UTA 018: OOPs
Templates in C++

Introduction

Templates in C++ allow you to write **generic code** that can work with any data type. Instead of writing multiple versions of the same function or class for different data types, templates let you create **type-independent functions and classes** that automatically adapt to the type of data they work with.

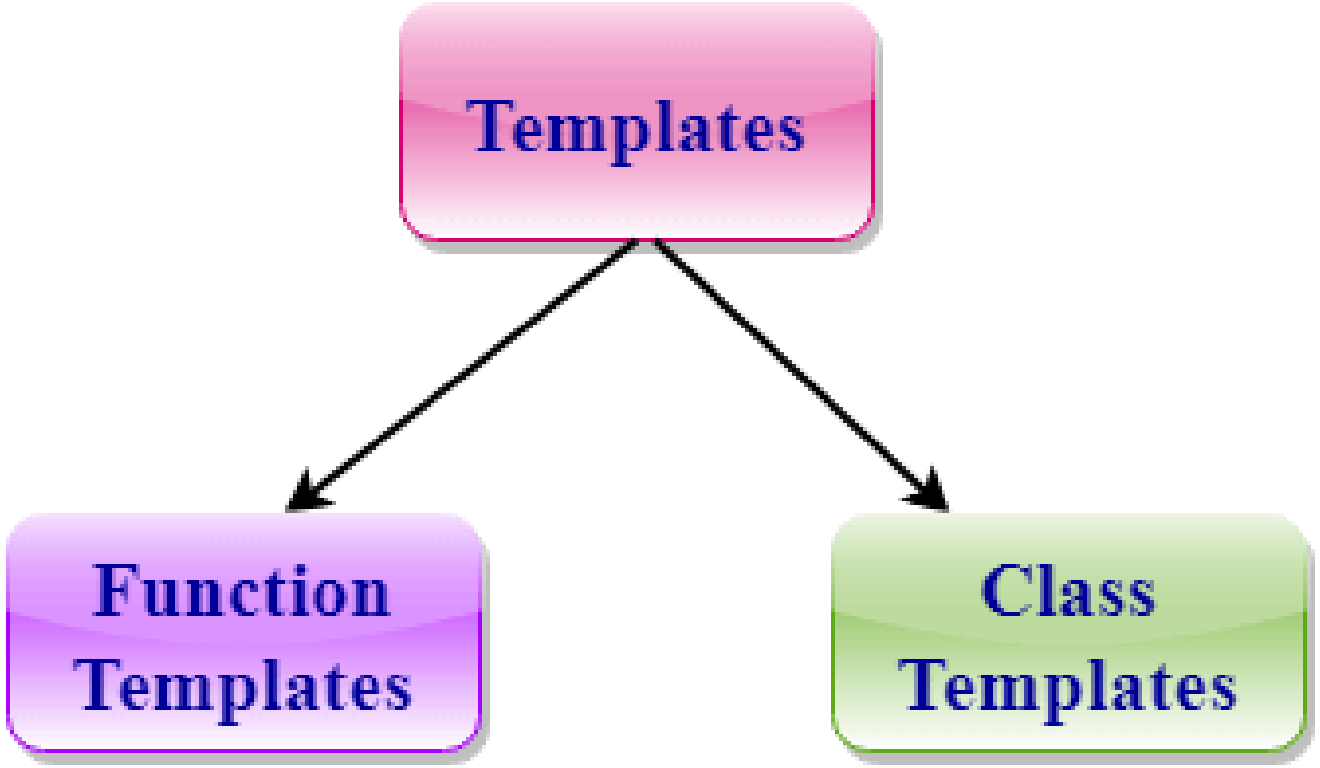
- Templates come in two main forms:
  - **Function Templates**: Used to create functions that can operate on any data type.
  - **Class Templates**: Used to create classes that can store, manipulate, or process data of any type.

Templates are a cornerstone of **generic programming** in C++, providing flexibility, reusability, and type safety.

Need of Template

- To reduce code *duplication* when supporting numerous data types

```
int MaxElement (int x, int y){ return x > y ? x : y ; }
float MaxElement (float x, float y){ return x > y ? x : y ; }
char MaxElement (char x, char y){ return x > y ? x : y ; }

T MaxElement (T x, T y){ return x>y?x:y;}
```



We can write a generic code for a function e.g. add() for integers, double, float etc.

We can write a generic code for a class, to manipulate group of member variables & functions e.g. linked list of strings, integers etc.

Function Template

A **template function** is a function that works with any data type. Instead of specifying a particular data type, you use a **placeholder type**. When the function is called, the compiler replaces the placeholder type with the actual data type of the argument(s).

Syntax of a Template Function:

```
template <class/typename T>
T functionName(T parameter1, T parameter2,…T Parameter_n)
{
        // Function implementation
}
```

Example:1

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

template <class T>
T MaxElement(T x, T y){ return x>y?x:y;}

int main(){
cout<<MaxElement (10,20)<<endl;
cout<<MaxElement ('a','z')<<endl;
cout<<MaxElement (-2.5,7.7)<<endl;
}
```

Defining Function Template

Calling Function Template

Example:2

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

template <typename T>
void swapValues(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 5, y = 10;
    double p = 3.5, q = 7.8;

    swapValues(x, y);     // Swaps two integers
    swapValues(p, q);     // Swaps two doubles

    cout << "Swapped integers: " << x << ", " << y << endl;
    cout << "Swapped doubles: " << p << ", " << q << endl;

    return 0;
}
```

Example: Bubble Sort without Template

```
void bubbleSort(int a[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (a[j] < a[j - 1])
              swap(a[j], a[j - 1]);
}
```

Example: Bubble Sort with Template

```
template <class T>
void bubbleSort(T arr[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (a[j] < a[j - 1])
              swap(a[j], a[j - 1]);
}
```

Function Template with Multiple Parameters

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

template <typename T1, typename T2>
void displayPair(T1 a, T2 b) {
    cout << "First: " << a << ", Second: " << b << endl;
}

int main() {
    displayPair(10, 15.5);      // Integer and double
    displayPair("Hello", 20);  // String and integer

    return 0;
}
```

Example: Write a template for

```
int main(){
show(100,"hello hello");
show('k',1500);
show(1.23,2987);
}
```

Example: Write a template for

```
template <class T1, class T2>
void show(T1 a, T2 b){
cout<<a<<“, “<<b<<endl;
}
```

```
int main(){
show(100,"hello hello");
show('k',1500);
show(1.23,2987);
}
```

Function template with non-type parameter

- Non-type parameter is not a type (datatype) but a value e.g. 100
- They are used to initialize a class or to specify the sizes of class members

- template <class T, int size> // size is the non-type parameter

Function template with non-type parameter

```
#include<iostream>
using namespace std;
template <class T, int size>
void show(T a){cout<<a<<", "<<size;}
int main(){
show <char,10> ('c');
}
```

Overloading function templates

```
template <class T1, class T2>
void show(T1 a, T2 b){
        cout<<a<<", "<<b<<endl;
}
void show(int a, int b)    {
        cout<<”For integer cases";
}

 int main(){
 show(100,"hello hello");
 show(3,3);
 }
```

Usage of Template Argument

- **No-argument template function**

```
Template <class T>
T void (void)                    //error: T is not used as an argument
{
//…..
Return parameter;
}
```

- **Template-type Argument Unused**

```
Template <class T>
Void test (int x)                    //error: T is not used as an argument
{
//…..
Return parameter;
}
```

- **Usage of partial Number of Template Argument**

```
Template <class T, class U>
Void insert (T &x)                    //error: U is not used as an argument
{
//…..
Return parameter;
}
```

Class Template

```
// Template class example
template <class/typename T> class Test {
T var;
public:
Test (T i) {var=i;}
T divideBy2 () {return var/2;}
};
int main(){
Test <int> t1(50);
Test <double> t2(-10.20);
cout<<t1.divideBy2()<<endl;
cout<<t2.divideBy2()<<endl;
}
```

```
// Template function  example
#include <iostream>
using namespace std;
template <class T>
T MaxElement(T x, T y){
        return x>y?x:y;
}
int main(){
cout<<MaxElement(10,20)<<endl;
cout<<MaxElement('a','z')<<endl;
}
```

Defining function outside the template class

```
template <class/typename T>
class Test {
T var;
public:
Test (T i) {var=i;}
T divideBy2 ();
};
template <class T>
T Test<T> :: divideBy2(){return var/2;}
```

template class with non-type parameter

```
template <class/typename T, int n>
class Test {
T var;
public:
Test () {var = n; cout<<"n = "<< n <<endl;}
T divideBy2 () {return var/2;}
};
int main(){
Test <int,10> t1;
Test <double,20> t2;
cout<<t1.divideBy2()<<" "<<t2.divideBy2()<<endl;
}
```

Output:

```
n = 10
n = 20
5 10
```

Class Template with Multiple Parameters

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

template <typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;
public:
    Pair(T1 a, T2 b) : first(a), second(b) {}

    void display() const {
        cout << "First: " << first << ", Second: " << second << endl;
    }
};

int main() {
    Pair<int, double> p1(10, 20.5);       // Pair of int and double
    Pair<string, char> p2("Hello", 'A'); // Pair of string and char

    p1.display();
    p2.display();

    return 0;
}
```

Advantages of Using Templates

- **Code Reusability**: Templates allow you to write code once and use it for different data types without duplication.
- **Type Safety**: The compiler enforces type checking at compile time, ensuring the correct data type is used.
- **Generic Programming**: Templates facilitate writing generic algorithms and data structures, making code more flexible and adaptable.

Template Specialization

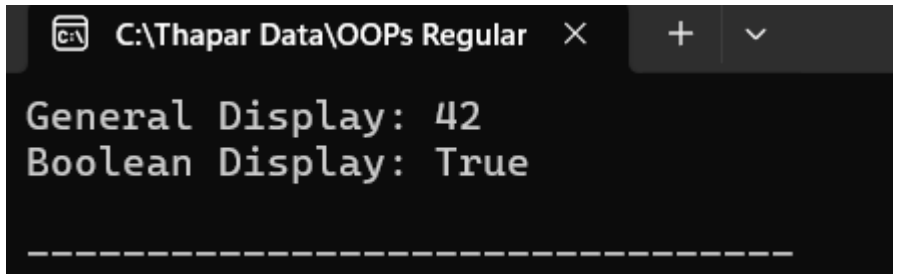
Template specialization allows you to define a specific implementation of a template function or class for a particular type. This is useful when you need specialized behavior for a certain data type, while still having a general template for other types.

Example of Class Template Specialization

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

template <typename T>
class Display {
public:
    void show(T value) {
        cout << "General Display: " << value << endl;
    }
};

// Specialization for `bool` type
template <>
class Display<bool> {
public:
    void show(bool value) {
        cout << "Boolean Display: " << (value ? "True" : "False") << endl;
    }
};
```

Example of Class Template Specialization

```
int main() {
    Display<int> intDisplay;
    intDisplay.show(42);    // Uses the general template

    Display<bool> boolDisplay;
    boolDisplay.show(true); // Uses the specialized template

    return 0;
}
```



- Display<int> uses the general Display template, while Display<bool> uses the specialized version specifically designed for bool.

- This feature is particularly useful when you need customized behavior for certain data types, such as printing true/false instead of 1/0.

Partial Specialization (Class Templates Only)

Partial specialization is only available for **class templates** (not function templates) and allows you to specialize a template for a subset of its template parameters.

Partial Specialization (Class Templates Only)

```
template <typename T, typename U>
class Pair {
public:
    void show() {
        cout << "General Pair" << endl;
    }
};
// Partial specialization when both types are the same
template <typename T>
class Pair<T, T> {
public:
    void show() {
        cout << "Specialized Pair for the same types" << endl;
    }
};
int main() {
    Pair<int, double> p1;
    p1.show();  // Uses general template

    Pair<int, int> p2;
    p2.show();  // Uses partially specialized template

    return 0;
}
```

Note:

Pair<int, double> uses the general Pair template, while Pair<int, int> uses the partially specialized version where both template parameters are the same.

Variadic Templates

- Variadic templates allow a template to accept **an arbitrary number of template parameters**. This is useful for functions like printf-style functions or containers that handle multiple elements of varying types.

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

template <typename T>
void print(T value) {
    cout << value << endl;
}

template <typename T, typename... Args>
void print(T value, Args... args) {
    cout << value << " ";
    print(args...);
    // Recursive call with reduced argument pack
}

int main() {
    print(1, 2.5, "Hello", 'A');
    print(5);
    print("OOPs");
    print("Todays Topic", "Function and Class Template");
    return 0;
}
```

Note:

- The first print function is the base case for recursion.

- The second print function is the recursive function that processes one argument at a time.

- Variadic templates enable flexible function calls with any number and type of arguments, making them powerful for generic programming.

```cpp
#include<iostream>
using namespace std;
struct A{
    int x,y;
};
struct B{
    int x;
    double y;
};
template <typename T>
void Assign_A(T a, T b, A&s1){
    s1.x=a;
    s1.y=b;
}
template<typename T, typename U>
void Assign_B(T a, U b, B&s2){
    s2.x=a;
    s2.y=b;
}
```

```cpp
int main(void){
    A s1;
    B s2;
    Assign_A(4, 5, s1);
    cout<<"Struct A:"<<s1.x<<", "<<s1.y<<endl;
    Assign_B(3.4, 4.3, s2);
    cout<<"Struct B:"<<s2.x<<", "<<s2.y<<endl;
    return 0;
}
```

```cpp
#include<iostream>
using namespace std;
template <class T>
class vector{
    T *v;
    int size;
    public:
        vector(int vector_size){
            size=vector_size;
            v= new T[size];
        }
        ~vector(){
            delete v;
        }
        T &elem(int i){
            if(i>=size)
                cout<<"error: out of range\n";
            return v[i];
        }
        void show();
};
template <class T>
void vector<T>::show(){
    for(int i=0;i<size;i++)
        cout<<elem(i)<<", ";
}
```

```cpp
int main(void){
    int i;
    vector<int> int_vect(5);
    vector<float> float_vect(4);
    for(i=0;i<5;i++)
        int_vect.elem(i)=i+1;
    for(i=0;i<4;i++)
        float_vect.elem(i)=float(i+1.5);
    cout<<"integer vector:"<<endl;
    int_vect.show();cout<<endl;

    cout<<"float vector:"<<endl;
    float_vect.show();
    return 0;
}
```

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

template<typename T>
int Search(T data[], T search_item, int low, int high){
    if(low>high)
        return -1;
    int mid=(low+high)/2;

    if(search_item<data[mid])
        return Search(data, search_item,low,mid-1);
    else
        if(search_item>data[mid])
            return Search(data,search_item,mid+1,high);
        return mid;
}
```

```cpp
int  main(void){
    int element, size, num[5],index;
    cout<<"How many elements:"<<endl;
    cin>>size;
    cout<<"enter the elements in ascending order:\n";
    for(int i=0;i<size;i++)
        cin>>num[i];
    cout<<"enter the elements to be searched:\n";
    cin>>element;
    index=Search(num,element,0,size);
    if(index==-1)
        cout<<"Element is not found\n";
    else
        cout<<"element foud at position:"<<index;
    return 0;
}
```

Inheritance of Class Template

Use of templates with respect to inheritance involves the following:

1. Derive a class template from a base class, which is a template class.
2. Derive a class template from a base class, which is a template class and add more template members in the derived class.
3. Derive a class from a base class which is not a template, and add template member to that class.
4. Derive a class from a base class which is a template class and restrict the template feature, so that the derived class and its derivatives do not have the template feature.

Syntax

```
template<class T>
class Base{
        //template type data and functions
};
template<class T1, …>
class Derive: public Base<T1, …>{
        //template type data and functions.
};
```

Example:01

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

// Base class template
template <typename T>
class Base {
public:
    T value;
    Base(T val) : value(val) {}
    void show() const {
        cout << "Base value: " << value << endl;
    }
};

// Derived class template inheriting from Base
template <typename T>
class Derived : public Base<T> {
public:
    Derived(T val) : Base<T>(val) {}
    void display() const {
        cout << "Derived value: " << this->value << endl;
    }
};
```

```cpp
int main() {
    Derived<int> obj(10);
    obj.show();     // Call Base class function
    obj.display();  // Call Derived class function

    return 0;
}
```

Example:02

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

// Base class template
template <typename T>
class Base {
public:
    T value;
    Base(T val) : value(val) {}
    void show() const {
        cout << "Base value: " << value << endl;
    }
};

// Derived class template with two template parameters
template <typename T, typename U>
class Derived : public Base<T> {
public:
    U extraValue;
    Derived(T val, U extra) : Base<T>(val), extraValue(extra) {}

    void display() const {
        cout << "Derived value: " << this->value << ", Extra value: " << extraValue << endl;
    }
};
```

```cpp
int main() {
    Derived<int, double> obj(10, 20.5);
    obj.show();      // Call Base class function
    obj.display();   // Call Derived class function

    return 0;
}
```

Example:03

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

// Non-template base class
class Base {
public:
    void show() const {
        cout << "Base class show function" << endl;
    }
};

// Derived class with a template member function
class Derived : public Base {
public:
    template <typename T>
    void display(T value) const {
        cout << "Derived class display with value: " << value << endl;
    }
};

int main() {
    Derived obj;
    obj.show();          // Call Base class function
    obj.display(10);     // Call Derived class template function with an int
    obj.display(3.14);   // Call Derived class template function with a double

    return 0;
}
```

Example:04

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

// Base class template
template <typename T>
class Base {
public:
    T value;
    Base(T val) : value(val) {}
    void show() const {
        cout << "Base value: " << value << endl;
    }
};

// Non-template Derived class inheriting from a specific instantiation of Base
class Derived : public Base<int> {  // `int` is specified, so Derived is not a template
public:
    Derived(int val) : Base<int>(val) {}
    void display() const {
        cout << "Derived class with fixed int type: " << this->value << endl;
    }
};

int main() {
    Derived obj(10);
    obj.show();       // Call Base class function
    obj.display();    // Call Derived class function

    return 0;
}
```

Thank You