User-Defined Exception Types

13.20 The COOL exception mechanism detects and raises an exception and finds the appropriate exception handler. To define a user-specific exception class, you must derive from the **Exception** class or one of the predefined exception types **Error**, **Fatal**, **System_Error**, **System_Signa**l, or **Verify_Error**. All new data members should be public. The **report** member function will need to be changed to reflect the nature of the newly created type of exception.

NOTE: Derived exception classes should have public data members. Initialize these data members with an assignment statement in the **EXCEPTION** macro invocation, and access the data members by exception handler functions.

To handle a specific type of exception, define an exception handler function that takes as its first argument a pointer to the exception object and returns void. An exception handler object is passed a pointer to this function through its constructor. The exception handler function can be defined with more than one argument, but a new exception handler class must be defined with a new version of the virtual **invoke_handler** member function. For example, the **Jump_Handler** class modifies the **invoke_handler** member function to call a function with two arguments: a pointer to the exception object and a pointer to the exception handler object.

Other user-derived exception classes can include data members for saving the wrong values detected by a program. These values report the problem to the exception handler and are often used when reporting the exception or error message to an output stream. Data members can also be included in an exception class so the signaler (the exception raiser) can indicate to an exception handler ways of proceeding from the exception.

For example, if an exception occurs because a variable has a wrong value, an exception object is first created and then raised. The exception object defined for this problem would have a data member with the wrong value and a data member for a new value. An exception handler resolves this problem by supplying a new value (usually by informing the user about the wrong value and querying the user for a new value). The handler stores this new value in the exception object and returns that object to the signaler. The signaler then assigns this new value to the variable.

IGNORE_ERRORS Example

13.18 In this example, **IGNORE_ERRORS** checks for an exception of type **Error** raised while summing up a vector of integers. In this simplistic example, the size of the vector is unknown. If during the loop, an exception is raised, an error message prints, and the function continues execution after the body of statements. If **IGNORE_ERRORS** were not used and an exception of type **Error** was raised, the program would end.

```
1   int sum_up (vector& v) {
2     int sum = 0;
3     Error* excp;
4     IGNORE_ERRORS (excp) {
5     for (int i = 0; i < NUM_ELEMENTS; i++)
6                   sum += data[n];
7     }
8     if (excp != NULL)
9                   cerr << excp;
10    return sum;
11  }
```

Lines 1 through 11 implement a function that calculates the sum of the element values of a vector of integers. Line 2 initializes a variable to hold the running total. Line 3 declares a pointer to an exception of type `Error`. Line 4 begins the **IGNORE_ERRORS** invocation. The pointer to the exception object is passed as an argument, along with the body of statements between the braces. At the end of the body, the variable `excp` is checked to see if it contains an address. If so, an exception must have been raised, so the exception object is output to the standard error stream. If its value is **NULL**, the loop ends successfully. Finally, line 10 returns the sum of the element values.

Exceptions as Symbols and Package

13.19 The exception handling facility uses the COOL symbolic computing capability. **Exception** (along with most other COOL classes) is derived from the **Generic** class, which eases run-time type checking and object query. The **invoke_handler** member function of the exception handler takes advantage of this feature. It calls **is_type_of** on the raised exception object to determine if it is of the desired exception type. The exception name specified in the exception macros and the exception handler constructor are pointers to **Symbol** objects. All classes inheriting from **Generic** are represented as type symbols in the COOL global symbol package, `SYM`.

When the exception macros are expanded in the program, the formatted error message constructed and stored in the exception object is also added as a symbol to the COOL global error message package, **ERR_MSG**. This package is created with the **text_package** macro which contains symbols whose values are the same as the symbol names. All error messages in a COOL application are implemented as text symbols, and a symbol definition file is automatically created that contains a summary of all the error messages. These error message symbols can be represented in other languages by establishing a property list with the appropriate translation. See Section 11, Symbols and Packages, for more information on the COOL symbolic computing capabilities.

IGNORE_ERRORS

13.17 The **IGNORE_ERRORS** macro ignores an exception raised while executing a body of statements. If an exception is raised while executing these statements, the **Jump_Handler** created by the surrounding **IGNORE_ERRORS** macro saves a pointer to the exception object. Program control returns to the statement following **IGNORE_ERRORS** macro. This macro eliminates the return value of the last statement within the body if no exception was raised. In addition, **IGNORE_ERRORS** works only for exceptions raised with the macros **RAISE** and **STOP**. The default exception type is **Error**, if no exception type or group name is specified.

NOTE: IGNORE_ERRORS uses the system functions **setjmp** and **longjmp**. If an exception occurs while executing statements within the body argument of the macro, causing program control to be redirected, objects falling out of scope will not have their destructor called. This is because the ANSI C **setjmp/longjmp** mechanism does not support a mechanism for unwinding the stack.

Name: **IGNORE_ERRORS** — Ignores a raised exception within a body of code

Synopsis: **IGNORE_ERRORS** (**Exception*** *excp,* **Symbol*** *excp_type* = **Error**,
REST: *args*) { *body}*

excp — Pointer that is set to the exception object if one is raised while executing the statements in *body*; otherwise, this pointer is set to **NULL**

excp_type — A symbol representing the **Exception** class type of *excp* (that is, **Error**, **Warning**, and so forth)

args — One or more of the following comma-separated arguments or values:

group_name
One or more comma-separated pointers to **Symbol** objects representing aliases for this exception class type

body — Any valid C++ statements to be executed under the protection of the **IGNORE_ERRORS** macro

---

Jump_Handler Class

13.16 The **Jump_Handler** class is derived from the **Excp_Handler** class. It saves the current environment and also the exception object when an exception is raised. Instances of this class are used by the **IGNORE_ERRORS** macro discussed below. An exception handler function saves a pointer to the exception raised in the **Jump_Handler** exception object and then calls the system function **longjmp**, passing the environment that was saved in the **Jump_Handler** object by **setjmp**.

Note: The *excp_type* arguments in the **Jump_Handler** class constructors and member functions are pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` package, or some application-specific package. See section 11, Symbols and Packages, for more information.

Name: **Jump_Handler** — An exception handler class for ignoring exceptions.

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

Base Class: **Excp_Handler**

Friend Classes: None

Constructors:

Jump_Handler (**Jump_Handler_Function** *fn*, **Symbol*** *excp_type*)
Creates an jump handler object associated with the exception type *excp_type,* initializes the jump handler function data member to *fn,* and pushes itself on top of the global exception handler stack. The jump handler function is of type **void** (*Jump_Handler_Function*)(*Exception*, Excp_Handler**).

Jump_Handler (**Jump_Handler_Function** *fn*, **int** *number*,
Symbol* *excp_type1*, **Symbol*** *excp_type2, ...*);
Creates an jump handler object, creates *number* group names *excp_type1*, *excp_type2*, and so on if necessary, associates this jump handler object with *number* group names *excp_type1*, *excp_type2*, and so on, initializes the jump handler function data member to *fn,* and pushes itself on top of the global exception handler stack. The jump handler function is of type **void** (*Jump_Handler_Function*)(*Exception*, Excp_Handler*).

Member Functions:

virtual Boolean invoke_handler (**Exception*** *excp*)
Returns **TRUE** if the exception handler function was invoked for *excp*; otherwise, this function returns **FALSE**.

Friend Functions:

void ignore_errors_handler (**Exception*** *excp*, **Excp_Handler*** *fn*);
This exception handler function ignores exceptions raised through the macros **RAISE** and **STOP**. When invoked, this function saves a pointer to the exception object *excp* in the jump handler *fn*. It then calls **longjmp,** passing the environment saved in **Jump_Handler**. The program returns to the point after the call of **setjmp** in the macro **IGNORE_ERRORS** discussed below.

Name: **VERIFY** — Verify that an expression evaluates to non-zero

Synopsis: **VERIFY** (*test_expression*, **REST:** *args*);

test_expression Any valid C++ expression to be verified

args One or more of the following comma-separated arguments or values:

Symbol* *group_name*
One or more comma-separated pointers to **Symbol** objects representing aliases for this exception class type

const char* *format_string*
A character string compatible with the standard **printf** format containing the text of the error message

format_args
Any required argument(s) for the format string

key_value_args
The name(s) and value(s) of any public data members in the exception object

VERIFY Example

13.15 This example is another variation of the previous two examples. **VERIFY_ERROR** asserts that the index specified for a vector element is within range. It creates a **Verify_Error** exception object and raises the exception when the index for operator[] of a vector class is out of range.

```
1   inline int vector::operator[] (int n) {
2    VERIFY ((n >= 0 && n < this->number_elements),
3                   Error, "vector::operator[](): %d out of range", n);
4    return this->data[n];
5   }
```

Lines 1 through 5 implement the code necessary for a typical operator[] member function of a class for a vector of integers. However, before the indexed element is looked up and returned, the **VERIFY** macro invocation on lines 2 and 3 insures that the given index is within range. When the index provided is out of range, **VERIFY** creates an exception object and raises the exception to report the error. A handler for this exception could prompt the user for a new index and retry the operation. If no exception handler is found, program execution ends.

STOP Example

13.13 In this example, **STOP** creates an **Error** exception object and raises the exception when the index for operator[] of a vector class is out of range.

```
1   inline int vector::operator[] (int n) {
2     if (n >=0 && n < this->number_elements)
3                   return this->data[n];
4     else
5                   STOP (Error, "vector::operator[](): %d out of range", n);
6   }
```

Lines 1 through 6 implement the code necessary for a typical operator[] member function of a class for a vector of integers. However, when the index provided is out of range, the **STOP** macro invocation in line 5 creates an exception object and raises the exception to report the error. A handler for this exception could prompt the user for a new index and retry the operation. The distinction between the use of **STOP** and **RAISE** is that **STOP** guarantees to end the program if the exception is not handled, whereas **RAISE** will return.

VERIFY

13.14 The **VERIFY** macro asserts that an expression is **TRUE** by raising an exception of the appropriate type if it is **FALSE**. The exception type is optional, but if specified, is the group name or alias of the **VERIFY_ERROR** object created. This is because the macro assumes that a public data member named **test** is defined. If the exception type is not specified, no other arguments can be provided. **VERIFY** is similar to **RAISE** in that it uses **EXCEPTION** to construct the exception object and then calls the function **raise** to raise the exception. This function searches for an exception handler of the appropriate type to handle the exception and, if found, invokes the exception handler function and returns the exception object. If no exception handler is found, program execution ends.

NOTE: The **VERIFY** macro takes some arguments that are actually pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL SYM and ERR_MSG packages, or some application-specific package. See section 11, Symbols and Packages, for more information.

STOP

13.12 The **STOP** macro raises an exception and ends program execution with **exit** if the exception is not handled. By default in COOL, only exceptions of type **Error** will exit and exceptions of type **Fatal** will abort. **STOP** is similar to **RAISE** in that it uses **EXCEPTION** to construct the exception object and then calls its member function **raise** to raise the exception. This function searches for an exception handler of the appropriate type to handle the exception and, if found, invokes the exception handler function and returns the exception object. If no exception handler is found, however, program execution ends. There are many variations of **STOP** that provide flexible and efficient means of customizing the exception object and raising the exception. In particular, the variable number of group name arguments should reduce the need for many different types of exception classes whose only difference is the type name.

NOTE: The **STOP** macro takes some arguments that are actually pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` and `ERR_MSG` packages, or some application-specific package. See section 11, Symbols and Packages, for more information.

Name: **STOP** — Raise an exception and end the program if not handled

Synopsis: **STOP** (**Symbol*** *excp_type*, **REST:** *args);*

excp_type — A symbol representing the **Exception** class type (that is, **Error**, **Warning**, and so forth)

args — One or more of the following comma-separated arguments or values:

Symbol* *group_name*
: One or more comma-separated pointers to **Symbol** objects representing aliases for this exception class type

const char* *format_string*
: A character string compatible with the standard **printf** format containing the text of the error message

format_args
: Any required argument(s) for the format string

key_value_args
: The name(s) and value(s) of any public data members in the exception object

---

Name: **RAISE** — A COOL macro for constructing and raising an exception

Synopsis: **RAISE** (**Symbol*** *excp_type*, **REST:** *args*);

excp_type — A symbol representing the **Exception** class type (that is, **Error**, **Warning**, and so forth)

args — One or more of the following comma-separated arguments or values:

Symbol* *group_name*
One or more comma-separated pointers to **Symbol** objects representing aliases for this exception class type

const char* *format_string*
A character string compatible with the standard **printf** format containing the text of the error message

format_args
Any required argument(s) for the format string

key_value_args
The name(s) and value(s) of any public data members in the exception object

RAISE Example

13.11 In this example **RAISE** creates an **Error** exception object and raises the exception when the index for operator[] of a vector class is out of range.

```
1   inline int Vector::operator[] (int n) {
2     if (n >= 0 && n < this->number_elements
3                   return this->data[n];
4     else
5                   RAISE (Error, "vector::operator[](): %d out of range", n);
6   }
```

Lines 1 through 6 implement the code necessary for a typical operator[] member function of a class for vector of integers. However, when the index provided is out of range, the **RAISE** macro invocation in line 5 creates an exception object and raises the exception to report the error.

Lines 1 through 12 define a new exception class `Out_of_Range` derived from the COOL **Fatal** class. This new exception type has two public data members, `value` and `container`, whose values will be provided when creating an instance of this type. Lines 5 through 7 define the constructor for the new exception type that initializes the format message data member. Lines 8 through 11 implement a specialized **report** member function. It uses the polymorphic **type_of** member function of the container class inherited from **Generic**.

```
1    EXCEPTION(Out_of_Range, value=n, container=c1);
```

At some point in an application, line 1 invokes **EXCEPTION**, specifying the exception type `Out_Of_Range` as the first argument and intermixed format string arguments and data member initialization arguments of `value` and `container`. When expanded, this macro generates:

```
1    (Exception_g = new Out_of_Range(),
2    Exception_g->value = n,
3    Exception_g->container = c1,
4    Exception_g);
```

Line 1 assigns the global pointer `Exception_g` to point to a new instance of an `Out_of_Range` exception object. Lines 2 and 3 initialize the public data members of the exception object. Line 4 returns a pointer to the new exception object that can be raised as appropriate.

This example provides an interesting look at a general-purpose exception object that uses the polymorphic runtime type determination provided by the **Generic** class and the **class** macro. The exception type `Out_of_Range` could be used in many types of container classes (**Vector**<*Type*>, **List**<*Type*>, and so on) where a reference or index for some element is out of range. Any of these classes could raise this exception to display the error message and appropriate type-specific information without the need for a specialized exception type for each class.

RAISE

13.10 The **RAISE** macro allows an application program to create and raise an exception. **RAISE** uses **EXCEPTION** to construct the exception object and then calls its member function **raise**, defined as a friend function of the exception class, to raise the exception. This function searches for an exception handler of the appropriate type to handle the exception and, if found, invokes the exception handler function. It returns the exception object if the exception handler returns or if no exception handler is found. The exception object may be examined to determine if the exception was handled and if any alternate values were returned. There are many variations of **RAISE** that provide flexible and efficient means of customizing the exception object and raising the exception. In particular, the variable number of group name arguments should reduce the need for many different types of exception classes whose only difference is the type name.

NOTE: The **RAISE** macro takes some arguments that are actually pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` and `ERR_MSG` packages, or some application-specific package. See section 11, Symbols and Packages, for more information.

Lines 1 through 6 define a new exception class, `Bad_Argument_Error`, derived from the COOL **Fatal** class. This new exception type has two public data members, `arg_name` and `arg_value`, whose values will be provided when creating an instance of this type. Line 7 invokes **EXCEPTION**, specifying the exception type `Bad_Argument_Error` as the first argument. Notice there are no group names in this invocation. As a result, only an exception handler specifically created for exceptions of type `Bad_Argument_Error` can be called if a `Bad_Argument_Error` exception is raised. Line 8 contains the second argument, which is the error message control string, in standard **printf** format. Line 9 contains intermixed format string arguments and data member initialization arguments. When expanded, this macro generates:

```
1  (Exception_g = new Bad_Argument_Error(),
2  Exception_g->arg_name = "foo",
3  Exception_g->arg_value = x,
4  Exception_g->format_msg = hprintf(ERR_MSG("Argument %s has value %d\
5  which is out of range for vector %s."), "foo", x, vec1),
6  Exception_g);
```

Line 1 assigns the global pointer `Exception_g` to point to a new instance of a `Bad_Argument_Error` exception object. Lines 2 and 3 initialize the public data members of the exception object. Lines 4 and 5 initialize the format message field to the message argument passed, with the appropriate argument values inserted. Note that this message is actually a symbol in the COOL `ERR_MSG` package. Line 6 returns a pointer to the new exception object.

An exception handler for a `Bad_Argument_Error` exception could prompt the user for a new value for the named argument and return it in `arg_value` field if this exception object is raised. This can be done through the virtual **Exception::raise** member function or more conveniently with the **RAISE** macro discussed in paragraph 13.10 below.

Example 3:

This example is similar to the previous one, except that the constructor for the new exception type object initializes the format message field. This is a general-purpose exception type for any container class derived from the COOL **Generic** class as discussed in Section 12, Polymorphic Management. Providing a local **report** member function supercedes the virtual default implementation in the base **Exception** class.

```
1  Class Out_of_Range : public Fatal {
2  public:
3    int value;
4    Generic* container;
5    Out_of_Range() {
6     format_msg = "Value %d is out of range for container %s."
7    }
8    void report(ostream& os) {
9     Fatal::report (os);
10    os << form (format_msg, this->value, this->container->type_of());
11   }
12 };
```

EXCEPTION Examples

13.9 Here are three examples of the use of the **EXCEPTION** macro. Each makes use of a different form of the macro to show alternate features and usage. `Exception_g` is a global exception object pointer. `hprintf()` is a variation of the **printf** function which returns a format string allocated on the heap. Both of these are provided as part of the COOL exception handling facility. In addition, notice that the ordering of the *format_args* and *key_value_args* arguments in example two depends on the control characters in the format string. Finally, the code resulting from the macro expansion makes heavy use of the comma operator and is standard C++, although this might look a little confusing at first.

Example 1:

This is a simple use of **EXCEPTION** that specifies the exception type, a group name, and a format string:

```
1   EXCEPTION (Error, SYM(Serious_Error), "Serious problem here");
```

Line 1 contains an invocation of the `EXCEPTION` macro for an exception of type `Error` aliased with the group name `Serious_Error`. This group name symbol is reference through the COOL `SYM` package. The message text follows as the third argument. When expanded, this macro call generates:

```
2   (Exception_g = new Error(),
3   Exception_g.set_group_name (SYM(Serious_Error)),
4   Exception_g->format_msg = hprintf(ERR_MSG("Serious problem here.")),
5   Exception_g);
```

Line 2 assigns the global pointer `Exception_g` to point to a new instance of an `Error` exception object. Line 3 associates this exception object with the group name `Serious_Error`. Line 4 initializes the format message field to the message argument passed. Note that this message is actually a symbol in the COOL `ERR_MSG` package, thus facilitating collection of all error messages in one location, and affording the ability to have multiple translations of text for a single application. Line 5 returns a pointer to the new exception object.

An exception handler on the global exception handler stack that is associated with the group name `Serious_Error` will be called if this exception object is raised. This can be done through the virtual **Exception::raise** member function, or more conveniently with the **RAISE** macro discussed in paragraph 13.10 below.

Example 2:

In this example, a new exception class is derived containing two data members whose values are filled in when the exception object is created. **EXCEPTION** is invoked with an exception type, a format string, and a mix of data member arguments and format arguments.

```
1   Class Bad_Argument_Error : public Fatal {
2   public:
3     char* arg_name;
4     int arg_value;
5     Bad_Argument_Error ();
6   };

7   EXCEPTION (Bad_Argument_Error,
8    ERR_MSG("Argument %s has value %d that is out of range for vector %s "),
9    arg_name="foo", arg_value=x, vec1);
```

void Warning::default_handler ();

This function reports the exception message on the standard error stream and returns to the point at which the exception was raised.

EXCEPTION

13.8 The **EXCEPTION** macro simplifies the process of creating an instance of a particular type of exception object. It provides an interface for the application programmer to create an exception object using the specified arguments to indicate group name(s), initialize data members, or generate a format message. There are many variations of **EXCEPTION** that provide flexible and efficient means of customizing the exception object. In particular, the variable number of group name arguments should reduce the need for many types of exception classes whose only difference is the type name.

NOTE: The **EXCEPTION** macro takes some arguments that are actually pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` and `ERR_MSG` packages, or some application-specific package. See section 11, Symbols and Packages, for more information.

Name: **EXCEPTION** — A COOL macro for constructing an exception object

Synopsis: **EXCEPTION** (**Symbol*** *excp_type*, **REST:** *args*);

excp_type — A symbol representing the **Exception** class type (that is, **Error**, **Warning**, and so forth)

args — One or more of the following comma-separated arguments or values:

Symbol* *group_name*
One or more comma-separated pointers to **Symbol** objects representing aliases for this exception class type

const char* *format_string*
A character string compatible with the standard **printf** format containing the text of the error message

format_args
Any required argument(s) for the format string

key_value_args
The name(s) and value(s) of any public data members in the exception object

---

Predefined Exception Types and Handlers

13.7 COOL provides six predefined exception classes and five default exception handlers. Each of the predefined exception types has a default exception handler member function. The following rules apply in determining which handler function should be invoked for a particular type of exception:

- If no exception handler is found and the exception is of type **Error** or **Fatal**, its error message reports on the standard error stream and the program ends.
- If the exception is of type **Warning**, the warning message reports on the standard error stream and the program resumes at the point where the exception was raised.
- If the exception is of type **System_Error**, the system error message reports on the standard error stream and the program ends.
- If the exception is of type **System_Signal**, the signal error message reports on the standard error stream and the program resumes at the point where the system function **signal**() was called.
- If the exception is of type **Verify_Error**, the expression that failed assertion reports on the standard error stream and the program ends.

Exception is the base exception class and from it are derived **Warning, System_Signal, Fatal,** and **Error**. The **System_Error** and **Verify_Error** classes are derived from the **Error** class. The default exception handlers are called only if no other exception handler is established and available when an exception is raised.

For exceptions of type **Error** and **Fatal**, the exception handler reports the error message of the exception on standard error and ends the program. Exceptions of the type **Warning** report a warning message on standard error and return to the point at which the exception was raised. Exceptions of type **System_Error** report an error message on standard error and end the program. Finally, exceptions of type **System_Signal** report an error message on standard error and the program resumes execution at the point at which the system function **signal** was called. The following functions report exceptions and deal with them:

void Fatal::default_handler ();
This member function reports the exception message on the standard error stream and ends the program with a call to **abort**(), generating a core image that can be used for further debugging purposes.

void Error::default_handler ();
This member function reports the exception message on the standard error stream and ends the program normally with a call of **exit**(1).

void System_Error::default_handler ();
This member function reports the exception message on the standard error stream, sets the global system **errno** variable appropriately, and ends the program with a call to **abort**(), generating a core dump that can be used for further debugging purposes.

void System_Signal::default_handler ();
This function reports the exception message on the standard error stream and returns to the point at which a call to the system **signal()** function was made.

Excp_Handler Example

13.6 The following example shows a function that establishes an exception handler function for exceptions associated with a group name `File_Error` of type `My_Exceptions`. It then attempts to open each file indicated in an array of pointers to file name character strings.

```
1   #include <COOL/Exception.h>                                  // Include header
2   #include <My_Exceptions.h>                                   // My exception types
3   extern void my_file_handler (My_Exceptions* excp);           // Exception handler

4   FILE* open_f (char* file, char* mode) {
5     FILE* temp;                                                // Temp variable
6     if ((temp = fopen (file, mode)) == NULL) {                 // File open OK?
7      My_Excp1 (SYM(File_Error)) excp;                          // Create exception
8      excp->fname = file;                                       // Set file name
9      excp_>fmode = mode;                                       // Set file mode
10     excp->raise ();                                           // Raise exception
11    }
12  }

13  Boolean open_files (char** file_names, char** modes, FILE** f_handles) {
14    Excp_Handler eh (my_file_handler, SYM(File_Error));        // Setup handler
15    for (int i = 0; file_names[i] != NULL; i++)                // For each file
16     f_handles[i] = open_f (file_names[i], modes[i]);          // Open file
17  }
```

Line 1 includes the COOL **Exception** header file. Line 2 includes an application-specific header file that defines exception types derived from **Exception**. Line 3 is an external reference to some user-defined function to be called for exceptions of type `My_Exceptions`. For example, this function might prompt the user for a new file name and perform a retry operation. Lines 4 through 12 implement a function that attempts to open `file` in `mode` with the system function **fopen**. If the open fails, an exception `My_Excp1` associated with group name `File_Error` is created and raised. Line 7 uses the COOL `SYM` package in which to store the group name symbol. In a typical application, all application-specific symbols should be located in an application-specific package. Lines 8 and 9 set two public data member slots in the exception object, and line 10 raises the exception.

Lines 13 through 17 contain a function `open_files` that loops through an array of `file_names` and attempts to open each file in the function `open_f`. Line 14 is the heart of this function, where an exception handler object `eh` is created with a pointer to the function `my_file_handler` for exceptions of group name `File_Error`. This symbol is located in the COOL `SYM` package and would be referenced when an exception of type `My_Excp1` is raised, as in lines 7 through 10. See section 11, Symbols and Packages, for more information on the COOL symbol and package mechanism.

In this example, the exception handler `my_file_handler` is associated with the exception handler object `eh` created locally on line 14. When the constructor for `eh` is executed, a pointer to the exception handler object is placed on the global exception handler stack. While this object is in scope and not pre-empted by a more specific handler, any exception raised asssociated with the group name `File_Error` will be handled. When function `open_files` completes and destructor for `eh` called, the handler is removed from the global exception handler stack.

Excp_Handler Class

13.5 An exception handler provides a way to proceed from a particular type of exception by calling its exception handler function. An exception handler function could handle the exception by reporting the exception to standard error and ending the program, or dropping a core image for further debugging by the programmer. Another way of proceeding is to query the user for a fix, store the fix in the exception object, and return to the point where the exception was raised.

An instance of the **Excp_Handler** class is specified for a particular type of exception or one or more exception group names with an associated exception handler function. Such an instance invokes the specific exception handler function when an exception of the appropriate type is raised. The **Excp_Handler** class also contains data members that point to the top exception handler on the global exception handler stack and the next exception handler after itself. When an exception handler object is instantiated, it is placed at the top of the exception handler stack. When an exception is raised, the exception stack is searched from the top for an appropriate handler. When one is found, it is invoked and the exception object is passed as an argument. What action the exception handler function takes is determined by the type of exception and is discussed in paragraph 13.7, Predefined Exception Types and Handlers.

Name: **Excp_Handler** — The class for handling exceptions.

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

Base Class: **Generic**

Friend Class: **Exception**

Constructors:

Excp_Handler ()
Creates an exception handler object with defaults for the exception type and exception handler function and pushes itself on top of the global exception handler stack. The default exception type is **Error** and the default exception handler function is **void exit_handler**(*Exception**).

Excp_Handler (**Excp_Handler_Function** *fn*, **Symbol*** *excp_type*)
Creates an exception handler object associated with the exception type *excp_type,* initializes the exception handler function data member to *fn,* and pushes itself on top of the global exception handler stack. The exception handler function is of type **void** (*Excp_Handler_Function*)(*Exception**).

Excp_Handler (**Excp_Handler_Function** *fn*, **int** *number*,
Symbol* *excp_type1*, **Symbol*** *excp_type2, ...*);
Creates an exception handler object, creates *number* group names *excp_type1*, *excp_type2*, and so on if necessary, associates this exception handler object with *number* group names *excp_type1*, *excp_type2*, and so on, initializes the exception handler function data member to *fn,* and pushes itself on top of the global exception handler stack. The exception handler function is of type **void** (*Excp_Handler_Function*)(*Exception**).

Member Functions:

virtual Boolean invoke_handler (**Exception*** *excp*)
Returns **TRUE** if the exception handler function was invoked for *excp*; otherwise, this function returns **FALSE**.

void set_group_names (**Symbol*** *excp_type*);
Creates a group name *excp_type* in the COOL SYM package if necessary, and associates this exception object with the *excp_type* group name.

void set_group_names (**int** *number*, **Symbol*** *excp_type1*, **Symbol***
excp_type2, ...);
Creates *number* group names *excp_type1*, *excp_type2*, and so on in the COOL SYM package if necessary, and associates this exception object with group names *excp_type1*, *excp_type2*.

virtual void stop ();
Invoked to search for an exception handler when an exception is raised. If found, the associated handler function is called and the exception handled flag is set to **TRUE**; otherwise, this function sets the exception handled flag to **FALSE**. This function is identical to **raise** except that if the exception handler function returns or no exception handler is found, program execution is terminated.

Friend Functions:

friend ostream& operator<<(**ostream&** *os*, **const Exception*** *excp*)
Overloads the output operator to provide a formatted output capability for a pointer to an exception object *excp*.

friend ostream& operator<< (**ostream&** *os*, **const Exception&** *excp*);
Overloads the output operator to provide a formatted output capability for a reference to an exception object *excp*.

and illustrated in the example in paragraph 13.6, Excp_Handler Example. See section 11, Symbols and Packages, for more information.

Name: **Exception** — The base class for building exception objects.

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

Base Class: **Generic**

Friend Classes: None

Constructors:

Exception ();
Creates an exception object, initializes the format message and message prefix data members to **NULL**, and sets the exception handled flag to **FALSE**.

Exception (**Symbol*** *excp_type*);
Creates an exception object, creates a group name *excp_type* if necessary, associates this exception object with the *excp_type* group name, initializes the format message and message prefix message data members to **NULL**, and sets the exception handled flag to **FALSE**.

Exception (**int** *number*, **Symbol*** *excp_type1*, **Symbol*** *excp_type2, ...*);
Creates an exception object, creates *number* group names *excp_type1*, *excp_type2*, and so on if necessary, associates this exception object with group names *excp_type1*, *excp_type2*, and so on, sets the format and message prefix data members to **NULL**, and sets the exception handled flag to **FALSE**.

Member Functions:

virtual void default_handler ();
Default exception handler called when this type of exception is raised if no user-specified exception handler is found. This function does not set the exception handled flag in the exception object.

inline void handled (**Boolean** *handled*);
Sets the exception handled flag to *handled*.

inline Boolean is_handled () **const;**
Returns **TRUE** if the exception was handled; otherwise, return **FALSE**.

Boolean match (**Symbol*** *excp_type*);
Returns **TRUE** if this exception object is in the group name *excp_type*; otherwise, this function returns **FALSE**.

const char* message_prefix () **const;**
Returns the message prefix.

virtual void raise ();
Invoked to search for an exception handler when an exception is raised. If found, the associated handler function is called and the exception handled flag is set to **TRUE**; otherwise, this function sets the exception handled flag to **FALSE**. If the exception handler function returns or no exception handler is found, the program resumes execution at the point at which the exception was raised.

virtual void report (*ostream& os*) **const;**
Reports the exception message on the output stream *os*. The exception handler functions and the output operator function of **Exception** call this member function.

When an exception handler object is declared, it is placed on the top of a global exception handler stack. When an exception is raised, a search is made for an exception handler. The handler search starts at the top of the exception handler stack, with the most recently defined exception handler at the top of the stack. An exception handler function is called if a match is found between the exception type or group name of the exception raised, and a handler function on the exception handler stack.

The COOL exception handling facility provides several macros that simplify the process of creating, raising, and manipulating exceptions. These macros are implemented with the COOL macro facility discussed in Section 10, Macros. The **EXCEPTION** macro simplifies the process of creating an instance of a particular type of exception object. The **RAISE** macro allows the programmer to easily raise an exception and search for an exception handler. The **STOP** macro is similar to the **RAISE** macro, except that it guarantees to end the program if the exception is not handled. The **VERIFY** macro raises an exception if an assertion for some particular expression evaluates to **FALSE**. Finally, the **IGNORE_ERRORS** macro provides a mechanism to ignore an exception raised while executing a body of statements.

The COOL exception handling mechanism supports the concept of group names or aliases for classes of exceptions that require no specialization of the exception object, but do require a distinct name to provide a specific exception handler. For example, an index exception class to handle indexing range errors in a vector class could be defined with aliases established for **get** and **set** operations. The appropriate **get** and **set** member functions set the alias of the exception object as necessary, and provide a specialized exception handler. If an indexing exception is detected at some point by one of these member functions, the appropriate handler function can be invoked. As a result, two different exception handlers can be used while only one *type* of index exception object is required.

Exception Class

13.4 The **Exception** class reports exceptions through a message prefix and a format string. Both are implemented as public data members in the exception object. An exception handled status data member is also used to determine if the exception was handled. All of these are implemented as public data members, which makes access easier by the exception handler functions and the **EXCEPTION** macro. In addition, a list of exception types is maintained in a group names data member to support aliasing and subclassing of exception types. The **Exception** class includes member functions for reporting a message (using the message prefix and format string) on a specified output stream, determining if the exception object is of a particular type or group, and setting the exception handled status. Finally, it also includes a member function that searches for a handler to invoke on this exception type.

Classes derived from the **Exception** class save the state of the situation and communicate this information to exception handlers. When an exception can be fixed and proceeding from the exception is possible, information on how the exception handler proceeded from the exception is also stored in the exception object by the invoked exception handler function.

Note: The *excp_type* arguments in the **Exception** class constructors and member functions are pointers to **Symbol** objects. These arguments control the relationship of an exception object with one or more exception handlers. They can be the symbol representing the name of a class (as with **Error** or **Warning**) that is created automatically for any class derived from **Exception** through the **Generic** class and the **class** macro. The arguments can also be symbol aliases created in the COOL `SYM` package, or some application-specific package. This is discussed in the next paragraph, Excp_Handler Class,

EXCEPTION HANDLING

Introduction

13.1 The **Exception** class, its derived classes, **Excp_Handler** class, and the exception interface macros offer programmers an easy means of reporting and handling exceptions in an application. This section discusses the base **Exception** and **Excp_Handler** classes. It also covers predefined exceptions and exception handlers, referencing exceptions as symbols in a package, exception group names (aliases), the report message text package, and user-defined exceptions.

Requirements

13.2 This section assumes that you have a working knowledge of C++ and have read and understood Section 10, Macros, and Section 11, Symbols and Packages.

Exceptions

13.3 In COOL, program anomalies are known as exceptions. An exception can be an error, but it can also be a problem such as impossible division or information overflow. Exceptions can impede the development of object-oriented libraries. Exception handling offers a solution by providing a mechanism to manage such anomalies and simplify program code.

The C++ exception handling scheme is a raise, handle, and proceed mechanism similar to the Common Lisp Condition Handling system. When a program encounters an anomaly that is often (but not necessarily) an error, it has the following options:

- Represent the anomaly in an object called an exception
- Announce the anomaly by raising the exception
- Provide solutions to the anomaly by defining and establishing handlers
- Proceed from the anomaly by invoking a handler function

The COOL exception handling facility provides an exception class (**Exception**), an exception handler class (**Excp_Handler**), a set of predefined exception subclasses (**Warning**, **Error**, **Fatal**, and so on), and a set of predefined exception handler functions. In addition, the macros **EXCEPTION**, **RAISE**, **STOP**, and **VERIFY** allow the programmer to easily create and raise an exception at any point in a program.

When an exception is raised (through macros **RAISE** or **STOP**, for example), a search begins for an exception handler that handles this type of exception. An exception handler, if found, deals with the exception by calling its exception handler function. The exception handler function can correct the exception and continue execution, ignore the exception and resume execution, or end the program. In COOL, an exception handler for each of the predefined exception types exists on the global exception handler stack.

An exception handler invokes a specific exception handler function for a specific type of exception or group of exceptions. Handling an exception means proceeding from the exception. An exception handler function could report the exception to standard error and end the program, or drop a core image for further debugging by the programmer. Another way of proceeding is to query the user for a fix, store the fix in the exception object, and return to where the exception was raised.

Printed on: Wed Apr 18 07:14:46 1990

Last saved on: Tue Apr 17 13:37:47 1990

Document: s13

For: skc

pl2ps 3.2.1 Copyright 1987 Interleaf, Inc.