Illustrations List *(Main Page)*

```
template< class T >
void printArray( const T *array, const int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
}
```

Fig. 12.1 A function template.

```
// Fig 12.2: fig12_02.cpp
// Using template functions
#include <iostream.h>

template< class T >
void printArray( const T *array, const int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
}

int main()
{
   const int aCount = 5, bCount = 7, cCount = 6;
   int a[ aCount ] = { 1, 2, 3, 4, 5 };
   double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ cCount ] = "HELLO";  // 6th position for null

   cout << "Array a contains:" << endl;
   printArray( a, aCount );  // integer template function

   cout << "Array b contains:" << endl;
   printArray( b, bCount );  // double template function

   cout << "Array c contains:" << endl;
   printArray( c, cCount );  // character template function

   return 0;
}
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Fig. 12.2 Using template functions.

```
// Fig. 12.3: tstack1.h
// Class template Stack
#ifndef TSTACK1_H
#define TSTACK1_H

#include <iostream.h>

template< class T >
class Stack {
public:
   Stack( int = 10 );    // default constructor (stack size 10)
   ~Stack() { delete [] stackPtr; } // destructor
   bool push( const T& ); // push an element onto the stack
   bool pop( T& );        // pop an element off the stack
```

Fig. 12.3 Demonstrating class template `Stack` (part 1 of 4).

```
private:
   int size;             // # of elements in the stack
   int top;              // location of the top element
   T *stackPtr;          // pointer to the stack

   bool isEmpty() const { return top == -1; }      // utility
   bool isFull() const { return top == size - 1; } // functions
};

// Constructor with default size 10
template< class T >
Stack< T >::Stack( int s )
{
   size = s > 0 ? s : 10;
   top = -1;               // Stack is initially empty
   stackPtr = new T[ size ]; // allocate space for elements
}

// Push an element onto the stack
// return true if successful, false otherwise
template< class T >
bool Stack< T >::push( const T &pushValue )
{
   if ( !isFull() ) {
      stackPtr[ ++top ] = pushValue; // place item in Stack
      return true;  // push successful
   }
   return false;     // push unsuccessful
}

// Pop an element off the stack
template< class T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() ) {
      popValue = stackPtr[ top-- ];  // remove item from Stack
      return true;  // pop successful
   }
   return false;     // pop unsuccessful
}

#endif
```

Fig. 12.3 Demonstrating class template `Stack` (part 2 of 4).

```
// Fig. 12.3: fig12_03.cpp
// Test driver for Stack template
#include <iostream.h>
#include "tstack1.h"

int main()
{
   Stack< double > doubleStack( 5 );
   double f = 1.1;
   cout << "Pushing elements onto doubleStack\n";

   while ( doubleStack.push( f ) ) { // success true returned
      cout << f << ' ';
      f += 1.1;
   }

   cout << "\nStack is full. Cannot push " << f
        << "\n\nPopping elements from doubleStack\n";

   while ( doubleStack.pop( f ) )  // success true returned
      cout << f << ' ';

   cout << "\nStack is empty. Cannot pop\n";

   Stack< int > intStack;
   int i = 1;
   cout << "\nPushing elements onto intStack\n";

   while ( intStack.push( i ) ) { // success true returned
      cout << i << ' ';
      ++i;
   }

   cout << "\nStack is full. Cannot push " << i
        << "\n\nPopping elements from intStack\n";

   while ( intStack.pop( i ) )  // success true returned
      cout << i << ' ';

   cout << "\nStack is empty. Cannot pop\n";
   return 0;
}
```

Fig. 12.3 Demonstrating class template `Stack` (part 3 of 4).

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Fig. 12.3 Driver for class template `Stack` (part 4 of 4).

```
// Fig. 12.4: fig12_04.cpp
// Test driver for Stack template.
// Function main uses a function template to manipulate
// objects of type Stack< T >.
#include <iostream.h>
#include "tstack1.h"

// Function template to manipulate Stack< T >
template< class T >
void testStack(
   Stack< T > &theStack,    // reference to the Stack< T >
   T value,                 // initial value to be pushed
   T increment,             // increment for subsequent values
   const char *stackName ) // name of the Stack< T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';

   while ( theStack.push( value ) ) { // success true returned
      cout << value << ' ';
      value += increment;
   }

   cout << "\nStack is full. Cannot push " << value
        << "\n\nPopping elements from " << stackName << '\n';

   while ( theStack.pop( value ) )  // success true returned
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop\n";
}
```

Fig. 12.4 Passing a `Stack` template object to a function template (part 1 of 2).

```

int main()
{
   Stack< double > doubleStack( 5 );
   Stack< int > intStack;

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );

   return 0;
}
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Fig. 12.4 Passing a `Stack` template object to a function template (part 2 of 2).