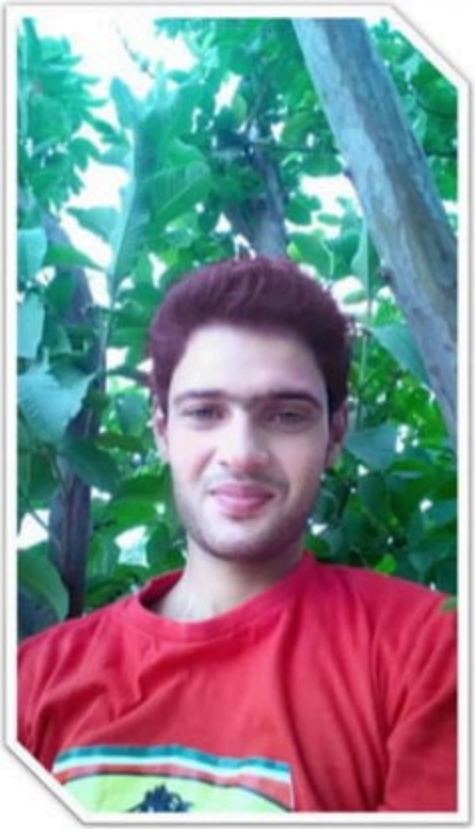
Templates in C++

Lecture Slides By Adil Aslam

My Email Address:

adilaslam5959@gmail.com



About Me

i am Student of BSCS
My email address:
adilaslam5959@gmail.com

Templates in C++

Templates in C++ Programming

- Templates in C++ programming allows function or class to work on **more** than one **data type** at once without writing **different codes** for **different data** types. Templates are often used in **larger programs** for the purpose of **code reusability** and flexibility of program. The concept of templates can be used in two different ways:

  - **Function Templates**
  - **Class Templates**
  - As of C++ 11, **Variable Template** has also been added.

Templates in C++ Programming

- Templates allow programmer to create a common class or function that can be used for a variety of data types. The parameters used during its definition is of generic type and can be replaced later by actual parameters. This is known as the concept of **generic programming**. The main advantage of using a template is the reuse of same algorithm for various data types, hence saving time from writing similar codes.
- **For example**, consider a situation where we have to sort a list of students according to their roll number and their percentage. Since, roll number is of integer type and percentage is of float type, we need to write separate sorting algorithm for this problem. But using template, we can define a generic data type for sorting which can be replaced later by integer and float data type.

Function Templates

- A **function templates** work in similar manner as function but with **one** key difference.
- A **single function** template can work on **different types** at once but, different functions are needed to perform **identical** task on **different data** types.
- If you need to perform **identical** operations on **two or more** types of data then, you can use function **overloading**. But better approach would be to use function **templates** because you can perform this task by writing **less code** and code is easier to maintain.

Function Templates

- A generic function that represents several functions performing same task but on different data types is called function template.

- **For example**, a function to add two integer and float numbers requires two functions. One function accept integer types and the other accept float types as parameters even though the functionality is the same. Using a function template, a single function can be used to perform both additions. It avoids unnecessary repetition of code for doing same task on various data types.

Why use Function Templates

- Templates are **instantiated** at compile-time with the source code.
- Templates are used **less code** than overloaded C++ functions.
- Templates are **type safe**.
- Templates allow **user-defined** specialization.
- Templates allow **non-type** parameters.

How to define function template?

- A function template starts with **keyword template** followed by template parameter/s inside **<>** which is followed by function declaration.

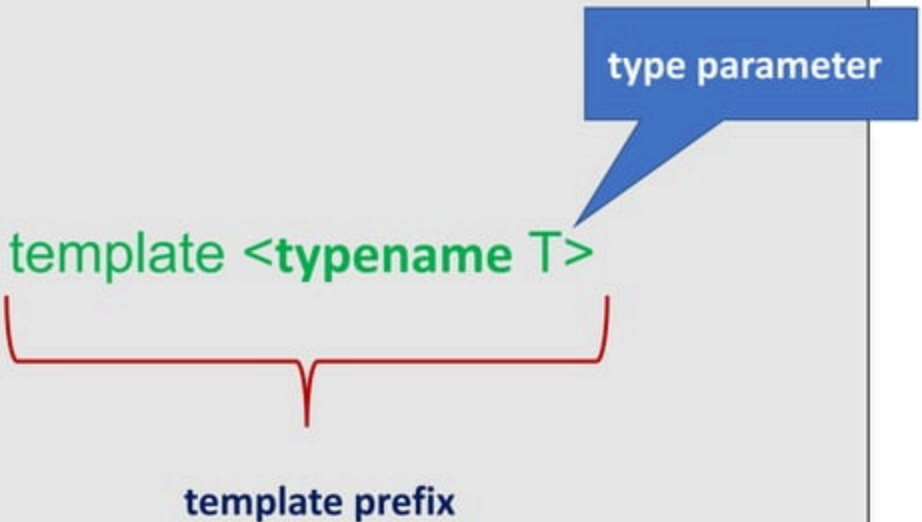
```
template <class T>
 T some_function(T argument)
{
    .... ... ....
 }
```

- **T** is a **template** argument and **class** is a keyword.
- We can also use keyword **typename** instead of class.
- When, an argument is passed to **some_function( )**, compiler generates new version of **some_function()** to work on argument of that type.

How to define function template?

- So ,
- The templated type keyword specified can be either "**class**" or " **typename**":
  - **template<class T>**
  - **template<typename T>**
- Both are valid and behave exactly the same. I prefer "**typename**".

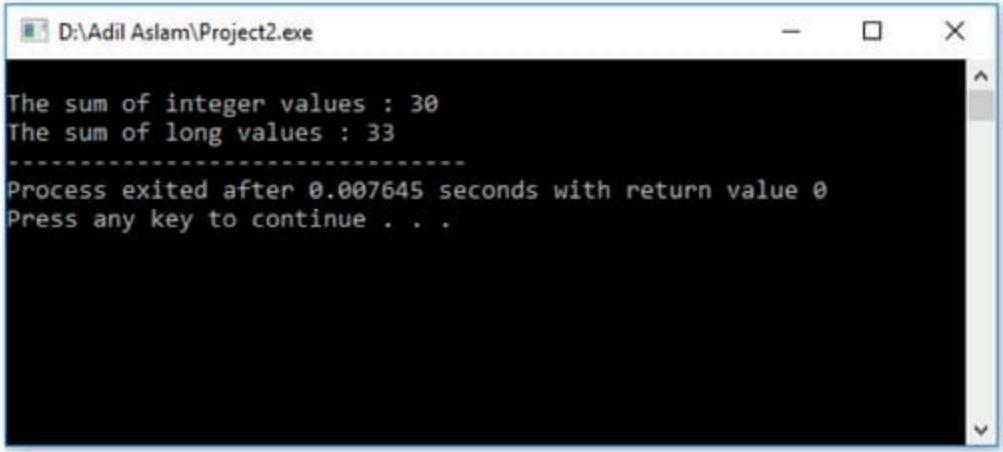
Function Templates



Example of Function Template

```
template <typename T>
T Sum(T n1, T n2) {  // Template function
   T rs;
         rs = n1 + n2;
         return rs;
}
int main() {
   int A=10,B=20,C;
         long I=11,J=22,K;
         C = Sum(A,B);   // Calling template function
         cout<<"\nThe sum of integer values : "<<C;
         K = Sum(I,J);   // Calling template function
         cout<<"\nThe sum of long values : "<<K;
}
```

Output of the Previous Program is :



Function Templates

Example to show you function template use less code than function overloading

Function Overloading Example

Overloaded functions specified for each data type

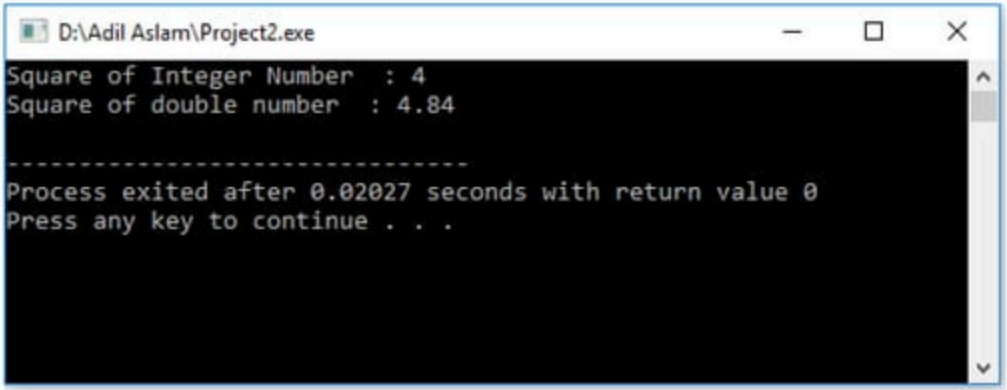
Function Overloading Example-1

```
#include <iostream>
using namespace std;
int square (int x)
{
  return x * x;
}
double square (double x)
{
  return x * x;
}
```

Function Overloading Example-2

```
int main()
{
  int   i, ii;
  double d, dd;
  i = 2;
  d = 2.2;
  ii = square(i);
  cout << "Square of Integer Number "<< " : " << ii << endl;
  dd = square(d);
  cout<< "Square of double number "<< " : " << dd << endl;
return 0;
}
```

Output of the Previous Program is :



Function Templates

Example to show you function template use less code than function overloading

Function Template Example

A single template to support all data types

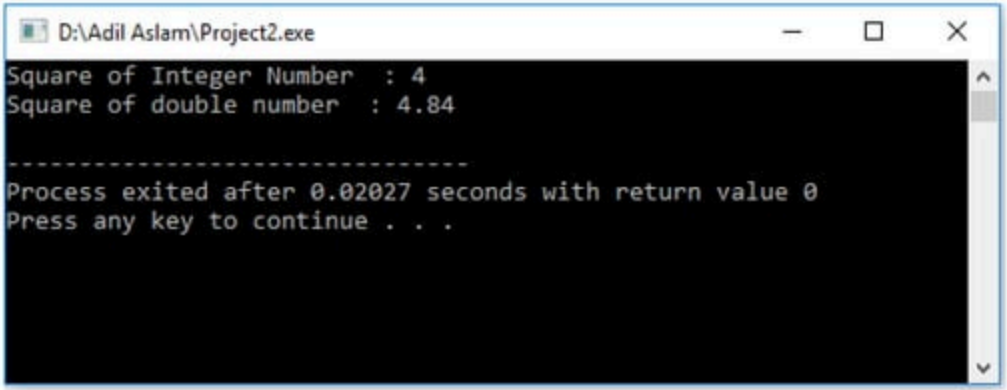
Function Template Example-1

```
#include <iostream>
using namespace std;
template <typename T>
T square(T x)
{
  T result;
  result = x * x;
  return result;
}
```

Function Template Example-2

```
int main()
{
  int   i, ii;
  double d, dd;
  i = 2;
  d = 2.2;
  ii = square(i);
  cout << "Square of Integer Number "<< ": " << ii << endl;
  dd = square(d);
  cout<< "Square of double number "<< " : " << dd << endl;
return 0;
}
```

Output of the Previous Program is :



Function Templates

Function Templates with one Argument

Square (Using Template)

```
template<class T>
void Square(T number)
{
    cout << number * number;
}
```

Single Argument

```
void Square(data_type number)
{
  cout << number * number;
}
```

Function Templates

Using Templates with Two or More than Arguments

Simple Multiplication

```
void multiply(int num1, double num2)
{
    cout << "Result:\t" << num1 * num2;
}
```

```
void multiply(double num1, int num2)
{
    cout << "Result:\t" << num1 * num2;
}
```

Simple Multiplication

```
void multiply(data_type1 num1, data_type2 num2)
{
    cout << "Result:\t" << num1 * num2;
}
```

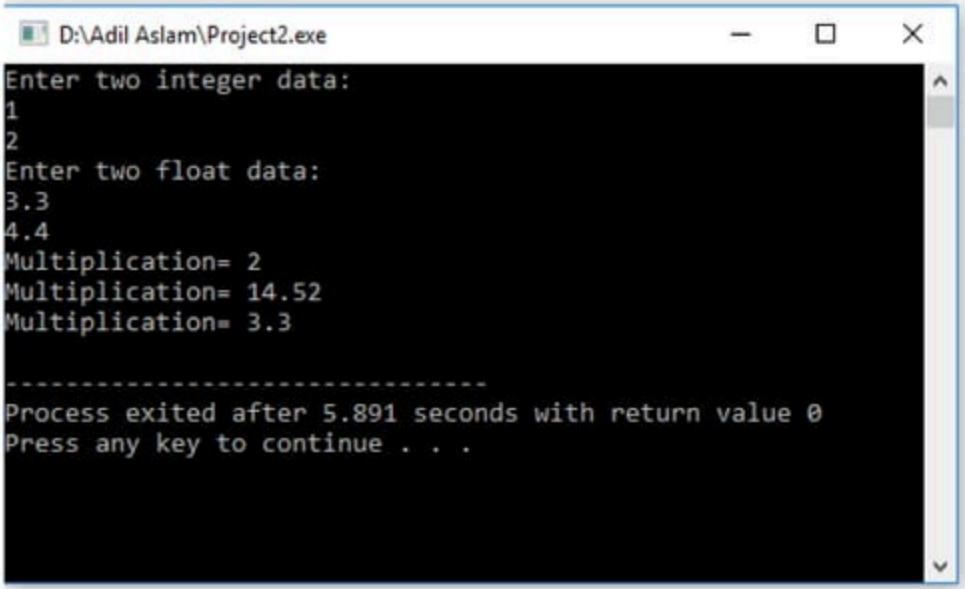
Multiplication (Using Template)

```
template<class T, class U>
void multiply(T num1, U num2)
{
    cout << "Result:\t" << num1 * num2;
}
```

Multiplication using Function Template

```
template<class T , class U>
void multiply(T a , U b) { // defining template function
   cout<<"Multiplication= "<<a*b<<endl;
}
int main() {
   int a,b;
   float x,y;
   cout<<"Enter two integer data: "<<endl;
   cin>>a>>b;
   cout<<"Enter two float data: "<<endl;
   cin>>x>>y;
   multiply(a,b); // Multiply two integer type data
   multiply(x,y); // Multiply two float type data
   multiply(a,x); // Multiply a float and integer type data
   return 0;
}
```

Output of the Previous Program is :



Explanation of the Previous

- This program illustrates the use of template function in C++. A template function multiply*()* is created which accepts two arguments and multiply them. The type of argument is not defined until the function is called. This single function is used to multiply two data of integer type, float type and, integer and float type. We don't need to write separate functions for different data types. In this way, a single function can be used to process data of various type using function template.
- Also we use template with two arguments

Your Task...☺

Write a **generic function** that **swaps** values in **two variables**. Your function should have two parameters of the same type. Test the function with int, double and string values.

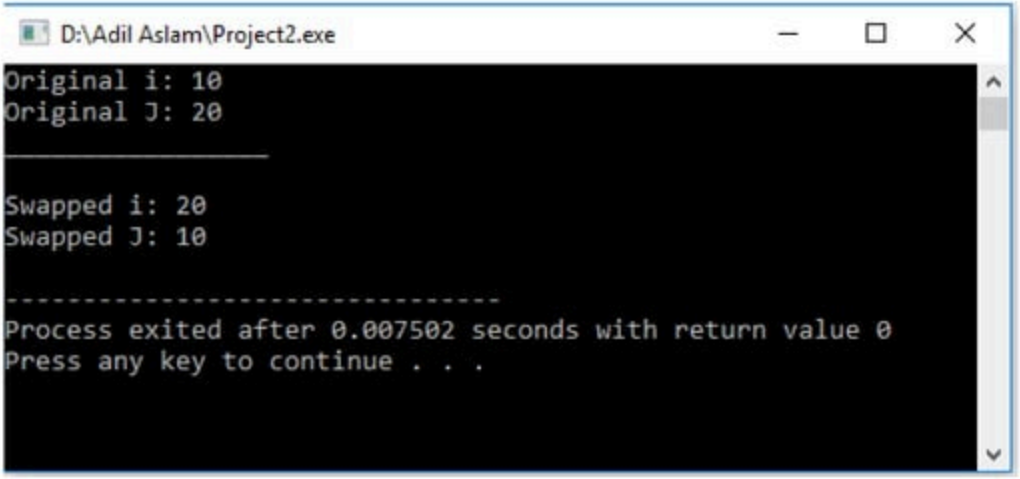
Solution of Previous Problem-1

```
#include <iostream>
using namespace std;
template <typename T>
void swap(T *a, T *b)
{
 T *temp;

 temp = a;
 a = b;
 b = temp;
}
```

Solution of Previous Problem-2

```
int main()
{
 int i=10, j=20;
 cout <<"Original i: "<<i<<endl;
 cout<<"Original J: "<< j << endl;
 cout<<"_____"<<endl<<endl;
 swap(i,j);  // swapping integers
 cout <<"Swapped i: "<<i<<endl;
 cout<<"Swapped J: "<< j << endl;

 return 0;
}
```

Output of the Previous Program is :



Your Home Work...☺

- Write a C++ program using function templates to add two numbers of int and float data types?

Class Template

- Like function template, a class template is a common class that can represent various similar classes operating on data of different types. Once a class template is defined, we can create an object of that class using a specific basic or user-defined data types to replace the generic data types used during class definition.
- **Syntax of Class Template**

```
template <class T1, class T2, ...>
class classname
{
   attributes;
   methods;
};
```

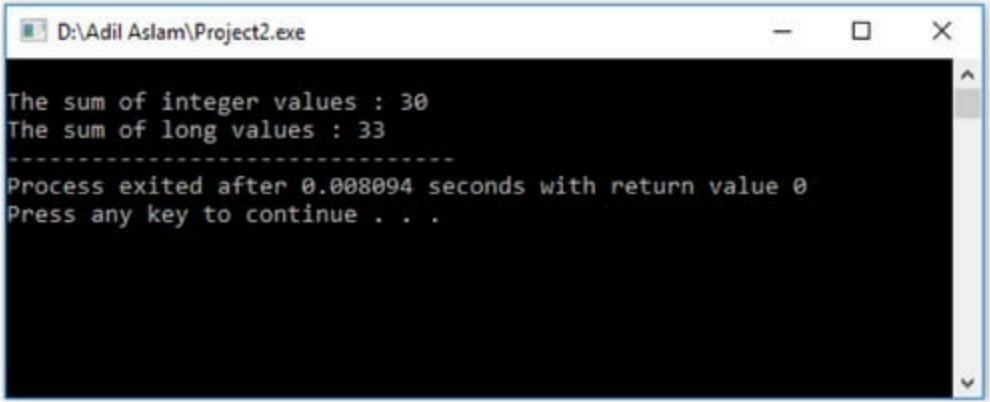
Example of Class Template-1

```
template <class T>
class Addition {      // Template class
        public:
        T Add(T, T);
};
template <class T>
T Addition<T>::Add(T n1, T n2) {
         T rs;
         rs = n1 + n2;
        return rs;
}
```

Example of Class Template-2

```
int main()
{
        Addition <int>obj1;
        Addition <long>obj2;
        int A=10,B=20,C;
        long I=11,J=22,K;
        C = obj1.Add(A,B);
        cout<<"\nThe sum of integer values : "<<C;
        K = obj2.Add(I,J);
        cout<<"\nThe sum of long values : "<<K;
}
```

Output of the Previous Program is :



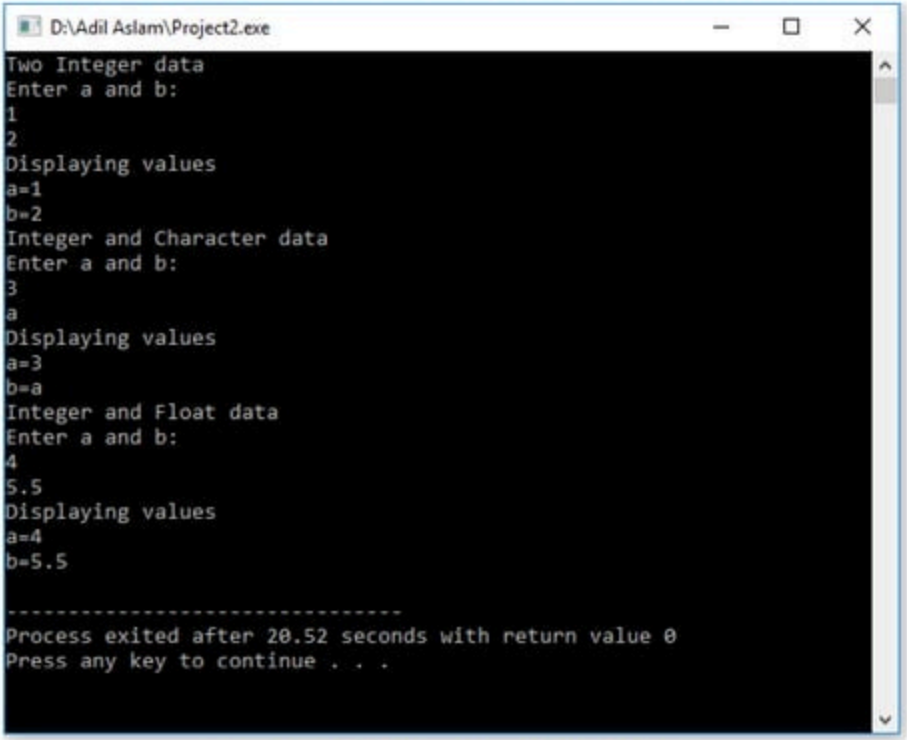
One More Example of Class Template-1

```
template<class T1,class T2>
class sample {
   T1 a;
   T2 b;
   public:
      void getdata() {
         cout<<"Enter a and b: "<<endl;
         cin>>a>>b;
      }
      void display() {
         cout<<"Displaying values"<<endl;
         cout<<"a="<<a<<endl;
         cout<<"b="<<b<<endl;
      }
};
```

One More Example of Class Template-2

```
int main() {
   sample<int,int> s1;
   sample<int,char> s2;
   sample<int,float> s3;
   cout <<"Two Integer data"<<endl;
   s1.getdata();
   s1.display();
   cout <<"Integer and Character data"<<endl;
   s2.getdata();
   s2.display();
   cout <<"Integer and Float data"<<endl;
   s3.getdata();
   s3.display();
   return 0;
}
```

Output of the Previous Program is :



Explanation of Previous Program

- In this program, a template class *sample* is created. It has two data a and b of generic types and two methods: *getdata()* to give input and *display()* to display data. Three object s1, s2 and s3 of this class is created. s1 operates on both integer data, s2 operates on one integer and another character data and s3 operates on one integer and another float data. Since, *sample* is a template class, it supports various data types.
- A class created form a class template is called a template class. The syntax foe defining an object of a template class is:

```
Classname <type> objectname(argument_list)
```

Overloading Template Function

- If there are more than one function of same name in a program which differ only by number and/or types of parameter, it is called function overloading. If at least one of these function is a template function, then it is called template function overloading. Template function can be overloaded either by using template functions or normal C++ functions of same name.

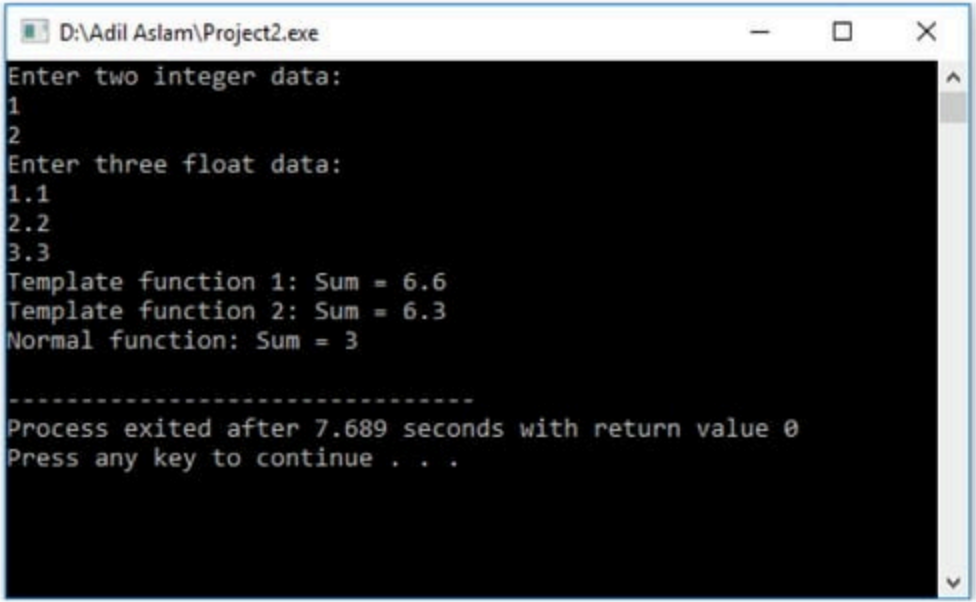
Example of Overloading Template Function-1

```
template<class T1>
void sum(T1 a,T1 b,T1 c) {
   cout<<"Template function 1: Sum = "<<a+b+c<<endl;
}

template <class T1,class T2>
void sum(T1 a,T1 b,T2 c) {
   cout<<"Template function 2: Sum = "<<a+b+c<<endl;
}

void sum(int a,int b) {
   cout<<"Normal function: Sum = "<<a+b<<endl;
}
```

Example of Overloading Template Function-2

```
int main()
{
   int a,b;
   float x,y,z;
   cout<<"Enter two integer data: "<<endl;
   cin>>a>>b;
   cout<<"Enter three float data: "<<endl;
   cin>>x>>y>>z;
   sum(x,y,z); // calls first template function
   sum(a,b,z); // calls first template function
   sum(a,b); // calls normal function
   return 0;
}
```

Output of the Previous Program is :



Explanation of Previous Program

- In this program, template function is overloaded by using normal function and template function. Three functions named *sum()* are created. The first function accepts three arguments of same type. The second function accepts three argument, two of same type and one of different and, the third function accepts two arguments of *int* type. First and second function are template functions while third is normal function. Function call is made from *main()* function and various arguments are sent. The compiler matches the argument in call statement with arguments in function definition and calls a function when match is found.

Templates and Static variables in C++

- **Function templates and static variables:**

- Each instantiation of function template has its own copy of local static variables. For example, in the following program there are two instances: *void fun(int)* and *void fun(double)*. So two copies of static variable *i* exist.

Templates and Static variables in C++

```
template <typename T>
void fun(const T& x) {
  static int i = 10;
  cout << ++i;
  return;
}
int main() {
  fun<int>(1);  // prints 11
  cout << endl;
  fun<int>(2);  // prints 12
  cout << endl;
  fun<double>(1.1); // prints 11
  return 0;
}
```

Templates and Static variables in C++

```
template <typename T>
void fun(const T& x) {
  static int i = 10;
  cout << ++i;
  return;
}
int main() {
  fun<int>(1);  // prints 11
  cout << endl;
  fun<int>(2);  // prints 12
  cout << endl;
  fun<double>(1.1); // prints 11
  return 0;
}
```

```
11
12
11
```

Templates and Static variables in C++

Class templates and static variables:

- The rule for class templates is same as function templates
- Each instantiation of class template has its own copy of member static variables. For example, in the following program there are two instances *Test* and *Test*. So two copies of static variable *count* exist.

Templates and Static variables-1

```
template <class T>
class Test {
private:
  T val;
public:
  static int count;
  Test()
  {
    count++;
  }
  // some other stuff in class
};
```

Templates and Static variables-1

```
template<class T>
int Test<T>::count = 0;
int main()
{
  Test<int> a;  // value of count for Test<int> is 1 now
  Test<int> b;  // value of count for Test<int> is 2 now
  Test<double> c;  // value of count for Test<double> is 1 now
  cout << Test<int>::count   << endl;  // prints 2
  cout << Test<double>::count << endl; //prints 1
  return 0;
}
```

```
2
1
```

Template Instantiation

- When the compiler generates a class, function or static data members from a template, it is referred to as template instantiation.
- A class generated from a class template is called a generated class.
- A function generated from a function template is called a generated function.
- A static data member generated from a static data member template is called a generated static data member.
- The compiler generates a class, function or static data members from a template when it sees an implicit instantiation or an explicit instantiation of the template.

Implicit Instantiation of a Class Template

- Consider the following sample. This is an example of implicit instantiation of a class template.

```
template <class T>
class Z {
 public:
    Z() {} ;
    ~Z() {} ;
    void f(){} ;
    void g(){} ;
} ;
int main() {
 Z<int> zi ;   //implicit instantiation generates class Z<int>
 Z<float> zf ; //implicit instantiation generates class Z<float>
return 0 ;
}
```

Implicit Instantiation of a Class Template

Consider the following sample. This sample uses the template class members Z<T>::f() and Z<T>::g().

```
template <class T>
class Z {
 public:
      Z() {} ;
      ~Z() {} ;
      void f(){} ;
      void g(){} ;
} ;
int main() {
      Z<int> zi ; //implicit instantiation generates class Z<int>
      zi.f() ;   //and generates function Z<int>::f()
      Z<float> zf ; //implicit instantiation generates class Z<float>
      zf.g() ;     //and generates function Z<float>::g()
return 0 ;
}
```

Implicit Instantiation of a Class Template

Consider the following sample. This sample uses the template class members Z<T>::f() and Z<T>::g().

```
template <class T>
class Z {
 public:
      Z() {} ;
      ~Z() {} ;
      void f(){} ;
      void g(){} ;
} ;
int main() {
      Z<int> zi ; //implicit instantiation generates class Z<int>
      zi.f() ;   //and generates function Z<int>::f()
      Z<float> zf ; //implicit instantiation generates class Z<float>
      zf.g() ;     //and generates function Z<float>::g()
return 0 ;
}
```

This time in addition to the generating classes Z<int> and Z<float>, with constructors and destructors, the compiler also generates definitions for Z<int>::f() and Z<float>::g(). The compiler does not generate definitions for functions, non-virtual member functions, class or member class that does not require instantiation. In this example, the compiler did not generate any definitions for Z<int>::g() and Z<float>::f(), since they were not required.

Implicit Instantiation of a Class Template-1

```
template <class any_data_type>
class Test {
public:
        // a constructor
        Test(){ };
        // a destructor
        ~Test(){ };
        // a member functions...
        any_data_type Funct1(any_data_type Var1)
        {return Var1;}
        any_data_type Funct2(any_data_type Var2)
        {return Var2;}
};
```

Implicit Instantiation of a Class Template-2

```
// explicit instantiation of class Test<int>
template class Test<int>;
// explicit instantiation of class Test<double>
template class Test<double>;
// do some testing
int main()
{
Test<int> Var1;
        Test<double> Var2;
        cout<<"Var1 = "<<Var1.Funct1(200)<<endl;
        cout<<"Var2 = "<<Var2.Funct2(3.123)<<endl;
return 0;
}
```

Implicit Instantiation of a Class Template-2

```
// explicit instantiation of class Test<int>
template class Test<int>;
// explicit instantiation of class Test<double>
template class Test<double>;
// do some testing
int main()
{
Test<int> Var1;
        Test<double> Var2;
        cout<<"Var1 = "<<Var1.Funct1(200)<<endl;
        cout<<"Var2 = "<<Var2.Funct2(3.123)<<endl;
return 0;
}
```

```
Var1 = 200
Var2 = 3.123
```

Explicit Instantiation of a Class Template

Consider the following sample. This is an example of explicit instantiation of a class template.

```
template <class T>
class Z
 {
  public:
      Z() {} ;
      ~Z() {} ;
      void f(){} ;
      void g(){} ;
} ;
int main() {
      template class Z<int> ; //explicit instantiation of class Z<int>
      template class Z<float> ; //explicit instantiation of  class Z<float>
return 0 ;
}
```

Explicit Instantiation of a Class Template

Consider the following sample. Will the compiler generate any classes in this case? The answer is NO.

```
template <class T>
class Z
{
  public:
      Z() {} ;
      ~Z() {} ;
      void f(){} ;
      void g(){} ;
} ;
int main()
{
      Z<int>* p_zi ;    //instantiation of class Z<int> not required
      Z<float>* p_zf ; //instantiation of class Z<float> not required
return 0 ;
}
```

Explicit Instantiation of a Class Template

Consider the following sample. Will the compiler generate any classes in this case? The answer is NO.

```
template <class T>
class Z
{
  public:
      Z() {} ;
      ~Z() {} ;
      void f(){} ;
      void g(){} ;
} ;
int main()
{
      Z<int>* p_zi ;    //instantiation of class Z<int> not required
      Z<float>* p_zf ; //instantiation of class Z<float> not required
return 0 ;
}
```

This time the compiler does not generate any definitions! There is no need for any definitions. It is similar to declaring a pointer to an undefined class or struct.

Implicit Instantiation of a Function Template

Consider the following sample. This is an example of implicit instantiation of a function template.

```
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
   return a > b ? a : b ;
}
int main() {
int I ;
I = max(10, 15) ; //implicit instantiation of max(int, int)
char c ;
c = max('k', 's') ; //implicit instantiation of max(char, char)
}
```

Implicit Instantiation of a Function Template

Consider the following sample. This is an example of implicit instantiation of a function template.

```
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
   return a > b ? a : b ;
}
int main() {
int I ;
I = max(10, 15) ; //implicit instantiation of max(int, int)
char c ;
c = max('k', 's') ; //implicit instantiation of max(char, char)
}
```

In this case the compiler generates functions max(int, int) and max(char, char). The compiler generates definitions using the template function max.

Template Specialization in C++

- Template is a great feature in C++. We write code once and use it for any data type including user defined data types.
- For example, sort() can be written and used to sort any data type items. A class stack can be created that can be used as a stack of any data type.
- What if we want a different code for a particular data type? Consider a big project that needs a function sort() for arrays of many different data types. Let Quick Sort be used for all datatypes except char. In case of char, total possible values are 256 and counting sort may be a better option. Is it possible to use different code only when sort() is called for char data type?
- It is possible in C++ to get a special behavior for a particular data type. This is called template specialization.

Template Specialization in C++

```
// A generic sort function
template <class T>
void sort(T arr[], int size)
{
   // code to implement Quick Sort
}
```

Another example could be a class *Set* that represents a set of elements and supports operations like union, intersection, etc. When the type of elements is char, we may want to use a simple boolean array of size 256 to make a set. For other data types, we have to use some other complex technique.

```
// Template Specialization: A function specialized for char data type
template <>
void sort<char>(char arr[], int size)
{
   // code to implement counting sort
}
```

Class Template Specialization

- In some cases it is possible to override the template-generated code by providing special definitions for specific types. This is called template specialization.
- ***An Example Program for class template specialization***
- In the following program, a specialized version of class Test is written for int data type.

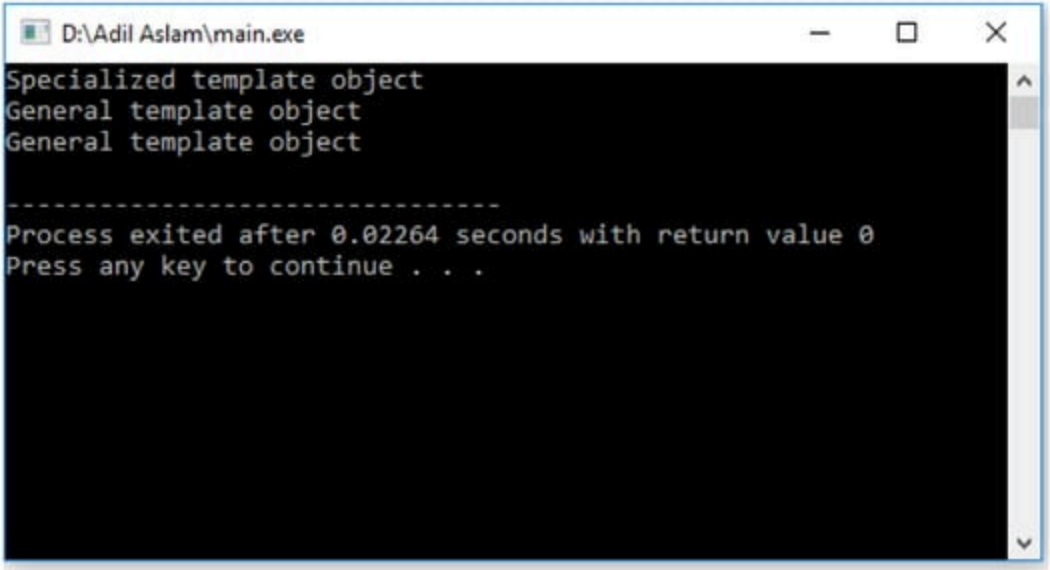
Class Template Specialization-1

```
#include <iostream>
using namespace std;
template <class T>
class Test
{
  // Data members of test
public:
   Test()
   {
      // Initialization of data members
      cout << "General template object \n";
   }
   // Other methods of Test
};
```

Class Template Specialization-2

```
template <>
class Test <int> {
public:
  Test() {
     // Initialization of data members
     cout << "Specialized template object\n";
  }
};
int main() {
   Test<int> a;
   Test<char> b;
   Test<float> c;
   return 0;
}
```

Output of the Previous Program is :



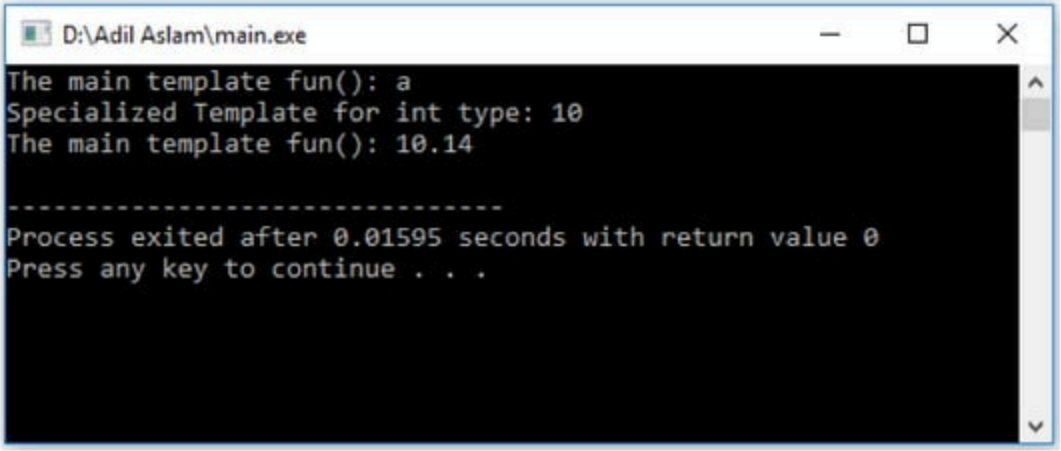
Template Function Specialization

- In some cases it is possible to override the template-generated code by providing special definitions for specific types. This is called template specialization.
- ***An Example Program for function template specialization***
- For example, consider the following simple code where we have general template fun() for all data types except int. For int, there is a specialized version of fun().

Template Function Specialization

```
#include <iostream>
using namespace std;
template <class T>
void fun(T a) {
  cout << "The main template fun(): " << a << endl;
}
template<>
void fun(int a) {
   cout << "Specialized Template for int type: " << a << endl;
}
int main() {
   fun<char>('a');
   fun<int>(10);
   fun<float>(10.14);
}
```

Output of the Previous Program is :



Your Task.. ☺

```
template <class T>
class stream {
        public:
                void f() { cout << "stream<T>::f()"<< endl ;}
} ;
template <>
class stream<char> {
        public:
                void f() { cout << "stream<char>::f()"<< endl ;}
} ;
int main() {
        stream<int> si ;
        stream<char> sc ;
        si.f() ;
        sc.f() ;
        return 0 ;
}
```

Your Task.. ☺

```
template <class T>
class stream {
        public:
                void f() { cout << "stream<T>::f()"<< endl ;}
} ;
template <>
class stream<char> {
        public:
                void f() { cout << "stream<char>::f()"<< endl ;}
} ;
int main() {
        stream<int> si ;
        stream<char> sc ;
        si.f() ;
        sc.f() ;
        return 0 ;
}
```

In this example, stream<char> is used as the definition of streams of chars; other streams will be handled by the template class generated from the class template.

```
stream<T>::f()
stream<char>::f()
```

Template Class Partial Specialization

- We may want to generate a specialization of the class for just one parameter, for example

```
//base template class
template<typename T1, typename T2>
class X {
};
template<typename T1>  //partial specialization
class X<T1, int> {
};
int main() {
      // generates an instantiation from the base template
      X<char, char> xcc ;

      //generates an instantiation from the partial specialization
       X<char, int> xii ;
      return 0 ;
}
```

Template Class Partial Specialization

- We may want to generate a specialization of the class for just one parameter, for example

```
//base template class
template<typename T1, typename T2>
class X {
};
template<typename T1>  //partial specialization
class X<T1, int> {
};
int main() {
      // generates an instantiation from the base template
      X<char, char> xcc ;

      //generates an instantiation from the partial specialization
       X<char, int> xii ;
      return 0 ;
}
```

A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list.

How does Template Specialization Work?

- When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being used for a new data type or new set of data types(in case of multiple template arguments).

- If a specialized version is present, compiler first checks with the specialized version and then the main template. Compiler first checks with the most specialized version by matching the passed parameter with the data type(s) specified in a specialized version.

Template Function Specialization

```
#include <iostream>
using namespace std ;
template <class T>
T max(T a, T b)
{
   return a > b ? a : b ;
}
int main()
{
   cout << "max(10, 15) = " << max(10, 15) << endl ;
   cout << "max('k', 's') = " << max('k', 's') << endl ;
   cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
   return 0 ;
}
```

Compiler Error

[Error] call of overloaded 'max(int, int)' is ambiguous

Template Friend Declarations

- Template friend declarations and definitions are permitted in class definitions and class template definitions. Below is an example of a class definition that declares a template class Peer<T> as a friend.

```
class Person
 {
   private:
      float my_money;
      template<class T> friend class Peer;
   };
```

Friends and Templates

- There are four kinds of relationships between classes and their friends when templates are involved:
- **One-to-many:** A non-template function may be a friend to all template class instantiations.
- ***Many-to-one*:** All instantiations of a template function may be friends to a regular non-template class.
- ***One-to-one*:** A template function instantiated with one set of template arguments may be a friend to one template class instantiated with the same set of template arguments. This is also the relationship between a regular non-template class and a regular non-template friend function.
- ***Many-to-many*:** All instantiations of a template function may be a friend to all instantiations of the template class.

A Non-Template Friend Class or Friend Function

```
/** - A nontemplate friend class or friend function.
* A nontemplate friend class or friend function. In the following example, the function foo
(), the member function bar()
* the member function bar() and the class foobar are friends to all instantiation of the clas
s tepmlate QueueItem .
*
*  this is a one-to-one relationship and the fiends are determined/fixed right in the declarati
on (there is only one declaration of the friends)
*/
class Foo {
    void bar();
};

class foobar {}

template <class T>
class QueueItem {
    friend class foobar;
    friend void foo();
    friend void Foo::bar();
};
```

A Non-Template Friend Class or Friend Function

```
/** - A nontemplate friend class or friend function.
* A nontemplate friend class or friend function. In the following example, the function foo
(), the member function bar()
* the member function bar() and the class foobar are friends to all instantiation of the clas
s tepmlate QueueItem .
*
*  this is a one-to-one relationship and the fiends are determined/fixed right in the declarati
on (there is only one declaration of the friends)
*/
class Foo {
    void bar();
};

class foobar {}

template <class T>
class QueueItem {
    friend class foobar;
    friend void foo();
    friend void Foo::bar();
};
```

It is the one-to-one relationship, and there is only one function . and no matter how many instance of the function template , there is only one declaration.

A Bound Friend Class Template or Function Template

```
/** - A bound friend class template or function template
*   this is a one-to-one relationship and the friends are not determined (the friends is create
d when the class is instantiated).-
*/
template <class Type>
class foobar {
//
};

template <class Type>
void foo(QueueItem<Type>);

template <class Type>
class Queue {
    void bar();
};

template <class Type>
class QueueItem {
    friend class foobar<Type>;
    friend class foo<Type>(QueueItem<Type>);
    friend class Queue<Type>::bar();
};
```

A Bound Friend Class Template or Function Template

```
/** - A bound friend class template or function template
*   this is a one-to-one relationship and the friends are not determined (the friends is create
d when the class is instantiated).-
*/
template <class Type>
class foobar {
//
};

template <class Type>
void foo(QueueItem<Type>);

template <class Type>
class Queue {
    void bar();
};

template <class Type>
class QueueItem {
    friend class foobar<Type>;
    friend class foo<Type>(QueueItem<Type>);
    friend class Queue<Type>::bar();
};
```

This is also a one-to-one relationship, but for a particular instantiation of the class template, there is one friend for that instantiation.

An Unbound Friend Class

```
/** - An unbound friend class template or function template
*  unbound friend classes are multiple-to-one relationship
*
*/
template <class Type>
template QueueItem
{
    friend class foobar;
    template <class T>
    friend void foo(QueueItem<T>);
    template <class T>
    friend void Queue<T>::bar();
};
```

There is a one-to-many relationship, where for a particular instantiation of the class template, there could be many function/classes that are the friends of the class template instantion.

Templates and Default Arguments

- **Default Parameters For Templates in C++:**
- Like function default arguments, templates can also have default arguments. For example, in the following program, the second parameter U has the default value as char.

```
template<class T, class U = char>
class A {
public:
   T x;
   U y;
};
int main() {
   A<char> a;
   A<int, int> b;
   cout<<sizeof(a)<<endl;
   cout<<sizeof(b)<<endl;
   return 0;
}
```

char takes 1 byte and int takes 4 bytes

```
2
8
```

Non-Type Template Arguments

- we have seen that a template can have multiples arguments. It is possible to use non-type arguments. That is in the addition to the type argument T, we can also use other augments such as strings, function names, constant expressions and built-in types.
- Consider the following example:

```
template<class T , int size>
class array
{
    T  a[size];
    //.....//
};
```

Non-Type Template Arguments

```
template<class T , int size>
class array
{
    T  a[size];
    //.....//
};
```

- This template supplies the size of the array as an argument. This implies that the size of the array is known to the compiler at the compile time itself. The arguments must be specified whenever a template class is created. Example

```
array<int, 10> a1;       //Array of 10 integers
array<float, 10> a1;     //Array of 5 floats
array<char, 20> a1;      //Char Array of size 20
//The size is given as an argument to the template class.
```

Non-Type Template Arguments

- We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of template.
- The important thing to note about non-type parameters is, they must be const. Compiler must know the value of non-type parameters at compile time.
- Because compiler needs to create functions/classes for a specified non-type value at compile time. In below program, if we replace 10000 or 25 with a variable, we get compiler error.

```
template <class T, int max>
int arrMin(T arr[], int n) {
  int m = max;
  for (int i = 0; i < n; i++)
    if (arr[i] < m)
      m = arr[i];
  return m;
}
int main() {
  int arr1[]  = {10, 20, 15, 12};
  int n1 = sizeof(arr1)/sizeof(arr1[0]);
  char arr2[] = {1, 2, 3};
  int n2 = sizeof(arr2)/sizeof(arr2[0]);
  // Second template parameter to arrMin must be a constant
  cout << arrMin<int, 10000>(arr1, n1) << endl;
  cout << arrMin<char, 256>(arr2, n2);
  return 0;
}
```

```
10
1
```

Non-Type Template Arguments-1

```
template <class T, int N>
class mysequence {
    T memblock [N];
    public:
    void setmember (int x, T value);
    T getmember (int x);
};
template <class T, int N>
void mysequence<T,N> :: setmember (int x, T value) {
    memblock[x] = value;
}
template <class T, int N>
T mysequence<T,N> :: getmember (int x) {
    return memblock[x];
}
```

Non-Type Template Arguments-2

```
int main ()
{
    mysequence <int, 5> myints;
    mysequence <double, 5> myfloats;
    myints.setmember (0, 100);
    myfloats.setmember (3, 3.1416);
    cout << myints.getmember(0) << '\n';
    cout << myfloats.getmember(3) << '\n';
    return 0;
}
```

```
100
3.1416
```

Thank You ☺