**Generics in C++**

Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces.

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. 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.

---------------------------------------------------------------------------------------------------------------------------------------

Generic Functions using Template:

We write a generic function that can be used for different data types. Examples of function templates are

sort(), max(), min(), printArray()

#include <iostream>

using namespace std;

// One function works for all data types.

// This would work even for user defined types

// if operator '>' is overloaded

template <typename 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;

}

---------------------------------------------------------------------------------------------------------------------------------------

Generic Class using Template:

Like function templates, class templates are useful when a class defines something that is independent of data type.

Can be useful for classes like LinkedList, binary tree, Stack, Queue, Array, etc.

Following is a simple example of template Array class.

#include <iostream>

using namespace std;

template <typename T>

class Array {

private:

T\* ptr;

int size;

public:

Array(T arr[], int s);

void print();

};

template <typename T>

Array<T>::Array(T arr[], int s)

{

ptr = new T[s];

size = s;

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

ptr[i] = arr[i];

}

template <typename T>

void Array<T>::print()

{

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

cout << " " << \*(ptr + i);

cout << endl;

}

int main()

{

int arr[5] = { 1, 2, 3, 4, 5 };

Array<int> a(arr, 5);

a.print();

return 0;

}

---------------------------------------------------------------------------------------------------------------------------------------

Working with multi-type Generics:

We can pass more than one data types as arguments to templates. The following example demonstrates the same.

#include <iostream>

using namespace std;

template <class T, class U>

class A {

T x;

U y;

public:

A()

{

cout << "Constructor Called" << endl;

}

};

int main()

{

A<char, char> a;

A<int, double> b;

return 0;

}

**Templates in C++**

---------------------------------------------------------------------------------------------------------------------------------------

A 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.

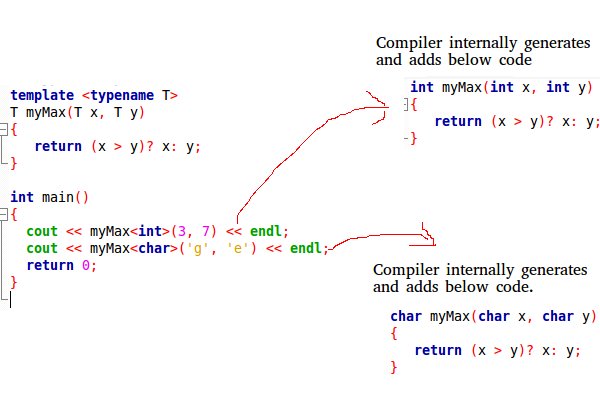
C++ adds two new keywords to support templates: ‘template’ and ‘typename’. The second keyword can always be replaced by keyword ‘class’.

---------------------------------------------------------------------------------------------------------------------------------------

How do templates work?

Templates are expanded at compiler time. This is like macros. The difference is, 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 same function/class.



---------------------------------------------------------------------------------------------------------------------------------------

Function Templates:

We write a generic function that can be used for different data types.

Examples of function templates are sort(), max(), min(), printArray().

#include <iostream>

using namespace std;

// One function works for all data types. This would work

// even for user defined types if operator '>' is overloaded

template <typename T>

T myMax(T x, T y)

{

return (x > y)? x: y;

}

int main()

{

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

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

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

return 0;

}

-----------------------------

// CPP code for bubble sort

// using template function

#include <iostream>

using namespace std;

// A template function to implement bubble sort.

// We can use this for any data type that supports

// comparison operator < and swap works for it.

template <class T>

void bubbleSort(T a[], int n) {

for (int i = 0; i < n - 1; i++)

for (int j = n - 1; i < j; j--)

if (a[j] < a[j - 1])

swap(a[j], a[j - 1]);

}

// Driver Code

int main() {

int a[5] = {10, 50, 30, 40, 20};

int n = sizeof(a) / sizeof(a[0]);

// calls template function

bubbleSort<int>(a, n);

cout << " Sorted array : ";

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

cout << a[i] << " ";

cout << endl;

return 0;

}

---------------------------------------------------------------------------------------------------------------------------------------

Class Templates:

Like function templates, class templates are useful when a class defines something that is independent of the data type.

Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

Following is a simple example of template Array class.

#include <iostream>

using namespace std;

template <typename T>

class Array {

private:

T \*ptr;

int size;

public:

Array(T arr[], int s);

void print();

};

template <typename T>

Array<T>::Array(T arr[], int s) {

ptr = new T[s];

size = s;

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

ptr[i] = arr[i];

}

template <typename T>

void Array<T>::print() {

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

cout<<" "<<\*(ptr + i);

cout<<endl;

}

int main() {

int arr[5] = {1, 2, 3, 4, 5};

Array<int> a(arr, 5);

a.print();

return 0;

}

---------------------------------------------------------------------------------------------------------------------------------------

Can there be more than one arguments to templates?

Yes, like normal parameters, we can pass more than one data types as arguments to templates.

The following example demonstrates the same.

#include<iostream>

using namespace std;

template<class T, class U>

class A {

T x;

U y;

public:

A() { cout<<"Constructor Called"<<endl; }

};

int main() {

A<char, char> a;

A<int, double> b;

return 0;

}

---------------------------------------------------------------------------------------------------------------------------------------

Can we specify default value for template arguments?

Yes, like normal parameters, we can specify default arguments to templates. The following example demonstrates the same.

#include<iostream>

using namespace std;

template<class T, class U = char>

class A {

public:

T x;

U y;

A() { cout<<"Constructor Called"<<endl; }

};

int main() {

A<char> a; // This will call A<char, char>

return 0;

}

---------------------------------------------------------------------------------------------------------------------------------------

What is the difference between function overloading and templates?

Both function overloading and templates are examples of polymorphism feature of OOP.

Function overloading is used when multiple functions do similar operations,

templates are used when multiple functions do identical operations.

---------------------------------------------------------------------------------------------------------------------------------------

What happens when there is a static member in a template class/function?

Each instance of a template contains its own static variable. See Templates and Static variables for more details.

---------------------------------------------------------------------------------------------------------------------------------------

What is template specialization?

Template specialization allows us to have different code for a particular data type. See Template Specialization for more details.

---------------------------------------------------------------------------------------------------------------------------------------

Can we pass non type parameters to templates?

We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant

value for a particular instance of a template. The important thing to note about non-type parameters is, they must be const.

The compiler must know the value of non-type parameters at compile time. Because the compiler needs to create functions/classes

for a specified non-type value at compile time. In below program, if we replace 10000 or 25 with a variable, we get a compiler error.

// A C++ program to demonstrate working of non-type

// parameters to templates in C++.

#include <iostream>

using namespace std;

template <class T, int max>

int arrMin(T arr[], int n)

{

int m = max;

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

if (arr[i] < m)

m = arr[i];

return m;

}

int main()

{

int arr1[] = {10, 20, 15, 12};

int n1 = sizeof(arr1)/sizeof(arr1[0]);

char arr2[] = {1, 2, 3};

int n2 = sizeof(arr2)/sizeof(arr2[0]);

// Second template parameter to arrMin must be a constant

cout << arrMin<int, 10000>(arr1, n1) << endl;

cout << arrMin<char, 256>(arr2, n2);

return 0;

}

---------------------------------------------------------------------------------------------------------------------------------------

**Templates & Static Variables**

---------------------------------------------------------------------------------------------------------

Function templates and static variables:

Each instantiation of function template has its own copy of local static variables. For example, in the following program there are

two instances: void fun(int ) and void fun(double ). So two copies of static variable i exist.

#include <iostream>

using namespace std;

template <typename T>

void fun(const T& x)

{

static int i = 10;

cout << ++i;

return;

}

int main()

{

fun<int>(1); // prints 11

cout << endl;

fun<int>(2); // prints 12

cout << endl;

fun<double>(1.1); // prints 11

cout << endl;

getchar();

return 0;

}

---------------------------------------------------------------------------------------------------------

Class templates and static variables:

The rule for class templates is same as function templates. Each instantiation of class template has its own copy of

member static variables. For example, in the following program there are two instances Test and Test.

So two copies of static variable count exist.

#include <iostream>

using namespace std;

template <class T> class Test

{

private:

T val;

public:

static int count;

Test()

{

count++;

}

// some other stuff in class

};

template<class T>

int Test<T>::count = 0;

int main()

{

Test<int> a; // value of count for Test<int> is 1 now

Test<int> b; // value of count for Test<int> is 2 now

Test<double> c; // value of count for Test<double> is 1 now

cout << Test<int>::count << endl; // prints 2

cout << Test<double>::count << endl; //prints 1

getchar();

return 0;

}

**Template Specialization**

---------------------------------------------------------------------------------------------------------------------------------------

Template in C++is a feature. We write code once and use it for any data type including user defined data types.

For example, sort() can be written and used to sort any data type items. A class stack can be created that can be used as a stack of

any data type.

What if we want a different code for a particular data type? Consider a big project that needs a function sort() for arrays of many

different data types. Let Quick Sort be used for all datatypes except char. In case of char, total possible values are 256 and counting

sort may be a better option. Is it possible to use different code only when sort() is called for char data type?

It is possible in C++ to get a special behavior for a particular data type. This is called template specialization.

--------------------------

// A generic sort function

template <class T>

void sort(T arr[], int size)

{

// code to implement Quick Sort

}

// Template Specialization: A function

// specialized for char data type

template <>

void sort<char>(char arr[], int size)

{

// code to implement counting sort

}

Another example could be a class Set that represents a set of elements and supports operations like union, intersection, etc.

When the type of elements is char, we may want to use a simple boolean array of size 256 to make a set. For other data types,

we have to use some other complex technique.

---------------------------------------------------------------------------------------------------------------------------------------

An Example Program for function template specialization:

Consider the following simple code where we have general template fun() for all data types except int.

For int, there is a specialized version of fun().

#include <iostream>

using namespace std;

template <class T>

void fun(T a)

{

cout << "The main template fun(): "

<< a << endl;

}

template<>

void fun(int a)

{

cout << "Specialized Template for int type: "

<< a << endl;

}

int main()

{

fun<char>('a');

fun<int>(10);

fun<float>(10.14);

}

---------------------------------------------------------------------------------------------------------------------------------------

An Example Program for class template specialization

In the following program, a specialized version of class Test is written for int data type.

#include <iostream>

using namespace std;

template <class T>

class Test

{

// Data members of test

public:

Test()

{

// Initialization of data members

cout << "General template object \n";

}

// Other methods of Test

};

template <>

class Test <int>

{

public:

Test()

{

// Initialization of data members

cout << "Specialized template object\n";

}

};

int main()

{

Test<int> a;

Test<char> b;

Test<float> c;

return 0;

}

---------------------------------------------------------------------------------------------------------------------------------------

How does template specialization work?

When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being

used for a new data type or new set of data types(in case of multiple template arguments).

If a specialized version is present, compiler first checks with the specialized version and then the main template.

Compiler first checks with the most specialized version by matching the passed parameter with the data type(s) specified in a

specialized version.