#include<stdio.h>

#include<string.h>

int main()

{

char ch[10]="hello";

int x=strlen(ch);

printf("%d %d",x,sizeof(ch));

}

#include<stdio.h>

#include<string.h>

int main()

{

char ch[10]="hel\0lo";

int x=strlen(ch);

printf("%d %d %s",x,sizeof(ch),ch);

}

#include<iostream>

using namespace std;

int ustrlen(char\*);

int main()

{

char ch[10];

cout<<"enter string\n";

cin>>ch;

int len=ustrlen(ch);

cout<<"length is "<<len;

}

int ustrlen(char\* ch1)

{

int len=0;

while(\*ch1!='\0')

{

len++;

ch1++;

}

return len;

}

====================================================================================

#include<iostream>

using namespace std;

void ustrcpy(char\*,char\*);

int main()

{

char sch[10],tch[10];

cout<<"enter string\n";

cin>>sch;

ustrcpy(tch,sch);

cout<<"target string is "<<tch;

}

void ustrcpy(char\* tch1,char\* sch1)

{

while(\*sch1 !='\0')

{

\*tch1=\*sch1;

tch1++;

sch1++;

}

\*tch1='\0';

}

=========================================================================

#include<iostream>

using namespace std;

void ustrcat(char\*,char\*);

int main()

{

char ch[10],ch1[10];

cout<<"enter string\n";

cin>>ch;//hello

cout<<"enter string\n";

cin>>ch1;//bye

ustrcat(ch,ch1);

cout<<"concatenated string is "<<ch;

}

void ustrcat(char\* ch2,char\* ch3)

{

while(\*ch2!='\0')

{

ch2++;

}

while(\*ch3!='\0')

{

\*ch2=\*ch3;

ch2++;

ch3++;

}

\*ch2='\0';

}

=================================================================================

#include<iostream>

using namespace std;

#include<string.h>

int main()

{

char ch[10],ch1[10];

int y=0;

cout<<"enter string\n";

cin>>ch;//hello

int x=strlen(ch)-1;//x=5

while(x>=0)

{

ch1[y]=ch[x];

x--;

y++;

}

ch1[y]='\0';

cout<<"reverse string is "<<ch1;

}

================================================================================

Generic Programming

Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to functions and classes For example, classes like an array, map, etc, which can be used using generics very efficiently. We can use them for any type. 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

Code Reusability

Avoid Function Overloading

Once written it can be used for multiple times and cases.

Generics can be implemented in C++ using Templates.

Templates

==> C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. 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.

==>How Do Templates Work?

Templates are expanded at compile time. This is like macros. The difference is, that the compiler does type-checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.

Templates can be represented in two ways:

Function template

==>We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

==>We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().

==>Function Template is used to write generic functions that can be used with different datatypes.The method of Generic Programming is implemented to increase the efficiency of the code.

==>The types of parameters passed to it during its call decide the actual data types which the function will operate upon. Generic functions are like 'write once use multiple times'.

For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.

==>This generality of coding reduces the programmers efforts of writing multiple function definitions having the same implementation or algorithm for different datatypes that might be passed to it. This saves time,disk space,and efforts to rewrite the source code.

==>==>Function templates are special functions that can operate with generic types. This allows creating a function template whose functionality can be adapted to more than one type without repeating the entire code for each type.

A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

template < class Ttype>

Ttype func\_name(parameter\_list)

{

// body of function.

}

Where Ttype: It is a placeholder name for a data type used by the function and this list is called as template parameter list and it cannot be empty. 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.

#include <iostream>

using namespace std;

template<class T>

T add(T &a,T &b)

{

T result = a+b;

return result;

}

int main()

{

int i =2;

int j =3;

float m = 2.3;

float n = 1.2;

cout<<"Addition of i and j is :"<<add(i,j);

cout<<'\n';

cout<<"Addition of m and n is :"<<add(m,n);

return 0;

}

Addition of i and j is :5

Addition of m and n is :3.5

In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

===================================================================

#include <iostream>

using namespace std;

template <class T>

T myMax(T x, T y)

{

return (x > y) ? x : y;

}

int main()

{

// Call myMax for int

cout << myMax<int>(3, 7) << endl;

// call myMax for double

cout << myMax<double>(3.0, 7.0) << endl;

// call myMax for char

cout << myMax<char>('g', 'e') << endl;

return 0;

}

==============================================================

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

template<class T1, class T2,.....>

return\_type function\_name (arguments of type T1, T2....)

{

// body of function.

}

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

#include <iostream>

using namespace std;

template<class X,class Y>

void fun(X a,Y b)

{

std::cout << "Value of a is : " <<a<< std::endl;

std::cout << "Value of b is : " <<b<< std::endl;

}

int main()

{

fun(15,12.3);

return 0;

}

Output:

Value of a is : 15

Value of b is : 12.3

In the above example, we use two generic types in the template function, i.e., X and Y.

============================================================================

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:

#include <iostream>

using namespace std;

template<class X>

void fun(X a)

{

std::cout << "Value of a is : " <<a<< std::endl;

}

template<class X,class Y>

void fun(X b ,Y c)

{

std::cout << "Value of b is : " <<b<< std::endl;

std::cout << "Value of c is : " <<c<< std::endl;

}

int main()

{

fun(10);

fun(20,30.5);

return 0;

}

Output:

Value of a is : 10

Value of b is : 20

Value of c is : 30.5

In the above example, template of fun() function is overloaded.

=========================================================================

=========================================================================

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

template<class Ttype>

class class\_name

{

.

.

}

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:

#include <iostream>

using namespace std;

template<class T>

class A

{

public:

T num1 = 5;

T num2 = 6;

void add()

{

std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;

}

};

int main()

{

A<int> d;

d.add();

return 0;

}

Output:

Addition of num1 and num2 : 11

In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

===================================================================================

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

template<class T1, class T2, ......>

class class\_name

{

// Body of the class.

}

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

#include <iostream>

using namespace std;

template<class T1, class T2>

class A

{

T1 a;

T2 b;

public:

A(T1 x,T2 y)

{

a = x;

b = y;

}

void display()

{

std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;

}

};

int main()

{

A<int,float> d(5,6.5);

d.display();

return 0;

}

Output:

Values of a and b are : 5,6.5

=========================================================================

#include <iostream>

using namespace std;

template<class T>

class A

{

public:

T a, b;

public:

void display(T,T);

};

template<class T>

void A<T>:: display(T a ,T b)

{

std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;

}

int main()

{

A<int> d;

d.display(5,6);

return 0;

}

========================================================================

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:

template<class T, int size>

class array

{

T arr[size]; // automatic array initialization.

};

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:

array<int, 15> t1; // array of 15 integers.

array<float, 10> t2; // array of 10 floats.

array<char, 4> t3; // array of 4 chars.

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

#include <iostream>

using namespace std;

template<class T, int size>

class A

{

public:

T arr[size];

void insert()

{

int i =1;

for (int j=0;j<size;j++)

{

arr[j] = i;

i++;

}

}

void display()

{

for(int i=0;i<size;i++)

{

std::cout << arr[i] << " ";

}

}

};

int main()

{

A<int,10> t1;

t1.insert();

t1.display();

return 0;

}

Output:

1 2 3 4 5 6 7 8 9 10

In the above example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

=========================================

#include<iostream>

using namespace std;

#include<string.h>

class Demo

{

int rollno;

int marks[5];

public:

void get()

{

cout<<"enter rollno\n";

cin>>rollno;

cout<<"enter marks\n";

for(int i=0;i<=4;i++)

{

cin>>marks[i];

}

}

void show()

{

cout<<" rollno"<<rollno;

cout<<"marks\n";

for(int i=0;i<=4;i++)

{

cout<<marks[i];

}

}

};

int main()

{

int n;

Demo d1[n];

d1.get();

d1.show();

}

======================================================

// C++ Program to Implement stack using Class Templates

#include <iostream>

#include <string>

using namespace std;

// Taking size of stack as 5

#define SIZE 5

// Class to represent stack using template

template <class T>

class Stack {

public:

Stack();

// Method 1: To add element to stack which can be any type using stack push() operation

void push(T k);

// Method 2

// To remove top element from stack using pop() operation

T pop();

// Method 3

// To print top element in stack using peek() method

T topElement();

// Method 4 To check whether stack is full using isFull() method

bool isFull();

// Method 5 To check whether stack is empty using isEmpty() method

bool isEmpty();

private:

int top;

T st[SIZE];

};

template <class T> Stack<T>::Stack()

{

top = -1;

}

template <class T>

void Stack<T>::push(T k)

{

if (isFull())

{

cout << "Stack is full\n";

}

// Inserted element

cout << "Inserted element " << k << endl;

// Incrementing the top by unity as element is to be inserted

top = top + 1;

// Now, adding element to stack

st[top] = k;

}

// Method 8

// To check if the stack is empty

template <class T>

bool Stack<T>::isEmpty()

{

if (top == -1)

return 1;

else

return 0;

}

// Utility methods

// Method 9

// To check if the stack is full or not

template <class T> bool Stack<T>::isFull()

{

// Till top in inside the stack

if (top == (SIZE - 1))

return 1;

else

// As top can not exceeds the size

return 0;

}

// Method 10

template <class T> T Stack<T>::pop()

{

// Initialising a variable to store popped element

T popped\_element = st[top];

// Decreasing the top as

// element is getting out from the stack

top--;

// Returning the element/s that is/are popped

return popped\_element;

}

// Method 11

template <class T> T Stack<T>::topElement()

{

// Initialising a variable to store top element

T top\_element = st[top];

// Returning the top element

return top\_element;

}

// Method 12

// Main driver method

int main()

{

// Creating object of Stack class in main() method

// Declaring objects of type Integer and String

Stack<int> integer\_stack;

Stack<string> string\_stack;

// Adding elements to integer stack object

// Custom integer entries

integer\_stack.push(2);

integer\_stack.push(54);

integer\_stack.push(255);

// Adding elements to string stack

// Custom string entries

string\_stack.push("Welcome");

string\_stack.push("to");

string\_stack.push("IET");

// Now, removing element from integer stack

cout << integer\_stack.pop() << " is removed from stack"

<< endl;

// Removing top element from string stack

cout << string\_stack.pop() << " is removed from stack "

<< endl;

// Print and display the top element in integer stack

cout << "Top element is " << integer\_stack.topElement()

<< endl;

// Print and display the top element in string stack

cout << "Top element is " << string\_stack.topElement()

<< endl;

return 0;

}=======================================================================================

Exception Handling

==>An exception is an unexpected problem that arises during the execution of a program our program terminates suddenly with some errors/issues. Exception occurs during the running of the program (runtime).

==>In C++, exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. The process of handling these exceptions is called exception handling.

Exception handling is a manner to handle the runtime error, we carry out exception handling, so, the normal flow of the program may be maintained even after runtime errors.

====================================================================================

Why do we need Exception Handling in C++?

The following are the main advantages of exception handling over traditional error handling:

1)Separation of Error Handling Code from Normal Code: There are always if-else conditions to handle errors in traditional error handling codes. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.

2)Grouping of Error Types: In C++, both basic types and objects can be thrown as exceptions.

3)Functions/Methods can handle only the exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

4)The Exception Handling mechanism offers a way to transfer control from one part of a program to another, This makes it clean to separate the mistake handling code from the code written to address the real functionality of the program.thats why detecting of errors becomes very easy

5)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.

====================================================================================

==========================================================================

Syntax of try-catch in C++

try {

// Code that might throw an exception

throw SomeExceptionType("Error message");

}

catch( ExceptionName e1 ) {

// catch block catches the exception that is thrown from try block

}

1. try in C++

The try keyword represents a block of code that may throw an exception placed inside the try block. It’s followed by one or more catch blocks. If an exception occurs, try block throws that exception.

2. catch in C++

The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.

3. throw in C++

An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.

Note: Multiple catch statements can be used to catch different type of exceptions thrown by try block.

==========================================================================================

// C++ program to show what happens when division by zero is done and exceptions are not handled

#include <iostream>

using namespace std;

int main()

{

int numerator = 5;

int denominator = 0;

int ans = numerator/denominator;

// The result will not be displayed

cout << ans << endl;

return 0;

}

===============================================================================

// C++ program to handle divide by zero exception

#include <iostream>

using namespace std;

int main()

{

int numerator = 5;

int denominator = 0;

int result;

try

{

if(denominator == 0)

{

throw denominator;

}

result = numerator / denominator;

}

catch(int ex)

{

cout << "Exception: Divide by zero not allowed :" <<

ex << endl;

}

cout << "Division Result is: " << result;

return 0;

}

================================================================

// C++ program to input an age integer if the age is less than 18 then return NO, but if the age is greater than or equal to 18 then return Yes

#include <iostream>

using namespace std;

int main()

{

try

{

int age = 10;

if (age >= 18)

{

cout << "YES, you are old enough.";

}

else

{

throw (age);

}

}

catch (int num)

{

cout << "No, You must be at least 18 years old" <<

endl;

cout << "Age is: " << num;

}

return 0;

}

Output

No, You must be at least 18 years old

Age is: 10

C++ Standard Exceptions

==>C++ defines a set of standard exceptions defined in <exception> which can be used in the programs. These exceptions are arranged in the parent-child class hierarchy.

==>==>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.

Below is the table listing the standard exceptions with description:

Exception Description

1. std::exception An exception and parent class of all the standard C++ exceptions.

2. std::bad\_alloc This exception can be thrown by new.

3. std::bad\_cast This exception can be thrown by dynamic\_cast.

4. std::bad\_typeid This can be thrown by type id.

==========================================================================================

==============================================================

Rethrowing an Exception:

In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using “throw; “.

Explain the concept of Rethrowing exceptions?

A rethrowing exception is a term used when we throw the exception again from one catch block to another. The exception is thrown towards another outside catch block.

#include <iostream>

using namespace std;

void help()

{

try {

throw 10;

}

catch (...) {

cout << "First throw called\n";

throw;

}

}

int main()

{

try {

help();

}

catch (...) {

cout << "Rethrowing throw called\n";

}

return 0;

}

Output

First throw called

Rethrowing throw called

=====================================================

Example

The following program shows try/catch blocks nesting.

CPP

// C++ program to demonstrate try/catch blocks can be nested

// in C++

#include <iostream>

using namespace std;

int main()

{

// nesting of try/catch

try {

try {

throw 20;

}

catch (int n) {

cout << "Handle Partially ";

throw; // Re-throwing an exception

}

}

catch (int n) {

cout << "Handle remaining ";

}

return 0;

}

Output

Handle Partially Handle remaining

A function can also re-throw a function using the same “throw; ” syntax. A function can handle a part and ask the caller to handle the remaining.

=================================================================

// C++ program to demonstate the use of try,catch and throw

// in exception handling.

#include <iostream>

using namespace std;

int main()

{

int x = -1;

// Some code

cout << "Before try \n";

// try block

try {

cout << "Inside try \n";

if (x < 0) {

// throwing an exception

throw x;

cout << "After throw (Never executed) \n";

}

}

// catch block

catch (int x) {

cout << "Exception Caught \n";

}

cout << "After catch (Will be executed) \n";

return 0;

}

======================================================================

Properties of Exception Handling in C++

Property 1

There is a special catch block called the ‘catch-all’ block, written as catch(…), that can be used to catch all types of exceptions.

Example

In the following program, an int is thrown as an exception, but there is no catch block for int, so the catch(…) block will be executed.

CPP

// C++ program to demonstate the use of catch all

// in exception handling.

#include <iostream>

using namespace std;

int main()

{

// try block

try {

// throw

throw 10;

}

// catch block

catch (char\* excp) {

cout << "Caught " << excp;

}

// catch all

catch (...) {

cout << "Default Exception\n";

}

return 0;

}

Output

Default Exception

====================================================================================

#include <iostream>

using namespace std;

int main()

{

try {

throw 10;

}

catch (...) {

cout << "default exception \n";

}

catch (int param) {

cout << "integer exception \n";

}

return 0;

}

Output:

An error will be raised

Explanation: The default catch block should be the last catch block or else other catch blocks will never be reached.

==========================================================

Inline Function

==>we can use several methods to optimize the code and enhance a program’s performance. For example, in the C language, the Macro function is used to reduce the program’s execution time. Since C++ is an extension of C, it also provides a function called the inline function, which can speed up a program by reducing the execution time.

==>C++ provides inline functions to reduce the function call overhead. An inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

==>Syntax:

inline return-type function-name(parameters)

{

// function code

}

==>Remember, inlining is only a request to the compiler, not a command. The compiler can ignore the request for inlining.

==>The compiler may not perform inlining in such circumstances as:

If a function contains a loop. (for, while and do-while)

If a function is recursive.

If a function contains a switch or nested loops

Why Inline Functions are Used?

1)When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack, and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register, and returns control to the calling function. This can become overhead if the execution time of the function is less than the switching time from the caller function to called function (callee).

For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function’s code. This overhead occurs for small functions because the execution time of a small function is less than the switching time.

==>When a function is invoked, it expands in line and is known as an inline function. When an inline function is invoked, its entire body of code is added or replaced at the inline function call location. At compile time, the C++ compiler makes this substitution.

2)The main use of the inline function in C++ is to save memory space. Whenever the function is called, then it takes a lot of time to execute the tasks, such as moving to the calling function. If the length of the function is small, then the substantial amount of execution time is spent in such overheads, and sometimes time taken required for moving to the calling function will be greater than the time taken required to execute that function.

The solution to this problem is to use macro definitions known as macros. The preprocessor macros are widely used in C, but the major drawback with the macros is that these are not normal functions which means the error checking process will not be done during the compilation.

C++ has provided one solution to this problem. In the case of function calling, the time for calling such small functions is huge, so to overcome such a problem, a new concept was introduced known as an inline function. When the function is encountered inside the main() method, it is expanded with its definition thus saving time.

Note:

Following are some key points that you need to keep in mind while dealing with inline functions:

Inline functions that are small have higher efficiency and better results than the lengthier inline functions. So, try to keep your inline functions small in length.

Although these functions increase the efficiency of the program and improve its execution, you should not convert all the functions into inline functions. If you convert larger functions to inline, it may lead to code bloat and reduce the functioning quality of the program.

Always try to define large functions outside the class, since functions defined inside a class are automatically defined as inline and this will affect the program negatively. You can use scope resolution (::) for this purpose.

Inline functions Advantages:

Function call overhead doesn’t occur.

It also saves the overhead of push/pop variables on the stack when a function is called.

It also saves the overhead of a return call from a function.

enhance the execution time and speed of your program.

The compilation speed of the program gets increased since the inline functions prevent function call overhead.

Inline function Disadvantages:

1)If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of the same code.

2)The inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because the compiler would be required to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

#include <iostream>

using namespace std;

inline int cube(int s)

{ return s \* s \* s;

}

int main()

{

cout << "The cube of 3 is: " << cube(3) << "\n";

return 0;

}

Output

The cube of 3 is: 27

=======================================================================

What is wrong with the macro?

There are some problems with the use of macros in C++.

Macro cannot access private members of the class.

the major drawback with the macros is that these are not normal functions which means the error checking process will not be done during the compilation.

One other thing is that the macros are managed by the preprocessor and inline functions are managed by the C++ compiler.

===========================================================================

=============================================================================

FAQ

Difference between Inline and Macro in C++

Macro : It is also called preprocessors directive. The macros are defined by the #define keyword. Before the program compilation, the preprocessor examines the program whenever the preprocessor detects the macros then preprocessor replaces the macro by the macro definition. Syntax of Macro:

#define MACRO\_NAME Macro\_definition

Example of Macro:

#include <iostream>

using namespace std;

// macro with parameter

#define MAXIMUM(a, b) (a > b) ? a : b

// Main function for the program

int main()

{

cout << "Max (100, 1000):";

int k = MAXIMUM(100, 1000);

cout << k << endl;

cout << "Max (20, 0):";

int k1 = MAXIMUM(20, 0);

cout << k1;

return 0;

}

Output:

Max (100, 1000):1000

Max (20, 0):20