NAME

errors – debugging a Shore application

DESCRIPTION

The *Shore return code* class, *shrc,* is a data structure that is fundamental to all error-handling for Shore applications. It is a pointer to a data structure that contains an error code and a small stack trace (limited to 3 levels) of the functions in which the error was detected.

Most Shore functions return a *shrc.* The include files for application programs define a set of macros that are useful for interpreting and using the *shrc.* The macros are described in some detail in the sections below. They are summarized here:

SH_DO(op)
: A macro that aborts the transaction if the operator *op* returns an error. See **transaction(oc)** for details.

SH_HANDLE_ERROR(op)
: Executes the operation *op* and if an error is returned, this macro calls the installed error handler and exits. See **errors(oc)** for details about installing an error handler.

SH_HANDLE_NONFATAL_ERROR(op)
: This is the same as SH_HANDLE_ERROR except that it does not exit after calling the installed error handler.

SH_NEW_ERROR_STACK(rc)
: This macro does not execute an operation; rather, it is meant to be called after an operation has already returned, and this macro manipulates the return code. It creates a new *shrc* containing the error code in *rc,* but with a new stack trace that contains only the line and file where this macro is invoked. Read on to see how this i used.

SH_RETURN_ERROR(rc)
: If *rc* represents an error, this macro adds stack trace information to *rc.* At most 3 lines of stack trace can be stored in a *shrc* structure.

The rest of this manual page describes some common errors that application programs have to address and some common programming errors. These examples use the following macro (not supplied in any of the Shore include files):

```
#define PERROR(rc) { if(rc) {    cerr << __LINE__ << " " <<__FILE__<<":"<<endl;
```

ERRORS IN new() and Ref<T>::new_persistent()

A major disadvantage of using the the C++ convention for creating objects, namely the operator **new,** is that there is no convenient way to deliver error indications to the caller. If a low-level error occurs during an attempt to create an object, an *error handler* is called. You can install your own error handler, or you can use the default one. In any case, the error handler function is passed a *shrc,* which can be printed by the error handler; unfortunately, it does not contain the stack trace information describing the code that called **new().** After the error handler is called, the program exits, because this is a fatal error that cannot be cleanly handled. (If the class has virtual functions, C++ virtual function tables are created after the space is allocated, which results in an ungraceful error caused by dereferencing the *this* null pointer.) The following example illustrates an error occurrence in **new().**

```
char *fname = "/nonexistentdir/junk";
Ref<SdlUnixFile> o;
o = new (fname, 0755) SdlUnixFile;
```

results in the following message, printed by the default error handler, when `/nonexistentdir` does not exist:

```
1. error in ObjCache.C:3539 The named object was not found
        called from:
        0) OCRef.C:69
        1) ../../src/sdl/include/sdl_UnixFile.h:28
        2) in operator new():0
2. error in ObjCache.C:949 The requested object was not found
```

The preferable way to create objects, when an error might occur during their creation, is with **Ref<T>::new_persistent.** The following example shows how:

```
char *fname = "/nonexistentdir/junk";
Ref<SdlUnixFile> o;
rc = Ref<SdlUnixFile>::new_persistent(fname, 0755, o);
SH_NEW_ERROR_STACK(rc);
if(!o) {
    cerr << "Could not create " << fname  << endl;
    PERROR(rc);
    ... perhaps abort the transaction
}
```

This results in the following message,

```
Could not create /nonexistentdir/junk
174 error.C:
1. error in error.C:171 The named object was not found
```

DEREFERENCING A BAD Ref<T>

If a bad Ref<T> is dereferenced, and if the lower layers can detect the problem in time (e.g., the reference is null), the error handler will be called. If the value of the reference is garbage, but not null (perhaps a wayward pointer resulted in scribbling on the reference), the lower layers will not detect the bad reference, and your program will behave in undefined ways like any other program that uses a garbage pointer.

References are initialized to null, so that these two statements are equivalent:

```
Ref<SdlUnixFile> o(0);
Ref<SdlUnixFile> o;
```

If the error handler is called during a dereference (operator->), the program exits after calling the handler. If you are debugging a program that suffers this fate, you can use a debugger to break (stop) in the error handler, and see a full stack trace that shows the source of the problem. If you have not installed your own error handler, set the breakpoint in

```
OCRef::default_error_handler
```

USING shrc TO GET STACK TRACES

When a *shrc* is printed directly after returning from a method in the class *Shore*, or from any low-level Shore library function, the stack trace is of little use to you if you don't care to look through the source code for the Shore libraries. You can use the macro SH_NEW_ERROR_STACK to replace this stack trace with a trace of the application functions in which the error is detected, or he can retain the low-level stack and attach a new stack that traces the application functions in which the error is detected, The following

example illustrates how the this is done.

```
shrc
stat_it(const char *fname)
{
    shrc rc;
    rc = Shore::stat(fname, &statbuf);
    SH_NEW_ERROR_STACK(rc);
    PERROR(rc); // prints rc, clears rc
    ...
    return rc;
}
```

In this example, when the application is handed a return code from a method of *class Shore*, it checks the return code, and if appropriate, it replaces the return code with a new one. The new return-code's stack trace begins with the line that creates the new return code.

The above code produces a message like this, if there is no object with the name *fname:*

```
1. error in error.C:35 The named object was not found
```

If your application program has many levels of function calls and you want the intermediate functions to add stack trace information to the return code, follow this example:

```
shrc
stat_it(const char *fname)
{
    shrc rc;
    rc = Shore::stat(fname, &statbuf);
    SH_NEW_ERROR_STACK(rc);
    // leave rc intact, don't print
    return rc;
}

shrc
intermediate_func(const char *fname)
{
    shrc rc = stat_it(fname);
    ...
    // add stack trace info if error
    SH_RETURN_ERROR(rc);
}

main() {
    ...

    if (rc=intermediate_func(fname))  PERROR(rc);
    ...
```

The above example yields error message like this, if the caller of

```
1. error in error.C:46 The named object was not found
        called from:
        0) error.C:53
```

The implementation of the return code classes permits at most 3-level stacks of trace information.

Finally, if you would rather keep the entire stack trace from lower layers, and you would like to use your installed error handler (see below), use the macro SH_HANDLE_NONFATAL_ERROR or the macro use the macro SH_HANDLE_ERROR as follows:

```
    SH_HANDLE_NONFATAL_ERROR(Shore::stat(fname, &statbuf));
or
    SH_HANDLE_ERROR(Shore::stat(fname, &statbuf));
```

both of which produce results like this:

```
1. error in ObjCache.C:3539 The named object was not found
        called from:
        0) Shore.C:457
        1) error.C:62
2. error in ObjCache.C:1258 The requested object was not found
```

The macro SH_HANDLE_NONFATAL_ERROR
calls the installed error handler and returns; SH_HANDLE_ERROR calls the installed error handler and exits. If you have not installed an error handler, the default error handler will be called. See **errors(oc)** for information about installing an error handler.

Error not checked

This message comes from the bowels of the code that implements the *shrc* class. It is printed if a **shrc** is created, then destroyed without ever having been checked, for example

```
{
    shrc rc1 = RC(EINTR);
    shrc rc2 = RC(EINTR);
    if(rc2) { ... }
}
```

When the scope of *rc1* is left, the message will be printed. When the scope of *rc2* is left, no message will be printed because *rc2* was checked.

You can locate the causes of these errors with *gdb* by setting a breakpoint in **w_rc_t::error_not_checked ,** and printing a stack trace when the breakpoint is reached.

If you try apply a non-const method to a const reference, you will get an error at compile-time.

ACCESS PERMISSIONS

When when the object in question is a registered object, you can check your access rights for that object before trying to use the object.

```
    int errno;

    SH_DO(Shore::access(fname, W_OK, errno));
    if(errno) {
        rc = RC(errno);
        SH_NEW_ERROR_STACK(rc);
        PERROR(rc);
    }
```

See **access(oc)** for details. The result of the above code is:

```
1. error in error.C:257 Permission denied
```

VERSION

This manual page applies to Version 1.1 of the Shore software.

SPONSORSHIP

The Shore project is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

SEE ALSO

intro(cxxlb), assign(cxxlb), create(cxxlb), ref(cxxlb), update(cxxlb), valid(cxxlb), access(oc), errors(oc), and **stat(oc).**