

Chapter 2: PARAMETERIZED TYPES - TEMPLATES

1) Templates.....	2-2
2) Overloading Functions	2-3
3) Template Functions	2-4
4) Specializing A Template Function.....	2-5
5) Disambiguation Under Specialization.....	2-6
6) Template Classes.....	2-7
7) Instantiating A Template Class Object.....	2-12
8) Rules for Template Classes	2-13
9) A Non-Member Function With A Template Argument.....	2-14
10) Friends Of Template Classes	2-15
11) Templates With Multiple Type Parameters.....	2-16
12) Comments Regarding Templates	2-18
13) Exercises.....	2-19

Templates

- Templates provide a way of implementing type independent code.

- C++ provides two template mechanisms:

- ▶ **Template functions** – good candidates might be:

sort average min max

- ▶ **Template classes** – good candidates might be:

Array Stack List Set

- A template function is similar to a macro except that template functions provide:

- ▶ type checking
- ▶ better diagnostics
- ▶ no side effects

```
(1) #define MAX(X,Y) (X) > (Y) ? (X) : (Y)
    int a = 5, b = 10, c;
    c = MAX(a++,b++);
```

```
(2) #define SQUARE(X) (X * X)
    int a = 2, b = 3; c;
    c = SQUARE(a + b);
```

Overloading Functions

- Note the following overloaded non-template functions:

```
double min(const double & a, const double & b)
{
    if ( a < b)
        return a;
    return b;
}
int min(const & int a, const int & b)
{
    if ( a < b)
        return a;
    return b;
}
String min(const String & a, const String & b)
{
    if ( a < b )
        return a;
    return b;
}
```

- One could imagine other `min` functions. One could also imagine a family of `max` or `sort` functions.

Template Functions

- The set of `min` functions differ only by their type. If the type could be parameterized, then only one such function needs to be written. This function could be instantiated on many types.
- In C++, parameterized types are achieved by using the `template` keyword as follows:

```
template < class T > T min(T a, T b)
{
    if ( a < b)
        return a;
    return b;
}

#include <iostream.>
using namespace std;
main( )
{
    int a = 5, b = 10;
    double x = 30.45, y = 57.35;
    String r("mike"), s("sue");

    cout << min(a,b) << endl;
    cout << min(x,y) << endl;
    cout << min(r,s) << endl;
}
```

- `T` is a variable whose value is a type.

Specializing A Template Function

- Templates are not without problems. The following call to the template `min` function causes incorrect behavior.

```
char *line = "there", *text = "hello";  
cout << min(line, text) << endl;
```

- It would be executed as if the following code had been written as:

```
char * min(char * a, char * b)  
{  
    if ( a < b)  
        return a;  
    return b;  
}
```

- What is needed is a function specifically coded for the `char` types! (i.e. a non-template function!).

```
char * min(char *a, char *b)  
{  
    return strcmp(a,b) < 0 ? a : b;  
}
```

Disambiguation Under Specialization

- Whenever there exist overloaded functions, one or more of which is a template function, the compiler uses the following algorithm to disambiguate:

- ▶ Step 1: Examine all non-template functions for exact match:
 - if there exists exactly 1 function, match is found
 - if there exist > 1 functions, ambiguity error
 - if no matches, go to next step
- ▶ Step 2: Examine all template functions for exact match:
 - if there exists 1 function, match is found
 - if there exist > 1 functions, ambiguity error
 - if no matches, go to next step
- ▶ Step 3: Reexamine all non-template instances, looking for a match through conversion

- Examples:

```
1. char line[100]="hello", word[100]="there";
   cout << min(line,word);
   // Step 1: match on non-template
   // char * min(char *, char *);

2. Fraction a(1,2), b(2,3);
   cout << min(a,b);
   // Step 2: match on template
   // template < class T> T min(T,T);

3. int a = 10;
   double x = 25.5;
   cout << min(a,x);
   // Step 3: match on non-template function
   // either int min(int, int);
   // double min(double, double);
```

Template Classes

- Examine the following two classes:

```
1. class IntStack {
2.     int *data;
3.     int howmany;
4.     int top_of_stack;
5. public:
6.     IntStack(int number = 10);
7.     ~IntStack( );
8.     void push(int value);
9.     int pop( );
10. };
11. IntStack::IntStack(int number)
12. {
13.     data = new int[howmany = number];
14.     top_of_stack = 0;
15. }
16. IntStack::~~IntStack()
17. {
18.     delete [ ] data;
19. }
20. void IntStack::push(int value)
21. {
22.     data[top_of_stack++] = value;
23. }
24. int IntStack::pop( )
25. {
26.     return(data[--top_of_stack]);
27. }
```

Template Classes

```
1. class DoubleStack {
2.     double *data;
3.     int howmany;
4.     int top_of_stack;
5. public:
6.     DoubleStack(int number = 10);
7.     ~DoubleStack( );
8.     void push(double value);
9.     double pop( );
10. };
11. DoubleStack::DoubleStack(int number)
12. {
13.     data = new double[howmany = number];
14.     top_of_stack = 0;
15. }
16. DoubleStack::~~DoubleStack()
17. {
18.     delete [ ] data;
19. }
20. void DoubleStack::push(double value)
21. {
22.     data[top_of_stack++] = value;
23. }
24. double DoubleStack::pop( )
25. {
26.     return(data[--top_of_stack]);
27. }
```

Template Classes

- The two preceding classes represent the same abstraction, a **stack**.
 - ▶ same functionality
 - ▶ same implementation
 - ▶ different data type
- It's easy to imagine other classes such as the ones on the preceding pages.

StringStack ComplexStack

- Classes that have identical interfaces and implementations, except for their types, should be implemented in C++ as a template class - a blueprint for a family of classes.

Template Classes

- Here is the template Stack class.

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4. //
5. // error function used later
6. //
7. void error(string s)
8. {
9.     cout << s << endl;
10.    exit(0);
11. }
12. template <class T>
13. class Stack
14. {
15.     T *data;
16.     int howmany;
17.     int top_of_stack;
18. public:
19.     Stack(int number = 10);
20.     ~Stack();
21.     void push(T value);
22.     T pop();
23. };
```

- Each instance of a type that needs to be parameterized has been replaced with a variable whose type value will be determined when a particular class is instantiated.
- Each member function is itself a template function and must be coded as such. The actual template functions are defined next.

Template Classes

```
1. template <class T>
2. Stack<T>::Stack(int number)
3. {
4.     data = new T[howmany = number];
5.     top_of_stack = 0;
6. }
7. template <class T>
8. Stack<T>::~~Stack()
9. {
10.    delete [ ] data;
11. }
12. template <class T>
13. void Stack<T>::push(T value)
14. {
15.     if ( top_of_stack >= howmany )
16.         error("stack overflow");
17.     data[top_of_stack++] = value;
18. }
19. template <class T>
20. T Stack<T>::pop( )
21. {
22.     if ( top_of_stack <= 0 )
23.         error("Stack underflow");
24.     return(data[--top_of_stack]);
25. }
```

- The notation:

```
template < class T>Array<T>::Array(int number)
```

- ▶ defines this function as a template function
- ▶ specifies it is from a template class

Instantiating A Template Class Object

- Objects of template classes are instantiated as shown here.

```
int main()
{
    Stack <int> ints(5);

    for (int i = 0; i < 5; i++)
        ints.push(i);
    for (i = 0; i < 5;i++)
        cout << ints.pop();
    return 0;
}
```

- For each different template `Stack`, the compiler generates a class with accompanying functions from the template definition.
- Once the template class has been instantiated, using a template class object is the same as using a non-template class object.

```
ints.push(14);
cout << ints.pop() << endl;
```

Rules for Template Classes

- Template classes can be defined inside other classes.
- A template function cannot be a virtual function.
- When a member function takes a template object or a template reference as an argument, the C++ standard says that either of the following notations is correct. However, some compilers insist on one or the other.

```
Stack(const Stack & a);  
int operator==(const Stack & a);  
Stack <T> & operator=(const Stack & a);
```

or

```
Stack(const Stack <T> & a);  
int operator==(const Stack<T> & a);  
Stack <T> & operator=(const Stack <T> & a);
```

A Non-Member Function With A Template Argument

- Suppose you wish to write a non-member function to sum the elements of a template `Array`. This can be done in two ways.

- ▶ **Generalize the function** – make the function a template function.

```
template <class T> T sum1(const Stack<T> x)
{
    T total = 0;
    ...
    return total;
}
```

- ▶ **Make it type specific** – do not make it a template function.

```
int sum2(const Stack<int> x)
{
    int total = 0;
    ...
    return total;
}
```

Friends Of Template Classes

- For example, the output operator:

```
Stack <int> ints(10);  
cout << ints << endl;
```

- Since friends are non-member functions, either of the preceding approaches could be used.

```
1. //  
2. // type specific  
3. //  
4. friend ostream &  
5. operator<<(ostream & os, const Stack<int> & a)  
6. {  
7.     int i;  
8.     for (i = a.top_of_stack - 1 ; i >= 0 i++)  
9.     {  
10.         os << a.data[i];  
11.     }  
12.     return os;  
13. }  
14.  
15. //  
16. // template  
17. //  
18. friend ostream &  
19. operator<<(ostream & os, const Stack<T> & a)  
20. {  
21.     int i  
22.     for (i = a.top_of_stack - 1; i >= 0 ; i--)  
23.         os << a.data[i];  
24.     return os;  
25. }
```

Templates With Multiple Type Parameters

- A template class represents a family of classes whose exact type is parameterized.
- The parameterization is not limited to one type. That is, there can be more than one type parameter

```
template <class L, class R>
class Pair {
    L left_d;
    R right_d;
public:
    Pair(const L & Left, const R & Right)
        : left_d(Left), right_d(Right) { }
    Pair() {}
    L getleft() const { return left_d; }
    void setleft(const L & c) { left_d = c; }
    R getright() const { return right_d; }
    void setright(const R & c) { right_d = c; }
};
```

- The above template class represents a family of classes whose data is a set of two (possibly distinct) types.
- The following are instantiations of Pair

```
char name[100];
Pair <char *, double> Account(name, 10000.00);

Fraction Pe(100,33);
String Symbol("TEI");
Pair <Fraction,String> Company(Pe,Symbol);
```

Templates With Multiple Type Parameters

- If you define the `Pair` constructor outside of the class definition, it would look like this:

```
template <class L, class R>
Pair<L,R>::Pair(const L & left,const R & right)
: left_d(left), right_d(right)
{ }
```

- Here is a partial program using the `Pair` template.

```
int main( )
{
    string s1("ibm");
    Fraction sunpe(100,33), ibmpe(200,44);
    String sun("sunmc"), ibm("ibm");
    Pair <String, Fraction> Nasdaq[2];

    Nasdaq[0].setleft(sun);
    Nasdaq[0].setright(sunpe);
    Nasdaq[1].setleft(ibm);
    Nasdaq[1].setright(ibmpe);
    for ( int i = 0; i < 2; i++)
    {
        cout << Nasdaq[i].getleft() << endl;
        cout << Nasdaq[i].getright( ) << endl;
    }
    if(Lookup(Nasdaq, 2, s1) == -1)
        cout << s1 << " not found\n";
    else
        cout << s1 << " found\n";
    return 0;
}
```

Comments Regarding Templates

- Template classes usually have assumptions such as:

- ▶ Instantiated functions have the necessary operations

```
template <class T> T min(T a, T b)
{
    return a < b ? a : b;
}

template<class T>Array<T>::Array(int size)
{
    data = new T[size];          // def cons
    ...
    ...
}
```

- There is no way to specify that all template instantiations are friends.

```
friend class Array<int>          // ok
friend template <class T> class Array; // error
```

- Template classes can be type arguments to a template class.

```
Matrix < Array<int> > M;          // spaces important
```

- ▶ M is an object from a template class. The elements of this Matrix are integer arrays.

Exercises

1. Turn the `Array` class into a `template` class:

- ▶ Take the non-template `Array` class in the `starters` directory and turn it into a `template` class. Instantiate it for `int`, `double`, and `Stack <int>`.

2. Define a `template` `Pair` class that takes any two types. In this exercise, your `Pair` `template` class will take a `string` and an `int` representing a golfer and his/her handicap respectively.

- ▶ Define a function named `enter` whose interface is shown here.

```
Pair <string, int> handicaps;  
handicaps.enter("Mike", 10);  
handicaps.enter("Susan", 25);
```

- ▶ Define the `operator []` function so that it takes the first of the `Pair` types and returns the second.

```
cout << handicaps["Susan"] << endl;  
cout << handicaps["Mike"] << endl;
```

This example illustrates the implementation of an associative array. How you actually implement the details is up to you.

This Page Intentionally Left Blank

Evolution