Templates in c++

-> Powerful feature in c++ which allows you to write generic programs. In simple terms you can create single function or a class to work with different data types using templates:

-> Concept of templates can be used in 2 different ways:

a. Function Templates

b. Class Templates

a. Function Templates

-> Similar to normal function, but it can work with different data types at once.

Example: Program to display largest of two numbers using function templates.

#include <iostream>

using namespace std;

// template function

template <class T>

T Large(T n1, T n2)

{

return (n1 > n2) ? n1 : n2;

}

int main()

{

int i1, i2;

float f1, f2;

char c1, c2;

cout << "Enter two integers:\n";

cin >> i1 >> i2;

cout << Large(i1, i2) <<" is larger." << endl;

cout << "\nEnter two floating-point numbers:\n";

cin >> f1 >> f2;

cout << Large(f1, f2) <<" is larger." << endl;

cout << "\nEnter two characters:\n";

cin >> c1 >> c2;

cout << Large(c1, c2) << " has larger ASCII value.";

return 0;

}

Output:

Enter two integers:

5

10

10 is larger.

Enter two floating-point numbers:

12.4

10.2

12.4 is larger.

Enter two characters:

z

Z

z has larger ASCII value.

b. Class Templates:

-> Like function templates, it is also possible to create class templates for generic class operations.

-> To create class template object, you need to define a data type inside a < > when creation.

ClassName<dataType> classObject

Example:

className<int> classObject;

className<float> classObject;

className<string> classObject;

Example: Simple calculator using class templates

#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()

{

cout << "Numbers are: " << num1 << " and " << num2 << "." << endl;

cout << "Addition is: " << add() << endl;

cout << "Subtraction is: " << subtract() << endl;

cout << "Product is: " << multiply() << endl;

cout << "Division is: " << divide() << endl;

}

T add() { return num1 + num2; }

T subtract() { return num1 - num2; }

T multiply() { return num1 \* num2; }

T divide() { return num1 / num2; }

};

int main()

{

Calculator<int> intCalc(2, 1);

Calculator<float> floatCalc(2.4, 1.2);

cout << "Int results:" << endl;

intCalc.displayResult();

cout << endl << "Float results:" << endl;

floatCalc.displayResult();

return 0;

}

Output:

Int results:

Numbers are: 2 and 1.

Addition is: 3

Subtraction is: 1

Product is: 2

Division is: 2

Float results:

Numbers are: 2.4 and 1.2.

Addition is: 3.6

Subtraction is: 1.2

Product is: 2.88

Division is: 2

Variadic Templates

-> Since we have understood what is template and then 2 types which are function templates and class templates, we will see about variadic templates.

-> Variadic templates are template that takes a variable number of arguments.

-> Variadic function templates are functions which can take multiple number of arguments.

-> Example: Here is the below example to display the use of variadic template.

// C++ program to demonstrate working of

// Variadic function Template

#include <iostream>

**using** **namespace** std;

// To handle base case of below recursive

// Variadic function Template

**void** print()

{

    cout << "I am empty function and "

            "I am called at last.\n" ;

}

// Variadic function Template that takes

// variable number of arguments and prints

// all of them.

**template** <**typename** T, **typename**... Types>

**void** print(T var1, Types... var2)

{

    cout << var1 << endl ;

    print(var2...) ;

}

// Driver code

**int** main()

{

    print(1, 2, 3.14, "Pass me any "

              "number of arguments",

                  "I will print\n");

**return** 0;

}

Output:

1

2

3.14

Pass me any number of arguments

I will print

I am empty function and I am called at last.

Need for Extern Templates, Extern Templates

-> Extern templates are used when we want to force the compiler to not instantiate (creating object) template when you know that it will be instantiated somewhere.

-> It is used to reduce compile time and object file size.

-> Let us understand extern template with an example.

Example 1: Without using extern

*// header.h*

*template<typename T>*

*void ReallyBigFunction()*

*{*

*// Body*

*}*

*// source1.cpp*

*#include "header.h"*

*void something1()*

*{*

*ReallyBigFunction<int>();*

*}*

*// source2.cpp*

*#include "header.h"*

*void something2()*

*{*

*ReallyBigFunction<int>();*

*}*

-> The above program creates 2 object files: source1.o and source2.o

*source1.o*

*void something1()*

*void ReallyBigFunction<int>() // Compiled first time*

*source2.o*

*void something2()*

*void ReallyBigFunction<int>() // Compiled second time*

-> When 2 files, source1.o and source2.o are linked together, one void ReallyBigFunction<int>() is discarded which leads to wasted compile time and object size.

-> To not waste compile time, there is need for extern keyword. This makes compile not compile a template function. This has to be used if and only if you know that it is used in same binary somewhere else.

-> Example 2: Using extern keyword

*// header.h*

*template<typename T>*

*void ReallyBigFunction()*

*{*

*// Body*

*}*

*// source1.cpp*

*#include "header.h"*

*void something1()*

*{*

*ReallyBigFunction<int>();*

*}*

*// source2.cpp*

*#include "header.h"*

*void something2()*

*{*

*ReallyBigFunction<int>();*

*}*

-> The above program creates 2 object files: source1.o and source2.o

*source1.o*

*void something1()*

*void ReallyBigFunction<int>() // compiled just one time*

*source2.o*

*void something2()*

*// No ReallyBigFunction<int> here because of the extern*

-> We can observe that code is compiled only once and second object file will just use the symbol from first file. No need to discard and no wasted compile time and object size.

STL Additions:

New Containers:

std::array

-> The new std::array is a container for constant size arrays. It's a sequential container class defined in <array> that specifies a fixed length array at compile time.

-> Syntax for std::array is given below:

T: std::array<int, 5> a = {1, 2, 3, 4, 5};

-> Example for std::array is given below:

/\* arr1.cpp \*/

#include <string>

#include <iterator>

#include <iostream>

#include <algorithm>

#include <array>

int main()

{

// construction uses aggregate initialization

std::array<int, 5> i\_array1{ {3, 4, 5, 1, 2} }; // double-braces required

std::array<int, 5> i\_array2 = {1, 2, 3, 4, 5}; // except after =

std::array<std::string, 2> string\_array = { {std::string("a"), "b"} };

std::cout << "Initial i\_array1 : ";

for(auto i: i\_array1)

std::cout << i << ' ';

// container operations are supported

std::sort(i\_array1.begin(), i\_array1.end());

std::cout << "\nsored i\_array1 : ";

for(auto i: i\_array1)

std::cout << i << ' ';

std::cout << "\nInitial i\_array2 : ";

for(auto i: i\_array2)

std::cout << i << ' ';

std::cout << "\nreversed i\_array2 : ";

std::reverse\_copy(i\_array2.begin(), i\_array2.end(),

std::ostream\_iterator<int>(std::cout, " "));

// ranged for loop is supported

std::cout << "\nstring\_array : ";

for(auto& s: string\_array)

std::cout << s << ' ';

return 0;

}

-> Output:

$ g++ -std=c++11 -o arr1 arr1.cpp

$ ./arr1

Initial i\_array1 : 3 4 5 1 2

sored i\_array1 : 1 2 3 4 5

Initial i\_array2 : 1 2 3 4 5

reversed i\_array2 : 5 4 3 2 1

std::forward\_list

-> It is basically linked list, which keeps track of only next element while list keeps track of both next and previous elements.

-> Drawback is it can only be iterated forward and cannot be iterated backward and individual elements cannot be accessed directly.

-> Can be used when only forward traversals are required.

-> Some of the operations on forward list are:

a. Assign(): Used to assign values to forward list, its another variant is used to assign repeated elements.

Example:

// C++ code to demonstrate forward list

// and assign()

#include<iostream>

#include<forward\_list>

**using** **namespace** std;

**int** main()

{

    // Declaring forward list

    forward\_list<**int**> flist1;

    // Declaring another forward list

    forward\_list<**int**> flist2;

    // Assigning values using assign()

    flist1.assign({1, 2, 3});

    // Assigning repeating values using assign()

    // 5 elements with value 10

    flist2.assign(5, 10);

    // Displaying forward lists

    cout << "The elements of first forward list are : ";

**for** (**int**&a : flist1)

        cout << a << " ";

    cout << endl;

    cout << "The elements of second forward list are : ";

**for** (**int**&b : flist2)

        cout << b << " ";

    cout << endl;

**return** 0;

}

Output:

The elements of first forward list are : 1 2 3

The elements of second forward list are : 10 10 10 10 10

b. push\_front(): Used to insert element at the first position on forward list. Value of this function is first copied to the space before the first element of container. Size of forward list increases by 1.

c. emplace\_front(): It is similar to previous function, but in this function, no copying operation occurs, the element is created directly at memory before the first element of the forward list.

d. pop\_front(): The function is used to delete the element which is in first.

e. insert\_after(): This function gives us choice to insert elements at any position in forward list. The arguments in this function are copied at desired position.

// Initializing forward list

    forward\_list<**int**> flist = {10, 20, 30};

    // Declaring a forward list iterator

    forward\_list<**int**>::iterator ptr;

    // Inserting value using insert\_after()

    // starts insertion from second position

    ptr =  flist.insert\_after(flist.begin(), {1, 2, 3});

-> Here {1,2,3} is inserted after 10.

Output: {10,1,2,3}

f. emplace\_after(): This function does the same except that it does not do any copy operation.

g. erase\_after: This function is used to erase elements from a particular function in a forward list.

Ptr = flist.erase\_after(ptr);

Output: 10,1,2,3,2,30( Here element 20 gets deleted which was present after 2).

h. remove(): Removes particular element from the forward list mentioned in the argument.

Eg: *forward\_list<****int****> flist = {10, 20, 30, 25, 40, 40};*

*// Removing element using remove()*

*// Removes all occurrences of 40*

*flist.****remove****(40);*

i. remove\_if(): This function removes according to the condition in its argument.

Eg: flist.remove\_if([](int x){ return x>20;});

-> This removes the elements in the list whose values are greater than 20.

unordered maps and sets

Unordered maps:

-> unordered\_map is an associated container that stores elements formed by combination of key value and a mapped value. The key value is used to uniquely identify the element and mapped value is the content associated with the key.

-> Example:

*// C++ program to demonstrate functionality of unordered\_map*

*#include <iostream>*

*#include <unordered\_map>*

*using namespace std;*

*int main()*

*{*

*// Declaring umap to be of <string, int> type*

*// key will be of string type and mapped value will*

*// be of double type*

*unordered\_map<string, int> umap;*

*// inserting values by using [] operator*

*umap["GeeksforGeeks"] = 10;*

*umap["Practice"] = 20;*

*umap["Contribute"] = 30;*

*// Traversing an unordered map*

*for (auto x : umap)*

*cout << x.first << " " << x.second << endl;*

*}*

Sets:

-> Sets are a part of the C++ STL. Sets are containers that store unique elements following a specific order.

-> Sets are a part of the C++ STL. Sets are containers that store unique elements following a specific order.

Added Operations emplace\_back, shrink\_to\_fit, data

New Algorithms std::for\_each

Tuples

Regular Expressions

New Containers:

std::array

std::unordered\_map

std::unordered\_set

std::unordered\_multimap

std::unordered\_multiset

std::forward\_list

std::vector vs std::array ❑ Dynamic vs Fixed size, Non-Type Template parameter for creation of array ❑ Unordered containers are based on hashing techniques (hash tables) whereas ordered containers (std::map, std::set etc.) are tree based

Additional container operations:

std::vector, std::list ==> push\_back(T&&)

std::vector ==> data, shrink\_to\_fit

most containers ==> T&&, std::initializer\_list with insert

most containers ==> emplace, emplace\_back

std::list ==> emplace\_front

std::map, std::set ==> emplace\_hint, try\_emplace

std::map ==> at

New Algorithms:

std::for\_each

std::copy\_if, std::copy\_n

std::std::move (algo)

all\_of, any\_of, none\_of

minmax, minmax\_element

std::iota, std::shuffle

std::is\_sorted, std::is\_sorted\_until

std::is\_heap, std::is\_heap\_until

std::is\_partitioned

std::partition\_copy, std::parition\_point

std::minmax, std::minmax\_element

Tuples

std:: regex usage:

Classes

std::basic\_regex

std::sub\_match

std::match\_results

Algorithms

std::regex\_match

std::regex\_search

std::regex\_replace

std::regex\_error

std::regex\_iterator

std::regex\_token\_iterator

Reference Wrapper

std::ref, std::cref

Smart Pointers:

Challenges with Raw pointers – Memory leaks, Heap Issues, Ownership – single/ multiple, transfer Lifetime control of heap objects Thread Safe Pointers Copy

on Write

std::unique\_ptr std::shared\_ptr std::make\_unique, std::make\_shared std::weak\_ptr

Polymorphic pointers, Default and Custom Deleters

Wrappers for creation std::make\_unique std::make\_shared

Type Casting std::dynamic\_pointer\_cast std::static\_pointer\_cast std::const\_pointer\_cast std::reinterpret\_pointer\_cast

Cyclic Redundancy problem & solution dynamic\_pointer\_cast, static\_pointer\_cast, const\_pointer\_cast, reinterpreter\_pointer\_cast