Templates:

* Templates are the foundation of generic programming, which 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.
* There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how do they work:

Function Templates:

The general form of a template function definition is shown here:

template <class Typee> ret-Typee func\_name(parameter list)

{

// body of function

}

Here, Typee is a placeholder name for a data Typee used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:

#include <iostream>

#include <string>

using namespace std;

template <Typeename T>

inline T const& Max (T const& a, T const& b)

{

return a < b ? b:a;

}

int main ()

{

int i = 39;

int j = 20;

cout << "Max(i, j): " << Max(i, j) << endl;

double f1 = 13.5;

double f2 = 20.7;

cout << "Max(f1, f2): " << Max(f1, f2) << endl;

string s1 = "Hello";

string s2 = "World";

cout << "Max(s1, s2): " << Max(s1, s2) << endl;

return 0;

}

If we compile and run above code, this would produce the following result:

Max(i, j): 39\

Max(f1, f2): 20.7

Max(s1, s2): World

Class Template:

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

template <class Typee> class class\_name {

.

.

.

}

Here, **Typee** is the placeholder Typee name, which will be specified when a class is instantiated. You can define more than one generic data Typee by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

#include <iostream>

using namespace std;

template <class T>

class Stack {

private:

T stk[]; int top; // elements

public:

void push(T const&); // push element

T pop(); // pop element

T top() const; // return top element

bool empty() const{ // return true if empty.

return (top==-1);

}

};

template <class T>

void Stack<T>::push (T const& elem)

{

}

template <class T>

T Stack<T>::pop ()

{

if (is\_empty()) {

{

cout<<”empty stack";

// remove last element

Stk[top--];

}

template <class T>

T Stack<T>::top () const

{

if (elems.empty()) {

throw out\_of\_range("Stack<>::top(): empty stack");

}

// return copy of last element

return elems.back();

}

int main()

{

try {

Stack<int> intStack; // stack of ints

Stack<string> stringStack; // stack of strings

// manipulate int stack

intStack.push(7);

cout << intStack.top() <<endl;

// manipulate string stack

stringStack.push("hello");

cout << stringStack.top() << std::endl;

stringStack.pop();

stringStack.pop();

}

catch (exception const& ex) {

cerr << "Exception: " << ex.what() <<endl;

return -1;

}

}

|  |  |
| --- | --- |
| **// stack.h: header file**  **template< class Type, int MaxStack >**  **class Stack {**  **Type \* items ;**  **int top;**  **public:**  **Stack();**  **~Stack();**  **void push(Type);**  **Type pop();**  **int empty();**  **int full();**  **};**  **-------------------------------------**  **// stacktest.cpp: use templated stack**  **#include <iostream.h>**  **#include "stack.h"**  **int main() {**  **Stack<char, 10> s; // 10 chars**  **char ch;**  **while ((ch = cin.get()) != '\n')**  **if (!s.full()) s.push(ch);**  **while (!s.empty())**  **cout << s.pop();**  **cout << endl;**  **Stack<double, 4> ds; // 4 doubles**  **double d[] =**  **{1.0, 3.0, 5.0, 7.0, 9.0, 0.0};**  **int i = 0;**  **while (d[i] != 0.0 && !ds.full())**  **if (!ds.full())**  **ds.push(d[i++]);**  **while (!ds.empty())**  **cout << ds.pop() << " ";**  **cout << endl;**  **return 0;**  **}** | **// stack.cpp: function definitions**  **#include "stack.h"**  **template< class Type, int MaxStack >**  **Stack< Type, MaxStack >::Stack() {**  **Items=new Type[MaxStack];**  **top = -1;**  **}**  **template< class Type, int MaxStack >**  **Stack< Type, MaxStack >::~Stack() {**  **delete[] Items;**  **}**  **template< class Type, int MaxStack >**  **void Stack< Type, MaxStack> :push(Type c)**  **{ items[ ++top ] = c; }**  **template< class Type, int MaxStack >**  **Type Stack< Type, MaxStack >::pop()**  **{ return items[ top-- ]; }**  **template< class Type, int MaxStack >**  **int Stack< Type, MaxStack >::full()**  **{ return (top == MaxStack – 1); }**  **template< class Type, int MaxStack >**  **int Stack< Type, MaxStack >::empty()**  **{ return (top == -1); }** |