



# PROBLEM SOLVING

WITH

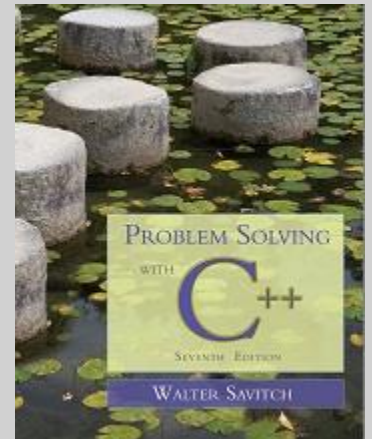
C++

SEVENTH EDITION

WALTER SAVITCH

# Chapter 17

## Templates



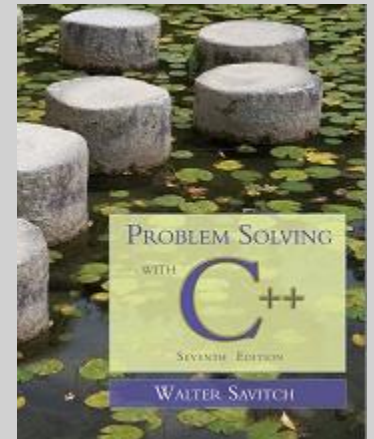
# Overview

17.1 Templates for Algorithm Abstraction

17.2 Templates for Data Abstraction

# 17.1

## Templates for Algorithm Abstraction



# Templates for Algorithm Abstraction

- Function definitions often use application specific adaptations of more general algorithms
  - For example: The general algorithm used in `swap_values` could swap variables of any type:

```
void swap_values(type_of_var& v1,  
                type_of_var& v2)  
{  
    type_of_var temp;  
    temp = v1;  
    v1 = v2;  
    v2 = temp;  
}
```

# swap\_values for char

- Here is a version of swap\_values to swap character variables:

- ```
void swap_values(char& v1, char& v2)
{
    char temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```


# A General swap\_values

- A generalized version of swap\_values is shown here.
  - ```
void swap_values(type_of_var& v1, type_of_var& v2)
{
    type_of_var temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```
  - This function, if `type_of_var` could accept any type, could be used to swap values of any type

# Templates for Functions

- A C++ function template will allow swap\_values to swap values of two variables of the same type

- Example:

**Template prefix** → `template<class T>`  **Type parameter**

```
void swap_values(T& v1, T& v2)
{
    T temp;
    temp = v1;
    v1 = v2;
    v = temp;
}
```



# Template Details

- `template<class T>` is the template prefix
  - Tells compiler that the declaration or definition that follows is a template
  - Tells compiler that `T` is a type parameter
    - `class` means type in this context (typename could replace `class` but `class` is usually used)
    - `T` can be replaced by any type argument (whether the type is a class or not)
- A template overloads the function name by replacing `T` with the type used in a function call

# Calling a Template Function

- Calling a function defined with a template is identical to calling a normal function
  - Example:
    - To call the template version of `swap_values`
    - `char s1, s2;`
    - `int i1, i2;`
    - `...`
    - `swap_values(s1, s2);`
    - `swap_values(i1, i2);`
  - The compiler checks the argument types and generates an appropriate version of `swap_values`

# Templates and Declarations

- A function template may also have a separate declaration
  - The template prefix and type parameter are used
  - Depending on your compiler
    - You may, or may not, be able to separate declaration and definitions of template functions just as you do with regular functions
  - To be safe, place template function definitions in the same file where they are used...with no declaration
    - A file included with `#include` is, in most cases, equivalent to being "in the same file"
    - This means including the `.cpp` file or `.h` file with implementation code

# The Type Parameter T

- T is the traditional name for the type parameter
  - Any valid, non-keyword, identifier can be used
  - "VariableType" could be used

```
template <class VariableType>
void swap_values(VariableType& v1,
                 VariableType& v2)
{
    VariableType temp;
    ...
}
```

**Display 17.1**

# Templates with Multiple Parameters

- Function templates may use more than one parameter

- Example:

```
template<class T1, class T2>
```

- All parameters must be used in the template function

# Algorithm Abstraction

- Using a template function we can express more general algorithms in C++
- Algorithm abstraction means expressing algorithms in a very general way so we can ignore incidental detail
  - This allows us to concentrate on the substantive part of the algorithm

# Program Example: A Generic Sorting Function

- The sort function below uses an algorithm that does not depend on the base type of the array

```
void sort(int a[], int number_used)
{
    int index_of_next_smallest;
    for (int index = 0; index < number_used - 1; index++)
    {
        index_of_next_smallest =
            index_of_smallest(a, index, number_used);
        swap_values(a[index], a[index_of_next_smallest]);
    }
}
```

- The same algorithm could be used to sort an array of any type

# Generic Sorting: Helping Functions

- sort uses two helper functions
  - index\_of\_smallest also uses a general algorithm and could be defined with a template
  - swap\_values has already been adapted as a template
- All three functions, defined with templates, are demonstrated in

**Display 17.2**

**Display 17.3 (1-2)**



# Templates and Operators

- The function `index_of_smallest` compares items in an array using the `<` operator
  - If a template function uses an operator, such as `<`, that operator must be defined for the types being compared
  - If a class type has the `<` operator overloaded for the class, then an array of objects of the class could be sorted with function template `sort`

# Defining Templates

- When defining a template it is a good idea...
  - To start with an ordinary function that accomplishes the task with one type
    - It is often easier to deal with a concrete case rather than the general case
  - Then debug the ordinary function
  - Next convert the function to a template by replacing type names with a type parameter

# Inappropriate Types for Templates

- Templates can be used for any type for which the code in the function makes sense
  - `swap_values` swaps individual objects of a type
  - This code would not work, because the assignment operator used in `swap_values` does not work with arrays:

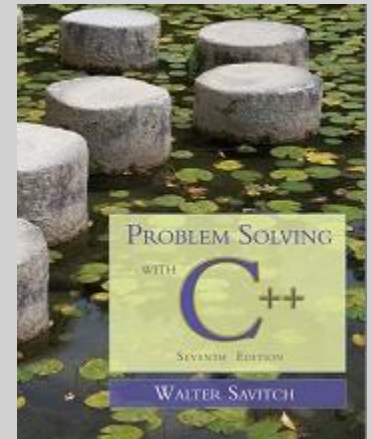
```
int a[10], b[10];  
<code to fill the arrays>  
swap_values(a, b);
```

# Section 17.1 Conclusion

- Can you
  - Identify a template prefix?
  - Identify a parameter type in a template prefix?
  - Compare and contrast function overloading with the use of templates?
    - What additional complexities are involved when class types are involved as parameter types?

# 17.2

## Templates for Data Abstraction



# Templates for Data Abstraction

- Class definitions can also be made more general with templates
  - The syntax for class templates is basically the same as for function templates
    - `template<class T>` comes before the template definition
    - Type parameter `T` is used in the class definition just like any other type
    - Type parameter `T` can represent any type

# A Class Template

- The following is a class template
  - An object of this class contains a pair of values of type T
  - ```
template <class T>
class Pair
{
    public:
        Pair( );
        Pair( T first_value, T second_value);

        ...

```

continued on next slide

# Template Class Pair (cont.)

- `void set_element(int position, T value);`  
//Precondition: position is 1 or 2  
//Postcondition: position indicated is set to value

`T get_element(int position) const;`  
// Precondition: position is 1 or 2  
// Returns value in position indicated

`private:`  
    `T first;`  
    `T second;`  
`};`



# Declaring Template Class Objects

- Once the class template is defined, objects may be declared
  - Declarations must indicate what type is to be used for T
  - Example: To declare an object so it can hold a pair of integers:

`Pair<int> score;`

or for a pair of characters:

`Pair<char> seats;`

# Using the Objects

- After declaration, objects based on a template class are used just like any other objects
  - Continuing the previous example:

```
score.set_element(1,3);  
score.set_element(2,0);  
seats.set_element(1, 'A');
```


# Defining the Member Functions

- Member functions of a template class are defined the same way as member functions of ordinary classes
  - The only difference is that the member function definitions are themselves templates

# Defining a Pair Constructor

- This is a definition of the constructor for class Pair that takes two arguments

```
template<class T>
Pair<T>::Pair(T first_value, T second_value)
    : first(first_value), second(second_value)
{
    //No body needed due to initialization above
}
```



- The class name includes <T>

# Defining set\_element

- Here is a definition for set\_element in the template class Pair

```
void Pair<T>::set_element(int position, T value)
{
    if (position == 1)
        first = value;
    else if (position == 2)
        second = value;
    else
        ...
}
```

# Template Class Names as Parameters

- The name of a template class may be used as the type of a function parameter
  - Example: To create a parameter of type `Pair<int>`:

```
int add_up(const Pair<int>& the_pair);  
//Returns the sum of two integers in the_pair
```

# Template Functions with Template Class Parameters

- Function `add_up` from a previous example can be made more general as a template function:

```
template<class T>  
T add_up(const Pair<T>& the_pair)  
    //Precondition: operator + is defined for T  
    //Returns sum of the two values in the_pair
```

# Program Example: An Array Class

- The example in the following displays is a class template whose objects are lists
  - The lists can be lists of any type
- The interface is found in

Display 17.4 (1-2)

The program is in

Display 17.5

The implementation is in

Display 17.6 (1-3)



# typedef and Templates

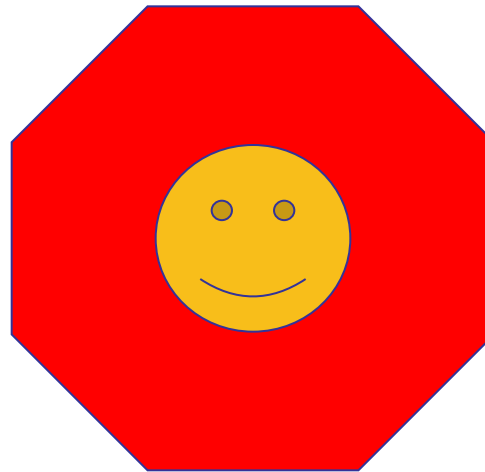
- You specialize a class template by giving a type argument to the class name such as `Pair<int>`
  - The specialized name, `Pair<int>`, is used just like any class name
- You can define a new class type name with the same meaning as the specialized name:  
`typedef Class_Name<Type_Arg>`  
`New_Type_Name;`

For example:      `typedef Pair<int> PairOfInt;`  
                     `PairOfInt pair1, pair2;`

# Section 17.2 Conclusion

- Can you
  - Give the definition for the member function `get_element` for the class template `Pair`?
  - Give the definition for the constructor with zero arguments for the template class `Pair`?

# Chapter 17 -- End



## A Function Template

```
//Program to demonstrate a function template.
#include <iostream>
using namespace std;

//Interchanges the values of variable1 and variable2.
template<class T>
void swap_values(T& variable1, T& variable2)
{
    T temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

int main()
{
    int integer1 = 1, integer2 = 2;
    cout << "Original integer values are "
         << integer1 << " " << integer2 << endl;
    swap_values(integer1, integer2);
    cout << "Swapped integer values are "
         << integer1 << " " << integer2 << endl;

    char symbol1 = 'A', symbol2 = 'B';
    cout << "Original character values are "
         << symbol1 << " " << symbol2 << endl;
    swap_values(symbol1, symbol2);
    cout << "Swapped character values are "
         << symbol1 << " " << symbol2 << endl;

    return 0;
}
```

## Output

```
Original integer values are 1 2
Swapped integer values are 2 1
Original character values are A B
Swapped character values are B A
```

# Display 17.1



# Display 17.2



## DISPLAY 17.2 A Generic Sorting Function

```
1  // This is file sortfunc.cpp
2
3  template<class T>
4  void swap_values(T& variable1, T& variable2)
5      <The rest of the definition of swap_values is given in Display 17.1.>
6
7  template<class BaseType>
8  int index_of_smallest(const BaseType a[], int start_index, int number_used)
9  {
10     BaseType min = a[start_index];
11     int index_of_min = start_index;
12
13     for (int index = start_index + 1; index < number_used; index++)
14         if (a[index] < min)
15         {
16             min = a[index];
17             index_of_min = index;
18             //min is the smallest of a[start_index] through a[index]
19         }
20     return index_of_min;
21 }
22
23 template<class BaseType>
24 void sort(BaseType a[], int number_used)
25 {
26     int index_of_next_smallest;
27     for(int index = 0; index < number_used - 1; index++)
28     {
29         //Place the correct value in a[index]:
30         index_of_next_smallest =
31             index_of_smallest(a, index, number_used);
32         swap_values(a[index], a[index_of_next_smallest]);
33         //a[0] <= a[1] <= ... <= a[index] are the smallest of the original array
34         //elements. The rest of the elements are in the remaining positions.
35     }
36 }
```

# Display 17.3 (1/2)

## Using a Generic Sorting Function (part 1 of 2)

```
//Demonstrates a generic sorting function.
#include <iostream>
using namespace std;

//The file sortfunc.cpp defines the following function:
//template<class BaseType>
//void sort(BaseType a[], int number_used);
//Precondition: number_used <= declared size of the array a.
//The array elements a[0] through a[number_used - 1] have values.
//Postcondition: The values of a[0] through a[number_used - 1] have
//been rearranged so that a[0] <= a[1] <= ... <= a[number_used - 1].
```

```
#include "sortfunc.cpp"
```

```
int main()
{
    int i;
    int a[10] = {9, 8, 7, 6, 5, 1, 2, 3, 0, 4};
    cout << "Unsorted integers:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;
    sort(a, 10);
    cout << "In sorted order the integers are:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;

    double b[5] = {5.5, 4.4, 1.1, 3.3, 2.2};
    cout << "Unsorted doubles:\n";
    for (i = 0; i < 5; i++)
        cout << b[i] << " ";
    cout << endl;
    sort(b, 5);
    cout << "In sorted order the doubles are:\n";
    for (i = 0; i < 5; i++)
        cout << b[i] << " ";
    cout << endl;
}
```

Many compilers will allow this function declaration to appear as a function declaration and not merely as a comment. However, including the function declaration is not needed, since the definition of the function is in the file `sortfunc.cpp`, and so the definition effectively appears before `main`.



```
char c[7] = {'G', 'E', 'N', 'E', 'R', 'I', 'C'};
cout << "Unsorted characters:\n";
for (i = 0; i < 7; i++)
    cout << c[i] << " ";
cout << endl;
sort(c, 7);
cout << "In sorted order the characters are:\n";
for (i = 0; i < 7; i++)
    cout << c[i] << " ";
cout << endl;

return 0;
}
```

### Output

```
Unsorted integers:
9 8 7 6 5 1 2 3 0 4
In sorted order the integers are:
0 1 2 3 4 5 6 7 8 9
Unsorted doubles:
5.5 4.4 1.1 3.3 2.2
In sorted order the doubles are:
1.1 2.2 3.3 4.4 5.5
Unsorted characters:
G E N E R I C
In sorted order the characters are:
C E E G I N R
```

## Display 17.3 (2/2)



# Display 17.4 (1/2)



## DISPLAY 17.4 Interface for the Class Template GenericList (part 1 of 2)

```
1  //This is the header file genericlist.h. This is the interface for the
2  //class GenericList. Objects of type GenericList can be a list of items
3  //of any type for which the operators << and = are defined.
4  //All the items on any one list must be of the same type. A list that
5  //can hold up to max items all of type Type_Name is declared as follows:
6  //    GenericList<Type_Name> the_object(max);
7  #ifndef GENERICLIST_H
8  #define GENERICLIST_H
9  #include <iostream>
10 using namespace std;
11
12 namespace listsavitch
13 {
14     template<class ItemType>
15     class GenericList
16     {
17     public:
18         GenericList(int max);
19         //Initializes the object to an empty list that can hold up to
20         //max items of type ItemType.
21
22         ~GenericList();
23         //Returns all the dynamic memory used by the object to the freestore.
24
25         int length() const;
26         //Returns the number of items on the list.
27
28         void add(ItemType new_item);
29         //Precondition: The list is not full.
30         //Postcondition: The new_item has been added to the list.
31
32         bool full() const;
33         //Returns true if the list is full.
34
```

(continued)



# Display 17.4

## (2/2)



### DISPLAY 17.4 Interface for the Class Template `GenericList` (part 2 of 2)

---

```
35         void erase();
36         //Removes all items from the list so that the list is empty.
37
38         friend ostream& operator <<(ostream& outs,
39                                     const GenericList<ItemType>& the_list);
40         //Overloads the << operator so it can be used to output the
41         //contents of the list. The items are output one per line.
42         //Precondition: If outs is a file output stream, then outs has
43         //already been connected to a file.
44     private:
45         ItemType *item; //pointer to the dynamic array that holds the list.
46         int max_length; //max number of items allowed on the list.
47         int current_length; //number of items currently on the list.
48     };
49 } //listsavitch
50 #endif //GENERICLIST_H
```

---

# Display 17.5

## 1/2



### DISPLAY 17.5 Program Using the GenericList Class Template (part 1 of 2)

```
1  //Program to demonstrate use of the class template GenericList.
2  #include <iostream>
3  #include "genericlist.h"
4  #include "genericlist.cpp"
5  using namespace std;
6  using namespace listsavitch;
7  int main()
8  {
9      GenericList<int> first_list(2);
10     first_list.add(1);
11     first_list.add(2);
12     cout << "first_list = \n"
13          << first_list;
14     GenericList<char> second_list(10);
15     second_list.add('A');
16     second_list.add('B');
17     second_list.add('C');
18     cout << "second_list = \n"
19          << second_list;
20     return 0;
21 }
```

*Since genericlist.cpp is included, you need compile only this one file (the one with the main).*

(continued)

# Display 17.5

## 2/2



### **DISPLAY 17.5** Program Using the GenericList Class Template *(part 2 of 2)*

---

#### ***Output***

```
first_list =  
1  
2  
second_list =  
A  
B  
C
```

# Display 17.6

## 1/3



### DISPLAY 17.6 Implementation of GenericList (part 1 of 3)

---

```
1  //This is the implementation file: genericlist.cpp
2  //This is the implementation of the class template named GenericList.
3  //The interface for the class template GenericList is in the
4  //header file genericlist.h.
5  #ifndef GENERICLIST_CPP
6  #define GENERICLIST_CPP
7  #include <iostream>
8  #include <cstdlib>
9  #include "genericlist.h"//This is not needed when used as we are using this file,
10                          //but the #ifndef in genericlist.h makes it safe.
11  using namespace std;
12
13  namespace listsavitch
14  {
15      //Uses cstdlib:
16      template<class ItemType>
17      GenericList<ItemType>::GenericList(int max) : max_length(max),
  current_length(0)
18
19      {
20          item = new ItemType[max];
21      }
22
23      template<class ItemType>
24      GenericList<ItemType>::~GenericList()
```

(continued)

# Display 17.6 (2/3)



## DISPLAY 17.6 Implementation of GenericList (part 2 of 3)

```
25     {
26         delete [] item;
27     }
28
29     template<class ItemType>
30     int GenericList<ItemType>::length() const
31     {
32         return (current_length);
33     }
34
35     //Uses iostream and cstdlib:
36     template<class ItemType>
37     void GenericList<ItemType>::add(ItemType new_item)
38     {
39         if ( full() )
40         {
41             cout << "Error: adding to a full list.\n";
42             exit(1);
43         }
44         else
45         {
46             item[current_length] = new_item;
47             current_length = current_length + 1;
48         }
49     }
50
51     template<class ItemType>
52     bool GenericList<ItemType>::full() const
53     {
54         return (current_length == max_length);
55     }
56
57     template<class ItemType>
58     void GenericList<ItemType>::erase()
59     {
60         current_length = 0;
61     }
62
63     //Uses iostream:
64     template<class ItemType>
65     ostream& operator <<(ostream& outs, const GenericList<ItemType>& the_list)
```

(continued)

# Display 17.6

## (3/3)



### DISPLAY 17.6 Implementation of GenericList (part 3 of 3)

---

```
66     {
67         for (int i = 0; i < the_list.current_length; i++)
68             outs << the_list.item[i] << endl;
69
70         return outs;
71     }
72 } //listsavitch
73 #endif // GENERICLIST_CPP Notice that we have enclosed all the template
74     // definitions in #ifndef... #endif.
```

A note is in order about compiling the code from Displays 17.4, 17.5, and 17.6. A safe solution to the compilation of this code is to `#include` the template class definition and the template function definitions before use, as we did. In that case only the file in Display 17.5 needs to be compiled. Be sure that you use the `#ifndef` `#define` `#endif` mechanism to prevent multiple file inclusion of all the files you are going to `#include`.

---