CHAPTER
15
Templates
Objectives

• To know the motivation and benefits of templates (§15.2).
• To declare template function with type parameters (§15.2).
• To develop a generic sort function using templates (§15.3).
• To develop generic classes using class templates (§§15.4-15.5).
15.1 Introduction
<side remark: templates>
C++ provides functions and classes for developing reusable software. Templates provide the capability to parameterize types in functions and classes. With this capability, you can define one function or one class with a generic type that can be substituted for a concrete type by the compiler. For example, you may define one function for finding the maximum number between two numbers of a generic type. If you invoke this function with two int arguments, the generic type is substituted by the int type.

This chapter introduced concept of templates and you will learn how to define function templates and class templates and use the templates with concrete types.

15.2 Templates Basics
Let us begin with a simple example to demonstrate the need for templates. Suppose you want to find the maximum of two integers, two doubles, and two characters, you might write three overloaded functions as follows:

```cpp
int maxValue(const int &value1, const int &value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}

double maxValue(const double &value1, const double &value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}

char maxValue(const char &value1, const char &value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}
```

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

<Side Remark line 1: int type>
<Side Remark line 9: double type>
<Side Remark line 17: char type>
These three functions are almost identical except that each uses a different type. The first function uses the int type in three places, the second uses the double type in three places, and the third uses the char type in three places. It would save typing, save space, make the program easy to maintain if you could simply define one function with a generic type as follows:

```cpp
<Side Remark line 1: generic type>

```GenericType``` maxValue(
    const ```GenericType``` &value1, const ```GenericType``` &value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}
```

This `GenericType` applies to all types such as `int`, `double`, and `char`.

C++ enables you to define a function template with generic types. Listing 15.1 defines a template function for finding a maximum value between two values of a generic type.

```cpp
#include <iostream>
#include <string>
using namespace std;

<Side Remark line 5: template prefix>
<Side Remark line 6: type parameter>
<Side Remark line 16: invoke maxValue>
<Side Remark line 18: invoke maxValue>
<Side Remark line 20: invoke maxValue>
<Side Remark line 22: invoke maxValue>

```template<typename T>
T``` maxValue(const T &value1, const T &value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}
```

Listing 15.1 GenericMaxValue.cpp (Finding Maximum Value)
The definition for the function template begins with the keyword `template` followed by a list of parameters. Each parameter must be preceded by the interchangeable keywords `typename` or `class` in the form `<typename typeParameter>` or `<class typeParameter>`. For example, line 5

```
    template<typename T>
```

begins the definition of the function template for `maxValue`. This line is also known as the `template prefix`. Here `T` is a type parameter. By convention, a single capital letter such as `T` is used to denote a type parameter.

The `maxValue` function is defined in lines 6-12. A type parameter can be used in the function just like a regular type. You can use it to specify the return type of a function, declare function parameters, or variables in the function.

```
<Side Remark: invoke a function>
The `maxValue` function is invoked to return the maximum `int`, `double`, `char`, and `string` in lines 16-22. For the function call `maxValue(1, 3)`, the compiler recognizes that the parameter type is `int` and replaces the type parameter `T` with `int` to invoke the `maxValue` function with a concrete `int` type. For the function call `maxValue("ABC", "ABD")`, the compiler recognizes that the parameter type is `string` and replaces the type parameter `T` with `string` to invoke the `maxValue` function with a concrete `string` type.
```
The generic `maxValue` function can be used to return a maximum of two values of any type, provided that

- The two values have the same type;
- The two values can be compared using the `>` operator.

For example, if one value is `int` and the other is `double` (e.g., `maxValue(1, 3.5)`), the compiler will report a syntax error because it cannot find a match for the call. If you invoke `maxValue(Circle(1), Circle(2))`, the compiler will report a syntax error because the `<` operator is not defined in the `Circle` class.

**NOTE END**

TIP

You can use either `<typename T>` or `<class T>` to specify a type parameter. Using `<typename T>` is better because `<typename T>` is descriptive. `<class T>` could be confused with class declaration.

**NOTE**

Occasionally, a template function may have more than one parameter. In this case, place the parameters together inside the brackets, separated by commas, such as `<typename T1, typename T2, typename T3>`.

15.3 Example: A Generic Sort

Listing 6.9, `SelectionSort.cpp`, gives a function to sort an array of `double` values. Here is a copy of the function:

```cpp
void selectionSort(double list[], int arraySize) {
    for (int i = arraySize - 1; i >= 1; i--)
        // Find the maximum in the list[0..i]
        double currentMax = list[0];
        int currentMaxIndex = 0;
        for (int j = 1; j <= i; j++)
            if (currentMax < list[j])
                currentMax = list[j];
                currentMaxIndex = j;
```
It is easy to modify this function to write new overloaded functions for sorting an array of int values, char values, string values, etc. All you need to do is to replace the word `double` by `int`, `char`, or `string` in two places (lines 1 and 6).

Instead of writing several overloaded sort functions, you can define just one template function that work for any type. Listing 15.2 defines a generic function for sorting an array of elements.

Listing 15.2 GenericSort.cpp (Sorting Elements)

```cpp
#include <iostream>
#include <string>
using namespace std;

template<typename T T>
void sort(T list[], int arraySize)
{
    for (int i = arraySize - 1; i >= 1; i--)
    {
        // Find the maximum in the list[0..i]
        T currentMax = list[0];
        int currentMaxIndex = 0;

        for (int j = 1; j <= i; j++)
        {
            if (currentMax < list[j])
            {
                currentMax = list[j];
                currentMaxIndex = j;
            }
        }
        // Swap list[i] with list[currentMaxIndex] if necessary;
        if (currentMaxIndex != i)
        {
            list[currentMaxIndex] = list[i];
            list[i] = currentMax;
        }
    }
}
```

**PD: Please add line numbers in the following code**

**Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.**

*Side Remark line 5: template prefix*
*Side Remark line 6: type parameter*
*Side Remark line 11: type parameter*
*Side Remark line 32: template prefix*
*Side Remark line 33: type parameter*
*Side Remark line 11: type parameter*
*Side Remark line 45: invoke sort*
*Side Remark line 46: invoke printArray*
currentMaxIndex = j;
}

// Swap list[i] with list[currentMaxIndex] if necessary:
if (currentMaxIndex != i)
{
    list[currentMaxIndex] = list[i];
    list[i] = currentMax;
}


<template<

typename T>

void printArray( T list[], int arraySize)
{
    for (int i = 0; i < arraySize; i++)
    {
        cout << list[i] << " ";
    }
    cout << endl;
}

int main()
{
    int list1[] = {3, 5, 1, 0, 2, 8, 7};
    sort(list1, 7);
    printArray(list1, 7);

    double list2[] = {3.5, 0.5, 1.4, 0.4, 2.5, 1.8, 4.7};
    sort(list2, 7);
    printArray(list2, 7);

    string list3[] = {"Atlanta", "Denver", "Chicago", "Dallas"};
    sort(list3, 4);
    printArray(list3, 4);

    return 0;
}

<Output>
0 1 2 3 5 7 8
0.4 0.5 1.4 1.8 2.5 3.5 4.7
Atlanta Chicago Dallas Denver
<End Output>

Two template functions are defined in this program. The template function sort (lines 5-30) uses the type parameter T to specify the element type in an array. This function is identical to the selectionSort function except that the parameter double is replaced by a generic type T.
The template function `printArray` (lines 32-40) uses the type parameter `T` to specify the element type in an array. This function displays all the elements in the array to the console.

The main function invokes the `sort` function to sort an array of `int`, `double`, and `string` values (lines 45, 49, 53) and invokes the `printArray` function to display these arrays (lines 46, 50, 54).

TIP
<side remark: developing generic function>
When you define a generic function, it is better to start with non-generic function, debug and test it, and then convert it to a generic function.

15.4 Class Templates
In the preceding sections, you defined template functions with type parameters for the function. You can also define template classes with type parameters for the class. The type parameters can be used everywhere in the class where a regular type appears.

Recall that the `StackOfIntegers` class, defined in §11.11, can be used to create a stack for `int` values. Here is a copy of the class with its UML class diagram as shown in Figure 15.1(a):

```cpp
#ifndef STACK_H
#define STACK_H

class StackOfIntegers
{
  public:
    StackOfIntegers();
    bool empty();
    int peek();
    int push(int value);
    int pop();
    int getSize();

  private:
    int elements[100];
    int size;
}
#endif
```

***PD: Please add line numbers in the following code***
<Side Remark line 9: int type>
<Side Remark line 10: int type>
<Side Remark line 11: int type>
<Side Remark line 16: int type>
StackOfIntegers::StackOfIntegers()
    size = 0;

bool StackOfIntegers::empty()
    return (size == 0);

int StackOfIntegers::peek()
    return elements[size - 1];

int StackOfIntegers::push(int value)
    return elements[size++] = value;

int StackOfIntegers::pop()
    return elements[--size];

int StackOfIntegers::getSize()
    return size;

#endif

<table>
<thead>
<tr>
<th>StackOfIntegers</th>
<th>Stack&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>-elements: int[100]</td>
<td>-elements: T[100]</td>
</tr>
<tr>
<td>-size: int</td>
<td>-size: int</td>
</tr>
<tr>
<td>+StackOfIntegers()</td>
<td>+Stack()</td>
</tr>
<tr>
<td>+empty(): bool</td>
<td>+empty(): bool</td>
</tr>
<tr>
<td>+peek(): int</td>
<td>+peek(): T</td>
</tr>
<tr>
<td>+push(value: int): int</td>
<td>+push(value: T): T</td>
</tr>
<tr>
<td>+pop(): int</td>
<td>+pop(): T</td>
</tr>
<tr>
<td>+getSize(): int</td>
<td>+getSize(): int</td>
</tr>
</tbody>
</table>

![Figure 15.1](image)

**Figure 15.1**

Stack\(\langle T\rangle\) is a generic version of the Stack class.

By replacing the highlighted int in the preceding code with double, char or string, you can easily modify this class to define classes such as StackOfDouble, StackOfChar, and
StackOfStringLength for representing a stack of int, double, and string values. But, instead of writing almost identical code for these classes, you can define just one template class that work for the element of any type. Figure 15.1(b) shows the UML class diagram for the new generic Stack class. Listing 15.3 defines a generic stack class for storing elements of certain type.

Listing 15.3 GenericStack.h (Generic Stack Class)

```cpp
#ifndef STACK_H
#define STACK_H

template<typename T>
class Stack
{
public:
    Stack();
    bool empty();
    T peek();
    T push(T value);
    T pop();
    int getSize();

private:
    T elements[100];
    int size;
};

template<typename T>
Stack<T>::Stack()
    size = 0;

```

```cpp
ifndef STACK_H
define STACK_H

template<typename T>
class Stack
{
public:
    Stack();
    bool empty();
    T peek();
    T push(T value);
    T pop();
    int getSize();

private:
    T elements[100];
    int size;
};

template<typename T>
Stack<T>::Stack()
    size = 0;

```

```cpp
#endif
```

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

.Side Remark line 4: template prefix
.Side Remark line 10: type parameter
.Side Remark line 11: type parameter
.Side Remark line 16: type parameter
.Side Remark line 20: function template
.Side Remark line 26: function template
.Side Remark line 32: function template
.Side Remark line 38: function template
.Side Remark line 44: function template
.Side Remark line 50: function template
The syntax for class templates is basically the same as that for function templates. You place the template prefix before the class declaration (line 4), just like you place the template prefix before the function template.

The type parameter can be used in the class just like any regular data type. Here, the type T is used to declare functions peek() (line 10), push(T value) (line 11), and pop() (line 12). T is also use to declare array elements in line 16.

The constructors and functions are defined the same way for regular classes, except that the constructors and functions themselves are templates. So you have to place the template prefix before the constructor and function header. For example,
Stack<T>::Stack()
  
  size = 0;

template<typename T>
bool Stack<T>::empty()
  
  return (size == 0);

template<typename T>
T Stack<T>::peek()
  
  return elements[size - 1];

Also please note that the class name before the scope
resolution operator :: is Stack<T> not Stack.

TIP
(side remark: compile issue)
GenericStack.h combines class declaration and class
implementation into one file. Normally, you put class
declaration and class implementation into two separate
files. However, it is safer to put them together for
class templates, because some compliers cannot compile
them separately.

Listing 15.4 gives a test program that creates a stack for
int values in line 17 and a stack for strings in line 23.

Listing 15.4 TestGenericStack.cpp (Test Generic Stack Class)

```cpp
#include <iostream>
#include <string>
#include "GenericStack.h"
using namespace std;

int main()
{
  // Create a stack of int values
  Stack<int> intStack;
  for (int i = 0; i < 10; i++)
    intStack.push(i);
```

PD: Please add line numbers in the following code***

Layout: Please layout exactly. Don’t skip the space. This
is true for all source code in the book. Thanks, AU.

<Side Remark line 3: generic Stack>
<Side Remark line 9: int stack>
<Side Remark line 18: string stack>
while (!intStack.empty())
    cout << intStack.pop() << " ";
    cout << endl;

// Create a stack of strings
Stack<string> stringStack;
stringStack.push("Chicago");
stringStack.push("Denver");
stringStack.push("London");

while (!stringStack.empty())
    cout << stringStack.pop() << " ";
    cout << endl;

return 0;

<Output>
9 8 7 6 5 4 3 2 1 0
London Denver Chicago
<End Output>

<Side Remark: declaring objects>
To declare an object from a template class, you have to specify a concrete type for the type parameter T. For example,

Stack<int> intStack;

This declaration replaces the type parameter T with int. So, intStack is a stack for int values. The object intStack is just like any other objects. The program invokes the push function on intStack to add ten int values to the stack (lines 18-19), and displays the elements from the stack (lines 13-14).

The program declares a stack object for storing strings in line 18, adds three strings in the stack (lines 19-21), and displays the strings from the stack (line 23-24).

Note that the code in lines 13-15

    while (!intStack.empty())
        cout << intStack.pop() << " ";
        cout << endl;

and the in lines 23-25

    while (!stringStack.empty())
        cout << stringStack.pop() << " ";
        cout << endl;
These two fragments are almost identical. The difference is that the former operates on intStack and the latter is on stringStack. You can define a function with a stack parameter to display the elements in the stack. The new program is shown in Listing 14.5.

**NOTE**
<side remark: default type>
C++ allows you to assign a default type for a type parameter in a class template. For example, you may assign int as a default type in the generic Stack class as follows:

```cpp
template<typename T = int>
class Stack
{
...  
};
```

You can now declare an object using the default type like this:

```cpp
Stack<> stack;  // stack is a stack for int values
```

You can only use default type in class templates, not in function templates.

***END OF NOTE

NOTE

<side remark: Nontype parameter>

You can also use nontype parameters along with type parameters in a template prefix. For example, you may declare the array capacity as a parameter for the Stack class as follows:

```cpp
template<typename T, int capacity>
class Stack
{
...  
private:
    T elements[capacity];
    int size;
};
```

So, when you create a stack, you can specify the capacity for the array. For example,

```cpp
Stack<string, 500> stack;
```

declares a stack that can hold up to 500 strings.

***END OF NOTE

NOTE

<side remark: templates and inheritance>
A nontemplate class can be derived from a class template specialization. A class template can be derived from a nontemplate class. A class template can be derived from a class template.

***END OF NOTE

NOTE

<side remark: template class friends>
Friends are used exactly the same for template and nontemplate classes.

***END OF NOTE

NOTE

<side remark: static members>
You can define static members in a template class. Each template specialization has its own copy of a static data field.

***END OF NOTE

15.5 Improving the Stack Class

There is a problem in the Stack class. The elements of the stack are stored in an array with a fixed size 100 (see line 16 in Listing 15.3). So, you cannot store more than 100 elements in a stack. You may change 100 to a larger number. But this would waste space if the actual stack is small. One way to resolve this dilemma is to allocate more memory when needed.

The size property in the Stack<T> class represents the number of elements in the stack. Let us add a new property named capacity that represents the current size of the array for storing the elements. The no-arg constructor of Stack<T> creates an array with capacity 16. When you add a new element to the stack, you may need to increase the array size in order to store the new element if the current capacity is full.

How do you increase the array capacity? You cannot increase the array capacity once the array is declared. To circumvent this restriction, you may create a new larger size array, copy the contents of the old array to this new array, and destroys the old array.

The improved Stack<T> class is shown in Listing 15.6.

Listing 15.6 ImprovedStack.h (Improved Stack Class)
```cpp
#ifndef IMPROVEDSTACK_H
#define IMPROVEDSTACK_H

template<typename T>
class Stack
{
public:
    Stack();
    bool empty();
    T peek();
    T push(T value);
    T pop();
    int getSize();

private:
    T *elements;
    int size;
    int capacity;
    void ensureCapacity();
};

template<typename T>
Stack<T>::Stack()
{ size(0), capacity(16) elements = new T[capacity];
}

template<typename T>
bool Stack<T>::empty()
{ return (size == 0); }

template<typename T>
T Stack<T>::peek()
{ return elements[size - 1]; }

template<typename T>
T Stack<T>::push(T value)
{ ensureCapacity();
    return elements[size++] = value;
}
```
template<typename T>
void Stack<T>::ensureCapacity()
{
    if (size >= capacity)
    {
        T *old = elements;
        capacity = 2 * size;
        elements = new T[size * 2];
        for (int i = 0; i < size; i++)
            elements[i] = old[i];
        delete old;
    }
}

template<typename T>
T Stack<T>::pop()
{
    return elements[--size];
}

template<typename T>
int Stack<T>::getSize()
{
    return size;
}

The push(T value) function (lines 40-45) adds a new element to the stack. This function first invokes ensureCapacity() (line 43), which ensures that there is a space in the array for the new element.

The ensureCapacity() function (lines 47-61) checks whether the array is full. If so, create a new array that doubles the current array size, set the new array as the current array, copy the old array to the new array, and deletes the old array (line 59).

Key Terms

***PD: Please place terms in two columns same as in intro5e.

- template class
- template function
- template prefix
- type parameter
Chapter Summary

- Templates provide the capability to parameterize types in functions and classes. With this capability, you can define one function or one class with a generic type that can be substituted for a concrete type by the compiler.

- The definition for the function template begins with the keyword `template` followed by a list of parameters. Each parameter must be preceded by the interchangeable keywords `class` or `typename` in the form `<typename typeParameter>` or `<class typeParameter>`.

- The generic `maxValue` function can be used to return a maximum of two values of any type, provided that (1) the two values have the same type; and (2) The two values can be compared using the `>` operator.

- When you define a generic function, it is better to start with non-generic function, debug and test it, and then convert it to a generic function.

- The syntax for class templates is basically the same as that for function templates. You place the template prefix before the class declaration, just like you place the template prefix before the function template.

- The constructors and functions are defined the same way for regular classes, except that the constructors and functions themselves are templates. So you have place the template prefix before the constructor and function header.

REVIEW QUESTIONS

Section 15.2 Templates Basics

15.1
For the `maxValue` function in Listing 15.1, can you invoke it with two arguments of different types such as `maxValue(1, 1.5)`?

15.2
For the `maxValue` function in Listing 15.1, can you invoke it with two arguments of strings such as `maxValue("ABC", "ABD")`? can you invoke it with two arguments of circles such as `maxValue(Circle(2), Circle(3))`?
Can `template<typename T>` be replaced by `template<class T>`?

15.4 Can a type parameter be named using any identifier other than a keyword?

`template<typename T>` be replaced by `template<class T>`?

15.5 Can a type parameter be of a primitive type or an object type?

15.6 What is wrong in the following code?

```cpp
#include <iostream>
#include <string>
using namespace std;

template<typename T>
T maxValue(const T &value1, const T &value2)
{
    int result;
    if (value1 > value2)
        result = value1;
    else
        result = value2;
    return result;
}

int main()
{
    cout << "Maximum between 1 and 3 is " << maxValue(1, 3) << endl;
    cout << "Maximum between 1.5 and 0.3 is " << maxValue(1.5, 0.3) << endl;
    cout << "Maximum between 'A' and 'N' is " << maxValue('A', 'N') << endl;
    cout << "Maximum between "ABC" and "ABD" is " << maxValue("ABC", "ABD") << endl;

    return 0;
}
```

15.7 If you define the `maxValue` function as follows:

```cpp
template<typename T1, typename T2>
T1 maxValue(const T1 &value1, const T2 &value2)
{
    if (value1 > value2)
        return value1;
```
else
    return value2;

What would be the return value from invoking `maxValue(1, 2.5)`, `maxValue(1.4, 2.5)`, and `maxValue(1.5, 2)`?

15.8
If you define the `swap` function as follows:

```cpp
template<typename T>
void swap(T &var1, T &var2)
{
    T temp = var1;
    var1 = var2;
    var2 = temp;
}
```

What is wrong in the following code?

```cpp
***PD: Please add line numbers in the following code***
int main()
{
    int v1 = 1;
    int v2 = 2;
    swap(v1, v2);

    double d1 = 1;
    double d2 = 2;
    swap(d1, d2);

    swap(v1, d2);
    swap(1, 2);

    return 0;
}
```

Section 15.4 Class Templates
15.9
Do you have to use the template prefix for each function in the class declaration? Do you have to use the template prefix for each function in the class implementation?

15.10. What is wrong in the following code?

```cpp
template<typename T = int>
void printArray(T list[], int arraySize)
{
    for (int i = 0; i < arraySize; i++)
    {
        cout << list[i] << " ";
    }
```
15.11
What is wrong in the following code?

***PD: Please add line numbers in the following code***

```cpp
template<typename T>
class Foo{
public:
    Foo();
    T f1(T value);
    T f2();
};

Foo::Foo()
{
    ...
}

T Stack::f1(T value)
{
    ...
}

T Stack::f2()
{
    ...
}
#endif
```

15.12
Suppose the template prefix for the Stack class is

```cpp
template<typename T = string>
```

Can you create a stack of strings using

```cpp
Stack stack;
```

PROGRAMMING EXERCISES

15.1
(Maximum in array) Design a generic function that returns a maximum element from an array. The function should have two parameters. One is the array of a generic type and the other is the size of the array. Test the function with array of int, double, string, and rational values.
15.2
(Binary search) Rewrite the binary search function in Listing 6.8, BinarySearch.cpp, to use a generic type for array elements. Test the function with array of int, double, string, and rational values.

15.3
(Recursive binary search) Rewrite the recursive binary search function in Listing 8.6, RecursiveBinarySearch.cpp, to use a generic type for array elements. Test the function with array of int, double, string, and rational values.

15.4
(Insertion sort) Rewrite the insertion search function in Listing 6.10, InsertionSort.cpp, to use a generic type for array elements. Test the function with array of int, double, string, and rational values.

15.5
(Swap values) Write a generic function that swap values in two variables. Your function should have two parameters of the same type. Test the function with int, double, string, and rational values.

15.6*
(Add two matrices) Write a generic function that adds two matrices. Your function should have two parameters of the two-dimensional array. Both arrays have the same generic type. The function returns an array. Test the function with int, double, and rational arrays.

15.7**
(Multiply two matrices) Write a generic function that multiplies two matrices. Your function should have two parameters of the two-dimensional array. Both arrays have the same generic type. The function returns an array. Test the function with int, double, and rational arrays.

15.8*
(Function printStack) Add the printStack function into the Stack class as an instance function to display all the elements in the stack. The Stack class was introduced in Listing 15.3, GenericStack.h.

15.9*
(Function `contains`) Add the `contains(T element)` function into the `Stack` class as an instance function to check whether the element is in the stack. The `Stack` class was introduced in Listing 15.3, `GenericStack.h`.

15.10*

(Implementing `vector` class) The `vector` class is provided in the standard C++ library. Implement the `vector` class as an exercise.