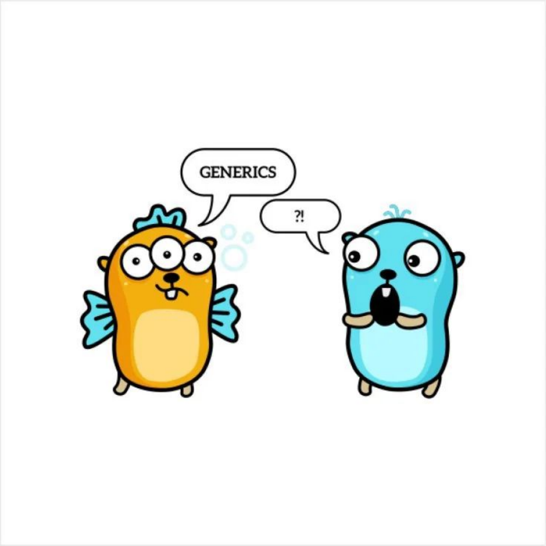
Generic Programming



Generic Programming

Generic Programming is the idea to allow type (Integer, String, ... etc and user-defined types) to be a parameter to methods, classes and interfaces.

The method of Generic Programming is implemented to increase the efficiency of the code.

Generic Programming enables the programmer to write a general algorithm which will work with all data types.

Advantages

- Code Reusability
- Avoid Function Overloading
- Once written it can be used for multiple times and cases

Templates

Generics can be implemented in C++ using Templates.

Templates!

Function Templates


```
template <class T>
ret-type function-name(parameters)
{
// body of function
}
```

T is a placeholder that the compiler will automatically replace with an actual data type.

Function Templates

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

//template <typename T> // you can write any one of them
template <class T>


T findMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {

    int intA = 5, intB = 10;
    cout << "Max integer value: " << findMax(intA, intB) << endl;

    double doubleA = 3.5, doubleB = 7.2;
    cout << "Max double value: " << findMax(doubleA, doubleB) << endl;

    cout << "Max Char value: " << findMax('A', 'a') << endl;
    return 0;
}
```

Class Activity

Write a generic function to swap two variables.

```cpp
template <class T>
void swapargs(T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

```cpp
int main()
{
        int i=10;
        int j=20;
        double x=10.1;
        double y=23.3;
        char a='x';
        char b='z';

        swapargs(i, j); // swap integers
        swapargs(x, y); // swap floats
        swapargs(a, b); // swap chars

        cout<<"i:   "<<i<<endl;
        cout<<"j:   "<<j<<endl;
        cout<<"x:   "<<x<<endl;
        cout<<"y:   "<<y<<endl;
        cout<<"a:   "<<a<<endl;
        cout<<"b:   "<<b<<endl;
}
```

C:\Users\basit.jasani\Desktop\Untitled2.exe

```
i:      20
j:      10
x:      23.3
y:      10.1
a:      z
b:      x
```

Template Function with Two Generic Types

You can define more than one generic data type in the template statement by using a comma-separated list

```
template <class T1, class T2>
  void myfunc(T1 a, T2 b)
  {
   cout << a << “  &  ” << b << '\n';
  }
```

Specialized Template

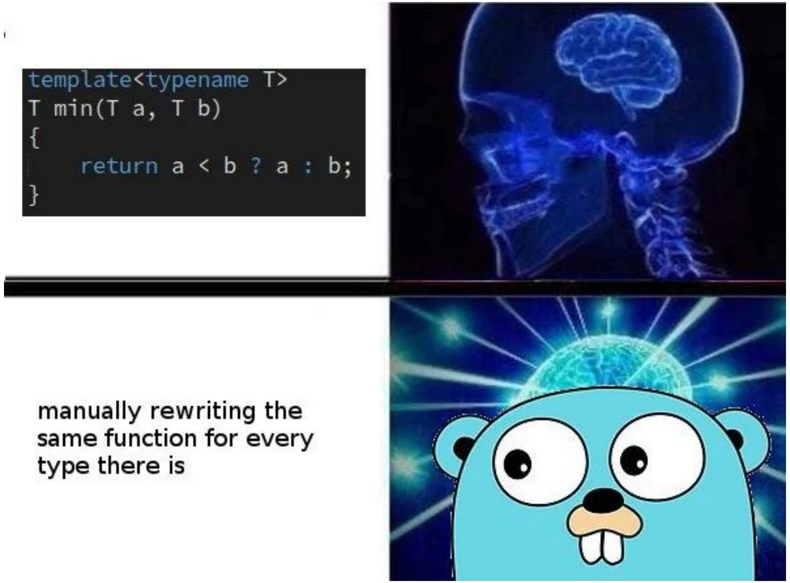
```
#include <iostream>
using namespace std;
template<class T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}
template<>
int findMax(int a, int b) {
    cout<<"I am template for int only"<<endl;
    return (a > b) ? a : b;
}
```

```
int main() {
    int intA = 5, intB = 10;
    cout << "Max integer value: " <<
findMax(intA, intB) << endl;
    double doubleA = 3.5, doubleB = 7.2;
    cout << "Max double value: " <<
findMax(doubleA, doubleB) << endl;
    cout << "Max Char value: " << findMax('A',
'a') << endl;
    return 0;
}
```

Specialized Template

```
#include <iostream>
using namespace std;
template<class T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}
template<>
int findMax(int a, int b) {
    cout<<"I am template for
      int only"<<endl;
    return (a > b) ? a : b;
}
```

```
int main() {
     cout<<findMax<int>(10,5) <<endl;
     cout<<findMax <char>('A', 'a') << endl;
    return 0;
}
```



Overloading a Generic Function

In addition to creating explicit, overloaded versions of a generic function, you can also overload the template specification itself

To do so, simply create another version of the template that differs from any others in its parameter list

```
// First version of f() template
   template

template <class X>

void f(X a)

{

    cout << "Inside f(X a)";
   b)";

}
```

```
// Second version of f()

template <class X, class Y>

void f(X a, Y b)

    {

    cout << "Inside f(X a, Y

    }
```

Using Normal Parameters in Generic Functions

You can mix non-generic parameters with generic parameters in a template function:

```
template<class X> void func(X a, int b){
        cout << “General Data:  ” << a;
        cout << “Integer Data:  ”  << b;
}
```

Generic Classes

The actual type of the data being used (in class) will be specified as a parameter when objects of that class are created.

Generic classes are useful when a class uses logic that can be generalized e.g. Stacks, Queues

```
template <class T> class class-name
{
. . .
}
```

Generic Classes

If necessary, we can define more than one generic data type using a comma-separated list

We create a specific instance of that class using the following general form:

```
class-name <type> ob;
```

Generic Class

```
#include <iostream>
using namespace std;
template <class T1, class T2>
class myclass {
  T1 i;
  T2 j;
public:
  myclass (T1 a, T2 b) { i = a; j = b; }
  void show( ) { cout << i << “ & ” << j; }
  T1 getmax();
};

template<class T1, class T2>
T1 myclass<T1, T2>::getmax() {
    return i > j ? i : j;
}
```

```
int main(){
  myclass<int, double> ob1(10, 0.23);
  myclass<char, char *> ob2('X',
"Hello");

  ob1.show(); // show int, double
  ob2.show(); // show char, char *
}
```

Generic Classes

In a generic class, we can also specify non-type arguments:

```
template <class T, int size>
class MyClass
{
T arr[size]; // length of array is passed in size
// rest of the code in class
}
int main()
{
MyClass<int, 10> intob;
MyClass<double, 15> doubleob;
}
```

Generic Base Classes & Derived Classes

```
template <class T>
class Base
{

};
template <class U, class T>
class Derived:public Base <T>
{};
class Derived:public Base <int>
{};
```

Generic Class

```cpp
#include <iostream>
using namespace std;
template<typename T>
class Base {
protected:
    T value;
public:
    Base(const T& val) : value(val) {}
    void display() {
        cout << "Value in base class: " << value << endl;
    }
};

template<typename T>
class Derived : public Base<T> {
public:
    Derived(const T& val) : Base<T>(val) {}
    void display() {
        cout << "Value in derived class: " << this->value << endl;
    }
};
```

```cpp
int main() {
    Base<int> baseObj(5);
    Derived<double> derivedObj(3.14);

    baseObj.display();     // Output: Base class: 5
    derivedObj.display(); // Output: Derived class:
3.14
    return 0;
}
```

Class Activity

Write a template class to manage an array of different data types showing behaviour of stack. The class must have following functions.

Push : when you push a variable onto the stack, it gets added to the top of the stack.

Pop: when you pop an integer from the stack, you remove the top integer from the stack.

Peek: when you peek at the stack, you can see the integer that is currently at the top of the stack, but it remains on the stack.

Stack Class

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

template<typename T>
class Stack {
private:
    static const int MAX_SIZE = 100;
    T elements[MAX_SIZE];
    int topIndex;

public:
    Stack() : topIndex(-1) {}
```

Stack Class

```
void push(const T& item) {
        if (topIndex == MAX_SIZE - 1) {
            cout << "Error: Stack is full" << endl;
            return;
        }
        elements[++topIndex] = item;
    }
```

Stack Class

```
T pop() {
        if (topIndex == -1) {
            cout << "Error: Stack is empty" << endl;
        }
        return elements[topIndex--];
    }
```

Stack Class

```
T peek() const {
        if (topIndex == -1) {
            cout << "Error: Stack is empty" << endl;
        }
        return elements[topIndex];
    }
```

Stack Class

```
 int main() {
    Stack<int> intStack;

    intStack.push(10);
    intStack.push(20);
    intStack.push(30);

    cout << "Top element: " << intStack.peek() << endl;

    int popped = intStack.pop();
    cout << "Popped element: " << popped << endl;
    return 0;
}
```

Class Activity

Write a template class to manage an array of different data types showing behaviour of queue. The class must have following functions.

Enqueue: the enqueue operation adds an element to the back (or end) of the queue.
Dequeue: the dequeue operation removes and returns the element at the front of the queue.

Front: the front function returns the element at the front of the queue without removing it.

Queue Class

```
void enqueue(T value) {
        if (count == SIZE) {
            cout << "Queue is full\n";
            return;
        }
        rearIndex = (rearIndex + 1) % SIZE;
        arr[rearIndex] = value;
        count++;
    }
```

Queue Class

```
T dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty\n";
            return T();  // Return default value
        }
        T value = arr[frontIndex];
        frontIndex = (frontIndex + 1) % SIZE;
        count--;
        return value;
    }
```

A. For the given class, you are required to create a specialized template that manages computations specifically when both arrays are characters with a size of 10. Overload the function so that it returns a string containing all elements of arr1 followed by all elements of arr2.

```
template <class T, int size>
class QuestionTemplate {
    T arr1[size];
    T arr2[size];
public:
    QuestionTemplate() {
        //assume numbers only for now
        for (int i = 0; i < size; i++){
            arr1[i] = i;
            arr2[size - i - 1] = i;
        }
    }
```

```
    T* add() {
        T* arr = new T[size];
        for (int i = 0; i < size; i++)
            arr[i] = arr1[i] + arr2[i];
        return arr;
    }
};

int main() {
    QuestionTemplate <int, 10> qt;
    int* res = qt.add();
    for (int i = 0; i < 10; i++)
        cout << res[i] << endl;
    QuestionTemplate <char, 10> ct;
    cout << ct.add();
}
```

```
//---- start code completion -----
template <>
class QuestionTemplate <char, 10> {
    char arr1[10];
    char arr2[10];
public:
    QuestionTemplate() {
        char c = 'a';
        for (int i = 0; i < 10; i++) {
            arr1[i] = c + i;
            arr2[9 - i] = c + i;
        }
    }
    string add() {
        string str = "";
        for (int i = 0; i < 10; i++)
            str += arr1[i];
        for (int i = 0; i < 10; i++)
            str += arr2[i];

        return str;
    }
};
//---- finish code completion -----
```

Class Activity

Create a base class called Course that contains common properties and methods for all courses. The class has attributes such as name, course_code, credithours, and instructor. You define methods such as print_details() which will be override in the derived class.

Next, you create several specific course classes that inherit from the Course class. For example, you create a ThoeryCourse class that has additional attributes such as projects and mid1 and mid2 and final marks, and a LabCourse class that has attributes such as lab_tasks and lab_mid and lab_final marks. Both classes have get_grade() function which generates grades based on their evaluation criteria.

Class Activity

Then, you create a generic function called display_grade() that takes any Course object either TheoryCourse or LabCourse as an argument and calls the get_grade() function.

Define a generic filter_courses() function to filter courses by field value. It takes an array of courses, a second parameter indicating the field to filter by (e.g., “instructor” or credit hours). The function should call print detail functions for only those courses that match the specified value.