TEXAS INSTRUMENTS

C++ Object-Oriented Library User’s Manual

MANUAL REVISION HISTORY

C++ Object-Oriented Library User's Manual (2566801-0001)

Original Issue . . . . March 1990

Texas Instruments Incorporated
Information Technology Group
Austin, Texas

C++ OBJECT-ORIENTED LIBRARY USER’S MANUAL

CONTENTS

ABOUT THIS MANUAL

Introduction

This manual contains supporting software documentation for COOL (the C++ Object-Oriented Library), a collection of classes, objects, templates, and macros that extend the capabilities of the C++ language. This manual is written for high level C++ application programmers using COOL.

Organization

This manual is divided into the following sections and appendixes:

- **Section 1:** Overview of COOL — Contains an introduction to the *COOL User's Manual* and what to expect on the various classes, macros and other enhancements to C++ that are discussed in this manual.
- **Section 2:** String Classes — Describes a collection of classes that implement textual operations and functions.
- **Section 3:** Number Classes — Contains information on a collection of numerically oriented classes that augment the built-in numerical data types to provide extended precision, range-checked types, and complex numbers.
- **Section 4:** System Interface Classes — Includes classes for calculating the date and time in different time zones and countries.
- **Section 5:** Parameterized Templates — Describes classes that allow a programmer to design and implement a class template without specifying the data type.
- **Section 6:** Ordered Sequence Classes — Describes classes that are a collection of basic data structures that implement sequential-access data structures as parameterized classes.
- **Section 7:** Unordered Sequence Classes — Describes classes that are a collection of basic data structures that implement random-access data structures as parameterized classes.
- **Section 8:** Set Classes — Describes classes that implement two basic data structures for random-access set operations.
- **Section 9:** Node and Tree Classes — Describes classes that are a collection of basic data structures that implement several standard tree data structures as parameterized classes.
- **Section 10:** Macros — Describes COOL macro facilities that are an extension to the standard ANSI C macro preprocessor functions and that support constant symbols, keyword and body arguments, parameterized templates and complex expression evaluation.
- **Section 11:** Symbols and Packages — Describes functions that manage error message textual descriptions, provide polymorphic extensions to C++ for object type and contents queries, and support sophisticated symbolic computing.

- **Section 12:** Polymorphic Management — Describes the mode of managing symbolic constants and run-time symbolic objects and packages. Also discusses the **Generic** class, which is inherited by most COOL classes and manipulates lists of symbols to manage type information.

- **Section 13:** Exception Handling — Describes COOL exception handling, which is a raise, handle, and proceed mechanism that uses the COOL symbolic computing capability.

- **Section 14:** COOL Methodology — Describes COOL methodology for managers and programmers who need a brief overview of COOL components, organization, style guide, and rules for extending the library.

Conventions

Certain conventions and syntax are used to simplify presentation of the material in this manual. The following typographical conventions are used:

- **Bold** type — Words in bold represent either a class name, macro name, or system-supplied function.

- *Italics* — Words in italics enclosed by angle brackets or parenthesis represent arguments that must be specified by the programmer .

- `Monowidth Font` — Program examples and output are distinguished by monowidth font of a smaller size than the normal manual text. (e.g., `#include <COOL/String.h>`)

Program examples illustrated and defined within this manual are supplied and can be found in the `~COOL/examples` subdirectory. Examples are given specific file names by appending a suffix of `.C` to its associated paragraph number. For example, the string class example in paragraph 2.4 is located in the `2.4.C` filename.

Each section includes a requirements paragraph that states the prerequisites needed to understand and use the components or functions being discussed.

Certain paragraphs document reference information about COOL classes. A special format is used that provides the following information:

- Class Name
- Synopsis
- Base Classes
- Friend Classes
- Constructors
- Member Functions
- Friend Functions
- Example Program

OVERVIEW OF COOL

1

Introduction

1.1 The C++ Object-Oriented Library (COOL) is a collection of classes, objects, templates, and macros to extend the capabilities of the C++ language for developing complex problem-solving applications. Significant language features in COOL, such as parameterized types, symbolic computing, and exception handling, are implemented with sophisticated C++ macro facilities. These features and facilities are designed to enhance and improve a programmer's development capability.

COOL is intended to simplify the programming task by allowing the programmer to concentrate on the application problem to be solved, not on implementing base data structures, macros, and classes. In addition, COOL provides a system-independent software platform on which applications are built. An application built on top of COOL will compile and run on any platform supporting COOL.

Audience

1.2 This manual is intended for use by programmers who have a working understanding of the C++ programming language as implemented by AT&T in release 2.0 and type system. Users must also understand the distinction between the concepts and principles associated with overloaded operators and friend functions.

Features

1.3 The major features that COOL contributes to enhancing the C++ language and program development capabilities are the following:

- An enhanced macro language that supports constant symbols, keyword and body arguments, parameterized templates, and complex expression evaluation
- Parameterized templates that allow development of type-independent container classes with support for multiple iterators
- Dynamic, user-defined packages implementing name spaces for symbols with names, property lists, and values
- Polymorphic features derived from the **Generic** virtual base class that supports `is_type_of()` run-time queries
- A multi-level exception handling mechanism that utilizes macros, symbols, and a global error package and is similar in design to the Common Lisp Condition Handling System
- A collection of classes implementing a wide range of useful data structures and system interface facilities

The following paragraphs provide brief descriptions of each of these features, including information on what to expect in the rest of this manual on the various classes, macros, symbolic computing facilities, exception handling routines, and methodology that governs the implementation of COOL.

Macros

1.4 Supplied as part of the library, the COOL macro facilities are an extension to the standard ANSI C macro preprocessor functions and are portable and compiler-independent. The COOL macro facilities support constant symbols, keyword and body arguments, parameterized templates, and complex expression evaluation. Some macros, such as those that support the parameterized types, are implementations of theoretical design papers published by Bjarne Stroustrup.

The COOL preprocessor is derived from and based upon the DECUS ANSI C preprocessor made available by the DEC User's group in the public domain and supplied on the X11R3 source tape from MIT. The preprocessor complies with the draft ANSI C specification with the exception that trigraph sequences are not implemented.

The preprocessor was modified to recognize a **#pragma defmacro** statement to allow a programmer to define powerful extensions to the C++ language. The proposed draft ANSI C standard indicates that extensions and changes to the language and features implemented in a preprocessor and compiler should be made by using the **#pragma** statement. The COOL preprocessor follows this recommendation and uses this as the means by which all macro extensions are made. The **#pragma defmacro** statement is the single hook through which features such as the class macro, parameterized templates, and polymorphic enhancements are implemented. This statement also allows arbitrary filter programs and macro expanders to be run on C++ code fragments passing through the preprocessor. Note, however, that once a macro is expanded, the resulting code is conventional C++ 2.0 syntax acceptable to any conforming C++ translator or compiler.

Parameterized Templates

1.5 Parameterized classes allow a programmer to design and implement a class template without specifying the data type. The user can then customize the class by specifying the type when it is used in a program. Parameterized classes can be thought of as *metaclasses* in that only one source base needs to be maintained to support numerous variations of a *type* of class.

An important and useful type of parameterized class is known as a *container* class. A container class is a special type of parameterized class where you put objects of a particular type. A container class that is parameterized over an object does not require the user to manage memory, activate destructors, and so forth. COOL supplies several common container class data structures that include support for the notion of a built-in iterator that maintains a current position in the container object. Multiple iterators into an instance of a container class are provided by the **Iterator**<*Type*> class.

Parameterized classes are handled 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.

Alternately, a declaration macro can be used to instantiate a type-independent parameterized class for a user-specified type by introducing a new valid type name to the compiler. An implementation macro defines the member functions of a parameterized class for a specific type.

Symbolic Computing

1.6 COOL symbol and package facilities provide the following capabilities:

- management of error message text
- polymorphic extensions to C++ for object type and contents queries
- support of sophisticated symbolic computing normally unavailable in conventional languages

A *package* provides a relatively isolated namespace for various COOL components called *symbols*. Each symbol is unique within its own package and can be used as a dynamic enumeration type. Symbols also can be run-time variables, with the package acting as a symbol table. Those symbols grouped into a particular package are said to be owned (interned) by that package. The package system provides logical groupings of symbols that support relationships established between named objects and the values they contain. COOL provides several kinds of macros to simplify the usage and manipulation of symbols and packages.

COOL supports efficient and flexible symbolic computing by providing symbolic constants and run-time symbol objects. You can create symbolic constants at compile-time and dynamically create and manipulate symbol objects in a package at run-time by using any of several simple macros or by directly manipulating the objects.

The COOL **DEFPACKAGE** macro allows for efficient symbol and package manipulation and is used extensively by COOL to implement run-time type checking and type query. **DEFPACKAGE** allows an application programmer to declare a package that is a program-wide database of constant symbols with associated default values and properties.

A package is created with the **DEFPACKAGE** macro, and macros for adding and retrieving constant symbols in a package are defined with the **DEFPACKAGE_SYMBOL** macro. In COOL, the most common types of packages are made easier to use by the following four macros:

- **enumeration_package**
- **symbol_package**
- **text_package**
- **once_only**

Polymorphic Management

1.7 COOL supports enhanced polymorphic management capabilities with a programmer-selectable collection of macros, classes, symbolic constants, run-time symbolic objects, and dynamic packages. The **Generic** class, combined with macros, symbols, and packages, provides efficient run-time object type checking, object query, and enhanced polymorphic performance unavailable in the C++ language otherwise.

COOL supplies several sophisticated macros that augment and manipulate the symbol objects maintained in the COOL global symbol package. The **type_of** and **is_type_of** virtual member functions provide run-time object type query support. **Describe** and **print** member functions provide symbolic and value-oriented output capabilities. In addition, the **typecase** macro provides an efficient mechanism analogous to the C++ **switch** statement for branching, based upon an object's type. Finally, the **class** macro provides a user-extensible system for querying an object to determine if a particular named function or data member accessor is available or should be created.

Exception Handling

1.8 COOL exception handling is a raise, handle, and proceed mechanism that uses the COOL symbolic computing capability. When a program encounters an anomaly it can:

- Represent the anomaly in an exception object
- Announce that the anomaly has occurred by raising the exception
- Provide ways to deal with the anomaly by defining handlers
- Proceed from the anomaly by invoking a handler

The exception handling facility provides an exception class, an exception handler class, a set of predefined subclasses of the exception class, and a set of predefined exception handler functions. Each exception subclass is provided a default exception handler function that is called if no other exception handler is established. The **Exception** class inherits from the **Generic** class to facilitate run-time type checking and query of exception objects.

Also available are macros that simplify the process of creating exceptions, raising exceptions, and ignoring raised exceptions. These include the **EXCEPTION**, **RAISE**, **STOP**, **VERIFY**, and **IGNORE_ERRORS** macros.

There are six predefined exception type classes provided as part of COOL. The **Exception** class is the base class from which specialized exception subclasses are derived. Derived from **Exception** are **Warning**, **System_Signal**, **System_Error**, **Fatal**, and **Error**. These classes are a means of saving the status information that represents a particular problem or condition, and communicating this information to the appropriate exception handler.

Classes

1.9 Following is a brief description of the various classes developed for COOL to supplement the development of C++ applications.

NOTE: All COOL constants such as **TRUE** and **FALSE** are defined in the `~COOL/misc.h` header file.

String Classes – The **String** class provides dynamic, efficient strings for a C++ application. The intent is to provide efficient char*-like functionality that frees the programmer from worrying about memory allocation and deallocation problems, yet retains the speed and compactness of a standard char* implementation. All typical string operations are provided including concatenation, case-sensitive and case-insensitive lexical comparison, string search, yank, delete, and replacement.

The **Regexp** class provides a convenient mechanism to present regular expressions for complex pattern matching and replacement and utilizes the built-in char* data type. The **Gen_String** class provides general purpose, dynamic strings for a C++ application with support for reference counting, delayed copy, and regular expression pattern-matching. The intent is to provide a sophisticated character string function for the application programmer. The **Gen_String** class combines the functions of the **String** and **Regexp** classes, along with reference counting and self-garbage collection, to provide advanced character string manipulation.

Number Classes – The **Number** classes are a collection of numerically oriented classes that augment the built-in numerical data types to provide such features as extended precision, range-checked types, and complex numbers. Included are the **Random**, **Complex**, **Rational**, **Bignum** and **Range** classes.

The **Random** class implements five variations of random number generator objects. The **Complex** class implements the complex number type for C++ and provides basic arithmetic and trigonometric functions, conversion to and from built-in types, and simple arithmetic exception handling. The **Rational** class implements an extended precision rational data type for inadequate round-off or truncation results from the built-in numerical data types. The **Bignum** class implements near-infinite precision integer arithmetic. Finally, the parameterized **Range**<*Type*> class enables arbitrary user-defined ranges to be implemented in C++ classes. Typically, this is used with other number classes to select a range of valid values for a particular numerical type.

System Interface Classes – System Interface classes include classes for calculating the date and time in different time zones and countries and measuring the time duration between two points in some application program.

The **Date_Time** class executes time zone–independent date and time functions. This class also supports all time zones in the world, along with several special cases requiring alternate handling based upon political or daylight saving time differences. The **Timer** class is publicly derived from the **Generic** class and provides an interface to system timing. It allows a C++ program to record the time between a reference point (mark) and now.

Ordered Sequence Classes – The ordered sequence classes are a collection of basic data structures that implement sequential access data structures as parameterized classes, thus allowing the user to customize a generic template to create a user-defined class. The ordered sequence classes include **Vector**, **Stack**, **Queue** and **Matrix**.

The **Vector**<*Type*> class implements single dimension vectors of a user-specified type. The **Stack**<*Type*> class implements a conventional first-in, last-out data structure, while the **Queue**<*Type*> class implements a conventional first-in, first-out data structure. These two classes each hold a user-specified data type. The **Matrix**<*Type*> class implements two-dimensional arithmetic matrices for a user-specified numeric data type. The **Vector**, **Stack**, and **Queue** classes can be dynamic in size.

Unordered Sequence Classes – The unordered sequence classes are a collection of basic data structures that implement random access data structures as parameterized classes, thus allowing the user to customize a generic template to create a specific user-defined class. The unordered sequence classes include **List**, **Pair**, **Association**, and **Hash_Table**.

The **List**<*Type*> class implements Common Lisp style lists providing a collection of member functions for list manipulation and management. A list consists of a collection of nodes, each of which contains a reference count, a pointer to the next node in the list, and a data element of a user-specified type.

The **Pair**<*T1*,*T2*> class implements an association between one object and another. The objects may be of different types, with the first representing the *key* of the pair and the second representing the *value* of the pair. The **Association**<*Ktype,Vtype*> class is privately derived from the **Vector**<*Type*> class and implements a collection of pairs. As above, the first of the pair is called the *key* and the second of the pair is called the *value*. The **Hash_Table**<*Ktype,VType*> class implements hash tables of user-specified types for the key and the value.

Set Classes – The set classes implement two basic data structures for random-access set operations as parameterized classes, thus allowing the user to customize a generic template to create a specific user-defined class. The set classes include **Set** and **Bit_Set**.

The **Set**<*Type*> class implements random access sets of objects of a user-specified type. Classical set operations such as union, intersection, and difference are available. The **Set**<*Type*> class is publicly derived from the **Hash_Table**<*KType,VType*> class and is dynamic in nature.

The **Bit_Set** class implements efficient bit sets. These bits are stored in an arbitrary-length vector of bytes (unsigned char) large enough to represent the specified number of elements. Elements can be integers, enumerated values, constant symbols from the enumeration package, or any other type of object or expression that results in an integral value.

Node and Tree Classes – The node and tree classes are a collection of basic data structures that implement several standard tree data structures as parameterized classes, thus allowing the user to customize a generic template to create a specific user-defined class. The node and tree classes include **Binary_Node**, **Binary_Tree**, **N–Node**, **D-Node**, **AVL_Tree** and **N–Tree**.

The **Binary_Node**<*Type*> class implements parameterized nodes for binary trees. The **Binary_Tree**<*Type*> class implements simple, dynamic, sorted sequences in a tree where each node has two subtree pointers. The **AVL_Tree**<*Type*> class implements height-balanced, dynamic, binary trees. The **AVL_Tree**<*Type*> class is publicly derived from the **Binary_Tree**<*Type*> class.

The **N_Node**<*Type,nchild*> class implements parameterized nodes of a static size for n-ary trees. The **D_Node**<*Type,nchild*> class implements parameterized nodes of a dynamic size for n-ary trees. The **D_Node**<*Type,nchild*> class is dynamic in the sense that the number of subtrees allowed for each node is not fixed. **D_Node**<*Type,nchild*> uses the **Vector**<*Type*> class to support run-time growth characteristics. Both classes are parameterized for the type and a number of subtrees that each node may have. In addition, the constructors for both classes are declared in the public section to allow the user to create nodes and control the building and structure of an n-ary tree where the ordering can have a specific meaning, as with an expression tree.

The **N_Tree**<*Node,Type,nchild*> class implements n-ary trees, providing the organizational structure for a tree (collection) of nodes while knowing nothing about the specific type of node used. **N_Tree**<*Node,Type,nchild*> is parameterized over a node type, a data type, and subtree count, where the node specified must have a data member of the same *Type* as the tree class. The subtree count indicates the number of possible subtree pointers (children) from any given node. Two node classes are provided, but others can also be written.

Symbol and Package Classes – The **Symbol** and **Package** classes implement the basic COOL symbolic computing support as standard C++ classes. These classes support efficient and flexible symbolic computing by providing symbolic constants and run-time symbol objects. Programmers can create symbolic constants at compile-time and manipulate symbol objects in a package at run-time

The **Symbol** class implements the notion of a symbol that has a name with an optional value and property list. Symbols are interned into a package, which is a mechanism for establishing separate name spaces. Because each named symbol is unique within its own package, the symbol can be used as a dynamic enumeration type and as a run-time variable. The **Package** class implements a package as a hash table of named symbols and includes support for adding, retrieving, updating, and removing symbols at run-time. It also provides completion and spelling correction on a **Symbol** name.

Generic Class – The **Generic** class is inherited by most other COOL classes and manipulates lists of symbols to manage type information. **Generic** adds run-time type checking and object queries, formatted print capabilities, and a describe mechanism to any derived class. The COOL **class** macro automatically generates the necessary implementation code for these member functions in the derived classes. A significant benefit of this common base class is the ability to declare heterogeneous container classes parameterized over the **Generic*** type. These classes, combined with the current position and parameterized iterator class, lets the programmer manipulate collections of objects of different types in a simple, efficient manner.

Class Hierarchy

1.10 The COOL class hierarchy implements a rather flat inheritance tree, as opposed to the deeply nested SmallTalk model. All complex classes are derived from the **Generic** class, to facilitate run-time type checking and object query. Simple classes are not derived from the **Generic** class due to space efficiency concerns. The parameterized container classes inherit from a base class that results in shared type-independent code. This reduces code replication when a particular type of container is parameterized several times for different objects in an application. The COOL class hierarchy is shown on the following page.

- **Pair**<*Ktype,Vtype*>
- **Range**
  - **Range**<*Type*>
- **Rational**
- **Complex**
- **Bignum**
- **Generic**
  - **String**
  - **Gen_String**
  - **Regexp**
  - **Vector**
    - **Vector**<*Type*>
      - **Association**<*Ktype,Vtype*>
  - **List_Node**
    - **List_Node**<*Type*>
  - **List**
    - **List**<*Type*>
  - **Date_Time**
  - **Timer**
  - **Bit_Set**
  - **Exception**
    - **Warning**
    - **Error**
      - **System_Error**
      - **Verify_Error**
    - **Fatal**
    - **System_Signal**
  - **Excp_Handler**
    - **Jump_Handler**
  - **Hash_Table**
    - **Set**
    - **Hash_Table**<*Key,Value*>
      - **Package**
  - **Matrix**
    - **Matrix**<*Type*>
  - **Queue**
    - **Queue**<*Type*>
  - **Random**
  - **Stack**
    - **Stack**<*Type*>
  - **Symbol**
  - **Binary_Node**
    - **Binary_Node**<*Type*>
  - **Binary_Tree**
    - **Binary_Tree**<*Type*>
      - **AVL_Tree**<*Type*>
  - **N_Node**<*Type,nchild*>
  - **D_Node**<*Type,nchild*>
  - **N_Tree**<*Type,Node,nchild*>

STRING CLASSES

2

Introduction

2.1 The string classes are a collection of classes that implement textual operations and functions for such commonplace actions as string concatenation and growth, allocating memory as necessary and thus relieving the programmer from having to perform this task manually. The following classes and functions are discussed in this section:

- **String Class**
- **char*** functions
- **Regular Expression (Regexp) Class**
- **Gen_String Class**

The **String** class implements dynamic, efficient strings comparable to the built-in **char*** data type. The **char*** functions supplement the standard ANSI C character string library functions. Taken together, **String** and **char*** provide such operations as string concatenation, case sensitive and insensitive comparison, case conversion, and simple string search and replacement. The **Regexp** class provides a convenient mechanism to present regular expressions for complex pattern matching and replacement, and uses the built-in **char*** data type. The **Gen_String** class combines the functions of the **String** and **Regexp** classes, along with reference counting and garbage collection, to provide sophisticated character string manipulation.

Requirements

2.2 This section assumes you have a working knowledge of the C++ language and type system. In addition, you should understand the distinction between the concepts associated with overloaded operators, member functions, and friend functions.

String Class

2.3 The **String** class provides dynamic, efficient strings for a C++ application. The intent is to provide efficient **char***-like functionality that frees the programmer from worrying about memory allocation and deallocation problems, yet retains the speed and compactness of a standard **char*** implementation. The **String** class is dynamic in the sense that if an operation such as concatenate results in more characters than can fit in the currently allocated memory, the string object *grows* according to some established size or ratio value. All typical string operations are provided, including concatenation, case-sensitive and case-insensitive lexical comparison, string search, yank, delete, and replacement. System-provided functions for **char*** such as **strcpy** and **strcmp** are also available via the overloaded **operator char*** member function.

Name: **String** — Simple, dynamic string class

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

Base Classes: **Generic**

Friend Classes: None

Constructors:

String ();
Initializes an empty string object with a default size block of memory allocated to hold 100 characters.

String (**char** *c*);
Initializes a string object with the default size block of memory allocated to hold 100 characters whose value is the string consisting of the single character argument *c*.

String (**const char*** *str*);
Initializes a string object with the default size block of memory allocated to hold 100 characters whose value is copied from the specified character string argument *str*. If *str* is longer, the string will grow as necessary.

String (**const char*** *str*, **long** *size*);
Allocates an initial block of memory the size of the integer argument *size*, or the **strlen**(*str*), whichever is longer. and initializes the string object with a copy of the specified character string *str*. Note that *size* is ignored if less than the length of *str*.

String (**const String&** *str*);
Duplicates the size and value of another string object *str*.

String (**const String&** *str*, **long** *size*);
Duplicates the size and value of another string object *str* by allocating an initial block of memory to be the size of the integer argument *size*, or **strlen**(*str*), whichever is longer. The duplication is then performed.

Member Functions:

inline long capacity () **const**;
Returns the maximum number of characters that the string object can contain without having to grow.

void clear ();
Resets the **NULL** character string terminator to the beginning of the string and sets the length of the string to zero.

Boolean insert (**const char*** *str*, **long** *position*);
Inserts a copy of the sequence of characters pointed to by the first argument *str* at the zero-relative index provided by the second argument *position*. This function returns **TRUE** if successful; otherwise, this function returns **FALSE** if the index is out of range.

inline operator char* () **const**;
Provides an implicit conversion operator to convert a string object into a **char*** value.

String operator+ (**char** *c*);
Overloads the addition operator to concatenate a single character *c* to a string object. This function returns a new string object, via the stack.

String operator+ (**const char*** *str*);
Overloads the addition operator to concatenate a copy of the specified character sequence *str* to a string object. This function returns a new string object.

String operator+ (**const String&** *str*);
Overloads the addition operator to concatenate the value of the specified string object *str* to a string object. This function returns a new string object.

inline String& operator= (**char** *c*);
Overloads the assignment operator to assign a single character *c* to a string object. This function returns a reference to the modified string object.

inline String& operator= (**const char*** *str*);
Overloads the assignment operator to assign a copy of the specified character sequence *str* to a string object. This function returns a reference to the modified string object.

inline String& operator= (**const String&** *str*);
Overloads the assignment operator to assign the value of another string object *str* to a string object. This function returns a reference to the modified string object.

inline String& operator+= (**char** *c*);
Overloads the addition-and-assignment operator to concatenate a single character *c* to a string object. This function returns a reference to the modified string object.

inline String& operator+= (**const char*** *str*);
Overloads the addition-and-assignment operator to concatenate a copy of the specified character sequence *str* to a string object. This function returns a reference to the modified string object.

inline String& operator+= (**const String&** *str*);
Overloads the addition-and-assignment operator to concatenate the value of the specified string object *str* to a string object. This function returns a reference to the modified string object.

inline Boolean operator== (**const char*** *str*) **const**;
Overloads the equality operator for the **String** class. This function returns **TRUE** if the string object and *str* have the same sequence of characters; otherwise, this function returns **FALSE**.

inline Boolean operator== (**const String&** *str*) **const**;
Overloads the equality operator for the **String** class. This function returns **TRUE** if the strings have the same sequence of characters; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const char*** *str*) **const**;
Overloads the inequality operator for the **String** class. This function returns **FALSE** if the string object and *str* have the same sequence of characters; otherwise, this function returns **TRUE**.

inline Boolean operator!= (**const String&** *str*) **const**;
Overloads the inequality operator for the **String** class. This function returns **FALSE** if the strings have the same sequence of characters; otherwise, this function returns **TRUE**.

inline Boolean operator< (**const char*** *str*) **const**;
Overloads the less-than operator for the **String** class. This function returns **TRUE** if the string is lexically less than the *char* s* argument; otherwise, this function returns **FALSE**.

inline Boolean operator< (**const String&** *str*) **const**;
Overloads the less-than operator for the **String** class. This function returns **TRUE** if the string object is lexically less than the string *str*; otherwise, this function returns **FALSE**.

inline Boolean operator<= (**const char*** *str*) **const**;
Overloads the less-than-or-equal operator for the **String** class. This function returns **TRUE** if the string object is lexically less than or equal to the character string argument *str*; otherwise, this function returns **FALSE**.

inline Boolean operator<= (**const String&** *str*) **const**;
Overloads the less-than-or-equal operator for the **String** class. This function returns **TRUE** if the string object is lexically less than or equal to the string *str*; otherwise, this function returns **FALSE**.

inline Boolean operator> (**const char*** *str*) **const**;
Overloads the greater-than operator for the **String** class. This function returns **TRUE** if the string object is lexically greater than the character string argument *str*; otherwise, this function returns **FALSE**.

inline Boolean operator> (**const String&** *str*) **const**;
Overloads the greater-than operator for the **String** class. This function returns **TRUE** if the string object is lexically greater than the string *str*; otherwise, this function returns **FALSE**.

inline Boolean operator>= (**const char*** *str*) **const**;
Overloads the greater-than-or-equal operator for the **String** class. This function returns **TRUE** if the string object is lexically greater than or equal to the character string argument *str*; otherwise, this function returns **FALSE**.

inline Boolean operator>= (**const String&** *str*) **const**;
Overloads the greater-than-or-equal operator for the **String** class. This function returns **TRUE** if the string object is lexically greater than or equal to the string *str*; otherwise, this function returns **FALSE**.

inline char operator[] (**long** *position*) **const**;
Returns the character at the zero-relative index *position* in the string. If the index is invalid, an Error exception is raised.

Boolean remove (**long** *start*, **long** *end*);
Removes the sequence of characters between the zero-relative inclusive *start* and exclusive *end* indexes. This function returns **TRUE** if successful; otherwise, this function returns **FALSE** if either one or both of the indexes is out of range.

Boolean replace (**const char*** *str*, **long** *start*, **long** *end*);
Replaces the sequence of characters between the zero-relative inclusive *start* and exclusive *end* indexes with a copy of the character string *str*. This function returns **TRUE** if successful; otherwise, this function returns **FALSE** if either one or both of the indexes is out of range.

void resize (**long** *size*);
Resizes the string object to hold at least *size* number of characters. If *size* is less than existing length, the operation is ignored.

void reverse ();
Reverses the ordering of the characters in a string object.

inline void set_alloc_size (**int** *size*);
Updates the allocation growth size to be used when the growth ratio is zero. Default allocation growth size is 100 bytes. If *size* is negative, an Error exception is raised.

inline void set_growth_ratio (**float** *ratio*);
Updates the growth ratio for this instance of a string to the specified value. When a string needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

void sub_string (**String&** *str*, **long** *start*, **long** *end*);
Sets the given string object *str* to the values in the character sequence between the zero-relative inclusive *start* and exclusive *end* indexes provided. This function returns **TRUE** if successful; otherwise, this function returns **FALSE** if either one or both of the indexes is out of range.

void yank (**String&** *str*, **long** *start*, **long** *end*);
Deletes the sequence of characters between the zero-relative inclusive *start* and exclusive *end* indexes provided and sets the given string object *str* to the value of the deleted characters.

Friend Functions:

inline friend double atof (**const String&** *str*);
Returns the floating-point value represented by the characters in the string object *str*.

friend int atoi (**const String&** *str*);
Returns the decimal radix integer number represented by the characters in the string object *str*.

friend long atol (**const String&** *str*);
Returns the decimal radix long number represented by the characters in the string object *str*.

friend String& capitalize (**String&** *str*);
Capitalizes each word and returns the modified string *str*. A word is defined to be any subsequence of alphanumeric characters. This function returns a reference to the modified string *str*.

friend String& downcase (**String&** *str*);
Converts any alphabetic character to lowercase. This function returns a reference to the modified string *str*.

friend String& left_trim (**String&** *str1*, **const char*** *str2*);
Removes any prefix occurrence of the character string *str2* specified from the string object *str1*. This function returns a reference to the modified string *str1*.

friend ostream& operator<< (**ostream&** *os*, **const String&** *str*);
Overloads the output operator for a reference to a string object *str*.

inline friend ostream& operator<< (**ostream&** *os*, **const String*** *str*);
Overloads the output operator for a pointer to a string object *str*.

friend String& right_trim (**String&** *str1*, **const char*** *str2*);
Removes any suffix occurrence of the character string *str2* specified from the string object *str1*. This function returns a reference to the modified string *str2*.

friend String& strcat (**String&** *str*, **char** *c*);
Returns the result of concatenating the character *c* to a string object *str*.

friend String& strcat (**String&** *str1*, **const char*** *str2*);
Returns the result of concatenating a copy of the specified character string *str2* to a string object *str1*.

friend String& strcat (**String&** *str1*, **const String&** *str2*);
Returns the result of concatenating one string object *str2* to another *str1*. This function returns the modified string object *str1*.

friend char* strchr (**const String&** *str*, **char** *c*);
Overloads the forward character search function to scan from left to right through a string object *str* for the first occurrence of the character *c*. This function returns a pointer to the character if found; otherwise, this function returns **NULL**.

friend String& strcpy (**String&** *str1*, **char** *str2*);
Overloads the copy string function to copy the character string *str2* into a string argument *str1*. This function returns a reference to the modified string object *str1*.

friend String& strcpy (**String&** *str1*, **const char*** *str2*);
Overloads the copy string function to copy the specified character string *str2* argument into the string argument *str1*. This function returns a reference to the modified string object *str1*.

friend String& strcpy (**String&** *str1*, **const String&** *str2*);
Overloads the copy string function to copy the second string argument *str2* into the first string argument *str1*. This function returns the modified string object *str1*.

inline friend long strlen (**const String&** *str*);
Returns the number of characters (length) of the string *str*.

friend String& strncat (**String&** *str1*, **const char*** *str2*, **int** *length*);
Returns the result of concatenating a copy of some number of characters *length* from a character string *str2* to a string object *str1*. This function returns a reference to the modified string object *str1*.

friend String& strncat (**String&** *str1*, **const String&** *str2*, **int** *length*);
Returns the result of concatenating some number of characters *length* from one string object *str2* to another *str1*. This function returns a reference to the modified string object *str1*.

friend String& strncpy (**String&** *str1*, **const char*** *str2*, **long** *length*);
Overloads the **strncpy** function to copy some number of characters *length* from the specified character string argument *str2* into the string argument *str1*. This function returns a reference to the modified string object *str1*.

friend char* strrchr (**const String&** *str*, **char** *c*);
Overloads the backward character search function to scan from right to left through a string object *str* for the last occurrence of a specific character *c*. This function returns a pointer to the character if found; otherwise, this function returns **NULL**.

friend double strtod (**const String&** *str*, **char**** *ptr* = **NULL**);
Returns the double floating-point value represented by the characters in the string object *str*. If the second argument is non-zero, it is set to the character terminating the converted string value.

friend long strtol (**const String&** *str*, **char**** *ptr* = **NULL**, **int** *radix*=10);

Returns the long number represented by the characters in the string object *str*. If a specific radix is not specified, the default radix is decimal. If the second argument is non-zero, it is set to the character terminating the converted string value.

friend String& trim (**String&** *str1*, **const char*** *str2*);

Removes any occurrence of the character string *str2* from the string object *str1*. This function returns a reference to the modified string *str1*.

friend String& upcase (*S***tring&** *str*);

Converts any alphabetic character to uppercase. This function returns a reference to the modified string *str*.

String Example

2.4 The following program declares a string object and manipulates it with several member functions to change its value and size. Several of the overloaded **String** operators are used to perform concatenation and assignment. After each operation is complete, the resulting string is printed.

```
 1   #include <COOL/String.h>

 2   int main (void) {
 3     String s1 = "Hello";                                       // Create string
 4     cout << "s1 reads: " << s1 << "\n";                        // Display string
 5     cout << "s1 has " << strlen (s1) << " characters\n";       // Display count
 6     s1 = s1 + " " + "world!";                                  // Concatenate
 7     cout << "s1 reads: " << s1 << "\n";                        // Display string
 8     cout << "s1 has " << strlen (s1) << " characters\n";       // Display count
 9     s1.reverse ();                                             //  Reverse order
10     cout << "s1 backwards reads: " << s1 << "\n";              // Output string
11     s1.reverse ();                                             // Restore order
12     cout << "s1 upper case: " << upcase (s1) << "\n";          // Uppercase value
13     cout << "s1 lower case: " << downcase (s1) << "\n";        // Downcase value
14     cout << "s1 capitalized: " << capitalize (s1) << "\n";     // Capitalized value
15     s1.insert ("Oh, ", 0);                                     // Insert at start
16     cout << "s1 reads: " << s1 << "\n";                        // Display string
17     s1.replace ("Goodbye", 4, 9);                              // Replace ‘hello’
18     cout << "s1 reads: " << s1 << "\n";                        // Display string
19     s1.remove (4, 12);                                         // Remove ‘goodbye’
20     cout << "s1 reads: " << s1 << "\n";                        // Display string
21     exit (0);                                                  // Exit with OK
22     }
```

Line 1 includes the COOL `String.h` class header file. Line 3 declares a new string object and initializes it with the word `Hello`. Lines 4 and 5 output the value of the string and the number of characters it contains. Line 6 uses the overloaded **operator+** to concatenate a space and the word `World` to the string object. The result and new character count is then output in lines 7 and 8. Line 9 uses the **reverse()** member function to reverse the order of the letters in the string object. Again, the results are output in line 10, and then reverted back in line 11 with another call to the same reverse member function. Lines 12 through 14 change the case and output the value of the string object. Line 15 adds some letters to the string object by using the **insert()** member function, and line 16 outputs the result. Lines 17 through 19 replace and then remove letters from the string object with the result outputted after each operation. Finally, the program ends with a valid exit code.

The following shows the output from the program:

```
s1 reads: Hello
s1 has 5 characters
s1 reads: Hello world!
s1 has 12 characters
s1 backwards reads: !dlrow olleH
s1 upper case: HELLO WORLD!
s1 lower case: hello world!
s1 capitalized: Hello World!
s1 reads: Oh, Hello World!
s1 reads: Oh, Goodbye World!
s1 reads: Oh, World!
```

Auxiliary char* Functions

2.5 The ANSI C specification requires a collection of standard functions for the manipulation of character strings. However, these do not include some of the more useful functions found in the **String** and **Gen_String** classes. In addition, many generic character string functions can be used for the **String** and **Gen_String** objects because of the implicit **operator char***. For these reasons, the following auxiliary functions are defined for the built-in **char*** data type:

Name: **char*** — Auxiliary character string functionality

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

Friend Functions:

Boolean is_equal (const char* *str1***, const char*** *str2***, Boolean** *case_flag* = **FALSE);**
Compares two character strings for lexical equality. If *case_flag* is **TRUE**, a case-sensitive comparison is made; otherwise, a case-insensitive comparison is made. This function returns **TRUE** if the strings are lexically equivalent; otherwise, this function returns **FALSE**.

Boolean is_not_equal (const char* *str1***, const char*** *str1***, Boolean** *case_flag* = **FALSE);**
Compares two character strings for lexical inequality. If *case_flag* is **TRUE**, a case-sensitive comparison is made; otherwise, a case-insensitive comparison is made. This function returns **FALSE** if the strings are lexically equivalent; otherwise, this function returns **TRUE**.

Boolean is_equal_n (const char* *str1***, const char*** *str1***, int** *n***, Boolean** *case_flag* = **FALSE);**
Compares *n* characters in two character strings for lexical equality. If *case_flag* is **TRUE**, a case-sensitive comparison is made; otherwise, a case-insensitive comparison is made. This function returns **TRUE** if the strings are lexically equivalent; otherwise, this function returns **FALSE**.

Boolean is_ge (const char* *str1***, const char*** *str2***, Boolean** *case_flag***);**
This function returns **TRUE** if *str1* is lexically greater than or equal to *str2*; otherwise, this function returns **FALSE**. If *case_flag* is **TRUE**, a case-sensitive comparison is made; otherwise, a case-insensitive comparison is made.

Boolean is_gt (const char* *str1***, const char*** *str1***, Boolean** *case_flag***);**
This function returns **TRUE** if *str1* is lexically greater than *str2*; otherwise, this function returns **FALSE**. If *case_flag* is **TRUE**, a case-sensitive comparison is made; otherwise, a case-insensitive comparison is made.

Boolean is_le (const char* *str1***, const char*** *str1***, Boolean** *case_flag***);**
This function returns **TRUE** if *str1* is lexically less than or equal to *str2*; otherwise, this function returns **FALSE**. If *case_flag* is **TRUE**, a case-sensitive comparison is made; otherwise, a case-insensitive comparison is made.

Boolean is_lt (const char* *str1***, const char*** *str2***, Boolean** *case_flag***);**
This function returns **TRUE** if *str1* is lexically less than *str2*; otherwise, this function returns **FALSE**. If the *Boolean* value is **TRUE**, a case-sensitive comparison is made; otherwise, a case-insensitive comparison is made.

char* c_capitalize (char* *str***);**
Capitalizes each word and returns a pointer to the modified character string *str*. A word is defined to be any subsequence of alphanumeric characters.

char* c_downcase (char* *str***);**
Converts any alphabetic character to lowercase. This function returns a pointer to the modified character string *str*.

char* c_left_trim (char* *str1***, const char*** *str2***);**
Removes any prefix occurrence of the second character string *str2* in the first character string *str1*. This function returns a pointer to the modified character string *str1*.

char* c_right_trim (char* *str1***, const char*** *str2***);**
Removes any suffix occurrence of the second character string *str2* in the first character string *str1*. This function returns a pointer to the modified character string *str1*.

char* c_trim (char* *str1***, const char*** *str2***);**
Removes any occurrence of the second character string *str2* from the first character string *str1*. This function returns a pointer to the modified character string *str1*.

char* c_upcase (char* *str***);**
Converts any alphabetic character to uppercase. This function returns a pointer to the modified character string *str*.

char* strfind (const char* *str1*, **const char*** *str2*, **long&** *start* = 0,
long& *end* = 0**);**
This function provides simple pattern matching by scanning from left to right through *str1* for the first occurrence of *str2*. An asterisk can be used to match for zero or more characters and a question mark can be used to match for any single character. This function returns a pointer to the start of the matching character string and updates the zero-relative *start* and *end* indexes if found; otherwise, this function returns **NULL**.

char* strrfind (const char* *str1*, **const char*** *str2*, **long&** *start* = 0,
long& *end* = 0**);**
This function provides simple pattern matching by scanning from right to left through *str1* for the first occurrence of *str2*. An asterisk can be used to match for zero or more characters and a question mark can be used to match for any single character. This function returns a pointer to the start of the matching character string and updates the zero-relative *start* and *end* indexes if found; otherwise, this function returns **NULL**.

char* strndup (const char* *str*, **long** *position***);**

Duplicates into a new character string allocated off the heap the sequence of characters from the beginning of *str* to the zero-relative index *position*. This function returns a pointer to the duplicated character string if successful; otherwise, this function returns **NULL** if the index is out of range.

char* strnremove (char* *str*, **long** *position***);**

Removes the sequence of characters from the beginning of *str* to the zero-relative index *position*. This function returns a pointer to the new character string if successful; otherwise, this function returns **NULL** if the index is out of range.

char* stryank (char* *str*, **long** *position***);**

Deletes the sequence of characters from the beginning of *str* to the zero-relative index *position* and allocates a new character string off the heap whose value is the deleted characters. This function returns a pointer to the yanked string if successful; otherwise, this function returns **NULL** if the index is out of range.

void reverse (char* *str***);**

Reverses the order of characters in a string *str*.

Regular Expression (Regexp) Class

2.6 A regular expression allows a programmer to specify complex patterns that can be searched for and matched against the character string of a string object. In its simplest form, a regular expression is a sequence of characters used to search for exact character matches. However, many times the exact sequence to be found is not known, or only a match at the beginning or end of a string is desired. The COOL regular expression class implements regular expression pattern matching as is found and implemented in many UNIX commands and utilities.

The regular expression class provides a convenient mechanism for specifying and manipulating regular expressions. The regular expression object allows specification of such patterns by using the following regular expression meta-characters:

^	Matches at beginning of a line
$	Matches at end of a line
.	Matches any single character
[ ]	Matches any character(s) inside the brackets
[^]	Matches any character(s) *not* inside the brackets
–	Matches any character in range on either side of a dash
*	Matches preceding pattern zero or more times
+	Matches preceding pattern one or more times
?	Matches preceding pattern zero or once only
()	Saves a matched expression and uses it in a later match

Note that more than one of these metacharacters can be used in a single regular expression in order to create complex search patterns. For example, the pattern `[^ab1-9]` says to match any character sequence that does not begin with the characters `"ab"` followed by numbers in the series one through nine.

Name:	**Regexp** — Regular expression pattern matching
Synopsis:	**#includ**e <COOL/Regexp.h>
Base Classes:	**Generic**
Friend Classes:	None

Constructors:

inline Regexp ();
Creates an empty regular expression object with no private data initialized.

inline Regexp (**char*** *str*);
Creates a regular expression object and initializes all the data by compiling the regular expression provided *str*. If an invalid regular expression is detected, an Error exception is raised.

Regexp (**const Regexp&** *reg*);
Creates a regular expression object and duplicates the values and regular expression of another regular expression object *reg*.

Member Functions:

void compile (**char*** *str*);
Creates a compiled version of the argument *str* and places it in the private data. If an invalid expression is detected, an Error exception is raised.

Boolean deep_equal (**const Regexp&** *reg*) **const**;
Determines if two regular expressions are the same, including the zero-relative start and end indexes of the last successful pattern match. This function returns **TRUE** if the expressions are the same; otherwise, this function returns **FALSE**.

inline long end () **const**;
Returns an index into the last character string successfully searched for by this object. The index corresponds to the character after the last item found. If none are found, its value is **NULL**.

Boolean find (**char*** *str*);
Searches for the already specified regular expression in *str*. If the expression is found, this function returns **TRUE** and sets start and end indexes appropriately. If an invalid expression is detected, an Error exception is raised.

inline Boolean is_valid () **const**;
Returns **TRUE** if a valid regular expression is compiled and ready for use; otherwise, this function returns **FALSE**.

Boolean operator== (**const Regexp&** *reg*) **const**;
Determines if two regular expression objects are the same. This function returns **TRUE** if the expression objects are the same; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Regexp&** *reg*) **const**;
Determines if two regular expression objects are not the same. This function returns **TRUE** if the expression objects are different; otherwise, this function returns **FALSE**.

inline void set_invalid ();
Invalidates the current regular expression of the regular expression object.

inline long start () **const**;

Returns an index into the last character string successfully searched for by this object. The index corresponds to the beginning of the last item found. If none are found, its value is **NULL**.

Regular Expression Example

2.7 The following program utilizes the COOL regular expression class to set and search for several patterns. The first is a simple character match, the second a search for a range of characters, and the third a complex match using sub-patterns. Each pattern and string to be searched is printed, along with the ensuing matches and zero-relative index results.

```
#include <COOL/Regexp.h>                        // Include Regexp header file

int main (void) {
 Regexp r1("Hi There");                         // Define simple pattern
 char* dummy = "Garbage Hi There garbage";      // Dummy string to search
 cout << "The pattern 'Hi There' ";             // Output start of sentence
 if (r1.find (dummy) == TRUE)                   // Pattern found in string?
  cout << "is";                                 // Yes, indicate affirmative
 else
  cout << "is not";                             // Else indicate failure
 cout << " found in '" << dummy << "\n";        // And complete output
 cout << "The pattern begins at zero-relative index " << r1.start ();
 cout << " and ends at index " << r1.end () << "\n";
 r1.compile("[^ab1-9]");                        // Complex pattern
 strcpy (dummy, "ab123QQ59ba");                 // Another string to search
 cout << "The pattern '[^ab1-9]' ";             // Output start of sentence
16     if (r1.find (dummy) == TRUE)              // Pattern found in string?
17      cout << "is";                            // Yes, indicate affirmative
18     else
19      cout << "is not";                        // Else indicate failure
20     cout << " found in '" << dummy << "\n";   // And complete output
21     cout << "The pattern begins at zero-relative index " << r1.start ();
22     cout << " and ends at index " << r1.end () << "\n";
23     r1.compile("O(.*r)");                     // New complex pattern
24     strcpy (dummy,"That's OK for me. OK for you?");// Another string to search
25     cout << "The pattern 'O(.*r)' ";          // Output start of sentence
26     if (r1.find (dummy) == TRUE)              // Pattern found in string?
27      cout << "is";                            // Yes, indicate affirmative
28     else
29      cout << "is not";                        // Else indicate failure
30     cout << " found in '" << dummy << "\n";   // And complete output
31     cout << "The pattern begins at zero-relative index " << r1.start ();
32     cout << " and ends at index " << r1.end () << "\n";
33     exit (0);                                 // Exit with OK status
34    }
```

Line 1 includes the COOL `Regexp.h` class header file. Lines 3 and 4 define a simple regular expression object and a character string pattern to be searched. Lines 5 through 12 search the character string for the pattern, output the search results, and indicate the zero-relative start and end points. Line 13 sets a more complex pattern for the regular expression object. This says to match anything that does not begin with the characters `"ab"` followed by numbers in the series one through nine. Lines 15 through 22 search the character string for this pattern, output the search results, and indicate the zero-relative start and end points. Line 23 establishes a pattern that matches a character string beginning with `"O"`, followed by a sequence of zero or more characters that ends with `"r"`. Lines 25 through 32 search the character string for this pattern, output the search results, and indicate the zero-relative start and end points.

The following shows the output from the program:

```
The pattern 'Hi There' is found in 'Garbage Hi There garbage'
The pattern begins at zero-relative index 8 and ends at index 16
The pattern '[^ab1-9]' is found in 'ab123QQ59ba'
The pattern begins at zero-relative index 5 and ends at index 6
The pattern 'O(.*r)' is found in 'That's OK for me. OK for you?'
The pattern begins at zero-relative index 7 and ends at index 24
```

General String Class

2.8 The **Gen_String** class provides general-purpose, dynamic strings for a C++ application with support for reference counting, delayed copy, and regular expression pattern matching. The intent is to provide sophisticated character string functionality for the application programmer. As in the **String** class, interface and member functions provide typical string operations. These include concatenation, case-sensitive and case-insensitive lexical comparison, string search, yank, delete, and replacement. In addition, the inclusion of regular expression pattern matching facilitates easier use of this COOL class with character strings. The **Gen_String** class is dynamic in the sense that if an operation such as concatenate results in more characters than can fit in the currently allocated memory, the string object *grows* according to some established size or ratio value. System-provided functions for **char*** such as **strcpy** and **strcmp** are also available via the overloaded **operator char*** member function.

Name: **Gen_String** — Dynamic general-purpose strings with reference counting, delayed copy, and regular expression pattern matching

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

Base Classes: **Generic**

Friend Classes: None

Constructors:

Gen_String ();
Initializes an empty general string object with the default size block of memory allocated to hold 100 characters.

Gen_String (**char** *c*);
Initializes a general string object with the default size block of memory allocated to hold 100 characters whose value is the general string consisting of the single character *c*.

Gen_String (**const char*** *str*);
Initializes a general string object with a default size block of memory allocated to hold 100 characters whose value is a copy of the specified character string argument *str*. If *str* is longer than 100 char then the **Gen_String** will grow to the correct size.

Gen_String *(***const char*** *str*, **long** *size*);
Allocates an initial block of memory of size *size* or size **strlen**(*str)* whichever is longer. Initializes the general string object with a copy of the specified character string *str*.

Gen_String (**const Gen_String&** *str*);
Duplicates the size and value of another general string object *str*.

Gen_String (**const Gen_String&** *str*, **long** *size*);
Duplicates the size and value of another general string object *str* and allocates an initial block of memory of size *size* or size **strlen**(*str*) whichever is longer.

Member Functions:

inline long capacity () **const**;
Returns the maximum number of characters that the general string object can contain without having to grow.

void clear ();
Resets the **NULL** character sting terminator to the beginning of the general string and sets the length of the general string to zero.

void compile (**char*** *str*);
Creates a compiled version of the regular expression argument *str*. If an invalid expression is detected, an Error exception is raised. Note that you must recompile a regular expression after changing the value of the **Gen_String** object.

inline long end () **const**;
Returns an index into the last string successfully searched for by this object. The index corresponds to the character after the last item found, or if none was found (the uninitialized state) then its value is **NULL**.

Boolean find ();
Searches for the first or next established regular expression in the string. If the expression is found, this function returns **TRUE** and sets start and end appropriately. If an invalid expression is detected, an Error exception is raised.

Boolean insert (**const char*** *str*, **long** *position*);
Inserts a copy of the sequence of characters *str* at the zero-relative index *position*. This function returns **TRUE** if successful; otherwise, this function returns **FALSE** if the index *position* is out of range.

inline Boolean is_valid () **const**;
Returns **TRUE** if a valid regular expression is compiled and ready for use; otherwise, this function returns **FALSE**.

Gen_String operator+ (**char** *c*);
Overloads the addition operator to concatenate a single character *c* to a general string object. This function returns a new general string object.

Gen_String operator+ (**const char*** *str*);
Overloads the addition operator to concatenate a copy of the character sequence *str* to a general string object. This function returns a new general string object.

Gen_String operator+ (**const Gen_String&** *str*);
Overloads the addition operator to concatenate the value of another general string object *str* to a general string object. This function returns a new general string object.

inline Gen_String& operator= (**char** *c);*
Overloads the assignment operator to assign a single character *c* to a general string object. This function returns a reference to the modified general string object.

inline Gen_String& operator= (**const char*** *str*);
Overloads the assignment operator to assign a copy of the character sequence *str* to a general string object. This function returns a reference to the modified general string object.

inline Gen_String& operator= (**const Gen_String&** *str*);
Overloads the assignment operator to assign the value of another general string object *str* to a general string object. This function returns a reference to the modified general string object.

Gen_String& operator+= (**char** *c*);
Overloads the addition-and-assignment operator to concatenate a single character *c* to a general string object. This function returns a reference to the modified general string object.

Gen_String& operator+= (**const char*** *str*);
Overloads the addition-and-assignment operator to concatenate a copy of the character sequence *str* to a general string object. This function returns a reference to the modified general string object.

Gen_String& operator+= (**const Gen_String&** *str*);
Overloads the addition-and-assignment operator to concatenate the value of another general string object *str* to a general string object. This function returns a reference to the modified general string object.

inline Boolean operator== (**const char*** *str*) **const**;
Overloads the equality operator for the **Gen_String** class. This function returns **TRUE** if the general string object and *str* have the same sequence of characters; otherwise, this function returns **FALSE**.

inline Boolean operator== (**const Gen_String&** *str*) **const**;

Overloads the equality operator for the **Gen_String** class. This function returns **TRUE** if the general strings have the same sequence of characters; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const char*** *str*) **const**;

Overloads the inequality operator for the **Gen_String** class. This function returns **FALSE** if the general string object and *str* have the same sequence of characters; otherwise, this function returns **TRUE**.

inline Boolean operator!= (**const Gen_String&** *str*) **const**;

Overloads the inequality operator for the **Gen_String** class. This function returns **FALSE** if the general strings have the same sequence of characters; otherwise, this function returns **TRUE**.

inline Boolean operator< (**const char*** *str*) **const**;

Overloads the less-than operator for the **Gen_String** class. This function returns **TRUE** if the general string object is lexically less than *str*; otherwise, this function returns **FALSE**.

inline Boolean operator< (**const Gen_String&** *str*) **const**;

Overloads the less-than operator for the **Gen_String** class. This function returns **TRUE** if the general string object is lexically less than *str*; otherwise, this function returns **FALSE**.

inline Boolean operator<= (**const** *c***har*** *str*) **const**;

Overloads the less-than-or-equal operator for the **Gen_String** class. This function returns **TRUE** if the general string object is lexically less than or equal to *str*; otherwise, this function returns **FALSE**.

inline Boolean operator<= (**const Gen_String&** *str*) **const**;

Overloads the less-than-or-equal operator for the **Gen_String** class. This function returns **TRUE** if the general string object is lexically less than or equal to *str*; otherwise, this function returns **FALSE**.

inline Boolean operator> (**const char*** *str*) **const**;

Overloads the greater-than operator for the **Gen_String** class. This function returns **TRUE** if the general string object is lexically greater than *str*; otherwise, this function returns **FALSE**.

inline Boolean operator> (**const Gen_String&** *str*) **const**;

Overloads the greater-than operator for the **Gen_String** class. This function returns **TRUE** if the general string object is lexically greater than *str*; otherwise, this function returns **FALSE**.

inline Boolean operator>= (**const char*** *str*) **const**;

Overloads the greater-than-or-equal operator for the **Gen_String** class. This function returns **TRUE** if the general string object is lexically greater than or equal to *str*; otherwise, this function returns **FALSE**.

inline Boolean operator>= (**const Gen_String&** *str*) **const**;

Overloads the greater-than-or-equal operator for the **Gen_String** class. This function returns **TRUE** if the general string object is lexically greater than or equal to *str*; otherwise, this function returns **FALSE**.

inline char operator[] (**long** *position*) **const**;

Returns the character at the zero-relative index *position* into the general string. If an invalid index is specified, an **Error** exception is raised.

inline operator char* () **const**;
Provides an implicit conversion operator to convert a string object into a **char*** value.

Boolean remove (**long** *start*, **long** *end*);
Removes the sequence of characters between the zero-relative inclusive *start* and exclusive *end* indexes provided. This function returns **TRUE** if successful; otherwise, this function returns **FALSE** if either one or both of the indexes is out of range.

Boolean replace (**const char*** *str*, **long** *start*, **long** *end*);
Replaces the sequence of characters between the zero-relative inclusive *start* and exclusive *end* indexes with a copy of the character string *str*. This function returns **TRUE** if successful; otherwise, this function returns **FALSE** if either one or both of the indexes is out of range.

void resize (**long** *size*);
Resizes the general string object to hold at least *size* characters. If a negative size is specified, an **Error** exception is raised.

void reverse ();
Reverses the ordering of the characters in a general string object. The **Reg_Exp** does not need to be recompiled.

inline void set_alloc_size (**int** *size*);
Updates the allocation growth size to be used when the growth ratio is zero. Default allocation growth size is 100 bytes. If a negative size is specified, an **Error** exception is raised.

inline void set_growth_ratio (**float** *ratio*);
Updates the growth ratio for this instance of a **Gen_String** class object to the specified value. When a string needs to grow, the current size is multiplied by the ratio to determine the new size. If a negative growth ratio is specified, an **Error** exception is raised.

inline long start () **const**;
Returns an index into the last string successfully searched for by that object. The index corresponds to the beginning of the last item found.

void sub_string (**Gen_String&** *str*, **long** *start*, **long** *end*);
Sets the general string object *str* to the values of the character sequence between the zero-relative inclusive *start* and exclusive *end* indexes provided. This function returns **TRUE** if successful; otherwise, if the start or end indexes or both are out of range, an **Error** exception is raised and, if the handler returns, this function returns **FALSE**.

void yank (**Gen_String&** *str*, **long** *start*, **long** *end*);
Deletes the sequence of characters between the zero-relative inclusive *start* and exclusive *end* indexes provided and sets the string object *str* to the value of the deleted characters. If the start or end or both indexes are out of range, an **Error** exception is raised.

Friend Functions:

inline friend double atof (**const Gen_String&** *str);*
Returns the floating-point value represented by the characters in the general string object *str*.

friend int atoi (**const Gen_String&** *str*);
Returns the decimal radix integer number represented by the characters in the general string object *str*.

friend long atol (**const Gen_String&** *str*);
Returns the decimal radix long number represented by the characters in the general string object *str*.

friend Gen_String& capitalize (**Gen_String&** *str*);
Capitalizes each word and returns the modified general string *str*. A word is defined to be any subsequence of alphanumeric characters, but it must start with a letter. This function returns a reference to the modified general string.

friend Gen_String& downcase (**Gen_String&** *str*);
Converts any alphabetic character to lowercase. This function returns a reference to the modified general string *str*.

friend Gen_String& left_trim (**Gen_String&** *str1*, **const char*** *str2*);
Removes any prefix occurrence of the character string *str2* in the general string object *str1*. This function returns a reference to the modified general string *str1*.

friend ostream& operator<< (**ostream&** *os*, **const Gen_String&** *str*);
Overloads the output operator for a reference to a general string object.

inline friend ostream& operator<< (**ostream&** *os*, **const Gen_String*** *str*);
Overloads the output operator for a pointer to a general string object.

friend Gen_String& right_trim (**Gen_String&** *str1*, **const char*** *str2*);
Removes any suffix occurrence of the character string *str2* in the general string object *str1*. This function returns a reference to the modified general string *str1*.

friend Gen_String& strcat (**Gen_String&** *str*, **char** *c*);
Concatenates a single character *c* to a general string object *str*. This function returns a reference to the modified general string *str*.

friend Gen_String& strcat (**Gen_String&** *str1*, **const char*** *str2*);
Concatenates a copy of the character string *str2* to a general string object *str1*. This function returns a reference to the modified general string *str1*.

friend Gen_String& strcat (**Gen_String&** *str1*, **const Gen_String&** *str2*);
Concatenates one general string object *str2* to another general string object *str1*. This function returns a reference to the modified general string *str1*.

friend char* strchr (**const Gen_String&** *str*, **char** *c*);
Overloads the forward character-search function to scan from left to right through a general string object *str* for the first occurrence of a specific character *c*. This function returns a pointer to the character if found; otherwise, this function returns **NULL**.

friend Gen_String& strcpy (**Gen_String&** *str*, **char** *c*);
Returns the result of copying a single character into a general string object *str*. This function returns a reference to the modified general string object *str*.

friend Gen_String& strcpy (**Gen_String&** *str1***, const char*** *str2*);
Copies a character string *str2* into a general string object *str1*. This function returns a reference to the modified general string object *str1*.

friend Gen_String& strcpy (**Gen_String&** *str1*, **const Gen_String&** *str2*);
Copies one general string object *str2* into another general string object *str1*. This function returns a reference to the modified general string object *str1*.

inline friend long strlen (**const Gen_String&** *str*);
Returns the number of characters (length) of the general string *str*.

friend Gen_String& strncat (**Gen_String&** *str1*, **const char*** *str2***, int** *n*);
Concatenates some number of characters *n* from a character string *str2* to a general string object *str1*. This function returns a reference to the modified general string object *str1*. If a negative length is specified, an **Error** exception is raised.

friend Gen_String& strncat (**Gen_String&** *str1*, **const Gen_String&** *str2*, *int n*);
Concatenates some number of characters *n* from one general string object *str2* to another general string object *str1*. This function returns a reference to the modified general string object *str1*. If a negative length is specified, an **Error** exception is raised.

friend Gen_String& strncpy (**Gen_String&** *str1*, **const char*** *str2*, *long n*);
Copies some number of characters *n* from the character string *str2* into the general string object *str1*. This function returns a reference to the modified general string object *str1*. If a negative number is specified, an **Error** exception is raised.

friend char* strrchr (**const Gen_String&** *str*, **char** *c*);
Overloads the backward character-search function to scan from right to left through a general string object *str* for the last occurrence of a specific character *c*. This function returns a pointer to the character if found; otherwise, this function returns **NULL**.

friend double strtod (**const Gen_String&** *str*, **char**** *ptr* = **NULL**);
Returns the double floating-point value represented by the characters in the general string object *str*. If the second argument is non-zero, it is set to the character terminating the converted string value.

friend long strtol (**const Gen_String&** *str*, **char**** *ptr*=**NULL**, *int radix*=10);
Returns the long number represented by the characters in the general string object *str*. If no specific radix is specified, the default radix is decimal. If the second argument is non-zero, it is set to the character terminating the converted string value.

friend Gen_String& trim (**Gen_String&** *str1*, **const char*** *str2*);
Removes any occurrence of the character string *str2* in the general string object *str1*. This function returns a reference to the modified general string *str1*.

friend Gen_String& upcase (**Gen_String&** *str*);
Converts any alphabetic character to uppercase. This function returns a reference to the modified general string *str*.

General String Example

2.9 The following program uses the COOL general string class to show the combined functionality of the previous two classes. A general string object is declared and manipulated with several of the member functions to change its value and size. Several of the overloaded **Gen_String** operators are used to perform concatenation and assignment. After each operation is complete, the resulting string is printed. In addition, several search operations using the built-in regular expression capability are performed. The first is a simple character match, the second a search for a range of characters, and the third a complex match using sub-patterns. Each pattern and string to be searched is printed, along with the ensuing matches and zero-relative index results.

```
#include <COOL/Gen_String.h>              // Include header file
#include <COOL/Regexp.h>                  // Include header file

int main (void) {
 Gen_String s1 = ″Hello″;                                     // Create string
 cout << ″s1 reads: ″ << s1 << ″\n″;                          // Display string
 cout << ″s1 has ″ << strlen (s1) << ″ characters\n″;         // Display count
 s1 = s1 + ″ ″ + ″world!″;                                    // Join characters
 cout << ″s1 reads: ″ << s1 << ″\n″;                          // Display string
 cout << ″s1 has ″ << strlen (s1) << ″ characters\n″;         // Display count
 s1.reverse ();                                               // Reverse order
 cout << ″s1 backwards reads: ″ << s1 << ″\n″;                // Output string
 s1.reverse ();                                               // Restore order
 cout << ″s1 upper case: ″ << upcase (s1) << ″\n″;            // Uppercase value
 cout << ″s1 lower case: ″ << downcase (s1) << ″\n″;          // Downcase value
 cout << ″s1 capitalized: ″ << capitalize (s1) << ″\n″;       // Capitalized value
 s1.insert (″Oh, ″, 0);                                       // Insert at start
 cout << ″s1 reads: ″ << s1 << ″\n″;                          // Display string
 s1.replace (″Goodbye″, 4, 9);                                // Replace ‘hello’
 cout << ″s1 reads: ″ << s1 << ″\n″;                          // Display string
 s1.remove (4, 12);                                           // Remove ‘goodbye’
 cout << ″s1 reads: ″ << s1 << ″\n″;                          // Display string
 s1.compile(″Hi There″);                                      // Define pattern
 s1 = ″Garbage Hi There garbage″;                             // Set search string
 cout << ″The pattern ‘Hi There’ ″;                           // Output start
 if (s1.find () == TRUE)                                      // Pattern found?
  cout << ″is″;                                               //  Yes
 else
  cout << ″is not″;                                           // Else failure
 cout << ″ found in ‘″ << s1 << ″\n″;                         // Complete output
 cout << ″The pattern begins at zero-relative index ″ << s1.start ();
 cout << ″ and ends at index ″ << s1.end () << ″\n″;
 s1.compile(″[^ab1-9]″);                                      // Complex pattern
 s1 = ″ab123QQ59ba″;                                          // Search string
 cout << ″The pattern ‘[^ab1-9]’ ″;                           // Output start
 if (s1.find () == TRUE)                                      // Pattern found?
  cout << ″is″;                                               //  Yes
 else
  cout << ″is not″;                                           // Else failure
 cout << ″ found in ‘″ << s1 << ″\n″;                         // Complete output
 cout << ″The pattern begins at zero-relative index ″ << s1.start ();
 cout << ″ and ends at index ″ << s1.end () << ″\n″;
 s1.compile(″O(.*r)″);                                        // New pattern
 s1 = ″That’s OK for me. OK for you?″;                        // Another string
 cout << ″The pattern ‘O(.*r)’ ″;                             // Output start
 if (s1.find () == TRUE)                                      // Pattern found?
  cout << ″is″;                                               //  Yes
47    else
48     cout << "is not";                                  // Else failure
49    cout << " found in '" << s1 << "\n";                // Complete output
50    cout << "The pattern begins at zero-relative index " << s1.start ();
51    cout << " and ends at index " << s1.end () << "\n";
52    exit (0);                                           // Exit with OK
53   }
```

Line 1 includes the COOL `Gen_String.h` class header file, and line 2 includes the `Regexp.h` class header file. Lines 4 through 9 perform the assignment and concatenation of a character string to the **Gen_String** object. Lines 10–15 reverse the order of the characters in the object and manipulate the case of the words. Lines 16 through 21 insert, replace, and remove various characters from the object. Lines 22 through 31 demonstrate use of the built-in regular expression function in the **Gen_String** class with the first pattern used in the **Regexp** example program. Lines 32 through 41 demonstrate use of the second pattern, and lines 42 through 51 use the third pattern from the regular expression program. Finally, line 52 ends the program with a successful status.

The following shows the output from the program:

```
s1 reads: Hello
s1 has 5 characters
s1 reads: Hello world!
s1 has 12 characters
s1 backwards reads: !dlrow olleH
s1 upper case: HELLO WORLD!
s1 lower case: hello world!
s1 capitalized: Hello World!
s1 reads: Oh, Hello World!
s1 reads: Oh, Goodbye World!
s1 reads: Oh, World!
The pattern 'Hi There' is found in 'Garbage Hi There garbage'
The patter begins at zero-relative index 8 and ends at index 16
The pattern '[^ab1-9]' is found in 'ab123QQ59ba'
The pattern begins at zero-relative index 5 and ends at index 6
The pattern 'O(.*r)' is found in 'That's OK for me. OK for you?'
The pattern begins at zero-relative index 7 and ends at index 24
```

3 NUMBER CLASSES

Introduction

3.1 Simple integers and floating point numbers do not provide the needed precision for many applications. The COOL number classes are a collection of numerically-oriented classes that augment the built-in numerical data types to provide such features as extended precision, range-checked types, and complex numbers. The following classes are discussed in this section:

- **Random**
- **Complex**
- **Rational**
- **Bignum**
- **Range**<*Type,lbound,hbound*>

The **Random** class implements five variations of random number generators, each with different portability, efficiency, and accuracy characteristics. The **Complex** class implements the complex number type for C++ and provides all of the basic arithmetic and trigonometric functions. The **Rational** class uses the built-in **long** type to implement an extended precision rational data type for resolving inadequate round-off or truncation results from the built-in numerical data types. The **Bignum** class implements near-infinite precision integers and arithmetic by using a dynamic bit vector.

Finally, the parameterized **Range**<*Type,lbound,hbound*> class enables arbitrary user-defined ranges to be implemented in C++ classes. Typically, but not always, this is used with other number classes to select a range of valid values for a particular numerical type. Features and advantages of the **Range**<*Type,lbound,hbound*> class are discussed in this section. However, complete details of parameterized templates are provided in Section 5.

Requirements

3.2 This section discusses the number classes. It assumes you have a working understanding of the C++ language and type system. In addition, you should understand the distinction between the concepts and ideas associated with overloaded operators and friend functions.

Random Class

3.3 The **Random** class provides several general-purpose random number generators with features similar to those as described in Chapter 7 of *Numerical Recipes in C*, written by William T. Vetterling. The ANSI C draft standard specifies the **rand** function that allows an application to obtain successive random numbers in a sequence by repeated calls. However, system-supplied random number generators in the form of the **rand** function are generally of poor quality, particularly when true random distribution over a range is important. Specifically, system random number generators are almost always linear congruential generators whose period is not very large. The ANSI C draft specification only requires a modulus of 32767, which can be disastrous for such uses as a Monte Carlo integration over 10^6 points.

The **Random** class allows an application to select one of five types of random number generators based upon the usage requirements. Each generator function has different characteristics and all are defined to be of type *RNG_TYPE*. The **SIMPLE** and **SHUFFLE** functions use the system **rand** function, while the **ONE_CONGRUENTIAL**, **THREE_CONGRUENTIAL**, and **SUBTRACTIVE** functions are self-contained, portable implementations. Following are descriptions of each generator function.

- **SIMPLE** — When speed is the predominant concern, this function uses the system-supplied **rand** function. Although sequential correlation of successive random values is a high probability, this function at least ensures that the value's least significant bits are as random as the most significant bits. In many system random generator functions, the value's least significant bits are often less random than the most significant bits.

- **SHUFFLE** — This function uses the **rand** function and a shuffling procedure. Random numbers are stored in a buffer and selected randomly to break up sequential correlation in the system-supplied function.

- **ONE_CONGRUENTIAL** — This self-contained function uses one linear congruential generator instead of the **rand** function to implement a portable random number generator. This guarantees no sequential correlation between the random values returned.

- **THREE_CONGRUENTIAL** — This portable function uses three linear congruential generators to implement a random number generator whose period is essentially infinite and has no sequential correlations.

- **SUBTRACTIVE** — This function implements a portable random number generator that does not use linear congruential generators, but rather an original subtractive member function as suggested in Volume 2 of *The Art of Computer Programmin*g, written by Donald Knuth.

Name:	**Random** — A portable, user-selectable random number generator
Synopsis:	**#include** <COOL/Random.h>
Base Classes:	**Generic**
Friend Classes:	None

Constructor:

Random (**RNG_TYPE** *r_type*, **int** *seed* = 1, **float** *lower* = 0.0,
float *upper* = 100.0);

Constructor for a floating-point random number generator that initializes the selected random number generator function with the user-supplied *seed* value.

Member Functions:

inline double next ();

Returns the next double floating-point random number within the user-specified range.

inline int get_seed () **const;**

Returns the seed value for the currently-selected random number generator.

inline void set_rng (**RNG_TYPE** *r_type*);

Sets the random number generator function to the type selected by the user and reinitializes the state.

inline void set_seed (**int** *seed*);
Sets the seed value for the currently-selected random number generator function and reinitilizes the state.

Random Class Example

3.4 The following program creates two random number objects using different generator algorithms to provide random numbers within a specified range. The first uses a variation of the system-supplied `rand()` function and the second a three-congruential linear generator. Ten random numbers from each are sent to the standard output.

```
1    #include <COOL/Random.h>                          // Include Random class

2    int main (void) {
3      Random r1 (SIMPLE, 1, 3.0, 9.0);                // Simple rand() generator
4      Random r2 (THREE_CONGRUENTIAL,1,5.0,11.5);      // Highly random generator
5      cout << ″Simple random number generator:\n″;    // Output banner title
6      for (int i = 0; i < 10; i++)                    // Generate 10 random numbers
7        cout << ″ Random number ″ << i << ″ is: ″ << r1.next () << ″\n″;
8      cout << ″\nThree congruential linear random number generator:\n″;
9      for (i = 0; i < 10; i++)                        // Generate 10 random numbers
10       cout << ″ Random number ″ << i << ″ is: ″ << r2.next () << ″\n″;
11     return (0);                                     // Exit with OK status
12   }
```

Line 1 includes the COOL `Random.h` class header file. Line 3 defines a random number generator of type `SIMPLE` for generator numbers within the range of 3.0 to 9.0 inclusive. Line 4 defines a random number generator of type `THREE_CONGRUENTIAL` for generator numbers within the range of 5.0 to 11.5 inclusive. Lines 6 through 10 utilize two loops to generate and print ten numbers from each generator. Finally, the program ends with a valid exit code.

The following shows the output from the program:

Simple random number generator:
```
Random number 0 is: 6.08322
Random number 1 is: 4.05445
Random number 2 is: 4.85191
Random number 3 is: 6.2072
Random number 4 is: 8.68577
Random number 5 is: 4.03042
Random number 6 is: 7.21339
Random number 7 is: 4.35858
Random number 8 is: 5.96864
Random number 9 is: 3.74832
```

Three congruential linear random number generator:
```
Random number 0 is: 9.26861
Random number 1 is: 7.84012
Random number 2 is: 8.84924
Random number 3 is: 7.22898
Random number 4 is: 8.1818
Random number 5 is: 7.3039
Random number 6 is: 9.18251
Random number 7 is: 10.0368
Random number 8 is: 10.3957
Random number 9 is: 11.3929
```

Complex Class

3.5 The **Complex** class is a complex number class with basic arithmetic support, conversion to and from built-in types, and simple arithmetic exception handling. A **Complex** object has the same precision and range of values as the system-defined type **double.** Implicit conversion to the system-defined types **short**, **int**, **long**, **float**, and **double** is supported by overloaded operator member functions. However, despite the implicit conversions and judicious use of inline member functions, arithmetic operations on **Complex** objects are slower than the built-in types.

The **Complex** class implements common arithmetic exception handling and provides application support for detecting negative infinity, positive infinity, overflow, and underflow that may result from an operation. If one of these conditions is detected or an attempt to convert from a **Complex** with no value to a built-in type is made, an exception is raised. The programmer can provide an exception handler at runtime to take care of this problem. If no such handler is available, an error message is printed and program execution ends. See Section 13 for more information on the COOL exception handling mechanism.

Name: **Complex** — Complex number class

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

Base Classes: None

Friend Classes: None

Constructors:

inline Complex ();
Creates a new complex number object initialized to floating point zero.

inline Complex (**double** *real,* **double** *imaginary* = 0.0);
Creates a new complex number object whose real part is set to *real* and whose imaginary part is initialized to the value of *imaginary*.

inline Complex (**const Complex&** *c*);
Creates a new complex number object whose real and imaginary parts are initialized to the values of those of another complex number *c*.

Member Functions:

inline Complex conjugate () const;
Calculates the conjugate of a complex number and returns a new object whose value is the negated imaginary value of the object. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Complex cos (**Complex&** *c*) **const**;
Calculates the cosine of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Complex cosh (**Complex&** *c*) **const**;
Calculates the hyperbolic cosine of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

operator double ();
Overloaded operator to provide implicit conversion between complex objects and the built-in **double** type when appropriate.

operator float ();

Overloaded operator to provide implicit conversion between complex objects and the built-in **float** type when appropriate.

inline double imaginary () **const**;

Returns the imaginary part of the complex number.

inline Complex invert () const;

Returns the reciprocal of a complex number. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

operator int ();

Overloaded operator to provide implicit conversion between complex objects and the built-in **int** type when appropriate.

operator long ();

Overloaded operator to provide implicit conversion between complex objects and the built-in **long** type when appropriate.

Complex operator– ();

Overloads the unary minus operator for the **Complex** class and returns a new object whose value is the negated real value of the object. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Complex& operator= (**const Complex&** *c*);

Overloads the assignment operator for the **Complex** class and assigns one complex number to have the value of another. A reference to the updated object is returned.

inline void operator+= (**const Complex&** *c*);

Overloads the addition-with-assignment operator for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline void operator–= (**const Complex&** *c*);

Overloads the subtraction-with-assignment operator for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline void operator*= (**const Complex&** *c*);

Overloads the multiplication-with-assignment operator for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline void operator/= (**const Complex&** *c*);

Overloads the division-with-assignment operator for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Complex& operator++ ();

Overloads the increment operator to provide an increment capability for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised. A reference to the updated complex object is returned.

inline Complex& operator— ();
Overloads the decrement operator to provide a decrement capability for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised. A reference to the updated complex object is returned.

inline Boolean operator! () **const**;
Overloads the logical NOT operator for the **Complex** class and returns **TRUE** if the complex number has a zero value; otherwise, this function returns **FALSE**.

inline Boolean operator== (**const Complex&** *c*) **const**;
Overloads the equality operator for the **Complex** class. This function returns **TRUE** if the complex numbers have the same value; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Complex&** *c*) **const**;
Overloads the inequality operator for the **Complex** class. This function returns **TRUE** if the complex numbers have different values; otherwise, this function returns **FALSE**.

inline double real () **const**;
Returns the real part of the complex number.

operator short ();
Overloaded operator to provide implicit conversion between complex objects and the built-in **short** type when appropriate.

inline Complex sin (**Complex&** *c*) **const**;
Calculates the sine of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Complex sinh (**Complex&** *c*) **const**;
Calculates the hyperbolic sine of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline N_Status status () **const**;
Returns the numerical exception state of the complex object.

inline Complex tan (**Complex&** *c*) **const**;
Calculates the tangent of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Complex tanh (**Complex&** *c*) **const**;
Calculates the hyperbolic tangent of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Friend Functions:

inline friend Complex operator+ (**const Complex&** *c1*, **const Complex&** *c2*);
Overloads the addition operator to provide addition for the **Complex** class. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend Complex operator– (const Complex& *c1*,
const Complex& *c2*);
Overloads the subtraction operator for the **Complex** class. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend Complex operator* (**const Complex&** *c1*,
const Complex& *c2*);
Overloads the multiplication operator for the **Complex** class. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Complex operator/ (**const Complex&** *c1*, **const Complex&** *c2*);
Overloads the division operator to provide division for the **Complex** class. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend ostream& operator<< (**ostream&** *os*, **const Complex&** *c*);
Overloads the output operator for a reference to a complex object to provide a formatted output.

inline friend ostream& operator<< (**ostream&** *os*, **const Complex*** *c*);
Overloads the output operator for a pointer to a complex object to provide a formatted output.

Complex Example

3.6 The following impedance example using complex numbers is accredited to a LISP program published in *LISP*, written by Patrick Henry Winston and Berthold Klaus Paul Horn. This example calculates the impedance of an electrical circuit operating at a given frequency by using standard formulas from basic hardware design texts.

```
#include <COOL/Complex.h>                    // Include complex header file
#define FREQUENCY 346.87
#define OMEGA (2 * 3.14159265358979323846 * FREQUENCY)

inline Complex in_series (const Complex& c1, const Complex& c2) {
 return (c1+c2);
}

inline Complex in_parallel (const Complex& c1, const Complex& c2) {
 return ((c1.invert() + c2.invert()).invert ());
}

inline Complex resistor (double r) {
 return Complex (r);
}

inline Complex inductor (double i) {
 return (Complex (0.0, i * OMEGA));
}

inline Complex capacitor (double c) {
 return (Complex (0.0, -1.0 / (c * OMEGA)));
}

int main (void) {
 Complex circuit;
 circuit = in_series (resistor (1.0),
22                  in_parallel (in_series (resistor (100.0), inductor (0.2)),
23                             in_parallel (capacitor (0.000001),
24                                  resistor (10000000.0))));
25       cout << "Circuit impedance is " << circuit << " at frequency " <<
                    FREQUENCY << "\n";
26       return 0;                               // Exit with OK status
27     }
```

Line 1 includes the COOL `Complex.h` class header file. Lines 2 and 3 define a frequency constant and a value `OMEGA` based upon pi and the frequency and is used in calculating impedance formulas. Lines 4 through 6 define a function for calculating the impedance of two components placed in series. Similarly, lines 7 through 9 define a function for calculating the impedance of two components placed in parallel. Lines 10 through 18 provide functions for determining the impedance of resistors, inductors, and capacitors, based upon their tolerances. Line 21 is the heart of the program that calls the necessary functions to calculate the impedance of a circuit. Finally, the result is sent to the standard output and the program ends with a successful exit code. Figure 3.1 illustrates the circuit used in this example program.

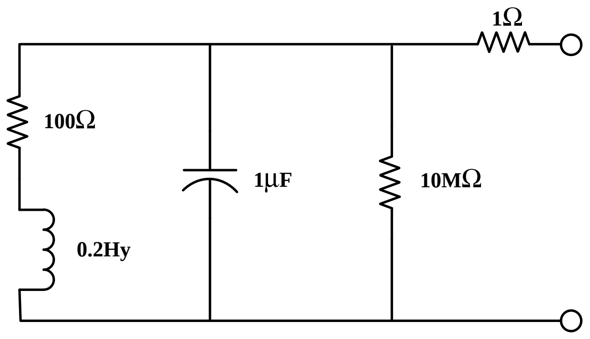

Figure 3.1

The following shows the output from the program:

```
Circuit impedance is (2000.6,0.00747744) at frequency 346.87
```

Rational Class

3.7 The **Rational** class provides infinite precision rational numbers and arithmetic using the built-in **long** type for the numerator and denominator objects. Consequently, a rational object will grow in 32-bit chunks as necessary. Implicit conversion to the system-defined types **short**, **int**, **long**, **float**, and **double** is supported by overloaded operator member functions. However, arithmetic operations on rational objects are slower than the built-in integer types.

The **Rational** class implements common arithmetic exception handling and provides application support for detecting negative infinity, positive infinity, overflow, and underflow that may result from an operation. If one of these conditions is detected or if an attempt to convert from a **Rational** with no value to a built-in type is made, an exception is raised. The programmer can provide an exception handler at runtime to take care of this problem. If no such handler is available, an error message is printed and program execution terminates. See Section 13 for more information on the COOL exception handling mechanism.

Name: **Rational** — Infinite precision rational numbers

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

Base Classes: **Generic**

Friend Classes: None

Constructors:

inline Rational ();
Simple constructor to create a new rational object.

Rational (**long** *n,* **long** *d* = 1);
Constructor that specifies an integer numerator and optional denominator arguments to create a new rational object.

Rational (**const Rational&** *r*);
Constructor that takes a reference to an existing rational object and creates a new object with the same value.

Member Functions:

inline long ceiling () **const**;
Returns an integer that represents the value of the rational object truncated towards positive infinity.

inline long denominator () **const**;
Returns the denominator value of the object.

inline operator double ();
Overloaded operator to provide implicit conversion between rational objects and the built-in **double** type when appropriate.

operator float ();
Overloaded operator to provide implicit conversion between rational objects and the built-in **float** type when appropriate.

inline long floor () **const**;
Returns an integer that represents the value of the rational object truncated towards negative infinity.

operator int ();
Overloaded operator to provide implicit conversion between rational objects and the built-in **int** type when appropriate.

Rational& invert ();
Returns a reference to the inverse of the rational number object.

operator long ();
Overloaded operator to provide implicit conversion between rational objects and the built-in **long** type when appropriate.

inline long numerator () **const**;
Returns the numerator value of the object.

inline Rational operator–();
Overloads the unary minus operator for the **Rational** class and returns a new object whose value is the negated value of the object. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Rational& operator= (**const Rational&** *r*);
Overloads the assignment operator for the **Rational** class and assigns one rational number to have the value of another. A reference to the updated object is returned.

void operator+= (**const Rational&** *r*);
Overloads the addition-with-assignment operator for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline void operator–= (**const Rational&** *r*);
Overloads the subtraction-with-assignment operator for the **Rational** class. If the operation results in an arithmetic exception of some type, the appropriate exception is raised.

void operator*= (**const Rational&** *r*);
Overloads the multiplication-with-assignment operator for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline void operator/= (**const Rational&** *r*);
Overloads the division-with-assignment operator for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator%= (**const Rational&** *r*);
Overloads the modulus with assignment operator for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Boolean operator!() **const**;
Overloads the logical NOT operator for the **Rational** class and returns **TRUE** if the complex number has a zero value; otherwise, this function returns **FALSE**.

inline Rational& operator++ ();
Provides an increment capability for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised. A reference to the modified Rational object is returned.

inline Rational& operator— ();
Provides a decrement capability for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised. A reference to the modified Rational object is returned.

inline Boolean operator== (**const Rational&** *r*) **const**;
Overloads the equality operator for the **Rational** class. This function returns **TRUE** if the rational numbers have the same value; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Rational&** *r*) **const**;
Overloads the inequality operator for the **Rational** class. This function returns **TRUE** if the rational numbers have different values; otherwise, this function returns **FALSE**.

Boolean operator< (**const Rational&** *r*) **const**;
Overloads the less-than-operator for the **Rational** class and returns **TRUE** if the object is less than the specified argument; otherwise, this function returns **FALSE**.

inline Boolean operator<= (**const Rational&** *r*) **const**;
Overloads the less-than-or-equal operator for the **Rational** class. This function returns **TRUE** if the object is less than or equal to the value of the specified argument; otherwise, this function returns **FALSE**.

Boolean operator> (**const Rational&** *r*) **const**;
Overloads the greater-than operator for the **Rational** class and returns **TRUE** if the object is greater than the specified argument; otherwise, this function returns **FALSE**.

inline Boolean operator>= (**const Rational&** *r*) **const**;
Overloads the greater-than-or-equal operator for the **Rational** class. This function returns **TRUE** if the object is greater than or equal to the value of the specified argument ; otherwise, this function returns **FALSE**.

long round () **const**;
Returns an integer that represents the value of the rational object truncated towards the nearest integer.

operator short ();
Overloaded operator to provide implicit conversion between rational objects and the built-in **short** type when appropriate.

inline N_Status status () **const**;
Returns the numerical exception state of the rational object.

inline long truncate () **const**;
Returns an integer that represents the value of the rational object truncated towards zero.

Friend Functions:

friend Rational operator+ (**const Rational&** *r1*, **const Rational&** *r2*);
Overloads the addition operator for the **Rational** class. A new rational object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend Rational operator– (**const Rational&** *r1*, **const Rational&** *r2*);
Overloads the subtraction operator to provide subtraction for the **Rational** class. A new rational object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Rational operator* (**const Rational&** *r1*, **const Rational&** *r2*);
Overloads the multiplication operator for the **Rational** class. A new rational object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend Rational operator/ (**const Rational&** *r1*, **const Rational&** *r2*);
Overloads the division operator for the **Rational** class. A new rational object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Rational operator% (**const Rational&** *r1*, **const Rational& r2**);
Overloads the modulus operator for the **Rational** class. A new rational object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend ostream& operator<< (**ostream&** *os*, **const Rational&** *r*);
Overloads the output operator for a reference to a rational object to provide a formatted output capability.

inline friend ostream& operator<< (**ostream&** *os*, **const Rational*** *r*);
Overloads the output operator for a pointer to a rational object to provide a formatted output capability.

Rational Example

3.8 The following program uses the **Rational** class and the built-in **float** type to illustrate the added precision available for calculations involving multiplication, division, addition, and determining the remainder for numeric ratios. The first half of the program calculates answers for problems using the **Rational** class. The second half calculates answers for the same problems using the built-in **float** type. The results from each are printed on the standard output stream for comparison of precision.

```
1     #include <COOL/Rational.h>                    // Include COOL Rational class

2     int main (void) {
3       Rational r1 (10,3);                         // Create rational object
4       Rational r2 (-4,27), r3;                    // Create rational objects
5       r3 = r1 + r2;                               // Calculate sum of values
6       cout << r1 << " + " << r2 << " = " << r3 << "\n"; // And display result
7       r3 = r1 * r2;                               // Calculate product of values
8       cout << r1 << " * " << r2 << " = " << r3 << "\n"; // And display result
9       r3 = r1 / r2;                               // Calculate quotient of values
10      cout << r1 << " / " << r2 << " = " << r3 << "\n"; // And display result
11      r3 = r1 % r2;                               // Calculate remainder of values
12      cout << r1 << " % " << r2 << " = " << r3 << "\n"; // And display result

13      double d1 = double (10.0 / 3.0);            // Create double ratio
14      double d2 = double (-4.0 / 27.0), d3;       // Create double ratios
15      d3 = d1 + d2;                               // Calculate sum of values
16      cout << d1 << " + " << d2 << " = " << d3 << "\n"; // And display result
17      d3 = d1 * d2;                               // Calculate product of values
18      cout << d1 << " * " << d2 << " = " << d3 << "\n"; // And display result
19      d3 = d1 / d2;                               // Calculate quotient of values
20      cout << d1 << " / " << d2 << " = " << d3 << "\n"; // And display result
21      return 0;                                   // Return valid success code
22    }
```

Line 1 includes the COOL `Rational.h` class header file. Lines 3 and 4 declare three rational objects (`r1`, `r2`, `r3`), the first two of which have initial values of 10/3 and –4/27, respectively. Lines 5 and 6 calculate the sum of the two rational objects, assign it to the third, and display the answer. Likewise, lines 7 and 8 calculate the product, lines 9 and 10 calculate the quotient, and lines 11 and 12 calculate the remainder of the same two rational numbers. Lines 13 through 20 perform the same calculations with the built-in type **double** as were performed in lines 3 through 10. As indicated from the results, a loss of precision occurs from the floating point calculations, thus highlighting the potential benefit of using the ratios maintained by the **Rational** number class. Finally, the program ends with a valid exit code.

The following shows the output from the program:

```
10/3 + -4/27 = 86/27
10/3 * -4/27 = -40/81
10/3 / -4/27 = -45/2
10/3 % -4/27 = 2/27
3.33333 + -.148148 = 3.18519
3.33333 * -.148148 = -.493827
3.33333 / -.148148 = -22.5
```

Bignum Class

3.9 The **Bignum** class implements near-infinite precision integers and arithmetic by using a dynamic bit vector. A **Bignum** object will grow in size as necessary to hold its integer value. Implicit conversion to the system defined types **short**, **int**, **long**, **float**, and **double** is supported by overloaded operator member functions. Addition and subtraction operators are performed by simple bitwise addition and subtraction on **unsigned short** boundaries with checks for carry flag propagation. The multiplication, division, and remainder operations utilize the algorithms from Knuth's Volume 2 of "*The Art of Computer Programmin*g". However, despite the use of these algorithms and inline member functions, arithmetic operations on **Bignum** objects are considerably slower than the built-in integer types that use hardware integer arithmetic capabilities.

NOTE: The **Bignum** class requires that the built-in type **long** is larger than the built-in type **short** and can accommodate the result of multiplying two **short** values. The maximum positive value that can be represented by the **Bignum** class is:
`(2^(sizeof(unsigned long) * sizeof(unsigned short)))-1.`

The **Bignum** class supports the parsing of character string representations of all the literal number formats. The following table shows an example of a character string representation on the left and a brief description of the interpreted meaning on the right:

Character String Representation	*Interpreted Meaning*
1234	1234
1234l	1234
1234L	1234
1234u	1234
1234U	1234
1234ul	1234
1234UL	1234
01234	1234 in octal (leading 0)
0x1234	1234 in hexadecimal (leading 0x)
0X1234	1234 in hexadecimal (leading 0X)
123.4	123 (value truncated)
1.234e2	123 (exponent expanded/truncated)
1.234e–5	0 (truncated value less than 1)

The **Bignum** class implements common arithmetic exception handling and provides application support for detecting negative infinity, positive infinity, overflow, and underflow that may result from an operation. If one of these conditions is detected, an exception is raised. The programmer can provide an exception handler at runtime to take care of this problem. If no such handler is available, an error message is printed and program execution terminates. See Section 13 for more information on the COOL exception handling mechanism.

Name: **Bignum** — Infinite precision integers

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

Base Classes: **Generic**

Friend Classes: None

Constructors:

inline Bignum ();
Simple constructor to create a near-infinite precision integer object initialized to zero.

Bignum (**const char*** *str*);
Constructor to create a near-infinite precision integer object from the character string representation *str*.

Bignum (**double** *d*);
Constructor to create a near-infinite precision integer object from the double value *d*.

Bignum (**long** *l*);
Constructor to create a near-infinite precision integer object from the long integer value *l*.

Bignum (**const Bignum&** *bn*);
Constructor to create a near-infinite precision integer object from *bn*.

Member Functions:

operator double ();
Overloaded operator to provide implicit conversion between **Bignum** objects and the built-in **double** type when appropriate.

operator float ();
Overloaded operator to provide implicit conversion between **Bignum** objects and the built-in **float** type when appropriate.

operator int ();
Overloaded operator to provide implicit conversion between **Bignum** objects and the built-in **int** type when appropriate.

operator long ();
Overloaded operator to provide implicit conversion between **Bignum** objects and the built-in **long** type when appropriate.

Bignum operator– () **const**;
Overloads the unary minus operator for the **Bignum** class and returns a new object whose value is the negated value of the object. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Bignum& operator= (**const char*** *str*);
Overloads the assignment operator for the **Bignum** class and assigns the integer representation from the character string *str* to the near-infinite precision integer object. A reference to the updated object is returned.

Bignum& operator= (**const** *Bignum& bn*);
Overloads the assignment operator for the **Bignum** class and assigns *bn* to the near-infinite precision integer object. A reference to the updated object is returned.

inline Boolean operator! () **const**;
Overloads the unary negation operator for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Bignum operator~ () **const**;
Overloads the unary exclusive-or operator for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Bignum& operator++ ();
Overloads the increment operator to provide an increment capability for the **Bignum** class. A reference to the modified **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Bignum& operator— ();
Overloads the decrement operator to provide a decrement capability for the **Bignum** class. A reference to the modified **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator+= (**const Bignum&** *bn*);
Overloads the addition with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator–= (**const Bignum&** *bn*);
Overloads the subtraction with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator*= (**const Bignum&** *bn*);
Overloads the multiplication with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator/= (**const Bignum&** *bn*);
Overloads the division with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator%= (**const Bignum&** *bn*);
Overloads the modulus with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator&= (**const Bignum&** *bn*);
Overloads the logical AND with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator^= (**const Bignum&** *bn*);
Overloads the exclusive-or with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator|= (**const Bignum&** *bn*);
Overloads the logical OR with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator>>= (**const Bignum&** *bn*);
Overloads the right shift with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator<<= (**const Bignum&** *bn*);
Overloads the left shift with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Boolean operator== (**const Bignum&** *bn*) **const**;
Overloads the equality operator for the **Bignum** class. This function returns **TRUE** if the near-infinite precision integers have the same value; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Bignum&** *bn*) **const**;
Overloads the inequality operator for the **Bignum** class. This function returns **TRUE** if the near-infinite precision integers have different values; otherwise, this function returns **FALSE**.

Boolean operator< (**const Bignum&** *bn*) **const**;
Overloads the less than operator for the **Bignum** class and returns **TRUE** if the object is less than the specified argument; otherwise, this function returns **FALSE**.

inline Boolean operator<= (**const Bignum&** *bn*) **const**;
Overloads the less than or equal operator for the **Bignum** class. This function returns **TRUE** if the object is less than or equal to the value of the specified argument ; otherwise, this function returns **FALSE**.

Boolean operator> (**const Bignum&** *bn*) **const**;
Overloads the greater than operator for the **Bignum** class and returns **TRUE** if the object is greater than the specified argument; otherwise, this function returns **FALSE**.

inline Boolean operator>= (**const Bignum&** *bn*) **const**;
Overloads the greater than or equal operator for the **Bignum** class. This function returns **TRUE** if the object is greater than or equal to the value of the specified argument ; otherwise, this function returns **FALSE**.

operator short ();
Overloaded operator to provide implicit conversion between **Bignum** objects and the built-in **short** type when appropriate.

inline N_Status status () **const**;
Returns the numerical exception state of the Bignum object.

Friend Functions:

friend Bignum operator+ (**const** *Bignum& bn1*, **const** *Bignum& bn2*);
Overloads the addition operator to provide addition for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend Bignum operator– (**const Bignum&** *bn1*,
const Bignum& *bn2*);

Overloads the subtraction operator to provide subtraction for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator* (**const Bignum&** *bn1*, **const Bignum&** *bn2*);

Overloads the multiplication operator to provide multiplication for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator/ (**const Bignum&** *bn1*, **const Bignum&** *bn2*);

Overloads the division operator for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator% (**const Bignum&** *bn1*, **const Bignum&** *bn2*);

Overloads the modulus operator for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator& (**const Bignum&** *bn1*, **const Bignum&** *bn2*);

Overloads the logical AND operator for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator^ (**const Bignum&** *bn1*, **const Bignum&** *bn2*);

Overloads the logical exclusive-or operator for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator| (**const Bignum&** *bn1*, **const Bignum&** *bn2*);

Overloads the logical OR operator for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator>> (**const Bignum&** *bn1*, **const Bignum&** *bn2*);

Overloads the right shift operator for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator<< (**const Bignum&** *bn1*, **const Bignum&** *bn2*);

Overloads the left shift operator for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend ostream& operator<< (**ostream&** *os*, **const Bignum&** *bn*);

Overloads the output operator for a reference to a **Bignum** object to provide a formatted output.

inline friend ostream& operator<< (**ostream&** *os*, **const Bignum*** *bn*);

Overloads the output operator for a pointer to a **Bignum** object to provide a formatted output.

Bignum Example

3.10 The following program uses the **Bignum** integer data type in a semantically equivalent manner to the built-in **int** or **long** data types to perform arithmetic and logical operations. The only difference is that the values manipulated are larger than `MAX_INT` or `MAX_LONG` would allow on a 32-bit computer.

```
1   #include <COOL/Bignum.h>                  // Include Bignum class

2   int main (void) {
3    Bignum b1;                               // Create Bignum object
4    Bignum b2 = ”0xFFFFFFFF”;                // Create Bignum object
5    Bignum b3 = ”1.2345e30”;                 // Create Bignum object
6    cout << ”b2 = ” << b2 << ”\n”;           // Display value of b2
7    cout << ”b3 = ” << b3 << ”\n”;           // Display value of b3
8    b1 = b2 + b3;                            // Add b2 and b3
9    cout << ”b2 + b3 = ” << b1 << ”\n”;      // Display result
10   b1 = b2 – b3;                            // Subtract b3 from b2
11   cout << ”b2 – b3 = ” << b1 << ”\n”;      // Display result
12   b1 = b2 * b3;                            // Multiply b2 and b3
13   cout << ”b2 * b3 = ” << b1 << ”\n”;      // Display result
14   b1 = b3 / b2;                            // Divide b2 into b3
15   cout << ”b3 / b2 = ” << b1 << ”\n”;      // Display result
16   b1 = b3 % b2;                            // Get b3 modulo b2
17   cout << ”b3 % b2 = ” << b1 << ”\n”;      // Display result
18     return 0;                              // Exit with status code
19  }
```

Line 1 includes the COOL **Bignum** class header file. Line 3 creates a **bignum** object initialized to zero. Lines 3 and 4 create **Bignum** objects initialized to very large integer values. Lines 5 and 6 output these values on the standard output stream. Lines 8 through 17 compute the sum, difference, product, quotient, and remainder of various **Bignum** values and output the answer. Finally, the program ends with a valid exit code.

The following shows the output from the program:

```
b2 = 4294967295
b3 = 1234500000000000000000000000000
b2 + b3 = 1234500000000000000004294967295
b2 – b3 = -1234499999999999999995705032705
b2 * b3 = 5302137125674500000000000000000000000000
b3 / b2 = 287429429657624435997
b3 % b2 = 64281885
```

Range Class

3.11 The parameterized **Range**<*Type*,*lbound*,*hbound*> class enables arbitrary user-defined ranges to be implemented in C++ classes. Typically, but not always, this is used with other number classes to select a range of valid values for a particular numerical type. Features and advantages of this class are discussed in this section. However, complete details of parameterized templates are provided in Section 5.

The **Range**<*Type*,*lbound*,*hbound*> class is publicly derived from the **Range** class and supports user-defined ranges for a type of object or built-in data type. This allows other higher level data structures such as the **Rational** and **Complex** classes to be restricted to a range of values. The programmer does not have to add bounds-checking code to the application. A vector of positive integers, for example, would be easy to declare, facilitating bounds checking restricted to the code that implements the type, not the vector.

The inclusive low and high bounds for the range are specified as arguments to the parameterized type declaration and implementation macro calls. They are declared as C++ constants of the appropriate type. No storage is allocated, and all references are compiled out by the compiler. Once declared, a **Range**<*Type*,*lbound*,*hbound*> object cannot have its upper or lower bounds changed because maintenance of all instances would require significant and unwarranted overhead.

Name: **Range**<*Type*,*lbound*,*hbound*> — A parameterized range

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

Base Classes: **Range**

Friend Classes: None

Constructors:

Range<*Type*,*lbound*,*hbound*> ();
Creates an empty range object of the specified type and ranges.

Range<*Type*,*lbound*,*hbound*> (**const Type&** *value*);
Creates a range object with the specified value. If *value* is outside of the lower and upper bounds, an Error exception is raised.

Range<*Type*,*lbound*,*hbound*> (**const Range**<*Type*,*lbound*,*hbound*>**&** *r*);
Creates a new range object with the same value as the range object *r*.

Member Functions:

inline const Type& high () **const**;
Returns a reference to the upper limit of the range.

inline const Type& low () **const**;
Returns a reference to the lower limit of the range.

inline Range<*Type*,*lbound*,*hbound*>& **operator=**
(**const Range**<*Type*,*lbound*,*hbound*>**&** *r*);
Overloads the assignment operator for the **Range**<*Type*,*lbound*,*hbound*> class and assigns the range object the value of *r*. This function returns a reference to the updated object.

inline void set (**const Type&** *value*);
Sets the value of the range object to *value* if within the lower and upper limits; otherwise, this function raises an **Error** exception.

inline void set_compare (**Range_Compare** *r_fcn*);
Sets the compare function for this class of **Range**<*Type*,*lbound*,*hbound*>. **Range_Compare** is a function of type **int** (**Function*)(**const** *Type&*, **const** *Type&*).

inline operator Type () **const**;
Overloads the implicit conversion operator for the parameterized type to facilitate mixed-type expressions and statements.

Range Example

3.12 The following program declares two range-checking objects, one of type **double** and one of type **char***. Each has type-specific upper and lower bounds that, if violated, result in a run-time exception. Values are assigned to each object and the implicit use of the type conversion operator is demonstrated.

```
1   #include <COOL/Range.h>                     // Include range header file
2   #include <string.h>                         // C++ ANSI C string functions

3   DECLARE Range<double,2.5,8.8>;              // Declare range of doubles
4   IMPLEMENT Range<double,2.5,8.8>;            // Implement range of doubles
5   DECLARE Range<char*,"D", "K">;              // Declare range of strings
6   IMPLEMENT Range<char*, "D", "K">;           // Implement range of strings

7   int my_compare (const charP& s1, const charP& s2) {
8     return (strcmp (s1, s2));
9   }

10  int main (void) {
11                                              // Range-checked double
12    r1.set(4.3);                              // Assign value
13    cout << "r1 has an inclusive low bound of " << r1.low(); // Output low and
14    cout << "an inclusive high bound of " << r1.high() << ",\n"; // High
15    cout << "and a value of " << (double)r1 << "\n"; // Output value
16    double d1 = 1.9;                          // Declare a double
17    cout << (double)r1 << " * " << d1 << " = ";    // Output equation
18    r1.set (d1 * r1);                         // Calculate value
19    cout << (double)r1 << "\n";// And display it
20    Range<charP,"D","K"> r2;                  // Range-checked string
21    r2.set_compare (&my_compare);             // Set compare function
22    r2.set("EFG");                            // Assign value
23    cout << "r2 has an inclusive low bound of " << r2.low();
24    cout << "an inclusive high bound of " << r2.high() << ",\n";
25    cout << "a value of " << (char*)r2;       // Output string value
26    cout << ", and a length of " << strlen (r2) << "\n"; // Output length
27    return 0;                                 // Exit with OK status
28  }
```

Line 1 includes the COOL `Range.h` class header file and line 2 includes the COOL `String.h` class header file. Lines 3 through 6 declare and implement two kinds of range-checking objects: one a **double** with a low bound of 2.5 and a high bound of 8.8, and the other a character string object with a low bound of `"D"` and a high bound of `"K"`. Lines 7 through 9 define a comparison function for the range-checked string object, although in this program, it is not actually used. Line 11 declares a range object of type **double** with upper and lower bounds as before and line 12 gives this object a value. Lines 13 and 14 output the lower and upper bounds and line 15 displays the value of the object via a cast. Lines 16 through 18 show the object used in an arithmetic expression and line 19 prints the result.

Line 20 declares a range-checked string object and line 21 sets the default comparison routine for this object, should one be needed. Line 22 initializes the object with a string value. Lines 23 through 25 output the lower and upper bounds and the value. Line 26 displays the number of characters in the string by means of a system-supplied string length function and the implicit type conversion operator for the **Range** class. Finally, the program ends with a valid exit code.

The following shows the output from the program:

```
r1 has an inclusive low bound of 2.5, an inclusive high bound of 8.8,
and a value of 4.3
4.3 * 1.9 = 8.17
r2 has an inclusive low bound of D, an inclusive high bound of K,
a value of EFG, and a length of 3
```

SYSTEM INTERFACE CLASSES

4

Introduction

4.1 The COOL system interface classes encapsulate common system-specific functionality such as date-and-time manipulation and timing facilities. These classes provide a single interface for an application program no matter which of the supported platforms it is running on. This facilitates a single source base for an application designed to run on several types of hardware. The following classes are discussed in this section:

- **Date_Time**
- **Timer**

The **Date_Time** class implements time zone-independent date and time functions, including time zone changes, calendar date manipulation, and complete input parsing and output formatting capability for significant country or language formats. The **Timer** class uses the system **time**(2) interface to provide time resolution between a reference point and now. The accuracy of the time period reported is system-dependent, but will generally be either at millisecond or microsecond granularity.

Requirements

4.2 This section discusses the system interface classes. It assumes you have a working understanding of the C++ language and type system. In addition, you should understand the distinction between overloaded operators and friend functions.

Date_Time Class

4.3 The **Date_Time** class executes time zone-independent date and time functions. This class supports calendar operations and input and output based upon the value of an environmental synonym, such as **US_CENTRAL.** This class supports all time zones in the world, along with several special cases requiring alternate handling based upon political or daylight saving time differences. Unlike the ANSI C date and time functions, this class supports dates before the epoch (January 1, 1970). Year values specified between 0 and 99 are assumed to be in the twentieth century.

Name:	**Date_Time** — Time zone–independent date and time class
Synopsis:	**#include** <COOL/Date_Time.h>
Base Classes:	**Generic**
Friend Classes:	None

Public Constructors:

Date_Time ();
Allocates a date and time object with the default time zone and country. A Warning exception is raised if the default country or the default time has not been set for the class.

Date_Time (**const Date_Time&** *dt*);
Duplicates the size and entries of a date and time object *dt*.

Date_Time (**time_zone** *tz*, **countr***y c*);
Allocates a date and time object with time zone *tz* and country code *c*.

Member Functions:

const char* ascii_date () **const**;
Returns the date in ASCII format for the appropriate time zone and country.

const char* ascii_date_time () **const**;
Returns the date and time in ASCII format for the appropriate time zone and country.

const char* ascii_duration (**const Date_Time&** *dt*) **const**;
Returns the duration of time between the date/time object and *dt* in ASCII format.

const char* ascii_time () **const**;
Returns the time in ASCII format for the appropriate time zone and country.

inline void decr_day (**int** *n* = 1);
Decrements the time by the specified number of days. The default is one.

inline void decr_hour (**int** *n* = 1);
Decrements the time by the specified number of hours. The default is one.

inline void decr_min (**int** *n* = 1);
Decrements the time by specified number of minutes. The default is one.

void decr_month (**int** *n* = 1);
Decrements the time by specified number of months. The default is one.

inline void decr_sec (**int** *n* = 1);
Decrements the time by specified number of seconds. The default is one.

inline void decr_week (**int** *n* = 1);
Decrements the time by the specified number of weeks. The default is one.

void decr_year (**int** *n* = 1);
Decrements the time by the specified number of years. The default is one.

void end_day (**int** *n* = 1);
Advances the time by the specified number of days, setting the time to 23:59:59. The default is one.

void end_hour (**int** *n* = 1);
Advances the time by the specified number of hours, setting the time to hh:59:59. The default is one.

void end_min (**int** *n* = 1);
Advances the time by the specified number of minutes, setting the time to hh:mm:59. The default is one.

void end_month (**int** *n* = 1);
Advances the time by the specified number of months, setting the time to 31/mm/yyyy 23:59:59. The default is one.

void end_week (**int** *n* = 1);
Advances the time by the specified number of weeks, setting the time to Sunday 23:59:59. The default is one.

void end_year (**int** *n* = 1);
Advances the time by the specified number of months, setting the time to 31/12/yyyy 23:59:59. The default is one.

inline const char* get_country () **const**;
Returns the country in ASCII format.

inline int get_hour () **const**;
Returns the value of the hour data member in the object (0–23).

inline int get_mday () **const**;
Returns the value of the day of the month data member in the object (1–31).

inline int get_min () **const**;
Returns the value of the minutes data member in the object (0–59).

inline int get_mon () **const**;
Returns the value of the months data member in the object (0–11).

inline int get_sec () **const**;
Returns the value of the seconds data member in the object (0–59).

inline const char* get_time_zone () **const**;
Returns the time zone in ASCII format.

inline int get_wday () **const**;
Returns the value of the day of the week data member in the object (Sunday=0).

inline int get_yday () **const**;
Returns the value of the day of the year data member in the object (0–365)

inline int get_year () **const**;
Returns the value of the year data member in the object.

void incr_day (**int** *n* = 1);
Increments the time by the specified number of days. The default is one.

void incr_hour (**int** *n* = 1);
Increments the time by the specified number of hours. The default is one.

void incr_min (**int** *n* = 1);
Increments the time by the specified number of minutes. The default is one.

void incr_month (**int** *n* = 1);
Increments the time by the specified number of months. The default is one.

void incr_sec (**int** *n* = 1);
Increments the time by the specified number of seconds. The default is one.

void incr_week (**int** *n* = 1);
Increments the time by the specified number of weeks. The default is one.

void incr_year (**int** *n* = 1);
Increments the time by the specified number of years. The default is one.

inline Boolean is_day_light_savings () **const**;
Returns **TRUE** if daylight saving time is in effect; otherwise, returns **FALSE**.

inline long operator– (**const Date_Time&** *dt*);
Computes the interval of time between the date and time object and *dt*.

Date_Time& operator= (**const Date_Time&** *dt*);
Overloads the assignment operator to replicate the value of one date and time object to another.

Date_Time& operator+= (**long** *seconds*);
Performs interval addition and assignment.

Date_Time& operator–= (**long** *seconds*);
Performs interval subtraction and assignment.

inline Boolean operator== (**const Date_Time&** *dt*) **const**;
Overloads the equality operator for the **Date_Time** class. This function returns **TRUE** if two objects represent the same time; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Date_Time&** *dt*) **const**;
Overloads the inequality operator for the **Date_Time** class. This function returns **FALSE** if two objects represent the same time; otherwise, this function returns **TRUE**.

inline Boolean operator< (**const Date_Time&** *dt*) **const**;
Overloads the less-than operator for the **Date_Time** class. This function returns **TRUE** if the date and time object represents a date and time before *dt*; otherwise, this function returns **FALSE**.

inline Boolean operator<= (**const Date_Time&** *dt*) **const**;
Overloads the less-than-or-equal operator for the **Date_Time** class. This function returns **TRUE** if the date and time object represents a date and time before or equal to *dt*; otherwise, this function returns **FALSE**.

inline Boolean operator> (**const Date_Time&** *dt*) **const**;
Overloads the greater-than operator for the **Date_Time** class. This function returns **TRUE** if the date and time object represents a date and time after *dt*; otherwise, this function returns **FALSE**.

inline Boolean operator>= (**const Date_Time&** *dt*) **const**;
Overloads the greater-than-or-equal operator for the **Date_Time** class. This function returns **TRUE** if the date and time object represents a date and time equal to or after *dt*; otherwise, this function returns **FALSE**.

void parse (**char*** *str*, **int** *settz* = 0);
Parses the character string *str* input and fills all appropriate data members of the date and time object. If no value is provided for *settz,* the parsing algorithm does not search for a time zone. The parser recognizes most valid input and always parses relative to the time zone. Fields not specified are defaulted where appropriate. Illegal input results in an **Error** exception being raised.

inline void set_country (**country** *c*);
Sets the country to the value *c*.

void set_gm_time ();
Sets the date and time to Greenwich mean time.

void set_local_time ();
Sets the date and time to local time as determined by the time zone and country code values.

inline void set_time_zone (**time_zone** *tz*);
Sets the time zone to the value *tz*.

void start_day (**int** *n* = 1);
Advances the time the specified number of days, setting the time to 00:00:00. The default is one.

void start_hour (**int** *n* = 1);
Advances the time by the specified number of hours, setting the time to hh:00:00. The default is one.

void start_min (**int** *n* = 1);
Advances the time by the specified number of minutes, setting the time to hh:mm:00. The default is one.

void start_month (**int** *n* = 1);
Advances the time by the specified number of months, setting the time to 01/mm/yyyy 00:00:00. The default is one.

void start_week (**int** *n* = 1);
Advances the time by the specified number of weeks, setting the time to Monday 00:00:00. The default is one.

void start_year (**int** *n* = 1);
Advances the time by the specified number of years, setting the time to 01/01/yyyy 00:00:00. The default is one.

Friend Functions:

friend istream operator>> (**istream&** *is*, **Date_Time&** *dt*);
Overloads the input operator to read the input stream *is*, and parses the character string containing the date and time information. The result is returned in the date and time object *dt*.

friend ostream operator<< (**ostream&** *os*, **const Date_Time*** *dt*);
Overloads the output operator for a pointer to a date-and-time object. The object writes the output stream *os* with a character string representing the date and time object *dt* formatted for the appropriate country.

friend ostream operator<< (**ostream&** *os*, **const Date_Time&** *dt*);
Overloads the output operator for a reference to a date-and-time object. The object writes the output stream *os* with a character string representing the date and time object *dt* formatted for the appropriate country.

inline friend void set_default_country (**country** *c*);
Sets the default country for the class to the value *c*.

inline friend void set_default_time_zone (**time_zone** *tz*);
Sets the default time zone for the class to the value *tz*.

Time_zone.h File

4.4 The `time_zone.h` include file contains enumeration declarations for time zone names of type **time_zone**. The file declares a static **char*** array of printable names. The constants in the enumerated type can be used as indexes for these names. In the following table, the enum declaration is on the left and the matching static **char*** string is on the right.

Name: `time_zone.h` — Symbolic and string time zone names

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

Enumeration Declaration	*Character String*
UNKNOWN_TIME_ZONE	"Unknown Time Zone"
US_EASTERN	"US/Eastern"
US_CENTRAL	"US/Central"
US_MOUNTAIN	"US/Mountain"
US_PACIFIC	"US/Pacific"
US_PACIFIC_NEW	"US/Pacific–New"
US_YUKON	"US/Yukon"
US_EAST_INDIANA	"US/East–Indiana"
US_ARIZONA	"US/Arizona"
US_HAWAII	"US/Hawaii"
CANADA_NEWFOUNDLAND	"Canada/Newfoundland"
CANADA_ATLANTIC	"Canada/Atlantic"
CANADA_EASTERN	"Canada/Eastern"
CANADA_CENTRAL	"Canada/Central"
CANADA_EAST_SASKATCHEWAN	"Canada/East–Saskatchewan"
CANADA_MOUNTAIN	"Canada/Mountain"
CANADA_PACIFIC	"Canada/Pacific"
CANADA_YUKON	"Canada/Yukon"
GB_EIRE	"GB–Eire"
WET	"WET"
ICELAND	"Iceland"
MET	"MET"
POLAND	"Poland"
EET	"EET"
TURKEY	"Turkey"
W_SU	"W–SU"
PRC	"PRC"
KOREA	"Korea"
JAPAN	"Japan"
SINGAPORE	"Singapore"
HONGKONG	"Hongkong"
ROC	"ROC"
AUSTRALIA_TASMANIA	"Australia/Tasmania"
AUSTRALIA_QUEENSLAND	"Australia/Queensland"
AUSTRALIA_NORTH	"Australia/North"
AUSTRALIA_WEST	"Australia/West"
AUSTRALIA_SOUTH	"Australia/South"
AUSTRALIA_VICTORIA	"Australia/Victoria"
AUSTRALIA_NSW	"Australia/NSW"
NZ	"NZ"

Country.h File

4.5 The `country.h` include file contains enumeration declarations for country names of type **country**. The file declares a static **char*** array of printable country names. The constants in the enumerated type can be used as indexes for these names. In the following table, the enumeration declaration is on the left and the static **char*** string is on the right.

Name: `country.h` — Symbolic and string country names

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

Enumeration Declaration	*Character String*
UNKNOWN_COUNTRY	"Unknown Country"
UNITED_STATES	"United States"
FRENCH_CANADIAN	"French Canadian"
LATIN_AMERICA	"Latin America"
NETHERLANDS	"Netherlands"
BELGIUM	"Belgium"
FRANCE	"France"
SPAIN	"Spain"
ITALY	"Italy"
SWITZERLAND	"Switzerland"
UNITED_KINGDOM	"United Kingdom"
DENMARK	"Denmark"
SWEDEN	"Sweden"
NORWAY	"Norway"
GERMANY	"Germany"
PORTUGAL	"Portugal"
FINLAND	"Finland"
ARABIC_COUNTRIES	"Arabic Countries"
ISRAEL	"Israel"

Calendar.h File

4.6 The `calendar.h` include file contains enumeration declarations for day and month names of the types **day_of_week** and **months**. The file declares two static **char*** arrays of printable day and month names. The constants in the enumerated types can be used as indexes for these names. In addition, an array indexed by type **month** specifying the number of days in the month is also provided. Finally, the file defines several macros for typical date and time constants, along with a macro determining if a year is a leap year. In the following tables, the enum declaration is on the left and the static **char*** string is on the right.

Name: `calendar.h` — Symbolic and string calendar names

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

Enumeration Declaration *Character String*

SUNDAY	“Sunday”
MONDAY	“Monday”
TUESDAY	“Tuesday”
WEDNESDAY	“Wednesday”
THURSDAY	“Thursday”
FRIDAY	“Friday”
SATURDAY	“Saturday”

Enumeration Declaration	*Character String*
JANUARY	“January”
FEBRUARY	“February”
MARCH	“March”
APRIL	“April”
MAY	“May”
JUNE	“June”
JULY	“July”
AUGUST	“August”
SEPTEMBER	“September”
OCTOBER	“October”
NOVEMBER	“November”
DECEMBER	“December”

Date_Time Example

4.7 The following program creates two **Date_Time** objects and initializes one to the current system date and time and the other to the date and time specified in a character string. Several conversions between country formats and time zones are performed, along with manipulating one of the dates by subtracting three months. Finally, the length of time between the two objects is displayed.

```
1    #include <COOL/Date_Time.h>                 // Include Date_Time class

2    int main (void) {
3     set_default_country (UNITED_STATES);       // Set default country code
4     set_default_time_zone (US_CENTRAL);        // Set default time zone
5     Date_Time d1;                              // Create Date_Time object
6     d1.set_local_time ();                      // Set current system time
7     cout << "Local date/time is: " << d1 << "\n"; // Output date in US format
8     d1.set_country (UNITED_KINGDOM);           // Set country to UK
9     d1.set_time_zone (GB_EIRE);                // Set Greenwich Mean Time
10    cout << "GMT date/time is: " << d1 << "\n";   // Output date/time at GMT
11    d1.parse("1 April 1890, 4:30pm");          // Parse some date in UK format
12    cout << "Date/time parsed is: " << d1 << "\n"; // Output date/time parsed
13    d1.set_country (FRANCE);                   // Set country to France
14    d1.set_time_zone (WET);                    // Western European Time zone
15    cout << "Date/time in France: "<< d1 << "\n"; // Output date/time in France
16    Date_Time d2;                              // Create another object
17    d2.set_local_time ();                      // Set current system time
18    cout << "Date/time set is: " << d2 << "\n";   // Output date in US format
19    d2.decr_month (3);                         // Move back three months
20    cout << "Date/time three months earlier: " << d2 << "\n"; // Output date
21    cout << "Duration between dates is ";      //
22    cout << d1.ascii_duration (d2) << "\n";    // Output time duration
23    return 0;                                  // Return valid success code
24   }
```

Line 1 includes the COOL **Date_Time** class header file. Lines 3 and 4 establish the default country and time zone for all **Date_Time** objects in this application to be `UNITED_STATES` and `US_CENTRAL`, respectively. Line 5 instantiates an uninitialized object, line 6 sets its value to be the local system date and time, and line 7 outputs this value. Lines 8 and 9 change the country to `UNITED_KINGDOM` and the time zone to `GB_EIRE` (Greenwich Mean Time). Line 10 outputs the time zone corrected date and time values in English format. Line 11 sets the new value of the **Date_Time** object by parsing a character string, and line 12 outputs the new setting. Lines 13 and 14 change the country to `FRANCE` and the time zone to `WET` and output the value again. Note that the time zone didn't affect the value printed, but the format based on the country code changed. Lines 16 through 18 output another **Date_Time** object for the `UNITED_STATES` in `US_MOUNTAIN` time zone, and sets its value to the current system time. Line 19 decrements the date by three months, and line 20 shows the resulting value. Lines 21 and 22 output in ASCII format the time difference between the two objects. Finally, line 23 exits the program with a valid successful completion code.

The following shows the output of the program:

```
Local date/time is: United States 02-13-1990 11:28:40 US/Central
GMT date/time is: United Kingdom 13-02-1990 17:28:40 GB-Eire
Date/time parsed is: United Kingdom 01-04-1890 16:30:00 GB-Eire
Date/time in France: France 01-01/1990 16:30:00 WET
Date/time set is: United States 02-13-1990 10:28:40 US/Mountain
Date/time three months earlier: United States 11-15-1989 10:28:40 US/Mountain
Duration between dates is 99 years, 35 weeks, 2 days, 0 hours, 58 minutes, 40 seconds
```

Timer Class

4.8 The **Timer** class is publicly derived from the **Generic** class and provides an interface to system timing. It allows a C++ program to record the time between a reference point (mark) and now. This class uses the system **time**(2) interface to provide time resolution at either millisecond or microsecond granularity, depending upon operating system support and features. Since the time duration is stored in a 32-bit word, the maximum time period before rollover occurs is about 71 minutes.

Due to operating system dependencies, the accuracy of all member function results may not be as documented. For example, some operating systems do not support timers with microsecond resolution. In those cases, the values returned are provided to the nearest millisecond or other unit of time as appropriate. See the `Timer.h` header file for system-specific notes.

Name: **Timer** — A timing facility for C++

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

Base Classes: **Generic**

Friend Classes: None

Constructors: **Timer** ();
Creates an instance of the **Timer** class with the mark set to creation time.

Member Functions:

long all();
Returns the number of milliseconds spent in the user process and the operating system since the last reference point (mark).

long all_usec();
Returns the number of microseconds spent in the user process and the operating system since the last reference point (mark).

void mark ();
Sets the reference time to now.

long real();
Returns the number of milliseconds of wall clock time since the last reference point (mark).

long system();
Returns the number of milliseconds spent in the operating system since the last reference point (mark).

long system_usec();
Returns the number of microseconds spent in the operating system since the last reference point (mark).

long user();
Returns the number of milliseconds spent in the user process since the last reference point (mark).

long user_usec();
Returns the number of microseconds spent in the user process since the last reference point (mark).

Timer Example

4.9 The following program uses the COOL **Timer** class to calculate the time for a loop to sum up a sequence of integer values. Note that although this example reports results in milliseconds, support timing granularity on your particular computer and operating system may be different.

```
1     #include <COOL/Timer.h>                        // Includes COOL timer class

2     int main (void) {
3      Timer t1;                                     // Create a timer object
4      t1.mark ();                                   // Set start reference point
5      for (int i = 0, j = 0; i < 10000; i++)        // Loop for 10000 times and
6       j = j + i;                                   // Sum up numbers
7      cout << ”Summation of integers from 0 through 10000 took ”;
8      cout << t1.real () << ” milliseconds\n”;      // Output time since mark
9      return 0;                                     // Return valid completion code
10    }
```

Line 1 includes the COOL **Timer** header file. Line 3 creates a new timer object and line 4 establishes the starting point of the timing operation by setting the mark. Lines 5 and 6 implement a loop counting from 1 to 10000 that calculates the sum of these values. Line 8 contains an embedded call to the timer object to report the elapsed time from the mark to now. Note that since this call is embedded in the output statements, the time reported is not technically correct. A more accurate reading could be established by calling this function and saving the value in a temporary variable for later use in the output statement. Finally, line 9 returns a successful completion code.

The following shows the output of the program:

```
Summation of integers from 0 through 10000 took 20 milliseconds
```

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.

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

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.

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.

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 ...
```

```
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.

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.

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.

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.

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.

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.

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.

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.

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.

ORDERED SEQUENCE CLASSES

6

Introduction

6.1 The ordered sequence classes are a collection of basic data structures that implement sequential access data structures as parameterized classes, thus allowing the user to customize a generic template to create a specific user-defined class. The following classes are discussed in this section:

- **Vector**<*Type*>
- **Stack**<*Type*>
- **Queue**<*Type*>
- **Matrix**<*Type*>

The **Vector**<*Type*> class implements dynamic, one-dimensional vectors supporting such functions as insert, delete, replace, search, reverse, print, and sort. The **Stack**<*Type*> class implements dynamic stacks with the functions push, pop, find, position, and empty. The **Queue**<*Type*> class implements a dynamic, circular buffer queue with support for get, unget, put, and unput to access elements at either end of the queue. The **Matrix**<*Type*> class implements static-sized, two-dimensional matrices with support for the basic arithmetic operations. The **Vector**<*Type*> and **Queue**<*Type*> classes support the notion of a current position. See Section 5, Parameterized Templates, for more information regarding the current position mechanism and the **Iterator**<*Type*> class.

In order to achieve successful compilation and usage, certain operations must be supported by any user-specified type over which a sequence class is parameterized. The member functions **operator=**, **operator<**, **operator>**, **operator==**, and **operator<<** for both pointer and reference must be overloaded for any class object used as the type. In addition, the **Matrix**<*Type*> class requires the supplied type to support **operator+**, **operator–**, **operator/**, and **operator***. Note that built-in types already have these functions defined.

NOTE: The ordered sequence classes use **operator=** of the parameterized type when copying elements. You should be careful when parameterizing an ordered sequence class over a pointer to a type, since the default pointer assignment operator usually copies the pointer, not the value pointed at.

Requirements

6.2 This section discusses the parameterized ordered sequence container classes. It assumes that you have read and understood Section 5, Parameterized Templates. In addition, no attempt is made to discuss the concepts and algorithms for the data structures discussed. You should refer to a general data structures or computer science text for this information.

Vector Class

6.3 The **Vector**<*Type*> class implements one-dimensional vectors of a user-specified type. All memory management and initialization is encapsulated and performed by the class constructors and member functions. Vector objects can be either static-sized or dynamic. Vectors are, by default, dynamic in nature. A static-sized vector object is selected by setting the growth allocation size to zero or by passing in a pointer to a block of user-supplied storage to the constructor. If a vector is of static size and an operation is performed that requires more storage, an **Error** exception is raised.

The **Vector**<*Type*> class implements the notion of a current position. This is useful for iterating through the elements of a vector. The current position is maintained in a data member of type **Vector_state** and is set or reset by all member functions affecting elements in the class. Member functions are provided to reset the current position, move to the next and previous elements, find an element, and get the value at the current position. The **Iterator**<*Type*> class provides a mechanism to save and restore the state associated with the current position, thus allowing the programmer to use multiple iterators over the same instance of a vector.

The **Vector**<*Type*> class follows conventional object-oriented programming techniques and encapsulates the actual data elements from the user. The advantage of this approach is that the class can automatically manage memory, maintain the element count, and be aware of any changes made to the vector. The user of the class facilitates this operation by using the **insert**, **push**, **pop**, and **remove** member functions and their variants. However, the **Vector**<*Type*> class also overloads the **operator[]** and allows the user to access a specific element directly. This is done partly for efficiency and partly for compatibility with past usage.

The drawback of this approach, however, is that the object may not always know its current state. For example, a newly declared vector object has no elements. Each use of **push** to add an element will increment the element count by one. However, elements added at random locations via the **operator[]** will not be counted. A user may get unexpected results by mixing these approaches. For this reason, the **set_length**() member function allows the user to manually set the element count before use of **operator[]** for random-access write operations.

Name:	**Vector**<*Type*> — A dynamic, parameterized vector class
Synopsis:	**#include** <COOL/Vector.h>
Base Classes:	**Vector, Generic**
Friend Classes:	None

Constructors:

Vector<*Type*> ();
Creates an empty vector of the specified type.

Vector<*Type*> (**unsigned long** *number*);
Allocates enough storage for a vector of a specific type to hold *number* elements. Elements are not initialized.

Vector<*Type*> (**unsigned long** *number*, **const** *Type& value*);
Allocates enough storage for a vector of a specific type to hold *number* elements, each of which is initialized with *value*.

Vector<*Type*> (**unsigned long** *number*, **int** *init_num*, ...);
Allocates enough storage for a vector of a specific type to hold *number* elements. The second argument *init_num* specifies the number of optional initialization values provided for consecutive elements of the vector. Any remaining elements are not initialized.

Vector<*Type*> (**Vector**<*Type*>*& vec*);
Duplicates the size and value of another vector object *vec*. Element values are copied by **operator=** for the type specified.

Vector<*Type*> (**void*** *storeage*, **unsigned long** *number*);
Creates a static-sized vector object for *number* elements whose storage *storeage* is provided by the user. If a vector object created in this manner attempts to grow dynamically or the resize member function is invoked, an **Error** exception is raised.

Member Functions:

Boolean append (**const Vector**<*Type*>& *vec*);
Adds the elements of vector *vec* to the end of a vector object. The current position in the vector object is set to the position of the last element of *vec*. If required and not prohibited, this function grows the destination vector and returns **TRUE**; otherwise, this function returns **FALSE**.

inline long capacity () **const**;
Returns the maximum number of elements the vector can contain without growing.

void clear ()
Removes all elements in the object and invalidates the current position.

void copy (**const Vector**<*Type*>& *vec*, **unsigned long** *start* = 0,
long *end* = –1);
Copies the specified range (*start* inclusive and *end* exclusive) from the source vector *vec* to the vector object. The destination vector will grow if necessary and if allowed. The current position is set to the last element copied into the destination. If *end* is equal to minus one (the default), all elements from *start* to the end of the vector are copied. If the *start* or *end* or both indexes are invalid or the vector needs to grow dynamically and this is prohibited, an **Error** exception is raised.

inline Vector_state& current_position ();
Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of vector.

void fill (**const** *Type& value*, **unsigned long** *start* = 0, **long** *end* = –1);
Sets all elements within the specified range (*start* inclusive and *end* exclusive) to *value* and invalidates the current position. If *end* is equal to minus one (the default), all elements from *start* to the end of the vector are filled. If the *start* or *end* or both indexes are invalid, an **Error** exception is raised.

Boolean find (**const** *Type& value*, **unsigned long** *start* = 0);
Searches the vector for *value* beginning at the specified *start* index. If the element is found, this function sets the current position and returns **TRUE**; otherwise, this function invalidates the current position and returns **FALSE**.

inline *Type&* **get** (**int** *n*)
Returns a reference to the *n*th (zero-relative) element in the object. This function sets the current position to this element. If the index is negative or out of range, an **Error** exception is raised.

Boolean insert_after (**const** *Type& value*);

Inserts (not replaces) the element *value* after the current position and does not change the current position. If the current position is invalid, this function returns **FALSE**. If required and not prohibited, this function grows the target vector and returns **TRUE**; otherwise, this function returns **FALSE**.

Boolean insert_after (**const** *Type& value*, *long index*);

Inserts (not replaces) the element *value* after the specified zero-relative *index* and updates the current position to the specified index. If the index is out of range, this function returns **FALSE**. If required and not prohibited, this function grows the target vector and returns **TRUE**; otherwise, this function returns **FALSE**.

Boolean insert_before (**const** *Type& value*);

Inserts (not replaces) the element *value* before the current position and advances the current position one element, thus leaving it pointing at the same element. If the current position is invalid**,** this function returns **FALSE**. If required and not prohibited, this function grows the target vector and returns **TRUE**; otherwise, this function returns **FALSE**.

Boolean insert_before (**const** *Type& value*, *long index*);

Inserts (not replaces) the element *value* before the specified zero-relative *index* and updates the current position to the specified index. If the index is out of range, this function returns **FALSE**. If required and not prohibited, this function grows the target vector and returns **TRUE**; otherwise, this function returns **FALSE**.

inline Boolean is_empty ();

Returns **TRUE** if the vector contains no entries; otherwise, returns **FALSE**.

inline long length () **const**;

Returns the number of elements in the vector.

void merge (**const Vector**<*Type*>& *vec*, *Predicate p*);

Merges the elements of one vector into another by using the supplied predicate for determining the collating sequence. The current position in the destination vector is invalidated. *Predicate* is a user-defined function of type **int** (**Function*) (**const** *Type&*, **const** *Type&*) that returns –1 if the first argument should precede the second, zero if they are equal, and 1 if the first argument should follow the second.

inline Boolean next ();

Advances the current position to the next element in the vector and returns **TRUE**. If the current position is invalid, this function sets the current position to the first element and returns **TRUE**. If the current position is the last element of the vector, this function invalidates the current position and returns **FALSE**.

Vector<*Type*>& **operator=** (**const** *Type& value*);

Overloads the assignment operator for the **Vector**<*Type*> class and assigns all elements *value*. If dynamic growth of the vector is prohibited, an **Error** exception is raised. If there is enough room, the current position is invalidated and a reference to the vector is returned.

Vector<*Type*>& **operator=** (**const Vector**<*Type*>& *vec*);

Overloads the assignment operator for the **Vector**<*Type*> class and assigns *vec* to the vector object, duplicating the size and element values. The current position in the destination vector is invalidated. A reference to the vector object is returned.

Boolean operator== (**const Vector**<*Type*>*& vec*) **const**;

Overloads the equality operator for the **Vector**<*Type*> class and returns **TRUE** if the vector object has the same number of elements with the same values as *vec*; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Vector**<*Type*>*& vec*) **const**;

Overloads the inequality operator for the **Vector**<*Type*> class and returns **TRUE** if the vector object does not have the same number of elements or the same values as *vec*; otherwise, this function returns **FALSE**.

inline Type& operator[] (**unsigned long** *index*) **const**;

Overloads the brackets operator for the **Vector**<*Type*> class and returns a reference to an individual element from the vector at the zero-relative *index* specified. If *index* is invalid or out of range, an **Error** exception is raised. You should be careful when using **operator[]** because an index is out of range if it is greater than the number of elements in the vector. If random access to all allocated space is desired, first use the **set_length()** function.

Type& pop ();

Returns a reference to the last element in the vector and invalidates the current position.

inline long position () **const**;

Returns the current position as a zero-relative index into the vector that can be used with the overloaded **operator[]**.

inline long position (**const** *Type& value*) **const**;

Searches the vector for *value*. If the element is found, this function updates the current position and returns the zero-relative index of the element; otherwise, this function invalidates the current position and returns –1.

Boolean prepend (**const Vector**<*Type*>*& vec*);

Inserts the elements of one vector *vec* at the beginning of a the vector object. The current position is set to the new position of the first element of the old destination vector. If required and not prohibited, this function grows the destination vector and returns **TRUE**; otherwise, this function returns **FALSE**.

inline Boolean prev ();

Moves the current position pointer to the previous element in the vector and returns **TRUE**. If the current position is invalid, this function moves to the last element and returns **TRUE**. If the current position is the first element in the vector, this function invalidates the current position and returns **FALSE**.

Boolean push (**const** *Type& value*);

Adds *value* to the end of a vector. If required and not prohibited, this function grows the vector object. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. This function sets the current position to point to the new element added.

Boolean push_new (**const** *Type& value*);

Adds *value* to the end of a vector if it is not already in the vector. If required and not prohibited, this function grows the vector object. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. This function sets the current position to point to the new element added.

inline Boolean put (**const** *Type& value*, **long** *n*)

Replaces the *n*th (zero-relative) element in the object with *value*. This function returns **TRUE** if the *n*th element exists; otherwise, this function returns **FALSE**. If the index is negative, an **Error** exception is raised.

Type remove ();

Removes and returns the element at the current position. This function sets the current position to the element immediately following the element removed. If the element is found but at the end of the vector, this function invalidates the current position.

Boolean remove (**const** *Type& value*);

Searches for *value* and, if found, this function removes and sets the current position to the element immediately following the element removed, and then it returns **TRUE**. If *value* is found but at the end of the vector, this function invalidates the current position and returns **TRUE**. If *value* is not found, this function returns **FALSE**.

Boolean remove_duplicates ();

Removes any duplicate elements from the vector and invalidates the current position. This function returns **TRUE** if any elements were removed; otherwise, this function returns **FALSE**.

Boolean replace (**const** *Type& value1*, **const** *Type& value2*);

Replaces the first occurrence of *value2* with *value1*. If *value2* is found, this function returns **TRUE**; otherwise, this function returns **FALSE**.

Boolean replace_all (**const** *Type& value1*, **const** *Type& value2*);

Replaces all occurrences of *value2* with *value1* and sets the current position to the last replaced element. This function returns **TRUE** if any elements were replaced; otherwise, this function returns **FALSE**.

inline void reset ();

Invalidates the current position.

void resize (**unsigned long** *size*);

Resizes the vector for at least *size* number of elements and invalidates the current position. If a growth ratio has been selected and it satisfies the resize request, the vector is grown by this ratio. If the new size is negative, an **Error** exception is raised.

void reverse ();

Reverses the order of elements in a vector and invalidates the current position.

Boolean search (**const Vector**<*Type*>& *vec*, **long** *start*=0, **long** *end*=–1);

Searches within the specified range (*start* inclusive, *end* exclusive) of a vector object for a sequence *vec*. If *end* is equal to minus one (the default), all elements from *start* to the end of the vector are filled. If the sequence is found, this function sets the current position in the destination vector to the start of the matched sequence and returns **TRUE**; otherwise, this function returns **FALSE**.

inline void set_alloc_size (**int** *size*);

Updates the allocation growth size for all instances of the class to be used when the growth ratio is zero. Default allocation growth size is 100 bytes. Setting the allocation growth size to zero results in a static-sized object. If the size specified is negative, an **Error** exception is raised.

inline void set_compare (*Vector_Compare* = **NULL**);

Updates the compare function for this class of vector. *Vector_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the vector is parameterized is used..

inline void set_growth_ratio (**float** *ratio*);

Updates the growth ratio for this instance of a vector to the specified value. When a vector needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

inline long set_length (*long*);

Specifies the number of elements in a vector to allow random access via the overloaded **operator[]** member function. If the number requested is larger than the storage allocated, this function truncates to the largest value that the allocated size will support. This function returns the updated number of elements. If the length is negative, an **Error** exception is raised.

void sort (*Predicate p*);

Sorts the elements of a vector by using the supplied predicate for determining the collating sequence and invalidates the current position. *Predicate* is a user-defined function of type **int** (**Function*) (**const** *Type&*, **const** *Type&*) that returns –1 if the first argument should precede the second, zero if they are equal, and 1 if the first argument should follow the second.

inline Type& value ();

Returns a reference to the element at the current position. If the current position is invalid, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (*ostream& os*, **const Vector**<*Type*>& *vec*);

Overloads the output operator for a reference to a **Vector**<*Type*> object to provide a formatted output capability.

friend ostream& operator<< (*ostream& os*, **const Vector**<*Type*>* *vec*);

Overloads the output operator for a pointer to a **Vector**<*Type*> object to provide a formatted output capability.

Vector Example

6.4 The following program declares a vector of five strings whose contents are initialized with state names. The element values are first printed by using **operator<<**, then sorted in reverse alphabetical order, and finally printed by iterating through the vector by using the current position functions.

```
 1   #include <COOL/String.h>                   // COOL String class
 2   #include <COOL/Vector.h>                   // COOL Vector class

 3   DECLARE Vector<String>;                    // Declare vector of strings
 4   IMPLEMENT Vector<String>;                  // Implement vector of strings

 5   Boolean my_compare (const String& s1, const String& s2) {
 6     return ((s1 <= s2) ? FALSE : TRUE);      // Reverse alphabetize
 7   }

 8   int main (void) {
 9     Vector<String> v1(5);                    // Declare vector of strings
10     v1.push ("Texas");                       // Add "Texas"
11     v1.push ("Alaska");                      // Add "Alaska"
12     v1.push ("New York");                    // Add "New York"
13     v1.push ("Alabama");                     // Add "Alabama"
14     v1.push ("North Dakota");                // Add "North Dakota"
15     cout << v1 << "\n";                      // Output the vector
16     v1.sort (my_compare);                    // Reverse sort the vector
17     for (v1.reset(); v1.next(); )            // For each element
18       cout << v1.value() << "\n";            // Output the value
19     exit (0);                                // Exit with OK status
20     }
```

Lines 1 through 4 define the **Vector** and **String** classes. Lines 5 through 7 declare a simple sort function that reverses the lexical comparison test performed by **operator<=** in the **String** class. Line 9 defines a vector of strings with initial storage for five elements. Lines 10 through 14 push five literal character strings into the vector. Line 15 uses **operator<<** for the **Vector** class to output the element values. Line 16 sorts the vector according to the predicate function provided. Finally, lines 17 and 18 use the current position functions to iterate through the elements, printing each one.

The following shows the output for the program:

```
Texas Alaska New York Alabama North Dakota
Texas
North Dakota
New York
Alaska
Alabama
```

Stack Class

6.5 The **Stack**<*Type*> class implements a conventional first-in, last-out data structure that holds a user-specified data type. All memory management and initialization is encapsulated and performed by the class constructors and member functions. Stack objects can be either static-sized or dynamic. Stacks are, by default, dynamic in nature. A static-sized stack object is selected by setting the growth allocation size to zero or by passing in a pointer to a block of user-supplied storage to the constructor. If a stack is of static size and an operation is performed that requires more storage, an **Error** exception is raised.

Name:	**Stack**<*Type*> — A dynamic, parameterized stack
Synopsis:	**#include** <COOL/Stack.h>
Base Classes:	**Stack**, **Generic**
Friend Classes:	None

Constructors:

Stack<*Type*> ();
Creates an empty stack of the specified type.

Stack<*Type*> (**unsigned long** *number*);
Allocates enough storage for a stack of a specific type to hold *number* of elements.

Stack<*Type*> (**const Stack**<*Type*>& *stk*);
Duplicates the size and value of a stack object *stk*.

Stack<*Type*> (**void*** *storage*, **unsigned long** *number*);
Creates a static-sized stack object for *number* of elements whose storage *storage* is provided by the user. If an object of this type attempts to grow dynamically or the programmer invokes the **resize** member function, an Error exception is raised.

Member Functions:

inline long capacity () **const**;
Returns the maximum number of elements the stack can contain.

inline void clear ();
Sets the number of elements in the stack to zero.

Boolean find (**const** *Type& value*);
Searches the stack for *value*. If *value* is found, this function returns **TRUE**; otherwise, this function returns **FALSE**.

inline Boolean is_empty () **const**;
Returns **TRUE** if the stack has no elements; otherwise, this function returns **FALSE**.

inline long length () **const**;
Returns the number of elements in the stack.

Stack<*Type*>& **operator=** (**const Stack**<*Type*>& *stk*);
Overloads the assignment operator for the **Stack**<*Type*> class and assigns *stk* to the stack object by duplicating the size and element values. If the stack object is prohibited from dynamically growing, an **Error** exception is raised.

Boolean operator== (**const Stack**<*Type*>& *stk*) **const**;
Overloads the equality operator for the **Stack**<*Type*> class. Returns **TRUE** if the stacks have an equal number of elements with the same values; otherwise this function returns **FALSE**.

inline Boolean operator!= (**const Stack**<*Type*>& *stk*) **const**;
Overloads the inequality operator for the **Stack**<*Type*> class. This function returns **TRUE** if the stacks have a unequal number of elements or unequal values; otherwise this function returns **FALSE**.

inline Type& operator[] (**unsigned long** *number*);
Overloads the index operator for the **Stack**<*Type*> class. This function returns a reference to the element of the stack that is *number* of elements from the top of the stack. If *number* is greater than the size of the stack, an **Error** exception is raised.

inline Type& pop ();
Removes and returns a reference to the top element on the stack. If the number of elements (that is, length) has been set to a zero-relative index greater than the size of the stack, an **Error** exception is raised.

Type& popn (**long** *n*);
Pops *n* elements off the stack, returning a reference to the last one popped off the stack. With an argument of zero, this function returns the top item of the stack without removing it. If the number of elements to pop is negative, an **Error** exception is raised.

long position (**const** *Type& value)* **const**;
Searches the stack for *value*. If *value* is found, this function returns the zero-relative index, from the top of the stack, of that element; otherwise, this function returns –1.

inline Boolean push (**const** *Type& value*);
Pushes *value* onto the top of a stack. If required and not prohibited, this function grows the stack object. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. If the stack is prohibited from growing dynamically, an **Error** exception is raised.

Boolean pushn (**const** *Type& value, long n*);
Pushes *n* items onto the top of the stack, all of which have the specified *value*. When *n* is zero, this function replaces the top item on the stack with *value*. If required and not prohibited, this function grows the stack object. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. If the stack is prohibited from growing dynamically, an **Error** exception is raised.

void resize (**long** *number*);
Resizes the stack for at least *number* of elements. If a growth ratio has been selected and it satisfies the resize request, the stack is grown by this ratio. If the stack is prohibited from dynamically growing, an **Error** exception is raised.

inline void set_alloc_size (**int** *size*);
Updates the allocation growth size to be used when the growth ratio is zero. Default allocation growth size is 100 bytes. Setting the allocation growth size to zero results in a static-sized object. If *size* is zero or negative, an **Error** exception is raised.

inline void set_compare (*Stack_Compare* = **NULL**);
Sets the compare function for this class of stack. *Stack_Compare* is a user-defined function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no such function is provided, the **operator==** for the type over which the class is parameterized is used.

inline void set_growth_ratio (float *ratio*);
Updates the growth ratio for this instance of a stack to *ratio*. When a stack needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

inline long set_length (**long** *number*);
Specifies the number of elements in a stack to allow random access via the overloaded **operator[]** member function. If *number* is larger than the storage allocated, this function truncates *number* to the largest value that the allocated size will support. This function returns the new element count. If *number* is negative, an **Error** exception is raised.

inline Type& top ();
Returns (without removing) a reference to the top element of the stack. If the number of elements (that is, length) has been set to a zero-relative index greater than the size of the stack, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (*ostream& os*, **const Stack**<*Type*>& *stk*);
Overloads the output operator for a reference to a **Stack**<*Type*> object to provide a formatted output capability.

inline friend ostream& operator<< (*ostream& os*, **const Stack**<*Type*>* *stk*);
Overloads the output operator for a pointer to a **Stack**<*Type*> object to provide a formatted output capability.

Stack Example

6.6 The following program declares a stack capable of initially holding 10 integers. Integer values are pushed onto the stack in a loop. Notice that since more than 10 elements are pushed onto the stack, it must grow automatically and add storage capacity as necessary to hold the extra elements. Finally, these elements are then popped from the stack and printed.

```
 1   #include <COOL/Stack.h>                      // COOL Stack class

 2   DECLARE Stack<int>;                          // Declare stack of integers
 3   IMPLEMENT Stack<int>;                        // Implement stack of integers

 4   int main (void) {
 5     Stack<int> s1(3);                          // Declare stack of integers
 6     for (int i = 1; i <= 5; i++)               // In a small loop, push "n"
 7       s1.pushn (i,i);                          // copies of an integer value
 8     for (i = 0; i < 5; i++) {                  // In another similar loop up to
 9       for (int j = 0; j < s1.top(); j++)       // the top element value, get
10         cout << s1.pop();                      // a value from stack and print
11      cout << "\n";                             // Output a newline and repeat
12     }
13     exit (0);                                  // Exit with OK status
14   }
```

Lines 1 through 3 define the **Stack** class. Line 5 defines a stack of integers with initial storage for 10 elements. Lines 6 and 7 loop from one through five and push the current loop number on the stack. Thus, the first time through the loop, one element whose value is one is pushed. The second time, two elements whose values are two are pushed, and so on. Because more than 10 elements are added to the stack, an automatic resize is performed by the stack object to accommodate more elements. Because no user-specified growth factor was given, enough storage is allocated to hold 100 elements. Lines 8 through 11 contain nested loops that read the top value on the stack, loop that many times, pop off a value, and print it. After each inner loop completes, a newline character is printed.

The following shows the output from the program:

```
55555
4444
333
22
1
```

Queue Class

6.7 The **Queue**<*Type*> class implements a conventional first-in, first-out data structure that holds a user-specified data type. All memory management and initialization is encapsulated and performed by the class constructors and member functions. Queue objects can be either static-sized or dynamic. Queues are, by default, dynamic in nature. A static-sized queue object is selected by setting the growth allocation size to zero or by passing in a pointer to a block of user-supplied storage to the constructor. If a queue is of static size and an operation is performed that requires more storage, an **Error** exception is raised.

The **Queue**<*Type*> class implements the notion of a current position. This is useful for iterating through the elements of a queue. The current position is maintained in a data member of type **Queue_state** and is set or reset by all member functions affecting elements in the class. Member functions are provided to reset the current position, move to the next and previous elements, find an element, and get the value at the current position. The **Iterator**<*Type*> class provides a mechanism to save and restore the state associated with the current position, thus allowing the programmer to use multiple iterators over the same instance of a queue.

The **Queue**<*Type*> class allows the programmer to add and/or remove items from either end of the queue. In addition, the current position and iterator functions allow the programmer to examine other entries in the middle of the queue and remove or change them. This would be useful in implementing a prioritized queue where the entries may need to be rearranged at times.

Name: **Queue**<*Type*> — A dynamic, parameterized queue

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

Base Classes: **Queue, Generic**

Friend Classes: None

Constructors:

Queue<*Type*> ();
Creates an empty queue of the specified type.

Queue<*Type*> (**unsigned long** *number*);
Allocates enough storage for a queue of a specific type to hold *number* of elements specified by the argument.

Queue<*Type*> (**const Queue**<*Type*>& *q*);
Duplicates the size and value of a queue object *q*.

Queue<*Type*> (**void*** *storage*, **unsigned long** *number*);
Creates a static-sized queue object for *number* of elements whose storage *storage* is provided by the user. If an object of this type attempts to grow dynamically or the programmer invokes the **resize** member function, an Error exception is raised.

Member Functions:

inline long capacity () **const**;
Returns the maximum number of elements the stack can contain.

void clear ();
Sets the number of items in the queue to zero. This function invalidates the current position.

inline Queue_state& current_position ();

Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of queue.

Boolean find (**const** *Type& value*);

Searches the queue for *value*. If *value* is found, this function sets the current position and returns **TRUE**; otherwise, this function resets the current position and returns **FALSE**.

Type& get ();

Removes and returns a reference to the first-in item on the queue. If there are no elements in the queue, an **Error** exception is raised.

inline Boolean is_empty () **const**;

Returns **TRUE** if there are no items in the queue. Otherwise, this function returns **FALSE**.

inline long length () **const**;

Returns the number of elements in the queue.

inline Type& look ();

Returns the first-in item on the queue. If there are no elements in the queue, an **Error** exception is raised.

Boolean next ();

Advances the current position to the next element in the queue and returns **TRUE**. If the current position is invalid, this function sets the current position to the first element and returns **TRUE**. If the current position is the last element of the queue, this function invalidates the current position and returns **FALSE**.

Queue<*Type*>*&* **operator=** (**const Queue**<*Type*>*& q*);

Overloads the assignment operator for the **Queue** class and assigns *q* to the queue object by duplicating the size and element values. This function invalidates the current position.

Boolean operator== (**const Queue**<*Type*>& *q*) **const**;

Overloads the equality operator for the **Queue** class. This function returns **TRUE** if the queues have an equal number of elements with the same values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Queue**<*Type*>& *q*) **const**;

Overloads the inequality operator for the **Queue** class. This function returns **TRUE** if the queues have an unequal number of elements or unequal values.

Boolean prev ();

Moves the current position pointer to the previous element in the queue and returns **TRUE**. If the current position is invalid, this function moves to the last element and returns **TRUE**. If the current position is the first element in the queue, this function invalidates the current position and returns **FALSE**.

Boolean put (**const** *Type& value*);

Puts *value* onto the back of the queue, making it the last-in item. If required and not prohibited, this function grows the queue, puts the new last-in item on the queue, and returns **TRUE**. Otherwise, this function returns **FALSE**.

Boolean remove ();

Removes the element at the current position. This function returns **FALSE** if the current position is invalid; otherwise, this function sets the current position to the element immediately following the element removed (if not at end of queue) and returns **TRUE**. If the current position is at the last element before removing, this function invalidates the current position and returns **TRUE** after removing the element.

Boolean remove (**const** *Type& value*);

Searches for *value* and, if found, this function removes and sets the current position to the element immediately following the element removed, and then it returns **TRUE**. If *value* is found but at the end of the queue, this function invalidates the current position and returns **TRUE**. If the element is not found, this function returns **FALSE**.

inline void reset ();

Invalidates the current position.

void resize (**long** *number*);

Resizes the queue for at least *number* of elements. If a growth ratio has been selected and it satisfies the resize request, the queue is grown by this ratio, the current position is invalidated, and **TRUE** is returned. Otherwise, this function returns **FALSE**. If the size specified is zero or negative, an **Error** exception is raised.

inline void set_alloc_size (**int** *size*);

Updates the allocation growth size to be used when the growth ratio is zero. Default allocation growth size is 100 bytes. If the size specified is negative, an **Error** exception is raised.

inline void set_compare (*Queue_Compare* = **NULL**);

Updates the compare function for this class of queue. *Queue_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the class is parameterized is used.

inline void set_growth_ratio (**float** *ratio*);

Updates the growth ratio for this instance of a queue to *ratio*. When a queue needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

Boolean unget (**const** *Type& value*);

Puts *value* onto the front of the queue. If required and not prohibited, this function grows the queue, puts the first-in item on the queue, and returns **TRUE**. Otherwise, this function returns **FALSE**. If there are no elements in the queue, an **Error** exception is raised.

Type& unput ();

Removes and returns a reference to the last-in item on the queue.

inline Type& value ();

Returns a reference to the element at the current position. If the current position is invalid, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (*ostream& os*, **const Queue**<*Type*>& *q*);

Overloads the output operator for a reference to a **Queue**<*Type*> object to provide a formatted output capability.

inline friend ostream& operator<< (*ostream& os*, **const Queue**<*Type*>* *q*);
Overloads the output operator for a pointer to a **Queue**<*Type*> object to provide a formatted output capability.

Queue Example

6.8 The following program declares a queue of doubles. Random floating-point values are added to the queue in a loop. The elements added are then output. Next, a loop iterates through the elements of the queue by using the current position functionality. If any random number added to the queue is below some arbitrary tolerance, it is removed. Finally, the remaining elements are printed.

```
 1   #include <COOL/Queue.h>                          // COOL Queue class
 2   #include <COOL/Random.h>                         // COOL Random number class

 3   DECLARE Queue<double>;                           // Declare a queue of doubles
 4   IMPLEMENT Queue<double>;                         // Implement a queue of doubles

 5   int main (void) {
 6     Queue<double> q1;                              // Create empty queue
 7     Random r (SIMPLE, 1, 3.0, 9.0);                // Simple random generator
 8     for (int i = 0; i < 5; i++)                    // Put five random numbers
 9               q1.put (r.next ());                  // into the queue
10     cout << q1;                                    // Output queue elements
11     for (q1.reset(); q1.next(); )                  // For each element in queue
12               if (q1.value() < 4.5)                // If less than tolerance
13                         q1.remove ();              // Remove from queue
14     cout << "\n" << q1;                            // Output queue elements
15     exit (0);                                      // Exit with OK status
16   }
```

Lines 1 through 4 define the `Queue<double>` class, and line 6 declares an instance of this class. Line 7 declares a random number generator whose values are guaranteed to be within the range 3.0 to 9.0 inclusive (see Section 3, Number Classes, for a discussion about the **Random** class). Lines 8 and 9 contain a simple loop that adds five random numbers to the queue. Line 10 uses **operator<<** for the **Queue** class to output the element values. Lines 11 through 13 use the current position functions to iterate through the elements, removing any entry below an arbitrary tolerance. Finally, the remaining elements are output.

The following shows the output from a sample run of the program:

```
<First in> 6.08322 4.05445 4.85191 6.2072 8.68577 <Last in>
<First in> 6.08322 4.85191 6.2072 8.68577 <Last in>
```

Matrix Class

6.9 The **Matrix**<*Type*> class implements two-dimensional arithmetic matrices for a user-specified numeric data type. Using the parameterized types facility of C++, it is possible, for example, for the user to create a matrix of rational numbers by parameterizing the **Matrix** class over the **Rational** class (see Section 3, Number Classes, for a discussion regarding the **Rational** class). The only requirement for the type is that it support the basic arithmetic operators. Note that unlike the other sequence classes, the **Matrix**<*Type*> class is fixed-size only (that is, it will not grow once the size has been specified to the constructor).

Name:	**Matrix**<*Type*> — A parameterized matrix class
Synopsis:	**#include** <COOL/Matrix.h>
Base Classes:	**Matrix, Generic**
Friend Classes:	None

Constructors:

Matrix<*Type*> (**unsigned int** *row*, **unsigned int** *col*);
Allocates enough storage for a matrix of a specific type with the specified number of rows and columns.

Matrix<*Type*> (**unsigned int** *row*, **unsigned int** *col*, **int** *init_num*, ...);
Allocates enough storage for a matrix of the specified type and size. The third argument *init_num* indicates the number of optional initialization values. Matrix elements are initialized in row-major order.

Matrix<*Type*> (**unsigned int** *row*, **unsigned int** *col*, **const** *Type& value*);
Allocates enough storage for a matrix of a specific type with the specified number of rows and columns. In addition, each element of the matrix is initialized to *value*.

Matrix<*Type*> (**const Matrix**<*Type*>& *m*);
Duplicates the size and value of a **Matrix**<*Type*> object *m*.

Member Functions:

inline int columns () **const**;
Returns the number of columns in the matrix.

void fill (**const** *Type& value*);
Sets all elements in the matrix to *value*.

inline Type get (**unsigned int** *row*, **unsigned int** *col*) **const**;
Returns the value of the element at the indicated row and column. If the row or column specification is out of range, an **Error** exception is raised.

Matrix<*Type*> **operator+** (**const Matrix**<*Type*>& *m*) **const**;
Overloads the addition operator to provide matrix addition for the **Matrix**<*Type*> class. A new matrix is returned as the result. If the matrices are of a different size, an **Error** exception is raised.

Matrix<*Type*> **operator+** (**const** *Type& value*) **const**;
Overloads the addition operator to provide scalar addition for the **Matrix**<*Type*> class. A new matrix is returned as the result.

Matrix<*Type*> **operator*** (**const Matrix**<*Type*>& *m*) **const**;
Overloads the multiplication operator to provide matrix multiplication for the **Matrix**<*Type*> class. A new matrix is returned as the result. If the matrices are of a different size, an **Error** exception is raised.

Matrix<*Type*> **operator*** (**const** *Type& value*) **const**;
Overloads the multiplication operator to provide scalar multiplication for the **Matrix**<*Type*> class. A new matrix is returned as the result.

Matrix<*Type*>& **operator=** (**const** *Type& value*);
Overloads the assignment operator for the **Matrix**<*Type*> class and assigns all elements of a matrix to *value*.

Matrix<*Type*>& **operator=** (**const Matrix**<*Type*>& *m*);
Overloads the assignment operator for the **Matrix**<*Type*> class and assigns one **Matrix**<*Type*> object to have the value of another by duplicating the size and element values.

Matrix<*Type*>& **operator+=** (**const Matrix**<*Type*>& *m*);
Overloads the addition–with–assignment operator to provide matrix addition for the **Matrix**<*Type*> class. The source is modified to contain the result. If the matrices are of a different size, an **Error** exception is raised.

Matrix<*Type*>& **operator+=** (**const** *Type& value*);
Overloads the addition-with-assignment operator to provide scalar addition for the **Matrix**<*Type*> class. The source is modified to contain the result. If the matrices are of a different size, an **Error** exception is raised.

Matrix<*Type*>& **operator*=** (**const Matrix**<*Type*>& *m*);
Overloads the multiplication-with-assignment operator to provide matrix multiplication for the **Matrix**<*Type*> class. The source is modified to contain the result. If the matrices are of a different size, an **Error** exception is raised.

Matrix<*Type*>& **operator*=** (**const** *Type& value*);
Overloads the multiplication-with-assignment operator to provide scalar multiplication for the **Matrix**<*Type*> class. The source is modified to contain the result

Boolean operator== (**const Matrix**<*Type*>& *m*) **const**;
Overloads the equality operator for the **Matrix**<*Type*> class. This function returns **TRUE** if the matrices have the same number of elements with the same values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Matrix**<*Type*>& *m*) **const**;
Overloads the inequality operator for the **Matrix**<*Type*> class. This function returns **TRUE** if the matrices have a different number of elements or different values.

inline void put (**unsigned int** *row*, **unsigned int** *col*, *Type value*);
Assigns *value* to the element at the specified *row* and *col*. If the row or column specification is out of range, an **Error** exception is raised.

inline int rows () **const**;
Returns the number of rows in the matrix.

inline void set_compare (*Matrix_Compare* = **NULL**);
Updates the compare function for this class of matrix. *Matrix_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the class is parameterized is used.

Friend Functions:

friend ostream& operator<< (**ostream&** *os*, **const Matrix**<*Type*>& *m*);
Overloads the output operator for a reference to a **Matrix**<*Type*> object *m* to provide a formatted output capability.

inline friend ostream& operator<< (**ostream&** *os*, **const Matrix**<*Type*>* *m*);
Overloads the output operator for a pointer to a **Matrix**<*Type*> object *m* to provide a formatted output capability.

Matrix Example

6.10 The following program declares two matrices of integers. The first matrix is filled with a series of integral values computed in the nested loops, and the second matrix is derived from the first. Several of the **Matrix**<*Type*> overloaded operators are used, and the resulting matrices are printed.

```
 1   #include <COOL/Matrix.h>                      // COOL Matrix class

 2   DECLARE Matrix<int>                           // Declare a matrix of integers
 3   IMPLEMENT Matrix<int>                         // Implement matrix of integers

 4   int main (void) {
 5     Matrix<int> mc1(3,4), mc2(3,4);             // Two 3x4 matrices of integers
 6     for (int i = 0; i < 3; i++)                 // For each row in matrix
 7       for (int j = 0; j < 4; j++)               // For each column in matrix
 8         mc1.put(i,j,(i+2)*(j+3));               // Assign element value
 9     mc2 = mc1 + 5;                              // Copy matrix with added value
10     mc1 = mc1 + mc2;                            // Add the matrices together
11     cout << mc1 << ”\n” << mc2 << ”\n”;         // Output the starting matrices
12     exit (0);                                   // Exit with OK status
13   }
```

Line 1 includes the COOL `Matrix.h` class header file. Lines 2 and 3 declare and implement the `Matrix<int>` class. Line 5 declares two `Matrix<int>` variables, each of which have three rows and four columns. Lines 6 through 8 generate a series of integral values that are copied into the elements of the first matrix. Line 9 uses the overloaded addition and assignment operators for the **Matrix**<*Type*> class and computes the value of the second matrix. Line 10 uses the overloaded addition operator to add the two matrices together. Line 11 uses the overloaded output operator to display the contents of each matrix. Finally, the program ends with a valid exit code on line 12.

The following shows the output from the program:

```
17 21 25 29
23 29 35 41
29 37 45 53

11 13 15 17
14 17 20 23
17 21 25 29
```

UNORDERED SEQUENCE CLASSES

7

Introduction

7.1 The unordered sequence classes are a collection of basic data structures that implement random-access data structures as parameterized classes, thus allowing the user to customize a generic template to create a specific user-defined class. The following classes are discussed in this section:

- **List**<*Type*>
- **Pair**<*T1, T2*>
- **Association**<*Ktype, Vtype*>
- **Hash_Table**<*Ktype,Vtype*>

The **List**<*Type*> class implements dynamic, Common Lisp-style lists supporting such functions as insert, delete, replace, search, reverse, print, and sort. The **Pair**<*T1,T2*> class implements a simple object that contains two other objects, primarily for use in the **Association**<*Ktype, Vtype*> class. The **Association**<*Ktype, Vtype*> class maintains a collection of associated objects. The **Hash_Table**<K*type, Vtype*> class implements dynamic hash tables with the option for user-defined hashing functions. The **List**<*Type*>, **Hash_Table**<*Ktype, Vtype*>, and **Association**<K*type, Vtype*> classes support the notion of a current position. The example programs in this section solve the same problem using different data structures, allowing the reader to compare the different features of each.

In order to achieve successful compilation and usage, certain operations must be supported by any user-specified type over which an unordered sequence class is parameterized. The member functions **operator=**, **operator<**, **operator>**, **operator<<**, and **operator==** must be overloaded for any class object used as the type. Note that built-in types already have these functions defined.

NOTE: The unordered sequence classes use **operator=** of the parameterized type when copying elements. You should be careful when parameterizing an unordered sequence class over a pointer to a type, since the default pointer assignment operator usually copies the pointer, not the value pointed at.

Requirements

7.2 This section discusses the parameterized unordered sequence container classes. It assumes that you have read and understood Section 5, Parameterized Templates. In addition, no attempt is made to discuss the concepts and algorithms for the data structures discussed. You should refer to a general data structures or computer science text for this information.

List Class

7.3 The **List**<*Type*> class implements Common Lisp-style lists that provide a rich collection of member functions for list manipulation and management. A list consists of a collection of nodes, each of which contains a reference count, a pointer to the next node in the list, and a data element of a user-specified type. The **List**<*Type*> class uses reference counting to allow more efficient sharing and reuse of list node objects. The reference count indicates the number of list or node objects pointing to a node and ensures that the node and the data are deallocated when the node is no longer referenced.

Considerable attention has been paid to performance and efficiency concerns in the **List**<*Type*> class. The **Base_List** class implements generic list functionality required by the parameterized **List** class. The **Base_List** class is not usable as a stand-alone class, but used to derive the **List**<*Type*> class. By providing generic operations in a base class, the quantity of code generated for each implementation of a parameterized class is reduced considerably. Consequently, most member functions for **List**<*Type*> are inline calls to the generic equivalent function in the base **List** class.

Name: **List**<*Type*> — A parameterized list

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

Base Classes: **List, Generic**

Friend Classes: None

Constructors:

List<*Type*> ();
Creates an empty list of the specified type.

List<*Type*> (**const** *Type& value*);
Creates a list with one element of the specified type and value.

List<*Type*> (**int** *number*, *Type&*, ...);
Creates a list of *number* elements of the specified type initialized with the optional values provided.

List<*Type*> (**List**<*Type*>& *l*);
Creates a list from *l* of the specified type.

List<*Type*> (**const** *Type& value*, **List**<*Type*>& *l*);
Creates a list of the specified type with *value* as the first element and *l* as the tail.

Member Functions:

Boolean append (**const List**<*Type*>& *l*);
Adds the elements of *l* to the end of the object and returns **TRUE**. This function sets the current position to the first element added. This function returns **FALSE** if a new node cannot be created.

void but_last (**List**<*Type*>& *l*, **int** *n* = 1);
Sets *l* to point to all but the last *n* elements of the object. When no second argument is specified, **but_last** sets *l* to point to all but the last element of the object. A second argument whose value is zero sets *l* to point to all of the elements in the object. This function sets *l* to **NIL** if the second argument is greater than or equal to the number of elements in the object. This function invalidates the current position of *l*.

void clear ();
Removes all elements in the object and invalidates the current position.

void copy (**List**<*Type*>& *l*) **const**;
Sets *l* to a copy of the object. This function invalidates the current position of *l*.

inline List_state& current_position () **const;**
Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of list.

void describe (*ostream& os*);
Provides a formatted output capability for displaying the internal structure of the list including reference counts and values for each node.

void difference (**const List**<*Type*>& *l*);
Removes from the object the elements that also appear in *l*. This function invalidates the current position of the list object.

void exclusive_or (**const List**<*Type*>& *l*);
Performs an exclusive-or operation by setting the object to contain all the elements in the object that are not in *l*, and all the elements in *l* that are not in the object. This function invalidates the current position of the list object.

inline Boolean find (**const** *Type& value*, **List_state** *start* = **NULL**);
Searches the object for *value* beginning at the current position specified. If a starting point is not provided, this function begins the search at the head of the list. If *value* is found, this function sets the current position to this element and returns **TRUE**; otherwise, this function returns **FALSE**. If *value* is not found, this function returns **FALSE** and resets the current position of the list object.

inline Type& get (**int** *n* = 0);
Returns a reference to the *n*th (zero-relative) element in the object. This function sets the current position to this element. If the index is negative or is greater than the number of nodes in the list, an **Error** exception is raised.

inline Boolean insert_after (**const** *Type& value1*, **const** *Type& value2*);
Inserts *value1* after *value2* in the object, sets the current position to this new element, and returns **TRUE**. If *value2* is not in the object, this function returns **FALSE**.

inline Boolean insert_after (**const** *Type& value*);
Inserts *value* after the element at the current position in the object, sets the current position to this new element, and returns **TRUE**. If the current position is invalid, an **Error** exception is raised.

inline Boolean insert_before (**const** *Type& value1*, **const** *Type& value2*);
Inserts *value1* before *value2* in the object, sets the current position to this new element, and returns **TRUE**. If *value2* is not in the object, this function returns **FALSE**.

inline Boolean insert_before (**const** *Type& value*);
Inserts *value* before the element at the current position in the object, sets the current position to this new element, and returns **TRUE**. If the current position is invalid, an **Error** exception is raised.

void intersection (**const List**<*Type*>& *l*);
Sets the object to contain only the elements that exist in both the object and *l*. This function invalidates the current position of the list object.

inline Boolean is_empty ();
Returns **TRUE** if the object does not have any elements; otherwise, this function returns **FALSE**.

void last (**List**<*Type*>& *l*, **int** *n* = 1);
Sets *l* to point to the last *n* elements of the object. When it has no second arguments, **last** sets *l* to point to the last element of the object. If *n* is equal to the number of elements in the object, this function sets *l* to point to all the elements of the object. If *n* is greater than the number of elements in the object or *n* is zero, this function sets *l* to **NIL**, a list with no elements. This function sets the current position to the first of the last *n* elements of the list object.

int length ();
Returns the number of elements in the object.

void lunion (**const List**<*Type*>& *l*);
Sets the object to contain everything that is an element of either the object or *l*. This function invalidates the current position of the list object.

inline Boolean member (**List**<*Type*>& *l*, **const** *Type& value*);
Searches the object for *value*. If the element is found, this function sets the current position to this element, sets *l* to the sublist within the object starting with the desired element, and returns **TRUE**. If the *value* is not found, this function sets *l* to **NIL** and returns **FALSE**.

inline void merge (**const List**<*Type*>& *l*, *List_Predicate p*);
Merges the elements of *l* into the object by using the supplied predicate *p* for determining the collating sequence. *List_Predicate* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*).

inline Boolean next ();
Advances the current position to the next element in the object and returns **TRUE**. If the current position is invalid, this function sets the current position to the first element and returns **TRUE**. If the current position is the last element of the object, this function invalidates the current position and returns **FALSE**.

Boolean next_difference (**const List**<*Type*>& *l*);
Sets the current position to the next element in the difference of the object and *l* and returns **TRUE**. If there are no more elements in the difference, this function invalidates the current position and returns **FALSE**.

Boolean next_exclusive_or (**const List**<*Type*>& *l*);
Sets the current position to the next element in the exclusive-or of the object and *l* and returns **TRUE**. If there are no more elements in the exclusive-or, this function invalidates the current position and returns **FALSE**.

Boolean next_intersection (**const List**<*Type*>& *l*);
Sets the current position to the next element in the intersection of the object and *l* and returns **TRUE**. If there are no more elements in the intersection, this function invalidates the current position and returns **FALSE**.

Boolean next_lunion (**const List**<*Type*>& *l*);
Sets the current position to the next element in the union of the object and *l* and returns **TRUE**. If there are no more elements in the union, this function invalidates the current position and returns **FALSE**.

inline List<*Type*>& **operator+** (**const List**<*Type*>& *l*);
Returns a reference to a new list containing the concatenation of the object and *l*. Since the new list is allocated from heap memory, you must delete its storage.

inline List<*Type*>& **operator–** (**const List**<*Type*>& *l*);
Returns a reference to a new list containing the difference of the object and *l*. Since the new list is allocated from heap memory, you must delete its storage.

inline List<*Type*>& **operator^** (**const List**<*Type*>& *l*);
Returns a reference to a new list containing the exclusive-or of the object and *l*. Since the new list is allocated from heap memory, you must delete its storage.

inline List<*Type*>& **operator&** (**const List**<*Type*>& *l*);
Returns a reference to a new list containing the intersection of the object and *l*. Since the new list is allocated from heap memory, you must delete its storage.

inline List<*Type*>& **operator|** (**const List**<*Type*>& *l*);
Returns a reference to a new list containing the union of the object and *l*. Since the new list is allocated from heap memory, you must delete its storage.

inline List<*Type*>& **operator=** (**List**<*Type*>& *l*);
Assigns the object (the list on the left-hand side of the assignment) to point to the same set of elements as *l* (on the right-hand side of the assignment) and returns a reference to the updated object. Also, this function invalidates the current position.

inline List<*Type*>& **operator+=** (**const List**<*Type*>& *l*);
Sets the object (the list on the left-hand side of the operator) to the concatenation of the object and *l* (the list on the right-hand side of the operator) and returns a reference to the updated object. Also, this function invalidates the current position.

inline List<*Type*>& **operator–=** (**const List**<*Type*>& *l*);
Returns a reference to the modified object containing the difference of the object and *l*.

inline List<*Type*>& **operator^=** (**const List**<*Type*>& *l*);
Returns a reference to the modified object containing the exclusive-or of the object and *l*.

inline List<*Type*>& **operator&=** (**const List**<*Type*>&);
Returns a reference to the modified object containing the intersection of the object and *l*.

inline List<*Type*>& **operator|=** (**const List**<*Type*>&);
Returns a reference to the modified object containing the union of the object and *l*.

Boolean operator== (**const List**<*Type*>& *l*) **const**;
Returns **TRUE** if the elements of the two lists have the same values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const List**<*Type*>& *l*) **const**;
Returns **TRUE** if the elements of the two lists have different values; otherwise, this function returns **FALSE**.

Type& operator[] (**int** *n*);
Returns a reference to the the *n*th (zero-relative) element in the object. This function sets the current position to this element. If the index is negative or is greater than the number of nodes in the list, an **Error** exception is raised.

Type pop ();
Removes and returns the first element in the object. This function invalidates the current position. If there is nothing in the object, an **Error** exception is raised.

Boolean pop (*Type& value*);
Copies the first element in the object to *value* and removes it from the object. This function invalidates the current position. If there is nothing in the list, an **Error** exception is raised.

int position ();
Returns the current position as a zero-relative index into the object that can be used with the overloaded **operator[]**.

int position (**const** *Type& value*);
Searches the object for *value*. If the element is found, this function sets the current position to this element and returns the zero-relative index of this element; otherwise, this function returns –1.

inline Boolean prepend (**const List**<*Type*>& *l*);
Adds the elements of *l* to the beginning of the object and returns **TRUE**. This function sets the current position to the first element added. This function returns **FALSE** if the specified list argument is **NIL**.

Boolean prev ();
Moves the current position to the previous element in the object and returns **TRUE**. If the current position is invalid, this function sets the current position to the last element and returns **TRUE**. If the current position is the first element in the object, this function invalidates the current position and returns **FALSE**.

Boolean push (**const** *Type& value*);
Adds *value* to the beginning of the object and returns **TRUE**. This function sets the current position to the first element of the object. This function returns **FALSE** when heap memory is exhausted.

inline Boolean push_end (**const** *Type& value*);
Adds *value* to the end of the object and returns **TRUE**. This function sets the current position to the last element of the object. This function returns **FALSE** when heap memory is exhausted.

inline Boolean push_end_new (**const** *Type& value*);
Adds *value* to the end of the object (if it is not already in the object) and sets the current position to this element. This function returns **TRUE** if the element is added to the object; otherwise, this function returns **FALSE**.

inline Boolean push_new (**const** *Type& value*);
Adds *value* to the beginning of the object (if it is not already in the object) and sets the current position to this element. This function returns **TRUE** if the element is added to the object; otherwise, this function returns **FALSE**.

Boolean put (**const** *Type& value*, **int** *n* = 0);
Replaces the *n*th (zero-relative) element in the object with *value*. This function returns **TRUE** if the *n*th element exists; otherwise, this function returns **FALSE**. If *n* is negative, an **Error** exception is raised.

Type& remove ();

Removes the element at the current position in the object and returns a reference to the element. This function sets the current position to the element immediately following the element removed. If the current position is invalid, an **Error** exception is raised.

inline Boolean remove (**const** *Type& value*);

Removes the first occurrence of *value* in the object. If the element is found, this function removes it from the object, sets the current position to the element immediately following the element removed, and returns **TRUE**. If the element is not found, this function returns **FALSE**.

Boolean remove_duplicates ();

Removes any duplicate elements from the object. This function returns **TRUE** if any elements were removed; otherwise, this function returns **FALSE**. This function invalidates the current position.

inline Boolean replace (**const** *Type& value1*, **const** *Type& value2*);

Replaces the first occurrence of *value1* in the object with *value2* and sets the current position to this element. If *value1* is found, this function returns **TRUE**; otherwise, this function returns **FALSE**.

inline Boolean replace_all (**const** *Type& value1*, **const** *Type& value1*);

Replaces all occurrences of *value1* in the object with *value2*. This function returns **TRUE** if any elements were replaced; otherwise, this function returns **FALSE**. This function invalidates the current position.

inline void reset ();

Invalidates the current position in the object.

void reverse ();

Reverses the order of the elements in the object. This function invalidates the current position.

Boolean search (**const List**<*Type*>& *l*);

Searches for the sublist *l* in the object. If *l* is a sublist in the object, this function sets the current position in the object to the first element of the sublist and returns **TRUE**; otherwise, this function returns **FALSE**.

inline void set_compare (*List_Compare* = **NULL**);

Updates the compare function for the object. *List_Compare* is a function of type **Boolean** (**Function*) (**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator=** for the type over which the class is parameterized is used.

Boolean set_tail (**const List**<*Type*>& *l*, **int** *n* = 1);

Sets the *n*th tail of the object to *l*. This function sets the current position of the object to the first element of the *n*th tail. This function returns **TRUE** if the object has an *n*th tail; otherwise, this function returns **FALSE**.

inline void sort (*List_Predicate p*);

Sorts the elements of the object by using *p* for determining the collating sequence. *List_Predicate* is a function of type **int** (**Function*) (**const** *Type&*, **const** *Type&*) that returns –1 if the first argument should precede the second, zero if they are equal, and 1 if the first argument should follow the second.

Boolean sublist (**List**<*Type*>& *l1*, **const List**<*Type*>& *l2*);
Searches for *l2* in the object. If *l2* is a sublist of the object, this function sets the current position of the object to the first element of *l2*, sets *l1* to the sublist within the object (starting at the new current position), and returns **TRUE**; otherwise, this function sets *l1* list to **NIL** (an empty list) and returns **FALSE.**

void tail (**List**<*Type*>& *l*, **int** *n* = 1);
Sets *l* to point to the *n*th tail of the object. The *n*th tail is a list whose first element is the *n*th (zero-relative) element of the object. When it has no second argument, **tail** sets *l* to point to the first tail of the object. *n*=1 sets *l* to all of the elements in the object. This function sets *l* to **NIL** if *n* is greater than or equal to the number of elements in the object. This function sets the current position to the *n*th element of the object.

inline *Type&* **value** ();
Returns the element at the current position in the object. An **Error** exception is raised if the current position is invalid.

Friend Functions:

friend ostream& operator<< (*ostream& os*, **const List**<*Type*>& *l*);
Provides a formatted output capability for a reference to a list.

inline friend ostream& operator<< (*ostream& os*, **const List**<*Type*>* *l*);
Provides a formatted output capability for a pointer to a list.

List Example

7.4 The following program declares a list of strings and stores the words in a paragraph of text in individual nodes. The list of words is traversed using a parameterized iterator, and the nodes are manipulated to determine the total number of words, the number of unique words, and the most commonly used word in the paragraph. These results are sent to the standard output, and the program then ends.

```
#include <cool/List.h>                    // Include list header file
#include <cool/Gen_String.h>              // Include COOL String class
#include <cool/Iterator.h>                // Include COOL Iterator class
#include "paragraph.h"                    // Include Stroustrup text

DECLARE List<Gen_String>                  // Declare list type
IMPLEMENT List<Gen_String>                // Implement list type
DECLARE Iterator<List>                    // Declare list iterator type
IMPLEMENT Iterator<List>                  // Implement list iterator type

int main (void) {
  List<Gen_String> l1;                    // Declare list variable
  Gen_String s;                           // Temporary string variable
  int max_count = 0;                      // Temporary counting variable
  cout << text;                           // Output paragraph
  text.compile ("[a-zA-Z]+");             // Match any alphabetical word
  while (text.find ()) {                  // While still more words
   text.sub_string (s, text.start (), text.end ()); // Get word
   l1.push (s);                           // And add to list
  }
  l1.reset ();                            // Invalidate current position
  while (l1.next ()) {                    // While there are still nodes
    int counter = 0;                      // Initialize counter
    Gen_String cur_word;                  // Temporary string variable
    Iterator<List> i1 = l1.current_position (); // Save current position
    cur_word = l1.value ();               // Get word to be counted
    l1.reset ();                          // Invalidate current position
26       while (l1.next ())                         // While there are still nodes
27         if (l1.value () == cur_word)             // If word appears in list
28                 counter++;                       // Increment usage count
29       if (counter > max_count) {                 // If most used word so far
30         max_count = counter;                     // Update maximum count
31         s = cur_word;                            // And save word
32       }
33       l1.current_position () = i1;               // Restore old current position
34     }
35     cout << "There are " << l1.length () << " words\n";
36     l1.remove_duplicates (); // Remove duplicate words
37     cout << "There are " << l1.length () << " unique words\n";
38     cout << "The most common word is '" << s << "' and is used " <<
               max_count << " times\n";
39     exit (0);                                    // Exit with successful status
40   }
```

Lines 1 through 3 include the COOL `List.h`, `Gen_String.h`, and `Iterator.h` class header files. Line 4 includes a statically allocated **Gen_String** object that contains a paragraph of text to be scanned by the program. Lines 5 and 6 define a container class of a list of strings and lines 7 through 8 define an iterator for the list class. Lines 10 through 13 declare various variables and print the complete paragraph. A regular expression to match sequences of alphabetical characters (that is, words) is compiled in line 14. Lines 15 through 18 contain a loop that finds each word in the paragraph and pushes it onto the list. Line 18 resets the internal current position iterator inside the list object.

Lines 20 through 34 are the heart of the program. The loop iterates through the elements of the list, assigning each word to a current word variable and the current position to a list iterator object. An inner loop uses the current position functionality to loop through the elements of the list counting the number occurrences of the current word. If this count is the largest so far, both the word and the count are saved. When the inner loop terminates, the outer loop establishes the previous current position maintained by the iterator object and the procedure is repeated again until all words have been scanned. Lines 35 through 38 output the results of the word search and counting. Finally, the program ends with a successful completion code.

The following shows the output for the program:

```
A programming language serves two related purposes: it provides a
vehicle for the programmer to specify actions to be executed and a
set of concepts for the programmer to use when thinking about what
can be done. The first aspect ideally requires a language that is
'close to the machine', so that all important aspects of a machine
are handled simply and efficiently in a way that is reasonably
obvious to the programmer. The C language was primarily designed with
this in mind. The second aspect ideally requires a language that is
'close to the problem to be solved' so that the concepts of a
solution can be expressed directly and concisely. The facilities
added to C to create C++ were primarily designed with this in mind.

      -- Bjarne Stroustrup

There are 129 words
There are 71 unique words
The most common word is 'to' and is used 9 times
```

Pair Class

7.5 The parameterized **Pair**<*T1*,*T2*> class implements an association between one object and another. The objects may be of different types, with the first representing the *key* of the pair and the second representing the *value* of the pair. The **Pair**<*T1*,*T2*> class is used by the **Association**<*Ktype*,*Vtype*> class to implement an association list (that is, a vector of pairs of associated values).

Name: **Pair**<*T1*,*T2*> — A parameterized pair

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

Base Classes: None

Friend Classes: None

Constructors:

Pair<*T1*,*T2*> ();
Creates an empty pair of the specified types.

Pair<*T1*,*T2*> (**const Pair**<*T1*,*T2*>&);
Duplicates the size and value of a pair object.

Pair<*T1*,*T2*> (**const** *T1&*, **const** *T2&*);
Creates a pair from the two specified elements.

Member Functions:

inline const *T1&* **get_first** () **const**;
Returns a constant reference to the first element of the pair.

inline const *T2&* **get_second** () **const**;
Returns a constant reference to the second element of the pair.

inline T1& first ();
Returns a reference to the first element of the pair.

Pair<*T1*,*T2*>& **operator=** (**Pair**<*T1*,*T2*>&);
Assigns one pair object to have the value of another by duplicating element values.

Boolean operator== (**Pair**<*T1*,*T2*>&) **const**;
Returns **TRUE** if the pairs have the same element values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**Pair**<*T1*,*T2*>&) **const**;
Returns **TRUE** if the pairs have different element values; otherwise, this function returns **FALSE** .

inline T2& second ();
Returns a reference to the second element of the pair.

inline void set_compare (*Pair_Compare* = **NULL**);
Updates the compare function for this class of pair. *Pair_Compare* is a function of type **Boolean** (**Function*)(**const Pair**<*T1*,*T2*>&, **const Pair**<*T1*,*T2*>&). If no argument is provided, the **operator==** for the types over which the class is parameterized are used.

inline void set_first (**const** *T1&*);
Sets the first element of the pair to the specified value.

inline void set_second (**const** *T2&*);
Sets the second element of the pair to the specified value.

Friend Functions:

friend ostream& operator<< (*ostream&*, **const Pair**<*T1*,*T2*>&);
Provides a formatted output capability for reference to a **Pair**<*T1*,*T2*> object.

inline friend ostream& operator<< (*ostream&*, **const Pair**<*T1*,*T2*>*);
Provides a formatted output capability for a pointer to a **Pair**<*T1*,*T2*> object.

Association Class

7.6 The **Association**<*Ktype*,*Vtype*> class is privately derived from the **Vector**<*Type*>class and implements a collection of pairs. The first of the pair is called the *key*, and the second of the pair is called the *value*. The **Association**<*Ktype*,*Vtype*> class implements a one-dimensional vector parameterized over a pair of objects. The first type specifies the type of the key, and the second type specifies the type of the value. Many of the member functions for **Association**<*Ktype*,*Vtype*> are inherited from **Vector**<*Type*> and, consequently, are inline calls to the vector member function of the same name.

The **Association**<*Ktype*,*Vtype*> class inherits the dynamic growth capability of the **Vector** class. Vectors are, by default, dynamic in nature. A static-sized vector object is selected by setting the growth allocation size to zero or by passing in a pointer to a block of user-supplied storage to the constructor. If a vector is of static size and an operation is performed that requires more storage, an **Error** exception is raised.

The **Association**<K*type, Vtype*> class implements the notion of a current position. This is useful for iterating through the elements of a vector. The current position is maintained in a data member of type **Association_state** and is set or reset by all member functions affecting elements in the class. Member functions are provided to reset the current position, move to the next and previous elements, find an element, and get the value at the current position. The **Iterator**<*Type*> class provides a mechanism to save and restore the state associated with the current position, thus allowing the programmer to use multiple iterators over the same instance of an association object.

Name: **Association**<*Ktype*,*Vtype*> — A dynamic, parameterized association

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

Base Classes: **Vector**<*Type*>, **Vector**, **Generic**

Friend Classes: None

Constructors:

Association<*Ktype*,*Vtype*> ();
: Creates an empty association of the specified type.

Association<*Ktype*,*Vtype*> (**const Association**<*Ktype*,*Vtype*>& *assoc*);
: Duplicates the size and value of an association object.

Association<*Ktype*,*Vtype*> (**unsigned long** *number*);
: Allocates enough storage for an association of a specific type to hold *number* elements.

Association<*Type*> (**void*** *storage*, **unsigned long** *number*);
: Creates a static-sized association object for *number* elements whose storage *storage* is provided by the user. If an object of this type attempts to grow dynamically or the programmer invokes the **resize** member function, an **Error** exception is raised.

Member Functions:

inline long capacity ();
: Returns the maximum number of elements the association can contain.

inline void clear ()
: Removes all elements in the object and invalidates the current position.

inline Association_state& current_position ()**;**
: Returns the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of association.

Boolean find (**const** *Ktype& key*);
: Searches the association for *key*. If found, this function sets the current position and returns **TRUE**; otherwise, this function resets the current position and returns **FALSE**.

Boolean get (**const** *Ktype& key*, *Vtype& value*);
: Gets the associated *value* for *key*. This function returns **TRUE** and modifies *value* to contain the associated value. If *key* is not found, this function returns **FALSE** and does not modify *value*.

Boolean get_key (**const** *Vtype& value*, *Ktype& key*) **const**;
: Gets the first associated *key* for *value*. This function returns **TRUE** and modifies *key* to contain the associated key. If *value* is not found, this function returns **FALSE** and does not modify *key*.

inline const *Ktype&* **key** () **const**;
: Returns the key of the key/value pair at the current position.

inline long length ();
: Returns the number of elements (pairs) in the association.

inline Boolean next ();
Advances the current position pointer to the next element in the association and returns **TRUE.** If the current position is invalid, this function advances to the first element and returns **TRUE**. If advancing past the last element, this function invalidates the current position and returns **FALSE**.

Association<*Ktype*,*Vtype*>& **operator=** (**const Association**<*Ktype*,*Vtype*>&);
Overloads the assignment operator for the **Association** class and assigns one association object to have the value of another by duplicating the size and element values. This function invalidates the current position. If the association is prohibited from dynamically growing as necessary, an **Error** exception is raised.

Boolean operator== (**const Association**<*Ktype*,*Vtype*>& *assoc*) **const**;
Overloads the equality operator for the **Association** class. This function returns **TRUE** if the associations have the same number of elements with the same values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Association**<*Ktype*,*Vtype*>& *assoc*) **const**;
Overloads the inequality operator for the **Association** class. This function returns **TRUE** if the associations have a different number of elements or different values; otherwise, this function returns **FALSE**.

inline Boolean prev ();
Moves the current position pointer to the previous element in the association and returns **TRUE**. If the current position is invalid, this function moves to the last element and returns **TRUE**. If moving to the previous element passes the first element in the association, this function invalidates the current position and returns **FALSE**.

Boolean put (**const** *Ktype& key*, **const** *Vtype& value*);
Puts the *key*/*value* pair into the association. If a pair already exists with the specified key, the value for that pair is replaced with *value*. If required and not prohibited, the association is grown. If the new pair is successfully put into the association, **TRUE** is returned; otherwise, **FALSE** is returned.

Vtype& remove ();
Removes and returns a reference to the element at the current position. This function sets the current position to the element immediately following the element removed. If the element removed is at the end of the association, this function invalidates the current position. If the current position is invalid, and **Error** exception is raised.

Boolean remove (**const** *Ktype& key*);
Searches for *key* and, if found, this function removes the pair associated with *key* and sets the current position to the element immediately following the element removed; then, the function returns **TRUE**. If *key* is found at the end of the association, this function invalidates the current position and returns **TRUE**. If *key* is not found, this function returns **FALSE**.

inline void reset ();
Invalidates the current position.

inline void resize (**long** *number*);
Resizes the association for at least *number* elements. If a growth ratio has been selected and it satisfies the resize request, the association is grown by this ratio. This function invalidates the current position. If the size specified is zero or negative, an **Error** exception is raised.

inline void set_alloc_size (**int** *size*);

Updates the allocation growth size to be used when the growth ratio is zero. Default allocation growth size is 100 bytes. If the size specified is negative, an **Error** exception is raised.

inline void set_growth_ratio (**float** *ratio*);

Updates the growth ratio for this instance of an association to the specified value. When an association needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

inline void set_key_compare (*Assoc_Key_Compare* = **NULL**);

Updates the key compare function for this class of association. *Assoc_Key_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for *Ktype* over which the key for the association class is parameterized is used.

inline long set_length (**long** *number*);

Specifies the *number* of elements in an association to allow random access via the overloaded **operator[]** member function. If *number* is larger than the storage allocated, this function truncates *number* to the largest value the allocated size will support. This function returns the updated number of elements.

inline void set_value_compare (*Assoc_Value_Compare* = **NULL**);

Updates the value compare function for this class of association. *Assoc_Value_Compare* is a function of type **Boolean** (**Function*)(**const** *Ktype&*, **const** *Vtype&*). If no argument is provided, the **operator==** for *Vtype* over which the value for the association class is parameterized is used.

inline Vtype& value ();

Returns a reference to the value of the key/value pair at the current position.

Friend Functions:

friend ostream& operator<< (*ostream os*,
const Association<*Ktype,Vtype*>*& assoc*);

Provides a formatted output capability for reference to an **Association**<*Ktype,Vtype*> object.

inline friend ostream& operator<< (*ostream& os*,
const Association<*Ktype,Vtype*>** assoc*);

Provides a formatted output capability for a pointer to an **Association**<*Ktype,Vtype*> object.

Association Example

7.7 The following program declares an association of strings and integers, storing each word and its frequency of occurrence in a paragraph of text in individual elements. The association of words is traversed using the current position functionality of the class to determine the total the number of words and the most commonly used word in the paragraph.

```
#include <cool/Association.h>            // Include Association class
#include <cool/Gen_String.h>             // Include COOL String class
#include <cool/Iterator.h>               // Include COOL Iterator class
#include "paragraph.h"                   // Include Stroustrup text

DECLARE Association<Gen_String,int>      // Declare association type
IMPLEMENT Association<Gen_String,int>    //  Implement association type
DECLARE Iterator<Association>            // Declare assoc iterator
IMPLEMENT Iterator<Association>          // Implement assoc iterator
```

```
 9    int main (void) {
10      Association<Gen_String,int> a1;          // Declare Association variable
11      Gen_String s;                            // Temporary string variable
12      int counter = 0, max_count = 0;          // Initialize word counters
13      cout << text;                            // Output paragraph
14      text.compile (”[a-zA-Z]+”);              // Match any alphabetical word
15      while (text.find ()) {                   // While still more words
16       text.sub_string (s, text.start (), text.end ()); // Get word
17       if (a1.find (s))                        // If word already found
18         ++a1.value ();                        // Increment use count
19     else a1.put (s, 1);                       // Else add word
20      }
21      a1.reset ();                             // Invalidate current position
22     Iterator<Association> i1;                 // Iterator object
23     while (a1.next ()) {                      // While there are still nodes
24      counter += a1.value ();                  // Sum number of words used
25      if (a1.value () > max_count) {           // If most used word so far
26        i1 = a1.current_position ();           // Save position in list
27        max_count = a1.value ();               // And keep track of usage
28      }
29     }
30    cout << ”There are ” << counter << ” words\n”;
31    cout << ”There are ” << a1.length () << ” unique words\n”;
32    a1.current_position () = i1;               // Set position of most used word
33    cout << ”The most common word is ‘” << a1.key () << ”’ and is used ” <<
                    a1.value () << ” times\n”;
34    exit (0);                                  // Exit with successful status
35   }
```

Lines 1 through 3 include the COOL `Association.h`, `Gen_String.h`, and `Iterator.h` class header files. Line 4 includes a statically allocated **Gen_String** object that contains a paragraph of text to be scanned by the program. Lines 5 and 6 define a container class of an association of strings and integers, and lines 7 and 8 define an iterator for the association class. Lines 10 through 13 declare various variables and print the complete paragraph. A regular expression to match sequences of alphabetical characters (that is, words) is compiled in line 14. Lines 15 through 20 contain a loop that finds each word in the paragraph and adds it to the association if not already there. Otherwise, the current frequency is incremented. Line 21 resets the internal current position iterator inside the association object, and line 22 defines an iterator for an association object.

Lines 23 through 29 are the heart of the program. The loop iterates through the elements of the association summing up the frequencies of all the words to get a total word count. In addition, if the count for a given word is the largest so far, the position in the association is saved in the iterator object. This procedure repeats until all words have been scanned. Lines 30 through 33 output the results of the word search and counting. Finally, the program ends with a successful completion code.

The following shows the output for the program:

```
A programming language serves two related purposes: it provides a
vehicle for the programmer to specify actions to be executed and a
set of concepts for the programmer to use when thinking about what
can be done. The first aspect ideally requires a language that is
'close to the machine', so that all important aspects of a machine
are handled simply and efficiently in a way that is reasonably
obvious to the programmer. The C language was primarily designed with
this in mind. The second aspect ideally requires a language that is
'close to the problem to be solved' so that the concepts of a
solution can be expressed directly and concisely. The facilities
added to C to create C++ were primarily designed with this in mind.

     -- Bjarne Stroustrup

There are 129 words
There are 71 unique words
The most common word is 'to' and is used 9 times
```

Hash_Table Class

7.8 The **Hash_Table**<*Ktype,VType*> class is publicly derived from the **Hash_Table** class and implements hash tables of user-specified types for both the key and the value. This is accomplished by using the parameterized type capability of C++. The **Hash_Table** class is dynamic in nature. Its size (that is, the number of buckets in the table) is always a prime number. Each bucket holds eight items. No holes are left in a bucket; if a key/value pair is removed from the middle of a bucket, the following entries are moved up. When a hash on a key ends up in a bucket that is full, the table is enlarged.

Name: **Hash_Table**<*Ktype,Vtype*> — A dynamic, parameterized hash table

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

Base Classes: **Hash_Table, Generic**

Friend Classes: None

Constructors:

Hash_Table<*Ktype,Vtype*> ();
Allocates a hash table of the default size (three buckets).

Hash_Table<*Ktype,Vtype*> (**unsigned long** *number*);
Allocates a hash table with at least enough buckets for *number* entries.

Hash_Table<*Ktype,Vtyp*e> (**const Hash_Table**<*Ktype,Vtype*>& *ht*);
Duplicates the size and entries of another hash table object *ht*.

Member Functions:

inline long capacity () **const**;
Returns the maximum number of entries that the hash table can hold.

void clear ();
Removes all entries from the hash table and adjusts the appropriate counts.

inline Hash_Table_state& current_position () **const;**
Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of hash table.

Boolean find (**const** *Ktype& key*);

Searches the hash table for *key* and returns **TRUE** if found; otherwise, this function returns **FALSE**. If *key* is found, this function sets the current position to the key/value entry; otherwise, this function invalidates the current position.

Boolean get (**const** *Ktype& key, Vtype& value*);

Calculates the hash value for *key* and returns the value associated with that key in the table by copying it to *value*. This function sets the current position to the key/value pair. If *key* is found, this function returns **TRUE**; otherwise, this function returns **FALSE**.

inline long get_bucket_count () **const**;

Returns the prime number of buckets currently allocated for the hash table.

inline int get_count_in_bucket (**long** *n)* **const**;

Returns the number of keys currently hashed to the zero-relative *n*th bucket.

Boolean get_key (**const** *Vtype& value, Ktype& key*);

Searches the table for *value*. If found, this function copies the associated key into *key*, sets the current position to the key/value pair, and returns **TRUE**. If *value* is not found in the hash table, this function invalidates the current position and returns **FALSE**.

inline Boolean is_empty () **const**;

Returns **TRUE** if the hash table contains no entries; otherwise, this function returns **FALSE**.

const *Ktype&* **key** ();

Returns the key of the key/value pair at the current position. If the current position is invalid, an **Error** exception is raised.

inline long length () **const**;

Returns the number of entries in the hash table.

Boolean next ();

Advances the current position pointer to the next entry in the hash table and returns **TRUE**. If the current position is invalid, this function advances to the first entry and returns **TRUE**. If advancing past the last entry in the hash table, this function invalidates the current position and returns **FALSE**.

Hash_Table<*Ktype,Vtype*>& **operator=** (**const**
    Hash_Table<*Ktype,Vtype*>& *ht*);

Overloads the assignment operator for the class and assigns one hash table object to have the value of another by duplicating the size and entries. This function invalidates the current position of the object.

Boolean operator== (**const Hash_Table**<*Ktype,Vtype*>& *ht*);

Overloads the equality operator for the hash table class. This function returns **TRUE** if the tables have the same number of buckets with the same key/value pairs; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Hash_Table**<*Ktype,Vtype*>& *ht*);

Overloads the inequality operator for the hash table class. This function returns **TRUE** if the tables have a different number of buckets or different key/value pairs; otherwise, this function returns **FALSE**.

Boolean prev ();
Moves the current position pointer to the previous entry in the hash table and returns **TRUE**. If the current position is invalid, this function moves to the last entry and returns **TRUE**. If moving to the previous entry passes the first entry in the hash table, this function invalidates the current position and returns **FALSE**.

Boolean put (**const** *Ktype& key*, **const** *Vtype& value*);
Searches the hash table for *key* and puts the corresponding key/value pair into the hash table. If *key* is not there, the key/value pair is added and **TRUE** is returned; otherwise, if *key* is already there, this function updates the value with *value* and returns **FALSE**. If the bucket determined by the hash is full, the table grows and the key/value pairs are rehashed and inserted. This function sets the current position to the key/value pair.

Boolean remove ();
Removes the key/value at the current position and returns **TRUE**. This function sets the current position to the entry immediately following the entry removed if in the same bucket; otherwise, this function invalidates the current position. If the current position is invalid, an **Error** exception is raised and, if the handler returns, this function returns **FALSE**.

Boolean remove (**const** *Ktype& key*);
Searches the hash table for *key*, removes the indicated key/value pair from the table, sets the current position to the old location of the key/value pair, and returns **TRUE**. If *key* is not found in the hash table, this function returns **FALSE**.

inline void reset ();
Invalidates the current position.

Boolean resize (**long** *number*);
Resizes the hash table for at least the indicated number of entries. If a growth ratio has been selected and it satisfies the resize request, the table is grown by this ratio. This function invalidates the current position. If the resize value is zero or negative, an **Error** exception is raised.

inline void set_hash (*Hash h*);
Updates the hash function for this instance of hash table. *Hash* is a function of type **unsigned long** (**Function*) (**const** *Ktype&*). If the key is of type **char***, the hash is the result of successively exclusive-or-ing each byte with the current hash value shifted left seven bits. If the key is not of type **char***, the default hash function is the computation of a 32-bit value shifted left three bits with the result then modulo the prime number of buckets. If the size of (*Ktype*) is greater than four bytes, the 32-bit value is computed by successively exclusive-or-ing 32-bit values for the length of the key.

void set_key_compare (*Hash_Key_Compare* = **NULL**);
Updates the key compare function for this instance of hash table. *Hash_Key_Compare* is a function of type **Boolean** (**Function*)(**const** *Ktype&*, **const** *Ktype&*). If no argument is provided, the **operator==** for *Ktype*, the key over which the class is parameterized, is used. If the key is a **char***, a **String**, or a **Gen_String**, the default compare function is a string comparison.

inline void set_ratio (*float*);
Updates the growth ratio for this instance of the hash table to the specified value. When a hash table needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

void set_value_compare (*Hash_Value_Compare* = **NULL**);
Updates the value compare function for this instance of hash table. *Hash_Value_Compare* is a function of type **Boolean** (**Function*)(**const** *Vtype&*,**const** *Vtype&*). If no argument is provided, the **operator==** for *Vtype*, the value over which the class is parameterized, is used.

const *Vtype&* **value** ();
Returns a reference to the value of the key/value pair at the current position. If the current position is invalid, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (*ostream& os*,
const Hash_Table<*Ktype,Vtype*>& *ht*);
Overloads the output operator for a reference to a **Hash_Table**<*Ktype,Vtype*> object. This function provides a formatted output with key/value pairs printed one per line.

inline friend ostream& operator<< (*ostream& os*,
const Hash_Table<*Ktype,Vtype*>* *ht*);
Overloads the output operator for a pointer to a **Hash_Table**<*Ktype,Vtype*> object. This function provides a formatted output with key/value pairs printed one per line.

Hash Table Example

7.9 The following program declares a hash table of strings and integers, storing each word as the key and its frequency of occurrence in a paragraph of text as the value. The hash table is traversed using the current position function of the class to determine the total number of words and the most commonly used word in the paragraph.

```
#include <cool/Hash_Table.h>              // Include Hash_Table class
#include <cool/Gen_String.h>              // Include COOL String class
#include <cool/Iterator.h>                // Include COOL Iterator class
#include "paragraph.h"                    // Include Stroustrup text

DECLARE Hash_Table<Gen_String,int>        // Declare Hash_Table type
IMPLEMENT Hash_Table<Gen_String,int>      // Implement Hash_Table type
DECLARE Iterator<Hash_Table>              // Declare Hash_Table iterator
IMPLEMENT Iterator<Hash_Table>            // Implement Hash_Table iterator
```

```
 9   int main (void) {
10     Hash_Table<Gen_String,int> a1;           // Declare Hash_Table variable
11     Gen_String s;                            // Temporary string variable
12     int counter = 0, max_count = 0;          // Initialize word counters
13     cout << text;                            // Output paragraph
14     text.compile (”[a-zA-Z]+”);              // Match any alphabetical word
15     while (text.find ()) {                   // While still more words
16      text.sub_string (s, text.start (), text.end ()); // Get word
17      if (h1.find (s))                        // If word already found
18        h1.put (h1.key (), h1.value ()+1);    // Update use count
19    else h1.put (s, 1);                       // Else add word
20     }
21     h1.reset ();                             // Invalidate current position
22   Iterator<Hash_Table> i1;                   // Iterator object
23   while (h1.next ()) {                       // While there are still nodes
24    counter += h1.value ();                   // Sum number of words used
25    if (h1.value () > max_count) {            // If most used word so far
26      i1 = h1.current_position ();            // Save position in list
27      max_count = h1.value ();                // And keep track of usage
28    }
29   }
30   cout << ”There are ” << counter << ” words in the paragraph\n”;
31   cout << ”There are ” << h1.length () << ” unique words in the paragraph\n”;
32   h1.current_position () = i1;               // Set position of most used word
33   cout << ”The most common word is ‘” << h1.key () << ”’ and is used ” <<
                   h1.value () << ” times\n”;
34   exit (0);                                  // Exit with successful status
35   }
```

Lines 1 through 3 include the COOL `Hash_Table.h`, `Gen_String.h`, and `Iterator.h` class header files. Line 4 includes a statically allocated paragraph of text to be scanned by the program. Lines 5 and 6 define a container class of a hash table whose key is a string and whose value is an integer. Lines 7 and 8 define an iterator for the hash table class. Lines 10 through 13 declare various variables and print the complete paragraph. A regular expression to match sequences of alphabetical characters (that is, words) is compiled in line 14. Lines 15 through 20 contain a loop that finds each word in the paragraph and adds it to the hash table if not already there. Otherwise, the current frequency is incremented and used as the new value for the key. Line 21 resets the internal current position iterator inside the hash table object and line 22 defines an iterator for a hash table object.

Lines 23 through 29 are the heart of the program. The loop iterates through the elements of the hash table summing up the frequencies of all the words to get a total word count. In addition, if the count for a given word is the largest so far, the position in the table is saved in the iterator object. This procedure repeats until all words have been scanned. Lines 30 through 33 output the results of the word search and count. Finally, the program ends with a successful completion code.

The following shows the output for the program:

```
A programming language serves two related purposes: it provides a
vehicle for the programmer to specify actions to be executed and a
set of concepts for the programmer to use when thinking about what
can be done. The first aspect ideally requires a language that is
'close to the machine', so that all important aspects of a machine
are handled simply and efficiently in a way that is reasonably
obvious to the programmer. The C language was primarily designed with
this in mind. The second aspect ideally requires a language that is
'close to the problem to be solved' so that the concepts of a
solution can be expressed directly and concisely. The facilities
added to C to create C++ were primarily designed with this in mind.

      -- Bjarne Stroustrup

There are 129 words
There are 71 unique words
The most common word is 'to' and is used 9 times
```

SET CLASSES

8

Introduction

8.1 The set classes implement two basic data structures for random-access set operations.The following classes are discussed in this section:

- **Set**<*Type*>
- **Bit_Set**

The **Set**<*Type*> class implements random-access sets of objects of a user-specified type. Classical set operations such as union, intersection, and difference are available. In addition, the **Set**<*Type*> class supports the notion of a current position. See Section 5, Parameterized Templates, for more information regarding the current position mechanism and the **Iterator**<*Type*> class. The **Bit_Set** class implements efficient bit sets stored in an arbitrary-length vector of bytes (unsigned **char**) large enough to represent the specified number of elements. A bit set is indexed by integer values so elements can be integers, enum values, constant symbols from the enumeration package (discussed in Section 11, Symbols and Packages), or any other type of object or expression that results in an integral value.

In order to achieve successful compilation and usage, certain operations must be supported by any user-specified type over which the **Set**<*Type*> class is parameterized. The member functions **operator=**, **operator<**, **operator>**, **operator==**, and **operator<<** for both pointer and reference must be overloaded for any class object used as the type. Note that built-in types already have these functions defined.

NOTE: The **Set** class uses **operator=** of the parameterized type when copying elements. You should be careful when parameterizing a set over a pointer to a type, since the default pointer assignment operator usually copies the pointer, not the value pointed at.

Requirements

8.2 This section discusses set classes. It assumes that you have read and understood Section 5, Parameterized Templates. In addition, no attempt is made to discuss the concepts and algorithms for the data structures discussed. You should refer to a general data structures or computer science text for this information.

Set Class

8.3 The **Set**<*Type*> class implements a set of elements of a user-specified type. The **Set**<*Type*> class implements a simple one-element hash table where the key and the value are the same. The type of the **Set**<*Type*> class is the key/value in the hash table. The **Set**<*Type*> class is publicly derived from the **Hash_Table** class and is dynamic in nature. Its size (that is, the number of buckets in the table) is always a prime number. Each bucket holds eight items. No holes are left in a bucket; if a key/value is removed from the middle of a bucket, the following entries are moved up. When a hash on a key ends up in a bucket that is full, the table is enlarged. Growth of a specific instance of the class can be controlled automatically by setting a growth ratio or manually by a resize member function.

Name: **Set** — A dynamic, parameterized set

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

Base Classes: **Hash_Table**, **Generic**

Friend Classes: None

Public Constructors:

Set<*Type*> ();
Allocates a set of the default size (24 elements).

Set<*Type*> (**unsigned long** *number*);
Allocates a set with at least enough storage for *number* elements.

Set<*Type*> (**const Set**<*Type*>& *st*);
Duplicates the size and elements of another set object *st*.

Member Functions:

Boolean find (**const** *Type& value*);
Searches the set for *value*. If the element is found, this function updates the current position and returns **TRUE**; otherwise, this function invalidates the current position and returns **FALSE**.

Boolean next_difference (**Set**<*Type*>& *st*);
Determines the next element in the difference of the set object and *st*. This function sets the current position in the set object to that element and returns **TRUE**. If no more elements are in the difference, this function invalidates the current position and returns **FALSE**.

Boolean next_intersection (**Set**<*Type*>& *st*);
Determines the next element in the intersection of the set object and *st*. This function sets the current position in the set object to that element and returns **TRUE**. If no more elements are in the intersection, this function invalidates the current position and returns **FALSE**.

Boolean next_union (**Set**<*Type*>& *st*);
Determines the next element in the union of the set object and *st*. This function sets the current position in the set object to that element and returns **TRUE**. If no more elements are in the union, this function invalidates the current position and returns **FALSE**.

Boolean next_xor (**Set**<*Type*>& *st*);
Determines the next element in the exclusive-or of the set object and *st*. This function sets the current position in the set object to that element and returns **TRUE**. If no more elements are in the exclusive-or, this function invalidates the current position and returns **FALSE**.

Set<*Type*> **operator–** (**Set**<*Type*>& *st*);
Determines the logical difference of the set object and *st* and returns the result. This function invalidates the current position.

Set<*Type*> **operator^** (**Set**<*Type*>& *st*);
Determines the logical exclusive-or of two sets and returns the result. This function invalidates the current position.

Set<*Type*> **operator&** (**Set**<*Type*>& *st*);
Determines the logical intersection of two sets and returns the result. This function invalidates the current position.

Set<*Type*> **operator|** (**Set**<*Type*>& *st*);
Determines the logical union of two sets and returns the result. This function invalidates the current position.

Set<*Type*>**& operator=** (**const Set**<*Type*>& *st*);
Duplicates the size and elements of another set object *st*. This function invalidates the current position of the set object and returns a reference to the modified object.

Set<*Type*>**& operator–=** (**Set**<*Type*>& *st*);
Determines the logical difference of two sets and returns the modified set object. This function invalidates the current position of the set object and returns a reference to the modified object.

Set<*Type*>**& operator^=** (**Set**<*Type*>& *st*);
Determines the logical exclusive-or of two sets and returns the modified set object. This function invalidates the current position of the set object and returns a reference to the modified object.

Set<*Type*>**& operator&=** (**Set**<*Type*>& *st*);
Determines the logical intersection of two sets and returns the modified set object. This function invalidates the current position of the set object and returns a reference to the modified object.

Set<*Type*>**& operator|=** (**Set**<*Type*>& *st*);
Determines the logical union of two sets and returns the modified set object. The current position is invalidated of the set object and returns a reference to the modified object.

Boolean operator== (**const Set**<*Type*>& *st*) **const**;
This function returns **TRUE** if the sets have an equal number of elements with the same values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Set**<*Type*>& *st*) **const**;
This function returns **TRUE** if the sets have an unequal number of elements or values; otherwise, this function returns **FALSE**.

Boolean put (**const** *Type& value*);
Adds *value* to the set. If the set is not large enough and it can grow, this function allocates enough storage, copies the old set elements, sets the current position to *value,* and returns **TRUE**; otherwise, this function returns **FALSE**.

Boolean remove ();

Removes the element at the current position and returns **TRUE**. This function sets the current position to the element immediately following the element removed if not at end of the set; otherwise, this function invalidates the current position and returns **TRUE**. If the current position is invalid, an **Error** exception is raised and this function returns **FALSE**.

Boolean remove (**const** *Type& value*);

Searches for the specified element. If the element is found, this function removes the element, sets the current position to the element immediately following the element removed, and returns **TRUE**. If the element found is the last element in the set, this function invalidates the current position and returns **TRUE**. Otherwise, this function returns **FALSE**.

Boolean resize (**long** *number*);

Resizes the set for at least *number* elements. This function invalidates the current position. If *number* is zero or negative, an **Error** exception is raised.

Boolean search (**Set**<*Type*>& *st*);

Determines if *st* is a subset of the set object. If found, this function sets the current position to the start of the subset and returns **TRUE**; otherwise, this function returns **FALSE**.

inline void set_compare (*Set_Compare* = **NULL**);

Updates the compare function for this instance of set. *Set_Compare* is a function of type **Boolean** (**Function*) (**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the set is parameterized is used.

inline void set_difference (**Set**<*Type*>& *st*);

Determines the logical difference of two sets and modifies the source with the result. This function invalidates the current position in the set object.

inline void set_ratio (**float** *ratio*);

Updates the growth ratio for this instance of a set to *ratio*. When a set needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, and **Error** exception is raised.

inline void set_hash (*Set_Hash*);

Updates the hash function for this instance of set. *Set_Hash* is a function of type **unsigned long** (**Function*) (**const** *Type&*). If the set object is parameterized over the type **char***, the default hash is the result of successively exclusive-or-ing each byte with the current hash value shifted left seven bits. If the type is not **char***, the default hash function is the computation of a 32-bit value that is shifted left three bits with the result then modulo the prime number of buckets. If the size of *Type* is greater than four, the 32-bit value is computed by successively exclusive-or-ing 32-bit values for the length of the key.

inline void set_intersection (**Set**<*Type*>& *st*);

Determines the logical intersection of *st* and the set object, modifying the set object with the result. This function invalidates the current position in the set object.

inline void set_union (**Set**<*Type*>& *st*);

Determines the logical union of *st* and the set object, modifying the set object with the result. This function invalidates the current position in the set object.

inline void set_xor (**Set**<*Type*>& *st*);

Determines the logical exclusive-or of *st* and the set object, modifying the set object with the result. This function invalidates the current position in the set object.

Type& value ();
Returns the element at the current position. If the current position is invalid, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (**ostream&** *os*, **const Set**<*Type*>& *st*);
Overloads the output operator for a reference to a **Set**<*Type*> object to provide a formatted output capability.

inline friend ostream& operator<< (**ostream&** *os*, **const Set**<*Type*>* *st*);
Overloads the output operator for a pointer to a **Set**<*Type*> object to provide a formatted output capability.

Set Class Example

8.4 The following program manipulates sets of string objects representing the names of various colors. Two sets with different elements are created and several set operations are then applied. The results of each operation are sent to the standard output via the overloaded **operator<<** function.

```
 1   #include <COOL/String.h>                    // COOL String class
 2   #include <COOL/Set.h>                       // COOL Set class
 3   DECLARE Set<String>;                        // Declare set of strings
 4   IMPLEMENT Set<String>;                      // Implement set of strings

 5   Boolean my_compare (const String& s1, const String& s2) {
 6     return ((strcmp (s1, s2) == 0) ? TRUE : FALSE);
 7   }
 8   static String color_table[] = { "RED", "YELLOW", "PINK", "GREEN",
                       "ORANGE", "PURPLE", "BLUE" };

 9   int main (void) {
10     Set<String> a(5), b(5);                   // Declare two set objects
11     a.set_compare (my_compare);               // Establish compare function
12     for (int i = 0; i < 5; i++) {             // For each color defined
13                a.put (color_table[i]);        // Add object to first set
14                b.put (color_table[6-i]);      // Add end object to second set
15     }
16     cout << "Set A contains: " << a;          // Elements of set 1
17     cout << "Set B contains: " << b;          // Elements of set 2
18     cout << "A | B: " << (a | b);             // Display union
19     cout << "A & B: " << (a & b);             // Display intersection
20     cout << "A ^ B: " << (a ^ b);             // Display exclusive-or
21     cout << "A - B: " << (a - b);             // Display difference
22     exit (0);                                 // Exit with OK status
23   }
```

Lines 1 through 4 include the COOL `String.h` and `Set.h` header files, and then declare and implement a set of strings. Lines 5 through 7 define a string comparison routine to be used by the set class. Line 8 defines a static array of strings whose values are the names of colors to be used as element values in a set. Line 10 defines two objects each of which is a set with initial storage for five elements. Line 11 establishes the comparison routine to be used by the set objects. Lines 12 through 17 add some elements to each set and output the resulting objects. Lines 18 through 21 perform four set operations and display the results. The program ends with a successful completion code.

The following shows the output from the program:

```
Set A contains: [ RED YELLOW ORANGE PINK GREEN ]
Set B contains: [ BLUE PURPLE ORANGE GREEN PINK ]
A | B: [ RED YELLOW BLUE PURPLE ORANGE PINK GREEN ]
A & B: [ ORANGE PINK GREEN ]
A ^ B: [ RED YELLOW BLUE PURPLE ]
A - B: [ RED YELLOW ]
```

Bit_Set Class

8.5 The **Bit_Set** class is publicly derived from the **Generic** class and implements efficient bit sets. These bits are stored in an arbitrary-length vector of bytes (unsigned **char**) large enough to represent the specified number of elements. A bit set is indexed by integer values: Zero represents the first bit, one the second, two the third, and so on with each integer value actually indicating the zero-relative bit position in the bit vector. Elements can be integers, enum values, constant symbols from the enumeration package, or any other type of object or expression that results in an integral value. All operations involving bit shifting are performed in byte increments, giving the most efficient operation on common hardware architectures.

Name: **Bit_Set** — Efficient, dynamic bit sets

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

Base Classes: None

Friend Classes: None

Public Constructors:

Bit_Set ();
Allocates a bit set of the default size (1 byte).

Bit_Set (**unsigned int** *number*);
Allocates a bit set with at least enough storage for *number* elements.

Bit_Set (**const Bit_Set&** *bs*);
Duplicates the size and elements of another bit set object *bs*.

Member Functions:

inline int capacity () **const**;
Returns the maximum number of elements that the bit set can hold.

void clear ();
Removes all elements from the bit set and adjusts the appropriate counts.

inline Bit_Set_state& current_position () **const;**
Returns the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of bit set.

Boolean find (**unsigned int** *n*);
Searches the bit set for the *n*th zero-relative bit. If the bit is set, this function updates the current position and returns **TRUE**; otherwise, this function invalidates the current position and returns **FALSE**. If *n* is out of range, an **Error** exception is raised.

inline Boolean is_empty () **const**;
Returns **TRUE** if the bit set contains no elements; otherwise, this function returns **FALSE**.

int length () **const**;
Returns the number of elements in the bit set.

Boolean next ();
Advances the current position pointer to the next element in the bit set and returns **TRUE**. If the current position is invalid, this function advances to the first element and returns **TRUE**. If advancing past the last element in the bit set, this function invalidates the current position and returns **FALSE**.

Boolean next_difference (**const Bit_Set&** *bs*);
Determines the next element in the difference of the set object and *bs*. This function sets the current position in the set object to that element and returns **TRUE**. If no more elements are in the different, this function invalidates the current position and returns **FALSE**.

Boolean next_intersection (**const Bit_Set&** *bs*);
Determines the next element in the intersection of the set object and *bs*. This function sets the current position of the set object to that element and returns **TRUE**. If no more elements are in the intersection, this function invalidates the current position and returns **FALSE**.

Boolean next_union (**const Bit_Set&** *bs*);
Determines the next element in the union of the set object and *bs*. This function sets the current position of the set object to that element and returns **TRUE**. If no more elements are in the union, this function invalidates the current position and returns **FALSE**.

Boolean next_xor (**const Bit_Set&** *bs*);
Determines the next element in the exclusive-or of the set object and *bs*. This function sets the current position of the set object to that element and returns **TRUE**. If no more elements are in the exclusive-or, this function invalidates the current position and returns **FALSE**.

Bit_Set operator– ();
Overloads the unary minus operator to return the complement bit set.

Bit_Set operator– (**const Bit_Set&** *bs*);
Determines the logical difference of the set object and *bs* and returns the result. This function invalidates the current position of the set object.

inline Bit_Set operator~ ();
Returns the complement of a bit set.

Bit_Set operator^ (**const Bit_Set&** *bs*);
Determines the logical exclusive-or of the set object and *bs* and returns the result. This function invalidates the current position of the set object.

Bit_Set operator& (**const Bit_Set&** *bs*);
Determines the logical intersection of the set object and *bs* and returns the result. This function invalidates the current position of the set object.

Bit_Set operator| (**const Bit_Set&** *bs*);
Determines the logical union of the set object and *bs* and returns the result. This function invalidates the current position of the set object.

Bit_Set& operator= (**const Bit_Set&** *bs*);
Duplicates the size and elements of another bit set. This function invalidates the current position of the set object and returns a reference to the updated object.

Bit_Set& operator–= (**const Bit_Set&** *bs*);
Determines the logical difference of the set object and *bs* and modifies the source with the result. This function returns a reference to the modified bit set and invalidates the current position of the set object.

Bit_Set& operator^= (**const Bit_Set&** *bs*);
Determines the logical exclusive-or of the set object and *bs* and modifies the source with the result. This function returns a reference to the modified bit set and invalidates the current position of the set object.

Bit_Set& operator&= (**const Bit_Set&** *bs*);
Determines the logical intersection of the set object and *bs* and modifies the source with the result. This function returns a reference to the modified bit set and invalidates the current position of the set object.

Bit_Set& operator|= (**const Bit_Set&** *bs*);
Determines the logical union of the set object and *bs* and modifies the source with the result. This function returns a reference to the modified bit set and invalidates the current position of the set object.

Boolean operator== (**const Bit_Set&** *bs*) **const**;
Overloads the equality operator for the **Bit_Set** class. This function returns **TRUE** if the sets have an equal number of elements with the same values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Bit_Set&** *bs*) **const**;
Overloads the inequality operator for the **Bit_Set** class. This function returns **TRUE** if the sets have an unequal number of elements or unequal values otherwise, this function returns **FALSE**.

inline Boolean operator[] (**int** *n*) **const**;
Returns **TRUE** or **FALSE** to indicate the setting of the zero-relative *n*th bit. If the index is out of range, an **Error** exception is raised.

Boolean prev ();
Moves the current position pointer to the previous element in the bit set and returns **TRUE**. If the current position is invalid, this function moves to the last element and returns **TRUE**. If moving to the previous element passes the first element in the bit set, this function invalidates the current position and returns **FALSE**.

Boolean put (**int** *n*);
Adds the zero-relative *n*th element to the bit set. If the bit vector is not large enough and it can grow, this function allocates enough storage, copies the old bit set elements, updates the current position, and returns **TRUE**; otherwise, this function returns **FALSE**. If the index is out of range, an **Error** exception is raised.

Boolean put (**int** *start*, **int** *end*);
Adds the specified range of elements (inclusive) to the bit set by setting the appropriate zero-relative bits. If the bit vector is not large enough and it can grow, this function allocates enough storage, copies the old bit set elements, updates the current position, and returns **TRUE**; otherwise, this function returns **FALSE**. If the *start* or *end* are out of range, an **Error** exception is raised.

Boolean remove ();

This function removes the element at the current position, sets the current position to the element immediately following the element removed (if not at the end of the vector), and returns **TRUE.** If the element is at the end of the bit vector, this function removes the element, invalidates the current position, and returns **TRUE**. Otherwise, this function invalidates the current position and returns **FALSE**. If the current position is invalid, an **Error** exception is raised and this function returns **FALSE**.

Boolean remove (**int** *n*);

Searches for the zero-relative *n*th element. If the element is found, this function removes the element, sets the current position to the element immediately following the element removed, and returns **TRUE**. If the element is found but at the end of the bit vector, this function removes the element, invalidates the current position, and returns **TRUE**. Otherwise, this function returns **FALSE**. If the index is out of range, an **Error** exception is raised.

Boolean remove (**int** *start*, **int** *end*);

Searches for the specified range of elements. If the range is found, this function removes the range of elements, sets the current position to the starting element position, and returns **TRUE**. Otherwise, this function returns **FALSE**. If either index is out of range, an **Error** exception is raised.

inline void reset ();

Invalidates the current position.

resize (**int** *number*);

Resizes the bit set for at least *number* elements. This function invalidates the current position.

Boolean search (**const Bit_Set&** *bs*) **const**;

Determines if *bs* is a subset of the bit set object. If found, this function returns **TRUE**; otherwise, this function returns **FALSE**.

inline void set_alloc_size (**int** *number*);

Sets the allocation growth size to *number* of bytes. The growth allocation size is used when the growth ratio is zero. Default allocation growth size is four bytes.

inline void set_difference (**const Bit_Set&** *bs*);

Determines the logical difference of *bs* and the bit set object, modifying the source with the result. This function invalidates the current position in the bit set object.

inline void set_growth_ratio (**float** *ratio*);

Updates the growth ratio for this instance of a bit set to *ratio*. When a bit set needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, and **Error** exception is raised.

inline void set_intersection (**const Bit_Set&** *bs*);

Determines the logical intersection of *bs* and the bit set object, modifying the source with the result. This function invalidates the current position of the bit set object.

inline void set_union (**const Bit_Set&** *bs*);

Determines the logical union of *bs* and the bit set object, modifying the source with the result. This function invalidates the current position of the bit set object.

inline void set_xor (**const Bit_Set&** *bs*);

Determines the logical exclusive-or of *bs* and the bit set object, modifying the source with the result. This function invalidates the current position of the bit set object.

inline int value ();

Returns the value (zero-relative bit position) of the bit at the current position. If the current position is invalid, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (**ostream&** *os*, **const Bit_Set&** *bs*);

Overloads the output operator for a reference to a **Bit_Set** object to provide a formatted output capability for the class.

inline friend ostream& operator<< (**ostream&** *os*, **const Bit_Set*** *bs*);

Overloads the output operator for a reference to a **Bit_Set** object to provide a formatted output capability for the class.

Bit_Set Class Example

8.6 The following program manipulates sets of objects representing the names of various colors, similar to the previous **Set** example. However, this program utilizes enumerated types and a bit vector to represent the objects. Two bit sets with different elements are created and several set operations are then applied. The results of each operation are sent to the standard output.

```
1   #include <COOL/Bit_Set.h>                  // COOL Bit Set class
2   enum colors { RED=1, YELLOW, PINK, GREEN, ORANGE, PURPLE, BLUE };

3   static colors c_tbl[] = {RED, YELLOW, PINK, GREEN, ORANGE, PURPLE, BLUE};

4   int main (void) {
5     Bit_Set a, b;                            // Declare two bit set objects
6     for (int i = 0; i < 5; i++) {            // For each color defined
7                 a.put (c_tbl[i]);            // Add object to first set
8                 b.put (c_tbl[6-i]);          // Add end object to second set
9     }
10    cout << ”Set A contains: ” << a;         // Elements of set 1
11    cout << ”Set B contains: ” << b;         // Elements of set 2
12    cout << ”A | B: ” << (a | b);            // Display union
13    cout << ”A & B: ” << (a & b);            // Display intersection
14    cout << ”A ^ B: ” << (a ^ b);            // Display exclusive-or
15    cout << ”A – B: ” << (a – b);            // Display difference
16    return (0);                              // Exit with OK status
17  }
```

Line 1 includes the COOL `Bit_Set.h` class header file. Line 2 defines an enumerated color type and line 3 defines a static array of enumerated color values to be used as elements in the bit sets in the main program. Notice that the enumerated color type defined in line 2 begins with an initial value of 1. This insures that the bit set will behave correctly when the bit representing `RED` is set. Line 5 defines two bit set objects with default storage capacity. Lines 6 through 11 add some elements to each set and output the resulting objects. Lines 12 through 15 perform four set operations and display the results. Finally, the program ends with a successful completion code.

The following shows the output from the program:

```
Set A contains: [ 01111100 ]
Set B contains: [ 00011111 ]
A | B: [ 01111111 ]
A & B: [ 00011100 ]
A ^ B: [ 01100011 ]
A – B: [ 01100000 ]
```

9 NODE AND TREE CLASSES

Introduction

9.1 The node and tree classes implement several tree data structures as parameterized classes. The following classes are discussed in this section:

- **Binary_Node**<*Type*>
- **Binary_Tree**<*Type*>
- **AVL_Tree**<*Type*>
- **N_Node**<*Type,nchild*>
- **D_Node**<*Type,nchild*>
- **N_Tree**<*Node,Type,nchild*>

The **Binary_Node**<*Type*> class implements parameterized nodes for use by the **Binary_Tree**<*Type*> class, which in turn implements simple, dynamic, sorted sequences in a tree where each node has two subtree pointers. The **AVL_Tree**<*Type*> class implements height-balanced binary trees. The **N_Node**<*Type,nchild*> class implements static-size nodes for use by the n-ary tree class. The **D_Node**<*Type,nchild*> class implements dynamic-sized nodes for use by the n-ary tree class. The **N_Tree**<*Node,Type,nchild*> class implements n-ary trees, providing the organizational structure for a tree of nodes, but knowing nothing about the specific type of node used. **N_Tree**<*Node,Type,nchild*> is parameterized over a node type, a data type, and an initial subtree count. The **Binary_Tree**<*Type*>, **AVL_Tree**<*Type*>, and **N_Tree**<*Node, Type, nchild*> classes support the notion of a current position. See Section 5, Parameterized Templates, for more information regarding the current position mechanism and the **Iterator**<*Type*> class.

In order to achieve successful compilation and usage, there are certain operations that must be supported by any user-specified type over which a node or tree class is parameterized. The member functions **operator=**, **operator<**, **operator>**, **operator==**, and **operator<<** for both pointer and reference must be overloaded for any class object used as the type. Note that built-in types already have these functions defined.

NOTE: The node and tree classes use **operator=** of the parameterized type when copying elements. You should be careful when parameterizing a node or tree class over a pointer to a type, since the default pointer assignment operator usually copies the pointer, not the value pointed at.

Requirements

9.2 This section discusses the parameterized tree container classes. It assumes that you have read read and understood Section 5, Parameterized Templates. In addition, no attempt is made to discuss the concepts and algorithms for the data structures discussed. You should refer to a general data structures or computer science text for this information.

Binary_Node Class

9.3 The **Binary_Node**<*Type*> class implements parameterized nodes for binary trees. This class is privately derived from the **Binary_Node** class that contains left and right subtree pointers. The **Binary_Node**<*Type*> class adds a data member of the required type in the private section. Since the **Binary_Node**<*Type*> class is intended for use by the **Binary_Tree**<*Type*> class, the **Binary_Tree**<*Type*> class is declared a friend class.

Name: **Binary_Node**<*Type*> — Parameterized binary node class

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

Base Classes: **Binary_Node**

Friend Classes: **Binary_Tree**<*Type*>

Constructors:

Binary_Node<*Type*> ();
Allocates a binary node with left and right subtree pointers set to **NULL**.

Binary_Node<*Type*> (**const Binary_Node**<*Type*>& *bn*);
Duplicates the value of another binary node object *bn*.

Binary_Node<*Type*> (**const** *Type& value*);
Allocates a binary node with left and right subtree pointers set to **NULL** and initializes the value of the node to *value*.

Member Functions:

Binary_Node<*Type*>& **operator=** (**const Binary_Node**<*Type*>& *bn*);
Overloads the assignment operator to assign the values of the left and right subtree pointers and to assign the value in *bn* to the binary node object. This function returns a reference to the updated node.

inline Type& get () **const**;
Returns a reference to the value of the data member.

inline Binary_Node<*Type*>* **get_ltree** () **const**;
Returns a pointer to the left subtree.

inline Binary_Node<*Type*>* **get_rtree** () **const**;
Returns a pointer to the right subtree.

inline Boolean is_leaf () **const**;
Determines if the node is a terminal node by evaluating the left and right subtree pointers. If both are **NULL**, this function returns **TRUE**; otherwise, this function returns **FALSE**.

inline void set (**const** *Type& value*);
Sets the value of the data member in the node to *value*.

inline void set_ltree (**Binary_Node**<*Type*>* *bn*);
Sets the value of the left subtree pointer of the binary node object to *bn*.

inline void set_rtree (**Binary_Node**<*Type*>* *bn*);
Sets the value of the right subtree pointer of the binary node object to *bn*.

Binary_Tree Class

9.4 The **Binary_Tree**<*Type*> class implements simple, dynamic, sorted sequences. Users requiring a data structure for unsorted sequences whose structure and organization is more under the control of the programmer are referred to the **N_Tree** class. The **Binary_Tree**<*Type*> class is derived from **Binary_Tree** and is a friend of the **Binary_Node**<*Type*> class, also parameterized over the same *Type*. There is no attempt made to balance or prune the tree. Nodes are added to a particular subtree at the direction of the collating function. For example, a lopsided tree results if a tree is parameterized for integers and the tree uses the default integer comparison operators whose elements are added in increasing order. Likewise, a lopsided tree results after many items have been added and removed.

The **Binary_Tree**<*Type*> class implements the notion of a current position. This is useful for iterating through the nodes of a tree. The current position is maintained in a data member of type **Binary_Tree_state** and is set or reset by all member functions affecting elements in the class. Member functions are provided to reset the current position, move to the next and previous elements, find an element, and get the value at the current position. The **Iterator**<*Type*> class provides a mechanism to save and restore the state associated with the current position, thus allowing the programmer to use multiple iterators over the same instance of a tree.

Name: **Binary_Tree**<*Type*> — A parameterized binary tree class

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

Base Classes: **Binary_Tree**, **Generic**

Friend Classes: None

Public Constructors:

Binary_Tree<*Type*> ();
Allocates a binary tree object with the root pointer set to **NULL**.

Binary_Tree<*Type*> (**const Binary_Tree**<*Type*>& *bt*);
Duplicates the structure of another binary tree object *bt*.

Member Functions:

void balance ();
Builds a perfectly balanced binary tree from the existing tree structure and deletes the old tree and storage.

void clear ();
Empties the tree and deallocates all memory for nodes and internal structures.

inline long count () **const**;
Returns the number of nodes in the tree structure.

inline Binary_Tree_state& current_position ();
Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of binary tree.

Boolean find (**const** *Type& value*);
Searches for *value* in the tree structure. If found, this function updates the current position and returns **TRUE**; otherwise, this function invalidates the current position and returns **FALSE**.

Binary_Node<*Type*>*** get_root** () **const**;
Returns a pointer to the root node of the tree.

Boolean next ();

Advances the current position to the next element in the tree and returns **TRUE**. If the current position is invalid, this function sets the current position to the first element and returns **TRUE.** If the current position is the last element in the tree, this function invalidates the current position and returns **FALSE**.

Binary_Node<*Type*>*** node** ();

Returns a pointer to the current node object.

Binary_Tree<*Type*>& **operator=** (**Binary_Tree**<*Type*>& *bt*);

Overloads the assignment operator to duplicate another binary tree object *bt* by copying all nodes and value to the binary tree object. This function returns a reference to the updated binary tree object.

inline Boolean operator== (**const Binary_Tree**<*Type*>*& bt*) **const**;

Overloads the equality operator for the **Binary_Tree**<*Type*> class. This function returns **TRUE** if *bt* is equal to the binary tree object; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Binary_Tree**<*Type*>*& bt*) **const**;

Overloads the inequality operator for the **Binary_Tree**<*Type*> class. This function returns **TRUE** if *bt* is unequal to the binary tree object; otherwise, this function returns **FALSE**.

Boolean prev ();

Moves the current position to the previous element in the tree and returns **TRUE**. If the current position is invalid, this function sets the current position to the last element and returns **TRUE**, thus facilitating reverse traversal through the tree. If the current position is the first element in the tree, this function invalidates the current position and returns **FALSE**.

inline Boolean put (**const** *Type& value*);

Adds *value* to the tree structure if not already present. This function returns **TRUE** if the item is added; otherwise, this function returns **FALSE**. This function invalidates the current position.

inline Boolean remove ();

Removes the node at the current position from the tree structure and returns **TRUE**. This function invalidates the current position of the binary tree object. If the current position is out of range, an **Error** exception is raised and this function returns **FALSE**.

inline Boolean remove (**const** *Type& value*);

Removes *value* from the tree structure if present. This function returns **TRUE** if the specified argument is successfully removed; otherwise, this function returns **FALSE**. This function invalidates the current position.

inline void reset ();

Invalidates the current position of the binary tree object.

inline void set_compare (*Binary_Tree_Compare* **= NULL**);

Sets the comparison function that is to be used in all logical comparison tests. *Binary_Tree_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the class is parameterized is used.

inline long tree_depth ();
Returns the zero-relative depth of the tree structure. Note that this function is potentially very expensive, since the tree depth is calculated by traversing all nodes in the tree.

Type& value ();
Returns a reference to the node value at the current position. If the current position is invalid, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (**ostream&** *os*, **const Binary_Tree**<*Type*>& *bt*);
Accepts a binary tree reference and outputs the structure by printing it sideways, where the root is printed at the left margin. To obtain the standard orientation, rotate the output 90 degrees clockwise. This function returns a reference to the output stream.

friend ostream& operator<< (**ostream&** *os*, **const Binary_Tree**<*Type*>* *bt*);
Accepts a binary tree pointer and outputs the structure by printing it sideways, where the root is printed at the left margin. To obtain the standard orientation, rotate the output 90 degrees clockwise. This function returns a reference to the output stream.

Binary_Tree Example

9.5 The following program processes the words in a character string using the regular expression feature of the **Gen_String** class as was done for the examples in Section 7. Each unique word is then converted to uppercase and added to the binary tree.

```
#include <COOL/Binary_Tree.h>              // Include Binary tree class
#include <COOL/String.h>                   // Include COOL String class
#include <COOL/Gen_String.h>               // Include COOL Gen_String class

static Gen_String text ("\n\
A programming language serves two related purposes: it provides a\n\
vehicle for the programmer to specify actions to be executed and a\n\
set of concepts for the programmer to use when thinking about what\n\
can be done.");

DECLARE Binary_Tree<String>                // Declare tree type
IMPLEMENT Binary_Tree<String>              // Implement tree type

int main (void) {
  Binary_Tree<String> bt1;                 // Declare tree variable
  Gen_String s;                            // Temporary string variable
  text.compile ("[a-zA-Z]+");              // Match any alphabetical word
  while (text.find ()) {                   // While still more words
   text.sub_string (s, text.start (), text.end ()); // Get word
   bt1.put (*(new String(uppcase (s))));   // And add to tree
  }
  cout << bt1;                             // Output tree structure
  exit (0);                                // Exit with successful status
}
```

Lines 1 through 3 include the COOL `Binary_Tree.h`, `String.h`, and `Gen_String.h` header files. Line 4 defines a static character string containing the first sentence of the paragraph in section 7 quoted from Stroustrup. Lines 5 and 6 declare and implement the binary tree type containing **String** objects. Line 8 declares a **Binary_Tree** object and line 9 declares a temporary string variable. A regular expression to match sequences of alphabetical characters (that is, words) is compiled in line 10. Lines 11 through 14 contain a loop that finds each word in the paragraph and adds it to the binary tree. Note the use of `operator new()` to create a new **String** object for each item stored in the tree. Line 15 outputs a representation of the structure of the **Binary_tree** object rotated 90∇ counter clockwise. Finally, the program ends with a valid exit code on line 16.

The following shows the output for the program:

```
               WHEN
                  WHAT
            VEHICLE
               USE
         TWO
               TO
                  THINKING
            THE
               SPECIFY
                  SET
      SERVES
         RELATED
            PURPOSES
               PROVIDES
   PROGRAMMING
         PROGRAMMER
            OF
      LANGUAGE
         IT
            FOR
                     EXECUTED
                           DONE
                        CONCEPTS
                           CAN
                  BE
                     AND
               ACTIONS
                  ABOUT
A
```

Each unique string added to the binary tree is inserted at a node such that all strings contained in the left subtree of the node are lexically less than (that is, come before) the string. All strings contained in the right subtree of the node are lexically greater than (that is, come after) the string. Thus, the order in which items are added to the tree significantly alter its internal structure.

AVL_Tree Class

9.6 The **AVL_Tree**<*Type*> class implements height-balanced, dynamic, binary trees. The **AVL_Tree**<*Type*> class is publicly derived from the **Binary_Tree**<*Type*> class, and both are parameterized over some *Type*. An AVL tree is a compromise between the expense of a fully balanced binary tree and the desire for efficient search times for both average and worst-case scenarios. As a result, an AVL tree maintains a binary tree that is height-balanced, ensuring that the difference between the depth of the left subtree and right subtree for every node is no more than one.

The **AVL_Tree**<*Type*> class implements the notion of a current position. This is useful for iterating through the nodes of a tree. The current position is maintained in a data member of type **AVL_Tree_state** and is set or reset by all member functions affecting elements in the class. Member functions are provided to reset the current position, move to the next and previous elements, find an element, and get the value at the current position. The **Iterator**<*Type*> class provides a mechanism to save and restore the state associated with the current position, thus allowing the programmer to use multiple iterators over the same instance of a tree.

The **AVL_Tree**<*Type*> class inherits all its member functions publicly from the **Binary_Tree**<*Type*> class. The only member functions that are overloaded are those that affect the structure of the tree, thus potentially requiring one or more subtrees to be restructured.

Name: **AVL_Tree**<*Type*> — A parameterized, height-balanced binary tree class

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

Base Classes: **Binary_Tree**<*Type*>

Friend Classes: None

Public Constructors:

AVL_Tree<*Type*> ()
Simple constructor to create an empty tree.

AVL_Tree<*Type*> (**const Binary_Tree**<*Type*>& *bt*);
Duplicates a **Binary_Tree** object *bt*, adjusting the organization and structure as necessary to create an AVL tree.

AVL_Tree<*Type*> (**const AVL_Tree**<*Type*>& *at*);
Duplicates the structure of another **AVL_Tree**<*Type*> object *at*.

Member Functions:

void balance ();
Builds a perfectly balanced binary tree from the existing tree structure and deletes the old tree and storage.

void clear ();
Empties the tree and deallocates all memory for nodes and internal structures.

inline long count () **const**;
Returns the number of nodes in the tree structure.

inline AVL_Tree_state& current_position ();
Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus allowing multiple iterators over an instance of a binary tree.

Boolean find (**const** *Type& value*);
Searches for *value* in the tree structure. If found, this function updates the current position and returns **TRUE**; otherwise, this function invalidates the current position and returns **FALSE**.

inline Binary_Node* get_root () **const**;
Accesses the root pointer for the tree structure.

Boolean next ();
Advances the current position to the next element in the tree and returns **TRUE**. If the current position is invalid, this function sets the current position to the first element and returns **TRUE**. If the current position is the last element of the tree, this function invalidates the current position and returns **FALSE**.

Binary_Node<*Type*>*** node** ();
Returns a pointer to the current node object.

AVL_Tree<*Type*>& **operator=** (**Binary_Tree**<*Type*>& *bt*);
Overloads the assignment operator to create an AVL tree from a binary tree object *bt*. This function returns a reference to the updated AVL tree object.

AVL_Tree<*Type*>& **operator=** (**AVL_Tree**<*Type*>& *at*);
Overloads the assignment operator to duplicate another AVL tree object *at*. This function returns a reference to the updated AVL tree object.

inline Boolean operator== (**const AVL_Tree**<*Type*>*& at*) **const**;
Overloads the equality operator for the **AVL_Tree**<*Type*> class. This function returns **TRUE** if *at* is equal to the AVL tree object; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const AVL_Tree**<*Type*>*& at*) **const**;
Overloads the inequality operator for the **AVL_Tree**<*Type*> class. This function returns **TRUE** if *at* is unequal to the AVL tree object; otherwise, this function returns **FALSE**.

Boolean prev ();
Moves the current position to the previous element in the tree and returns **TRUE**. If the current position is invalid, this function sets the current position to the last element and returns **TRUE**, thus facilitating reverse traversal through the tree. If the current position is the first element in the object, this function invalidates the current position and returns **FALSE.**

Boolean put (**const** *Type& value*);
Adds the value passed to the tree structure if not already present. This function returns **TRUE** if the item is added; otherwise, this function returns **FALSE**. This function invalidates the current position and balances the tree structure if necessary.

inline Boolean remove ();
Removes the node at the current position from the tree structure and returns **TRUE**. This function invalidates the current position and balances the tree structure if necessary. If the current position is out of range, an **Error** exception is raised and this function returns **FALSE**.

Boolean remove (**const** *Type& value*);
Searches for the specified node. If the node is found, this function removes the node and sets the current position to the node immediately following the node removed; then the function returns **TRUE**. If the node is found at the end of the tree structure, this function invalidates the current position, balances the tree structure if necessary, and returns **TRUE**. If the node is not found, this function returns **FALSE.**

inline void reset ();
Invalidates the current position pointer and deallocates the memory allocated for the node cache.

inline void set_compare (*AVL_Tree_Compare* = **NULL**);
Sets the comparison function that is to be used in all comparison tests. *AVL_Tree_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the class is parameterized is used.

inline Type& value ();
Returns a reference to the value of the node at the current position. If the current position is invalid, an **Error** exception is raised.

inline long tree_depth ();
Returns the zero-relative depth of the tree structure. Note that this function is potentially very expensive, since the tree depth is calculated by traversing all nodes in the tree.

Friend Functions:

friend ostream& operator<< (**ostream&** *os*, **AVL_Tree**<*Type*>& *at*);
Accepts an AVL tree reference and outputs the structure by printing it sideways, where the root is printed at the left margin. To obtain the standard orientation, rotate the output 90 degrees clockwise. This function returns a reference to the output stream.

friend ostream& operator<< (**ostream&** *os*, **AVL_Tree**<*Type*>* *at*);
Accepts an AVL tree pointer and outputs the structure by printing it sideways, where the root is printed at the left margin. To obtain the standard orientation, rotate the output 90 degrees clockwise. This function returns a reference to the output stream.

AVL Tree Example

9.7 The following program processes the words in a character string using the regular expression features of the **Gen_String** class as was done for the examples in Section 7. Each unique word is then converted to uppercase and added to the binary tree. This is the same as the previous example except that an AVL tree is used, resulting in the creation of a height-balanced tree.

```
#include <COOL/AVL_Tree.h>                   // Include AVL tree class
#include <COOL/String.h>                     // Include COOL String class
#include <COOL/Gen_String.h>                 // Include COOL Gen_String class

static Gen_String text ("\n\
A programming language serves two related purposes: it provides a\n\
vehicle for the programmer to specify actions to be executed and a\n\
set of concepts for the programmer to use when thinking about what\n\
can be done.");

DECLARE AVL_Tree<String>                     // Declare tree type
IMPLEMENT AVL_Tree<String>                   // Implement tree type

int main (void) {
  AVL_Tree<String> avl1;                     // Declare tree variable
  Gen_String s;                              // Temporary string variable
  text.compile ("[a-zA-Z]+");                // Match any alphabetical word
  while (text.find ()) {                     // While still more words
   text.sub_string (s, text.start (), text.end ()); // Get word
   avl1.put (*(new String(uppcase (s))));    // And add to tree
  }
  cout << avl1;                              // Output tree structure
  exit (0);                                  // Exit with successful status
}
```

Lines 1 through 3 include the COOL `AVL_Tree.h`, `String.h`, and `Gen_String.h` header files. Line 4 defines a static character string containing the first sentence of the paragraph in section 7 quoted from Stroustrup. Lines 5 and 6 declare and implement the AVL tree type containing **String** objects. Line 8 declares an **AVL_Tree** object and line 9 declares a temporary string variable. A regular expression to match sequences of alphabetical characters (that is, words) is compiled in line 10. Lines 11 through 14 contain a loop that finds each word in the paragraph and adds it to the AVL tree. Note the use of `operator new()` to create a new **String** object for each item stored in the tree. Line 15 outputs a representation of the structure of the **AVL_tree** object rotated 90∇ counter clockwise. Finally, the program ends with a valid exit code on line 16.

The following shows the output for the program:

```
       WHEN
          WHAT
     VEHICLE
          USE
       TWO
          TO
  THINKING
          THE
       SPECIFY
          SET
     SERVES
          RELATED
       PURPOSES
          PROVIDES
PROGRAMMING
       PROGRAMMER
          OF
     LANGUAGE
          IT
       FOR
          EXECUTED
  DONE
          CONCEPTS
       CAN
          BE
     AND
          ACTIONS
       ABOUT
          A
```

Each unique string added to the AVL tree is inserted at a node such that all strings contained in the left subtree of the node are lexically less than (that is, come before) the string. All strings contained in the right subtree of the node are lexically greater than (that is, come after) the string. However, unlike a binary tree, an AVL tree guarantees to maintain a balanced structure. Consequently, the order in which items are added to the tree has no bearing upon its internal structure.

N_Node Class

9.8 The **N_Node**<*Type*,*nchild*> class implements parameterized nodes of a static size for n-ary trees. This node class is parameterized for both the type and some initial number of subtrees that each node may have. The constructors for the **N_Node**<*Type*,*nchild*> class are declared in the public section to allow the user to create nodes and control the building and structure of an n-ary tree where the ordering can have a specific meaning, as with an expression tree.

Name:	**N_Node**<*Type*,*nchild*> — Parameterized static-sized n-ary node class
Synopsis:	**#include** <COOL/N_Node.h>
Base Classes:	None
Friend Classes:	**N_Tree**<*Node*,*Type*,*nchild*>

Public Constructors:

N_Node<*Type*,*nchild*> ();
Allocates an N-node with all subtree pointers set to **NULL**.

N_Node<*Type*,*nchild*> (**const** *Type& value*);
Allocates an N-node with all subtree pointers set to **NULL** and initializes the value of the node to *value*.

N_Node<*Type*,*nchild*> (**const N_Node**<*Type*,*nchild*>& *nn*);
Duplicates the value of another N-node object *nn*.

Member Functions:

inline Type& get () **const**;
Returns a reference to the value of the data member.

Boolean insert_after (**N_Node**<*Type*,*nchild*>& *nn*, **int** *index*);
Inserts a subtree pointer to *nn* after the zero-relative *index* given. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. If *index* is negative or out of range, an **Error** exception is raised.

Boolean insert_before (**N_Node**<*Type*,*nchild*>& *nn*, **int** *index*);
Inserts a subtree pointer to *nn* before the zero-relative *index*. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. If *index* is negative or out of range, an **Error** exception is raised.

Boolean is_leaf () **const**;
Determines if the node is a terminal node by evaluating the subtree pointers. If all pointers are **NULL**, this function returns **TRUE**; otherwise, this function returns **FALSE**.

inline int num_subtrees () **const**;
Returns the maximum number of subtrees possible for a node.

N_Node<*Type*,*nchild*>& **operator=** (**N_Node**<*Type*,*nchild*>& *nn*);
Overloads the assignment operator for the class to assign the values of the subtree pointers and the value in *nn* to the node object. This function returns a reference to the updated node.

N_Node<*Type*,*nchild*>& **operator=** (**N_Node**<*Type*,*nchild*>* *nn*);
Overloads the assignment operator for the class to assign the values of the subtree pointers and the value in *nn* to the node object. This function returns a reference to the updated node.

inline Boolean operator== (**const** *Type& value*) **const**;
Overloads the equality operator for the **N_Node**<*Type*> class. This function returns **TRUE** if *value* is equal to the value of the node object; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const** *Type& value*) **const**;
Overloads the inequality operator for the **N_Node**<*Type*> class. This function returns **TRUE** if *value* is not equal to the value of the node object; otherwise, this function returns **FALSE**.

inline Boolean operator< (**const** *Type& value*) **const**;
Overloads the less-than operator for the **N_Node**<*Type*> class. This function returns **TRUE** if *value* is less than the value of the node object; otherwise, this function returns **FALSE**.

inline Boolean operator<= (**const** *Type& value*) **const**;
Overloads the less-than-or-equal operator for the **N_Node**<*Type*> class. This function returns **TRUE** if *value* is less than or equal to the value of the node object; otherwise, this function returns **FALSE**.

inline Boolean operator> (**const** *Type& value*) **const**;
Overloads the greater-than operator for the **N_Node**<*Type*> class. This function returns **TRUE** if *value* is greater than the value of the node object; otherwise, this function returns **FALSE**.

inline Boolean operator>= (**const** *Type& value*) **const**;
Overloads the greater-than-or-equal operator for the **N_Node**<*Type*> class. This function returns **TRUE** if *value* is greater than or equal to the value of the node object; otherwise, this function returns **FALSE**.

inline N_Node<*Type*,*nchild*>*& **operator[]** (**int** *index*);
Returns a reference to a pointer to the subtree at the zero-relative *index*. If *index* is negative or out of range, an **Error** exception is raised.

inline void set (**const** *Type& value*);
Sets the value of the data member in the node to *value*.

inline void set_compare (*N_Node_Compare* = **NULL**);
Sets the comparison function that is to be used in all comparison tests. *N_Node_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the class is parameterized is used.

D_Node Class

9.9 The **D_Node**<*Type*,*nchild*> class implements parameterized nodes of a dynamic size for n-ary trees. This node class is parameterized for the type and some initial number of subtrees that each node may have. The **D_Node**<*Type*,*nchild*> class is dynamic in the sense that the number of subtrees allowed for each node is not fixed. **D_Node**<*Type*,*nchild*> uses the **Vector**<*Type*> class, which supports run-time growth characteristics. As a result, the **D_Node**<*Type*,*nchild*> class should be used as the node type for the **N_Tree**<*Node*,*Type*,*nchild*> class when the number of subtrees is variable, unknown at compile time, or needs to increase on a per-node basis at run-time. This capability is suited for hierarchical trees such as may be used in an organization chart. Also, specialization of the **N_Tree**<*Node*,*Type*,*nchild*> class would allow for relatively easy implementation of a DAG class.

Name:	**D_Node**<*Type*,*nchild*> — Parameterized, dynamic-size n-ary node class
Synopsis:	**#include** <COOL/D_Node.h>
Base Classes:	None
Friend Classes:	**N_Tree**<*Node*,*Type*,*nchild*>
Public Constructors:	**D_Node**<*Type*,*nchild*> ();

Allocates a D-node and a vector of subtree pointers of the initial size, all of which are set to **NULL**.

D_Node<*Type*,*nchild*> (**const** *Type& value*);
Allocates a D-node and a vector of subtree pointers of the initial size, all of which are set to **NULL,** and initializes the value of the node to *value*.

D_Node<*Type*,*nchild*> (**const D_Node**<*Type*,*nchild*>& *dn*);
Duplicates the value of another D-node object *dn*.

Member Functions:

inline Type& get () **const**;
Returns a reference to the value of the data member.

Boolean insert_after (**D_Node**<*Type*,*nchild*>& *dn*, **int** *index*);
Inserts a subtree pointer to *dn* after the zero-relative *index*. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. If *index* is negative or out of range, an **Error** exception is raised.

Boolean insert_before (**D_Node**<*Type*,*nchild*>& *dn*, **int** *index*);
Inserts a subtree pointer to *dn* before the zero-relative *index*. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. If *index* is negative or out of range, an **Error** exception is raised.

Boolean is_leaf () **const**;
Determines if the node is a terminal node by evaluating the subtree pointers. If all pointers are **NULL**, this function returns **TRUE**; otherwise, this function returns **FALSE**.

inline int num_subtrees () **const**;
Returns the number of subtrees for a node.

D_Node<*Type*,*nchild*>& **operator=** (**D_Node**<*Type*,*nchild*>& *dn*);
Overloads the assignment operator for the class to assign the values of the subtree pointers and the value in *dn* to the node object. This function returns a reference to the updated node.

D_Node<*Type*,*nchild*>& **operator=** (**D_Node**<*Type*,*nchild*>* *dn*);
Overloads the assignment operator for the class to assign the values of the subtree pointers and the value in *dn* to the node object. This function returns a reference to the updated node.

inline Boolean operator== (**const** *Type& value*) **const**;
Overloads the equality operator for the **D_Node**<*Type*> class. This function returns **TRUE** if *value* is equal to the value of the node object; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const** *Type& value*) **const**;
Overloads the inequality operator for the **D_Node**<*Type*> class. This function returns **TRUE** if *value* is not equal to the value of the node object; otherwise, this function returns **FALSE**.

inline Boolean operator< (**const** *Type& value*) **const**;
Overloads the less-than operator for the **D_Node**<*Type*> class. This function returns **TRUE** if *value* is less than the value of the node object; otherwise, this function returns **FALSE**.

inline Boolean operator<= (**const** *Type& value*) **const**;
Overloads the less-than-or-equal operator for the **D_Node**<*Type*> class. This function returns **TRUE** if *value* is less than or equal to the value of the node object; otherwise, this function returns **FALSE**.

inline Boolean operator> (**const** *Type& value*) **const**;
Overloads the greater-than operator for the **D_Node**<*Type*> class. This function returns **TRUE** if *value* is greater than the value of the node object; otherwise, this function returns **FALSE**.

inline Boolean operator>= (**const** *Type& value*) **const**;
Overloads the greater-than-or-equal operator for the **D_Node**<*Type*> class. This function returns **TRUE** if *value* is greater than or equal to the value of the node object; otherwise, this function returns **FALSE**.

inline D_Node<*Type,nchild*>*& **operator[]** (**int** *index*);
Returns a reference to a pointer to the subtree at the zero-relative *index*. If *index* is negative or out of range, an **Error** exception is raised.

inline void set (**const** *Type& value*);
Sets the value of the data member in the node to *value*.

inline void set_compare (*D_Node_Compare* = **NULL**);
Sets the comparison function that is to be used in all comparison tests. *D_Node_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the class is parameterized is used.

N_Tree Class

9.10 The **N_Tree**<*Node,Type,nchild*> class implements n-ary trees, providing the organizational structure for a tree (collection) of nodes, but knowing nothing about the specific type of node used. **N_Tree**<*Node,Type,nchild*> is parameterized over a node type, a data type, and a subtree count, where the node specified must have a data member of the same *Type* as the tree class and the subtree count indicates the number of possible subtree pointers (children) from any given node. Two node classes are provided, but others could also be written. The **N_Node**<*Type*> class implements static-sized nodes for some distinct number of subtrees, and the **D_Node**<*Type*> class implements dynamic-sized nodes derived from the **Vector**<*Type*> class.

Since the organization of a tree is important (as with an expression tree), the user must supervise the construction of the tree by directing specific node and subtree assignments and layout. No attempt is made by the **N_Tree**<*Node,Type,nchild*> class to balance or prune the tree.

The **N_Tree**<*Node,Type,nchild*> class implements the notion of a current position. This is useful for iterating through the nodes of a tree. The current position is maintained in a data member of type **N_Tree_state** and is set or reset by all member functions affecting elements in the class. Member functions are provided to reset the current position, move to the next and previous elements, find an element, and get the value at the current position. The **Iterator**<*Type*> class provides a mechanism to save and restore the state associated with the current position, thus allowing the programmer to use multiple iterators over the same instance of a tree.

Traversal through an n-ary tree using the current position mechanism and the **Iterator**<*Type*> class can be controlled by setting the traversal mode. An enumerated type *Traversal_Type* is defined for the following values:

- `PREORDER`
- `PREORDER_REVERSE`
- `INORDER`

- `INORDER_REVERSE`
- `POSTORDER`
- `POSTORDER_REVERSE`

Inorder traversal for an n-ary tree is defined to traverse the left-most subtree, visit the node, then traverse all remaining subtrees from left to right. Postorder traversal of an n-ary tree is defined to traverse all subtrees from left to right, then visit the node. Preorder traversal for an n-ary tree is defined to visit the node, then traverse all subtrees from left to right. The reverse traversal modes are similar, except that they visit subtrees from right to left.

Name: **N_Tree**<*Node*,*Type*,*nchild*> — A parameterized N-ary tree class

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

Base Classes: **Generic**

Friend Classes: None

Constructors:

N_Tree<*Node*,*Type*,*nchild*> (**Node**<*Type*,*nchild*>& *n*);
Allocates an n-ary tree object with the root pointer set to *n*.

N_Tree<*Node*,*Type*,*nchild*> (**Node**<*Type*,*nchild*>* *n*);
Allocates an n-ary tree object with the root pointer set to *n*.

N_Tree<*Node*,*Type*,*nchild*> (**const N_Tree**<*Node*,*Type*,*nchild*>& *nt*);
Duplicates the structure of another n-ary tree object *nt*.

Member Functions:

void clear ();
Empties the tree and deallocates the memory for all nodes and internal structures.

inline long count () **const**;
Returns the number of nodes in the tree structure. Note that this function is potentially very expensive, since the tree depth is calculated by traversing all nodes in the tree.

inline long current_depth ();
Returns the zero-relative depth in the n-ary tree object of the node at the current position. If the current position is invalid, this function returns zero.

inline N_Tree_state& current_position ();
Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of n-ary tree.

Boolean find (**const** *Type& value*);
Searches for *value* in the tree. If found, this function updates the current position and returns **TRUE**; otherwise, this function invalidates the current position and returns **FALSE**.

void inorder (*Node_Apply_Function fn*);
Performs an in-order traversal of the tree structure and applies the function *fn* to each node. Inorder traversal for an n-ary tree is defined to traverse the left-most subtree, visit the node, then traverse all remaining subtrees from left to right. *Node_Apply_Function* is a function of type **Boolean** (**Function*)(**const** *Type& value)* where *value* is the value from each node visited that is substituted during the traversal.

inline Boolean next ();
Advances the current position to the next element if there is one. This function returns **TRUE** if successful. If the current position is invalid, this function sets the current position to the first element and returns **TRUE**. If the current position is the last element in the tree, this function invalidates the current position and returns **FALSE**.

inline Node<*Type*,*nchild*>***& operator[]** (**int** *index*);
Returns a reference to a pointer to the zero-relative indexed subtree. If *index* is negative or out of range, an **Error** exception is raised.

inline operator Node<*Type*,*nchild*>();
Provides an implicit conversion operator from an n-ary tree object to the node over which the class is parameterized.

void postorder (*Node_Apply_Function fn*);
Performs a post-order traversal of the tree structure and applies the function *fn* to each node. Postorder traversal of an n-ary tree is defined to traverse all subtrees from left to right, then visit the node. *Node_Apply_Function* is a function of type **Boolean** (**Function*)(**const** *Type& value)* where *value* is the value from each node visited that is substituted during the traversal.

void preorder (*Node_Apply_Function fn*);
Performs a pre-order traversal of the tree structure and applies the function *fn* to each node. Preorder traversal for an n-ary tree is defined to visit the node, then traverse all subtrees from left to right. *Node_Apply_Function* is a function of type **Boolean** (**Function*)(**const** *Type& value)* where *value* is the value from each node visited that is substituted during the traversal.

Boolean prev ();
Moves the current position to the previous element in the tree and returns **TRUE**. If the current position is invalid, this function sets the current position to the last element and returns **TRUE**. If the current position is the first element in the tree, this function invalidates the current position and returns **FALSE.**

inline void reset ();
Invalidates the current position for the n-ary tree object.

inline Traversal_Type& traversal ();
Returns a reference to the traversal mode. This member function can be used to set or get the current traversal mode.

Type& value ();
Returns a reference to the value of the node at the current position. If the current position is invalid, an **Error** exception is raised.

N_Tree Example

9.11 Unlike the **Binary_Tree**<*Type*> and **AVL_Tree**<*Type*> classes discussed earlier in this section, the **N_Tree**<*Node*,*Type*,*nchild*> requires the user to direct the construction and control the structure of the tree. The following program requires this flexibility and uses dynamic nodes and the n-ary tree class to create, and then navigate through a hypothetical organizational chart.

```
1   #include <COOL/D_Node.h>                    // Include node class
2   #include <COOL/N_Tree.h>                    // Include n-ary tree class
3   #include <COOL/String.h>                    // Include string class

4   DECLARE N_Tree<D_Node,String,3>             // Declare tree type
5   IMPLEMENT N_Tree<D_Node,String,3>           // Implement tree type

6   int main (void) {
7    D_Node<String,3> president (String(”President”));     // Create president
8    N_Tree<D_Node,String,3> org_chart (president);        // Setup top of tree
9    D_Node<String,3> sales (String(”Sales”));             // Create sales
10   D_Node<String,3> service (String(”Service”));         // Create service
11   D_Node<String,3> finance (String(”Finance”));         // Create finance
12   D_Node<String,3> legal (String(”Legal”));             // Create legal
13   president[0] = &sales;                                // Add sales to chart
14   president.insert_after(service, 0);                   // Add service to chart
15   president.insert_after(finance, 1);                   // Add finance to chart
16   president.insert_after(legal, 2);                     // Add legal to chart
17   sales[0] = new D_Node<String,3> (String(”Domestic”)); // Domestics sales
18   D_Node<String,3> international (String(”International”)); // International
19   sales.insert_after(international, 0);
20   international[0] = new D_Node<String,3> (String(”Asia”));
21   international.insert_after(*(new D_Node<String,3> (String(”Europe”))), 0);
22   international.insert_after(*(new D_Node<String,3> (String(”Africa”))), 1);
23   finance[0] = new D_Node<String,3> (String(”Short Term”));
24   finance.insert_after(*(new D_Node<String,3> (String(”Long Term”))), 0);
25   finance.insert_after(*(new D_Node<String,3> (String(”Collections”))), 1);
26   org_chart.traversal() = PREORDER;                     // Set traversal mode
27   for (org_chart.reset (); org_chart.next (); ) {       // For each node in tree
28    for (int i = 0; i < org_chart.current_depth (); i++)  // Indent level
29     cout << ”  ”;
30    cout << org_chart.value () << ”\n”;                  // Print value of node
31   }
32   return (0);                                           // Return success
33  }
```

Lines 1 through 3 include the **D_Node**<*Type*>, **N_Tree**<*Node*,*Type*,*nchild*> and **String** class header files. Lines 4 and 5 declare an n-ary tree class using dynamic nodes that initially support three subtrees per node and whose value is a **String**. Line 7 creates the top level node to represent the president in the organization. Line 8 creates the `org_chart` tree object and establishes the `president` node as the root. Lines 9 through 12 create the four department nodes. Notice that the dynamic nature of **D_Node** is automatically used since we add four departments, but initially parameterized the class for three subtrees per node. Line 13 adds `sales` as the first node under `president`. Lines 14 through 16 then add the `service`, `finance`, and `legal` nodes. Lines 17 and 18 establish the `Domestic` and `International` nodes under `sales`, and lines 19 through 22 setup the international distribution nodes. Similarly, lines 23 through 25 setup the finance nodes. Line 26 establishes the `PREORDER` traversal mode for the n-ary tree object.

Lines 27 through 31 contain a loop that uses the current position mechanism to iterate through the nodes of the tree. For each node, the indentation for the output is determined by its depth in the tree. After the indentation is printed on the output stream, the value of the node (the **String** value) is also printed. Finally, when all nodes in the tree have been visited, the program ends with a successful completion code.

The following shows the output for the program:

```
President
  Sales
    Domestic
    International
      Asia
      Europe
      Africa
  Service
  Finance
    Short Term
    Long Term
    Collections
  Legal
```

MACROS 10

Introduction

10.1 The COOL macro facility is an extension to the standard ANSI C macro preprocessing functions available with the **#define** statement. The COOL preprocessor is a modified ANSI C preprocessor that allows a programmer to define powerful extensions to the C++ language in an unobtrusive manner. This enhanced preprocessor is portable and compiler-independent, and can execute arbitrary-filter programs or macro expanders on C++ code fragments. It is important to note, however, that once a macro is expanded, the resulting code is conventional C++ 2.0 syntax acceptable to any conforming C++ translator or compiler.

The COOL macro facilities have many components. Macros such as those that support parameterized templates are implementations of theoretical design papers published by Bjarne Stroustrup. Others provide significant language features and enhanced power for the programmer heretofore unavailable with conventional C++ implementations.

This section provides information on the COOL macro facility that forms the basis for many of the advanced features covered in later sections. The following topics are discussed in this section:

- COOL preprocessor
- **defmacro**
- **MACRO**
- Example COOL macros

Requirements

10.2 This section discusses the macro facilities of COOL. It assumes that you have a working knowledge of the C++ language and are familiar with the concept of macros and macro expansion as found in the standard C preprocessor.

COOL Preprocessor

10.3 The COOL preprocessor is supplied as part of the library and is the point at which all language and computing enhancements available in COOL are implemented. The proposed draft ANSI C standard indicates that extensions and changes to the language or features implemented in a preprocessor or compiler should be made by using the **#pragma** statement. The COOL preprocessor follows this recommendation and uses this to make all macro extensions. The **#pragma defmacro** statement is the single hook through which features such as the class macro, parameterized templates, and polymorphic enhancements have been implemented.

Porting COOL to a new platform or operating system starts with the preprocessor. The preprocessor contains support for the **defmacro** statement and also implements several important macros internally for efficiency and performance considerations. These include **template**, **class**, **DEFPACKAGE**, and **DEFPACKAGE_SYMBOL**.

The COOL preprocessor is derived from and based upon the DECUS ANSI C preprocessor made available by the DEC User's group in the public domain and supplied on the X11R3 source tape from MIT. It complies with the draft ANSI C specification with the exception that trigraph sequences are not implemented. In addition to support for COOL macro processing discussed above, the preprocessor has several new command line options to support C++ comments. These command line options also have include-file debugging aids.

Name: **ccpp** — The COOL C/C++ preprocessor

Synopsis: **ccpp** [–*options*] [infile [outfile]]

Options:

–B
Recognizes the C++ double slash (//) comment character and treats all characters following up to the next newline character as commentary text.

–C
If set, source-file comments are written to the output file. This allows **ccpp** output to be used as input to a program such as **lint(1)** that expects comments to be specially formatted.

–D*name***[=value]**
Defines *name* as if the programmer had defined it in the program. If no value is provided, a default value of 1 is used.

–E
Always returns a successful status completion code to the operating system, even if errors were detected.

–I*directory*
Adds the specified directory to the list of directories searched when looking for an include file. Note that there is no space between the option letter and the directory name.

–U*name*
Undefines *name* as if the programmer had undefined it in the program.

–X[*number***]**
Enables debugging output from the preprocessor. A value of 1 for *number* will cause the pathname of each included file to be sent to the standard error stream. A value of 2 for *number* will cause **#control** statements to be inserted as comments in the output. A value of 3 for *number* will enable both debugging modes. If no value for *number* is provided, a default value of 1 is used. Note that this option is designed to be a debugging aid for use when the preprocessor is run as stand alone and not when invoked by the control program. Other values for *number* are ignored.

defmacro

10.4 The **#pragma defmacro** statement is implemented in the COOL preprocessor and is the single hook through which features such as the class macro, parameterized templates, and polymorphic enhancements have been implemented. The **defmacro** facility provides a way to execute arbitrary-filter programs on C++ code fragments passing through the preprocessor. When a **defmacro** style macro name is found, the name and everything until the delimiter (including all matching {} [] () <> “” ‘’ and comments found along the way) is piped onto the standard input stream of the indicated program or filter procedure. The procedure’s standard output is scanned by the preprocessor for further processing. The expansion replaces the macro call and is passed onto the compiler for parsing.

The implementation of a **defmacro** can be either external to the preprocessor (as in the case of files and programs) or internal to the preprocessor. For example, the **template, declare,** and **implement** macros that implement parameterized types is internal to the preprocessor to provide a more efficient implementation. The **defmacro** facility first searches for a file or program in the same search path as that used for include files. If a match is not found, an internal preprocessor table is searched. If a match is still not found, the error message `"Error: Cannot open macro file [`*xxx*`]"` is sent to the standard error stream where *xxx* is the name as it appears in the source code. The fundamental COOL macros are defined with **defmacro** in the header file <COOL/misc.h>, which is included by all COOL C++ source files.

Name: **defmacro** — The COOL C/C++ preprocessor extension mechanism

Synopsis:
#pragma defmacro *name <file> options*
#pragma defmacro *name* “file” *options*
#pragma defmacro *name program options*

name	A character string identifying the macro
file	The name of a file implementing the macro
program	The name of a filter program implementing the macro
options	One or more of the following space-separate parameters:

recursive
When present, the macro may be recursively expanded.

expanding
When present, input to the macro is macro-expanded.

delimiter=*c*
The default delimiter ‘;’ is replaced with *c*.

condition=*c*
When present, the macro will not be invoked unless followed by *c*.

REST: *args*
Other arguments are passed to the macro expander.

MACRO

10.5 **MACRO** provides a powerful and flexible macro language used to simplify many of the features and functions contained in the library. The **defmacro** feature previously discussed is used to declare the **MACRO** keyword whose implementation is a preprocessor-internal routine named *macro*. The terminating delimiter for a **MACRO** is the closing brace character. **MACRO** implements an enhanced **#define** syntax that supports multiple line, arbitrary length, nested macros, and preprocessor directives with positional, optional, optional keyword, required keyword, rest, and body arguments.

Name: **MACRO** — Enhanced COOL macro language

Synopsis: **MACRO** *name* [*expanding*] (*arglist*) { *body* }

name The name of the macro

expanding Optional argument that, when present, indicates that argument names themselves should be macro-expanded before passing onto and invoking the *name* macro.

arglist A list of comma separated arguments

KEY: *identifier* [= *value*]
All ensuing arguments are taken to be keyword arguments that allow the user to specify a particular value. Default values are supported by an equal sign and value, and can be applied to both regular and keyword arguments.

REST: *name* [= *count*]
Indicates that there are some number of arguments, all of which are referenced by the one named identifier. An optional equal sign and identifier contains the number of arguments remaining. This is typically used when an outer level macro must pass some number of arguments to an inner level macro.

BODY: *body*
Indicates that *body* is to be expanded to include all text within the braces after the macro call. This is useful for identifying a section of code that implements some part of the macro or should be passed to other nested macros.

body Statements substituted when the macro is expanded. These statements can be any valid C++ statements terminated with a semicolon and surrounded by curly-braces.

MACRO Examples

10.6 Following are three examples of **MACRO**, each using various features and concepts to highlight some of the COOL macro capabilities. More detailed and complex examples follow in subsequent sections. It cannot be emphasized enough how important the macro facility is to the implementation of COOL. Without it, many features and functions would not be possible or would be more cumbersome and difficult to use. As an example of this type of use, the aggressive reader is referred to the end of Section 11, Symbols and Packages, for a detailed examination of the **symbol_package** macro.

Example 1:

This is a simple use of **MACRO** to implement a wrapper to an initialization routine that provides greater flexibility in passing arguments than is possible with straight C++ 2.0 syntax.

```
1       MACRO set_val (size, value=0, KEY: low = 0, high) {
2         init (size, value, low, high-low)}
```

Line 1 contains the function prototype for the macro `set_val` defined between the following braces. This macro takes four arguments:

- `size` is a required positional argument;
- `value` is an optional positional argument that if not specified in a particular call has a default value of `0`;
- `low` is an optional keyword argument with a default value of zero;
- `high` is a required keyword argument.

Line 2 contains the body of the macro which in this case involves a call to the `init()` function. The following shows several legal invocations of the macro, along with the resulting macro expansions:

```
set_val (0, high=20)           —>  init (0, 0, 0, 20-0);
set_val (0, low=5, high=15)    —>  init (0, 0, 5, 15-5);
set_val (1, 2, high=25)        —>  init (1, 2, 0, 25-0);
```

Example 2:

The next example makes use of the **REST:** argument list modifier and recursive calls of the macro defined. Note that there are two macros, the first calls the second to do most of the work. The results of both are combined and placed on the standard output of the preprocessor:

```
1       MACRO build_table (name, REST: rest) {
2         char* name[] = { build_table_internal (rest) NULL}
3       }

4       MACRO build_table_internal (first, REST: rest=count) {
5         #first,
6         #if count
7         build_table_internal (rest)
8         #endif
9       }
```

The macro `build_table` is defined on lines 1 through 3 and takes two arguments: a name to associate with the table and a **REST:** argument called `rest` that refers to all remaining arguments. A **char*** variable called `name` is defined on line 2 and contains an embedded call to a second macro with the `rest` argument mentioned above. Note also that the embedded call is within the initialization braces of the character string variable and is followed by a **NULL** symbol.

The second macro defined in lines 4 through 9 loops through the `rest` argument values and recursively calls itself. Line 4 contains the prototype with two arguments. The first argument `first` is stripped from the incoming argument list and the remaining `count` arguments are left alone in the `rest` argument. Line 5 uses the ANSI # character on an argument to double quote the value. Then, a conditional clause tests `count` to see if there are remaining arguments and, if so, recursively calls the macro. When there are no more arguments, the `build_table` macro regains control and appends the **NULL** and closing brace to the result of the second macro.

A sample use of this macro is shown below to illustrate the construction of a **NULL**-terminated table containing character strings. Line 1 shows the macro call and line 2 shows the resulting macro expansion:

```
1       build_table (table, 1,2,3,4,5,6,7);
2       char* table[] = {"1", "2", "3", "4", "5", "6", "7", NULL};
```

Example 3:

As a final example, here is a macro that uses the **BODY:** modifier. It takes advantage of the current position feature found in the COOL container classes to implement a general purpose **LOOP** macro similar to that found in Common Lisp. Since all COOL container classes implement the current position iterator capability, this macro will work equally well with **List**, **Vector**, **Set**, and so on:

```
1       MACRO LOOP (type, variable, container, BODY: body) {
2         { type variable;
3             for (container.reset (); container.next (); ) {
4                 variable = container.value ();
5                 body
6             }
7         }
8       }
```

Line 1 contains the prototype of the macro `LOOP` that takes four arguments: a container class element type; a variable name (of the type) to be declared; the name of a container class instance; and a **BODY:** argument of code to be applied to each element. Line 2 declares an instance of the element type in the specified container class. Lines 3 through 6 implement a loop that iterates through the elements of the container. Line 4 assigns the value of the element at the current position to the local variable declared on line 2. Line 5 expands the body argument specified.

A specific example for the **Vector**<*Type*> class is shown below. Lines 1 and 2 show the macro call and lines 3 through 8 show the resulting macro expansion:

```
1       Vector<int> v1;
2       LOOP (int, e, v1) { cout << e << ", ";}

3       { int e;
4         for (v1.reset(); v1.next(); ) {
5             e = v1.value();
6             cout << e << ", ";
7         }
8       }
```

This example contains an instance of `Vector<int>` called `v1`. The `LOOP` macro iterates through the vector and assigns each element to a temporary variable `e`. This is then used in the expanded `body` argument. The net result is to print all elements in the vector separated by commas.

ISSAME

10.7 The **ISSAME** macro is used in the preprocessor to compare two strings to see if they are the same. This macro is intended to be used in a similar manner as the preprocessor **#if** directive, which allows a symbol to be compared to some integer value. If the character strings are the same, **ISSAME** returns one; otherwise, it returns zero.

Name: **ISSAME** — Compares two character strings at compile time

Synopsis: **ISSAME** (*arg1*, *arg2*)

arg1	The first character string
arg2	The second character string

Example: This macro is used in the COOL **Hash_Table**<*T1*,*T2*> class to select the hash function based on the key type. If the hash table is parameterized such that the key type is **char***, a specific hashing function suited for character strings is implemented as the default hashing scheme. If not, an alternate hashing function is used. In the example below, line 1 compares the key type to several string type names. If a match is indicated, the statements at line 2 will be used. If no match is indicated, the statements at line 4 will be used.

```
1       #if ISSAME (T1, char*, String, Gen_String)
2       ...
3       #else
4       ....
5       #endif
```

KEYARGS

10.8 The **KEYARGS** macro implements a keyword argument feature for standard C++ functions similar to the **KEY:** modifier available with **MACRO** which supports optional keyword arguments.

Name: **KEYARGS** — Provides keyword arguments for C++ functions

Synopsis: **KEYARGS** *type name* (*arglist*)

type	Function return type
name	Name of the function
arglist	A C++ function argument list that supports keyword arguments:

[**KEY:**] *identifier* [= *default*] [, *arglist*]
All ensuing arguments are taken to be keyword arguments that allow the user to specify a particular value. Default values are supported by an equal sign and value, and can be applied to both regular and keyword arguments.

Example: This example defines the function `set` that returns a Boolean value. The first argument (`size`) is a required positional argument, while the second and third (`low` and `high`) are optional keyword arguments. A skeleton implementation of this function is shown in lines 1 through 3 below:

```
1       KEYARGS Boolean set (int size, KEY: int low=0, int high=100) {
2          ...
3       }
```

Lines 4 through 6 show a call to `set` with a value of `512` for the first argument and a value `1024` for the key argument `high`. The value of the keyword argument `low` will default to value 0. Lines 7 through 9 show the results of this macro expansion:

```
4       if (set (512, high=1024) == TRUE) {
5          ...
6       }

7       if (set (512, 0, 1024) == TRUE) {
8          ...
9       }
```

ONCE_ONLY

10.9 The **ONCE_ONLY** macro allows an application to control the expansion or insertion of a section of code or function. **ONCE_ONLY** creates a symbol in a package whose value is the file name where the symbol was first encountered. If the current value of the symbol is the same as the current file (available from the standard preprocessor symbol `__FILE__`), the code is expanded and compiled. If not, nothing happens. **ONCE_ONLY** uses symbol and package objects and is more completely discussed in Section 11, Symbols and Packages.

Name: **ONCE_ONLY** — A macro whose body is expanded only once

Synopsis: **ONCE_ONLY** (*name*) {*body*}

name	Symbolic name given to this operation
body	Statements substituted when the macro is expanded

Example: The C++ parameterized type macros generate two sets of code: a declaration that must always be included and implemented code that only needs to be compiled once. This is particularly important when the definition of the parameterized type is in a header file. By using the **ONCE_ONLY** macro, all macros and expansion of code are controlled and located in a single header file. The code implementing the parameterized type is expanded by the first application source file that included the header file.

The **DECLARE** macro used to declare a specific type of parameterized class only declares the class type and inline member functions. This could be changed to also implement the member functions by invoking the **IMPLEMENT** macro, if this is only done once during compilation. The macro `AUTO_DECLARE` declared below would implement the member functions one time only.

```
1      MACRO AUTO_DECLARE (name, REST: parms) {
2        DECLARE name<parms>;
3        ONCE_ONLY (Implement_##name<parms>) {
4          IMPLEMENT name<parms>;
5        }
6      }
```

Line 1 declares the macro `AUTO_DECLARE` with two arguments. The first argument specifies the parameterized class name and the second specifies any necessary arguments, including the type. Line 2 declares the parameterized class of the specified type. Lines 3-5 utilize **ONCE_ONLY** to implement the parameterized class if it has never been implemented before. This mechanism is not the default mechanism used in COOL because it prevents the fracturing of the source code template to reduce program size. This feature is available with CCC and is discussed in section 5, Parameterized Types.

EXPAND_ARGS

10.10 The **EXPAND_ARGS** macro is useful when one or more of the arguments to some **MACRO** are themselves macros that must be expanded first. This feature is also available via the **expanding** option in the **MACRO** syntax discussed earlier. The major difference between the two is that **EXPAND_ARGS** allows this function to be added to existing macros that may not have this already in place.

Name: **EXPAND_ARGS** — Expand macro arguments before invocation

Synopsis: **EXPAND_ARGS** (*name*, **REST:** *args*)

name	Name of the macro to be invoked
args	Arguments to be expanded and then passed to the macro

Example: The <`stdarg.h`> header file provides a set of preprocessor macros to allow the C++ compiler to accept a variable number of arguments in a function call. The syntax of one of these macros is `va_arg` (*argp*, *type*), where *type* is the type of the arguments expected. In the case of such things as COOL parameterized classes, however, a type like **Pair**<*Generic**, *Symbol**> is not recognized as a valid type by `va_arg` because it too is a macro that must be expanded first. The solution is to pass the name of the macro and its arguments to the **EXPAND_ARGS** macro, as shown below in line 2, which results in the *type* argument being expanded before being passed on, instead of the standard call as in line 1.

```
1       va_arg (argp, type)
2       EXPAND_ARGS (va_arg, argp, type)
```

INITIALIZE

10.11 The **INITIALIZE** macro guarantees to execute a body of code before the main program is called. This is often necessary in an application when a table or state information needs to be initialized before constructors can be called. **INITIALIZE** works by creating a static function containing the body of code to be executed. It initializes a global static variable, `For_Initialization_Only`, with a pointer to this function. `For_Initialization_Only` is a class whose constructor executes the function. The C++ language guarantees to execute the constructors for all global and static class instances before the main program is run. However, there is no mechanism by which the user can control the ordering of global static constructors themselves.

Name: **INITIALIZE** — A **MACRO** whose body is executed once

Synopsis: **INITIALIZE** (*name*) { *body* }

name	Name of the initialization sequence
body	Statements substituted when the macro is expanded

Example: In the following example, a global instance of a hash table is created on line 1 where both the key and the value are character strings. Lines 2 through 6 contain the **INITIALIZE** macro invocation to initialize this hash table by invoking the **put** member function of the **Hash_Table** class.

```
1       Hash_Table<char*, char*> capitals_g;
2       INITIALIZE (capitals_g) {
3         capitals_g.put ("Texas", "Austin");
4         capitals_g.put ("Arkansas", "Little Rock");
5         capitals_g.put ("Michigan", "Lansing");
6       }
```

IGNORE MACRO

10.12 The **IGNORE** macro silences warnings from the compiler relating to unused variables or function arguments. An application often has no control over the interface to a function and does not require all of the arguments. In other situations, an object might be created so that a friend function can access some private static data member. Without this macro, warnings of the type `"Warning: variable <`*foo*`> declared but not used"` appear. The **IGNORE** macro suppresses these warning messages.

Name: **IGNORE** — Silences compiler warnings from unused variables

Synopsis: **IGNORE** (*name*)

name The name of the argument/variable not used

Example: The following example shows the **main** function of a program with its two standard arguments. However, in this example, these arguments are unused. By using the **IGNORE** macro, the warning error messages are never generated by the compiler.

```
main (int argc, char** argv) {
  IGNORE (argc);                    // Don't use argument
  IGNORE (argv);                    // Don't use argument
  ....
}
```

11 SYMBOLS AND PACKAGES

Introduction

11.1 A package provides a relatively isolated namespace for various COOL components called *symbols*. Those symbols grouped into a particular package are said to be owned by that package. A symbol that is owned by a particular package is said to be interned in that package. In general, the term interned means that a particular object is uniquely identifiable in some context. When a symbol is interned, it becomes uniquely identifiable by the symbol name within a namespace context. The package system provides logical groupings of symbols supporting relationships established between named objects and the values they contain. Although the notion of symbols being grouped into packages is fairly straightforward, the nature of the relationships that can exist between packages and the way in which they establish a namespace can be quite complex. COOL provides several kinds of macros discussed later in this section to simplify the usage and manipulation of symbols and packages.

A symbol is a data object that defines a relationship between a name, a package, a value, and a property list. The name is a character string used to identify the symbol. Once a name is established for a symbol, you are not allowed to change it. The value field is used to refer to some C++ object. Property lists are lists of alternating names and values. The property list allows you to associate supplemental attributes with a symbol. Initially, the property list for a symbol is empty. This section discusses the symbolic computing facilities provided with COOL. The following items are covered:

- **Symbol**
- **Package**
- **DEFPACKAGE** and **DEFPACKAGE_SYMBOL**
- Package macros

The **Symbol** and **Package** classes implement the basic symbolic computing support. **DEFPACKAGE** and **DEFPACKAGE_SYMBOL** are flexible, low-level macros used to create and manipulate symbols and packages at both compile time and run time. Finally, the package macros discussed in the latter portion of this section provide a flexible and easy interface to the symbol and package features and allow a programmer to quickly use powerful constructs and features.

NOTE: The symbol and package classes use **operator=** when copying names and values. You should be careful when reusing memory, since the default pointer assignment operator copies the pointer, not the value pointed at.

Requirements

11.2 This section discusses the symbol and package facilities of COOL. It assumes that you have a working knowledge of the C++ language and have read and understood Section 10, Macros.

Symbol and Package Classes

11.3 COOL supports efficient and flexible symbolic computing by providing symbolic constants and run time symbol objects. You can create symbolic constants at compile time and dynamically create and manipulate symbol objects in a package at run time by using any of several simple macros or by directly manipulating the objects.

The **Symbol** class implements the notion of a symbol that has a name with an optional value and property list. Symbols are interned into a package, which is merely a mechanism for establishing separate name spaces. The **Package** class implements a package as a hash table of symbols and includes public member functions for adding, retrieving, updating, and removing symbols.

Symbols and packages in COOL manage error message textual descriptions, provide polymorphic extensions to C++ for object type and contents queries, and support sophisticated symbolic computing not normally available in conventional languages.

Symbol Class

11.4 The **Symbol** class implements the notion of a symbol that has a name with an optional value and property list. The **Symbol** class is publicly derived from the **Generic** class. Symbols are interned in a package, which is merely a mechanism for establishing a namespace whereby there is only one symbol with a given name in a given package. Packages are implemented as hash tables by the COOL **Package** class, which is a friend of the **Symbol** class. Because each named symbol is unique within its own package, the symbol can be used as a dynamic enumeration type and as a run time variable.

The name of a symbol is specified by a character string. The value of a symbol is specified as a pointer to a **Generic** object. The property list of a symbol is specified by an **Association**<**Symbol***,**Generic***>, where the name of the property is a pointer to a **Symbol** object and the value of the named property is a pointer to a **Generic** object.

Name: **Symbol** — Named, interned objects with a value and property list

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

Base Classes: **Generic**

Friend Classes: **Package**

Protected Constructors:

inline Symbol (**const char*** *name*);
Creates a symbol object with the name *name*. This member function is for use by the **Package::intern** member function. An application program should only create symbols interned and associated with a specific package.

Public Constructors:

inline Symbol ();
Applications should use the **Package::intern** member function to create symbols. The public constructor is provided for use by COOL macros to create and initialize constant symbols used for run time type query.

Member Functions:

Boolean get (**const Symbol*** *name*, **Generic*** *value*);
Looks up the named property *name* on the property list of the symbol object. If found, this member function copies the associated value into *value* and returns **TRUE**; otherwise, this member function returns **FALSE**.

inline const char* name () **const**;
Returns a constant pointer to the name associated with a symbol object.

inline Properties* plist ();
Returns a pointer to the property list associated with a symbol. *Properties* is an object of type **Association**<*Symbol**, *Generic**>.

void put (**const Symbol*** *name*, **Generic*** *value*);
Looks up the named property *name* on the property list of the symbol object. If found, this member function updates the value of the property with *value*; otherwise, this member function adds a new property *name* with the value *value*. If this is the first property added to the list, enough storage for four properties is allocated.

Boolean remove (**const Symbol*** *name*);
Looks up the named property *name* on the property list of the symbol object. If found, this member function removes the property and returns **TRUE**; otherwise, this member function returns **FALSE**.

inline Generic* set (**Generic*** *value*);
Sets the value associated with the symbol object to *value* and returns the new value. The destructor for the old value is *not* called automatically.

inline Generic* value ();
Returns a pointer to the value associated with the symbol object.

Friend Functions:

friend ostream& operator<< (**const ostream&** *os*,
const Symbol* *name*);
Overloads the output operator to provide a formatted output capability for a pointer to a symbol object *name*.

friend ostream& operator<< (**const ostream&** *os*,
const Symbol& *name*);
Overloads the output operator to provide a formatted output capability for a reference to a symbol object *name*.

Package Class

11.5 The **Package** class acts as a symbol table for a collection of **Symbol** objects. It is publicly derived from the **Hash_Table**<*char**, *Symbol**> class and implements a hash table of symbols. The **Package** class includes public member functions for adding, retrieving, updating, and removing symbols. It also provides completion and spelling correction on a symbol name (see the example programs later in this section).

Name: **Package** — A namespace for a collection of symbols

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

Base Classes: **Hash_Table**<*char**, *Symbol**>, **Generic**

Friend Classes: **Symbol**

Constructors:

inline Package ();
Creates a package object of default size to hold 24 entries.

inline Package (**unsigned long** *number*);
Creates a package to hold at least *number* entries.

Package (**unsigned long** *number*, **Package_Initializer** *fn*);

Creates a package of constant symbols to hold at least *number* entries. *Package_Initializer* is a function of type **void** (*Package_Initializer*)(*Package**) that allows the programmer to perform an operation initializing a package object. This constructor is primarily for use by the package macros. A detailed explanation of the macros and construction of the **symbol_package** macro is provided in the paragraph entitled, Symbol Package Implementation, at the end of this section below.

inline Package (**Package&** *pkg*);

Creates a new package, duplicating the size and values of another package object *pkg*.

Member Functions:

inline long capacity () **const**;

Returns the maximum number of entries the package can hold.

void clear ();

Removes all entries from the package and adjusts the appropriate counts.

inline Package_state& current_position () **const;**

Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of package.

Boolean find (**const char*&** *name*);

Searches the package for a symbol whose name matches *name*. If found, this function sets the current position to the symbol matching the character string and returns **TRUE**; otherwise, this function invalidates the current position and returns **FALSE**.

Boolean get (**const char*&** *name*, **Symbol&** *sym*);

Searches the package for a symbol whose name matches *name*. If found, this function sets the current position to the symbol matching the character string, updates *sym* with the symbol object found, and returns **TRUE**; otherwise, this function invalidates the current position and returns **FALSE**.

inline Boolean get_key (**const Symbol*** *sym*, **char*&** *name*);

Searches the package for the name associated with the symbol *sym*. If found, this function sets the current position to the symbol entry, updates *name* with name of the symbol object found, and returns **TRUE**; otherwise, this function invalidates the current position and returns **FALSE**.

Symbol* intern (**const char*** *name*);

Creates a new symbol object with the name *name*, or returns an existing symbol with the same name. This function updates the current position to the new or existing entry.

inline Boolean is_empty () **const**;

Returns **TRUE** if the package contains no entries; otherwise, this function returns **FALSE**.

const char*& key ();

Returns a reference to the character string name of the symbol at the current position. If the current position is invalid, an **Error** exception is raised.

inline long length () **const**;

Returns the number of entries in the package.

Boolean next ();

Advances the current position pointer to the next entry in the package and returns **TRUE**. If the current position is invalid, this function advances to the first entry and returns **TRUE**. If advancing past the last entry in the package, this function invalidates the current position and returns **FALSE**.

Package& operator= (**const Package&** *pkg*);

Overloads the assignment operator for the class and assigns the package object to have the value of *pkg* by duplicating the size and entries. This function invalidates the current position of the package object.

Boolean operator== (**const Package&** *pkg*);

This function returns **TRUE** if the package object has the same symbol entries as *pkg*; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Package&** *pkg*);

This function returns **TRUE** if the package object has different symbol entries as *pkg*; otherwise, this function returns **FALSE**.

Boolean prev ();

Moves the current position pointer to the previous entry in the package and returns **TRUE**. If the current position is invalid, this function moves to the last entry and returns **TRUE**. If moving to the previous entry passes the first entry in the package, this function invalidates the current position and returns **FALSE**.

Boolean put (**const char*** *name*, **Symbol&** *sym*);

Searched for the symbol associated with name *name* and, if found, updates with the new symbol *sym*. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. The current position is updated to the added entry *sym*.

Boolean remove ();

Removes the symbol at the current position, deallocates its storage, and returns **TRUE**. This function sets the current position to the entry immediately following the entry removed if in the same bucket; otherwise, this function invalidates the current position. If the current position is invalid, an **Error** exception is raised and, if the handler returns, this function returns **FALSE**.

inline Boolean remove (**char*** *name*);

Searches the package for the symbol *name*. If the symbol is found, this function removes the symbol, deallocates its storage, sets the current position to the old location of the symbol, and returns **TRUE**; otherwise, this function returns **FALSE**.

Boolean remove (**Symbol*** *sym*);

Searches the package for the symbol entry *sym*. If found, this function removes the symbol, deallocates its storage, sets the current position to the old location of the symbol, and returns **TRUE**; otherwise, this function returns **FALSE**.

inline void reset ();

Invalidates the current position.

void resize (**long** *number*);

Resizes the package for at least *number* entries. If a growth ratio has been selected and it satisfies the resize request, the package grows by this ratio. This function invalidates the current position. If the resize value is zero or negative, an **Error** exception is raised.

inline void set_ratio (**float** *ratio*);
Updates the growth ratio for this instance of a package to *ratio*. When a package needs to grow, the current size is multiplied by the ratio to determine the new size. If the ratio is negative, an **Error** exception is raised.

const Symbol& value ();
Returns a reference to the symbol at the current position. If the current position is invalid, an **Error** exception is raised.

Friend Functions:

Boolean apropos (**Package&** *pkg*, **const char*** *name*);
Finds the next symbol from the current position in the package *pkg* whose name is *name*. If the symbol is found, this function returns **TRUE** and sets the new current position; otherwise, this function returns **FALSE**.

int complete (**Package&** *pkg*, **String&** *name*,
Boolean *sensitive* = **FALSE**);
Provides completion on *name*. If *sensitive* is **TRUE**, a case-sensitive character comparison is made; otherwise, a case-insensitive comparison is performed. This function modifies *name* to the completed value, returns the count of possible matches, and sets the current position of the package to the last match found.

Boolean completions (**Package&** *pkg*, **const char*** *name*,
Boolean *sensitive* = **FALSE**);
Finds the next symbol in the package *pkg* after the current position whose name starts with *name*. If *sensitive* is **TRUE**, a case-sensitive character comparison is made; otherwise, a case-insensitive comparison is performed. If the symbol is found, this function returns **TRUE** and sets the new current position; otherwise, this function returns **FALSE**.

int correct (**Package&** *pkg*, **const char*** *name*,
Boolean *sensitive* = **FALSE**, **int*** *errors* = **NULL**);
Performs spelling correction on a symbol whose name is *name* in the package *pkg*. If *sensitive* is **TRUE**, a case-sensitive character comparison is made; otherwise, a case-insensitive comparison is performed. This function returns the number of matches and sets the current position of *pkg* to the best match found. The number of corrections is provided in the optional *errors* argument.

friend ostream& operator<< (**ostream&** os, **const Package&** *pkg*);
Overloads the output operator for a reference to a package *pkg* to provide a formatted output capability for the **Package** class. This function returns a reference to the output stream.

inline friend ostream& operator<< (**ostream&** *os*, **const Package*** *pkg*);
Overloads the output operator for a pointer to a package *pkg* to provide a formatted output capability for the **Package** class. This function returns a reference to the output stream.

DEFPACKAGE

11.6 The **DEFPACKAGE** macro enables a programmer to declare a program-wide database of constant symbols with associated default values and properties. This is useful when the programmer needs to set up a table of symbols and knows all instances and requirements at compile time, as with the COOL **ERR_MSG** package discussed later in this section. Under such circumstances, the run time overhead associated with the **Package** class is avoided. The package database (that is, the place where the constant symbols are kept) is stored in a file on the include path. This file contains macro calls that can be used in an application to associate data with compile-time symbols.

Name: **DEFPACKAGE** — Symbolic C++ constant symbol mechanism

Synopsis: **DEFPACKAGE** *name <path> options*

name	A character string to be used as a symbol prefix
path	The name of an include file where symbol definitions are kept
options	One or more of the following comma-separate parameters:

count = *identifier*
The package file should define the specified preprocessor identifier whose value is the number of symbols defined in the package.

use_first = *int*
When nonzero, the value used is the first definition. Redefinition attempts are ignored. This option is used by the **ONCE_ONLY** macro.

noblank = *int*
When nonzero, removes all whitespace from symbol names.

case = **upper**
Converts all symbol name alphabetic characters to uppercase.

case = **lower**
Converts all symbol name alphabetic characters to lowercase.

case = **cap**
Capitalizes the first letter of each symbol name, and converts remaining letters to lowercase.

case = **sensitive**
Preserves the case of the symbol name as used. This is the default behavior.

start = *int*
Uses the provided value as the first (that is, starting) point for each enumerated symbol index. The default is zero.

increment = *int*
Increments symbol index values by the specified value. The default is one.

template = *int*
The value is inclusive-or'ed with the index every symbol value. The default is zero.

max = *int*
Generates an error when the number of constant symbols in the package exceeds the specified value.

While the **DEFPACKAGE** macro provides great flexibility and versatility in creating a package of constant symbols for an application, the creation of the most common types of packages likely to be needed by the programmer is made easier by the following macros:

- **MACRO** *enumeration_package* **(***name***,** *file***, REST:** *options***)**
- **MACRO** *text_package* **(***name***,** *file***, REST:** *options***)**
- **MACRO** *symbol_package* **(***name***,** *file***, REST:** *options***)**
- **MACRO** *once_only (name, file,* **REST:** *options)*

The first three allow the programmer to easily create packages of symbols with varying levels of sophistication. The fourth is used by the various COOL components to ensure that certain functions are performed only once during the compilation phase. Complete information and usage of these macros is discussed later in this section.

Adding Symbols To A Package

11.7 The **DEFPACKAGE_SYMBOL** macro adds, updates, and retrieves constant symbols, their values, and properties from a package created with the **DEFPACKAGE** macro. **DEFPACKAGE_SYMBOL** updates the program-wide database of constant symbols stored in a file with macro definitions and calls that can be used in an application to associate data and property lists with compile time symbols. As with the **DEFPACKAGE** macro, **DEFPACKAGE_SYMBOL** is a flexible, low-level function. The most common types of packages and constant symbol manipulation requirements are made easier by the four macros mentioned above and discussed later in this section.

Name: **DEFPACKAGE_SYMBOL** — Symbolic C++ constant symbol manipulation

Synopsis: **DEFPACKAGE_SYMBOL** (*package*, *symbol*, *type*, *value*, *property*, *expander*)

package	The name of a package to access. Note that the package must have already been defined with **DEFPACKAGE**
symbol	The name of the symbol to be added, updated, or retrieved
type	The optional type of the value
value	The optional value of the symbol or property
property	The optional name of the property
expander	When present, replaces the **DEFPACKAGE_SYMBOL** invocation with the result of calling the specified macro *expander (index, symbol, type, value)* where:

expander
The expander macro to be called and specified in the invocation of **DEFPACKAGE_SYMBOL.**

index
The symbol's index number.

symbol
The name of the symbol.

type
The optional type of the value.

value
The optional type of the symbol or property.

If you use **DEFPACKAGE** to create your own specialized package, you will probably want to write simple macros that expand into calls to manipulate the constant symbol entries. **DEFPACKAGE_SYMBOL** writes three other macro definitions to the package's definition file for use in a user-application. Each use of the **DEFPACKAGE_SYMBOL** macro generates another macro. <*Package*>**_DEFINITIONS**. This macro expands to use the three macros mentioned above to define the symbol, set the symbol value, and set the symbol properties. It is invoked at compile time to create the constant symbol package.

Name: <*Package*>**_DEFINITIONS** — Create symbolic C++ constant symbol values

Synopsis: <*Package*>**_DEFINITIONS** (*define*, *value*, *property*);

define Macro to be used to create the constant symbol of the form:

define_macro (index, name)

value Macro to be used to set the value of the constant symbol of the form:

value_macro (index, type, value)

property Macro to be used to set a property of the constant symbol of the form:

define_macro (index, property, type, value)

Under most circumstances, the programmer will never have the need to use these macros. However, for those interested, further information about these macros and their use in constructing a constant symbol package is available in the documentation and examples in the `~COOL/Package/defpackage.h` header file and the COOL `SYM` and `ERR_MSG` package files in the COOL `include` subdirectory. Finally, a detailed explanation of the macros and construction of the **symbol_package** macro is provided in the paragraph entitled, Symbol Package Implementation, at the end of this section.

NOTE: Constant symbol packages defined and manipulated by the macros discussed in this section must have storage allocated for them and code to initialize them at program startup time. This is managed by the COOL file `symbols.C` that should be compiled and linked with every application that uses COOL components. This file does not need to be changed unless you create your own symbol packages, in which case you should add the appropriate include and initialization statements (see the examples later in this section). An automated method for ensuring correct package setup and symbol initialization is shown in the make file for the example programs for this manual in the `COOL/examples` subdirectory.

Enumeration Package

11.8 The **enumeration_package** macro is for use in applications that need to create a collection of constant symbols. Enumeration symbols can be used anywhere that an enumeration type can be used. One reason for selecting symbols in an enumeration package over the standard **enum** type is that it is easier to add new symbols. The enumeration package macro automatically collects them from across the source base and maintains a single database in the specified header file.

NOTE: An enumeration package is stored in a file located somewhere on the include directory search path. This header file must initially be created as an empty file by the programmer, since the macro does not know which subdirectory on the include file search path to select. A convenient mechanism for creating this file on UNIX systems is the touch(1) command.

Once an enumeration package has been created, symbols can be added and retrieved by using the package name with the symbol name surrounded by parentheses. If the symbol contained between parentheses has not already been added to the package, it is added and the new value is returned. If the symbol is already present in the package, the existing value is returned.

Name: **enumeration_package** — Enumerated constant symbol package macro

Synopsis: **enumeration_package** (*name*, *file*, **REST:** *options*)

name	Specifies the name of the package.
file	The file located somewhere on the include file directory search path that acts as a database for the symbols across the application source base.
options	Any other valid **DEFPACKAGE** options.

Macros: *name* (*sym***)**

Defines the symbol *sym* in the package *name* if it is undefined, and returns a pointer to the symbol entry.

Enumeration Package Example

11.9 The following program declares an enumeration package of constant symbols that are dynamically added in the program text. The enumerated symbol objects behave exactly like the built-in **enum** type in that there is no storage allocated. However, they have the added benefit that they can be created or added at any time in any source file in the program. The enumeration package macro ensures that they are collected in a single database.

```
#include <COOL/Package.h>              //Include COOL Package header
enumeration_package (MY_ENUM, ”my_enum.p”); //Create enum package

int main (void) {
 cout << ”MY_ENUM (Red) has a value of ” << MY_ENUM (Red) << ”\n”;
 cout << ”MY_ENUM (Yellow) has a value of ” << MY_ENUM (Yellow) << ”\n”;
 cout << ”MY_ENUM (Green) has a value of ” << MY_ENUM (Green) << ”\n”;
 return 0;                             //Return valid success code
}
```

Line 1 includes the COOL `Package.h` class header file. Line 2 creates an enumeration package `MY_ENUM` whose database is kept in the file `my_enum.p` somewhere on the include file search path. This file must initially be created as an empty file by the programmer, since the macro does not know which subdirectory on the include file search path to select. A convenient mechanism for creating this file on UNIX systems is the touch(1) command. Lines 4 through 6 add the enumerated symbols `Red`, `Yellow`, and `Green` to the package and display their respective values. Finally, line 7 returns a valid successful completion code.

The following shows the output from the program:

```
MY_ENUM (Red) has a value of 0
MY_ENUM (Yellow) has a value of 1
MY_ENUM (Green) has a value of 2
```

At first glance, this example doesn't seem interesting because this is a simple three-line, one-source file program. However, imagine an application that solves a complex communications problem and requires many flags. A programmer could use the dynamic COOL **Bit_Set** class and use an enumerated package of symbols defined across many files to index the bits in the vector. This will result in a very flexible and efficient (1 bit/flag) implementation that can easily be altered and extended.

Text Package

11.10 The **text_package** macro is for use in applications that need to create a collection of symbols with values the same as the symbol name. This is useful for the manipulation of error messages in an application, since the symbol definition file contains a summary of all the messages. In addition, the message text may be substituted in another language at run time. The text package macro automatically collects text symbols from across the source base and maintains a single database in the specified header file.

NOTE: A text package is stored in a file located somewhere on the include directory search path. This header file must initially be created as an empty file by the programmer, since the macro does not know which subdirectory on the include file search path to select. A convenient mechanism for creating this file on UNIX systems is the touch(1) command.

Once a package has been created, symbols can be added and retrieved by using the macro whose name is the same as the package name and whose single argument is the symbol name. After creating a package, add and retrieve symbols with the package and symbol names in parentheses. If the symbol in parentheses has not already been added to the package, it is added and a pointer to the new value returned. If the symbol is already present in the package, the existing value returns.

The **ERR_MSG** text package is the COOL global error message package. It stores the text to all error messages in the COOL class and macro library. The **text_package** macro creates the **ERR_MSG** package. As exceptions are added to the program, a corresponding entry is automatically made into the error message package at compile time. The error message package loads into the `symbols.C` file and is always the last file compiled in an application that uses COOL components. This ensures that all symbol values have been collected up over the source base.

An application that uses a text package to store all textual information can support multiple languages through the language property field. All such messages are collected together in one file by the symbol and package macros. This one file can be edited by the programmer to change or add alternate translations for each message. The only change required of the application source code is an initial statement to set the program execution language. All error messages in COOL are implemented this way to facilitate ports to other language environments.

Name: **text_package** — Resource text symbol package macro

Synopsis: **text_package** (*name*, *file*, **REST:** *options*)

name	Specifies the name of the package.
file	The file located somewhere on the include file directory search path that acts as a database for the symbols across the application source base.
options	Any other valid **DEFPACKAGE** options

Macros: *name* (*sym***)**

Defines the symbol *sym* in the package *name* if it is undefined and returns a pointer to the symbol entry.

Friend Functions: **int set_text_language (Symbol*** *language* = **NULL,**
text_package_entry* *package* = **NULL);**

Sets a new language for a text package. The first argument is a symbol representing the name of the new language, and the second argument is the starting entry in the package from which to begin language translation. If the language symbol is not specified, the default language is the original from the program. If the entry point is not specified, the default is the first symbol in the package. When a package entry does not have a translation for the specified language, a **Warning** exception is raised. This function returns the number of entries in the package for which a translation does not exist.

Text Package Example

11.11 The following program uses the **text_package** macro to create a text resource file that can be maintained across all source files in an application. This example is split into two parts. In the first example, two symbols are added to the text package. The value of a symbol in a text package is identical to the name. Alternate languages can be supported by adding the appropriate property. An attempt is made to set the text language property to an unsupported language symbol. **Warning** exceptions are raised as a result.

```
1   #include <COOL/Package.h>             // Include COOL Package header
2   text_package (MY_TEXT, ”my_text.p”);  // Create text package

3   int main (void) {
4     cout << ”1st message: ” << MY_TEXT (”Hi! What’s up?”) << ”\n”;
5     cout << ”2nd message: ” << MY_TEXT (”See you later”) << ”\n”;
6     set_text_language (SYM (Southern), &MY_TEXT_entries[0]);
7     cout << ”1st message: ” << MY_TEXT (”Hi! What’s up?”) << ”\n”;
8     cout << ”2nd message: ” << MY_TEXT (”See you later”) << ”\n”;
9     set_text_language (NULL, &MY_TEXT_entries[0]);
10    cout << ”1st message: ” << MY_TEXT (”Hi! What’s up?”) << ”\n”;
11    cout << ”2nd message: ” << MY_TEXT (”See you later”) << ”\n”;
12    return 0;                           // Return valid success code
13  }
```

Line 1 includes the COOL `package` header file. Line 2 uses **text_package** to create a package whose name is `MY_TEXT` and whose values are stored in the file `my_text.p` somewhere on the include search path for this application. Note that this file must be initially created by the programmer, since the COOL package system cannot know in which directory the file should be placed. Lines 4 and 5 add two symbols to the text package. Line 6 attempts to set the language for the text package to `Southern`, a symbol interned in the global COOL symbol package `SYM` (discussed below in the paragraph, Symbol Package). Lines 7 and 8 print the values of the two symbols for the newly set language property. Line 9 restores the language property back to its initial value, *hacker english*. Lines 10 and 11 output the values of the symbols back in the default language. Finally, line 12 ends the program with a valid success code.

The following shows the output from the program:

```
1   1st message: Hi! What’s up?
2   2nd message: See you later
3   Warning: No Southern translation for ”Hi! What’s up?”
4   Warning: No Southern translation for ”See you later”
5   1st message: Hi! What’s up?
6   2nd message: See you later
7   1st message: Hi! What’s up?
8   2nd message: See you later
```

Lines 1 and 2 contain the values of the two symbols as they are added to the text package. Lines 3 and 4 are warning exceptions raised when the language property for the package was set to `Southern`, indicating that the two symbols do not have translations for this property. As a result, lines 5 and 6 output the same values for the two symbols. Lines 7 and 8 output the same values with the switch back to the default language. The COOL package system creates and maintains the text package symbol file `my_text.p` shown below:

```
/*
* DEFPACKAGE MY_TEXT       definitions file.
*
* This file is automatically generated by the cpp DEFPACKAGE facility
* DO NOT EDIT THIS FILE, because it may be re-written the next time CPP
* is run.
*
* This file is for:
* DEFPACKAGE MY_TEXT <MY_TEXT> name=my_text.p,
*   count=MY_TEXT_count, case=sensitive,
*   start=0, increment=1, template=0, max=0
*/
```

```
13      /* WARNING: Do not remove this line */
14      #define MY_TEXT_count 2

15      MACRO MY_TEXT_DEFINITIONS(define_macro, value_macro, property_macro) {
16        define_macro (0, "Hi! What's up?")
17        define_macro (1, "See you later")
18      }
```

Lines 1 through 12 contain the standard header commentary information, including the package creation specifications placed at the top of every symbol file. Line 13 is important in that the package and symbol macros use this as a marker for placement in the file. Line 14 is a preprocessor constant reflecting the number of symbols in the package. Line 15 contains a **MACRO** to create the text package. Lines 16 and 17 contain two macros defining the two symbols added in the program.

The following textual insertion shows the customized contents of the generic `symbols.C` file which is always the last file to be compiled in any application using COOL components. This file is responsible for including any package definition files created during the compilation of other program source files. It must always be last to ensure that all symbols have been added to the package before it is implemented. The programmer need never alter the contents of this file unless an application-specific package has been created, as is the case with this example program.

```
// This file must be compiled and linked with every application utilizing the
// COOL library. The sample makefile shows the procedure for compilation order.
// It is important that this be the last file compiled before the link process
// begins. The constant symbols in the SYM package and ERR_MSG package are
// initialized by invoking the implement macros defined in <COOL/defpackage.h>

 1      #include <COOL/String.h>
 2      #include <COOL/Package.h>
 3      #include <COOL/Properties.h>

 4      implement_symbol_package (SYM, "sym_package.p")
 5      implement_text_package (ERR_MSG, "err_package.p")

// The next three lines are added to insure that the text and symbol packages
// manipulated by examples 11.11a.C, 11.11b.C, and 11.13.C are allocated and
// initialized, respectively.

 6      implement_text_package (MY_TEXT, "my_text.p")
 7      //implement_text_package (MY_TEXT, "my_text2.p")
 8      //implement_symbol_package (MY_SYM, "my_sym.p")
```

Lines 1 through 3 include the necessary COOL header files to enable the package and symbol system to be implemented. Lines 4 and 5 are the default contents of this file and implement the COOL global symbol and error message packages through two macros. An application that uses any COOL components must have these two lines compiled in the last file in the compilation process. Lines 6 through 8 have been added for this and the next two examples to implement the packages created. Note that lines 7 and 8 are commented out. The next two examples will create the text and symbol packages referred to here and will also uncomment the appropriate line.

The second part of this example continues below. In the first example, the attempt to set the language property for the package to `Southern` caused two **Warning** exceptions to be raised. The continuation of this example will add translations for the `Southern` language property to the text package. The program below is identical to the previous one except for the name of the file in which the package is stored. Line 2 contains the macro to create the package, and the file this time is specified as `my_text2.p`.

```
#include <cool/Package.h>              // Include COOL Package header
text_package (MY_TEXT, ”my_text2.p”);  // Create text package

int main (void) {
 cout << ”1st message: ” << MY_TEXT (”Hi! What’s up?”) << ”\n”;
  cout << ”2nd message: ” << MY_TEXT (”See you later”) << ”\n”;
  set_text_language (SYM (Southern), &MY_TEXT_entries[0]);
  cout << ”1st message: ” << MY_TEXT (”Hi! What’s up?”) << ”\n”;
  cout << ”2nd message: ” << MY_TEXT (”See you later”) << ”\n”;
  set_text_language (NULL, &MY_TEXT_entries[0]);
  cout << ”1st message: ” << MY_TEXT (”Hi! What’s up?”) << ”\n”;
  cout << ”2nd message: ” << MY_TEXT (”See you later”) << ”\n”;
  return 0;                              // Return valid success code
}
```

The text package contained in the file `my_text2.p` is a copy of the previous example with the addition of a `Southern` property for each symbol. To add such properties, the programmer must edit the file and add the appropriate translation for each symbol entry, as shown below.

```
 1  /*
 2  * DEFPACKAGE MY_TEXT       definitions file.
 3  *
 4  * This file is automatically generated by the cpp DEFPACKAGE facility
 5  * DO NOT EDIT THIS FILE, because it may be re-written the next time CPP
 6  * is run.
 7  *
 8  * This file is for:
 9  * DEFPACKAGE MY_TEXT <MY_TEXT> name=my_text.p,
10  *   count=MY_TEXT_count, case=sensitive,
11  *   start=0, increment=1, template=0, max=0
12  */

13  /* WARNING: Do not remove this line */
14  #define MY_TEXT_count 2

15  MACRO MY_TEXT_DEFINITIONS(define_macro, value_macro, property_macro) {
16   define_macro (0, ”Hi! What’s up?”)
17   property_macro(0, Southern, char*, ”Howdy! What y’all up to?”)
18   define_macro (1, ”See you later”)
19   property_macro(1, Southern, char*, ”Y’all come back now, ya’ heah?”)
20  }
```

Lines 1 through 14 are identical to the previous package file. Lines 15 through 20 define the symbols contained in this package. Lines 16 and 18 are the same as before and contain macros to create the two text symbols. Lines 17 and 19 have been added by the programmer to establish a `Southern` property for each text symbol. Note that the first value of each definition and property macro is an integer. These must match to ensure correct package setup.

NOTE: A package file is recreated every time the compilation process is performed. Any changes made to support translations should be kept in a separate file and merged into the package file after the compilation is complete.

To complete this example, the `symbols.C` file must be changed slightly to implement the text package contained in the file `my_text2.p` with the new properties. The following shows the output of the program:

```
1     1st message: Hi! What's up?
2     2nd message: See you later
3     1st message: Howdy! What y'all up to?
4     2nd message: Y'all come back now, ya' heah?
5     1st message: Hi! What's up?
6     2nd message: See you later
```

Lines 1 and 2 output the value of the two text symbols added to the package. Lines 3 and 4 output the value of the `Southern` property for each symbol. Note, however, that the symbols used in the program did not have to be changed to support a different language. Lines 5 and 6 output the value of the symbols back in the default language.

Symbol Package

11.12 The **symbol_package** macro creates and accesses a **Package** object containing symbols whose values can be assigned at run time. Symbols in the **symbol_package** are pointers to **Symbol** objects. Symbols known and declared at compile time are interned in a table. The symbol package macro automatically collects these symbols from across the source base and maintains a single database in the specified header file. Additional symbols can be added at run time. Symbols have values and properties whose initial values can be declared. If not specified, the values and properties are nonexistent; that is, no space other than storage for a **NULL** pointer is allocated for them. The global **Package** object created has the name *name*`_package_g`, where *name* is the name of the package specified in the macro invocation.

NOTE: A symbol package is stored in a file located somewhere on the include directory search path. This header file must initially be created as an empty file by the programmer, since the macro does not know which subdirectory on the include file search path to select. A convenient mechanism for creating this file on UNIX systems is the touch(1) command.

The **symbol_package** macro defines three macros for adding, updating, and retrieving symbols in the package. The first adds new symbols or retrieves existing symbols. The second adds a value of a specified type to an existing symbol entry. The third adds a named property of the specified type to an existing symbol entry.

The `SYM` symbol package is created with the **symbol_package** macro and is the COOL global type package. It stores the type and inheritance hierarchy for all classes that inherit from the **Generic** class to support run time type and object query. Each such class is represented by a symbol that may have various values and properties. All type information is accessed and manipulated by the macros and functions discussed in Section 12, Polymorphic Management.

Name: **symbol_package** — Constant symbol package macro with run time update

Synopsis: **symbol_package** (*name*, *file*, **REST:** *options*)

name Specifies the name of the package.

file The file located somewhere on the include file directory search path that acts as a database for the symbols across the application source base.

options Any other valid **DEFPACKAGE** options

Macros: *name* (*sym*)
Defines the symbol *sym* in the package *name* if it is undefined and returns a pointer to the symbol entry.

DEF_*name* (*sym*, *type*, *value*)
Defines a value of the specified type to the symbol *sym* in the package *name*.

DEF_*name***_PROPERTY** (*sym*, *property*, *type*, *value*)
Defines a property of the specified type and value to the symbol *sym* in the package *name*.

Symbol Package Example

11.13 The following program uses the **symbol_package** macro to create a symbol package. This example shows the manipulation of symbols, their associated values, and properties in a symbol package at both compile time and run time. Two symbols are added at compile time. One of these has a value and property specified at compile time. The other has its value and property fields assigned at run time.

```
#include <COOL/Date_Time.h>           // Include COOL Date/Time header
#include <COOL/Package.h>             // Include COOL Package header

symbol_package (MY_SYM, ”my_sym.p”);              // Create symbol package

DEF_MY_SYM (sym1, String, new String(”Greetings!”));
DEF_MY_SYM_PROPERTY (sym1, MY_SYM (Prop. example), Symbol, MY_SYM (String));

int main (void) {
  Symbol *s1 = MY_SYM (sym1);                     // Lookup first symbol
  cout << ”First symbol is ” << s1->name() << ”\n”; // Output symbol name
  cout << ”Also available via MY_SYM(sym1): ” << MY_SYM(sym1)->name()<<”\n”;
  cout << s1 << ”\n”;                             // Output value/property list
  s1->set (new String (”Goodbye!”));              // Add new value
  cout << ”sym1 value is now ” << s1->value () << ”\n”; // Output value
  Date_Time d1 (US_CENTRAL, UNITED_STATES);       // Create date/time object
  d1.set_local_time ();                           // Set to current date/time
  Symbol* s2 = MY_SYM (sym2);                     // Create new symbol object
  cout << ”Second symbol is ” << s2->name() << ”\n”;// Output symbol name
  cout << ”Also available via MY_SYM(sym2): ” << MY_SYM(sym2)->name()<<”\n”;
  s2->put (MY_SYM (Creation Time), &d1);          // Add property
  cout << s2 << ”\n”;                             // Output runtime symbol
  return 0;                                       // Return valid code
}
```

Lines 1 and 2 include the COOL `Date_Time.h` and `Package.h` header files. Line 3 uses the **symbol_package** macro to create a package whose name is `MY_SYM` and whose values are stored in the file `my_sym.p` somewhere on the include search path for this application. Note that this file must be initially created by the programmer, since the COOL package system cannot know which directory the file should be placed. Line 4 adds a value to the first symbol in the package with the **DEF_MY_SYM** macro. Note that this macro has the name of the package concatenated to form a package-specific macro. This was created by the macro in line 3. Similarly, line 5 adds a property to the first symbol in the package with the **DEF_MY_SYM_PROPERTY** macro. Line 7 adds a new symbol to the package. Lines 8-10 output the name and value of the first symbol in the package. Line 11 changes the value added at compile time to a new string added at run time and line 12 outputs this new value.

Lines 13 and 14 create a date/time object initialized with the local time. Line 15 creates a second symbol for the package and lines 16 and 17 output its name. Line 18 adds the named property `Creation Time` with a value of a pointer to the date/time object instantiated in line 13 to the second symbol `sym2` in the package. Line 19 outputs the newly updated symbol and line 20 ends the program with a successful completion code.

The following shows the output from the program:

```
1   First symbol is sym1
2   Also available via MY_SYM(sym1): sym1
3   sym1 Greetings! [value-type String ]
4   sym1 value is now Goodbye!
5   Second symbol is sym2
6   Also available via MY_SYM(sym2): sym2
7   sym2 [Creation Time United States 01-19-1990 07:46:07 US/Central]
```

Lines 1 and 2 output the name of the first symbol in the package. Lines 3 and 4 output the initial and new value and property lists for this symbol. Lines 5 and 6 output the name of the newly created second symbol object, and line 7 outputs the name, value, and property list of this symbol.

The COOL package system creates and maintains the symbol package file `my_sym.p` shown below:

```
/*
 * DEFPACKAGE MY_SYM        definitions file.
 *
 * This file is automatically generated by the cpp DEFPACKAGE facility
 * DO NOT EDIT THIS FILE, because it may be re-written the next time CPP
 * is run.
 *
 * This file is for:
 * DEFPACKAGE MY_SYM <MY_SYM> name=my_sym.p,
 *  count=MY_SYM_count, case=sensitive,
 *  start=0, increment=1, template=0, max=0
 */

/* WARNING: Do not remove this line */
#define MY_SYM_count 6
```

```
15   MACRO MY_SYM_DEFINITIONS(define_macro, value_macro, property_macro) {
16    define_macro (0, "sym1")
17    value_macro  (0, String, new String("Greetings!"))
18    property_macro(0, (&MY_SYM_symbols[5]), Symbol, (&MY_SYM_symbols[2]))
19    property_macro(0, (&MY_SYM_symbols[1]), Symbol, (&MY_SYM_symbols[2]))
20    define_macro (1, "value-type")
21    define_macro (2, "String")
22    define_macro (3, "sym2")
23    define_macro (4, "Creation Time")
24    define_macro (5, "Property example")
25   }
```

Lines 1 through 12 contain the standard header commentary information, including the package creation specifications placed at the top of every symbol file. Line 13 is important in that the package and symbol macros use this as a marker for placement in the file. Line 14 is a preprocessor constant reflecting the number of symbols in the package. Line 15 contains a **MACRO** to create the symbol package. Lines 16 through 25 contain macros defining the symbols and their values and properties added in the program.

Under most circumstances, the programmer need never examine this file. It is presented here merely as an aid in understanding the COOL symbol and package system. Although not included here, the customized `symbols.C` file (always the last file to compile in any COOL application) must include an implement macro for the `MY_SYM` package, as was shown earlier for the text package example. This file (`symbols4.C`) can be found in the `COOL/examples` subdirectory.

ONCE_ONLY Package

11.14 The **ONCE_ONLY** macro (discussed in Section 10, Macros) allows an application to control the expansion of a section of code. This might be useful, for example, when a table needs to be initialized once and once only when a constructor for some class is first called. This could be accomplished by having a **static** flag for the class set on the first call, with later calls checking the flag and skipping the initialization. The **ONCE_ONLY** macro, however, provides an intelligent and more efficient conditional compilation feature. It uses the `Once_Only_Package` to control the expansion and compilation of code only once in a program.

When a **ONCE_ONLY** macro invocation is encountered for the first time, a symbol is created with a name related to the macro call. A value is created that is a character string representing the file name where the symbol is first defined. This symbol is added to the `Once_Only_Package` and the body of code expanded. The next time the same **ONCE_ONLY** macro is encountered, a symbol name is created and looked up in the `Once_Only_Package` object. If the value is the same as the current file (available from `__FILE__` in the preprocessor), the body of code is expanded, and ready to be compiled. However, if the symbol has a different value (that is, the macro invocation is in a different file), the code is not expanded and thus, not compiled.

The symbol name specified in the macro ensures that a specified body of code expands and compiles only once across an entire source base. These symbol names and the `Once_Only_Package` are not available for general use other than through this macro. It is included here to provide you with another example of the use and flexibility of COOL symbols and package objects.

Interfacing to the SYM Package

11.15 Under some circumstances, it might be necessary for an application to interface to the global COOL symbol package `SYM` to reference type information automatically created and stored there by various macros. This could be the case in an application-specific library that must have certain knowledge about all the possible types available in an application, such as an inference engine where certain user-defined objects can implement specific firing rules. The default firing rule for each type of object could be represented as the value of the symbol representing the object type.

In the following code fragment, a function is defined that processes a list of string names containing the names of all the rule types in a particular rules-based inference engine. These names came from a rules grammar file generated by a translator that runs over the user's knowledge-base-specific rules. The character string names match class names defined within the user's application and so have a corresponding symbol entry in the COOL global symbol package. This function finds a matching symbol for each name and attaches a default firing rule as the value of the symbol and returns the number of rules processed. Other rules may be added to the property list of the symbol at run time.

```
 1   #include<COOL/Package.h>                           // COOL Package header file
 2   #include<COOL/List.h>                              // COOL List header file
 3   #include<COOL/String.h>                            // COOL String header file
 4
 5   DECLARE List<String>;                              // Declare list of strings
 6   extern Package* SYM_package_g;                     // Pointer to global SYM

 7   int process_rules (List<String>& names, Generic* default_rule) {
 8     int i;                                           // Counter
 9     Symbol* temp;                                    // Temporary variable
10     for (i=0, names.reset(); names.next(); i++) {    // For each rule name
11      temp = SYM_package_g->get (names.value ());     // Get symbol for type
12      temp->set (default_rule);                       // Set default firing rule
13     }
14     return i;                                        // Return rule count
15   }
```

Lines 1 through 3 include the COOL header files for the **Package**, **List**, and **String** classes. Line 5 declares the type of a list of string objects. Line 6 contains an external reference to the pointer to the global `SYM` package object. Lines 7 through 12 define the function `process_rules` that takes two arguments: a reference to a list of strings that are the names of all rule types in the inference engine and a pointer to a default firing rule object. Lines 8 and 9 define two temporary variables. Lines 10-13 contain a loop that uses the current position iterator of the list object to move through all the strings in the list.

Line 11 gets the value of the string at the current position and uses the **get** member function of the package object to look up the character string name and return a pointer to the corresponding symbol object. Line 12 uses the **set** member function of the symbol object to set the value to a pointer to the default firing rule function. This loop continues until all names have been scanned and line 14 returns the number found.

Symbol Package Implementation

11.16 The **symbol_package** macro discussed previously is implemented with the **DEFPACKAGE** and **DEFPACKAGE_SYMBOL** macros and the COOL **MACRO** facility. This section discusses the implementation details of the **symbol_package** macro and should be of interest to programmers who wish to create their own specialized packages or more fully understand the macro capabilities. Others may skip these details.

The **symbol_package** macro is implemented with the COOL macro facilities and can be found in the `~COOL/Package/defpackage.h` header file. The relevant portion of this file is shown below:

```
1   MACRO symbol_package (name, file, REST: options) {
2    DEFPACKAGE name file length = name##_count, options
3    #define expand_##name(index, symbol, type, value) \
4      (&name##_symbols[index])
5    MACRO name (symbol) {
6     DEFPACKAGE_SYMBOL (name, #symbol,,,, expand_##name) }
7    MACRO EXPANDING DEF_##name (symbol, type, value) {
8     DEFPACKAGE_SYMBOL (name, #symbol, type, value,,) }
9    MACRO EXPANDING DEF_##name##_PROPERTY (symbol, property, type, value) {
10    DEFPACKAGE_SYMBOL (name, #symbol, type, value, property,) }
11   extern struct Package* name##_package_g;
12   extern Symbol name##_symbols[];
13  }

14  /* Runtime initialization of a symbol_package        */
15  MACRO implement_symbol_package (name, file) {
16   #include file
17   #if name##_count > 0
18    Symbol name##_symbols[name##_count];
19   #endif
20   #define MAKE_##name##_SYMBOL (index, symbol) \
21    pkg->put (symbol, name##_symbols[index]);
22   MACRO SET_##name##_VALUE (index, type, val) {
23    name##_symbols[index].set ((Generic*) val);}
24   MACRO SET_##name##_PROPERTY (index, prop, type, value) {
25    name##_symbols[index].put(prop, (Generic*) value);}
26   void name##_package_initializer (Package* pkg) {
27    name##_DEFINITIONS (MAKE_##name##_SYMBOL, SET_##name##_VALUE,
                            SET_##name##_PROPERTY)
28   }
29   static Package name##_package_s(name##_count*2,name##_package_initializer);
30   Package* name##_package_g = &name##_package_s;
31  }
```

A symbol package is created and implemented with two macros analogous to the declaration and implementation parts of a parameterized template. The **symbol_package** macro creates macros for adding and manipulating symbol objects. The **implement_symbol_package** macro is used in the `symbols.C` file and actually creates the package object. Lines 1 through 13 contain the declarative macro and lines 14 through 31 contain the implementation macro.

Line 1 starts the declarative macro and takes three arguments. The first, `name`, specifies the name of the package. The second, `file`, specifies the file in which the symbols for the package are to be maintained. The third is a **REST:** argument and may contain any number of options for **DEFPACKAGE**. Line 2 invokes **DEFPACKAGE** with the package name and file arguments, and maintains the number of symbols in the package in the preprocessor symbol `name##_count`, where the package name `name` is used as a prefix to the identifier `_count`.

Lines 3 and 4 define a standard preprocessor macro that, given an index, a symbol name, a type, and a value, returns an offset into a table of symbol objects. Line 5 implements a macro to create or return a symbol object in the package by using **DEFPACKAGE_SYMBOL**. Lines 7-10 implement macros to add or return the value and named property from a symbol in the package. Note that these are **EXPANDING** macros, which means their arguments are first expanded before being passed to **DEFPACKAGE_SYMBOL**. Finally, lines 11 and 12 declare two external objects, a pointer to the global package object `name##_package_g`, and an array of symbol objects `name##_symbols[]`.

Line 15 starts the implementation macro that takes two arguments: the first, `name`, specifies the name of the package; the second, `file`, specifies the file in which the symbols for the package are to be maintained. Line 16 includes the symbol file specified. Line 17 determines if any symbols are actually defined for the package using `name##_count` previously discussed. If there are symbols defined, an array of symbols is created. Lines 20-25 define macros to create and update a symbol object and its value and property list in the package at run time.

Lines 26-28 define a package initializer function. Line 29 creates a global static package object `name##_package_s` whose constructor takes a size and a pointer to a package initializer function. The C++ 2.0 language specification guarantees that the constructor of a global static object will be invoked before calling `main`. The package constructor calls the package initializer function to create and initialize the symbol objects. Finally, line 30 creates a global pointer `name##_package_g` pointing to the newly created package object.

The COOL symbol and package facilities provide an efficient and flexible programmer interface to the slightly more complicated **DEFPACKAGE** and **DEFPACKAGE_SYMBOL** macros. The COOL macro capabilities, combined with the features of C++ and the rules for static object constructor invocation, allow for a direct, although slightly complicated, implementation. The **once_only_package**, **enumeration_package**, and **text_package** macros are implemented in a similar manner. Under most circumstances, a programmer should be able to make use of these interfaces and never need to delve into the details discussed above. However, should a custom package macro be necessary for a specific application, a similar approach is appropriate.

POLYMORPHIC MANAGEMENT 12

Introduction

12.1 The C++ language version 2.0, as specified in the AT&T language reference manual, implements virtual member functions. This delays the binding of an object to a specific function implementation until run time. This delayed (or dynamic) binding is useful where the type of object might be one of several kinds, all derived from some common base class but requiring a specialized implementation of a function. The classic example is that of a graphics editor where, given a base class **graphic_object** from which **square**, **circle**, and **triangle** are derived, specialized virtual member functions to calculate the area are provided. A programmer can then write a function that takes a **graphic_object** argument and determines its area without knowing which of all the possible kinds of graphical objects the argument really is.

While powerful and more flexible than most other conventional programming languages, this dynamic binding capability of C++ is still not enough. Highly dynamic languages such as SmallTalk and Lisp allow the programmer to delay almost all decisions until run time. In addition, facilities are often present for querying an object at run time to determine its type or to request a list of all possible member functions available. These kinds of features are commonly used in many symbolic computing problems tackled today.

COOL supports enhanced polymorphic management capabilities with a programmer-selectable collection of macros, classes, symbolic constants, run time symbolic objects, and dynamic packages. Many of these individual concepts have been discussed in previous sections. This section discusses the **Generic** class that – combined with macros, symbols, and packages – provides efficient run time object type checking, object query, and enhanced polymorphic functionality unavailable in the C++ language. In this section, the following macros, queries, and classes are discussed:

- **Generic** class
- run time type checking
- **TYPE_CASE** macro
- heterogeneous container classes
- **class** macro

Requirements

12.2 This section discusses the **Generic** class and extended polymorphic management facilities of COOL. It assumes that you have a working knowledge of the C++ language and have read and understood Section 10, Macros, and Section 11, Symbols and Packages.

Generic Class

12.3 The **Generic** class is inherited by most other COOL classes and manipulates lists of symbols to manage type information. **Generic** adds to any derived class run-time type checking and object queries, formatted print capabilities, and a describe mechanism. The COOL **class** macro (discussed in paragraph 12.7) automatically generates the necessary implementation code for these member functions in the derived classes. A significant benefit of this common base class is the ability to declare heterogeneous container classes parameterized over the **Generic*** type. These classes, combined with the current position and parameterized iterator class, allow the programmer to manipulate collections of objects of different types in a simple, efficient manner.

The member functions added by **Generic** and the **class** macro to derived COOL classes manipulate symbols stored in the global **SYM** package. These symbols reflect the inheritance tree for a specific class. They may have optional property lists containing information that associates supported member functions with their respective argument lists. User-defined classes derived from **Generic** are also automatically supported in an identical fashion, resulting in additional symbols in the global symbol package. As discussed earlier, these symbols must have storage allocated for them and code to initialize the package at program startup time. This is managed by the COOL file `symbols.C` that should be compiled and linked with every application that uses COOL. An automated method for ensuring correct package setup and symbol initialization is shown in the make files associated with the example programs for this manual.

NOTE: All applications using COOL must have a copy of `symbols.C` linked into the final executable program. See the make file in the `~COOL/examples` subdirectory for a mechanism to automate this procedure.

Name: **Generic** — Base class supporting run time object typing and query

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

Base Classes: None

Friend Classes: None

Protected Constructors:

Generic ();

There are no public constructors for a **Generic** object. You can only create a pointer to a **Generic** object. Since **Generic** has no private or public data (as a pure virtual base class), it is actually used as an implementation requirements guide for derived classes.

Protected Member Functions:

virtual void print (**ostream&** *os*) **const**;

Utility member function used by the overloaded output operator to provide a default print capability for all classes derived from **Generic**. This intermediate function is required since friend functions cannot be virtual.

int select_type_of (**const Symbol**** *sym_list*) **const**;

Supports an efficient type-case macro (discusses later) by examining the **NULL**-terminated *sym_list* of symbols passed as an argument (from **type_list**) and returns an integer index of the matching type symbol if found; otherwise, this function returns –1.

virtual Symbol type_list**() **const**;
Returns a **NULL**-terminated array whose first element is a pointer to a symbol representing the type of object. The remaining elements of the array are pointers to the **symbol** type lists of the base classes.

Member Functions

virtual void describe (**ostream&** *os*);
Uses the **map_over_slots** member function to display the data members and their types of the object on the specified stream *os*.

Boolean is_type_of (**Symbol*** *sym*) **const**;
Type checking predicate that returns **TRUE** if the object is of type *sym* or inherits from that type somewhere in the class hierarchy; otherwise, this predicate returns **FALSE**.

virtual Boolean map_over_slots (**Slot_Mapper** *sm*, **void*** *rock*=**NULL**);
Calls the mapping function *sm* on every data member in the object and returns **TRUE** if all calls return **TRUE**; otherwise, this function returns **FALSE**. The *rock* argument is a pointer to some arbitrary piece of data for optional use by the mapper function. *sn* is a function of type **Boolean** (*Slot_Mapper*)(*Generic**, *char**, *void**, *Symbol**, *void**), where *Generic** is a pointer to the object, *char** is a character string representation of the data member name, *void** is a pointer to the data member value, *Symbol** is a symbol table entry for the data member type, and *void** is the miscellaneous programmer-defined optional data value.

inline Symbol* type_of () **const**;
Returns a pointer to the type symbol associated with an object.

Friend Functions:

Boolean compare_types (**Symbol**** *type_list*, **Symbol*** *sym*);
Searches *type_list* for *sym* and returns **TRUE** if found; otherwise, this function returns **FALSE**. This function is used by the **is_type_of** member function.

int compare_multiple_types (**Symbol**** *sym_list1*, **Symbol**** *sym_list2*);
Searches *sym_list1* for any symbol match against *sym_list2* and returns **TRUE** if found; otherwise, this function returns **FALSE**. This function is used by the **select_type_of** member function.

friend ostream& operator<< (**ostream&** *os*, **const Generic&** *g*);
Overloads the output operator for a reference to a generic object and calls the protected virtual **print** member function to provide a default output capability for all classes derived from **Generic**.

friend ostream& operator<< (**ostream&** *os*, **const Generic*** *g*);
Overloads the output operator for a pointer to a generic object and calls the protected virtual **print** member function to provide a default output capability for all classes derived from **Generic**.

Run Time Type Checking Example

12.4 One of the simplest and most useful features facilitated by **Generic** is the run-time type checking capability. The **type_of** and **is_type_of** virtual member functions accomplish this. The following code fragment provides an example of the kind of run time type query available for an object that is derived at some point from the COOL **Generic** class. A more complete example is in the discussion on heterogeneous container classes.

The parameterized **Vector**<*Type*> class is derived from the type-independent **Vector** class, which is in turn derived from **Generic**. Similarly, the **List**<*Type*> class is derived from **List,** which is derived from **Generic**. Suppose a general-purpose function in an application is written that at some point needs to determine the type of the object being manipulated and respond appropriately. If there are many possibilities, the **TYPE_CASE** macro discussed later might be appropriate. If there are few, the following mechanism can be used:

```
 1    void foo (Generic* g) {
 2      ....                                     // Some processing
 3      if (g->is_type_of(SYM(Vector)))          // If derived from Vector
 4                 ....                          // Go do something
 5      else if (g->is_type_of(SYM(List)))       // Else if from List
 6                 ....                          // Go do something
 7      else {                                   // Else something else
 8                 ....                          // Do something else
 9      }
10      ...                                      // Sometime later
11      cout << "Object is a " << g->type_of();  // Output type
12      }
```

Lines 1 through 12 contain a code fragment that queries the type of object pointed to by a **Generic*** argument. Lines 3 and 5 are similar and use the virtual **is_type_of** member function that takes a **Symbol** as an argument to determine if the object is an instance of a class or is derived at some point from that class. Note that since **Vector**<*Type*> is derived from the **Vector** class, the application merely queries to see if this object is of type **Vector**, not of `Vector<int>`. The more specified version could also be used as the symbol representing the class. Presumably, the programmer will perform some type-specific operation on lines 4 and 6 as appropriate. If the object is neither a vector or a list, some default action is performed. Similarly, line 11 uses the **type_of** member function and the overloaded output operator to send the class type name of the object (that is, the symbol name for the class) to the standard output stream. In all cases, the function bindings for theses operations are determined at run time, not compile time.

TYPE_CASE Macro

12.5 Type determination and function dispatch can become quite tedious if there are many types of objects. Ideally, each would be derived from a common base and include a virtual member function for each important operation that might be required. However, it is sometimes not feasible to have such a situation, especially with a high number of objects or member functions. The **TYPE_CASE** macro provides an alternate scheme to do this.

The following code fragment shows an abbreviated function that takes a single argument of a pointer to a **Generic** object. This function uses the **TYPE_CASE** statement to dispatch some particular member function call based upon the type of the object. This might be useful in a situation where every object that inherits from **Generic** does not implement the same functions, but rather has a specialized subset appropriate for that object only. For example, `foo` might want to modify the elements of the COOL **Vector** and **List** classes in a different manner.

```
 1   void foo (Generic* g) {
 2     TYPE_CASE (g) {
 3     case Vector:                          // If the object is a vector
 4     ....                                  // Do something for Vector
 5      break;
 6     case List:                            // If the object is a list
 7     ....                                  // Do something for List
 8      break;
 9      default:                             // Else do the rest
10     }
11     cout << "Object is a " << g->type_of();    // Output type
12   }
```

Lines 1 through 12 implement the same operation as the previous example but this time use the **TYPE_CASE** macro instead of **is_type_of** and **type_of**. Line 2 begins a macro analogous to the C++ `switch` statement. It gathers all possible cases and allows the user to symbolically dispatch on the type of object represented by the case statements. This automates some of the symbol collection and manipulation required with the earlier example. Yet another variation is discussed later using hooks available to the programmer with the **class** macro.

Heterogeneous Container Example

12.6 As a final example, the polymorphic capabilities available with **Generic** and its associated functions and macros can implement heterogeneous container classes. A heterogeneous container class can contain many types of objects. For example, the graphics editor mentioned earlier might store all instances of graphic objects in a list, regardless of whether they are circles, squares, or dodecahedrons. The example below creates a list of pointers to **Generic** objects and uses the virtual member functions associated with both the derived classes and the COOL **Symbol** class to accomplish what would otherwise be a relatively difficult task:

```
1     #include <COOL/String.h>                    // COOL String class
2     #include <COOL/Date_Time.h>                 // COOL DateTime class
3     #include <COOL/List.h>                      // COOL List class

4     DECLARE List<Generic*>;                     // Define list of Generic*
5     IMPLEMENT List<Generic*>;                   // Implement list of Generic*

6     class my_class : public Generic {
7     private:
8      int i;
9     public:
10      my_class (int value) {
11       this->i = value;
12     }
13      int& get() {
14       return this->i;
15      }
16      friend ostream& operator<< (ostream& os, my_class* m) {
17       os << m->get();
18       return os;
19      }
20      friend ostream& operator<< (ostream& os, my_class& m) {
21       os << m.get();
22       return os;
23      }
24    };

25    void process_list (List<Generic*>& g) {
26     for (g.reset(); g.next(); ) {
27      cout << "Item is a '" << ((g.value())->type_of())->name() << "' ";
28      cout << "and its value is: " << g.value() << "\n";
29     }
30    }

31    int main () {
32     String s1 ("This is a string object");      // Initialize string object
33     set_default_country(SWEDEN);                // Set Sweden country code
34     set_default_time_zone(WET);                 // Western Europe time zone
35     Date_Time d1;                               // Declare DateTime object
36     d1.parse("5:44pm 86-10-15");                // Parse a date/time string
37     my_class m1(3);                             // Initialize my_class object
38     List<Generic*> lg (3, &s1, &d1, &m1);       // List with 3 generic objects
39     process_list (lg);                          // Iterate through list
40     return 0;                                   // Exit with valid return code
41    }
```

Lines 1-3 include three COOL classes, and lines 4 and 5 implement a list of pointers to generic objects. Lines 6-24 declare and implement a new simple class `my_class`, derived from the **Generic** class. Lines 25-30 are the heart of this polymorphic example. A function, `process_list`, is declared that takes one argument, a reference to a list of pointers to generic objects. Lines 26-29 implement a loop using the current position iterator built into the COOL **List**<*Type*> class to access all elements of the list. Line 27 uses the **type_of** member function to return a pointer to the Symbol object representing the type of the value of the object at the current position in the list. The **name** function of **Symbol** is used to return the name so it can be printed. Line 28 outputs the value of the object at the current position in the list.

Lines 31 through 41 constitute the main body of the program. Line 32 declares a **String** object and initializes it with a character string value. Lines 33 through 36 declare a **Date_Time** object whose value is set to the local system time formatted for Sweden in Western European Time. Line 37 declares an instance of **my_class** with an integral value of three. Line 38 declares an instance of the list of pointers to a generic object with three values, the address of the string, date/time, and `my_class` objects. Line 39 calls the `process_list` function to output the types and values of the objects in the list. Finally, line 41 ends the program with a valid exit code.

The output of this program is shown below:

```
1   Item is a 'String' and its value is: This is a string object
2   Item is a 'Date_Time' and its value is: Sweden 1986-15-10 17.44.00 WET
3   Item is a 'my_class' and its value is: 3
```

As can be seen from the preceding output, this program was successful in querying each object in the list for its type, printing the name of that type, and outputting the value to the standard output stream. Line 1 shows the type and value of the **String** object, line 2 shows the type and value of the **Date_Time** object, and line 3 shows the type and value of the application-specific object.

Class Macro

12.7 The **class** keyword is implemented as a COOL macro to add symbolic computing abilities to class definitions. It takes a standard C++ class definition and, if the class contains **Generic** somewhere in its inheritance hierarchy, it generates member functions for support of run time type checking and query. In addition, a symbol for the derived **Generic** class type is added to the COOL global symbol package `SYM`. The **class** macro also has two hooks, allowing a programmer to customize the results. The actual code, which is expanded in a class definition and after a class definition, is controlled by the **classmac** macro that **class** calls.

The **classmac** macro allows data member and member function hooks to be specified by user-defined macros. There may be more than one **classmac** macro hook specified by the programmer. COOL has several, and other user-defined macros are simply chained together in a calling sequence ordered according to order of definition. Each **classmac** macro defines how the **class** macro should expand the class definition. The **class** macro does not actually generate the code itself. This is defined in user-modifiable header files that specify a **classmac** macro. For example, a general-purpose mechanism that automatically creates accessor member functions to get and set each data member can be created by defining a **classmac** macro that is attached to the data member hook of the **class** macro (see the following example). No changes to the COOL preprocessor are required.

A user-defined combination of data members and member functions of a class definition are passed as arguments to macros that can be changed or customized by the application programmer. The virtual **map_over_slots** member function takes a pointer to a function as one of its arguments. Each data member selected is passed to this procedure, providing the customization point for the user. The COOL **Generic** class uses the data member hook to implement the **map_over_slots** member function.

Name: **classmac** — User-definable class macro

Synopsis: **classmac** (*name,* **REST:** *args*);

name	Name of macro to call

args One or more of the following comma-separated arguments:

arg = macro_name
Calls *macro_name* on the preceding type of argument

inside
Expands the macro inside the class definition

outside
Expands the macro outside the class definition

slots
Evaluates the macro for data members in the class

methods
Evaluates the macro for member functions in the class

virtual
Evaluates the macro for virtual member functions only

inline
Evaluates the macro for inline member functions only

normal
Evaluates the macro for non-inline, non-virtual member functions only

private
Evaluates the macro for private data members or private member functions only

protected
Evaluates the macro for protected data or protected member functions only

public
Evaluates the macro for public data or public member functions only

---

The *arg=macro_name* option allows the programmer to specify the name of a macro to call on arguments of the preceding type. This is typically used to specify the name of the macro to call for either the data members or member functions, as in the following example. If neither the *inside* nor *outside* arguments are specified, the macro will be expanded outside and after the class definition. Either the *slots* or *methods* keyword must be specified, but not both. If neither the *virtual*, *inline*, nor *normal* keywords are specified, all member functions in the class are used. If neither the *private*, *protected*, nor *public* keywords are specified, all data members and member functions in the class are used.

Class Macro Example

12.8 The following example shows a mechanism to automatically generate a member function accessor for each private data member in a class. This is performed for any class that inherits from **Generic** in its inheritance tree and in an environment where the **classmac** data member hook macro shown has been defined. This operation is not performed by default for COOL, but rather requires explicit programmer action. The following lines contain several macros and a skeleton class definition to pass through the preprocessor:

```
1   #pragma defmacro classmac "classmac" delimiter=)
2   classmac (generate_slot_accessors, inside, slots=slot_accessor)

3   MACRO generate_slot_accessors (class_name, base_class, BODY: methods) {
4     methods }

5   MACRO slot_accessor (type, name, value) {
6     const type& get_##name() { return name }
7   }

8   class foo: public Generic {
9    private:
10    int* data;                                  // Pointer to allocated storage
11    char *a, *b, c;                             // Three miscellaneous variables
12    int size;                                   // Size of foo object
13    void grow (int new_size);                   // Private function to grow foo
14   public:
15    foo (int);                                  // Constructor with size
16    ~foo ();                                    // Destructor
17    int& operator[] (int);                      // Operator[] overload for Type
18    Boolean find (const int&);                  // Find element in foo
19  };
```

Line 1 instructs the preprocessor to recognize the COOL macro **classmac** and to call the internal preprocessor macro **classmac**. The terminating delimiter of this macro is a closing parentheses, which means that all input from the **classmac** keyword up to and including a matched, right parenthesis will be passed to and processed by the macro. Line 2 tells the **classmac** macro to call the `generate_slot_accessors` macro for each data member in the class definition and place the expanded macro results inside the definition. Note the `slots=slot_accessor` argument that ensures that each data member will be processed by the named macro passed through the **BODY:** argument.

Lines 3 and 4 define the `generate_slot_accessors` macro. **classmac** passes this macro the class name, the base class name, and the **BODY:** argument `slot_accessor` as specified by the `slots` option on line 2. Lines 5 through 7 define a macro `slot_accessor` of type *(Symbol*, Symbol*, char*)* where the first argument is a symbol representing the type, the second argument is a symbol representing the name, and the third argument is a character string of the arguments or initial values. These arguments and their order are always passed by the **classmac** macro to all data member and member function macros specified by the user. Line 6 contains the line of code that gets generated for the accessor function with argument names substituted appropriately. Lines 8 through 19 declare a simple class with several data members.

The preprocessor expands the macros and generates the following:

```
1     class foo :public Generic{
2     private:
3       int* data;                          // Pointer to allocated storage
4       char *a, *b, c;                     // Three miscellaneous variables
5       int size;                           // Size of foo object
6       void grow (int new_size);           // Private function to grow foo
7     public:
8       foo (int);                          // Constructor with size
9       ~foo ();                            // Destructor
10      int& operator[] (int);              // Operator[] overload for Type
11      Boolean find (const int&);          // Find element in foo
12      const int*& get_data() { return data }
13      const char*& get_a() { return a }
14      const char*& get_b() { return b }
15      const char*& get_c() { return c }
16      const int& get_size() { return size }
17    };
```

Lines 1 through 11 are the same as before entering the preprocessor and contain the class definition as specified by the programmer. Lines 12 through 16 contain inline accessor member functions generated by the macros specified. These were added inline as a result of the *inside* specifier on the **classmac** macro directive for `generate_slot_accessors`.

EXCEPTION HANDLING

13

Introduction

13.1 The **Exception** class, its derived classes, **Excp_Handler** class, and the exception interface macros offer programmers an easy means of reporting and handling exceptions in an application. This section discusses the base **Exception** and **Excp_Handler** classes. It also covers predefined exceptions and exception handlers, referencing exceptions as symbols in a package, exception group names (aliases), the report message text package, and user-defined exceptions.

Requirements

13.2 This section assumes that you have a working knowledge of C++ and have read and understood Section 10, Macros, and Section 11, Symbols and Packages.

Exceptions

13.3 In COOL, program anomalies are known as exceptions. An exception can be an error, but it can also be a problem such as impossible division or information overflow. Exceptions can impede the development of object-oriented libraries. Exception handling offers a solution by providing a mechanism to manage such anomalies and simplify program code.

The C++ exception handling scheme is a raise, handle, and proceed mechanism similar to the Common Lisp Condition Handling system. When a program encounters an anomaly that is often (but not necessarily) an error, it has the following options:

- Represent the anomaly in an object called an exception
- Announce the anomaly by raising the exception
- Provide solutions to the anomaly by defining and establishing handlers
- Proceed from the anomaly by invoking a handler function

The COOL exception handling facility provides an exception class (**Exception**), an exception handler class (**Excp_Handler**), a set of predefined exception subclasses (**Warning**, **Error**, **Fatal**, and so on**)**, and a set of predefined exception handler functions. In addition, the macros **EXCEPTION**, **RAISE**, **STOP**, and **VERIFY** allow the programmer to easily create and raise an exception at any point in a program.

When an exception is raised (through macros **RAISE** or **STOP**, for example), a search begins for an exception handler that handles this type of exception. An exception handler, if found, deals with the exception by calling its exception handler function. The exception handler function can correct the exception and continue execution, ignore the exception and resume execution, or end the program. In COOL, an exception handler for each of the predefined exception types exists on the global exception handler stack.

An exception handler invokes a specific exception handler function for a specific type of exception or group of exceptions. Handling an exception means proceeding from the exception. An exception handler function could report the exception to standard error and end the program, or drop a core image for further debugging by the programmer. Another way of proceeding is to query the user for a fix, store the fix in the exception object, and return to where the exception was raised.

When an exception handler object is declared, it is placed on the top of a global exception handler stack. When an exception is raised, a search is made for an exception handler. The handler search starts at the top of the exception handler stack, with the most recently defined exception handler at the top of the stack. An exception handler function is called if a match is found between the exception type or group name of the exception raised, and a handler function on the exception handler stack.

The COOL exception handling facility provides several macros that simplify the process of creating, raising, and manipulating exceptions. These macros are implemented with the COOL macro facility discussed in Section 10, Macros. The **EXCEPTION** macro simplifies the process of creating an instance of a particular type of exception object. The **RAISE** macro allows the programmer to easily raise an exception and search for an exception handler. The **STOP** macro is similar to the **RAISE** macro, except that it guarantees to end the program if the exception is not handled. The **VERIFY** macro raises an exception if an assertion for some particular expression evaluates to **FALSE**. Finally, the **IGNORE_ERROR**S macro provides a mechanism to ignore an exception raised while executing a body of statements.

The COOL exception handling mechanism supports the concept of group names or aliases for classes of exceptions that require no specialization of the exception object, but do require a distinct name to provide a specific exception handler. For example, an index exception class to handle indexing range errors in a vector class could be defined with aliases established for **get** and **set** operations. The appropriate **get** and **set** member functions set the alias of the exception object as necessary, and provide a specialized exception handler. If an indexing exception is detected at some point by one of these member functions, the appropriate handler function can be invoked. As a result, two different exception handlers can be used while only one *type* of index exception object is required.

Exception Class

13.4 The **Exception** class reports exceptions through a message prefix and a format string. Both are implemented as public data members in the exception object. An exception handled status data member is also used to determine if the exception was handled. All of these are implemented as public data members, which makes access easier by the exception handler functions and the **EXCEPTION** macro. In addition, a list of exception types is maintained in a group names data member to support aliasing and subclassing of exception types. The **Exception** class includes member functions for reporting a message (using the message prefix and format string) on a specified output stream, determining if the exception object is of a particular type or group, and setting the exception handled status. Finally, it also includes a member function that searches for a handler to invoke on this exception type.

Classes derived from the **Exception** class save the state of the situation and communicate this information to exception handlers. When an exception can be fixed and proceeding from the exception is possible, information on how the exception handler proceeded from the exception is also stored in the exception object by the invoked exception handler function.

Note: The *excp_type* arguments in the **Exception** class constructors and member functions are pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` package, or some application-specific package. This is discussed in the next paragraph, Excp_Handler Class, and illustrated in the example in paragraph 13.6, Excp_Handler Example. See section 11, Symbols and Packages, for more information.

Name: **Exception** — The base class for building exception objects.

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

Base Class: **Generic**

Friend Classes: None

Constructors:

Exception ();
Creates an exception object, initializes the format message and message prefix data members to **NULL**, and sets the exception handled flag to **FALSE**.

Exception (**Symbol*** *excp_type*);
Creates an exception object, creates a group name *excp_type* if necessary, associates this exception object with the *excp_type* group name, initializes the format message and message prefix message data members to **NULL**, and sets the exception handled flag to **FALSE**.

Exception (**int** *number*, **Symbol*** *excp_type1*, **Symbol*** *excp_type2*, ...);
Creates an exception object, creates *number* group names *excp_type1*, *excp_type2*, and so on if necessary, associates this exception object with group names *excp_type1*, *excp_type2*, and so on, sets the format and message prefix data members to **NULL**, and sets the exception handled flag to **FALSE**.

Member Functions:

virtual void default_handler ();
Default exception handler called when this type of exception is raised if no user-specified exception handler is found. This function does not set the exception handled flag in the exception object.

inline void handled (**Boolean** *handled*);
Sets the exception handled flag to *handled*.

inline Boolean is_handled () **const;**
Returns **TRUE** if the exception was handled; otherwise, return **FALSE**.

Boolean match (**Symbol*** *excp_type*);
Returns **TRUE** if this exception object is in the group name *excp_type*; otherwise, this function returns **FALSE**.

const char* message_prefix () **const;**
Returns the message prefix.

virtual void raise ();
Invoked to search for an exception handler when an exception is raised. If found, the associated handler function is called and the exception handled flag is set to **TRUE**; otherwise, this function sets the exception handled flag to **FALSE**. If the exception handler function returns or no exception handler is found, the program resumes execution at the point at which the exception was raised.

virtual void report (*ostream& os*) **const;**
Reports the exception message on the output stream *os*. The exception handler functions and the output operator function of **Exception** call this member function.

void set_group_names (**Symbol*** *excp_type*);
Creates a group name *excp_type* in the COOL SYM package if necessary, and associates this exception object with the *excp_type* group name.

void set_group_names (**int** *number*, **Symbol*** *excp_type1*, **Symbol*** *excp_type2*, ...);
Creates *number* group names *excp_type1*, *excp_type2*, and so on in the COOL SYM package if necessary, and associates this exception object with group names *excp_type1*, *excp_type2*.

virtual void stop ();
Invoked to search for an exception handler when an exception is raised. If found, the associated handler function is called and the exception handled flag is set to **TRUE**; otherwise, this function sets the exception handled flag to **FALSE**. This function is identical to **raise** except that if the exception handler function returns or no exception handler is found, program execution is terminated.

Friend Functions:

friend ostream& operator<<(**ostream&** *os*, **const Exception*** *excp*)
Overloads the output operator to provide a formatted output capability for a pointer to an exception object *excp*.

friend ostream& operator<< (**ostream&** *os*, **const Exception&** *excp*);
Overloads the output operator to provide a formatted output capability for a reference to an exception object *excp*.

Excp_Handler Class

13.5 An exception handler provides a way to proceed from a particular type of exception by calling its exception handler function. An exception handler function could handle the exception by reporting the exception to standard error and ending the program, or dropping a core image for further debugging by the programmer. Another way of proceeding is to query the user for a fix, store the fix in the exception object, and return to the point where the exception was raised.

An instance of the **Excp_Handler** class is specified for a particular type of exception or one or more exception group names with an associated exception handler function. Such an instance invokes the specific exception handler function when an exception of the appropriate type is raised. The **Excp_Handler** class also contains data members that point to the top exception handler on the global exception handler stack and the next exception handler after itself. When an exception handler object is instantiated, it is placed at the top of the exception handler stack. When an exception is raised, the exception stack is searched from the top for an appropriate handler. When one is found, it is invoked and the exception object is passed as an argument. What action the exception handler function takes is determined by the type of exception and is discussed in paragraph 13.7, Predefined Exception Types and Handlers.

Name: **Excp_Handler** — The class for handling exceptions.

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

Base Class: **Generic**

Friend Class: **Exception**

Constructors:

Excp_Handler ()
Creates an exception handler object with defaults for the exception type and exception handler function and pushes itself on top of the global exception handler stack. The default exception type is **Error** and the default exception handler function is **void exit_handler**(*Exception**).

Excp_Handler (**Excp_Handler_Function** *fn*, **Symbol*** *excp_type*)
Creates an exception handler object associated with the exception type *excp_type*, initializes the exception handler function data member to *fn*, and pushes itself on top of the global exception handler stack. The exception handler function is of type **void** (*Excp_Handler_Function*)(*Exception**).

Excp_Handler (**Excp_Handler_Function** *fn*, **int** *number*,
    Symbol* *excp_type1*, **Symbol*** *excp_type2*, ...);
Creates an exception handler object, creates *number* group names *excp_type1*, *excp_type2*, and so on if necessary, associates this exception handler object with *number* group names *excp_type1*, *excp_type2*, and so on, initializes the exception handler function data member to *fn*, and pushes itself on top of the global exception handler stack. The exception handler function is of type **void** (*Excp_Handler_Function*)(*Exception**).

Member Functions:

virtual Boolean invoke_handler (**Exception*** *excp*)
Returns **TRUE** if the exception handler function was invoked for *excp*; otherwise, this function returns **FALSE**.

Excp_Handler Example

13.6 The following example shows a function that establishes an exception handler function for exceptions associated with a group name `File_Error` of type `My_Exceptions`. It then attempts to open each file indicated in an array of pointers to file name character strings.

```
 1   #include <COOL/Exception.h>                                  // Include header
 2   #include <My_Exceptions.h>                                   // My exception types
 3   extern void my_file_handler (My_Exceptions* excp);           // Exception handler

 4   FILE* open_f (char* file, char* mode) {
 5     FILE* temp;                                                // Temp variable
 6     if ((temp = fopen (file, mode)) == NULL) {                 // File open OK?
 7      My_Excp1 (SYM(File_Error)) excp;                          // Create exception
 8      excp->fname = file;                                       // Set file name
 9      excp_>fmode = mode;                                       // Set file mode
10      excp->raise ();                                           // Raise exception
11     }
12   }

13   Boolean open_files (char** file_names, char** modes, FILE** f_handles) {
14     Excp_Handler eh (my_file_handler, SYM(File_Error));        // Setup handler
15     for (int i = 0; file_names[i] != NULL; i++)                // For each file
16      f_handles[i] = open_f (file_names[i], modes[i]);          // Open file
17   }
```

Line 1 includes the COOL **Exception** header file. Line 2 includes an application-specific header file that defines exception types derived from **Exception**. Line 3 is an external reference to some user-defined function to be called for exceptions of type `My_Exceptions`. For example, this function might prompt the user for a new file name and perform a retry operation. Lines 4 through 12 implement a function that attempts to open `file` in `mode` with the system function **fopen**. If the open fails, an exception `My_Excp1` associated with group name `File_Error` is created and raised. Line 7 uses the COOL `SYM` package in which to store the group name symbol. In a typical application, all application-specific symbols should be located in an application-specific package. Lines 8 and 9 set two public data member slots in the exception object, and line 10 raises the exception.

Lines 13 through 17 contain a function `open_files` that loops through an array of `file_names` and attempts to open each file in the function `open_f`. Line 14 is the heart of this function, where an exception handler object `eh` is created with a pointer to the function `my_file_handler` for exceptions of group name `File_Error`. This symbol is located in the COOL `SYM` package and would be referenced when an exception of type `My_Excp1` is raised, as in lines 7 through 10. See section 11, Symbols and Packages, for more information on the COOL symbol and package mechanism.

In this example, the exception handler `my_file_handler` is associated with the exception handler object `eh` created locally on line 14. When the constructor for `eh` is executed, a pointer to the exception handler object is placed on the global exception handler stack. While this object is in scope and not pre-empted by a more specific handler, any exception raised asssociated with the group name `File_Error` will be handled. When function `open_files` completes and destructor for `eh` called, the handler is removed from the global exception handler stack.

Predefined Exception Types and Handlers

13.7 COOL provides six predefined exception classes and five default exception handlers. Each of the predefined exception types has a default exception handler member function. The following rules apply in determining which handler function should be invoked for a particular type of exception:

- If no exception handler is found and the exception is of type **Error** or **Fatal**, its error message reports on the standard error stream and the program ends.
- If the exception is of type **Warning**, the warning message reports on the standard error stream and the program resumes at the point where the exception was raised.
- If the exception is of type **System_Error**, the system error message reports on the standard error stream and the program ends.
- If the exception is of type **System_Signal**, the signal error message reports on the standard error stream and the program resumes at the point where the system function **signal**() was called.
- If the exception is of type **Verify_Error**, the expression that failed assertion reports on the standard error stream and the program ends.

Exception is the base exception class and from it are derived **Warning, System_Signal, Fatal,** and **Error**. The **System_Error** and **Verify_Error** classes are derived from the **Error** class. The default exception handlers are called only if no other exception handler is established and available when an exception is raised.

For exceptions of type **Error** and **Fatal**, the exception handler reports the error message of the exception on standard error and ends the program. Exceptions of the type **Warning** report a warning message on standard error and return to the point at which the exception was raised. Exceptions of type **System_Error** report an error message on standard error and end the program. Finally, exceptions of type **System_Signal** report an error message on standard error and the program resumes execution at the point at which the system function **signal** was called. The following functions report exceptions and deal with them:

void Fatal::default_handler ();
This member function reports the exception message on the standard error stream and ends the program with a call to **abort**(), generating a core image that can be used for further debugging purposes.

void Error::default_handler ();
This member function reports the exception message on the standard error stream and ends the program normally with a call of **exit**(1).

void System_Error::default_handler ();
This member function reports the exception message on the standard error stream, sets the global system **errno** variable appropriately, and ends the program with a call to **abort**(), generating a core dump that can be used for further debugging purposes.

void System_Signal::default_handler ();
This function reports the exception message on the standard error stream and returns to the point at which a call to the system **signal()** function was made.

void Warning::default_handler ();
This function reports the exception message on the standard error stream and returns to the point at which the exception was raised.

EXCEPTION

13.8 The **EXCEPTION** macro simplifies the process of creating an instance of a particular type of exception object. It provides an interface for the application programmer to create an exception object using the specified arguments to indicate group name(s), initialize data members, or generate a format message. There are many variations of **EXCEPTION** that provide flexible and efficient means of customizing the exception object. In particular, the variable number of group name arguments should reduce the need for many types of exception classes whose only difference is the type name.

NOTE: The **EXCEPTION** macro takes some arguments that are actually pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` and `ERR_MSG` packages, or some application-specific package. See section 11, Symbols and Packages, for more information.

Name: **EXCEPTION** — A COOL macro for constructing an exception object

Synopsis: **EXCEPTION** (**Symbol*** *excp_type*, **REST:** *args*);

excp_type A symbol representing the **Exception** class type (that is, **Error**, **Warning**, and so forth)

args One or more of the following comma-separated arguments or values:

Symbol* *group_name*
One or more comma-separated pointers to **Symbol** objects representing aliases for this exception class type

const char* *format_string*
A character string compatible with the standard **printf** format containing the text of the error message

format_args
Any required argument(s) for the format string

key_value_args
The name(s) and value(s) of any public data members in the exception object

---

EXCEPTION Examples

13.9 Here are three examples of the use of the **EXCEPTION** macro. Each makes use of a different form of the macro to show alternate features and usage. `Exception_g` is a global exception object pointer. `hprintf()` is a variation of the **printf** function which returns a format string allocated on the heap. Both of these are provided as part of the COOL exception handling facility. In addition, notice that the ordering of the *format_args* and *key_value_args* arguments in example two depends on the control characters in the format string. Finally, the code resulting from the macro expansion makes heavy use of the comma operator and is standard C++, although this might look a little confusing at first.

Example 1: This is a simple use of **EXCEPTION** that specifies the exception type, a group name, and a format string:

```
1    EXCEPTION (Error, SYM(Serious_Error), "Serious problem here");
```

Line 1 contains an invocation of the `EXCEPTION` macro for an exception of type `Error` aliased with the group name `Serious_Error`. This group name symbol is reference through the COOL `SYM` package. The message text follows as the third argument. When expanded, this macro call generates:

```
2    (Exception_g = new Error(),
3     Exception_g.set_group_name (SYM(Serious_Error)),
4     Exception_g->format_msg = hprintf(ERR_MSG("Serious problem here.")),
5     Exception_g);
```

Line 2 assigns the global pointer `Exception_g` to point to a new instance of an `Error` exception object. Line 3 associates this exception object with the group name `Serious_Error`. Line 4 initializes the format message field to the message argument passed. Note that this message is actually a symbol in the COOL `ERR_MSG` package, thus facilitating collection of all error messages in one location, and affording the ability to have multiple translations of text for a single application. Line 5 returns a pointer to the new exception object.

An exception handler on the global exception handler stack that is associated with the group name `Serious_Error` will be called if this exception object is raised. This can be done through the virtual **Exception::raise** member function, or more conveniently with the **RAISE** macro discussed in paragraph 13.10 below.

Example 2: In this example, a new exception class is derived containing two data members whose values are filled in when the exception object is created. **EXCEPTION** is invoked with an exception type, a format string, and a mix of data member arguments and format arguments.

```
1    Class Bad_Argument_Error : public Fatal {
2    public:
3      char* arg_name;
4      int arg_value;
5      Bad_Argument_Error ();
6    };

7    EXCEPTION (Bad_Argument_Error,
8     ERR_MSG("Argument %s has value %d that is out of range for vector %s "),
9     arg_name="foo", arg_value=x, vec1);
```

Lines 1 through 6 define a new exception class, `Bad_Argument_Error,` derived from the COOL **Fatal** class. This new exception type has two public data members, `arg_name` and `arg_value`, whose values will be provided when creating an instance of this type. Line 7 invokes **EXCEPTION**, specifying the exception type `Bad_Argument_Error` as the first argument. Notice there are no group names in this invocation. As a result, only an exception handler specifically created for exceptions of type `Bad_Argument_Error` can be called if a `Bad_Argument_Error` exception is raised. Line 8 contains the second argument, which is the error message control string, in standard **printf** format. Line 9 contains intermixed format string arguments and data member initialization arguments. When expanded, this macro generates:

```
1   (Exception_g = new Bad_Argument_Error(),
2   Exception_g->arg_name = "foo",
3   Exception_g->arg_value = x,
4   Exception_g->format_msg = hprintf(ERR_MSG("Argument %s has value %d\
5   which is out of range for vector %s."), "foo", x, vec1),
6   Exception_g);
```

Line 1 assigns the global pointer `Exception_g` to point to a new instance of a `Bad_Argument_Error` exception object. Lines 2 and 3 initialize the public data members of the exception object. Lines 4 and 5 initialize the format message field to the message argument passed, with the appropriate argument values inserted. Note that this message is actually a symbol in the COOL `ERR_MSG` package. Line 6 returns a pointer to the new exception object.

An exception handler for a `Bad_Argument_Error` exception could prompt the user for a new value for the named argument and return it in `arg_value` field if this exception object is raised. This can be done through the virtual **Exception::raise** member function or more conveniently with the **RAISE** macro discussed in paragraph 13.10 below.

Example 3: This example is similar to the previous one, except that the constructor for the new exception type object initializes the format message field. This is a general-purpose exception type for any container class derived from the COOL **Generic** class as discussed in Section 12, Polymorphic Management. Providing a local **report** member function supercedes the virtual default implementation in the base **Exception** class.

```
1   Class Out_of_Range : public Fatal {
2   public:
3     int value;
4     Generic* container;
5     Out_of_Range() {
6      format_msg = "Value %d is out of range for container %s."
7     }
8     void report(ostream& os) {
9      Fatal::report (os);
10     os << form (format_msg, this->value, this->container->type_of());
11    }
12  };
```

Lines 1 through 12 define a new exception class `Out_of_Range` derived from the COOL **Fatal** class. This new exception type has two public data members, `value` and `container`, whose values will be provided when creating an instance of this type. Lines 5 through 7 define the constructor for the new exception type that initializes the format message data member. Lines 8 through 11 implement a specialized **report** member function. It uses the polymorphic **type_of** member function of the container class inherited from **Generic**.

```
1       EXCEPTION(Out_of_Range, value=n, container=c1);
```

At some point in an application, line 1 invokes **EXCEPTION**, specifying the exception type `Out_Of_Range` as the first argument and intermixed format string arguments and data member initialization arguments of `value` and `container`. When expanded, this macro generates:

```
1       (Exception_g = new Out_of_Range(),
2        Exception_g->value = n,
3        Exception_g->container = c1,
4        Exception_g);
```

Line 1 assigns the global pointer `Exception_g` to point to a new instance of an `Out_of_Range` exception object. Lines 2 and 3 initialize the public data members of the exception object. Line 4 returns a pointer to the new exception object that can be raised as appropriate.

This example provides an interesting look at a general-purpose exception object that uses the polymorphic runtime type determination provided by the **Generic** class and the **class** macro. The exception type `Out_of_Range` could be used in many types of container classes (**Vector**<*Type*>, **List**<*Type*>, and so on) where a reference or index for some element is out of range. Any of these classes could raise this exception to display the error message and appropriate type-specific information without the need for a specialized exception type for each class.

RAISE

13.10 The **RAISE** macro allows an application program to create and raise an exception. **RAISE** uses **EXCEPTION** to construct the exception object and then calls its member function **raise**, defined as a friend function of the exception class, to raise the exception. This function searches for an exception handler of the appropriate type to handle the exception and, if found, invokes the exception handler function. It returns the exception object if the exception handler returns or if no exception handler is found. The exception object may be examined to determine if the exception was handled and if any alternate values were returned. There are many variations of **RAISE** that provide flexible and efficient means of customizing the exception object and raising the exception. In particular, the variable number of group name arguments should reduce the need for many different types of exception classes whose only difference is the type name.

NOTE: The **RAISE** macro takes some arguments that are actually pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` and `ERR_MSG` packages, or some application-specific package. See section 11, Symbols and Packages, for more information.

Name: **RAISE** — A COOL macro for constructing and raising an exception

Synopsis: **RAISE** (**Symbol*** *excp_type*, **REST:** *args*);

excp_type A symbol representing the **Exception** class type (that is, **Error**, **Warning**, and so forth)

args One or more of the following comma-separated arguments or values:

Symbol* *group_name*
One or more comma-separated pointers to **Symbol** objects representing aliases for this exception class type

const char* *format_string*
A character string compatible with the standard **printf** format containing the text of the error message

format_args
Any required argument(s) for the format string

key_value_args
The name(s) and value(s) of any public data members in the exception object

RAISE Example

13.11 In this example **RAISE** creates an **Error** exception object and raises the exception when the index for operator[] of a vector class is out of range.

```
1    inline int Vector::operator[] (int n) {
2      if (n >= 0 && n < this->number_elements
3                   return this->data[n];
4      else
5                   RAISE (Error, "vector::operator[](): %d out of range", n);
6    }
```

Lines 1 through 6 implement the code necessary for a typical operator[] member function of a class for vector of integers. However, when the index provided is out of range, the **RAISE** macro invocation in line 5 creates an exception object and raises the exception to report the error.

STOP

13.12 The **STOP** macro raises an exception and ends program execution with **exit** if the exception is not handled. By default in COOL, only exceptions of type **Error** will exit and exceptions of type **Fatal** will abort. **STOP** is similar to **RAISE** in that it uses **EXCEPTION** to construct the exception object and then calls its member function **raise** to raise the exception. This function searches for an exception handler of the appropriate type to handle the exception and, if found, invokes the exception handler function and returns the exception object. If no exception handler is found, however, program execution ends. There are many variations of **STOP** that provide flexible and efficient means of customizing the exception object and raising the exception. In particular, the variable number of group name arguments should reduce the need for many different types of exception classes whose only difference is the type name.

NOTE: The **STOP** macro takes some arguments that are actually pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` and `ERR_MSG` packages, or some application-specific package. See section 11, Symbols and Packages, for more information.

Name: **STOP** — Raise an exception and end the program if not handled

Synopsis: **STOP** (**Symbol*** *excp_type*, **REST:** *args);*

excp_type A symbol representing the **Exception** class type (that is, **Error**, **Warning**, and so forth)

args One or more of the following comma-separated arguments or values:

Symbol* *group_name*
One or more comma-separated pointers to **Symbol** objects representing aliases for this exception class type

const char* *format_string*
A character string compatible with the standard **printf** format containing the text of the error message

format_args
Any required argument(s) for the format string

key_value_args
The name(s) and value(s) of any public data members in the exception object

STOP Example

13.13 In this example, **STOP** creates an **Error** exception object and raises the exception when the index for operator[] of a vector class is out of range.

```
1    inline int vector::operator[] (int n) {
2      if (n >=0 && n < this->number_elements)
3                   return this->data[n];
4      else
5                   STOP (Error, "vector::operator[](): %d out of range", n);
6    }
```

Lines 1 through 6 implement the code necessary for a typical operator[] member function of a class for a vector of integers. However, when the index provided is out of range, the **STOP** macro invocation in line 5 creates an exception object and raises the exception to report the error. A handler for this exception could prompt the user for a new index and retry the operation. The distinction between the use of **STOP** and **RAISE** is that **STOP** guarantees to end the program if the exception is not handled, whereas **RAISE** will return.

VERIFY

13.14 The **VERIFY** macro asserts that an expression is **TRUE** by raising an exception of the appropriate type if it is **FALSE**. The exception type is optional, but if specified, is the group name or alias of the **VERIFY_ERROR** object created. This is because the macro assumes that a public data member named **test** is defined. If the exception type is not specified, no other arguments can be provided. **VERIFY** is similar to **RAISE** in that it uses **EXCEPTION** to construct the exception object and then calls the function **raise** to raise the exception. This function searches for an exception handler of the appropriate type to handle the exception and, if found, invokes the exception handler function and returns the exception object. If no exception handler is found, program execution ends.

NOTE: The **VERIFY** macro takes some arguments that are actually pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` and `ERR_MSG` packages, or some application-specific package. See section 11, Symbols and Packages, for more information.

Name: **VERIFY** — Verify that an expression evaluates to non-zero

Synopsis: **VERIFY** (*test_expression*, **REST:** *args*);

test_expression	Any valid C++ expression to be verified
args	One or more of the following comma-separated arguments or values:

Symbol* *group_name*
One or more comma-separated pointers to **Symbol** objects representing aliases for this exception class type

const char* *format_string*
A character string compatible with the standard **printf** format containing the text of the error message

format_args
Any required argument(s) for the format string

key_value_args
The name(s) and value(s) of any public data members in the exception object

VERIFY Example

13.15 This example is another variation of the previous two examples. **VERIFY_ERROR** asserts that the index specified for a vector element is within range. It creates a **Verify_Error** exception object and raises the exception when the index for operator[] of a vector class is out of range.

```
1   inline int vector::operator[] (int n) {
2    VERIFY ((n >= 0 && n < this->number_elements),
3                    Error, "vector::operator[](): %d out of range", n);
4    return this->data[n];
5   }
```

Lines 1 through 5 implement the code necessary for a typical operator[] member function of a class for a vector of integers. However, before the indexed element is looked up and returned, the **VERIFY** macro invocation on lines 2 and 3 insures that the given index is within range. When the index provided is out of range, **VERIFY** creates an exception object and raises the exception to report the error. A handler for this exception could prompt the user for a new index and retry the operation. If no exception handler is found, program execution ends.

Jump_Handler Class

13.16 The **Jump_Handler** class is derived from the **Excp_Handler** class. It saves the current environment and also the exception object when an exception is raised. Instances of this class are used by the **IGNORE_ERRORS** macro discussed below. An exception handler function saves a pointer to the exception raised in the **Jump_Handler** exception object and then calls the system function **longjmp**, passing the environment that was saved in the **Jump_Handler** object by **setjmp**.

Note: The *excp_type* arguments in the **Jump_Handler** class constructors and member functions are pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` package, or some application-specific package. See section 11, Symbols and Packages, for more information.

Name: **Jump_Handler** — An exception handler class for ignoring exceptions.

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

Base Class: **Excp_Handler**

Friend Classes: None

Constructors:

Jump_Handler (**Jump_Handler_Function** *fn*, **Symbol*** *excp_type*)
Creates an jump handler object associated with the exception type *excp_type*, initializes the jump handler function data member to *fn,* and pushes itself on top of the global exception handler stack. The jump handler function is of type **void** (*Jump_Handler_Function*)(*Exception*, Excp_Handler**).

Jump_Handler (**Jump_Handler_Function** *fn*, **int** *number*,
Symbol* *excp_type1*, **Symbol*** *excp_type2*, ...);
Creates an jump handler object, creates *number* group names *excp_type1*, *excp_type2*, and so on if necessary, associates this jump handler object with *number* group names *excp_type1*, *excp_type2*, and so on, initializes the jump handler function data member to *fn,* and pushes itself on top of the global exception handler stack. The jump handler function is of type **void** (*Jump_Handler_Function*)(*Exception*, Excp_Handler*).

Member Functions:

virtual Boolean invoke_handler (**Exception*** *excp*)
Returns **TRUE** if the exception handler function was invoked for *excp*; otherwise, this function returns **FALSE**.

Friend Functions:

void ignore_errors_handler (**Exception*** *excp*, **Excp_Handler*** *fn*);
This exception handler function ignores exceptions raised through the macros **RAISE** and **STOP**. When invoked, this function saves a pointer to the exception object *excp* in the jump handler *fn*. It then calls **longjmp,** passing the environment saved in **Jump_Handler**. The program returns to the point after the call of **setjmp** in the macro **IGNORE_ERRORS** discussed below.

IGNORE_ERRORS

13.17 The **IGNORE_ERRORS** macro ignores an exception raised while executing a body of statements. If an exception is raised while executing these statements, the **Jump_Handler** created by the surrounding **IGNORE_ERRORS** macro saves a pointer to the exception object. Program control returns to the statement following **IGNORE_ERRORS** macro. This macro eliminates the return value of the last statement within the body if no exception was raised. In addition, **IGNORE_ERRORS** works only for exceptions raised with the macros **RAISE** and **STOP**. The default exception type is **Error**, if no exception type or group name is specified.

NOTE: IGNORE_ERRORS uses the system functions **setjmp** and **longjmp**. If an exception occurs while executing statements within the body argument of the macro, causing program control to be redirected, objects falling out of scope will not have their destructor called. This is because the ANSI C **setjmp/longjmp** mechanism does not support a mechanism for unwinding the stack.

Name: **IGNORE_ERRORS** — Ignores a raised exception within a body of code

Synopsis: **IGNORE_ERRORS** (**Exception*** *excp*, **Symbol*** *excp_type* = **Error**,
REST: *args*) { *body}*

excp	Pointer that is set to the exception object if one is raised while executing the statements in *body*; otherwise, this pointer is set to **NULL**
excp_type	A symbol representing the **Exception** class type of *excp* (that is, **Error**, **Warning**, and so forth)
args	One or more of the following comma-separated arguments or values: *group_name* One or more comma-separated pointers to **Symbol** objects representing aliases for this exception class type
body	Any valid C++ statements to be executed under the protection of the **IGNORE_ERRORS** macro

IGNORE_ERRORS Example

13.18 In this example, **IGNORE_ERRORS** checks for an exception of type **Error** raised while summing up a vector of integers. In this simplistic example, the size of the vector is unknown. If during the loop, an exception is raised, an error message prints, and the function continues execution after the body of statements. If **IGNORE_ERRORS** were not used and an exception of type **Error** was raised, the program would end.

```
1     int sum_up (vector& v) {
2       int sum = 0;
3       Error* excp;
4       IGNORE_ERRORS (excp) {
5       for (int i = 0; i < NUM_ELEMENTS; i++)
6                     sum += data[n];
7       }
8       if (excp != NULL)
9                     cerr << excp;
10      return sum;
11    }
```

Lines 1 through 11 implement a function that calculates the sum of the element values of a vector of integers. Line 2 initializes a variable to hold the running total. Line 3 declares a pointer to an exception of type `Error`. Line 4 begins the **IGNORE_ERRORS** invocation. The pointer to the exception object is passed as an argument, along with the body of statements between the braces. At the end of the body, the variable `excp` is checked to see if it contains an address. If so, an exception must have been raised, so the exception object is output to the standard error stream. If its value is **NULL**, the loop ends successfully. Finally, line 10 returns the sum of the element values.

Exceptions as Symbols and Package

13.19 The exception handling facility uses the COOL symbolic computing capability. **Exception** (along with most other COOL classes) is derived from the **Generic** class, which eases run-time type checking and object query. The **invoke_handler** member function of the exception handler takes advantage of this feature. It calls **is_type_of** on the raised exception object to determine if it is of the desired exception type. The exception name specified in the exception macros and the exception handler constructor are pointers to **Symbol** objects. All classes inheriting from **Generic** are represented as type symbols in the COOL global symbol package, `SYM`.

When the exception macros are expanded in the program, the formatted error message constructed and stored in the exception object is also added as a symbol to the COOL global error message package, **ERR_MSG**. This package is created with the **text_package** macro which contains symbols whose values are the same as the symbol names. All error messages in a COOL application are implemented as text symbols, and a symbol definition file is automatically created that contains a summary of all the error messages. These error message symbols can be represented in other languages by establishing a property list with the appropriate translation. See Section 11, Symbols and Packages, for more information on the COOL symbolic computing capabilities.

User-Defined Exception Types

13.20 The COOL exception mechanism detects and raises an exception and finds the appropriate exception handler. To define a user-specific exception class, you must derive from the **Exception** class or one of the predefined exception types **Error**, **Fatal**, **System_Error**, **System_Signa**l, or **Verify_Error**. All new data members should be public. The **report** member function will need to be changed to reflect the nature of the newly created type of exception.

NOTE: Derived exception classes should have public data members. Initialize these data members with an assignment statement in the **EXCEPTION** macro invocation, and access the data members by exception handler functions.

To handle a specific type of exception, define an exception handler function that takes as its first argument a pointer to the exception object and returns void. An exception handler object is passed a pointer to this function through its constructor. The exception handler function can be defined with more than one argument, but a new exception handler class must be defined with a new version of the virtual **invoke_handler** member function. For example, the **Jump_Handler** class modifies the **invoke_handler** member function to call a function with two arguments: a pointer to the exception object and a pointer to the exception handler object.

Other user-derived exception classes can include data members for saving the wrong values detected by a program. These values report the problem to the exception handler and are often used when reporting the exception or error message to an output stream. Data members can also be included in an exception class so the signaler (the exception raiser) can indicate to an exception handler ways of proceeding from the exception.

For example, if an exception occurs because a variable has a wrong value, an exception object is first created and then raised. The exception object defined for this problem would have a data member with the wrong value and a data member for a new value. An exception handler resolves this problem by supplying a new value (usually by informing the user about the wrong value and querying the user for a new value). The handler stores this new value in the exception object and returns that object to the signaler. The signaler then assigns this new value to the variable.

COOL METHODOLOGY 14

Introduction

14.1 The C++ Object-Oriented Library (COOL) is a collection of classes, templates, and macros for use by C++ programmers writing complex applications. It raises the level of abstraction and allows the programmer to concentrate on the problem domain, not on implementing base data structures, macros, and classes. In addition to raising the level of abstraction, COOL also provides a system-independent software platform on top of which applications are built, since COOL encapsulates system-specific functionality such as date/time and exception handling. This section discusses the following topics:

- Preprocessor and macros
- Parameterized templates
- Symbols and packages
- Polymorphic management
- Exception handling
- Coding style and conventions
- Class hierarchy

COOL is an ever changing and growing C++ class library. As such, some constraints will be necessary in order to achieve compatible and seamless integration of new or modified features. This section outlines the major technologies and conventions that should be used and followed.

Requirements

14.2 This section discusses COOL methodology and should be used as an aid in understanding the COOL library, its organization, structure, and layout. It assumes you have a working knowledge of C++. For more detailed information and examples on each topic, you should refer to the appropriate section of this manual.

Preprocessor and Macros

14.3 The COOL macro facility is an extension to the standard ANSI C macro preprocessing functions available with the **#define** statement. The COOL preprocessor is a modified ANSI C preprocessor that allows a programmer to unobtrusively define powerful extensions to the C++ language.

This enhanced preprocessor is portable, compiler independent, and can execute arbitrary filter programs or macro expanders on C++ code fragments. Macros that support parameterized templates are implementations of theoretical design papers published by Bjarne Stroustrup. Other macros provide significant language features and enhanced power for the programmer previously unavailable with conventional C++ implementations. It is important to note, however, that once a macro is expanded, the resulting code is conventional C++ 2.0 syntax acceptable to any conforming C++ translator or compiler.

The COOL preprocessor is supplied as part of the library and is the implementation point for all language and computing enhancements available in COOL. The draft-proposed ANSI C standard indicates that extensions and changes to the language or features implemented in a preprocessor or compiler should be made by using the **#pragma** statement. The COOL preprocessor follows this recommendation and uses this for all macro extensions.

The COOL preprocessor is derived from and based upon the DECUS ANSI C preprocessor made available by the DEC User's group in the public domain and supplied on the X11R3 source tape from MIT. It complies with the draft ANSI C specification with the exception that trigraph sequences are not implemented. In addition to support for COOL macro processing discussed previously, the preprocessor has several new command line options to support C++ comments and includes file debugging aids.

The **#pragma defmacro** statement is implemented in the COOL C/C++ preprocessor and is the single hook through which features such as the class macro, parameterized templates, and polymorphic enhancements have been implemented. The **defmacro** facility provides a way to execute arbitrary filter programs on C++ code fragments passing through the preprocessor. When a **defmacro** style macro name is found, the name and contents up to the delimiter (including all matching {} [] () <> "" '' and comments found along the way) pipes onto the standard input stream of the indicated program or filter procedure. The preprocessor scans the procedure's standard output for further processing. The expansion replaces the macro call and is passed onto the compiler for parsing.

The implementation of a **defmacro** can be either external to the preprocessor (as in the case of files and programs) or internal to the preprocessor. For example, the **template, declare,** and **implement** macros that implement parameterized types are internal to the preprocessor, providing a more efficient implementation. The **defmacro** facility first searches for a file or program in the same search path used for include files. If a match is not found in the preprocessor table, an internal preprocessor table is searched. If a match is still not found, the error message is sent to the standard error stream: `"Error: Cannot open macro file [`*xxx*`]"`, where *xxx* is the name as it appears in the source code. The fundamental COOL macros are defined with **defmacro** in the header file <COOL/`misc.h`> that is included in all COOL C++ source files.

Porting COOL to a new platform or operating system starts with the preprocessor. The preprocessor contains support for the **defmacro** statement and also implements several important macros internally for efficiency and performance considerations. In addition, a powerful macro language that simplifies many library functions is available via the **MACRO** keyword (discussed in detail in Section 10). **MACRO** implements an enhanced **#define** syntax that supports multiple-line, arbitrary-length, nested macros, and preprocessor directives with positional, optional, optional keyword, required keyword, rest, and body arguments. Many of the COOL features would be very difficult, if not impossible, to implement without this enhanced macro language.

Parameterized Templates

14.4 The development and successful deployment of application libraries such as COOL is made easier and more useful by a language feature called parameterization. Parameterized templates allow a programmer to design and implement a class template without specifying the data type. The user customizes the template to produce a specific class by indicating the type in a program. Several versions of the same parameterized template (each 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 to support numerous variations of a type of class.

Regardless of the type of object a parameterized class is to manipulate, the structure and organization of the class 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 class. The user of the class then specifies the type of vector at compile time.

An important and useful type of parameterized template is known as a container class. A container class is a special kind of parameterized class where you put objects of a particular type. For example, the **Vector**<*Type*>, **List**<*Type*>, and **Hash_Table**<*KType,Vtype*> classes (discussed in Sections 6 and 7) 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 versions of very useful data structures. COOL supplies several common container class data structures that can be used in many typical application scenarios.

Each of the COOL parameterized container classes support 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 you to move through the collection of objects in some order and manipulate the element value at that position. This might be used, for example, in a function that takes a pointer to a generic object that is a type of container object. The function can iterate 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.

In addition to this built-in current position mechanism, COOL provides support for multiple iterators over the same class by using the **Iterator**<*Type*> class (discussed in detail in Section 5). For example, a programmer may need to write a function that moves through the elements of a container class and, at some point, needs to save the current position and begin processing elements at another location. After a period of time, the secondary processing ends, at which point flow of control returns to the previous stopping point. The current position is restored from the iterator object, and processing continues.

A programmer uses the COOL C++ Control program (**CCC**), instead of the normal **CC** procedure, to control the compilation process. This program provides all of the capabilities of the original **CC** program with additional support for the COOL preprocessor, parameterized types, and the 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 on to the system C++ compiler control program.

Symbols and Packages

14.5 A package provides a relatively isolated namespace for various COOL components called symbols. A symbol that is owned by a particular package is said to be *interned* in that package. In general, the term interned means that a particular object is uniquely identifiable in some context. When a symbol is interned, it becomes uniquely identifiable by the symbol name within a namespace context. The package system provides logical groupings of symbols supporting relationships established between named objects and the values they contain. Although the notion of symbols being grouped into packages is fairly straightforward, the nature of the relationships that can exist between packages and the way in which they establish a namespace can be quite complex. COOL provides several kinds of macros to simplify the usage and manipulation of symbols and packages.

A symbol is a data object that defines a relationship between a name, a package, a value, and a property list. The name is a character string used to identify the symbol. Once a name is established for a symbol, it may not be changed. The value field is used to refer to some C++ object. Property lists are lists of alternating names and values. The property list allows the programmer to associate supplemental attributes with a symbol. Initially, the property list for a symbol is empty.

The **Symbol** and **Package** classes implement the fundamental COOL symbolic computing support as standard C++ classes. The **Symbo**l class implements the notion of a symbol that has a name with an optional value and property list. Symbols are interned into a package, which is merely a mechanism for establishing separate namespaces. The **Package** class implements a package as a hash table of symbols and includes public member functions for adding, retrieving, updating, and removing symbols.

COOL supports efficient and flexible symbolic computing by providing symbolic constants and run time symbol objects. You can create symbolic constants at compile time and dynamically create and manipulate symbol objects in a package at run time by using any of several simple macros or by directly manipulating the objects. Symbols and packages in COOL manage error message textual descriptions with translations, provide polymorphic extensions to C++ for object type and contents queries, and support sophisticated symbolic computing normally unavailable in conventional languages.

Polymorphic Management

14.6 C++ version 2.0 as specified in the AT&T language reference manual implements virtual member functions that delay the binding of an object to a specific function implementation until run time. This delayed (or dynamic) binding is useful where the type of object might be one of several kinds, all derived from some common base class but requiring a specialized implementation of a function. The classic example is that of a graphics editor where, given a base class **graphic_object** from which **square**, **circle**, and **triangle** are derived, specialized virtual member functions to calculate the area are provided. In such a system, a programmer can write a function that takes a **graphic_object** argument and determine its area without knowing which of all the possible kinds of graphical objects the argument really is.

This dynamic binding capability of C++, while powerful and providing greater flexibility than most other conventional programming languages, is still not enough for some types of problems. Highly dynamic languages such as SmallTalk and Lisp allow the programmer to delay almost all decisions until run time. In addition, facilities are often present for querying an object at run time to determine its type or request a list of all available member functions. These kinds of features are commonly used in many symbolic computing and complex, knowledge-intensive operations management areas tackled today.

COOL supports enhanced polymorphic management capabilities with a programmer-selectable collection of macros, classes, symbolic constants, run time symbolic objects, and dynamic packages. This is facilitated by the **Generic** class that, combined with macros, symbols, and packages, provides efficient run-time object type checking, object query, and enhanced polymorphic management unavailable in the C++ language.

The **Generic** class is inherited by most other COOL classes and manipulates lists of symbols to manage type information. **Generic** adds run-time type checking and object queries, formatted print capabilities, and a describe mechanism to any derived class. The COOL **class** macro (discussed below) automatically generates the necessary implementation code for these member functions in the derived classes. A significant benefit of this common base class is the ability to declare heterogeneous container classes parameterized over the **Generic*** type. These classes, combined with the current position and parameterized iterator class, lets the programmer manipulate collections of objects of different types in a simple, efficient manner.

One of the simplest and most useful features facilitated by **Generic** is the runtime type checking capability. The **type_of** and **is_type_of** virtual member functions provide this kind of run-time type query for an object that is derived (at some point) from the COOL **Generic** class. Type determination and function dispatch can become quite tedious, however, if there are many types of objects. Ideally, each would be derived from a common base and include support for a virtual member function for each important operation that might be required. This is not always feasible, however, especially with a high number of objects obtained from several sources. An alternate scheme similar to the one mentioned above is the **type_case** macro, analogous to the C++ `switch` statement. It gathers all possible type cases and allows the user to symbolically dispatch on the type of object represented by the case statements. This automates some of the symbol collection and manipulation required with the earlier mechanism.

The **class** keyword is implemented as a COOL macro to add symbolic computing abilities to class definitions. It takes a standard C++ class definition and, if the class contains **Generic** somewhere in its inheritance hierarchy, it generates member functions for support of run time type checking and query. In addition, a symbol for the derived **Generic** class type is added to the COOL global symbol package `SYM`. The actual code which is expanded in a class definition and after a class definition is controlled by the **classmac** macro.

The **classmac** macro provides two hooks as a customization point by user-defined macros. A combination of data members and member functions of a class definition are passed as arguments to macros that can be changed or customized by the application programmer. The COOL **Generic** class uses the data member hook to implement the **map_over_slots** member function. There may be more than one **classmac** macro hook specified by the programmer. COOL has several, and other user-defined macros are simply chained together in a calling sequence ordered according to the order of definition. Each **classmac** macro defines how the **class** macro should expand the class definition. The **class** macro does not actually generate the code itself. This is defined in user-modifiable header files that specify a **classmac** macro. For example, a general-purpose mechanism that automatically creates accessor member functions to get and set each data member can be created by defining a **classmac** macro that is attached to the data member hook of the **class** macro. No changes to the COOL preprocessor are required.

The member functions added by **Generic** and the **class** macro to derived COOL classes manipulate symbols stored in the global `SYM` package. These symbols reflect the inheritance tree for a specific class. They may have optional property lists containing information associating supported member functions and their respective argument lists. User-defined classes derived from **Generic** are also automatically supported in an identical fashion, resulting in additional symbols in the global symbol package. As discussed earlier, these symbols must have storage allocated for them and code to initialize the package at program startup time. This is managed by the COOL file `symbols.C` which should be compiled and linked with every application that uses COOL. An automated method for ensuring correct package setup and symbol initialization is accomplished by establishing the correct dependency in an application make file.

Exceptions

14.7 In COOL, program anomalies are known as exceptions. An exception can be an error, but it can also be a problem such as impossible division or information overflow. Exceptions can impede the development of object-oriented libraries. Exception handling offers a solution by providing a mechanism to manage such anomalies and simplify program code. The COOL exception handling scheme is a raise, handle, and proceed mechanism similar to the Common Lisp Condition Handling system. When a program encounters an anomaly that is often (but not necessarily) an error, it can:

- Represent the anomaly in an object called an *exception*
- Announce the anomaly by *raising* the exception
- Provide solutions to the anomaly by defining and establishing *handlers*
- Proceed from the anomaly by invoking a *handler* function

The COOL exception handling facility provides an exception class (**Exception**), an exception handler class (**Excp_Handler**), a set of predefined exception subclasses (**Warning**, **Error**, **Fatal**, **System_Error**, **System_Signal**, and **Verify_Error**), and a set of predefined exception handler functions. In addition, the macros **EXCEPTION**, **RAISE**, **STOP**, and **VERIFY** allow the programmer to easily create and raise an exception at any point in a program.

When an exception is raised (through macros **RAISE** or **STOP**, for example), a search begins for an exception handler that handles this type of exception. An exception handler, if found, deals with the exception by calling its exception handler function. The exception handler function can correct the exception and continue execution, ignore the exception and resume execution, or end the program. In COOL, an exception handler for each of the predefined exception types exists on the global exception handler stack.

An exception handler invokes a specific exception handler function for a specific type of exception. Handling an exception means proceeding from the exception. An exception handler function could report the exception to standard error and end the program, or drop a core image for further debugging by the programmer. Another way of proceeding is to query the user for a fix, store the fix in the exception object, and return to where the exception was raised. When an exception handler object is declared, is is placed on the top of a global exception handler stack. When an exception is raised, a call searches for a handler. The handler search starts at the top of the exception handler stack.

There are six predefined exception type classes provided as part of COOL. The exception class is the base class from which specialized exception subclasses are derived. Derived from **Exception** are **Warning**, **System_Signal**, **Fatal** and **Error**. From the **Error** class, the **System_Error** and **Verify_ Error** classes are derived. The default exception handlers are called only if no other exception handler is established and available when an exception is raised. COOL offers users the option of defining their own exception types. Such types can be derived from the **Exception** class of one of the derived exception types. All user-defined exception classes should have public data slots. For more detailed information on creating your own exception types, refer to Section 13, Exception Handling.

The COOL exception handling facility provides several macros that simplify the process of creating, raising, and manipulating exceptions. These macros are implemented with the COOL macro facility discussed in Section 10, Macros. The **EXCEPTION** macro simplifies the process of creating an instance of a particular type of exception object. The **RAISE** macro allows the programmer to easily raise an exception and search for an exception handler. The **STOP** macro is similar to the **RAISE** macro, except that it guarantees to end the program if the exception is not handled. The **VERIFY** macro raises an exception if an assertion for some particular expression evaluates to **FALSE**. Finally, the **IGNORE_ERROR**S macro provides a mechanism to ignore an exception raised while executing a body of statements.

Coding Style and Conventions

14.8 A standard source code style allows several programmers to easily maintain and understand each other's code because additional semantic information can be inferred from the source code's format and style. In addition, a single style presents a more coherent, professional software package for potential source code users. This is particularly important for COOL, since parameterized templates require complete access to all source code. Finally, one of the foundations of object-oriented programming is code reuse. This is much easier if a programmer is able to browse through source code and understand its organization and layout. The COOL source code adopts the following C++ coding style convention:

- Variable and class naming conventions — A proposed definition for naming conventions for variables and classes and a coding style for writing C++ class definitions.

- Organization and contents of class header files — An ordering for all of the elements in a C++ class library. A uniform organization for C++ class definition elements will simplify a user's task in learning the interface of a class and in locating information when making later references to the class.

- Private/Protected/Public data members — Recommended usage for scoping data members in a class with respect to encapsulation, derivation, and an object-oriented data base (OODB).

- Source code documentation — Minimum standards and requirements consisting of at least an introductory, high-level algorithmic discussion and input/output documentation for each function.

- Source code indentation and layout — A flexible and easy to follow indentation and layout proposal, facilitated in part by a C++ mode distributed with COOL source code for the popular GNU Emacs editor.

- Error message text resource package — Use of the COOL exception handling mechanism provide a package containing all error messages in an application that eases internationalization of text message strings.

- Regression test suite — All modified and new C++ classes added to the COOL library should contain a complete, stand-alone test program that exercises all major features of the component and reports successes and failures via the test macros contained in the `~COOL/include/test.h` header file.

- Source code system independence — COOL places great importance upon system-independent code and features. As such, system-specific functions should be surrounded with preprocessor directives where appropriate.

- Build procedure — COOL contains a modified **imake** utility from the MIT X11R3 source tape that implements a system-independent build procedure. This should be used for all new classes and source code. It also provides configuration and rules files for localization or customization of system build utilities and commands to port to other operating systems and hardware platforms.

Naming Conventions

14.8.1 A prime objective for a naming convention is to allow programmers to recognize what sort of component a name refers to. Another goal is using meaningful names, which has not typically been done in C applications. The following naming conventions are used throughout the COOL source code. The reader is strongly encouraged to follow the same guidelines:

- Directory`, .C, and .h` filenames should be the same or close to the class being defined, and the declaration and implement files should be in a single directory. For example, the **String** class is defined and implemented in the files `String.h` and `String.C` and contained in the `~COOL/String` subdirectory.

- **Class**, **struct**, and **typedef** names should be capitalized with the words separated by underscores:

```
class Generic_Window { ... };
struct String_Layout { ... };
typedef int Boolean;
```

- All function names should be lowercase with each word separated by an underscore character:

```
void my_fun (int foo);
char* get_name (ostream&);
```

- Predicate functions should begin with `is_`:

```
Boolean is_type_of (int);
```

- Variable and data member names should be lowercase with words separated by underscores:

```
int ref_count;
char* name;
```

- Global and static variables should be appended with `_g` or `_s`, respectively:

```
int node_count_g;
static char* version_s;
```

- Preprocessor statements and **MACRO** names should be uppercase:

```
#define ABS ((x < 0) ? (-x) : x)
```

- Constants (**const**) declarations should be uppercase:

```
const int FALSE=0;
const int TRUE=!FALSE;
```

Class Header File Organization

14.8.2 All header files defining the structure of a class or parameterized template should be organized into sections in the following order:

- Included files and **typedef**s necessary for the class.
- Definition of private data members.
- Declaration of private member functions and friends.
- Definition of protected data members.
- Declaration of protected member functions and friends.
- Declaration of public member functions and friends.
- Inline member functions of the class follow the class definition.
- Other member and friend function definitions are located in a separate source code file.

In general, only the data member definitions and function prototypes of the member functions and friend functions should appear in the class construct. This separates the implementation from the specification and reduces clutter. Define inline functions after the class {...}; statements. In addition, the keyword **inline** should appear in both the class definition and in the actual implementation as a documentation aid. The optional **private** keyword should be explicitly stated. Finally, avoid multiple instances of scoped sections. There should be no more than one each of the private, protected, and public labels.

Private, Protected, and Public

14.8.3 In general, class data should be encapsulated in either the private or protected sections. Data specific to a particular class with no use for possible derived classes should be located in the private section. Data located in the protected section might include configuration or adjustment data members that a derived class might want to monitor or change. No COOL classes contain public data, and the user should not declare such data. Aside from being bad object-oriented programming style, classes with public data may be difficult to make persistent and stored in an OODB. The one exception to this standard are the derived exception classes, which may require public data members in order to allow query or update of alternate values.

Documentation

14.8.4 Documentation of all files is very important. Terseness should be the general rule for all header files, and completeness the rule for all code files. Parameterized templates have a single header/source file and all documentation should be located there. If in doubt, more documentation is better than less documentation. A high-level abstract at the top of each file should provide a description of the file's functionality. Class header files should also contain a brief description of the public interface.

Each function in a source code file should have a preceding block comment specifying the input and output parameters as well as giving a brief synopsis of the functionality. For complex inline definitions in header files, a block comment of this type should only be used when the purpose is not obvious because these comments do not appear in the code file. Since most inline functions contain trivial code (usually providing an accessor to some private data member), comment requirements for inline function can be relaxed.

All source code should be commented every few source lines. Specifically, large block comments every 100 lines is unacceptable. No comment should contain operating system specific names or terms unless that section of code is truly specific. When this is necessary, the code should be surrounded by conditional compilation constructs. These are handled by the preprocessor relative to that specific operating system.

Finally, documentation in the form of a man page should be written for every class. Layout and organization will be as that with the **–man** macro package available for **nroff(1)/troff(1)**. Section names and requirements for a class man page include Name, Synopsis, Base Class, Friend Classes, Description, Constructors (public or protected as necessary), Protected Member Functions (when appropriate), Public Member Functions, Files, See Also, and Bugs (when necessary). Introductory and high-level material should also be documented.

Source Code Indentation

14.8.5 Indentation and source code structure is relaxed, but it is suggested that the programmer use the C++ mode available for GNU Emacs and supplied with COOL. In general, statements should be restricted to one line with indentation reflecting block and scoping visibility. Location of such items as braces, spacing around parentheses, and so on is left up to the programmer. If the C++ mode is used, whole regions can be marked and indented appropriately, providing a simple means by which all source code can be brought into the same format.

Error Message Resource Package

14.8.6 All error message text strings in an application should use the **ERR_MSG** package available in COOL. The COOL exception handling scheme automatically uses this package ensuring that all text strings associated with error messages are stored as the value of a symbol (see Section 13). All error message symbols are automatically processed and located in one file, thus facilitating easy update or configuration. In particular, a language translation can be added to the property list of each symbol entry, providing an efficient and convenient means for internationalizing the text messages in an application.

Regression Test Suite

14.8.7 Each new or modified class contained in or added to COOL must also include a stand-alone test program. This should fully exercise all features and functions and report success or failure through the test macros contained in the `~COOL/include/test.h` header file. This test program is used in regression tests for new releases and ports to other software platforms to ensure a complete and working implementation.

Source Code System Independence

14.8.8 COOL places great importance upon system-independent code and features. As such, system-specific functions should be surrounded with `#if` preprocessor directives where appropriate. In general, small performance sacrifices in implementation are preferred if system independence and portability is improved.

Build Procedure

14.8.9 COOL contains a modified **imake** utility from the MIT X11R3 source tape that implements a system-independent build procedure. This should be used for all new classes and source code. **imake** provides configuration and rules files for localization or customization of system build utilities and commands to aid in porting activities to other operating systems and hardware platforms.

Class Hierarchy

14.9 The COOL class hierarchy implements a flat inheritance tree, as opposed to the nested SmallTalk model. Most COOL classes are derived from **Generic** to facilitate run time type checking and object query. Simple classes are not derived from **Generic** due to memory-space efficiency concerns. All parameterized container classes inherit from a base class that results in shared type-independent code. This reduces code replication when a particular type of container is parameterized several times for different objects in a single application. The COOL hierarchy is:

- **Pair**<*T1,T2*>
- **Range**
  - **Range**<*Type*>
- **Rational**
- **Complex**
- **Bignum**
- **Generic**
  - **String**
  - **Gen_String**
  - **Regexp**
  - **Vector**
    - **Vector**<*Type*>
      - **Association**<*T1,T2*>
  - **List_Node**
    - **List_Node**<*Type*>
  - **List**
    - **List**<*Type*>
  - **Date_Time**
  - **Timer**
  - **Bit_Set**
  - **Exception**
    - **Warning**

  - **Error**
    - **Verify_Error**
    - **System_Error**
  - **Fatal**
  - **System_Signal**
- **Excp_Handler**
  - **Jump_Handler**
- **Hash_Table**
  - **Set**
  - **Hash_Table**<*Key*,*Value*>
    - **Package**
- **Matrix**
  - **Matrix**<*Type*>
- **Queue**
  - **Queue**<*Type*>
- **Random**
- **Stack**
  - **Stack**<*Type*>
- **Symbol**
- **Binary_Node**
  - **Binary_Node**<*Type*>
- **Binary_Tree**
  - **Binary_Tree**<*Type*>
    - **AVL_Tree**<*Type*>
- **N_Node**<*Type*>
- **D_Node**<*Type*>
- **N_Tree**<*Type*,*Node*,*nchild*>

GLOSSARY

a

abstract data type	A set of values and a set of operations that can be applied to the new type.
accessor	1. A trivial inline function that gets or sets the value of a private or protected slot of a class. 2. A function designed to access (that is, read, write, or modify) the value(s) of a private or protected slot of a class.
array	A collection of objects of a single data type.
assignment statement	A programming language statement that gives a value to a variable.
association list	A data structure consisting of a list of pairs where each pair represents an association between its objects. The first element of the pair is the key and the second element is associated data. This is also referred to as an *alist*.

b

base class	The class from which subclasses are derived and inherit their properties.

c

class	The basic building blocks of an application, where each individual aspect of structure and behavior is defined separately.
constructors	Member functions with the same name as the class in which they are defined that provide for the automatic initialization of objects at their point of declaration.
container class	A class such as **Vector**, **List**, and **Hash_Table** that contains a set of application programmer defined data types. The COOL library contains a number of container classes.

d

DECLARE	A macro that expands to the declaration of a class.
#define	Preprocessor directive that defines a name and (optionally) the value that follows it.
derivation	The process by which one class is built on top of (specialized) from one or more base classes. For example, the **Bit_Set** class is derived from the **Generic** class, whereas the **Generic** class is the base class of **Bit_Set**.

derived class	A class that inherits from a base class.
destructors	Member functions providing for the automatic deallocation of the storage occupied by an object when the block containing the object is exited.
dymanic enumeration type	A modifiable data type used to define a set of named integral constants and to declare variables of that type.

e

encapsulation	A special type of form that surrounds another form and enhances the other form's operation without changing its basic functionality. A trace, for example, is an encapsulation.
enum	Keyword for declaring an enumeration.
enumeration	A set of symbolic integral constants.
environmental synonym	A variable contained in the operating system environment in which an application program runs and provides a specific value or customization directive to the program.
exception	Some noteworthy event that can occur during the execution of a program, such as an error or anomaly.
exception handling	A mechanism for managing program anomalies and errors.
extensibility	A language feature that allows programmers to create new types that can be endowed with specific properties and whose behavior is characterized in a class definition.

f

friend	A nonmember of a class that is given access to the nonpublic members of the class. Can be a nonmember function, a member function, or an entire other class.

g

Generic class	The class that is inherited by most COOL classes. It is used as a base class that adds run-time type checking and basic print capabilities to any derived class.

h

hash table	A table that derives a numeric index from some data key to index a specific value.
header file	A file containing information needed by several program modules. During compilation, the text of the header file becomes part of the program text that the compiler analyzes.
heterogeneous	The condition in which several objects are of different types.
homogeneous	The condition in which several objects are of the same type.

i

IMPLEMENT	A macro that expands into the function definitions of a class.
#include	A simple text manipulation mechanism for gathering source program fragments together into a single file for compilation.
inheritance	1. The ability of a class to use properties of another class. Enables the programmer to define an organization of classes that models the relationships among the various kinds of objects. 2. The capability for distinguishing between the generic properties of some class of object and the more specialized properties that only certain objects will share.
interned symbol	A symbol that belongs to a specific package.
iterator	A mechanism that automatically repeats the same series of steps until a predetermined stop is reached.

m

macro	A simple, symbolic programming-language statement that, when expanded, results in a series of more complex statements.
member function	The set of operations defined to manipulate an object within a class.

n

NULL	An empty or nonexistent or non-specified value.

o

object	A variable declared to be of a specific class. An object is not just passive data, but also the procedures which manipulate it. Objects are the modular building blocks for an object-oriented programming system.
object-oriented programming	A programming approach for designing and implementing software systems, centering around the concepts of abstract data types and classes, hierarchies, inheritance, and polymorphism. Noted primarily for its advantages of code reuse, extensibility, complexity control, and much closer linkage between software design and implementation.
operator	A symbol specifying an arithmetic, logical, or other manipulation of its operands.
ordered sequence class	A collection of basic data structures that implement sequential-access data structures as parameterized classes.
overloaded operator	An operator with an additional meaning assigned to it. When an operator is overloaded, its meaning is usually inferred from the types of their operands.

p

package	A collection of symbols that serves as a namespace. See also package system.
package system	A facility that establishes a mapping from names to symbols and helps prevent namespace conflicts. The package system allows different programs to use the same name for objects so that the programs and objects can coexist in the same environment.
parameterized class	A class in which one or more types can be declared at compile time.
pointer	A data type that holds the address of an object in memory.
polymorphic	The ability of different objects to respond differently to the same message at run time.
private	Information that cannot be manipulated by the programmer.
procedures	The operations or behaviors that the object can perform.
property list	A component of a symbol that effectively provides the symbol with many modifiable, named components. The property list has zero or more entries, with each entry consisting of a pair of elements. The first element of the pair is called the indicator and is used to name a particular property. Each indicator must be unique within that property list. The second element can be any object that represents the value of that property. Functions are available tomanipulate a symbol's property list.
protected	Information that can be manipulated in a limited manner by the programmer.
public	Information that can be manipulated by the programmer.

r

raise	To announce an exception.

s

scope	The spatial or textual region of a program or form within which it is possible to refer to an object.
standard error	A predefined I/O stream used to alert the user to some exceptional condition in the program during execution.
symbol	An object used as an identifier to specify a relationship between a name and other objects and variables. Internally, each symbol is represented as a structure with the following components: │ Value │ Property list These contain information about the symbol, and functions are provided to manipulate this information, as well as the symbol itself.

t

template	Provides a means of defining a complex macro that supports parameterized classes.
typedef	A storage class that is used to create new data types from existing data types.

u

unordered sequence class	A collection of basic data structures that implement random-access data structures as parameterized classes.

v

value	A constant or quantity assigned to a variable.
void	A data type that is used when declaring a function to indicate that the function does not return a value or that the function does not take any arguments.

INDEX