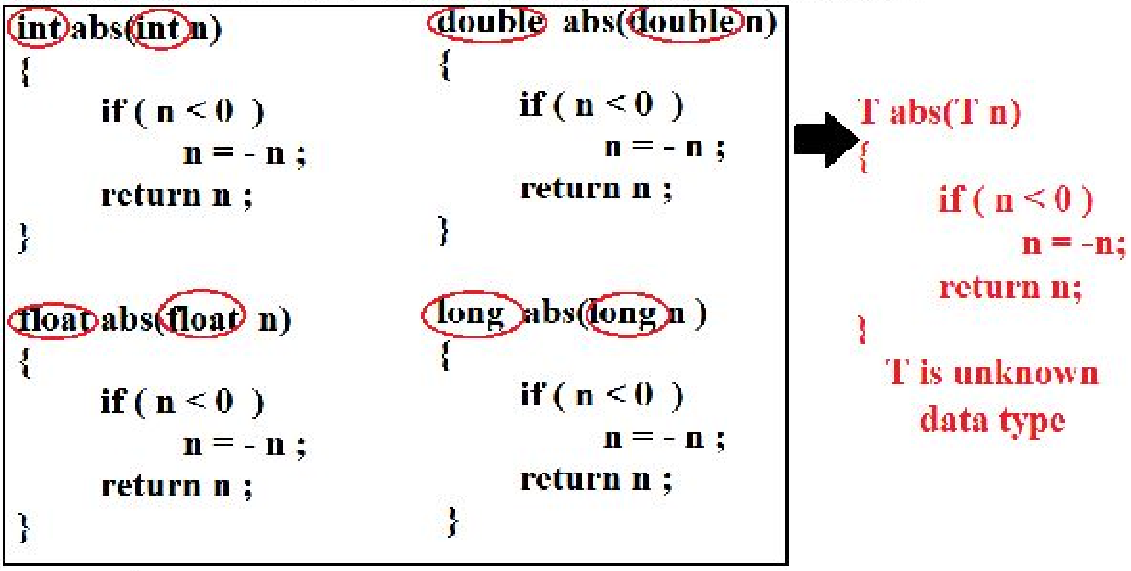
Template

- Templates are the ***foundation of generic programming***, which involves writing code in a way that is independent of any particular type.
- We can create ***function template and class template in C++.***
- A template is a blueprint or formula for creating a generic class or a function.



```
int abs(int n)
{
        if ( n < 0 )
                n = - n ;
        return n ;
}

double  abs(double n)
{
        if ( n < 0 )
                n = - n ;
        return n ;
}

float abs(float  n)
{
        if ( n < 0 )
                n = - n ;
        return n ;
}

long  abs(long n )
{
        if ( n < 0 )
                n = - n ;
        return n ;
}

T abs(T n)
{
        if ( n < 0 )
                n = -n;
        return n;
}
```

T is unknown data type

- For example :
  - To find out the absolute value of the number

```
#include<iostream>
int abs(int n)
{
        if ( n < 0 )
                n = - n ;
        return n ;
}

float abs(float  n)
{
        if ( n < 0 )
                n = - n ;
        return n ;
}

double  abs(double n)
{
        if ( n < 0 )
                n = - n ;
        return n ;
}

long  abs(long n )
{
        if ( n < 0 )
```

```
                        n = - n ;
                return n ;
        }

        main()
        {
                int a = - 5 , b ;
                float c = -1.4F, d ;
                long e = -2883L,f;
                double g= -5.3 , h ;
                b = abs (a) ;
                cout <<endl<<  b ;

                d = abs ( c ) ;
                cout << endl<< d ;

                f = abs ( e ) ;
                cout <<endl<<  f ;

                h = abs ( g ) ;
                cout <<endl<< h ;
}
```

- ***Some problem faced with this code***
  - Rewriting the same function body over and over for different Data types that is ***time consuming.***
  - The program ***consumes more disk space.***
  - If we locate any error in one such function, we need to remember to correct it in each function body

- ***Solution Using Template Function :***

```
#include<iostream>
Using namespace std;
template<class T>                    // This Is The Template Function
T abs(T n)                           // Which is Data Type Free (Means
{                                    // not bounded with the specific
                                     //data type
        cout << sizeof (T);
        if ( n < 0  )
                n = - n ;
        return n ;

}
 main()
 {
        int a = - 5 , b ;
        float c = -1.4F, d ;
        long e = -2883L,f;
        double g= -5.3 , h ;
        clrscr();
        b = abs (a) ;
```

```
            cout <<endl<<  b ;

            d = abs ( c ) ;
            cout << endl<< d ;

            f = abs ( e ) ;
            cout <<endl<<  f ;

            h = abs ( g ) ;
            cout <<endl<< h ;
}
```

- Note
  - When template function call, that Argument of template (T) ***is replaced with the specific type which send as an actual argument by Technology.***

```
template <class T>
T abs(T n)
{
        if ( n < 0  )
                n = - n ;
        return n ;
 }
```

- In template a data type can be represented by an argument (T in this case). It is the argument of template that can represent to ***any data type.***

- ***Function template override :***

  e.g.

```
#include<iostream>
using namespace std;
template <class T>
T abs(T n)
{
        if ( n < 0  )
                n = - n ;
        return n ;
}
int abs(int n)
{
        if ( n < 0 )
                n = - n ;
        /*some extra actiity*/
        cout << n ;
                return n ;
}
main()
{

        abs(1.2);
        abs(-5) ;
}
```

- Note
    - If we pass int then override function of int version get called otherwise version accordingly to template get called.

- **Multiple argument template**

```
#include<iostream>
using namespace std;
template<class Q, class R>
double divide(Q a , R b )
{
        return  a/b ;
}
main ()
{
        cout << endl << divide (1.2F,1.3) ;
        cout << endl << divide (1,1.3) ;
}
```

- **class template**

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

#define size 5
template <class T>
class stack
{
        private :
                T stk[size] ;
                int top ;
                public:
                stack()
                {
                        top = -1 ;
                }
                void push(T item)
                {
                        top++;
                        stk[top]= item;
                }
                void display()
                {
                        int i ;
                        cout << endl ;
                        for ( i = 0; i <= top; i++)
                        {
                                cout << setw(10) << stk[i];
```

```
                         }
                    }
};

main()
{
          stack <int> s1;
          stack <char> s2;
          stack <float> s3 ;

          s1.push(10);
          s1.push(20);
          s1.display();

          s2.push('A');
          s2.push('B');
          s2.display();

          s3.push(1.2F);
          s3.push(1.4F);
          s3.display();
}
}
```

- Note
  - The Actual CLASS name is ***stack<T>***

- ***Tips***
  - If we define member function outside the class

    ```
    void stack<T>::push ( T item )
    {
    }
    ```

  - When the member function return same type value and function define outside the class

    ```
    stack<T>  stack<T>::push ( T item )
    {
    }
    ```

  - Template argument can take default value

    ```
    template <class T ,  int max = 50 >
    class sample
    {
              private :
              T arr[max] ;
              ...
    };
    main ()
    {
              sample <float,40> s ;
              sample <float> s1 ;
    }
    ```

  - We can inherit the template

```
template <class T >
class deriveclass : public stack<T>
{
}
```

- Template should be used while creating a type safe collection class that can operate on data of any type.