Chapter 13: Templates

Templates are patterns. According to Merriam-Websters Unabridged dictionary:

> **template:** Something that establishes a pattern.
>
> **archetype:** The original model, form, or pattern from which something is made or from which something develops.

From a review of *Effective STL : 50 Specific Ways to Improve Your Use of the Standard Template Library* by ScottMeyers:

> It's hard to overestimate the importance of the Standard Template Library: the STL can simplify life for any C++ programmer who knows how to use it well. Ay, there's the rub.

Templates are the archetypes behind the "container" data structures whose processing methods depend only on the structuring method and not on the type of the data contained in the structure. The C++ standard library supports a set of pre-defined templates that implement stacks, queues, trees, and hash tables. A programmer might use the STL templates, extend them, or define entirely new template classes.

A template consists of declarations and functions with one or more type parameters or non-type parameters. Type parameters are used to represent the base type of a data structure. Non-type parameters are relatively new in C++; they are generally integers and are used to define such things as array lengths. Both kinds of parameters can be given default values.

By itself, a template is not compilable code. Before it can be compiled, actual types must be supplied to replace the type parameters. The replacement process is called *instantiation* and happens at compile time. The result of instantiation is a class declaration and definitions of the class functions.

13.1 Basic Template Syntax

A complete class template consists of a parameterized class declaration together with all its class functions and all related functions such as an extension of operator¡¡. The entire template is written within a .hpp file because it is a declaration, not compilable code and all these parts must be available at compile time for instantiation by a client class.

- A template declaration starts with a "template" line: the keyword `template` followed by angle brackets enclosing another keyword, `class`, and a name for the type parameter (or parameters). It is customary to name template parameters with single, upper-case letters. Most templates have only one parameter. If there are two or more, the parameter names are separated by commas. Examples:

```
template <class X>
template <class K, class T, int n>
```

- If a class template has remote functions or an extension for the output operator, each function (lines 32, 36, 44, 51) must start with a "template" declaration like the one that begins the class declaration (line 9). C++ supplies no reasonable syntax that lets us declare `template <class T>` once and include all the parts that are needed within its scope.

- Following each template declaration is a class or function declaration with normal syntax. Within this declaration, the name(s) of the type parameter(s) are used to refer to the future base type of the class.

- Being part of a template declaration does not require any extra words or punctuation within the class. Specifically, we refer to the class name within this part of the code without using angle brackets.

- A prototype or the definition of an inline function in a template is the same as for a non-template. (Note the definition of `FlexArray::flexlen` and the prototype for `FlexArray::operator[]`.)
- The definition of each remotely defined function starts with the name of the template WITH angle brackets and the name(s) of the parameters. For example, the name of the put function is: `FlexArray<T>::put`
- Even though operator `<<` is not part of the class, its definition must be given as part of the template (lines 32–33) because it depends on the type parameter, T.

A template for a recursively-defined type. Some data structures require definition of a pair of classes such that each class definition refers to the other class. An example is a linked list defined as a friendly pair of classes: List and Cell. The List class must have a data member of type Cell*. The Cell class must have a friendship declaration for List. This circular dependency poses no problem for a non-template declaration: both classes can be declared in one .hpp file, with the dependent class first, followed by the interface class.

However, when we transform this pair of classes into a template pair, a difficulty arises when the compiler tries to process the friend declaration. Suppose we were defining the `Cell<T>` template. In that definition, we must say: `friend class List<T>`. However, because the `Cell<T>` declaration comes before `List<T>`, the compiler gives a fatal error comment for the friend declaration: *Stack is not a template*. This can be cured by adding a *forward declaration*, like this, at the top of the file, above the `Cell<T>` declaration:

```
template <class T> class Stack;        // A forward declaration for Stack<T>.
```

Note: every reference to Cell in the List class must say `Cell<T>`, and every reference to List in the Cell class must say `List<T>`.

13.2 A Template for FlexArrays

```
 1  #ifndef FLEX
 2  #define FLEX
 3  // ===========================================================================
 4  // Template declaration for a flexible array of base type T.
 5  // A. Fischer, May 14, 2001                                    file: flexT.hpp
 6  #include "tools.hpp"
 7  #define FLEX_START 4    // Default length for initial array.
 8
 9  template <class T>
10  class FlexArray {
11    protected:// ------------------------------------------------------------
12      int Max;          // Current allocation size.
13      int N;            // Number of array slots that contain data.
14      T* Data;          // Pointer to dynamic array of T.
15
16    private: // -------------------------------------------------------------
17      void grow();      // Double the allocation length.
18
19    public: // --------------------------------------------------------------
20      FlexArray( int ss = FLEX_START ) : Max(ss), N(0), Data( new T[Max] ) {}
21      ~FlexArray() { if (Data != NULL) delete[] Data; }
22
23      int put( T data );
24      T&  operator[]( int k );
25      int flexlen() const { return N; }
26      T*  extract() { T* tmp=Data; Data=NULL; Max = N = 0; return tmp; }
27      ostream& print( ostream& out ) const {
28          for (int k=0; k<N; ++k) out << Data[k] <<" ";
29          return out;
30      }
31  };
32  template <class T> inline ostream&
33  operator<< ( ostream& out, FlexArray<T>& F){ return F.print(out); }
```

```
34
35  // ------------------------------------------ copy a T into the FlexArray.
36  template <class T> int
37  FlexArray<T>::put( T data ) {
38      if ( N == Max ) grow();    // Create more space if necessary.
39      Data[N] = data;
40      return N++;            // Return subscript at which item was stored.
41  }
42
43  //------------------------------------------ access the kth T in the array.
44  template <class T> T&
45  FlexArray<T>::operator[]( int k ) {
46      if ( k >= N ) fatal( "Flex_array bounds error." );
47      return Data[k];               // Return reference to desired array slot.
48  }
49
50  // ------------------------------------ double the allocation length.
51  template <class T> void
52  FlexArray<T>::grow() {
53      T* temp = Data;                     // hang onto old data array.
54      Max>0 ? Max*=2 : Max = FLEX_START;
55      Data = new T[Max];                  // allocate a bigger one.
56      memcpy(Data, temp, N*sizeof(T));    // copy info into new array.
57      delete temp;                        // recycle (free) old array.
58              // but do not free the things that were contained in it.
59  }
60  #endif
```

Turning a working class into a template. If you have a class that implements a general data structure, changing it into a template is easy.

1. Move all the code from the .cpp file to the end of the .hpp file.
2. Insert a template declaration before the class declaration and before every function that is outside of the class.
3. Using the search and replace feature of your editor, replace the name of the class's base type by a parameter name like T.
4. Insert a pair of angle brackets enclosing the parameter name before every occurrence of :: .

The FlexArray class used in prior chapters is similar to, but much simpler than, the STL Vector class. We use it to demonstrate how to define, extend, and use a template class.

13.3 Adapting a Template

All classes, including template classes, should encapsulate their members and provide public functions for all essential tasks. This leads to a design conflict: should a template definition include every possible function that someone might someday need? Or should the class be clean and focus all functions on its primary purpose?

Derivation solves this problem: the base template should be clean and derived templatess should be used to add functionality for special purposes. Sometimes it is useful to have two or more templates available for variations on one basic data structure. For example, the FlexArray template does not supply any functions for searching the array, yet some potential applications of a FlexArray may require searching. In such situations, the solution is to derive one or more variations from a basic template class.

Deriving from a template. Both template classes and non-template classes can be derived from a template, and both derivation patterns are common. Template derivation is used when the original template provides a useful data structure, but a different or more extensive interface is needed. Two FlexArray variations are presented here: the FlexFind template adds a single capability (a find() function) to the class, and the Stack template provides a wholly different interface. A few details should be noted:

- The data parts of the FlexArrray class are protected, not private, because we intend to use derivation with this class.
- Public derivation was used to build the derived template so that all the functionality of FlexArray will remain available to a client program that derives from FlexFind.
- When a class is derived from a template, its constructor **must** have a ctor that supplies a type parameter for the template class, plus any other parameters the template constructor needs.

Changing a template's interface. The stack is a familiar data structure with restrictions on access. The rule for pushing an item onto a stack fits well with the append-on-the-end rule of the FlexArray. However, a Stack does not support random access or sequential search. Because of the basic similarity, we can use a FlexArray to implement a stack. In this example, new functions provide the familiar stack interface, and private derivation is used to hide the inherited FlexArray class members.

```
61  #ifndef STACKT
62  #define STACKT
63  // ============================================================================
64  // Template class definition for a stack of T objects
65  // Alice E. Fischer  June 10, 2000                          file: stackT.hpp
66  //
67  #include "flexT.hpp"
68
69  template <class T>
70  class Stack : FlexArray<T> {
71      char* Name;
72    public:           //-------------------------------------------------------
73      Stack(char* nm, int sz=4): FlexArray<T>(sz), Name(nm) {
74          cerr << " Create " <<Name <<"  ";
75      }
76      Stack(const Stack<T>& s){ fatal( " Can't shallow-copy stack %s", s.Name ); }
77      ~Stack() { cerr << " Delete " <<Name <<"  ";}
78
79      void  push(T& c)    { put( c ); };
80      T     pop()         { return empty() ? (T)0 : Data[--N]; }
81      T     top()   const { return empty() ? (T)0 : Data[N-1]; }
82      bool  empty() const { return N == 0; }
83      int   depth() const { return N; }
84      ostream& print(ostream& out) const;
85  };
86
87  template <class T> //-----------------------------------------------------
88  ostream&
89  Stack<T>::print(ostream& out) const {
90      out <<"\nStack has " <<depth() <<" items: bottom<[";
91      FlexArray<T>::print( out );
92      out << " ]>top\n";
93      return out;
94  }
95
96  template <class T> inline //----------------------------------------------
97  ostream& operator<< ( ostream& out, Stack<T>& s ){ return s.print( out ); }
98  #endif
```

Adding new functionality. The FlexArray template implements a general strategy for storage management that can be useful in many kinds of situations. The FlexFind template extends the FlexArray template by adding a function, `find`, that performs a sequential search. A `find` function is not part of the basic class because searching is irrelevant in many array applications. The entire definition of the derived class is very brief, having only one constructor and one added function.

The comparison on line 111 illustrates a typical template problem. A template is supposed to work with many or all template parameter types, and the template creator cannot know what types will be used in the

future as template paramters. However, he must write code that works in a broad range of situations, and this must be done without breaking class encapsulation. The solution is to require a potential class parameter to provide one or more public functions. The template code is then written to use those functions, In this case, any type `F` that is used with this template must provide a definition for `operator ==`.

```
 99  // ============================================================================
100  // Template variant of the FlexArray that supports sequential search.
101  // This class relies on an etension of == in class F to compare two F's.
102  // Alice E. Fischer  June 10, 2000                              file: stackT.hpp
103  //----------------------------------------------------------------------------
104  template <class F>
105  class FlexFind : public FlexArray<F> {
106    public:
107      FlexFind(int ss = FLEX_START): FlexArray<F>(ss){}
108      bool find( const F key ) const{
109          int k;
110          for (k=0; k<N; ++k) {
111              if (Data[k] == key) return true;
112          }
113          return false;
114      }
115  };
```

13.4 A Precedence Parser: Instantiation of a Template

Instantiation is the process of using a template, with actual arguments, to create a class declaration and implementation. This is done at compile time, when the compiler reaches an instantiation command. In this example, we instantiate the derived template class, Stack$< T >$ twice to make two different kinds of stacks that are used to implement an infix expression evaluator. This simplified evaluator could easily be extended to handle parentheses and non-binary operators, but our purpose here is to explore the use of C++, not to write a general and powerful evaluator. The Eval class is presented first, followed by a small Operator class, the main program, and some output.

Using the Stack template.

- To instantiate a template, the programmer writes a declaration or a call on new using the class name with an argument list in angle brackets. Two examples are given here, lines 134 and 135:

  ```
  Stack<double> Ands;
  Stack<Operator> Ators;
  ```

- Even though the keyword `class` is used in the template declaration, a non-class type such as `double` may be used as an argument. A `struct` type may also be used.
- The name of the resulting class includes the `< >` and the type argument, and each instantiation with a different parameter creates a new class with a unique name. It also creates a new module of compilable code. Here, we create two complete and separate Stack classes, `Stack<double>` and `Stack<Operator>`. Each Stack function is compiled twice, once for `Stack<double>`, then again for `Stack<Operator>`.
- If a class template is instantiated twice with the same arguments, a compilable module is produced the first time and reused the second time. Suppose we added a third instantiation: `Stack<double> Results`. This third call would refer to the class created by the first instantiation, `Stack<double> Ands`, and both Results and Ands would be objects of the same class.

```
116  #ifndef EVAL_H
117  #define EVAL_H
118  // ========================================================================
119  // A. Fischer, June 9, 2002                                  file: eval.hpp
120  //
121  // Parse and evaluate a prefix expression.  All operators are binary;
122  // they are: +, -, *, /, % (a mod b) and ^ (a to the power b)
123  // Whitespace must be used to delimit both operators and operands.
124  // Operands must start with a digit and may or may not have a decimal point.
```

```
125  // Operands must not exceed 30 keystrokes.
126
127  #include "tools.hpp"
128  #include "stackT.hpp"
129  #include "operator.hpp"
130
131  class Eval {
132    private: // ------------------------------------------------------------
133      enum Intype { bad, number, op, end };
134      Stack<double> Ands;     // Stack of operands and intermediate results.
135      Stack<Operator> Ators;  // Stack of operators.
136      Intype classify( char ch );
137      void   dispatch();
138      void   force( int rprec );
139      double expError();
140
141    public: // -------------------------------------------------------------
142      Eval(): Ands("Ands"), Ators("Ators") {};
143      ~Eval(){}
144      static void instructions( void );
145      double evaluate( istream& in );
146      ostream& print( ostream& out );
147  };
148  #endif
```

A private type.

- A private enumerated type is declared (line 133) and used to simplify the input and parsing operations. One enumeration symbol is listed for each legal kind of keystroke and one for bad data. The enum symbols are returned by the function classify() and used in evaluate().
- Note that the type of the value returned by `classify()` is declared within the class (line 136) as simply `Intype`, and is used the same way to declare a variable in the definition of the `evaluate()` function (line 169). However, the return type of the remote function (line 212) must be written as `Eval::Intype` because:
  - The definition of the enumeration is inside the `Eval` class.
  - A remote function is defined outside its class and translated in the global context, not in the context of the class.
  - The encapsulated enum type is not visible to the compiler unless you qualify it with the class name.

A static class function. Static class functions can be called even when no object of the class type exists.

- Line 144 declares a static class function named `instructions`. It is static because we want `main` to be able to call this function before creating the first Eval object. It is a class function (not global) because it gives expert instructions about the usage and requirements of the `Eval` class.
- Note that the word `static` is used in the declaration on line 144 but not in the remote definition on line 156. This function is called from line 300, in main.

```
149  // ============================================================================
150  // A. Fischer, June 9, 2002                                      file: eval.cpp
151  //
152  #include "eval.hpp"
153  #include "operator.hpp"
154
155  // ------------------------------------------------ Instructions for the operator.
156  void Eval::instructions( void ){
157      cout << "This is an infix expression evaluator.\n"
158           << "* Operands start with a digit and may or may not have a decimal point.\n"
159           << "* Operands must not exceed 31 keystrokes.\n"
160           << "* All operators are binary operators.  Parentheses are not supported.\n"
161           << "* Operators are: +, -, *, /, % (a mod b) and ^ (a to the power b).\n"
162           << "* Whitespace must be used to delimit both operators and operands.\n"
```

```
163           << "* End each expression with a semicolon.\n\n"
164           << "To quit, type semicolon instead of an expression.\n";
165   }
166   //------------------------------------------------ Read input and evaluate expression.
167   double
168   Eval::evaluate( istream& in ) {
169       Intype next;    // Classification of next input character.
170       Operator inOp;  // Operator object constructed from inSymbol.
171       double inNum;   // Read input operands into this.
172       char ch;
173
174       for(;;) {
175           in >> ws >>ch;
176           if (in.eof()) next = end;
177           else next = classify( ch );
178           switch( next ){
179             case number:
180                   in.putback(ch);
181                   in >> inNum;
182                   if ( Ands.depth() != Ators.depth() ) return expError();
183                   Ands.push( inNum );
184                   break;
185
186             case op:
187                   inOp = Operator(ch);
188                   if ( Ands.depth() != Ators.depth()+1 ) return expError();
189                   force( inOp.precedence() );
190                   Ators.push( inOp );
191                   break;
192
193             case end:
194                   if ( Ands.depth() != Ators.depth()+1 ) return expError();
195                   force( 0 );
196                   return  Ands.pop();
197                   break;
198
199             case bad:
200             default: return expError();
201           }
202       }
203   }
204
205   // ------------------------------- Evaluate all higher precedence operators on stack.
206   void
207   Eval::force( int rprec ) {
208       while( Ators.depth()>0 && Ators.top().precedence() >= rprec ) dispatch();
209   }
210
211   //---------- Decide whether next input char is an operator, a semicolon, the beginning
212   Eval::Intype                                              // of an operand, or garbage.
213   Eval::classify( char ch ){
214       if (isdigit( ch )) return number;
215       switch(ch){
216         case '+':
217         case '-':
218         case '*':
219         case '/':
220         case '%':
221         case '^': return op;
222         case ';': return end;
223         default : return bad;
224       }
225   }
```

```
226
227 // ------------------------------------------------------------ Evaluate one operator.
228 void
229 Eval::dispatch() {
230     double result;
231     double right = Ands.pop();
232     double left = Ands.pop();
233     Operator op = Ators.pop();
234     switch (op.symbol()) {
235         case '+': result = left + right;        break;
236         case '-': result = left - right;        break;
237         case '*': result = left * right;        break;
238         case '/': result = left / right;        break;
239         case '%': result = fmod(left, right);   break;
240         case '^': result = pow (left, right);   break;
241     }
242     Ands.push( result );
243 }
244
245 // ------------------------------------------------------------------ Error comments.
246 double
247 Eval::expError(){
248     cerr << "\tIllegal expression.\n";
249     print(cerr);
250     return HUGE_VAL;
251 }
252
253 // ------------------------------------------------------------------ Print the stacks.
254 ostream&
255 Eval::print( ostream& out ){
256     out << "\tRemaining contents of operator stack: ";
257     out << "\tRemaining contents of operand stack: " <<Ands;
258     return out;
259 }
```

Evaluation using precedence. Precedence and associativity are used to define the meaning of operators in most modern languages. The evaluate function implements both. Basically, operators are kept on one stack and operands on another.

- When each operator is read, a decision must be made whether to evaluate the *previous* operator or stack the new one.
- If the precedence of the incoming operator is greater than the precedence of the stack-top operator, it is not yet time to evaluate either one, and the incoming operator is added to the stack.
- If the precedence of the two operators is equal, the rule for associativity comes into play. Arithmetic operators need left-to-right associativity, so we evaluate the leftmost, which is the one on the top of the stack. So we pop the stacked operator and evaluate it. Then we compare the incoming operator to the next one on the stack.
- Lower incoming precedence also means that the stacked operator should be popped and evaluated immediately (its operands are at the top of the Ands stack). Once that is done, the incoming operator must be compared to the new top-of-stack, and so on.
- Lines 193–197 handle the end of the expression. In this situation, all operands have been read and stacked, and all operators that are still on the stack must be evaluated, so we force everything with precedence greater than 0 to be dispatched. If there are no errors in the expression, the value that remains on the operand stack is the answer.
- The `force` function is the heart of a precedence parser. It is responsible for comparing the precedence of the incoming operator with the precedence of the operator on the top of the stack and evaluating as many operators as are appropriate, according to precedence. To evaluate an operator, it calls the `dispatch` function.

- The `dispatch` function (lines 228–243) pops two operands from the `Ands` stack and one operator from the `Ators` stack, interprets the operator, calls the appropriate C operator or library function, and puts the result back on the `Ands` stack.

Other notable things.

- `HUGE_VAL` is defined in `math.h`. It is the largest representable floating-point value, and is used here to signify an error.
- Error checking is done in the `classify` function (line 223) and throughout the `evaluate` function (lines 182, 188, 194, and 200) so that unsupported operators and ill-formed expressions are caught as soon as possible.

13.4.1 The Operator Class

```
260  #ifndef OPERATOR_H
261  #define OPERATOR_H
262  // ============================================================================
263  // A. Fischer, June 9, 2002                                  file: operator.hpp
264  //
265  #include "tools.hpp"
266  class Operator {
267    private: // ------------------------------------------------------------
268      char symb;
269      int  prec;
270
271    public: // ------------------------------------------------------------
272      Operator( char op = '!') : symb(op) {
273          switch (op){
274            case '+': case '-':           prec = 1; break;
275            case '*': case '/': case '%': prec = 2; break;
276            case '^':                     prec = 3; break;
277            default:                      prec = -1; break;
278          }
279      }
280      ~Operator() {}
281      int precedence() const { return prec; }
282      char symbol()    const { return symb; }
283      ostream& print(ostream& out)
284          { return out <<"Symbol: " <<symb <<"  Precedence: " <<prec <<endl; }
285  };
286  inline ostream& operator<<( ostream & out, Operator& op) {return op.print(out); }
287  #endif
```

- We represent an operator as a two-member object: the symbol used to denote the operator, and the precedence of the operator with respect to other operators that are supported. The switch statement in the Operator constructor defines the precedence.
- The precedence is used on line 189, where we move down the stack, popping and evaluating all operators that are higher precedence than the new input.
- The symbol is used on line 234 to select the C++ operator to use to evaluate the current subexpression.
- We use simple "get" functions (lines 281–282) to make these private members of Operator available to the application in a read-only form.

13.4.2 The Main Program

I have written this algorithm in C and Pascal, also. The C++ version is considerably cleaner, simpler, easier to write, and easier to understand. The improvement is amazing to me, and is made possible by having classes with their own constructors, print functions, error handling, etc.

```
288  // ========================================================================
289  // Template example -- Using a stack template.
290  // Alice E. Fischer June 9, 200file: Evaluate/main.cpp
291  //
292  #include "eval.hpp"
293  // ========================================================================
294  int main( void )
295  {
296      char buf[256] = "Hello";
297      double answer = 0;
298
299      banner();
300      Eval::instructions();
301      for(;;){
302          cout <<"\n\nEnter an expression: ";
303          cin >> ws;
304          cin.get( buf, 256 );
305          if ( buf[0] == ;) break;
306          istringstream inst( buf);
307          Eval E;
308          answer = E.evaluate( inst );
309          cout <<\n << answer <<" = " <<buf <<endl;
310      }
311      bye();
312      return 0;
313  }
```

Overall operation.

- We start by calling the instructions function. The instructions could be written as part of main, but since they are very closely tied to the capabilities of the Eval class, it is better to define an instructions function there and call it from main.
- Main contains the loop that interacts directly with the user. It reads a line of input, quits if it finds the end-of-job sentinel. Otherwise, it creates and uses a new expression evaluator and prints the result.
- We declare the evaluator as a local object (line 307) for each new expression. Using local declarations for the stringstream and the evaluator makes initialization very simple and avoids any possibility of having the remains of one expression affect the evaluation of the next one.

String-streams.

- Line 306 declares an input string-stream; this class uses a string as its source of data instead of an input file. Here, we initialize the string-stream to an array of characters that was read on line 304. The buf array is passed into the stringstream constructor and its contents become the contents of the stream. This stream is used as the argument to `evaulate` (line 308), which expects the data in the stream to be an infix expression.
- When a string-stream is used, all the standard input functions are available. Here, we use ws (line 175) to skip leading whitespace, eof() (line 176) to test for the end of the input, putback (line 180) to put the most recent character *back* into the stream, and `operator >>` (lines 175 and 181) to read the data into the appropriate type of variable and convert ASCII strings to floating-point numbers. We could do all this without a string-stream, but it would be a lot more work and much less clear.
- The `putback` function may be unfamiliar to some. Basically, it "backs up" the stream pointer by one keystroke (line 180) so that the input character may be read again in a different format. Here, we first read the input into a char variable, then, when we discover that it is numeric, put it back and reread it (line 181) using numeric input conversion. This is much simpler and nicer than using `strtol` or `atoi`.

The output.

```
This is an infix expression evaluator.
* Operands start with a digit and may or may not have a decimal point.
* Operands must not exceed 31 keystrokes.
* All operators are binary operators.  Parentheses are not supported.
* Operators are: +, -, *, /, % (a mod b) and ^ (a to the power b).
* Whitespace must be used to delimit both operators and operands.
* End each expression with a semicolon.

To quit, type semicolon instead of an expression.


Enter an expression: 2 + 3
 Create Ands   Create Ators
5 = 2 + 3
 Delete Ators   Delete Ands

Enter an expression: 3 ^ 2 * 94 % 7 + 2
 Create Ands   Create Ators
8 = 3 ^ 2 * 94 % 7 + 2
 Delete Ators   Delete Ands

Enter an expression: 26 ^ 0.5
 Create Ands   Create Ators
5.09902 = 26 ^ 0.5
 Delete Ators   Delete Ands

Enter an expression: ;

Normal termination.
```

13.4.3 UML for Templates

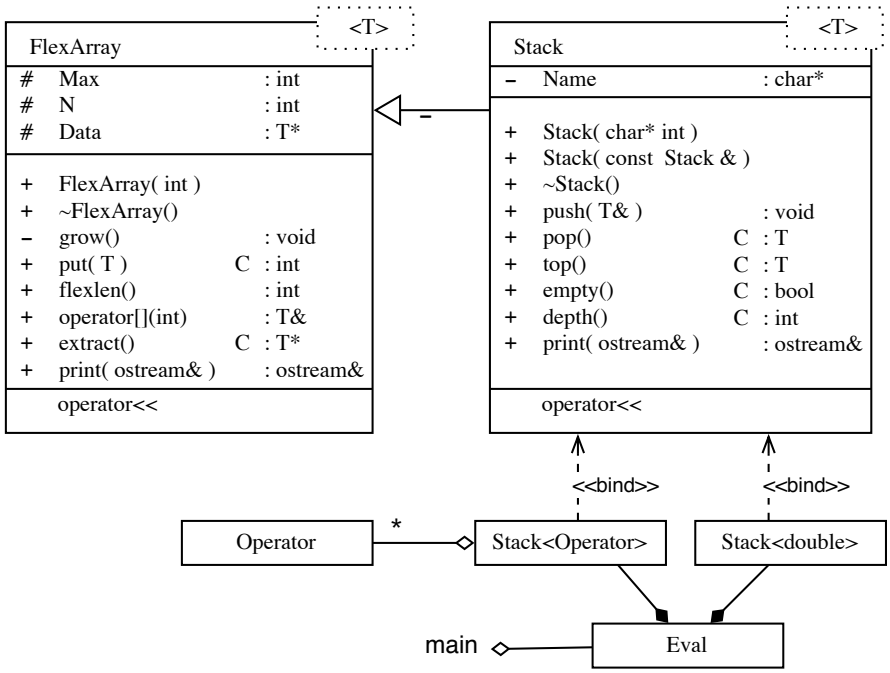

Figure 13.1: UML notation for a template, a derived template, and two instantiations.

We represent a template class in UML by drawing an ordinary class rectangle with a small dotted rectangle covering its upper-right corner. The names of the template parameters are written in the dotted rectangle. Figure 13.4.3 shows two template classes, FlexArray and Stack.

A derived template class is diagrammed like any other derived class. In Figure 13.4.3, the Stack class is joined to the FlexArray class with an ordinary derivation symbol.

In UML, an *instantiation* of a parameterized class is called a *bound element*. A bound element is diagrammed as a class box with a name like `Stack<char>`. with a dotted arrow joining it to the box for the template class. For example, our program that evaluates infix expressions uses two stacks: one for operators, the other for values, so we instantiate the Stack template twice, once as `Stack<Operator>` and once as `Stack<double>`. The instantiations create two new classes that are fully bound and can be compiled. These classes are composed by the Eval class.
It is quite common to derive and instantiate in one step. Suppose class GoodStuff were derived from the instantiation of the stack template with type char:

```
class GoodStuff :  Stack<char> { ... }
```

The diagram would look like this:

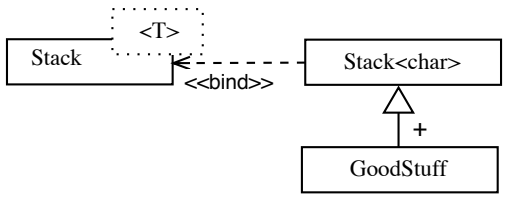

Figure 13.2: UML notation for simultaneous derivation and instantiation.

13.5 The Standard Template Library

STL was designed with extreme care so that it is complete and portable and as safe as possible within the context of standard C++. Among the design goals were:

- To provide standardized and efficient implementations of common data structures as templates, and of algorithms that operate on these structures.
- To produce efficient code. Instantiation of the generic container class is done at compile-time, producing code that is both correct and efficient at run time. This contrasts with derivation and polymorphism which can be used to achieve the same ends but are much less efficient at run time.
- To unify array and linked list concepts, terminology, and interface syntax. Code can be written and partially debugged before making a commitment to one kind of implementation or to another. This permits code to be designed and built in a truly top-down manner,

There are three major kinds of components in the STL:

- Containers manage a set of storage objects (list, tree, hashtable, etc). Twelve basic kinds are defined, and each kind has a corresponding allocator that manages storage for it.
- Iterators are pointer-like objects that provide a way to traverse through a container.
- Algorithms are computational procedures (sort, set_union, make_heap, etc.) that use iterators to act on containers and are needed in a broad range of applications.

In addition to these components, STL has several kinds of objects that support containers. These include pairs (key–value pairs, for the associative containers), allocators (to support dynamic allocation and deallocation) and function-objects (to "wrap" a function in an object). Function objects can be used in algorithms instead of function pointers.

13.6 Containers

The definition of each container class consists of template code, of course, but that code is not part of the standard. Instead, the standard gives a complete definition of the functional properties and time/space requirements that characterize the container. Two groups of containers are supported: sequence containers (lists,

vectors, queues, etc.) and sorted associative containers (maps, sets, etc). The intention is that a programmer will select a class based on the functions it supports and its performance characteristics. Although natural implementations of each container are suggested, the actual implementations are not standardized: any semantics that is operationally equivalent to the model code is permitted. Big-O notation is used to describe performance characteristics. In the following descriptions, an algorithm that is defined as time O(n), is never worse than O(n) but may often be better.

The basic building blocks are precisely organized and, within a group, interchangeable.

Member operations. Some member functions are defined for all containers. These include:

- Constructors: A null constructor, a constructor with one parameter of the container type, and a copy constructor. The latter two constructors operate in linear time.
- Destructor: It will be applied to every element of the container and all memory will be returned. Takes linear time.
- Traversal initialization: begin(), end(), rbegin(), rend(). These mark beginning and ending points for a traversal or reverse-traversal.
- The object's state: size() – current fill level, max_size() – allocation size, empty() – true or false.
- Assignment: `=` Assign one container to another. Linear time.
- Equality: `a == b, a != b` – Returns true or false. Two containers are equal when the sequenes of elements in both are elementwise equal (using the definition of operator== on the element type. Otherwise they are not equal Both take linear time.
- Order: `<, <=, >, >=` Lexicographic comparisons; linear time.
- Misc: a.swap(b) – swaps two conainers of the same type. Constant time.

Sequence Containers

These classes all have the following requirements:

- Constructor with two parameters, int n and base-type element t; Construct a sequence with n copies of t.
- Constructor with two forward-iterator parameters, j and k. Construct a sequence equal to the contents of the range [j, k).
- Traversal initialization: begin(), end(), rbegin(), rend(). These mark beginning and ending points for a traversal or reverse-traversal.
- The object's state: size() – current fill level, max_size() – allocation size, empty() – true or false.
- Assignment: `=` Assign one container to another. Linear time.
- Equality: `a == b, a != b` – Returns true or false. Two containers are equal when the sequenes of elements in a and b are elementwise equal (using the definition of operator== on the element type. Otherwise they are not equal Both take linear time.
- Order: `<, <=, >, >=` Lexicographic comparisons; linear time.
- Misc: a.swap(b) – swaps two conainers of the same type. Constant time.

Sorted Associative Containers

All associatiative containers have two parameters: a type `Key` and an ordering function `comp`, called the "comparison object" of the container, that implements the idea of `<=`. Two keys, `k1, k2` are *equivalent* if `comp(k1, k2` and `comp(k2, k2` both return false. These classes all define the following functions:

- Constructors that include a comparison-object as a parameter.
- Selectors that return the key_type, comparison object-type, and comparison objects.
- Insertion and deletion: four insertion functions with different parameters named `insert()` and `uniq_insert()`. Three `erase()` functions.

- Three iterator functions: `lower_bound(), upper_bound(), and equal_range()`
- Searching: `find(k)`, which returns a pointer to the element whose key is k, (or `end()` if such an element does not exist), and `count(k)` which returns the number of elements whose keys equal `k`.

13.7 Iterators

An iterator provides a general way to access the objects in a container, unifying and replacing both subscripts and pointers. It allows the traversal of all elements in an container in a uniform syntax that does not depend on the implementation of the container or on its content-type.

The underlying value (the value stored in the container) can be either mutable or constant. Exceptional values are also defined:

- Past-the-end values (not dereferenceable)
- Singular values (garbage).

There are five types of iterators, related as shown in figure 13.7. The different types of iterators allow a programmer to select the kind of traversal that is needed and the restrictions (read-only or write-only) to be placed on access. The last two types are called "reverse iterators": they are able to traverse a container backwards, from end to beginning. The examples in Section 13.8 will make all this clearer.

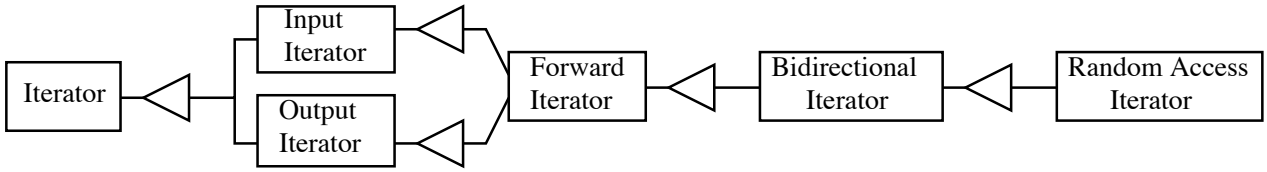

Figure 13.3: Iterator classes form a type hierarchy:

- Input iterators A input iterator class must have a constructor and support the operators `->, ++, *, ==, and !=`, and the `*` operator cannot be used as an l-value in an assignment.
- Output iterators support writing values to a container (using `=` but not reading from it. Restriction: assignment through the same iterator happens only once.
- Forward iterators can traverse the container from beginning to end (using ++)
- Bidirectional iterators can go back and forth in the container; they support the operator `--` in addition to the basic list.
- Random-access iterators must support these operators in addition to the basic list: `[], --, <. >, <=, >=, -=, and +=`. Further, the * operator must support assignment..

13.8 Using Simple STL Classes

The three STL classes introduced here are string, vector, and map. Vector is a sequence container and Map is a sorted associative container. String is derived from Vector, but is presented separately here because of its great usefulness. Please thank Joe Parker for producing the examples in this section.

13.8.1 String

```
1  // STL string example. Joseph Parker, modified by A. Fischer, March 26, 2003
2  #include <string>                          // Header file for STL strings
3  #include <iostream>
4  using namespace std;
5
```

```
 6  int main( void )
 7  {
 8      string str1 = "This is string number one.";   // Allocate and initialize.
 9      // string size
10      cout << "String str1 is: \"" << str1.c_str() <<"\". "
11          <<" Its length is: " << str1.size() << "\n\n";
12
13      cout << "Get a substring of six letters starting at subscript 8: ";
14      string str2 = str1.substr(8,6);
15      cout << str2.c_str() << endl;
16
17      // search first string for last instance of the letter 'e'
18      unsigned idx = str1.find_last_of("e");
19      if (idx != std::string::npos)
20          cout << "The last instance of the char 'e' in string str1 is at pos "
21              << idx <<endl;
22      else cout << "No char 'e' found in string str1" <<endl;
23
24      // search second string for first instance of the letter 'x'.
25      idx = str1.find_first_of("x");
26      if (idx != std::string::npos)
27          cout << "The first instance of the char 's' in string str2 is at pos "
28               << idx <<"\n\n";
29      else cout << "No char 'x' found in string str2\n\n";
30
31      cout << "Now replace \"string\" with \"xxxyyyxxx\".\n";
32      idx = str1.find("string");
33      if (idx != std::string::npos)
34          str1.replace(idx, string("string").length(), "xxxyyyxxx");
35      cout << "str1 with replacement is \"> " << str1.c_str() << "\"\n";
36      return 0;
37  }
```

The output:

```
String str1 is: "This is string number one.".  Its length is: 26

Get a substring of six letters starting at subscript 8: string
The last instance of the char 'e' in string str1 is at pos 24
No char 'x' found in string str2

Now replace "string" with "xxxyyyxxx".
str1 with replacement is "> This is xxxyyyxxx number one."

StringDemo has exited with status 0.
```

Please note these things:

- Line 2: the header function needed for this class.
- Line 10: how to get a C-style string out of a C++ string, and print it.
- Line 11: how to get the length of the string.
- Line 14: creating a new string from a substring of another.
- Lines 18 and 25: searching a string for a letter (first and last occurrenctss).
- Line 34: replace a substring with another string. Note that the old and new substrings do not need to be the same length.

13.8.2 Vector

```
40  // STL_Vector_Example.cpp by Joseph Parker, modified by A. Fischer, March 2003
41  #include <iostream>
42  #include <vector>
```

```
43  #include <algorithm>
44
45  using namespace std;
46
47  //-----------------------------------------------------------------------
48  void print(vector<int>& v)      // print out the elements of the vector
49  {
50      int idx = 0;
51      int count = v.size();
52      for ( ; idx < count; idx++)
53          cout << "Element " << idx << " = " << v.at(idx) << endl;
54  }
55
56  //-----------------------------------------------------------------------
57  int main( void )
58  {
59      vector<int> int_vector;         // create a vector of int's
60
61      // insert some numbers in random order
62      int_vector.push_back(11);
63      int_vector.push_back(82);
64      int_vector.push_back(24);
65      int_vector.push_back(56);
66      int_vector.push_back(6);
67
68      cout << "Before sorting: " <<endl;
69      print(int_vector);                          // print vector elements
70      sort(int_vector.begin(), int_vector.end()); // sort vector elements
71      cout << "\nAfter sorting: " <<endl;
72      print(int_vector);                          // print elements again
73
74      // search the vector for the number 3
75      int val = 3;
76      vector<int>::iterator pos;
77      pos = find(int_vector.begin(), int_vector.end(), val);
78      if (pos == int_vector.end())
79          cout << "\nThe value " << val << " was not found" << endl;
80
81      // print the first element
82      cout << "First element in vector is " << int_vector.front() << endl;
83
84      // remove last element
85      cout << "\nNow remove last element and element=24 "<< endl;
86      int_vector.pop_back();
87
88      // remove an element from the middle
89      val = 24;
90      pos = find(int_vector.begin(), int_vector.end(), val);
91      if (pos != int_vector.end())
92          int_vector.erase(pos);
93
94      // print vector elements
95      print(int_vector);
96      return 0;
97  }
```

The output:

```
Element 0 = 11
Element 1 = 82
Element 2 = 24
Element 3 = 56
Element 4 = 6

Element 0 = 6
```

```
Element 1 = 11
Element 2 = 24
Element 3 = 56
Element 4 = 82

The value 3 was not found
First element in vector is 6
Now remove last element and element=24
Element 0 = 6
Element 1 = 11
Element 2 = 56
```

Please note: A STL vector is a generalization of a FlexArray. You might want to use it because It is not as restricted and presents the same interface as the other STL sequence container classes. The FlexArray, however, is simpler and easier to use for those things it does implement.

- Line 42: the header function needed for this class.
- Line 59: we construct a vector, given the type of element to store within it. Initially this vector is empty.
- Lines 62–66: se put five elements into the vector (at the end).
- Line 69 and 72: we print the vector before and after sorting.
- Line 70: `begin()` and `end()` are functions that are defined on all containers. They return iterators associated with the first and last elements in the container. (It appears that `end()` is actually a pointer to the first array slot past the end of the vector.)
- Line 70: sort is one of the algorithms supported by vector. The arguments to sort are two iterators: one for the beginning and the other for the end of the portion of the vector to be sorted.
- Line 76: we declare an iterator variable of the right kind for vector and use it on the next line to store the position at which a specific element is found.
- Lines 77 and 90: the `find()` function searches part of the vector (specified by two iterators) for a key value (the third argument).
- Line 78 tests for the value `end()`, which is a unique value returned by the find function to signal failure to find the key value.
- Line 82: get the first element in the vector but do not erase it from the vector.
- Line 86: remove the last element from the vector: very efficient.
- Line 92: remove an element from the middle of a vector, using an iterator: not as efficient as `pop()`.

13.8.3 Map

A map is a collection of key/value pairs. When a value is stored into a map, it is stored using one of its fields, called the key field. To retrieve that value, you use the key field to locate it. As STL map can hold only unique keys; if more than one item with the same key can exist, you must use a multimap instead.

This class could be implemented as a hash table or a balanced search tree. Either one would make sense and serve the purpose. We do not actually know which is used, since that is not dictated by the standard. The standard guarantees performance properties, but not implementation. (One might be able to deduce the implementation from the performance properties, though.)

```
100  // STL_Map_Example.cpp by Joseph Parker, modified by A. Fischer, March 2003
101  #include <map>
102  #include <iostream>
103  #include <string>
104  using namespace std;
105
106  int main( void )
107  {
108      // create a map
109      map<int, string> myMap;
110      map<int, string>::iterator it;
```

```
111
112       // insert several elements into the map
113       myMap[1] = "Andrea";
114       myMap.insert(pair<int, string>(2, "Barbara"));
115
116       // print all elements
117       for (it = myMap.begin(); it != myMap.end(); ++it)
118           cout << "Key = " << it->first
119               << ", Value = " << it->second
120               << endl;
121
122       // try some operations
123       it = myMap.find(2);
124       if (it == myMap.end()) cout << "\nKey value 2 not found" << endl;
125       else cout <<"\nValue for key 2 = " << it->second << endl;
126
127       it = myMap.find(3);
128       if (it == myMap.end()) cout << "Value for key 3 not found\n";
129
130       // get # of elements in map
131       cout << "\nThe number of elements in myMap is " << myMap.size() << endl;
132       cout << "Now erase one element from map.\n";
133       myMap.erase(2);
134       cout << "The number of elements in myMap is " << myMap.size() << endl;
135       return 0;
136   }
```

The output:

```
Key = 1, Value = Andrea
Key = 2, Value = Barbara

Value for key 2 = Barbara
Value for key 3 not found

The number of elements in myMap is 2
Now erase one element from map.
The number of elements in myMap is 1

MapDemo has exited with status 0.
```

Please note:

- Line 101: the header function needed for this class.
- Lines 109 and 110 create a map object and an appropriate iterator. Note that both require two parameters: the type of the key field and the type items stored in the container.
- Lines 113 and 114 insert two pairs into the map. Note that the second pair is constructed within the argument list of the insert function.
- Lines 116–120 iterate through the container and print each element. Note that each pair has two members, named first and second, and that these members are used to access both the key value and the data.
- Lines123 and 127 call `map::find()`, supplying the key. Compare this to to the call on `vector::find()` in line 77: the required parameters are different but both return an iterator that points to the found item.
- Lines 124 and 128 test for success of the find operation by comparing the result to `myMap.end()`. This is exactly like the test on line 78 in the vector example.
- Lines 131 and 134 get the number of pairs stored in the map.
- Line 133 removes a specific pair from the map, given its key value.

Conclusion The three STL classes introduced here are among the simplest and most useful. But these examples are "only the tip of the iceberg"; the capabilities of these and other STL classes and algorithms go far beyond what is shown here. The three examples are only a starting point from which the student can continue to learn and master this important aspect of modern programming practice.[1] [2]

[1] The Schildt textbook contains detailed and up-to-date material covering all of the standard templates.

[2] I am also using the first STL book – Musser and Saini, *STL Tutorial and Reference Guide*, and I find it quite readable.