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`.

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.

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.

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

```
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.

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:

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.

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**.

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.

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.

Printed on: Wed Apr 18 07:13:53 1990

Last saved on: Tue Apr 17 13:35:10 1990

Document: s12

For: skc