Generic - Templates - Function templates - Class Templates - Exceptional Handling: try and catch - Multilevel exceptional - throw and throws - finally - User defined exceptional - Dynamic Modeling: Package Diagram - UML Component Diagram - UML Deployment Diagram

**Generics in C++**

The method of Generic Programming is implemented to increase the efficiency of the code. 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 if the data type is an integer, string or a character.

The advantages of Generic Programming are

1. Code Reusability
2. Avoid Function Overloading
3. Once written it can be used for multiple times and cases.

Generics can be implemented in C++ using [**Templates**](https://www.geeksforgeeks.org/templates-cpp/).

**Template** is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don’t need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

## Function Template

* Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
* The type of the data that the function will operate on depends on the type of the data passed as a parameter.
* For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
* A Generic function is created by using the keyword template. The template defines what function will do.

### Syntax of Function Template

1. **template** < **class** Ttype> ret\_type func\_name(parameter\_list)
2. {
3. // body of function.
4. }

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

**class**: A class keyword is used to specify a generic type in a template declaration.

**Let's see a simple example of a function template:**

1. #include <iostream>
2. **using** **namespace** std;
3. **template**<**class** T> T add(T &a,T &b)
4. {
5. T result = a+b;
6. **return** result;
8. }
9. **int** main()
10. {
11. **int** i =2;
12. **int** j =3;
13. **float** m = 2.3;
14. **float** n = 1.2;
15. cout<<"Addition of i and j is :"<<add(i,j);
16. cout<<'\n';
17. cout<<"Addition of m and n is :"<<add(m,n);
18. **return** 0;
19. }

**Output:**

Addition of i and j is :5

Addition of m and n is :3.5

### Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

## Syntax

1. **template**<**class** T1, **class** T2,.....>
2. return\_type function\_name (arguments of type T1, T2....)
3. {
4. // body of function.
5. }

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

**Let's see a simple example:**

1. #include <iostream>
2. **using** **namespace** std;
3. **template**<**class** X,**class** Y> **void** fun(X a,Y b)
4. {
5. std::cout << "Value of a is : " <<a<< std::endl;
6. std::cout << "Value of b is : " <<b<< std::endl;
7. }
9. **int** main()
10. {
11. fun(15,12.3);
13. **return** 0;
14. }

**Output:**

Value of a is : 15

Value of b is : 12.3

### Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

**Let's understand this through a simple example:**

1. #include <iostream>
2. **using** **namespace** std;
3. **template**<**class** X> **void** fun(X a)
4. {
5. std::cout << "Value of a is : " <<a<< std::endl;
6. }
7. **template**<**class** X,**class** Y> **void** fun(X b ,Y c)
8. {
9. std::cout << "Value of b is : " <<b<< std::endl;
10. std::cout << "Value of c is : " <<c<< std::endl;
11. }
12. **int** main()
13. {
14. fun(10);
15. fun(20,30.5);
16. **return** 0;
17. }

**Output:**

Value of a is : 10

Value of b is : 20

Value of c is : 30.5

## CLASS TEMPLATE

**Class Template** can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

## Syntax

1. **template**<**class** Ttype>
2. **class** class\_name
3. {
4. .
5. .
6. }

**Ttype** is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

class\_name<type> ob;

**where class\_name**: It is the name of the class.

**type**: It is the type of the data that the class is operating on.

**ob**: It is the name of the object.

**Let's see a simple example:**

1. #include <iostream>
2. **using** **namespace** std;
3. **template**<**class** T>
4. **class** A
5. {
6. **public**:
7. T num1 = 5;
8. T num2 = 6;
9. **void** add()
10. {
11. std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
12. }
14. };
16. **int** main()
17. {
18. A<**int**> d;
19. d.add();
20. **return** 0;
21. }

**Output:**

Addition of num1 and num2 : 11

### CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

## Syntax

1. **template**<**class** T1, **class** T2, ......>
2. **class** class\_name
3. {
4. // Body of the class.
5. }

**Let's see a simple example when class template contains two generic data types.**

1. #include <iostream>
2. **using** **namespace** std;
3. **template**<**class** T1, **class** T2>
4. **class** A
5. {
6. T1 a;
7. T2 b;
8. **public**:
9. A(T1 x,T2 y)
10. {
11. a = x;
12. b = y;
13. }
14. **void** display()
15. {
16. std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
17. }
18. };
20. **int** main()
21. {
22. A<**int**,**float**> d(5,6.5);
23. d.display();
24. **return** 0;
25. }

**Output:**

Values of a and b are : 5,6.5

### Nontype Template Arguments

The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.

**Let' s see the following example:**

1. **template**<**class** T, **int** size>
2. **class** array
3. {
4. T arr[size];           // automatic array initialization.
5. };

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

1. array<**int**, 15> t1;                        // array of 15 integers.
2. array<**float**, 10> t2;                    // array of 10 floats.
3. array<**char**, 4> t3;                      // array of 4 chars.

Let's see a simple example of nontype template arguments.

1. #include <iostream>
2. **using** **namespace** std;
3. **template**<**class** T, **int** size>
4. **class** A
5. {
6. **public**:
7. T arr[size];
8. **void** insert()
9. {
10. **int** i =1;
11. **for** (**int** j=0;j<size;j++)
12. {
13. arr[j] = i;
14. i++;
15. }
16. }
18. **void** display()
19. {
20. **for**(**int** i=0;i<size;i++)
21. {
22. std::cout << arr[i] << " ";
23. }
24. }
25. };
26. **int** main()
27. {
28. A<**int**,10> t1;
29. t1.insert();
30. t1.display();
31. **return** 0;
32. }

**Output:**

1 2 3 4 5 6 7 8 9 10

**Points to Remember**

* C++ supports a powerful feature known as a template to implement the concept of generic programming.
* A 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.

**Exception Handling in C++**

Errors are the problems that occur in the program due to an illegal operation performed by the user or by the fault of a programmer.

Exception Handling is the process of handling errors and exceptions such that the normal execution of the system is not halted. Exception handling in c++ consists of three keywords namely- try, catch, and throw.

### a. Compile Time Errors

Compile Time Errors are those errors that are caught during compilation time. Some of the most common compile-time errors are syntax errors, library references, incorrect import of library functions and methods, uneven bracket pair(s), etc.

#include <iostream>

using namespace std;

int main()

{

cout << "Hello Scaler Topics!"

return 0;

}

**Output:**

test.cpp: In function 'int main()':

test.cpp:6:35: error: expected ';' before 'return'

cout << "Hello Scaler Topics!"

^

;

test.cpp:8:5:

return 0;

### Run-Time Errors

Run-Time Errors are those errors that cannot be caught during compilation time. As we cannot check these errors during compile time, we name them Exceptions. Exceptions can cause some serious issues so we should handle them effectively.

#include <iostream>

using namespace std;

int main()

{

int a = 5;

// Dividing the number a by zero, so the program will compile easily

// but run time error will be generated.

cout << a / 0;

return 0;

}

**Output:**

**test.cpp: In function 'int main()':**

**test.cpp:11:14: warning: division by zero [-Wdiv-by-zero]**

**cout << a / 0;**

**~~^~~**

Exception handling in C++ was introduced to deal with abnormal run-time anomalies and abnormal conditions caused during run time.

One of the most common run time exceptions can be 0 division error. When we try to divide a number by 0, then the program will get executed successfully but during compile time, we will face an error causing an application crash.

For exception handling in C++, we use the try-catch-finally block.

**Syntax:**

try {

// Block of code to try

throw exception;

}

catch () {

// Block of code to handle errors

}

## Examples of Exception Handling in C++

#include <iostream>

using namespace std;

int division(int a, int b)

{

// checking if the denominator is 0 or not.

if (b == 0)

{

// if the denominator is 0, then we must throw an exception

throw "Division by zero!";

}

// if there is no exception, then we are returning the answer.

return int(a / b);

}

int main()

{

int x = 50;

int y = 0;

int answer = 0;

/\*

Using a try catch block because in divison, the denominator can be 0.

So, we must handle the 0 divion inside try block.

\*/

try

{

answer = division(x, y);

cout << " Output: " << answer << endl;

}

// printing the thrown exception from the function

catch (const char \*errorMessage)

{

cout << errorMessage << endl;

}

return 0;

}

**Output:**

Division by zero!

Let us take another example. Suppose we are trying to develop a program that gives access to its users when the user's age is more than or equal to 18 years. We can also use if-else statements for this purpose, but let us do age checking using exception handling in c++.

#include <iostream>

using namespace std;

/\*

Here, we want to throw exception (age as exception) if the age of the person is less than 18.

\*/

int main()

{

// checking if the age is more than 18 in tr block.

try

{

int age = 15;

if (age >= 18)

{

cout << "Access granted.";

}

// Throwing custom exception if the age is less than 18.

else

{

throw(age);

}

}

// catching the thrown exception and displaying the desired output (access denied!)

catch (int x)

{

cout << "Access denied!, Age is: " << x << endl;

}

return 0;

}

**Output:**

Access denied!, Age is: 15

### a. C++ try

The try block is used to keep the code that is expected to throw some exception. Whenever our code leads to any exception or error, the exception or error gets caught in the catch block. In simple terms, we can say that the try block is used to define the block of code that needs to be tested for errors while it is being executed.

**Example:** Suppose we are dealing with databases, we should put the code that is handling the database connection inside a try block as the database connection may raise some exceptions or errors.

### b. C++ catch

The catch block is used to catch and handle the error(s) thrown from the try block. If there are multiple exceptions thrown from the try block, then we can use multiple catch blocks after the try blocks for each exception. In this way, we can perform different actions for the various occurring exceptions. In simple terms, we can say that the catch block is used to define a block of code to be executed if an error occurs in the try block.

**Example:** Let us take the same above example that we are dealing with the database. Now, if during the connection, there is an exception raised inside the try block, then there should be a catch block present to catch or accept the exception and handle the exception. The catch block ensures that the normal flow of the code is not halted.

**Note:**

* The try and catch block comes is used in pairs.
* We can have multiple catch blocks for one try statement.

### c. C++ throw

The throw block is used to throw exceptions to the exception handler which further communicates the error. The type of exception thrown should be same in the catch block. The throw keyword accepts one parameter which is passed to the exception handler. We can throw both pre-defined as well as custom exception(s) as per the requirements.

## Exception Handling with Multiple Catch Example Program

## #include<iostream.h>

## #include<conio.h>

## void test(int x) {

## try {

## if (x > 0)

## throw x;

## else

## throw 'x';

## } catch (int x) {

## cout << "Catch a integer and that integer is:" << x;

## } catch (char x) {

## cout << "Catch a character and that character is:" << x;

## }

## }

## void main() {

## clrscr();

## cout << "Testing multiple catches\n:";

## test(10);

## test(0);

## getch();

## }

## Testing multiple catches

## Catch a integer and that integer is: 10

## Catch a character and that character is: x

**Simple C++ Program for Catch All or Default Exception Handling**

## /\*## Simple C++ Program for Catch All or Default Exception Handling \*/

## /\*## Exception Handling C++ Programs, Exception Handling Programming\*/

## // Header Files

## #include <iostream>

## #include<conio.h>

## using namespace std;

## int main() {

## int var = 0;

## cout << "Simple C++ Program for Catch All or Default Exception Handling\n";

## // try block - for exception code

## try {

## // Inside try block

## if (var == 0) {

## throw var;

## // Skip all code after throw

## }

## } // catch block

## catch (float ex) {

## // Code executed when exception Catch with float type

## cout << "Float Exception catch : Value :" << ex;

## } catch (...) {

## // Code executed when exception Catch : for default

## cout << "Default Exception Catch";

## }

## getch();

## return 0;

## }

## Sample Output

Simple C++ Program for Catch All or Default Exception Handling

Default Exception Catch

**Simple C++ Program for Nested Exception Handling**

/\*## Simple C++ Program for Nested Exception Handling \*/

/\*## Exception Handling C++ Programs, Exception Handling Programming\*/

## #include <iostream>

## #include<conio.h>

## using namespace std;

## int main() {

## int var = 0;

## 

## 

## try {

## 

## try {

## throw var;

## }

## catch (int ex) {

## 

## cout << "Nested Exception Catch : Value :" << ex;

## }

## } catch (...) {

## 

## cout << "Default Exception Catch";

## }

## getch();

## return 0;

## }

## Sample Output

Nested Exception Catch : Value :0

##### **User Defined Exception in C++**

There may be situations where you want to generate some user/program specific exceptions which are not pre-defined in C++. In such cases C++ provided us with the mechanism to create our own exceptions by **inheriting** the **exception** class in C++ and **overriding** its functionality according to our needs. Let us see an example of user-defined exception:

## #include <iostream>

## #include <exception>

## using namespace std;

## class MyException : public exception {

## public:

## char \* what () {

## return "C++ Exception";

## }

## };

## 

## int main() {

## try {

## throw MyException();

## }catch(MyException e) {

## cout << "MyException caught" <<endl;

## cout << e.what() <<endl;

## } catch(exception e) {

## //Other errors

## }

## 

## return 0;

## }

**Output**

MyException caught  
C++ Exception

##### **Program Explanation**

* In order to create a user defined exception we first included the <exception> header using the pre-processor directive. This header has the base class exception which we have to inherit and override to create any user defined exceptions
* In the next step we created a class named MyException and inherited all properties from the exception class. Then we created function what() which basically returns an error message string (in this case – C++ Exception). So whenever an exception of type MyException occurs, this message will be displayed. The return type of this function what() is char\* because we are returning a character string.
* In the last step, inside the main function we create a try{}…catch() block and inside the try block we create an object of MyException class and use the throw keyword to explicitly throw an exception using this object.
* This Object is then caught in the catch block where we print out the message by accessing the what() function of this(MyException) class’s object.

##### **User Defined Exception in C++ Sample Program**

## #include <iostream>

## #include <exception>

## using namespace std;

## class OverSpeed : public exception{

## int speed;

## public :

## const char\* what(){

## return "check out ur car speed you are in the car not in an aeroplane ";

## }

## };

## 

## int main()

## {

## int carspeed=0;

## try

## {

## while(1)

## {

## carspeed+=10;

## if(carspeed>100)

## {

## 

## OverSpeed s;

## throw s;

## }

## cout<<"Carspeed: "<<carspeed<<endl;

## }

## 

## }

## catch(OverSpeed ex)

## {

## cout<<ex.what();

## }

## 

## return 0;

## }

OUTPUT

Carspeed: 10

Carspeed: 20

Carspeed: 30

Carspeed: 40

Carspeed: 50

Carspeed: 60

Carspeed: 70

Carspeed: 80

Carspeed: 90

Carspeed: 100

check out ur car speed you are in the car not in an aeroplane