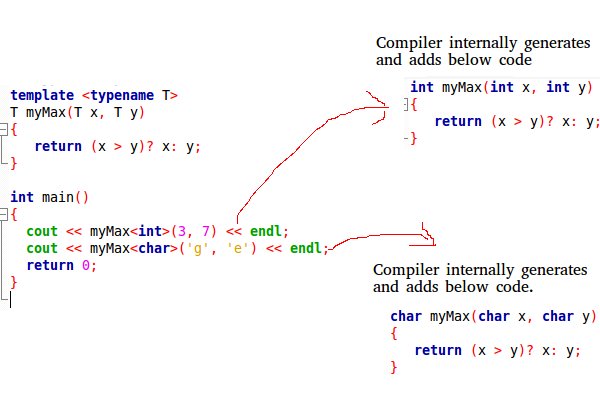
**Templates in C++**

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. 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. 



There are two ways we can implement templates:

* [Function Templates](https://programiz.com/cpp-programming/function-template)
* Class Templates

### Defining a Function Template

A function template starts with the keyword template followed by template parameter(s) inside <> which is followed by the function definition.

template <typename T>

T functionName(T parameter1, T parameter2, ...) {

// code

}

In the above code, T is a template argument that accepts different data types (int, float, etc.), and typename is a keyword.

When an argument of a data type is passed to functionName(), the compiler generates a new version of functionName() for the given data type.

Once we've declared and defined a function template, we can call it in other functions or templates (such as the main() function) with the following syntax

functionName<dataType>(parameter1, parameter2,...);

For example, let us consider a template that adds two numbers:

template <typename T>

T add(T num1, T num2) {

return (num1 + num2);

}

We can then call it in the main() function to add int and double numbers.

int main() {

int result1;

double result2;

// calling with int parameters

result1 = add<int>(2, 3);

cout << result1 << endl;

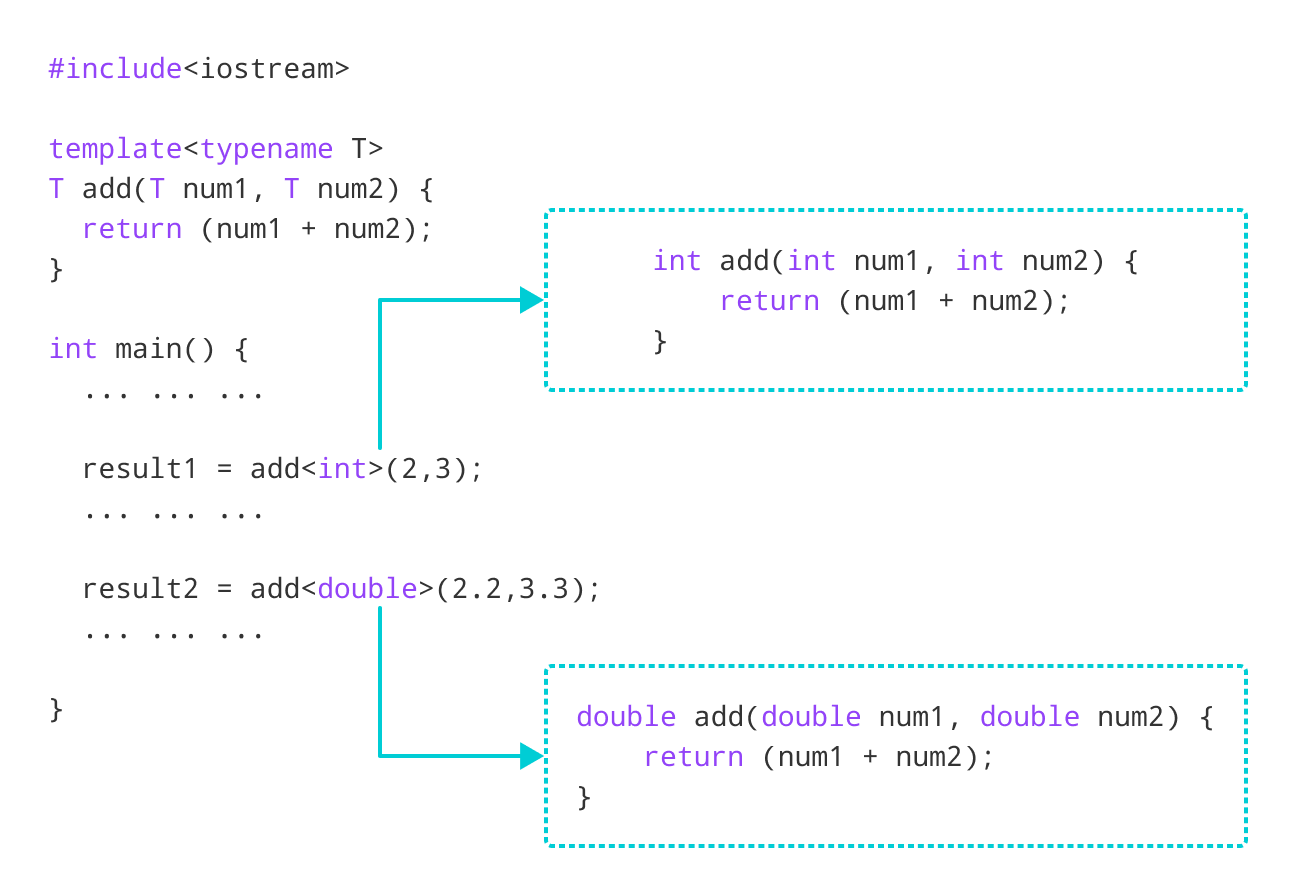
// calling with double parameters

result2 = add<double>(2.2, 3.3);

cout << result2 << endl;

return 0;

}



## Class Template Declaration

A class template starts with the keyword template followed by template parameter(s) inside <> which is followed by the class declaration.

template <class T>

class className {

private:

T var;

... .. ...

public:

T functionName(T arg);

... .. ...

};

In the above declaration, T is the template argument which is a placeholder for the data type used, and class is a keyword.

Inside the class body, a member variable var and a member function functionName() are both of type T.

## Creating a Class Template Object

Once we've declared and defined a class template, we can create its objects in other classes or functions (such as the main() function) with the following syntax

className<dataType> classObject;

For example,

className<int> classObject;

className<float> classObject;

className<string> classObject;

// C++ program to demonstrate the use of class templates

#include <iostream>

using namespace std;

// Class template

template <class T>

class Number {

private:

// Variable of type T

T num;

public:

Number(T n) : num(n) {} // constructor

T getNum() {

return num;

}

};

int main() {

// create object with int type

Number<int> numberInt(7);

// create object with double type

Number<double> numberDouble(7.7);

cout << "int Number = " << numberInt.getNum() << endl;

cout << "double Number = " << numberDouble.getNum() << endl;

return 0;

}

**Output**

int Number = 7

double Number = 7.7

In this program. we have created a class template Number with the code

template <class T>

class Number {

private:

T num;

public:

Number(T n) : num(n) {}

T getNum() { return num; }

};

Notice that the variable num, the constructor argument n, and the function getNum() are of type T, or have a return type T. That means that they can be of any type.

## Defining a Class Member Outside the Class Template

Suppose we need to define a function outside of the class template. We can do this with the following code:

template <class T>

class ClassName {

... .. ...

// Function prototype

returnType functionName();

};

// Function definition

template <class T>

returnType ClassName<T>::functionName() {

// code

}

Notice that the code template <class T> is repeated while defining the function outside of the class. This is necessary and is part of the syntax.

If we look at the code in **Example 1**, we have a function getNum() that is defined inside the class template Number.

We can define getNum() outside of Number with the following code:

template <class T>

class Number {

... .. ...

// Function prototype

T getnum();

};

// Function definition

template <class T>

T Number<T>::getNum() {

return num;

}

## C++ Class Templates With Multiple Parameters

In C++, we can use multiple template parameters and even use default arguments for those parameters. For example,

template <class T, class U, class V = int>

class ClassName {

private:

T member1;

U member2;

V member3;

... .. ...

public:

... .. ...

};

### Example 3: C++ Templates With Multiple Parameters

#include <iostream>

using namespace std;

// Class template with multiple and default parameters

template <class T, class U, class V = char>

class ClassTemplate {

private:

T var1;

U var2;

V var3;

public:

ClassTemplate(T v1, U v2, V v3) : var1(v1), var2(v2), var3(v3) {} // constructor

void printVar() {

cout << "var1 = " << var1 << endl;

cout << "var2 = " << var2 << endl;

cout << "var3 = " << var3 << endl;

}

};

int main() {

// create object with int, double and char types

ClassTemplate<int, double> obj1(7, 7.7, 'c');

cout << "obj1 values: " << endl;

obj1.printVar();

// create object with int, double and bool types

ClassTemplate<double, char, bool> obj2(8.8, 'a', false);

cout << "\nobj2 values: " << endl;

obj2.printVar();

return 0;

}

**Output**

obj1 values:

var1 = 7

var2 = 7.7

var3 = c

obj2 values:

var1 = 8.8

var2 = a

var3 = 0

In this program, we have created a class template, named ClassTemplate, with three parameters, with one of them being a default parameter.

template <class T, class U, class V = char>

class ClassTemplate {

// code

};

Notice the code class V = char. This means that V is a default parameter whose default type is char.

Inside ClassTemplate, we declare 3 variables var1, var2 and var3, each corresponding to one of the template parameters.

class ClassTemplate {

private:

T var1;

U var2;

V var3;

... .. ...

... .. ...

};

In main(), we create two objects of ClassTemplate with the code

// create object with int, double and char types

ClassTemplate<int, double> obj1(7, 7.7, 'c');

// create object with double, char and bool types

ClassTemplate<double, char, bool> obj2(8, 8.8, false);

Here,

|  |  |  |  |
| --- | --- | --- | --- |
| Object | T | U | V |
| obj1 | int | double | char |
| obj2 | double | char | bool |

For obj1, T = int, U = double and V = char.

For obj2, T = double, U = char and V = bool.

**Can we pass nontype 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. Please see [this](https://ide.geeksforgeeks.org/mgvysu).  
Below is a C++ program. 

* CPP

|  |
| --- |
| // 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;  } |

Output:

10

1