Since a user may have declared several kinds of a type of container class, the final program size can be significantly reduced if all type-independent code is placed in a base class. For example, the COOL **Vector**<*Type*> class implements a parameterized vector class. However, all member functions and data that are independent of the specific *Type* are placed in the base class **Vector**. This results in common functionality shared by several kinds of vector classes, thus reducing the needless code replication that would otherwise occur.

If not designed properly, a parameterized class can result in excessive code-replication when used in a single application many times. When you are designing your own parameterized classes, you can avoid this problem by putting all type-independent code in a base class from which the parameterized class is later derived. The COOL parameterized classes reduce the amount of code that is generated by doing this.

For example, if an application has a **Vector<int>**, **Vector<char*>**, and **Vector<String>**, there could be potentially three "copies" of all the member functions that implement these classes. However, the base **Vector** class implements many of the simple bookkeeping member functions and exception routines that do not require knowledge of or access to the type. The **Vector**<*Type*> class is derived from **Vector**. As a result, although an application may parameterize **Vector**<*Type*> with several different types, there will only be one copy of many of the member functions.

Storing Objects In Container Classes

5.14 The COOL container classes allow the programmer to specify the type of object that will be stored and manipulated by the class. The following member functions must be defined for any user-defined object that is to be contained in any container class (all built-in types already support these operations):

- *Type&* **operator= (const** *Type&***);**
- **Boolean operator== (const** *Type&***);**
- **Boolean operator< (const** *Type&***);**
- **Boolean operator> (const** *Type&***);**
- **friend ostream& operator<< (ostream&, const** *Type&***);**
- **inline friend ostream& operator<< (ostream&, const** *Type**__);__

These member functions are assumed to be available for the type of object over which the class has been parameterized. If any are missing, a compile time error is generated. Although the programmer may not use these directly, the container class uses them for such operations as assigning element values and printing the contents.

Member Functions: **inline** *Type* **##_state operator** *Type* **##_state** ();

Overloaded operator required by the compiler. It implicitly converts the current position state information contained in a type-specific iterator object to the data type expected by an associated container class object.

Iterator Example

5.12 The following program excerpt shows the use of an instance of the **Iterator**<*Type*> class with an instance of the **Vector**<*Type*> class to save and restore the current position.

```
1   #include <COOL/Vector.h>           // Include Vector header file
2   #include <COOL/Iterator.h>         // Include Iterator header file

3   DECLARE Iterator<Vector>;          // Declare Iterator for vector
4   DECLARE Vector<int>;               // Declare Vector of ints

5   Vector<int> v;                     // Declare a vector
6   Iterator<Vector> iv;               // Declare a vector iterator
7   iv = v.current_position();         // Save current position
8   ... /* go do something that may change current position*/
9   v.current_position() = iv;         // Restore previous position
10  ... /* some action continuing from old place, ie. remove */
```

Lines 1 and 2 include the COOL **Vector**<*Type*> and **Iterator**<*Type*> classes and lines 3 and 4 declare a vector of integers and an iterator for vectors. Lines 5 through 10 represent code that might be contained at a point in the source file. Line 5 creates a `Vector<int>` object and line 6 creates an `Iterator<Vector>` object. Line 7 saves the current position of a vector object so that the vector can be altered in line 8. Line 9 restores the previous position value and the program continues with processing in line 10.

Making Your Own Container Classes

5.13 COOL supplies several common container class data structures that can be used by the programmer in many application scenarios. However, there are many other cases where a specialized container class customized for a particular problem is needed (for example, a **BTree** class for a database project). To augment the COOL container classes with other compatible classes, a few requirements must be met:

- The class must contain a private data member maintaining the current position with member functions that update or reset this position as appropriate.
- The member functions **next**(), **prev**(), **reset**(), **value**(), **remove**(), and **find**() must be defined and supported.
- If the name of the container class is **Foo** defined in header file `Foo.h`, there must be a data structure of type **Foo_state** defined in `Foo.h` for use by the **Iterator**<*Type*> class.
- The member function **current_position**() must be defined to return a reference to the **Foo_state** data structure to allow the **Iterator**<*Type*> class to work efficiently.

Iterator Class

5.11 In addition to the built-in iterator previously described, you can also have multiple iterators over the same class by using the **Iterator**<*Type*> class. This is useful when you move through the elements of a container class, come to a point where you need to save the current position, and process elements at another location. After a period of time, you return to the previous stopping point and continue where you left off.

The **Iterator**<*Type*> class provides an independent mechanism for maintaining the state associated with the current position of an instance of a container class. Multiple iterators over the same instance of a class can be supported. Each container class supporting the current position notion has a data structure representing the state. This may be as simple as a type **long**, or more involved, such as with a union of bit fields or another class instance. In addition, each container class has a **current_position** member function to get or set the current position. This member function facilitates storage and retrieval of the current position.

The container-specific data structure used to hold the current position state in all COOL container classes is, by convention, named *class*_**state**, where *class* is the name of the container class header file. Thus, a user including `Vector.h` declares an **Iterator<Vector>** class, and the internal data structure that is created automatically and maintains the state is of type **Vector_state**. In this manner, the **Iterator**<*Type*> class parameterizes over the container class name (that is, **Bit_Set**, **Vector**, and so on). This class allocates a data member of the appropriate type by concatenating the *Type* name with the string "**_state**". The user need not know about internal implementation details.

Each container class has the **current_position** public member function that returns a reference to the iterator state data structure. The member functions supporting current position functionality always work on the current position as maintained in the private data section of the container class instance. A programmer can, at any point, change the current position state information by using this member function to get and/or set the current position of the container class.

Each state data structure implemented in every container class must support the assignment of **INVALID** (defined in `COOL/misc.h`). A state with this value will result in an **Error** exception if used by one of the current position member functions. Alternately, the user can specialize the **Iterator**<*Type*> class to behave differently for a specific class. This alternate mechanism is used by the COOL **List**<*Type*> class in the file `COOL/Iterator.h`.

Name: **Iterator**<*Type*> — A parameterized iterator class

Synopsis: **#include** <COOL/Iterator.h>

Base Classes: None

Friend Classes: None

Constructors:

inline Iterator<*Type*> ();
Simple constructor that initializes to **INVALID** the state information representing the current position for a specific *Type* of container class.

inline Iterator<*Type*> (*Type* **##_state&** *state*);
Constructor that takes a reference to the container-*Type* current position state and copies the value to the internal data member. This constructor calls the **current_position**() member function of some container class.

In many cases, you may need to create a specialized container class that is customized for a particular problem (for example, a BTree class for a database project). Paragraph 5.12, Making Your Own Container Classes, will discuss the requirements for such a case. However, first read the documentation for current position and iterators in the following paragraphs.

Container Example (Current Position)

5.10 Each of the COOL parameterized container classes supports the notion of a built-in iterator maintaining a current position in the container. When a container object is created, the current position is invalidated. Various member functions change the contents or order of elements in a container object, and update the current position marker as necessary (including invalidating it if appropriate). This might occur, for example, if the elements of a container object are sorted according to some new predicate, thus removing any significance to the current position setting.

In addition to this automatic tracking of the current position, the following member functions are common to all container classes and can be used in a generic manner regardless of the specific container class. The programmer uses the following member functions to move through and manipulate the collection of objects in the container:

Member Functions	*Description*
reset	Resets the current position
next	Advances to the next element
prev	Backs up to the previous element
value	Gets the element at the current position
remove	Removes the element at the current position
find	Finds an element and sets the current position

These member functions work efficiently for each container class. In most cases, an inline is all that is needed. Other classes have more efficient versions of a specific member function (such as, **next/prev** in **Vector**, or **find** in **Hash_Table**), but all have the same semantic meaning. These simple member functions combine to make powerful, general purpose functions and macros.

For example, you might define a function that takes a pointer to a generic object that is a type of container class (see the section titled Polymorphic Management later in this manual for more information on polymorphic functionality). The function iterates through the elements in the container by using the current position member functions without needing to know whether the object is a vector, a list, or a queue, and so forth. A complete and useful example of this feature is provided in the section titled Macros later in this manual.

Container Classes

5.9 A container class is a specialization of parameterized classes which contains objects of a particular type. For example, the **Vector**, **List**, and **Hash_Table** classes are container classes because they *contain* a set of programmer-defined data types. On the other hand, the **Range** and **Iterator** classes are parameterized classes, but not container classes, because you do not put objects into them. As container classes are so commonplace in many applications and programs, the COOL parameterized container classes provide a mechanism to maintain one source base for several versions of very useful data structures. The following container classes are currently available in COOL:

Association	An association list of pairs of objects
AVL_Tree	Height-balanced binary tree
Binary_Tree	Fast, efficient binary tree
Hash_Table	Dynamic hash table
List	Dynamic Common Lisp style lists
Matrix	Two-dimensional matrix
N_Tree	N-ary tree
Queue	Dynamic circular queue
Set	Unordered collection of objects
Stack	Dynamic stack
Vector	One dimensional vector

One of the convenient aspects of the container classes is ease from the programmer's point of view. A container class that is parameterized over an object does not require the user to manage memory. However, if the class is parameterized over a pointer to an object, the programmer must allocate and deallocate all storage for the objects.

Generally, there is no performance gain from parameterizing over a pointer to an object rather than the object itself because all COOL container classes use C++ references. In fact, doing so may be less efficient than parameterizing over the object itself. Constructors and destructors for the objects pointed to may be called every time you change, add, or remove an element in the container. If, on the other hand, you parameterize over the object itself, the constructor is called only once when the container class is created. Updates and changes are performed via the assignment and/or X(X&) constructor. A valid reason for choosing a pointer is when the size of each object might be different and/or unknown at compile-time.

Example:

```
1    #include <COOL/Vector.h>    // Bring in the template
2    DECLARE Vector<int>         // Define the type
3    static Vector<int> foo;     // Use the type
4    IMPLEMENT Vector<int>       // Support the type
```

In this example, line 1 includes the parameterized COOL container class **Vector**<*Type*>. Line 2 declares an instance of this class to contain integers. Any valid C++ statement containing a data type can now be used with this type. Line 3 shows a use of this new type to define a static variable. Line 4 must appear only once in all the source files in an application. Line 4 generates the type-specific code that implements the member functions of class `Vector<int>`. At this time, any member function can now be called for an object of this type.

CCC Example

5.8 Suppose you have an application where you require a **Vector**<*Type*> class template parameterized over the built1-in **int** type. You could use **DECLARE** and **IMPLEMENT** and get all of **Vector**<*Type*>'s member functions expanded and linked into your application. Typically, however, you are going to use only a small percentage of the member functions of the class. The remaining unused member functions get linked in as overhead into the executable image, increasing program size and memory requirements. Consider the following program example

```
1    #include <COOL/Vector.h>             // Include parameterized class
2    DECLARE Vector<int>;                 // Declare vector of integers

3    int main (void) {
4     Vector<int> v1;                     // Declare vector object
5     for (i = 0; i < 10; i++)            // Copy 10 elements into vector
6      v1.push (i);                       // Add value to vector
7     cout << v1;                         // Print the vector
8    }
```

Line 1 includes the **Vector**<*Type*> class header file. Line 2 declares the type so that the compiler knows about vectors of integers. Lines 3 through 8 implement a trivial program that adds 10 elements to the vector object and outputs the results. This program makes use of a constructor, the **push** member function, and the overload **operator<<**. If compiled and linked in the normal manner, all the other **Vector**<*Type*> member functions would also be linked into the application, even though they aren't used.

To resolve this problem, the following line can be used in your application make file (as in done for this example in `~COOL/examples/Makefile`):

```
$(CCC) $(CCFLAGS) $(INCLUDE) $(MY_LIB) COOL/Vector.h -oVecInt -X"Vector<int>"
```

This command line executes **CCC** with the usual options and include directory search path. In addition, an application-specific library archive file `MY_LIB` is designated to hold the fractured template object files. The `Vector.h` header file is given as the source file. The `-oVecInt` option causes **CCC** to generate object files named `VecInt0`, `VecInt1`, `VecInt2`, etc. Finally, the `-X"Vector<int>"` option indicates that **CCC** should generate code to support a vector of integers. The resulting object files (one for each member function) from the fractured template are stored in the library archive.

NOTE: As with any intermediate compilation step, the `-c` option must be specified as part of `CCFLAGS`, since it is passed onto the compiler indicating that it should not continue with the link phase.

To insure that the linker searches in the correct library archive for the fractured template object files, add the application-specific library archive to the final link step (as is done for this example in `~COOL/examples/Makefile`):

```
CCC -o $(PROGRAM) $(OBJECTS) -L$(LIB_DIR) -l$(MY_LIB) -lCOOL
```

This command line creates a final executable image named `$(PROGRAM)` from all object files specified by `$(OBJECTS)` using the libraries `$(MY_LIB)` and `libCOOL.a` to resolve any external references.

The user specifies one or more template files, a library archive name, and a specific expansion type as command line arguments. Other arguments for the C++ compiler, system linker, and so forth, are passed on unchanged to the various components of the compilation process. A single invocation of **CCC** processes either a template or proceeds with the compilation of a regular C++ source file, but not both.

Several of the primary COOL classes use **CCC** to fracture an instance of one or more parameterized classes. For example, the **Symbol** and **Package** classes (discussed in section 11, Symbols and Packages) use only a few of the member functions of the **Vector**<*Type*> and **Hash_Table**<Type> classes to implement the runtime type checking (discussed in section 12, Polymorphic Management). See the file `~COOL/Package/Makefile` for more information.

Name:	**CCC** — The COOL C++ control program
Synopsis:	**CCC** [*–options* **REST:** *args*] *template library type*

Options:

–X*"Name<Type>"*
Expands the template for class *Name* with type *Type*. A template expansion must be specified. The double quotation marks are required.

NOTE: The following options are used only in conjunction with the **–X** option; otherwise, they are passed to the system C++ control program.

–o *filename*
Specifies the optional *filename* prefix to be used as the base name for each object module. The default filename is the name of the class with an index appended to it (for example, Vector5.o and Vector6.o). The *filename* must be unique inside the library archive.

–l *library*
Places all resulting object files in the specified application *library* archive. A *library* archive must be specified.

–C
Keeps the fractured source files implementing each member function. This is useful as a debugging aid when a template does not expand correctly due to some user syntax error.

–I *pathname*
Searches the pathname for the specified header (template) source files.

To use this parameterized template, an application programmer includes the parameterized vector header file and adds a **DECLARE** statement in every source file that needs to know about the **Vector**<*Type*> class. In addition, an **IMPLEMENT** statement must be added to only one source file. The following lines could be added to an application program source file to use this parameterized vector class for type `double`:

```
#include <Vector.h>                     // Include parameterized class

DECLARE Vector<double>;                 // Declare vector of double
IMPLEMENT Vector<double>;               // Implement vector of double

void print (Vector<double>& v) {        // Function to print elements
 for (i = 0; i < v.count(); i++)        // For each element in vector
  cout << v[i] << "\n";                 // Print the value
}
```

This simple function takes a single argument of a reference to a parameterized vector of doubles object. It uses the `count()` member function inherited from the base class **Vector** to iterate through the elements of the object and print the value. An alternate procedure for iterating through the elements of a parameterized container class is discussed in paragraph 5.9.

NOTE: When **IMPLEMENT** is used in this manner, all the member functions of the parameterized template are linked into the final executable image, even if they are never referenced or used. To avoid this problem, use the CCC program as discussed below.

COOL C++ Control Program

5.7 Parameterized classes are compiled and manipulated by the COOL C++ Control program (**CCC**) which provides all functions of the original **CC** program and also supports the COOL preprocessor and COOL macro language. **CCC** controls and invokes the various components of the compilation process. In particular, it looks for command line arguments specific to the parameterized template process and processes them accordingly. Other options and arguments are passed onto the system C++ compiler control program.

When **IMPLEMENT** is used to expand a parameterized template, all the member functions are placed in one source file. With the simple linkers available on many operating systems today, a program links these member functions into the application executable image, even if only one or two are actually used. The **CCC** program takes each **template** specifying a member function, compiles it into a separate object module, and adds it to an application-specific object library. As a result, only those member functions actually used by the application get linked into the final program.

CCC takes the in-memory expanded code that implements a parameterized template and fractures it along template boundaries. Each member function for a class is in its own template. Each member function compiles into a separate object module named (by default) the name of the source file with a number appended that is incremented automatically for each member function. These separate object files are then added to an application library. At link time, the system linker uses the symbols in this archive to resolve external references. Since each member function is in its own object file in the library archive, only those member functions used in the application are linked into the final executable image.

```
13    #include <Base_Vector.h>              // Type-independent base class
14    #include <COOL/misc.h>                // COOL definitions
15    template<class Type> class Vector<Type> : public Vector {
16    private:
17      Type* v;                            // Vector of pointer to Type
18    public:
19      Vector<Type> ();                    // Empty constructor
20      Vector<Type> (int);                 // Constructor with size
21      Vector<Type> (Vector<Type>&);       // Constructor with reference
22      ~Vector<Type> ();                   // Destructor
23      inline Type& operator[](int n);     // Operator[] overload for Type
24    Type& element (int n);                // Return element of type Type
25      ...                                 // Other member functions ...
26    };

27    template<class Type>                  // Overload operator []
28    inline Type& Vector<Type>::operator[] (int n) {
29      return this->v[n];
30    }

31    template <class Type>                 // Constructor with size
32    Vector<Type>::Vector<Type> (int n) {
33      this->v = new Type[n];
34      this->size = n;
35      this->num_elements = 0;
36    }

37    ...                                   // Other member functions ...
```

Lines 1 through 8 declare a class **Vector** representing the generic functionality of the parameterized vector class. Data members such as object size and element count are in the base class. Lines 9 through 11 implement one of the inline member functions of this base class. Type-independent member functions like `count()` are provided in the public interface. Other member functions of this base class can be defined. The class declaration and the inline member functions (lines 1 through 11) are written to a file `Base_Vector.h` and the non-inline member functions (line 12) located in the file `Base_Vector.C`.

Line 13 includes the base **Vector** class and line 14 includes the COOL declarations and definitions necessary for the use of parameterized templates. Line 15 is a template for the class **Vector**<*Type*> that inherits the type-independent **Vector** base class. Lines 16 through 26 declare part of the interface for the class. A more complete class would have many other member functions and include support for the current position functionality discussed later. Lines 27 through 30 use a template for an inline member function, and lines 31 through 36 use another template for a constructor for the class. Unlike a non-parameterized class, the class declaration, the inline member functions, and the non-inline member functions are all located in the same file `Vector.h`.

This abbreviated example is exactly how the code is organized for the COOL **Vector**<*Type*> class. Lines 1 through 11 are located in the file `~COOL/Vector/Base_Vector.h` and specify type-independent features. Line 12 (that is, the member functions of the base class) is found in `~COOL/Vector/Base_Vector.C` and contains member function implementation code for the base vector class. Finally, lines 13 through 37 are located in `~COOL/Vector/Vector.h` and specify the parameterized vector class.

DECLARE and IMPLEMENT Example

5.5 Declaration and implementation statements are flexible and can be nested in a variety of operations, such as declaring a list of vectors of integers. In addition, an argument passed as a type name at one level can itself be used as an argument to be passed at a lower level. This is done in the COOL **Association**<*Ktype,Vtype*> class in conjunction with the fourth variation of **template** discussed earlier. An abbreviated header file for this class contains the following statements:

```
1    template <class Ktype, class Vtype> Association {
2      DECLARE Pair<Ktype, Vtype>;          // Declare pair object type
3      DECLARE Vector<Pair<Ktype,Vtype>>;   // Declare vector of pairs
4    }

5    template <class Ktype,class Vtype>
6    class Association : public Vector<Pair<Ktype,Vtype>> {
7    /* Association class interface specification */
8    };

9    template <class Ktype, class Vtype> Association {
10     IMPLEMENT Pair<Ktype,Vtype>;
11     IMPLEMENT Vector<Pair<Ktype,Vtype>>;
12   }
```

Lines 1 through 4 are placed before the **Association**<*Ktype,Vtype*> class definition, thus becoming linked with the declarative part of the template for the class. Lines 5 through 8 contain the actual class definition. Lines 9 through 12 are placed after the class definition, thus becoming linked with the implementation part of the template for the class. By using **template** in this manner, the **DECLARE** for the **Association**<*Ktype, Vtype*> class also invokes **DECLARE** for the correct types for the **Pair**<*Ktype,Vtype*> and **Vector<Pair**<K*type,Vtype*>> classes. Likewise, **IMPLEMENT** for the **Association** class invokes **IMPLEMENT** for the **Pair**<*Ktype,Vtype*> and **Vector<Pair**<*Ktype,Vtype*>> classes.

Template Example

5.6 Suppose a class programmer wants to implement a generic vector class with a simple, consistent interface for the application programmer, regardless of what object is to be stored in the vector. In addition, he wants to avoid replication of code for each specific type. He creates a parameterized vector template derived from a type-independent base class, as in the following abbreviated example:

```
class Vector {                          // Vector class
private:
  int num_elements;                     // Element count
  int size;                             // Size of vector object
public:
  inline int count ();                  // Number of elements
  ...                                   // Other member functions ...
};

inline int Vector::count (int n) {
  return this->num_elements;            // Return element count
}

...                                     // Other member functions ...
```

DECLARE and IMPLEMENT

5.4 As stated earlier, a parameterized template declares a metaclass that is type-independent. To use the metaclass, a programmer must specify the actual type and any other template arguments in order to use it in a program. This is accomplished in two steps: the declarative step and the implementation step. The declarative step uses **DECLARE** and the implementation step uses either **IMPLEMENT** or the Cool C++ Control program (**CCC**) discussed in paragraph 5.7.

Name:

DECLARE — Declares a parameterized class
IMPLEMENT — Implements a parameterized class

Synopsis:

#include <COOL/*Name*.h>
DECLARE *Name<Type>*;
IMPLEMENT *Name<Type>*;

Macros:

DECLARE *Name<Type>*
Declares a parameterized class named *Name* of type *Type*.

IMPLEMENT *Name<Type>*
Implements a parameterized class named *Name* of type *Type*.

DECLARE instantiates a type-independent parameterized template for a user-specified type. **DECLARE** is analogous to using **typedef** to indicate a new valid type name to the compiler, or including the header file for some standard C++ class declaration. **DECLARE** must be used in every file that includes or makes use of a parameterized template. Alternately, the **DECLARE** statement can be placed in a common header file that is included as necessary. **DECLARE** must be followed by a valid parameterized template name and a type name. Typically, this is done by including a header file with common information and definitions.

IMPLEMENT defines the member functions of a parameterized template for a specific type. **IMPLEMENT** is analogous to the C++ file that contains the source code implementing the member functions of a class. **IMPLEMENT** must be used only once in an application for a specific instantiation of a parameterized template; otherwise, you will receive errors from the linker about symbols being defined more than once. **IMPLEMENT** must be followed by a parameterized template name and a type name.Typically, **IMPLEMENT** is done in one of the C++ source files making up part of the application. The name and arguments must match those previously declared with **DECLARE**.

NOTE: When you use **IMPLEMENT**, all the member functions for a particular parameterized template are implemented in one source file. With the simple linkers available on many operating systems today, an application will get all of these member functions linked into the executable image even if only one or two are used. CCC provides a mechanism by which only member functions actually used in the application get linked into the final program. See paragraph 5.6, COOL C++ Control program, for further information.

The declarative part of the template may occur many times in an application and is analogous to including a header file for a class. Template variations here declare the class interface and define the inline member functions.

The implementation part of the template is analogous to the C++ file that contains the source code implementing the member functions of a class. Template variations here define the member and friend functions that constitute the parameterized class.

Name: **template** — C++ parameterized template keyword

Synopsis: **template**<**class** *parms*> **class** *name*<*parms*> { *class_description* };
Defines a template for the declaration of class *name*.

template<*parms*> *result name*<*parms*>::*function* { ... };
Defines a member function for the implementation of class *name*.

template<**class** *parms*> **inline** *result name*<*parms*>::*function* { ... };
Defines an inline member function for the declaration of the class *name* .

template<**class** *parms*> *name* { *anything* };
Defines anything else you want associated with a template.

The first variation of **template** declares a parameterized template in a header file. Typically, such a declaration is very similar to that of a standard C++ class, except for the appearance of the angle brackets and arguments. The second variation defines member functions of a parameterized template. The third variation defines inline member functions of a parameterized template. Again, these appear similar to that of a standard C++ class.

The last variation of **template** defines such miscellaneous items as a **typedef** or an overloaded friend function of a parameterized template. When this form is found before the class template, the contents are expanded before the class declaration. When this form is found after the class template, the contents are expanded as part of the class implementation. This has been used in several COOL container classes for defining predicate types for the class (see paragraph 5.5 example below).

Each of the **template** forms allow one or more optional parameters to be supplied between the angle brackets. These are used to allow the programmer to specify the type and other optional arguments to the template with the following syntax:

parms ::= *type name* [, *parms*]

where *type* is the type of the argument, for example, a **class**, an **int**, and so forth. *Name* is the name of the parameter that is substituted when the template is expanded. For example, an n-ary tree class might have the following template class declaration:

```
template <class Type, int nchild> class N_Tree<Type,nchild> {...};
```

In this example, class `N_Tree<Type,nchild>` is defined as a parameterized template with two arguments. The first, `Type`, specifies the type over which `N_Tree` is parameterized. The second, `nchild`, specifies the number of subtrees each node in the n-tree may have.

Parameterized Templates

5.3 A parameterized template is the mechanism that allows a programmer to define a metaclass representing a type–independent class. The class programmer uses this facility to implement a class without knowing the specific type of data the user might want to use. For example, a **Vector** class can be written by using parameterized templates so that the user of the class can create vectors of integers, vectors of doubles, and so on. This scheme allows the class programmer to maintain one source code base for multiple implementations of the class.

Regardless of the type of object a parameterized template is to manipulate,the structure and organization of the template and the implementation of the member functions are the same for every version of the class. For example, a programmer providing a **Vector** class knows that there will be several member functions such as insert, remove, print, sort, and so on that apply to every version of the class. By parameterizing the arguments and return values from the various member functions, the programmer provides only one implementation of the **Vector** template. The user of the class then specifies the type of vector at compile-time. The following parameterized templates are currently available in COOL:

Templates	*Description*
Association	An association list of pairs of objects
AVL_Tree	Height-balanced binary tree
Binary_Tree	Fast, efficient binary tree
Hash_Table	Dynamic hash table
Iterator	Container class iterators
List	Dynamic Common Lisp style lists
Matrix	Two-dimensional matrix
N_Tree	N-ary tree
Pair	Coupling of two objects
Queue	Dynamic circular queue
Range	User-specified type with limits
Set	Unordered collection of objects
Stack	Dynamic stack
Vector	One-dimensional vector

The syntax of the COOL parameterized templates grammar is as specified by Bjarne Stroustrup in his paper "Parameterized Types for C++" in the 1988 USENIX C++ Conference Proceedings. COOL fully implements the specified syntax so there will be minimal source code conversion necessary when this feature is finally implemented in the C++ language.

The **template** keyword provides a means of defining parameterized templates. COOL provides four variations of **template** for controlling the operation and generation of different parts of a class. Templates are expanded in two parts and each of the four variations is used in one of the two parts:

- The *declarative* part, which is needed by every program file that uses the parameterized class

- The *implementation* part, which needs to be compiled once for the class in any application that uses it

PARAMETERIZED TEMPLATES

5

Introduction

5.1 Parameterized templates allow a programmer to design and implement a general purpose class without specifying the exact type of object or data that is to be manipulated. The user can then customize this general purpose class by specifying the object or data type when it is used in a program. Several versions of the same parameterized template (each implemented with a different type) can exist in a single application. Parameterized templates can be thought of as *metaclasses* in that only one source base needs to be maintained in order to support numerous variations of a *type* of class.

An important and useful type of parameterized template is known as a *container class*. A container class is a special kind of parameterized template where you put objects of a particular type. For example, the **Vector**, **List**, and **Hash_Table** classes are container classes because they contain a set of programmer-defined data types. Since container classes are so commonplace in many applications and programs, parameterized container classes provide a mechanism to maintain one source base for several useful data structures. COOL supplies several common container class data structures that can be used by the programmer in many typical application scenarios.

Each of the COOL parameterized container classes supports the notion of a built-in iterator that maintains a current position in the container and is updated by various member functions. These member functions allow progression through the collection of objects in some order. For example, a function might take a pointer to a generic object that is a type of container object. The function can iterate through elements in the container by using current position member functions without needing to know whether the object is a vector, list, or queue.

In addition to this built-in iterator, you can also have multiple iterators over the same class by using the **Iterator** class. For example, you may be moving through the elements of a container class and come to a point where you need to save the current position and begin processing elements at another location. After a period of time, you return to the previous stopping point and continue where you left off.

Requirements

5.2 This section assumes you have an understanding of the C++ language and its type system. In addition, some familiarity with automated program build procedures such as **make** is also necessary.

Printed on: Wed Apr 18 07:05:31 1990

Last saved on: Tue Apr 17 13:56:04 1990

Document: s5

For: skc