IGNORE MACRO

10.12 The **IGNORE** macro silences warnings from the compiler relating to unused variables or function arguments. An application often has no control over the interface to a function and does not require all of the arguments. In other situations, an object might be created so that a friend function can access some private static data member. Without this macro, warnings of the type "`Warning: variable` <*foo*> `declared but not used`" appear. The **IGNORE** macro suppresses these warning messages.

Name: **IGNORE** — Silences compiler warnings from unused variables

Synopsis: **IGNORE** (*name*)

name The name of the argument/variable not used

Example: The following example shows the **main** function of a program with its two standard arguments. However, in this example, these arguments are unused. By using the **IGNORE** macro, the warning error messages are never generated by the compiler.

```
main (int argc, char** argv) {
  IGNORE (argc);                    // Don't use argument
  IGNORE (argv);                    // Don't use argument
  ....
}
```

Name: **EXPAND_ARGS** — Expand macro arguments before invocation

Synopsis: **EXPAND_ARGS** (*name*, **REST:** *args*)

name — Name of the macro to be invoked

args — Arguments to be expanded and then passed to the macro

Example: The `<stdarg.h>` header file provides a set of preprocessor macros to allow the C++ compiler to accept a variable number of arguments in a function call. The syntax of one of these macros is `va_arg (`*argp*, *type*`)`, where *type* is the type of the arguments expected. In the case of such things as COOL parameterized classes, however, a type like **Pair**<*Generic**, *Symbol**> is not recognized as a valid type by `va_arg` because it too is a macro that must be expanded first. The solution is to pass the name of the macro and its arguments to the **EXPAND_ARGS** macro, as shown below in line 2, which results in the *type* argument being expanded before being passed on, instead of the standard call as in line 1.

```
1    va_arg (argp, type)
2    EXPAND_ARGS (va_arg, argp, type)
```

INITIALIZE

10.11 The **INITIALIZE** macro guarantees to execute a body of code before the main program is called. This is often necessary in an application when a table or state information needs to be initialized before constructors can be called. **INITIALIZE** works by creating a static function containing the body of code to be executed. It initializes a global static variable, `For_Initialization_Only`, with a pointer to this function. `For_Initialization_Only` is a class whose constructor executes the function. The C++ language guarantees to execute the constructors for all global and static class instances before the main program is run. However, there is no mechanism by which the user can control the ordering of global static constructors themselves.

Name: **INITIALIZE** — A **MACRO** whose body is executed once

Synopsis: **INITIALIZE** (*name*) { *body* }

name — Name of the initialization sequence

body — Statements substituted when the macro is expanded

Example: In the following example, a global instance of a hash table is created on line 1 where both the key and the value are character strings. Lines 2 through 6 contain the **INITIALIZE** macro invocation to initialize this hash table by invoking the **put** member function of the **Hash_Table** class.

```
1    Hash_Table<char*, char*> capitals_g;
2    INITIALIZE (capitals_g) {
3      capitals_g.put ("Texas", "Austin");
4      capitals_g.put ("Arkansas", "Little Rock");
5      capitals_g.put ("Michigan", "Lansing");
6    }
```

ONCE_ONLY

10.9 The **ONCE_ONLY** macro allows an application to control the expansion or insertion of a section of code or function. **ONCE_ONLY** creates a symbol in a package whose value is the file name where the symbol was first encountered. If the current value of the symbol is the same as the current file (available from the standard preprocessor symbol `__FILE__`), the code is expanded and compiled. If not, nothing happens. **ONCE_ONLY** uses symbol and package objects and is more completely discussed in Section 11, Symbols and Packages.

Name: **ONCE_ONLY** — A macro whose body is expanded only once

Synopsis: **ONCE_ONLY** (*name*) {*body*}

name Symbolic name given to this operation

body Statements substituted when the macro is expanded

Example:

The C++ parameterized type macros generate two sets of code: a declaration that must always be included and implemented code that only needs to be compiled once. This is particularly important when the definition of the parameterized type is in a header file. By using the **ONCE_ONLY** macro, all macros and expansion of code are controlled and located in a single header file. The code implementing the parameterized type is expanded by the first application source file that included the header file.

The **DECLARE** macro used to declare a specific type of parameterized class only declares the class type and inline member functions. This could be changed to also implement the member functions by invoking the **IMPLEMENT** macro, if this is only done once during compilation. The macro `AUTO_DECLARE` declared below would implement the member functions one time only.

```
1    MACRO AUTO_DECLARE (name, REST: parms) {
2      DECLARE name<parms>;
3      ONCE_ONLY (Implement_##name<parms>) {
4        IMPLEMENT name<parms>;
5      }
6    }
```

Line 1 declares the macro `AUTO_DECLARE` with two arguments. The first argument specifies the parameterized class name and the second specifies any necessary arguments, including the type. Line 2 declares the parameterized class of the specified type. Lines 3-5 utilize **ONCE_ONLY** to implement the parameterized class if it has never been implemented before. This mechanism is not the default mechanism used in COOL because it prevents the fracturing of the source code template to reduce program size. This feature is available with CCC and is discussed in section 5, Parameterized Types.

EXPAND_ARGS

10.10 The **EXPAND_ARGS** macro is useful when one or more of the arguments to some **MACRO** are themselves macros that must be expanded first. This feature is also available via the **expanding** option in the **MACRO** syntax discussed earlier. The major difference between the two is that **EXPAND_ARGS** allows this function to be added to existing macros that may not have this already in place.

KEYARGS

10.8 The **KEYARGS** macro implements a keyword argument feature for standard C++ functions similar to the **KEY:** modifier available with **MACRO** which supports optional keyword arguments.

Name: **KEYARGS** — Provides keyword arguments for C++ functions

Synopsis: **KEYARGS** *type name* (*arglist*)

type Function return type

name Name of the function

arglist A C++ function argument list that supports keyword arguments:

[**KEY:**] *identifier* [= *default*] [, *arglist*]
All ensuing arguments are taken to be keyword arguments that allow the user to specify a particular value. Default values are supported by an equal sign and value, and can be applied to both regular and keyword arguments.

Example: This example defines the function `set` that returns a Boolean value. The first argument (`size`) is a required positional argument, while the second and third (`low` and `high`) are optional keyword arguments. A skeleton implementation of this function is shown in lines 1 through 3 below:

```
1    KEYARGS Boolean set (int size, KEY: int low=0, int high=100) {
2      ...
3    }
```

Lines 4 through 6 show a call to `set` with a value of `512` for the first argument and a value `1024` for the key argument `high`. The value of the keyword argument `low` will default to value 0. Lines 7 through 9 show the results of this macro expansion:

```
4    if (set (512, high=1024) == TRUE) {
5      ...
6    }

7    if (set (512, 0, 1024) == TRUE) {
8      ...
9    }
```

A specific example for the **Vector<*Type*>** class is shown below. Lines 1 and 2 show the macro call and lines 3 through 8 show the resulting macro expansion:

```
1   Vector<int> v1;
2   LOOP (int, e, v1) { cout << e << ", "; }

3   {  int e;
4      for (v1.reset(); v1.next(); ) {
5          e = v1.value();
6          cout << e << ", ";
7      }
8   }
```

This example contains an instance of `Vector<int>` called `v1`. The `LOOP` macro iterates through the vector and assigns each element to a temporary variable `e`. This is then used in the expanded `body` argument. The net result is to print all elements in the vector separated by commas.

ISSAME

10.7 The **ISSAME** macro is used in the preprocessor to compare two strings to see if they are the same. This macro is intended to be used in a similar manner as the preprocessor **#if** directive, which allows a symbol to be compared to some integer value. If the character strings are the same, **ISSAME** returns one; otherwise, it returns zero.

Name: **ISSAME** — Compares two character strings at compile time

Synopsis: **ISSAME** (*arg1*, *arg2*)

arg1 The first character string

arg2 The second character string

Example: This macro is used in the COOL **Hash_Table**<*T1*,*T2*> class to select the hash function based on the key type. If the hash table is parameterized such that the key type is **char***, a specific hashing function suited for character strings is implemented as the default hashing scheme. If not, an alternate hashing function is used. In the example below, line 1 compares the key type to several string type names. If a match is indicated, the statements at line 2 will be used. If no match is indicated, the statements at line 4 will be used.

```
1   #if ISSAME (T1, char*, String, Gen_String)
2   ...
3   #else
4   ....
5   #endif
```

The macro `build_table` is defined on lines 1 through 3 and takes two arguments: a name to associate with the table and a **REST:** argument called `rest` that refers to all remaining arguments. A **char*** variable called `name` is defined on line 2 and contains an embedded call to a second macro with the `rest` argument mentioned above. Note also that the embedded call is within the initialization braces of the character string variable and is followed by a **NULL** symbol.

The second macro defined in lines 4 through 9 loops through the `rest` argument values and recursively calls itself. Line 4 contains the prototype with two arguments. The first argument `first` is stripped from the incoming argument list and the remaining `count` arguments are left alone in the `rest` argument. Line 5 uses the ANSI # character on an argument to double quote the value. Then, a conditional clause tests `count` to see if there are remaining arguments and, if so, recursively calls the macro. When there are no more arguments, the `build_table` macro regains control and appends the **NULL** and closing brace to the result of the second macro.

A sample use of this macro is shown below to illustrate the construction of a **NULL**-terminated table containing character strings. Line 1 shows the macro call and line 2 shows the resulting macro expansion:

```
1       build_table (table, 1,2,3,4,5,6,7);
2       char* table[] = {"1", "2", "3", "4", "5", "6", "7", NULL};
```

Example 3:

As a final example, here is a macro that uses the **BODY:** modifier. It takes advantage of the current position feature found in the COOL container classes to implement a general purpose **LOOP** macro similar to that found in Common Lisp. Since all COOL container classes implement the current position iterator capability, this macro will work equally well with **List**, **Vector**, **Set**, and so on:

```
1       MACRO LOOP (type, variable, container, BODY: body) {
2         { type variable;
3             for (container.reset(); container.next(); ) {
4                 variable = container.value();
5                 body
6             }
7         }
8       }
```

Line 1 contains the prototype of the macro `LOOP` that takes four arguments: a container class element type; a variable name (of the type) to be declared; the name of a container class instance; and a **BODY:** argument of code to be applied to each element. Line 2 declares an instance of the element type in the specified container class. Lines 3 through 6 implement a loop that iterates through the elements of the container. Line 4 assigns the value of the element at the current position to the local variable declared on line 2. Line 5 expands the body argument specified.

MACRO Examples

10.6 Following are three examples of **MACRO**, each using various features and concepts to highlight some of the COOL macro capabilities. More detailed and complex examples follow in subsequent sections. It cannot be emphasized enough how important the macro facility is to the implementation of COOL. Without it, many features and functions would not be possible or would be more cumbersome and difficult to use. As an example of this type of use, the aggressive reader is referred to the end of Section 11, Symbols and Packages, for a detailed examination of the **symbol_package** macro.

Example 1:

This is a simple use of **MACRO** to implement a wrapper to an initialization routine that provides greater flexibility in passing arguments than is possible with straight C++ 2.0 syntax.

```
1       MACRO set_val (size, value=0, KEY: low = 0, high) {
2         init (size, value, low, high-low)}
```

Line 1 contains the function prototype for the macro `set_val` defined between the following braces. This macro takes four arguments:

- `size` is a required positional argument;
- `value` is an optional positional argument that if not specified in a particular call has a default value of `0`;
- `low` is an optional keyword argument with a default value of zero;
- `high` is a required keyword argument.

Line 2 contains the body of the macro which in this case involves a call to the `init()` function. The following shows several legal invocations of the macro, along with the resulting macro expansions:

```
set_val (0, high=20)            -> init (0, 0, 0, 20-0);
set_val (0, low=5, high=15)     -> init (0, 0, 5, 15-5);
set_val (1, 2, high=25)         -> init (1, 2, 0, 25-0);
```

Example 2:

The next example makes use of the **REST:** argument list modifier and recursive calls of the macro defined. Note that there are two macros, the first calls the second to do most of the work. The results of both are combined and placed on the standard output of the preprocessor:

```
MACRO build_table (name, REST: rest) {
  char* name[] = { build_table_internal (rest) NULL}
}

MACRO build_table_internal (first, REST: rest=count) {
  #first,
  #if count
  build_table_internal (rest)
  #endif
}
```

MACRO

10.5 **MACRO** provides a powerful and flexible macro language used to simplify many of the features and functions contained in the library. The **defmacro** feature previously discussed is used to declare the **MACRO** keyword whose implementation is a preprocessor-internal routine named *macro*. The terminating delimiter for a **MACRO** is the closing brace character. **MACRO** implements an enhanced **#define** syntax that supports multiple line, arbitrary length, nested macros, and preprocessor directives with positional, optional, optional keyword, required keyword, rest, and body arguments.

Name: **MACRO** — Enhanced COOL macro language

Synopsis: **MACRO** *name* [*expanding*] (*arglist*) { *body* }

name The name of the macro

expanding Optional argument that, when present, indicates that argument names themselves should be macro-expanded before passing onto and invoking the *name* macro.

arglist A list of comma separated arguments

KEY: *identifier* [= *value*]
All ensuing arguments are taken to be keyword arguments that allow the user to specify a particular value. Default values are supported by an equal sign and value, and can be applied to both regular and keyword arguments.

REST: *name* [= *count*]
Indicates that there are some number of arguments, all of which are referenced by the one named identifier. An optional equal sign and identifier contains the number of arguments remaining. This is typically used when an outer level macro must pass some number of arguments to an inner level macro.

BODY: *body*
Indicates that *body* is to be expanded to include all text within the braces after the macro call. This is useful for identifying a section of code that implements some part of the macro or should be passed to other nested macros.

body Statements substituted when the macro is expanded. These statements can be any valid C++ statements terminated with a semicolon and surrounded by curly-braces.

defmacro

10.4 The **#pragma defmacro** statement is implemented in the COOL preprocessor and is the single hook through which features such as the class macro, parameterized templates, and polymorphic enhancements have been implemented. The **defmacro** facility provides a way to execute arbitrary-filter programs on C++ code fragments passing through the preprocessor. When a **defmacro** style macro name is found, the name and everything until the delimiter (including all matching { } [] () <> “” ‘’ and comments found along the way) is piped onto the standard input stream of the indicated program or filter procedure. The procedure’s standard output is scanned by the preprocessor for further processing. The expansion replaces the macro call and is passed onto the compiler for parsing.

The implementation of a **defmacro** can be either external to the preprocessor (as in the case of files and programs) or internal to the preprocessor. For example, the **template, declare,** and **implement** macros that implement parameterized types is internal to the preprocessor to provide a more efficient implementation. The **defmacro** facility first searches for a file or program in the same search path as that used for include files. If a match is not found, an internal preprocessor table is searched. If a match is still not found, the error message "`Error: Cannot open macro file [`*xxx*`]`" is sent to the standard error stream where *xxx* is the name as it appears in the source code. The fundamental COOL macros are defined with **defmacro** in the header file <COOL/misc.h>, which is included by all COOL C++ source files.

Name: **defmacro** — The COOL C/C++ preprocessor extension mechanism

Synopsis:

#pragma defmacro *name* <*file*> *options*
#pragma defmacro *name* “file” *options*
#pragma defmacro *name program options*

name — A character string identifying the macro

file — The name of a file implementing the macro

program — The name of a filter program implementing the macro

options — One or more of the following space-separate parameters:

recursive
When present, the macro may be recursively expanded.

expanding
When present, input to the macro is macro-expanded.

delimiter=*c*
The default delimiter ‘;’ is replaced with *c*.

condition=*c*
When present, the macro will not be invoked unless followed by *c*.

REST: *args*
Other arguments are passed to the macro expander.

The COOL preprocessor is derived from and based upon the DECUS ANSI C preprocessor made available by the DEC User's group in the public domain and supplied on the X11R3 source tape from MIT. It complies with the draft ANSI C specification with the exception that trigraph sequences are not implemented. In addition to support for COOL macro processing discussed above, the preprocessor has several new command line options to support C++ comments. These command line options also have include-file debugging aids.

Name: **ccpp** — The COOL C/C++ preprocessor

Synopsis: **ccpp** [–*options*] [infile [outfile]]

Options:

–B
Recognizes the C++ double slash (//) comment character and treats all characters following up to the next newline character as commentary text.

–C
If set, source-file comments are written to the output file. This allows **ccpp** output to be used as input to a program such as **lint(1)** that expects comments to be specially formatted.

–D*name***[=value]**
Defines *name* as if the programmer had defined it in the program. If no value is provided, a default value of 1 is used.

–E
Always returns a successful status completion code to the operating system, even if errors were detected.

–I*directory*
Adds the specified directory to the list of directories searched when looking for an include file. Note that there is no space between the option letter and the directory name.

–U*name*
Undefines *name* as if the programmer had undefined it in the program.

–X[*number***]**
Enables debugging output from the preprocessor. A value of 1 for *number* will cause the pathname of each included file to be sent to the standard error stream. A value of 2 for *number* will cause **#control** statements to be inserted as comments in the output. A value of 3 for *number* will enable both debugging modes. If no value for *number* is provided, a default value of 1 is used. Note that this option is designed to be a debugging aid for use when the preprocessor is run as stand alone and not when invoked by the control program. Other values for *number* are ignored.

MACROS 10

Introduction

10.1 The COOL macro facility is an extension to the standard ANSI C macro preprocessing functions available with the **#define** statement. The COOL preprocessor is a modified ANSI C preprocessor that allows a programmer to define powerful extensions to the C++ language in an unobtrusive manner. This enhanced preprocessor is portable and compiler-independent, and can execute arbitrary-filter programs or macro expanders on C++ code fragments. It is important to note, however, that once a macro is expanded, the resulting code is conventional C++ 2.0 syntax acceptable to any conforming C++ translator or compiler.

The COOL macro facilities have many components. Macros such as those that support parameterized templates are implementations of theoretical design papers published by Bjarne Stroustrup. Others provide significant language features and enhanced power for the programmer heretofore unavailable with conventional C++ implementations.

This section provides information on the COOL macro facility that forms the basis for many of the advanced features covered in later sections. The following topics are discussed in this section:

- COOL preprocessor
- **defmacro**
- **MACRO**
- Example COOL macros

Requirements

10.2 This section discusses the macro facilities of COOL. It assumes that you have a working knowledge of the C++ language and are familiar with the concept of macros and macro expansion as found in the standard C preprocessor.

COOL Preprocessor

10.3 The COOL preprocessor is supplied as part of the library and is the point at which all language and computing enhancements available in COOL are implemented. The proposed draft ANSI C standard indicates that extensions and changes to the language or features implemented in a preprocessor or compiler should be made by using the **#pragma** statement. The COOL preprocessor follows this recommendation and uses this to make all macro extensions. The **#pragma defmacro** statement is the single hook through which features such as the class macro, parameterized templates, and polymorphic enhancements have been implemented.

Porting COOL to a new platform or operating system starts with the preprocessor. The preprocessor contains support for the **defmacro** statement and also implements several important macros internally for efficiency and performance considerations. These include **template**, **class**, **DEFPACKAGE**, and **DEFPACKAGE_SYMBOL**.

Printed on: Wed Apr 18 07:11:36 1990

Last saved on: Tue Apr 17 13:30:12 1990

Document: s10

For: skc