# Templates in C++

# Helping Manual

# C:\Users\Saad\AppData\Local\Microsoft\Windows\INetCache\Content.Word\download.png

Adopted from CS304 Handouts by VU

# 

[Templates in C++ 1](#_Toc17575394)

[Helping Manual 1](#_Toc17575395)

[Generic Programming 2](#_Toc17575396)

[Templates 2](#_Toc17575397)

[Function Templates 3](#_Toc17575398)

[Explicit Type Parameterization: 3](#_Toc17575399)

[Template Specializations 4](#_Toc17575400)

[Partial Specialization 5](#_Toc17575401)

[Multiple Type Arguments 5](#_Toc17575402)

[Class Template 5](#_Toc17575403)

# Generic Programming

Generic programming refers to programs containing generic abstractions (general code that is same in logic for all data types like printArray function), then we instantiate that generic program abstraction (function, class) for a particular data type, such abstractions can work with many different types of data.

**Advantages**

Major benefits of this approach are:

a. Reusability: Code can work for all data types.

b. Writability: Code takes lesser time to write.

c. Maintainability: Code is easy to maintain as changes are needed to be made in a single function or class instead of many functions or classes.

# Templates

In C++ generic programming is done using templates. Templates are of two kinds,

a. **Function Templates** (in case we want to write general function like printArray)

b. **Class Templates** (in case we want to write general class like Array class)

# Function Templates

A function template can be parameterized to operate on different types of data types.

**Declaration:**

We write template keyword above any function make any function as template function, they can be declared in any one of the following ways,

template< class T >

void funName(T x);

// OR

template< typename T >

void funName(T x);

// OR

template< class T, class U, … >

void funName(T x, U y, …);

template< typename T >

void printArray(T\* array, int size)

{

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

cout << array[i] <<","; // here data type of array is T

}

**Template function** will be instantiated for a particular data type according to passed argument as shown below**,**

int main() {

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

void printArray(iArray, 5); // Instantiated for int[] as passed array is of type int []

char cArray[3] = { ‘a’, ‘b’, ‘c’ };

void printArray(cArray, 3); // Instantiated for char[] as argument is of type

char[]

return 0;

}

## Explicit Type Parameterization:

In case a function template does not have any parameter then we have to explicitly mention the data type for which we want to create that function as shown below,

template <typename T>

T getInput() {

T x;

cin >> x;

return x;

}

int main() {

int x;

x = getInput(); // Error!

double y;

y = getInput(); // Error!

return 0;

}

int main() {

int x;

x = getInput< int >(); //will now work

double y;

y = getInput< double >(); //will now work

return 0;

}

## Template Specializations

A template compiler generated code may not handle all the types successfully; in that case we can give explicit specializations for a particular data type(s). For example suppose we have written a function isEqual(…, …) that compares two values of data type and return true or false depending upon the values are equal or not,

template< typename T >

bool isEqual(T x, T y) {

return (x == y);

}

*isEqual (6,6) should return true*

*isEqual (6,7) should return false*

*isEqual (6.6,6.6) should return true*

*isEqual (6.5,6.6) should return false*

*isEqual (‘A’,’A’) should return true*

*isEqual (‘A’,’a’) should return false*

Until here the function will work correctly but consider the statement below,

**isEqual (“abc”,”xyz”)**

This is instantiation of isEqual function for built in type char [] or char \*14, this function will fail to give correct result simply because we have given its implementation as

return (x == y); So here it will be translated by compiler in,

return ( char \* == char \*); or return ( char [] == char []);

So for char\* datatypes, we will specialize the function as:

template< >

bool isEqual< const char\* >(

const char\* x, const char\* y) {

return (strcmp(x, y) == 0);

}

## Partial Specialization

The above example is an example of “**complete specialization”**. Following partial specialization of this function deals with pointers to objects,

template< typename T >

bool isEqual( T\* x, T\* y ) {

return ( \*x == \*y );

}

## Multiple Type Arguments

template< typename T, typename U >

T my\_cast(U u) {

return (T)u; // U type will be converted to T type and will be returned

}

int main() {

double d = 10.5674;

int j = my\_cast(d); //Error

int i = my\_cast<int>(d); // need to explicity mention about type of T (int in this case) as it is used only For

//return type not as parameter

return 0;

}

# Class Template

Class template provides functionality to operate on different types of data and in this way facilitates reuse of classes.

We can definition a class template as follows:

• template< class T > class Xyz { … };

• template< typename T > class Xyz { … };