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*>

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.

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 **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.

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.

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.

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

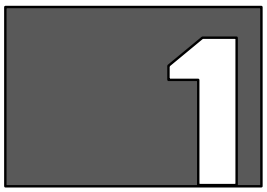
1.5 Parameterized classes allow a programmer to design and implement a class template without specifying the data type. The user can then customize the class by specifying the type when it is used in a program. Parameterized classes can be thought of as *metaclasses* in that only one source base needs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

OVERVIEW OF COOL



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.

Printed on: Wed Apr 18 06:58:45 1990

Last saved on: Tue Apr 17 13:22:25 1990

Document: s1

For: skc