- In the previous example, *T* is the template parameter and *int* or *char* (used at the time of *stack* object creation) are the template arguments.
- To use the default value of a template parameter, the corresponding template argument must be omitted :

```
template <class T = int>
class stack {
   T * arr ; int  top ;
   public : stack(int size ) {
                    arr = new T[size] ;
                    top = -1 ;
                    cout<<sizeof(T)<<"\n" ;
            }
         void push(T t) ;
         T pop( ) ; } ;

template <class T> void stack <T> :: push(T t) { }

template <class T> T stack <T>::pop( ) { }

main( ) {
   stack <>      st_int(10) ;
   stack <char>      st_char(20) ;
   // this would generate an error -- stack    var(8) ;
}
```

Output :
4
1

- These are generally declared in a "header file" with a *.h* extension, and the implementation (the definition of these functions) is in an independent file with c++ code.
- **Because templates are compiled when required, this forces a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as its declaration. That means that we cannot separate the interface in a separate header file, and that we must include both interface and implementation in any file that uses the templates.**
- **Since no code is generated until a template is instantiated when required, compilers are prepared to allow the inclusion more than once of the same template file with both declarations and definitions in a project without generating linkage errors.**

Templates and multiple-filed projects

- From the point of view of the compiler, templates are not normal functions or classes.
- They are compiled on demand, meaning that the code of a template function is not compiled until an instantiation with specific template arguments is required.
- At that moment, when an instantiation is required, the compiler generates a function specifically for those arguments from the template.
- For large code, it is usual to split the code of a program in different source code files.
- In these cases, the interface and implementation are generally separated.
- Taking a library of functions as example, the interface generally consists of declarations of the prototypes of all the functions that can be called.

```
class A {int  a1, a2 ;
    public : A(int x) : a1(x) , a2(2*x) { }
          int f( ) {return a1 ;}} ;

template <class T1, class T2>
T2 anything(T1 a , T1 b, T2 c) {return T2(a+b+c.f( ) ) ;}

char anything(char b) { return b ; }
char anything(int a, int b, char c) { return 'a' ; }

main( ) {
    int          x = 9, y = 10 ;
    long  p = 10, q = 11 ;
    char  c = 'Z' , d = 'A' ;
    A            a1(60) ;
    A a_obj = anything (p, q, a1) ;
    cout<<anything('q') ;
    cout<<anything(1, 2, 'q') ;
}
```

```
template <class T>
T max (T a, T b) {
    cout<<"templ\n" ;
    return a>b ?a:b ;
}

void main() {
    int     a = 90, b = 80 ;
    char  c = 'a', d = 'n' ;

    cout<<max(a, b)<<endl ;
    cout<<max(c, d)<<endl ;
    cout<<max(a, c)<<endl ;     // error : cannot generate max(int, char)
}
```

But if the following is added :

```
int max (int a, int b) {
    cout<<“specl\n" ;
    return a>b ?a:b ;
}
```

the first and the last function calls are resolved to the special version of the code.

Function Templates

Syntax :

```
template <class T>
return_type function_name(args_list)
{   }
```

Example :

```
template <class T>
void Swap(T &x1, T &x2) {
    T   temp = x1 ;
    x1 = x2 ; x2 = temp ; }

main( ) {
    int i = 10 , j = 80 ;
    Swap(i, j) ;
    char c = 'A' , d = 'B' ;
    Swap(c, d) ;}
```

- Templates enable us to define *generic* classes or functions

- Templates make it possible to use a single function or a class definition for a variety of data types i.e., to define a *family of functions or classes*.

- Template is defined with a parameter that would be replaced by specific data type at the time of actual use of class or function.

- A template can be considered as a type of a macro.

- Templates can be :
  - function templates or
  - class templates.

The need for template functions :

- To find minimum of two numbers, depending on the data type of the numbers, the functions have to be declared.

- Thus if the numbers are of int type, a function as given below is required :
  int min(int a, int b) ;
  For two float type numbers the following function is required :
  float min(float a, float b) ;

- That is, though the functionality remains the same, separate functions are required for separate data types.

- This repetition of code can be avoided by using template functions.

Performance

- The compiler does not compile any code when it encounters the template function definition.

- When the template function is invoked then at that point the compiler generates a specific version of the template function corresponding to the argument types.

- Finally at this point compiler generates the function call code.

- This it does every time the compiler comes across a call corresponding to a template function.

- Use of templates does not help save memory because all the different versions of the function are ultimately generated.

- The advantage is that the code can be written with the generic version only.

- Thus templates provide **reuse of the source code** in contrast to inheritance and containership which provide reuse of the object code.

- Templates can significantly reduce source code size and increase code flexibility without reducing type safety.

- A template function may be overloaded either by template functions or ordinary functions of its name.

- The overloading resolution is accomplished as follows :
  - Call an ordinary function that has an exact match.
  - Call a template function that could be created with an exact match.
  - Try normal overloading resolution to ordinary functions and call the one that matches.
- An error is generated if no match is found.

- No automatic conversions are applied to arguments on template funcs.

Example :

```
template <class T>  T Max(T a, T b) {return a>b?a:b ;}
void f(int a, int b, char c, char d) {
  int m1=  Max(a, b) ;   //max(int a, int b)
  char m2 = Max(c, d)   //max(char c, char d)
  int m3 = Max(a, c) ; } //error – 'c'  is a char and can not be converted to int
```

- Thus a template function can be overridden to perform a specific processing for a particular data type.
  Example :

```
template <class T>
void f(T a) {
    cout<<"inside template  "<<a<<endl ; }

void f(char c) {
    cout<<"non template "<<c<<endl ; }

main()
{
    f(10) ;
    f('A') ;
    f(10.8) ;
}
```

Output :

```
inside template 10
non template A
inside template 10.8
```

- Each template argument of a function template must affect the type of the function by affecting at least one input argument type of functions generated from the template.
  This ensures that functions can be selected and generated based on their arguments.

  Example :

```
template<class T> void f1(T) ;          // fine
template<class T> void f2(T*) ;         // fine
template<class T>  T f3(int) ;          // error
template<class T, class C> void f4(T t) ;       // error
template<class T> void f5(const  T&, complex) ; // fine
template<class T> void f6(Vector<List<T>>) ;    // fine
```

```
class A {int  a1, a2 ;
    public : A(int x) : a1(x) , a2(2*x) { }
          int f( ) {return a1 ;}} ;

template <class T1, class T2>
T2 anything(T1 a , T1 b, T2 c) {return T2(a+b+c.f( ) ) ;}

char anything(char b) { return b ; }
char anything(int a, int b, char c) { return 'a' ; }

main( ) {
    int          x = 9, y = 10 ;
    long  p = 10, q = 11 ;
    char  c = 'Z' , d = 'A' ;
    A            a1(60) ;
    A a_obj = anything (p, q, a1) ;
    cout<<anything('q') ;
    cout<<anything(1, 2, 'q') ;
}
```

```
class A {int  a1, a2 ;
    public : A(int x) : a1(x) , a2(2*x) { }
          int f( ) {return a1 ;}} ;

template <class T1, class T2>
T2 anything(T1 a , T1 b, T2 c) {return T2(a+b+c.f( ) ) ;}

char anything(char b) { return b ; }
char anything(int a, int b, char c) { return 'a' ; }

main( ) {
    int          x = 9, y = 10 ;
    long  p = 10, q = 11 ;
    char  c = 'Z' , d = 'A' ;
    A            a1(60) ;
    A a_obj = anything (p, q, a1) ;
    cout<<anything('q') ;
    cout<<anything(1, 2, 'q') ;
}
```

Class Templates

Example :

```
template <class T>
arrclass stack {
        T     *  ; int    top ;
        public : stack(int size ) {arr = new T[size] ; top = -1 ;}
                 void push(T &t) ;
                 T pop( ) ; } ;

template <class T>
void stack <T> :: push(T &t) { }

template <class T>
T stack <T>::pop( ) { }

main( ) {
        stack <int>   st_int(10) ;
        stack <char>      st_char(20) ; }
```

Syntax :

```
template <class T>
class class_name{ } ;
```

T may be any data type:

- built – in
- user defined
  - -structures
  - -classes

the keywords *class* and *typename* in a template parameter declaration