Templates are powerful features of C++ which allows us to write generic programs.

We can create a single function to work with different data types by using a template.

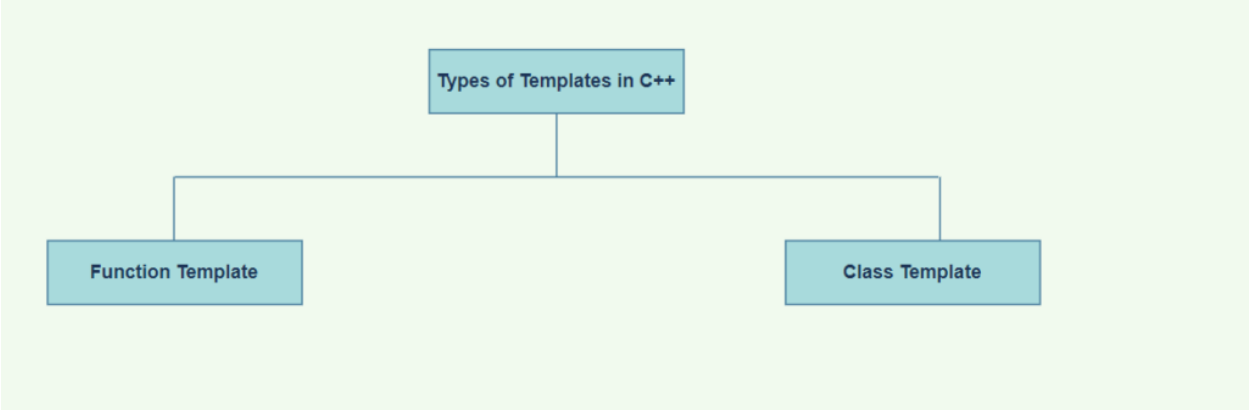
Templates in c++ is defined as a blueprint or formula for creating a generic class or a function. Generic Programming is an approach to programming where generic types are used as parameters in algorithms to work for a variety of data types.In C++, a template is a straightforward yet effective tool. To avoid having to write the same code for many data types, the simple concept is to pass the data type as a parameter.

Special functions that can interact with generic types are known as function templates. As a result, we can develop a function template whose functionality can be applied to multiple types or classes without having to duplicate the full code for each kind.

# Types of Templates in C++

There are two types of templates in C++

- **Function template**
- **Class templates**




# What is the function template in C++?

A function template in c++ is a single function template that works with multiple data types simultaneously, but a standard function works only with one set of data types.

## 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);
}
```

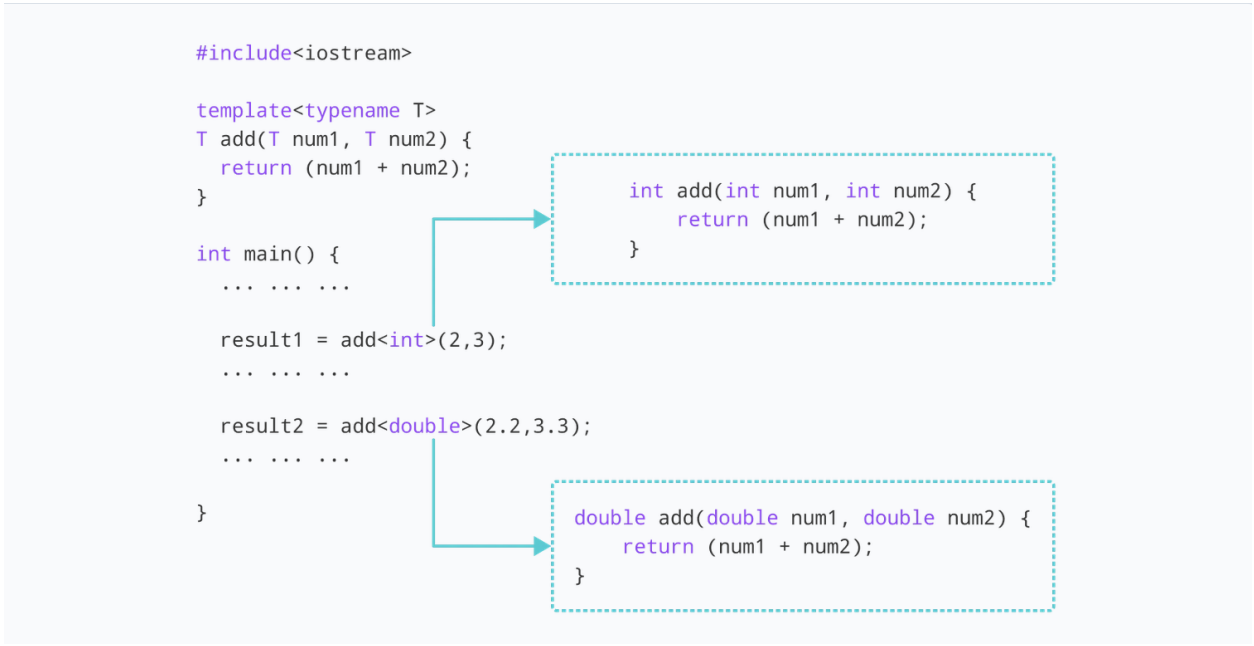
**We can then call it in the main() function to add int and double numbers.**

```
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;
}
```




```
#include <iostream>
using namespace std;

template <typename T>
T add(T num1, T num2) {
   return (num1 + num2);
}

int main() {
   int result1;
   double result2;
   // calling with int parameters
   result1 = add<int>(2, 3);
   cout << "2 + 3 = " << result1 << endl;

   // calling with double parameters
   result2 = add<double>(2.2, 3.3);
   cout << "2.2 + 3.3 = " << result2 << endl;

   return 0;
}
```

**Output**

```
2 + 3 = 5
```

2.2 + 3.3 = 5.5

Example:

```
#include <iostream>
using namespace std;
template<class X>//can replace 'class" keyword by "typename" keyword
X func( X a,X b)
{
return b;
}
int main(){
cout<<func(15,8)<<endl;//func(int,int);
cout<<func('p','q')<<endl;//func(char,char);
cout<<func(7.5,9.2)<<endl;//func(double,double)
        return 0;
}
```

### *Function Template with multiple Argument:*

```
#include <iostream>
using namespace std;
template<class X, class Y>
void display( X a,Y b)
{
        cout<<"a is = "<<a<<"\t"<<"b is = "<<b<<endl;
}
int main(){
        display(15,8);
        display('p',1);
        display(7,9.2);
        return 0;
}
```

### Class Template:

Similar to function templates, we can use class templates to create a single class to work with different data types.

Class templates come in handy as they can make our code shorter and more manageable.

# 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.

In main(), we have implemented the class template by creating its objects

Number<int> numberInt(7);

Number<double> numberDouble(7.7);

Notice the codes Number<int> and Number<double> in the code above.

This creates a class definition each for int and float, which are then used accordingly.

It is a good practice to specify the type when declaring objects of class templates.

# Simple Calculator Using Class Templates

This program uses a class template to perform addition, subtraction, multiplication and division of two variables num1 and num2.

The variables can be of any type, though we have only used int and float types in this example.

```
#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: " << num1 << " and " << num2 << "." << endl;
        cout << num1 << " + " << num2 << " = " << add() << endl;
        cout << num1 << " - " << num2 << " = " << subtract() << endl;
        cout << num1 << " * " << num2 << " = " << multiply() << endl;
        cout << num1 << " / " << num2 << " = " << 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: 2 and 1.
2 + 1 = 3
2 - 1 = 1
2 * 1 = 2
2 / 1 = 2

Float results:
Numbers: 2.4 and 1.2.
2.4 + 1.2 = 3.6
2.4 - 1.2 = 1.2
2.4 * 1.2 = 2.88
2.4 / 1.2 = 2
```

In the above program, we have declared a class template Calculator.

The class contains two private members of type T: num1 & num2, and a constructor to initialize the members.

We also have add(), subtract(), multiply(), and divide() functions that have the return type T. We also have a void function displayResult() that prints out the results of the other functions.

In main(), we have created two objects of Calculator: one for int data type and another for float data type.

```
Calculator<int> intCalc(2, 1);
Calculator<float> floatCalc(2.4, 1.2);
```

This prompts the compiler to create two class definitions for the respective data types during compilation.

## 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.

# C++ Exceptions

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

# Advantage

It maintains the normal flow of the application. In such cases, the rest of the code is executed even after exception.

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  return (x/y);
}
int main () {
  int i = 50;
  int j = 0;
  float k = 0;
        k = division(i, j);
        cout << k << endl;
  return 0;
}
```

Output:

Floating point exception (core dumped)

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an **exception** (throw an error).

# C++ try and catch

Exception handling in C++ consist of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

## Example

```
try {

  // Block of code to try

  throw exception; // Throw an exception when a problem arise

}

catch () {

  // Block of code to handle errors

}
```

```
#include <iostream>
using namespace std;

int main() {
  try {
          int age = 15;
          if (age >= 18) {
          cout << "Access granted - you are old enough.";
          } else {
          throw (age);
          }
  }
  catch (int myNum) {
          cout << "Access denied - You must be at least 18 years old.\n";
          cout << "Age is: " << myNum;
  }
  return 0;
}
```

Output:

```
Access denied - You must be at least 18 years old.
Age is: 15
```

We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error and do something about it. The catch statement takes a **parameter**: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be be greater than 18), the catch block is skipped:

```
#include <iostream>
using namespace std;

int main() {
  try {
        int age = 15;
        if (age >= 18) {
        cout << "Access granted - you are old enough.";
        } else {
        throw 505;
        }
  }
  catch (int myNum) {
        cout << "Access denied - You must be at least 18 years old.\n";
        cout << "Error number: " << myNum;
  }
  return 0;
}
```

OUTPUT:

```
Access denied - You must be at least 18 years old.
Error number: 505
```

# Handle Any Type of Exceptions (...)

If you do not know the throw **type** used in the try block, you can use the "three dots" syntax (...) inside the catch block, which will handle any type of exception:

```
#include <iostream>
using namespace std;

int main() {
  try {
        int age = 15;
        if (age >= 18) {
        cout << "Access granted - you are old enough.";
        } else {
        throw 505;
        }
  }
  catch (...) {
        cout << "Access denied - You must be at least 18 years old.\n";
  }
  return 0;
}
```

OUTPUT:

Access denied - You must be at least 18 years old.