Templates in C++

- We've looked at procedural programming
  –Reuse of *code* by packaging it into functions
- We've also looked at object-oriented programming
  –Reuse of *code and data* by packaging them into *classes*
- However, these techniques alone have limitations
  – Functions are still specific to the types they manipulate
    - E.g., swap (int, int) and swap (int *, int *) do essentially same thing
    - But, we must write two versions of swap to implement both
  – Classes alone are still specific to the types they contain
    - E.g., keep an array (not a vector) of dice and an array of players
  – Must write similar data structures and code repeatedly
    - E.g., adding a new element to an array

Generic programming aims to relieve these limitations

Templates are powerful features of C++ which allows us to write generic programs. There are two ways we can implement templates:

- Function Templates
- Class Templates

Function Templates/ Generic Functions

A generic function defines a general set of operations that will be applied to various types of data.

A generic function has the type of data that it will operate upon passed to it as a parameter.

Using this mechanism the same general procedure can be applied to a wide range of data.

Once we create the generic function the compiler automatically generates the correct code for the type of data that is actually used when we execute the function.

When we create a generic function we are creating a function that can automatically overload itself.

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.

Calling a Function Template

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);
}
```

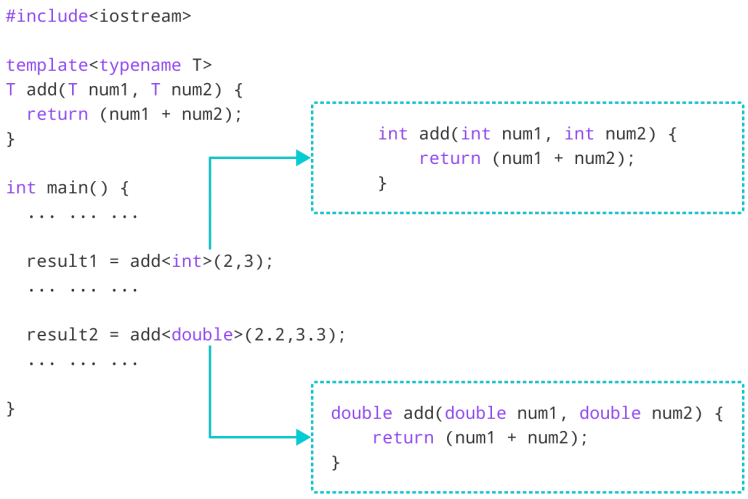
We can then call it in the `main()` function to add `int` and `double` numbers.

```
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;
}
```



Example: Adding Two Numbers Using Function Templates

```cpp
#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;
}
```

Source: https://www.programiz.com/cpp-programming/function-template

Class Templates

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;
```

Example 1: C++ Class Templates

```
// 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;
}
```

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.

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;
}
```

Example 2: 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: " << 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;
}
```

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++ 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;
}
```

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`.

Source: https://www.programiz.com/cpp-programming/class-templates