

# OOP

## Lecture 23

13/01/2021

### **Templates :**

A template is a blueprint or formula for creating a generic class or a function.

### **What is a generic class or function?**

Generic functions are functions declared with a generic data type parameter(s). It simply means that the function can accept parameters of any data type as parameters. They may be member functions in a class or struct, or standalone global functions. A single generic declaration of a class or a function implicitly/automatically declares a family of functions (many functions) that differ only in the substitution of a different actual data type for the generic data type parameter. When the generic function is called, the generic data type parameter(s) are replaced by the actual data type of the parameters passed.

Now, back to the templates. Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

### **Why do we need to define a template?**

As we have got the point that the templates are helpful for generic functions. Like, when we need to define the same logic for any type of parameters, we can define a template for that function that will accept any type of parameters. Do you remember any previous solution for the same problem? Yes, you got it right. We can do function overloading as well in this case, but do you want to copy the same code again and again and just change the data types of the parameters of the function? Still confused? 😞

## Function Templates :

Let's go through a very simple example, suppose you want to create an `add()` function that will add any three parameters passed to the function and return the sum. We can define our function like this,

```
int add(int x, int y, int z)
{
    int sum = x + y + z;
    return sum;
}
```

And now you want to create this function for float as well, in function overloading we will write,

```
float add(float x, float y, float z)
{
    float sum = x + y + z;
    return sum;
}
```

And the same for double, long, long long and long double as well.

```
double add(double x, double y, double z);
long double add(long double x, long double y, long double z);
```

Now, you can see that although it is solving the problem, it is not an efficient solution to it. To make best out of it we need to make this function generic, it means that we will write only one set of code and we can pass any type of parameters to it.

The important thing to remember is that we can make generic functions only when the logic of the function is the same for every type of parameters, if it is different with different parameters, like, we have to do different operations with float and int types, then we ultimately need to

overload the functions. Generic functions cannot help us with different types of logics.

### Syntax of Function Templates :

```
template <class Type>
return_type function_name( parameters )
{
    // Body of Functions
}
```

The bold words are keywords and should be written the same. The word Type inside angular brackets <> is the name we want to give to our generic data type. Any name can be used here.

Now, we are writing the same add() function which we defined earlier (unfortunately many times 😞) with the help of template as a generic function example,

```
template <class dataType>
dataType add( dataType x, dataType y, dataType z)
{
    dataType sum = x + y + z;
    return sum;
}
```

Now, this function is fully functional and generic, it can accept any type of arguments. And when the parameters are passed this function will act for the type the i.e word **dataType** will be replaced by **int** or **float** or **long** etc, depending upon which type of parameters are passed.

We can also define more than 1 type as generic by writing different names in the angular brackets. Like this,

```
template <class dataType1, class dataType2>
dataType2 add( dataType1 x, dataType1 y, dataType2 z);
```

But, the important thing is that all the types defined should exist in the parameter list so that it can be decided by the compiler based on parameters passed to the functions.

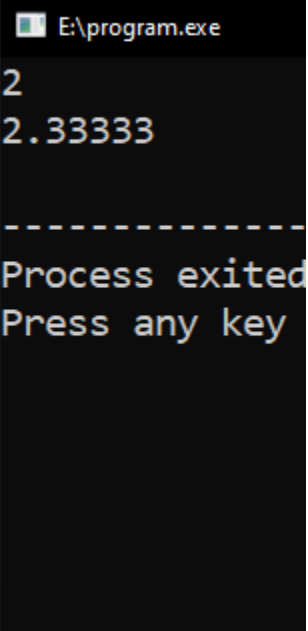
### Sample Programs : integer division vs float division

```
#include <iostream>

using namespace std;

template <class T>
T divide(T a, T b)
{
    return a / b;
}

int main()
{
    cout << divide (7,3) << '\n';
    cout << divide (7.0,3.0) << '\n';
    return 0;
}
```

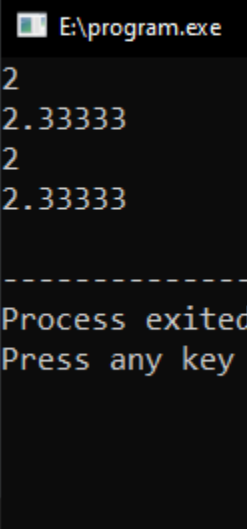


The screenshot shows the output of the program in a console window titled 'E:\program.exe'. It displays the results of integer and float division: '2' for 7/3 and '2.33333' for 7.0/3.0, separated by a dashed line. Below the output, it says 'Process exited' and 'Press any key'.

```
using namespace std;

template <class Type1, class Type2>
Type1 divide(Type1 a, Type2 b){
    return a/b;
}

int main(){
    cout << divide (7,3) << '\n';
    cout << divide (7.0,3) << '\n';
    cout << divide (7,3.0) << '\n';
    cout << divide (7.0,3.0) << '\n';
    return 0;
}
```



The screenshot shows the output of the program in a console window titled 'E:\program.exe'. It displays the results of integer and float division for four different cases: '2' for 7/3, '2.33333' for 7.0/3, '2' for 7/3.0, and '2.33333' for 7.0/3.0, separated by a dashed line. Below the output, it says 'Process exited' and 'Press any key'.

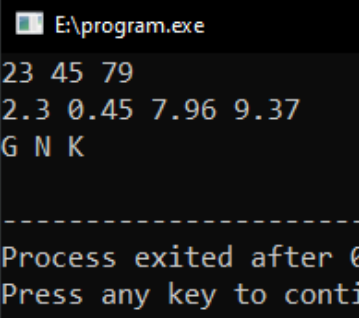
We can also write the prototypes of the functions and then we can write definitions of the function somewhere else in the program. But for this we need to write the template line again with the function. As follows,

### Sample Program :

```
template <class MyType>
void printArray(MyType*, const int); //Prototype

int main()
{
    int x[]={23,45,79};
    float f[]={2.3,0.45,7.96,9.37};
    char c[]={'G','N','K'};
    printArray(x,3);
    printArray(f,4);
    printArray(c,3);
    return 0;
}

template <class MyType> // FUnction Definition
void printArray(MyType *x, const int SIZE)
{
    for (int i=0;i<SIZE;i++)
        cout << x[i] << ' ';
    cout << '\n';
}
```



We can also pass user defined data types to this function. But remember the very important point stated earlier that the logic of the function should be same for that data types as well,

```
struct Point{
    int x,y;
    Point() { x = rand()%10; y=rand()%10; }
    friend ostream& operator << (ostream &out, const Point &p){
        out << p.x << ',' << p.y << ':';
        return out;
    }
};

template <class MyType>
void printArray(MyType*, const int);

int main(){
    srand(time(0));
    int x[]={23,45,79};
    float f[]={2.3,0.45,7.96,9.37};
    char c[]={'G','N','K'};
    Point p[6];
    printArray(x,3);
    printArray(f,4);
    printArray(c,3);
    printArray(p,6);
    return 0;
}

template <class MyType>
void printArray(MyType *x, const int SIZE){
    for (int i=0;i<SIZE;i++)
        cout << x[i] << ' ';
    cout << '\n';
}
```

```
E:\program.exe
23 45 79
2.3 0.45 7.96 9.37
G N K
1,1: 2,7: 0,4: 5,8: 2,9: 1,6:
-----
Process exited after 0.1562 seconds
Press any key to continue . . .
```

## Combining overloading and Templates :

Remember we said earlier that the logic of the program should remain the same for all the data types if we are creating a template. If the logic of the program is different from type to type then we have to do the function overloading. But consider a case, where we have to use different logic for one or two types and the rest of the types work with the same logic. Going with our first example of add() function. We created a template and the function will return the sum of 3 variables of any data type. Let's now we want to declare a different logic only for the **char** data. We want to set that every time when the parameters of **char** data types are passed, 48 should be subtracted from all the variables and then 48 should be added in the sum of the new values. Now it should be returned, and the other data types should work like normal with the same logic. Here, we need to overload the function with the char data type parameters. Like this,

```

template <class dataType>
dataType add( dataType x, dataType y, dataType z)
{
    dataType sum = x + y + z;
    return sum;
}
char add(char x, char y, char z){
    x -= 48; y -= 48; z -= 48;
    char sum = x + y + z;
    sum += 48;
    return sum;
}

```

Now, when we pass the **char** data type parameters, it will call our explicitly overloaded function, otherwise it will call the template function for rest of the data types.

Here's how the program works.

```

template <class dataType>
    dataType add( dataType x, dataType y, dataType z)
    {
        dataType sum = x + y + z;
        return sum;
    }

char add(char x, char y, char z)
{
    x -= 48; y -= 48; z -= 48;
    char sum = x + y + z;
    sum += 48;
    return sum;
}

int main()
{
    cout << add(3, 4, 5) << '\n';
    cout << add(3.5, 4.6, 5.3) << '\n';
    cout << add('3', '2', '3') << '\n';

    return 0;
}

```

```

E:\program.exe
12
13.4
8
-----
Process exited after
Press any key to cont

```

## Class Template :

Just like Functions, we can also define generic classes. In functions we write generic parameters, in classes we write generic data members and ultimately we can write generic member functions. In functions, we specify the data type by transferring the parameters to the functions, in classes we need to explicitly set the data type at the time of creation of the object of that class.

## Syntax of class Template :

```
template <class type>
class class_name{
    type member;
    int a, b
    -----
};
```

And while creating object in main() function we need to declare the object the like this,

```
class_name<datatype_name> obj_name;
```

It will replace the word type in class with the data type passed in the angular brackets <>.

## Class Activity :

Write a class **Stack** with three data members, integer pointer, size and position. Write 1 parametric constructor with default



parameter of size 10. Declare a dynamic array according to size, assign position to 0. Write the following functions :

1. bool isFull();
2. bool isEmpty();
3. void push(const int ELEMENT); Write value at position and increment
4. int pop(); return element from current position and decrement
5. int seeTop(); return element from current position don't decrement

**Solution :**

```
class Stack{
    int *x;
    int size, position;

public:
    Stack(int s=10)
    {
        x = new int[s];
        size=s;
        position=0;
    }

    bool isFull() const{    return position==size;    }

    bool isEmpty() const{return position==0;    }

    void push(const int ELEMENT)
    {
        if ( !isFull() )
            x[position++] = ELEMENT;
    }
    int pop()
    {
        if ( !isEmpty() )
            return x[--position];
    }
}
```

```

    }
    int seeTop() const
    {
        if ( !isEmpty() )
            return x[position-1];
    }

    ~Stack(){        delete []x;        }
};

```

Now we are going to convert it into a generic code which can work for any data type. All we have to do is just changing the data type of the pointer some parameters of the member functions to make it generic, check out this program,

```

template <class T>
class Stack{
    T *x;
    int size, position;
public:
    Stack(int s=10) {
        x = new T[s];
        size=s;
        position=0;
    }
    bool isFull() const{    return position==size;}
    bool isEmpty() const{    return position==0;}
    void push(const T ELEMENT) {
        if (!isFull())
            x[position++] = ELEMENT;
    }
    T pop() {
        if (!isEmpty())
            return x[--position];
    }
    T seeTop() const {
        if (!isEmpty())
            return x[position-1];
    }
    ~Stack(){        delete []x;        }
};

```

```

int main(){
    Stack<int> s(4);
    s.push(23);s.push(37);s.push(73);
    s.push(18);s.push(33);
    Stack<char> s1(5);
    s1.push('G');s1.push('T');s1.push(73);
    s1.push('K');s1.push('M');
}

```

E:\program.exe

18 73 37 23

M K I T G

-----  
Process exited af  
Press any key to

## Overloading Classes :

As we can overload functions along with the templates, we can also overload classes along with the template class to work with a specific data type with different logic. Consider the above example of Stack class, remember the struct Point we defined as our own data type? Go to page number 6 for the recap. Now, we want to overload our stack class for the Point data type. But as per the law we should change our logic to overcome the overloading we are going to do, as 1 template is fine for the same logic and we do not overload without applying different logic for a specific data type.

So, consider we want to create a double pointer of the Point class as data member and we will create the array of pointers in the constructor with specified size, default values is 10. The isFull() and isEmpty() functions will remain the same. The push() function will now accept a Point class object and the pointer at position index will start pointing to the passed object, The pop() function will return the object which is pointed by the pointer at position index.

```
template <>
class Stack<Point>{
    Point **x;
    int size, position;
public:
    Stack(int s=10) {
        cout << "Using specific class for Stack Point\n";
        x = new Point*[s];
        size=s;
        position=0;
    }
    bool isFull() const{    return position==size;}
    bool isEmpty() const{    return position==0;}
    void push(Point &ELEMENT) {
        if (!isFull())
            x[position++] = &ELEMENT; // Storing address of the ELEMENT at the position index (pointer)
    }
    Point& pop() {
        if (!isEmpty())
            return *x[--position]; // Returning the value of the pointer from position index, which is the Point object
    }
    Point& seeTop() const {
        if (!isEmpty())
            return *x[position-1];
    }
    ~Stack(){    delete []x;    }
};
```

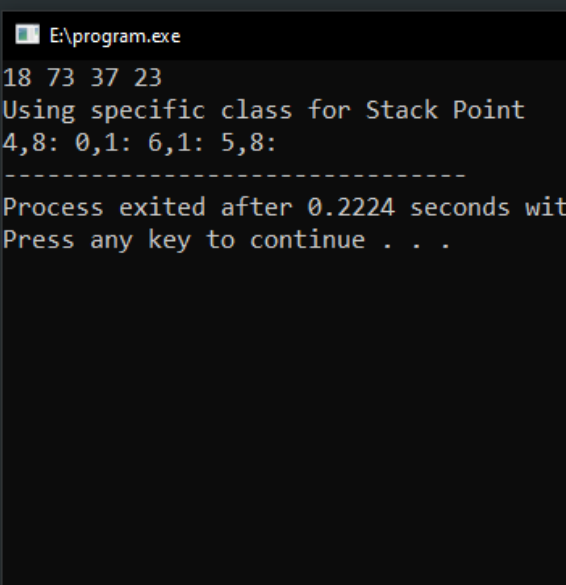
The struct Point and the template Stack class (used in previous code) is written above. This is the overloaded stack class code which is overloaded

only for the Point class objects, rest of the data types are dealt with by the template class.

## Output with main() function

```
int main(){
    srand(time(0));
    Stack<int> s(4);
    s.push(23);s.push(37);s.push(73);
    s.push(18);s.push(33);
    cout << s.pop() << ' ';
    cout << s.pop() << ' ';
    cout << s.pop() << ' ';
    cout << s.pop() << '\n';

    Stack<Point> s1(4);
    Point p[4];
    s1.push(p[0]);s1.push(p[1]);s1.push(p[2]);
    s1.push(p[3]);
    cout << s1.pop() << ' ';
    cout << s1.pop() << ' ';
    cout << s1.pop() << ' ';
    cout << s1.pop() << ' ';
    return 0;
}
```



***-The END-***