

UCLA Stat 130D Statistical Computing and Visualization in C++

Instructor: Ivo Dinov, Asst. Prof. in
Statistics / Neurology

University of California, Los Angeles, Winter 2004
http://www.stat.ucla.edu/~dinov/courses_students.html

Templates for More Abstraction

- Templates for Algorithm Abstraction
 - Templates for Functions
- Templates for Data Abstraction
 - Syntax for Class Templates

7

Templates for More Abstraction

- The C++ **template** facility provides the ability to define functions and classes that have parameter that carry type information. This is the generic facility of C++.
- A **generic facility** allows code to be written once then reused with different types.
- The classes and functions you will define in this chapter are much more general than before.
- It is widely held that a generic facility is one of the *most important* ideas in programming.

8

Templates for Algorithm Abstraction(1 of 2)

- In many functions that are written, the algorithm is more general than the implementation.
- This swap algorithm only works for *int* values:

```
void swap ( int & v1, int & v2)
{
    int temp = v1;
    v1 = v2;
    v2 = temp;
}
```
- If *int* is replaced by *double*, we have another valid swap routine.
- In fact, any type can be used if the type has a copy constructor and assignment defined.
- Do you see where these are needed?

9

Templates for Algorithm Abstraction(2 of 2)

- In many functions that are written, the algorithm is more general than the implementation.
- This swap algorithm only works for *int* values:

```
void swap ( int & v1, int & v2)
{
    int temp = v1;
    v1 = v2;
    v2 = temp;
}
```
- If *int* is replaced by *double*, we have another valid swap routine.
- In fact, any type can be used if the type has a copy constructor and assignment defined.
- Do you see where these are needed?

10

```
//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;
}
```

Templates for Functions(1 of 3)

- shows the syntax for a C++ template function, swap_values.
- The definition and prototype begin with the line `template<class T>`
- This is called the **template prefix**. It tells the compiler that the definition that follows is a **template**. It also provides a definition for the **type parameter** T within the template definition.
- The word *class* in the template prefix means *type*.
- The template definition provides a definition of the function for every possible type *from which only the required definition is selected*.

12

Templates for Functions(2 of 3)

- Calling a template function is easy. Write code as if the needed definition were present.
- When the compiler sees a function call, it searches for a ordinary function name with a sequence of types in the parameter list. If a definition that matches the name and parameter list types is found, that one is used.
- If no matches are present, then the compiler uses a template function definition with the correct number of parameters. If the compiler can deduce the types of the type parameters, then the compiler uses the template to produce a function with the deduced types.
- The rules are more complicated, but this is sufficient in most simple situations.

13

Templates for Functions(3 of 3)

- We put the template definition above before the main function, and used no prototype.
- A template function may have a template prototype, but separating the prototype from the definition is not usually possible with today's compiler technology.
- The safest strategy is to put the template definition in the same file where it is used, prior to use of the template.
- Most template functions need only one template parameter.
- Use of more than one type parameter is always possible. Consider, for example:

14

Function Template (1 of 2)

The function definition and the function prototype for a function

template are each prefaced with the following:
`template<class Type_Parameter>`

The prototype (if used) and definition are then the same as any ordinary function prototype and definition, except that the *Type_Parameter* can be used in place of a type.

Example,

```
template<class T>
void show_stuff( int stuff1, T stuff2, T stuff3);
template<class T>
void show_stuff( int stuff1, T stuff2, T stuff3)
{ cout << stuff1 << endl << stuff2 << endl << stuff3
<< endl;
}
```

15

Function Template (2 of 2)

The function given in this example is equivalent to having one function prototype and one function definition for each possible type name. The type name is substituted for the type parameter (which is T in the example here.) For instance consider the following function call:

```
show_stuff(2, 3.3, 4.4);
```

When this function call is executed, the compiler uses the function definition obtained by replacing T with the type name double. A separate definition will be produced for each different type for which you use the template, but not for any types you do not use. Only one definition should be generated regardless of the number of times you use the template.

NOTE: Some compilers require special instructions to ensure that only one copy of the function is generated. You won't see that problem in this course.

16

Algorithm Abstraction(1 of 2)

As we saw in our discussion of the swap_values function, there is a very general algorithm for interchanging the values of two variables and this more general algorithm applies to variables of any type.

Using a function template we were able to express this more general algorithm in C++. This is a very simple example of *algorithm abstraction*, where we say we are using **algorithm abstraction** we mean that we are expressing our algorithms in a very general way so that we can ignore incidental detail and concentrate on the substantive part of the algorithm. Function template are one feature of C++ that supports algorithm abstraction.

17

```
// A Generic Sorting Function (1 of 2)
// This is file sortfunc.cxx
template<class T>
void swap_values(T& variable1, T& variable2)
{
    T temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

template<class T>
int index_of_smallest(const T a[], int start_index, int number_used)
{
    T min = a[start_index];
    int index_of_min = start_index;
    for (int index = start_index + 1; index < number_used; index++)
        if (a[index] < min)
        {
            min = a[index];
            index_of_min = index;
            //min is the smallest of a[start_index] through a[index]
        }
    return index_of_min;
}
```

```
// A Generic Sorting Function (2 of 2)

template<class T>
void sort(T a[], int number_used)
{
    int index_of_next_smallest;
    for(int index = 0; index < number_used - 1; index++)
        { //Place the correct value in a[index]:
            index_of_next_smallest =
                index_of_smallest(a, index, number_used);
            swap_values(a[index], a[index_of_next_smallest]);
            //a[0] <= a[1] <=...<= a[index] are the smallest of the original
            array
            //elements. The rest of the elements are in the remaining
            positions.
        }
}
```

```
// Using a Generic Sorting Function (1 of 2)
//Demonstrates a generic sorting function.
#include <iostream>
using namespace std;

//The file sortfunc.cxx defines the following function:
//template<class T>
//void sort(T 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.cxx"
int main()
{
    int i, 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;
}
```

```
// Using a Generic Sorting Function (2 of 2)

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;
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;
}
```

Programming Tip: How to define Templates

- We created the template by writing an ordinary function that sorts for base type int.
- We created the template by replacing the base type of the array with type parameter T.
- This process has two advantages:
 1. The ordinary function is concrete which is easier to visualize.
 2. You are dealing with fewer details. You don't have to worry about template syntax rules.

22

Templates for Data Abstraction Syntax for Class Templates

- class template syntax requires a template prefix:


```
template<class T>
```
- where T is a type name. As in function templates, T can be a built-in type or a class.
- Example:


```
template<class T>
class Pair
{public:
    Pair();
    Pair(T first_value, T second_value);
    void set_element(int position, T value);
    T get_element(int position) const;
private:
    T first;
    T second;
};
```

23

Syntax for Class Templates(1 of 3)

- Once the class template is defined, you can define objects of the class:
`Pair<int> score1;`
`Pair<char> seats1;`
- Using the parameterized constructor
`Pair<int> score2(1, 2);`
`Pair<char> seats2('A', 'B');`
- The Objects are like any other objects. We apply the member function `set_element`:
`score1.set_element(1, 3);`
`score1.set_element(2, 0);`

24

Syntax for Class Templates(2 of 3)

Note the class name `Pair<T>` used for scope resolution, not just `Pair`.

```
// Uses iostream and cstdlib:  
template<class T>  
void Pair<T>::set_element(int position, T value)  
{  
    if (position == 1)  
        first = value;  
    else if (position == 2)  
        second = value;  
    else  
        { cout << "Error: Illegal pair position.\n";  
          exit(1);  
        }  
}
```

25

Syntax for Class Templates(3 of 3)

The constructors are:

```
template <class T>  
Pair<T>::Pair()  
{  
    //In the default constructor,  
    you  
    first = T(); //can initialize first and  
    second  
    second = T(); //using the default T's  
    constructor.  
}  
template <class T>  
Pair<T>::Pair(T first_value, T second_value)  
{  
    first = first_value;  
    second = second_value;  
}
```

26

Class Template Syntax (1 of 2)

The class definition and the definitions of the member functions are prefaced with the following:

```
template<class Type_Parameter>
```

The class and member function definitions are then the same as for any ordinary class except that `Type_Parameter` can be used in place of a type.

For example, the following is the beginning of a class template definition:

```
template <class T>  
class Pair  
{ public:  
    Pair();  
    Pair(T first_value, T second_value);  
    ...  
};
```

27

Class Template Syntax (2 of 2)

Member functions and overloaded operators are then defined as function templates. For example, the definition of the two argument **constructor** for the above sample class template would begin:

```
template <class T>  
Pair<T>::Pair(T first_value, T second_value)  
{  
    ...  
}
```

28

Type Definitions

You can specialize a class template by giving a type argument to the class name as in the following example:

```
Pair<int>
```

The specialized class name, like `Pair<int>`, can then be used just like any class name. It can be used to declare objects or to specify the type of a formal parameter.

You can define a new class type name that has the same meaning as a specialized class template name, such as `Pair<int>`. The syntax for such a defined class type name is:

```
typedef Class_Name<Type_argument>  
New_Type_Name;
```

For example: `typedef Pair<int> PairOfInt;`

The typename `PairOfInt` can then be used to declare objects of type `Pair<int>` as in this example:

```
PairOfInt pair1, pair2;
```

29

Programming Example: An Array Class

- The interface for a class template named List whose objects are lists of a user specified type which may be built-in or user defined.
- A demonstration program that uses the List template class. The purpose is to illustrate syntax.
- Contains the implementation of the List class template. The data is stored internally as a dynamic array.
- Members provided are the default constructor, a destructor, length, add, full, and erase.
- Notice the overloaded insertion operator, which we use to output an object of template class List.
- When we specify a parameter of List class type in a template function, whether a member or standalone, we use List<T>.
- When we specify a template List parameter type that holds

```
// Interface for the Class Template List (1 of 2)
// FILE: list.h.
// This is the INTERFACE for the class List. List objects can hold items of any
// type where operators << and = are defined. All the items on any one list must
// be of the same type. Declare a list of up to max items of type Type_Name as:
// List<Type_Name> the_object(max);

#ifndef LIST_H
#define LIST_H
#include <iostream>
using namespace std;
namespace savitchlist
{
    template<class T>
    class List
    {
    public:
        List(int max);
        //Initializes the object to an empty list that can hold up to max items of type T.

        ~List();
        //Returns all the dynamic memory used by the object to the heap.

        int length() const;
        //Returns the number of items on the list.
    };
};
```

```
Interface for the Class Template List (2 of 2) // FILE: list.h.
```

```
void add(T new_item);
//Precondition: The list is not full.
//Postcondition: The new_item has been added to the list.

int full() const;
//Returns true if the list is full.

void erase();
//Removes all items from the list so that the list is empty.

friend ostream& operator <<(ostream& outs, const List<T>& the_list);
//Overloads the << operator so it can be used to output the contents
// of the list. The items are output one per line.
//Precondition: If outs is a file output stream, then outs has already
//been connected to a file.

private:
    T *item; //pointer to the dynamic array that holds the list.
    int max_length; //max number of items allowed on the list.
    int current_length; //number of items currently on the list.
};
} //list
#endif //LIST_H
```

```
Program using the List class Template
```

```
//Program to demonstrate use of the class template List.
#include <iostream>
#include "list.h"
#include "list.cxx"
using namespace std;
using namespace list;

int main()
{
    List<int> first_list(2);
    first_list.add(1);
    first_list.add(2);
    cout << "first_list = \n" << first_list;

    List<char> second_list(10);
    second_list.add('A');
    second_list.add('B');
    second_list.add('C');
    cout << "second_list = \n" << second_list;

    return 0;
}
```

```
Implementation of List (1 of 3)
```

```
//FILE: list.cxx This is the IMPLEMENTATION of the class template List.
//The interface for this class is list.h.
#ifndef LIST_CXX
#define LIST_CXX
#include <iostream>
#include <cstdlib>
#include "list.h" //This is not needed when used as we are using this file,
//but the #ifndef in list.h makes it safe.
using namespace std;
namespace list
{
    //Uses cstdlib:
    template<class T>
    List<T>::List(int max)
    {
        max_length = max;
        current_length = 0;
        item = new T[max];
        if (item == NULL)
        {
            cout << "Error: Insufficient memory.\n";
            exit(1);
        }
    }
};
```

```
Implementation of List (2 of 3)
```

```
template<class T>
List<T>::~List()
{
    delete [] item;
}

template<class T>
int List<T>::length() const
{
    return (current_length);
}

//Uses iostream and cstdlib:
template<class T>
void List<T>::add(T new_item)
{
    if (full())
    {
        cout << "Error: adding to a full list.\n";
        exit(1);
    }
    else
    {
        item[current_length] = new_item;
        current_length = current_length + 1;
    }
}
```

Implementation of List (3 of 3)

```
template<class T>
int List<T>::full() const
{   return (current_length == max_length); }

template<class T>
void List<T>::erase()
{   current_length = 0; }

//Uses ostream:
template<class T>
ostream& operator <<(ostream& outs, const List<T>& the_list)
{   for (int i = 0; i < the_list.current_length; i++)
        outs << the_list.item[i] << endl;
    return outs;
}
} //list
#endif // we have enclosed all the template definitions in #ifndef... #endif.
```