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.

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`.

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.

```
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.

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:

```
1   /*
2    * DEFPACKAGE MY_SYM        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_SYM <MY_SYM> name=my_sym.p,
10   *  count=MY_SYM_count, case=sensitive,
11   *  start=0, increment=1, template=0, max=0
12   */

13  /* WARNING: Do not remove this line */
14  #define MY_SYM_count 6
```

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.

```
1   #include <COOL/Date_Time.h>            // Include COOL Date/Time header
2   #include <COOL/Package.h>              // Include COOL Package header

3   symbol_package (MY_SYM, "my_sym.p");              // Create symbol package

4   DEF_MY_SYM (sym1, String, new String("Greetings!"));
5   DEF_MY_SYM_PROPERTY (sym1, MY_SYM (Prop. example), Symbol, MY_SYM (String));

6   int main (void) {
7     Symbol *s1 = MY_SYM (sym1);                     // Lookup first symbol
8     cout << "First symbol is " << s1->name() << "\n"; // Output symbol name
9     cout << "Also available via MY_SYM(sym1): " << MY_SYM(sym1)->name()<<"\n";
10    cout << s1 << "\n";                             // Output value/property list
11    s1->set (new String ("Goodbye!"));              // Add new value
12    cout << "sym1 value is now " << s1->value () << "\n"; // Output value
13    Date_Time d1 (US_CENTRAL, UNITED_STATES);       // Create date/time object
14    d1.set_local_time ();                           // Set to current date/time
15    Symbol* s2 = MY_SYM (sym2);                     // Create new symbol object
16    cout << "Second symbol is " << s2->name() << "\n";// Output symbol name
17    cout << "Also available via MY_SYM(sym2): " << MY_SYM(sym2)->name()<<"\n";
18    s2->put (MY_SYM (Creation Time), &d1);          // Add property
19    cout << s2 << "\n";                             // Output runtime symbol
20    return 0;                                       // Return valid code
21  }
```

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.

```
1   #include <cool/Package.h>              // Include COOL Package header
2   text_package (MY_TEXT, "my_text2.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  }
```

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.

```
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`.

```
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:

```
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  */
```

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.

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.

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.

```
1  #include <COOL/Package.h>              //Include COOL Package header
2  enumeration_package (MY_ENUM, "my_enum.p"); //Create enum package

3  int main (void) {
4   cout << "MY_ENUM (Red) has a value of " << MY_ENUM (Red) << "\n";
5   cout << "MY_ENUM (Yellow) has a value of " << MY_ENUM (Yellow) << "\n";
6   cout << "MY_ENUM (Green) has a value of " << MY_ENUM (Green) << "\n";
7   return 0;                              //Return valid success code
8  }
```

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.

- **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.

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:

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.

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.

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.

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.

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:

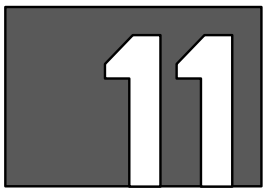
inline Symbol ();
Applications should use the **Package::intern** member function to create symbols. The public constructor is provided for use by COOL macros to create and initialize constant symbols used for run time type query.

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.

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.

Printed on: Wed Apr 18 07:12:35 1990

Last saved on: Tue Apr 17 13:33:25 1990

Document: s11

For: skc

pl2ps 3.2.1 Copyright 1987 Interleaf, Inc.