Templates

C++ supports code reuse in different ways. Inheritance (is-a) and composition (has-a, nested objects) provide a way to reuse object code.

The template feature in C++ provides way to **reuse *source code***.

Function Templates

Suppose you want to write a function that returns the absolute value of a number. Ordinarily, this function would be written for a particular data type (for example **int**):

```
int abs(int n)                  // absolute value of ints
{
   return (n<0) ? -n : n;   // if n is negative, return -n
}
```

Here the function is defined to take an argument of type int and to return a value of this same type.

But now suppose you want to find the absolute value of a type **long**. You need to write a completely new function:

```
long abs(long n) {              // absolute value of longs
    return (n<0) ? -n : n;
}
```

And again, for type **float**:

```
float abs(float n){             // absolute value of floats
    return (n<0) ? -n : n;
}
```


The body of the function is the same in each case, but they must be separate functions because they handle variables of different types.

It's true that in C++ these functions can all be overloaded to have the same name, but you must nevertheless write a separate definition for each one.

In the C language, which does not support overloading, functions for different types can't even have the same name. In the C function library, this leads to families of similarly named functions, such as abs(), fabs(), fabsl(), labs(), cabs(), and so on.

Rewriting the same function body over and over for different types wastes time as well as space in the listing.

Also, if you find you've made an error in one such function, you'll need to remember to correct it in each function body.

Failing to do this correctly is a good way to introduce inconsistencies into your program.

It would be nice if there were a way to write such a function just once and have it work for many different data types.

This is exactly what function templates do for you.

Example:

```
// template used for absolute value function

template <class T>      // function template
T abs(T n)
{
    return (n < 0) ? -n : n;
}

int main()
{
  int int1 = 5;
  int int2 = -6;
  long lon1 = 70000L;
  long lon2 = -80000L;
  double doub1 = 9.95;
  double doub2 = -10.15;
  // calls instantiate functions
  cout << abs(int1) << endl;        // abs(int)
  cout << abs(int2) << endl;        // abs(int)
  cout << abs(lon1) << endl;        // abs(long)
  cout << abs(lon2) << endl;        // abs(long)
  cout << abs(doub1) << endl;       // abs(double)
  cout << abs(doub2) << endl;       // abs(double)
 ....
```


The key innovation in function templates is to represent the data type used by the function not as a specific type such as int, but by a *generic name* that can stand for *any* type. In the function template above, this name is **T**.

The template keyword signals the compiler that I'm about to define a function template.

The keyword class, within the angle brackets, might just as well be called type. As you've seen, you can define your own data types using classes, so there's really no distinction between types and classes.

The variable following the keyword class (T in this example) is called the *template argument*.

What does the compiler do when it sees the template keyword and the function definition that follows it?

The function template itself doesn't cause the compiler to generate any code. It can't generate code because it doesn't know yet what data type the function will be working with. It simply remembers the template for possible future use.

Code is generated (compiled) according to the function call statement. This happens in expressions such as abs(int1) in the statement:

```
cout << abs(int1);
```

When the compiler sees a function call, it knows that the type to use is int.
So it generates a specific version of the abs(**T** n) function for type int, substituting **int** wherever it sees the name **T** in the function template.

int → **T**

This is called *instantiating* the function template, and each instantiated version of the function is called a *template function*.

See Example: e11_1.cpp

With which data types can a template function work?

Data type must support operations performed in the function. For example in the abs function two operators are used: (n < 0) and -n .

Each data type, which supports these operators can be used with the abs function.

Notice that the amount of RAM used by the program is the same whether I use the template approach or write three separate functions.

What I've saved is having to type three separate functions into the source file. This makes the listing shorter and easier to understand.

Also, if I want to change the way the function works, I need make the change in only one place in the listing instead of three.

Template Arguments may be Objects

For example a template function MAX can find maximum of two integers, floating point numbers or complex numbers. Integers and floats are built-in types, complex is a user defined type (class).

```
class ComplexT{                    // A class to define complex numbers
    float re, im;
  public:
          :   // other member functions
    bool operator>(const ComplexT&) const ;   // header of operator> function
};

/* The Body of the function for operator > */
bool ComplexT::operator>(const ComplexT& z) const
{
   float f1 = re * re + im * im;
   float f2 = z.re * z.re + z.im * z.im;
   return f1 > f2;
}
```

```
// template function
template <class type>
const type & MAX(const type &v1, const type & v2)
{
   if (v1 > v2) return v1;
      else       return v2;
}

int main()
{
  int i1=5,  i2= -3;
  char c1='D',  c2='N';
  float f1=3.05,  f2=12.47;
  ComplexT z1(1.4, 0.6),  z2(4.6, -3.8);
  cout << MAX(i1,i2) << endl;
  cout << MAX(c1,c2) << endl;
  cout << MAX(f1,f2) << endl;
  cout << MAX(z1,z2) << endl;     // operator << must be overloaded to print ComplexT
  return 0;
}
```

See Example: e11_2.cpp

Function Templates with Multiple Arguments

Let's look at another example of a function template. This one takes three arguments: two template arguments and one basic type.

The purpose of this function is to search an array for a specific value. The function returns the array index for that value if it finds it, or -1 if it can't find it.

The arguments are a pointer to the array, the value to search for, and the size of the array.

```
// function returns index number of item, or -1 if not found template
template <class atype>
int find(const atype* array, atype value, int size)
{
   for(int j = 0;  j < size; j++)
      if( array[ j ] == value ) return j;
   return -1;
}

char chrArr[ ] = {'a', 'c', 'f', 's', 'u', 'z'};          // array
char ch = 'f';                                            // value to find
int intArr[ ] = {1, 3, 5, 9, 11, 13};
int in = 6;
double dubArr[ ] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;
```

```
int main()
{
   cout << "\n 'f' in chrArray: index=" << find(chrArr, ch, 6);
   cout << "\n 6 in intArray: index=" << find(intArr, in, 6);
   cout << "\n 4 in dubArray: index=" << find(dubArr, db, 6);
  return 0;
}
```

Here, the name of the template argument is atype.
The compiler generates three versions of the function, one for each type used to call it.

Template Arguments Must Match:

When a template function is invoked, all instances of the same template argument must be of the same type.
For example, in find(), if the array name is of type int, the value to search for must also be of type int. You can't say

```
int intarray[ ] = {1, 3, 5, 7};        // int array
float f1 = 5.0;                        // float value
int value = find(intarray, f1, 4);     // ERROR!
```

because the compiler expects all instances of atype to be the same type. It can generate a function find(int*, int, int);

but it can't generate find(int*, float, int);

More Than One Template Argument

You can use more than one template argument in a function template.
For example, suppose you like the idea of the find() function template, but you aren't sure how large an array it might be applied to. If the array is too large, then type long would be necessary for the array size instead of type int.
On the other hand, you don't want to use type long if you don't need to. You want to select the type of the array size, as well as the type of data stored, when you call the function.

```
template <class atype, class btype>
btype find(const atype* array, atype value, btype size)
{
   for( btype j = 0;  j < size;  j++)                    // note use of btype
      if( array[j] == value ) return j;
   return static_cast<btype>(-1);
}
```

Now you can use type int or type long (or even a user-defined type) for the size, whichever is appropriate.
The compiler will generate different functions based not only on the type of the array and the value to be searched for, but also on the type of the array size.

```
short int result , si = 100;
int invalue = 5;
result = find(intArr, invalue, si)
```

```
long lonresult, li = 100000;
float fvalue = 5.2;
lonresult = find(floatArr, fvalue, li)
```

Note that multiple template arguments can lead to many functions being instantiated from a single template.
Two such arguments, if there were six basic types that could reasonably be used for each one, would allow the creation of up to 36 functions.
This can take up a lot of memory if the functions are large. On the other hand, you don't instantiate a version of the function unless you actually call it.

Why Not Macros?
Old-time C programmers may wonder why we don't use macros to create different versions of a function for different data types. For example, the abs() function could be defined as
#define abs(n) ((n<0) ? (-n) : (n))
This has a similar effect to the function template, because it performs a simple text substitution and can thus work with any type. However, macros aren't used much in C++. There are several problems with them.
They don't perform any type checking. There may be several arguments to the macro that should be of the same type, but the compiler won't check whether or not they are.
Also, the type of the value returned isn't specified, so the compiler can't tell if you're assigning it to an incompatible variable.
In any case, macros are confined to functions that can be expressed in a single statement. There are also other, more subtle, problems with macros.
On the whole, it's best to avoid them.

Class Templates

The template concept can be applied to classes as well as to functions.
Class templates are generally used for data storage (container) classes. Stacks and linked lists, are examples of data storage classes.
The previous examples of these classes could store data of only a single basic type. The Stack class in the program that is presented below, could store data only of type int:

```
class Stack {
      int st[MAX];          // array of ints
      int top;              // index number of top of stack
    public:
      Stack();              // constructor
      void push(int var);   // takes int as argument
      int pop();            // returns int value
  };
```

If I wanted to store data of type long in a stack, I would need to define a completely new class:

```
  class LongStack {
      long st[MAX];         // array of longs
      int top;              // index number of top of stack
    public:
      LongStack();          // constructor
      void push(long var);  // takes long as argument
      long pop();           // returns long value
   };
```

Solution with a class template:

```
template <class Type>
class Stack{
      enum {MAX=100};
      Type st[MAX];                  // stack: array of any type
      int top;                       // number of top of stack
   public:
      Stack(){top = 0;}              // constructor
      void push(Type );              // put number on stack
      Type pop();                    // take number off stack
};

template<class Type>
void Stack<Type>::push(Type var) // put number on stack
{
   if(top > MAX-1)                   // if stack full,
     throw "Stack is full!";         // throw exception
   st[top++] = var;
}
```

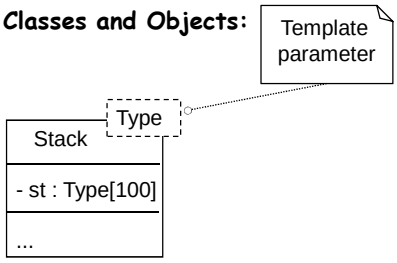


```
template<class Type>
Type Stack<Type>::pop()           // take number off stack
{
   if(top <= 0)                   // if stack empty,
        throw "Stack is empty!";  // throw exception
   return st[--top];
}

int main()
{
  // s1 is object of class Stack<float>
  Stack<float> s1;
  // push 2 floats, pop 2 floats
  try{
    s1.push(1111.1);
    s1.push(2222.2);
    cout << "1: " << s1.pop() << endl;
    cout << "2: " << s1.pop() << endl;
  }
  // exception handler
  catch(const char * msg) {
     cout << msg << endl;
  }

  // s2 is object of class Stack<long>
  Stack<long> s2;
  // push 2 longs, pop 2 longs
  try{
    s2.push(123123123L);
    s2.push(234234234L);
    cout << "1: " << s2.pop() << endl;
    cout << "2: " << s2.pop() << endl;
  }
  // exception handler
  catch(const char * msg) {
     cout << msg << endl;
  }
  return 0;
}    // End of program
```

See Example:e11_3.cpp

UML Notation for Template Classes and Objects:

A template class:



An object of template Stack.
In this example intStack object is an integer Stack .