
}
```

Output:

Attempted to divide by zero!

C++ user-defined exception example

Let's see the simple example of user-defined exception in which **std::exception** class is used to define the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception{
    public:
        const char * what() const
throw()
    {
        return "Attempted to divide by zero!\n";
    }
};
```

```
int main()
{
    try
    { int x, y;
      cout << "Enter the two numbers : \n";
      cin >> x >> y;
      if (y == 0)
      {
          MyException z;
          throw z;
      }
      else
      { cout << "x / y = " << x/y << endl;
      }
    }
    catch(exception& e)
    { cout << e.what();
    }
}
```


output

Enter the two numbers :

10 2

$x / y = 5$

Output

Enter the two numbers : 10 0 Attempted to divide by zero!

In above example `what()` is a public method provided by the exception class. It is used to return the cause of an exception.

Rethrow exception

```
#include <iostream>
using namespace std;
void MyHandler()
{
    try
    {
        throw "hello";
    }
    catch (const char*)
    {
        cout << "Caught exception inside
MyHandler\n";
        throw; //rethrow char* out of function
    }
}
```

```
int main()
{
    cout<< "Main start";
    try
    {
        MyHandler();
    }
    catch(const char*)
    {
        cout << "Caught exception inside Main\n";
    }
    cout << "Main end";
    return 0;
}
```

Output :Main start

Caught exception inside MyHandler

Caught exception inside Main

Main end