LIVING IN AN EXCEPTIONAL WORLD

Handling exceptions is a difficult but important part of developing Macintosh applications. This article provides a methodology as well as a set of C tools for handling exceptions and writing robust code. Techniques and examples are provided for dealing with some of the Toolbox idiosyncrasies, and some interesting features of the C preprocessor, MacsBug, and MPW are explored.

SEAN PARENT

Writing software on the Macintosh can be difficult. Writing robust software on the Macintosh is even more difficult. Every call to the Toolbox is a potential source of a bug and there are too many cases to handle—what if there isn't enough memory, or the disk containing the code has been ejected, or there isn't enough stack space, or the printer is unplugged, or . . . The list goes on, and a well-written application is expected to handle every case—always recovering without loss of information. By looking at how software is developed, this article introduces a methodology and tools for handling the exceptional cases with minimal impact on the code that handles the task at hand.

VERSION 1: NORMAL FLOW OF CONTROL

When writing code, programmers usually begin by writing the normal flow of control—no error handling. The code shown below is a reconstruction of the first version of a printing loop routine that eventually went out as a Macintosh Technical Note, "A Printing Loop That Cares . . ." (#161). Note that comments were removed to make the structure more apparent.

```
#include <Printing.h>
#include <Resources.h>
#include <Memory.h>

void PrintStuff(void)
{
   GrafPtr      oldPort;
```


SEAN PARENT (AppleLink PARENT, Internet parent@apple.com) *is* a parent, but Parent is his last name, not his title. He grew up in Renton, Washington, with his parents (you know, the people who produced him), who are also Parents. Sean came to Apple to pursue his lifelong interest in reference manuals. He enjoys a good ANSI standards document during breakfast, and likes catchy punch lines such as, "No, no! I said 'ANSI,' not 'ASCII'!" Sean also likes to write a good hack, and consistently comes in next-to-second-best at the annual MacHack MacHax Hack Contest. Unable to hide his prowess, he gave in to the inevitable job at Apple, and now he wants to change the world, one programming paradigm at a time.•

```
short       copies, firstPage, lastPage, numCopies, printmgrsResFile,
            realNumberOfPagesInDoc, pageNumber;
DialogPtr   printingStatusDialog;
THPrint     thePrRecHdl;
TPPrPort    thePrPort;
TPrStatus   theStatus;

GetPort(&oldPort);
UnLoadTheWorld();
thePrRecHdl = (THPrint)NewHandle(sizeof(TPrint));
PrOpen();
printmgrsResFile = CurResFile();
PrintDefault(thePrRecHdl);
if (PrStlDialog(thePrRecHdl)) {
   realNumberOfPagesInDoc = DetermineNumberOfPagesInDoc(
      (**thePrRecHdl).prInfo.rPage);
   if (PrJobDialog(thePrRecHdl)) {
      numCopies = (**thePrRecHdl).prJob.iCopies;
      firstPage = (**thePrRecHdl).prJob.iFstPage;
      lastPage = (**thePrRecHdl).prJob.iLstPage;
      (**thePrRecHdl).prJob.iFstPage = 1;
      (**thePrRecHdl).prJob.iLstPage = 9999;
      if (realNumberOfPagesInDoc < lastPage) {
         lastPage = realNumberOfPagesInDoc;
      }
      printingStatusDialog =
         GetNewDialog(257, nil, (WindowPtr) -1);
      for (copies = 1; copies <= numCopies; copies++) {
         (**thePrRecHdl).prJob.pIdleProc = CheckMyPrintDialogButton;
         UseResFile(printmgrsResFile);
         thePrPort = PrOpenDoc(thePrRecHdl, nil, nil);
         pageNumber = firstPage;
         while (pageNumber <= lastPage) {
            PrOpenPage(thePrPort, nil);
            DrawStuff((**thePrRecHdl).prInfo.rPage,
               (GrafPtr)thePrPort, pageNumber);
            PrClosePage(thePrPort);
            ++pageNumber;
         }
         PrCloseDoc(thePrPort);
      }
      if ((**thePrRecHdl).prJob.bJDocLoop == bSpoolLoop) {
         PrPicFile(thePrRecHdl, nil, nil, nil, &theStatus);
      }
   }
}
```

```
   PrClose();
   DisposeHandle((Handle)thePrRecHdl);
   DisposeDialog(printingStatusDialog);
   SetPort(oldPort);
} /* PrintStuff */
```

VERSION 2: ERROR HANDLING ADDED

With code in the preliminary stage shown above, the flow of control is easy to follow. After writing it, the programmer probably read through it and added some error-handling code. Adding "if (error == noErr)" logic wasn't difficult, but it took some thought to determine how to handle the cleanup and deal with the two loops. Some more error-handling code may have been added after running the routine under stressful conditions. Perhaps it was reviewed by lots of people before it went out as a Technical Note. Here's the new version of the code (with the added error-handling code shown in bold):

```
#include <Printing.h>
#include <Resources.h>
#include <Memory.h>

void PrintStuff(void)
{
   GrafPtr      oldPort;
   short        copies, firstPage, lastPage, numCopies, printmgrsResFile,
                realNumberOfPagesInDoc, pageNumber, printError;
   DialogPtr    printingStatusDialog;
   THPrint      thePrRecHdl;
   TPPrPort     thePrPort;
   TPrStatus    theStatus;

   GetPort(&oldPort);
   UnLoadTheWorld();
   thePrRecHdl = (THPrint)NewHandle(sizeof(TPrint));
   if (MemError() == noErr && thePrRecHdl != nil) {
      PrOpen();
      if (PrError() == noErr) {
         printmgrsResFile = CurResFile();
         PrintDefault(thePrRecHdl);
         if (PrError() == noErr) {
            if (PrStlDialog(thePrRecHdl)) {
               realNumberOfPagesInDoc = DetermineNumberOfPagesInDoc(
                  (**thePrRecHdl).prInfo.rPage);
               if (PrJobDialog(thePrRecHdl)) {
                  numCopies = (**thePrRecHdl).prJob.iCopies;
```

```
                        firstPage = (**thePrRecHdl).prJob.iFstPage;
                        lastPage = (**thePrRecHdl).prJob.iLstPage;
                        (**thePrRecHdl).prJob.iFstPage = 1;
                        (**thePrRecHdl).prJob.iLstPage = 9999;
                        if (realNumberOfPagesInDoc < lastPage) {
                           lastPage = realNumberOfPagesInDoc;
                        }
                        printingStatusDialog =
                           GetNewDialog(257, nil, (WindowPtr) -1);
                        for (copies = 1; copies <= numCopies; copies++) {
                           (**thePrRecHdl).prJob.pIdleProc =
                              CheckMyPrintDialogButton;
                           UseResFile(printmgrsResFile);
                           thePrPort = PrOpenDoc(thePrRecHdl, nil, nil);
                           if (PrError() == noErr) {
                              pageNumber = firstPage;
                              while (pageNumber <= lastPage &&
                                    PrError() == noErr) {
                                 PrOpenPage(thePrPort, nil);
                                 if (PrError() == noErr) {
                                    DrawStuff((**thePrRecHdl).prInfo.rPage,
                                       (GrafPtr)thePrPort, pageNumber);
                                 }
                                 PrClosePage(thePrPort);
                                 ++pageNumber;
                              }
                           }
                           PrCloseDoc(thePrPort);
                        }
                     } else PrSetError(iPrAbort);
                  } else PrSetError(iPrAbort);
               }
            }
            if (((**thePrRecHdl).prJob.bJDocLoop == bSpoolLoop) &&
                  (PrError() == noErr)) {
               PrPicFile(thePrRecHdl, nil, nil, nil, &theStatus);
            }
            printError = PrError();
            PrClose();
            if (printError != noErr) PostPrintingErrors(printError);
         }
         if (thePrRecHdl != nil) DisposeHandle((Handle)thePrRecHdl);
         if (printingStatusDialog != nil) DisposeDialog(printingStatusDialog);
         SetPort(oldPort);
      } /* PrintStuff */
```

Can you easily follow the normal flow of control in the second version? What if an error occurs? Could an error ever go unreported? Could this code crash because it didn't handle an error? Does this routine always clean up after itself? These questions are difficult to answer because the normal flow of control of this routine is intertwined with the flow that will occur in the event of an error. If the two could be separated, it would be much easier to tell what the routine does normally and how things are handled when something goes wrong. Besides making the code easier to read, a methodology that allowed such separation would make the code easier to write, debug, and, maintain.

PROGRAMMING BY CONTRACT

Programming by contract is based on the assumption that all correct routines have a contract, either stated or implied, with their caller. The contract states that if a given set of preconditions is met, the routine either succeeds or flags an exception and leaves the machine in a known or determinable state. This contract is flexible enough to be applied to any correct code.

The secret to writing robust code is to understand what the preconditions of a given routine are, when an exception can be flagged, and how to handle the exception. Separating the logic that checks conditions and handles exceptions from the algorithm of the routine allows code to be written in a straightforward way with the flow of control seen as easily as in our first version.

PRECONDITIONS

The preconditions of a routine specify the state the machine must be in for the routine to execute without failure (where failure implies a crash—not flagging an exception). A routine may require in its precondition items such as

- a previously called initialization routine
- valid ranges for value parameters
- available memory
- initialization of global state
- a specific software version

For some routines the preconditions may be readily apparent either in the interface or in the documentation. Sometimes it's necessary to experiment to discover the preconditions of a routine. When writing a routine, the "strength" of the precondition can be set according to the use of the routine. For example, a routine named DivideLong is written with a description that states:

> Given two numbers, numer and denom, DivideLong will divide numer by denom, set numer to the result, and return noErr. If denom is zero, DivideLong will return divideByZeroErr and leave numer unchanged.

With this description, numer and denom can be any numbers of the proper type. This is a weak precondition. Another description might read:

> Given two numbers, numer and denom, DivideLong will divide numer by denom and return the result. If denom is zero, DivideLong will fail.

With the second description, it would be the caller's responsibility to ensure that denom isn't zero. This is a strong precondition.

In general, it's better to have a strong precondition in a routine that is used within a sequence of related routines or shares conditions with other routines, because it generates more efficient code by eliminating error checking. It's better to have a weak precondition in routines that are called only once or at the start of a sequence of related routines. Routines with weak preconditions free the caller from ensuring the state of the machine before making the call.

A precondition can be strengthened by the caller but must not be weakened. Strengthening is useful when you're making a sequence of related calls where being sure additional conditions are met guarantees that no routine flags an exception. For example, given the first description of DivideLong it would be valid for a caller to do the following:

```
if (denom != 0) {
   (void)DivideLong(&numer_a, denom);     /* Ignore return. */
   (void)DivideLong(&numer_b, denom);     /* Ignore return. */
} else HandleError();
```

This may be more desirable than checking for a result of divideByZeroErr after each call. An example of weakening a precondition would be to call DivideLong as described in the second description without ensuring that denom isn't zero. This would constitute a bug.

POST-CONDITIONS

Post-conditions specify the state of the machine on the return of a routine. They include side effects and changes to global state as well as function results and variable parameters. The post-conditions of a routine must be determinable for the routine to be correct. They don't vary in strength and, if not met, the routine has a bug. A thorough understanding of the post-conditions of a routine is required to ensure that the routine is being called correctly and that cleanup can occur when the routine flags an exception.

Sometimes it's necessary to rephrase the preconditions and post-conditions of a routine to use it correctly. For example, a common misconception is that the only preconditions for calling TEKey are that it has passed a valid TEHandle and the appropriate Managers have been initialized. Since there's no mechanism for TEKey

to flag an exception, the assumption is that it can't fail. But TEKey may need to grow the hText handle if the character isn't replacing others and isn't a backspace. Growing a handle requires memory—something there may not be enough of. Since TEKey can fail without flagging an exception with these preconditions, it appears to be incorrect and contain a bug. However, by strengthening the preconditions to require that hText must be able to grow by the size of a character, the routine is once again correct. Strengthening preconditions is an easy fix often used in system software. (See the section "Preflighting Calls" for tips on how to ensure preconditions.)

HOW TO WRITE CHECKS

The **check** macro is used to ensure that static preconditions and post-conditions are being met during development. It also documents conditions for you, making it a very useful tool that adds to the maintainability of the code. Unfortunately, these conditions cannot be expressed directly in the interface so as to be more apparent to the caller. The syntax for **check** is

check(*assertion***);**

To use the **check** macro, include Exceptions.h (provided on the *Developer CD Series* disc). For MacsBug, use ResEdit to add Exceptions.rsrc to the DebuggerPrefs file in the System Folder.

What **check** does depends on the setting of the compile-time variable DEBUGLEVEL. DEBUGLEVEL can be set to one of the following values:

- DEBUGOFF or DEBUGWARN: **check** does nothing and *assertion* is not evaluated.
- DEBUGMIN or DEBUGSYM: *assertion* is evaluated and, if it's false (zero), a debugger break is executed. (The debugger break is Debugger() for DEBUGMIN and SysBreak() for DEBUGSYM. The first is useful for low-level debuggers like MacsBug or TMON, the second for symbolic debuggers like SourceBug, SADE, or THINK C.)
- DEBUGON or DEBUGFULL: *assertion* is evaluated and, if it's false (zero), MacsBug is entered and the dprintf dcmd is invoked to display more information. If DEBUGON, *assertion* is displayed and if DEBUGFULL, the source code file and line number are also displayed (see "Wonders of MacsBug and dprintf" for more information about dprintf).

Normally, **check** is used at the start and end of a routine. At the start it's used to ensure that parameters are within a given range and are not specific values (such as nil). At the end it's used to ensure that allocations succeeded and results are as desired.

WONDERS OF MACSBUG AND DPRINTF

The MacsBug dcmd, dprintf, is used by the **require** and **check** macros to display useful debugging information. The dprintf command is also a powerful tool that provides all the features of the standard printf but uses MacsBug as the console. The dprintf command assumes MPW parameter-passing conventions. The syntax for dprintf is

dprintf([**no**]**trace,** *formatString***,** ...**);**

where **trace** and **notrace** are used to specify whether or not to continue after displaying the information in MacsBug. The variable *formatString* is a printf style-format string with some extensions (see the comment in the Exceptions.h file on the *Developer CD Series* disc). Following *formatString* are the parameters to display. This can be a very useful tool for viewing complex structures or difficult-to-read values like floating- or fixed-point numbers.

The implementation of the dprintf dcmd is shown in the DPrintf.c file on the CD. It's fairly straightforward and can be extended easily to add any special data types required (for example, a **t** format character that would take a pointer to text and an integer length and display the text). The dcmd is invoked from C using the inline declaration for dprintf. The inline declaration invokes the DebugStr trap and pushes a long on the stack. The push is required because DebugStr uses Pascal calling conventions and so pops the [**no**]**trace** string from the stack. Since dprintf is a C-based function, the stack is fixed, so the string isn't popped twice. Both **trace** and **notrace** are macro Pascal strings containing ";dprintf;doTrace" and ";dprintf". Since the strings begin with a semicolon, MacsBug interprets them as commands and executes them. The dcmd then fetches the parameters from the stack according to *formatString* and displays them. The MacsBug macro **doTrace** evaluates to "g" or "". It's used to switch tracing between **trace** and **break** by entering either **traceGo** or **traceBreak** in MacsBug.

When developing software, it's useful to insert dprintf statements to display information in sections of code that are executed only in unusual circumstances. If dprintf is bracketed with #if debugon / #endif directives, it compiles out when DEBUGLEVEL is set to DEBUGWARN or DEBUGOFF. With **trace** the information is displayed without seriously interrupting the execution of the code. The **trace** macro is also useful for logging timing statistics by displaying Ticks. Since *formatString* is interpreted in MacsBug with interrupts disabled, even a complex *formatString* has minimal impact on timing results.

MACSBUG POWER USER TIP

If you have more than one monitor, you can use the swap command to make MacsBug always visible and use dprintf with **trace** to continually log information. You can set which screen MacsBug uses by opening the Monitors control panel, holding down the Option key, and dragging the "Happy Macintosh" to the monitor on which you want to display MacsBug (you have to restart for it to take effect).

MPW POWER USER TIP

At the end of the comment for dprintf in Exceptions.h is a section that uses Echo to pipe code to the assembler.

```
/*******************************************
Echo "                                       ∂n∂
   PRINT     OFF,NOHDR                       ∂n∂
   INCLUDE   'Traps.a'                       ∂n∂
   PRINT     ON                              ∂n∂
   PROC                                      ∂n∂
   _DebugStr                                 ∂n∂
   SUBQ       #4,SP      ; Fix the stack     ∂n∂
   ENDPROC                                   ∂n∂
   END                                       ∂n∂
" | Asm -l
*******************************************/
```

If you select this section and press Enter, it generates a listing with hex output. This is a handy way to generate and document inline functions.

REQUIREMENTS FOR BETTER LIVING

Although **check** can ensure that preconditions and post-conditions are being met during development, **check** is of limited value in situations where it cannot be determined whether the conditions are being met statically, because

- it disappears when DEBUGLEVEL is set to DEBUGOFF
- it doesn't provide sufficient support for handling exceptions to return the machine to a known state

What's needed is a mechanism that does not compile out and provides the ability to invoke a handler when *assertion* fails.

WHAT WE REQUIRE

The **require** macro was created to make handling exceptions simpler. The syntax for **require** is

require(*assertion***,** *exception***);**

If *assertion* evaluates to false (zero), execution continues at the handler *exception*. (The *exception* parameter, by convention, shares the name of the routine that failed, but this isn't mandatory.) Handlers are typically written as shells with control falling from one to the next, cleaning up after prior calls along the way. The extent of the cleanup needed gets deeper as more of the routine succeeds. Figure 1 shows an extended form of **require** called **require_action**. The extended form executes a statement when *assertion* fails before executing the handler. This is most useful for setting an error variable. The syntax for **require_action** is

require_action(*assertion***,** *exception***,** *action***);**

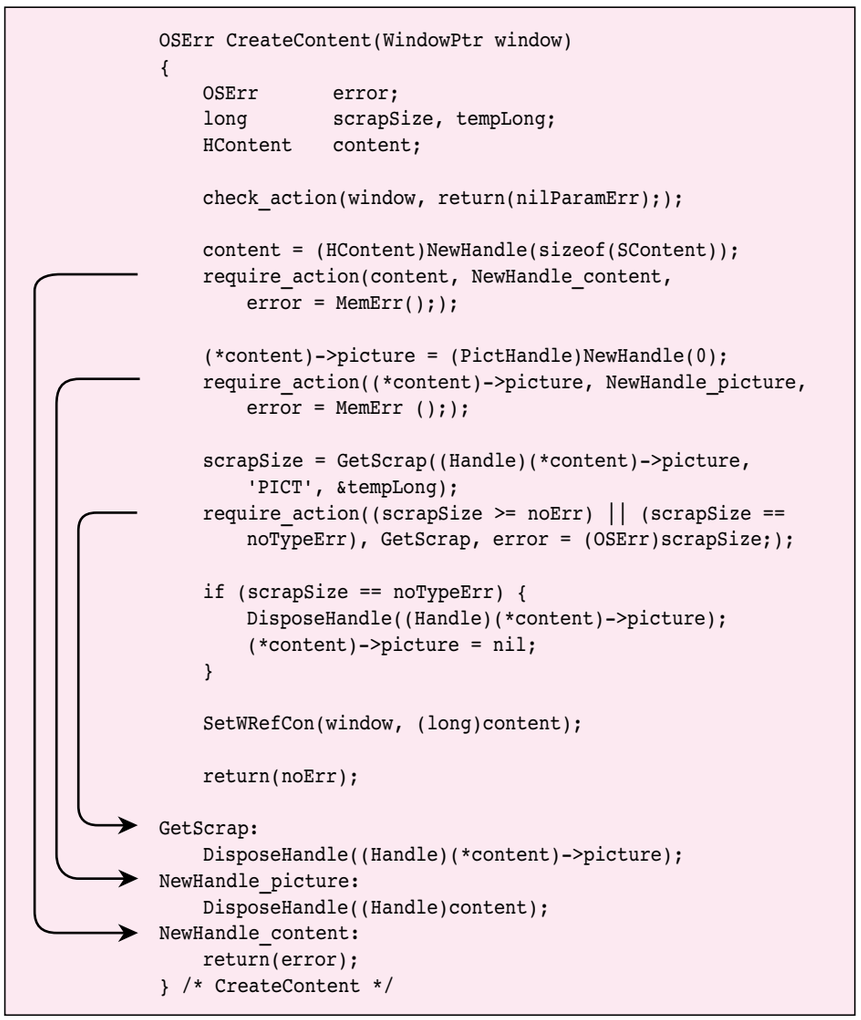
Like **check**, **require** breaks into MacsBug and displays pertinent information depending on the settings of DEBUGLEVEL. Unlike **check**, **require** does not compile out when DEBUGLEVEL is set to DEBUGWARN or DEBUGOFF. It evaluates *assertion* and invokes the handler (and *action*), but no break occurs.

The **nrequire** macro is equivalent to **require(!***assertion***,** *exception***)**. However, under rare circumstances it generates more efficient code, and when debugging is on, it displays the value of *assertion*. It's also easier to read. As a general rule, use **require** with handles and pointers and **nrequire** with errors.

VERSION 3: IMPROVED WITH REQUIRE

A close look at the code in version 2 reveals some problems:

- No error handling is done after PrCloseDoc, though any errors will get caught either after the next PrOpenDoc or on exit.



```
OSErr CreateContent(WindowPtr window)
{
    OSErr       error;
    long        scrapSize, tempLong;
    HContent    content;

    check_action(window, return(nilParamErr););

    content = (HContent)NewHandle(sizeof(SContent));
    require_action(content, NewHandle_content,
        error = MemErr(););

    (*content)->picture = (PictHandle)NewHandle(0);
    require_action((*content)->picture, NewHandle_picture,
        error = MemErr (););

    scrapSize = GetScrap((Handle)(*content)->picture,
        'PICT', &tempLong);
    require_action((scrapSize >= noErr) || (scrapSize ==
        noTypeErr), GetScrap, error = (OSErr)scrapSize;);

    if (scrapSize == noTypeErr) {
        DisposeHandle((Handle)(*content)->picture);
        (*content)->picture = nil;
    }

    SetWRefCon(window, (long)content);

    return(noErr);

GetScrap:
    DisposeHandle((Handle)(*content)->picture);
NewHandle_picture:
    DisposeHandle((Handle)content);
NewHandle_content:
    return(error);
} /* CreateContent */
```

Figure 1
Control Flow for **require_action**

- No error handling is done for GetNewDialog; if it fails it may result in a crash.
- If the NewHandle at the start of the code fails, it won't print and the user is never notified why.
- If an error occurs in the copies loop, the loop isn't terminated.

If printing is well behaved and does nothing once PrError has been set, none of these problems poses much of a threat to the actual stability of the code (with the exception of GetNewDialog). However, the use of **require** when writing the code could have avoided the problems and the code would be easier to understand and maintain. This is shown in the code below—version 3. The structure of the code in version 3 is almost identical to version 1 with the addition of the **require** statements and the handlers at the end. Writing code like this is straightforward. When a routine is called that can flag an exception, a **require** statement is added with a handler. The statements executed in a handler typically clean up after the routines called before the routine flagging the exception. (See the section "When To Clean Up" for more discussion.) Although PrClose should never cause an error, a **check** statement was added during development.

Here's version 3 of the code (with changes from version 1 shown in bold):

```
#include <Printing.h>
#include <Resources.h>
#include <Memory.h>
#include <Errors.h>
#include "Exceptions.h"

void PrintStuff(void)
{
   GrafPtr      oldPort;
   short        copies, firstPage, lastPage, numCopies, printmgrsResFile,
                realNumberOfPagesInDoc, pageNumber;
   DialogPtr    printingStatusDialog;
   OSErr        theError;
   THPrint      thePrRecHdl;
   TPPrPort     thePrPort;
   TPrStatus    theStatus;
   long         contig, total;

   enum { dialogSlop = 8192 };

   GetPort(&oldPort);
   UnLoadTheWorld();

   thePrRecHdl = (THPrint)NewHandle(sizeof(TPrint));
   require_action(thePrRecHdl, NewHandle, theError = MemError(););

   PrOpen();
   nrequire(theError = PrError(), PrOpen);

   printmgrsResFile = CurResFile();
```

```
PrintDefault(thePrRecHdl);
nrequire(theError = PrError(), PrintDefault);

if (PrStlDialog(thePrRecHdl)) {
   realNumberOfPagesInDoc = DetermineNumberOfPagesInDoc(
      (**thePrRecHdl).prInfo.rPage);
   if (PrJobDialog(thePrRecHdl)) {
      numCopies = (**thePrRecHdl).prJob.iCopies;
      firstPage = (**thePrRecHdl).prJob.iFstPage;
      lastPage = (**thePrRecHdl).prJob.iLstPage;
      (**thePrRecHdl).prJob.iFstPage = 1;
      (**thePrRecHdl).prJob.iLstPage = 9999;
      if (realNumberOfPagesInDoc < lastPage) {
         lastPage = realNumberOfPagesInDoc;
      }

      PurgeSpace(&total, &contig);
      require_action(contig >= dialogSlop, PurgeSpace,
         theError = memFullErr;);

      printingStatusDialog =
         GetNewDialog(257, nil, (WindowPtr) -1);
      require_action(printingStatusDialog, GetNewDialog,
         theError = memFullErr;);

      for (copies = 1; copies <= numCopies; copies++) {
         (**thePrRecHdl).prJob.pIdleProc = CheckMyPrintDialogButton;
         UseResFile(printmgrsResFile);
         thePrPort = PrOpenDoc(thePrRecHdl, nil, nil);
         nrequire(theError = PrError(), PrOpenDoc);

         pageNumber = firstPage;
         while (pageNumber <= lastPage) {
            PrOpenPage(thePrPort, nil);
            nrequire(theError = PrError(), PrOpenPage);

            DrawStuff((**thePrRecHdl).prInfo.rPage,
               (GrafPtr)thePrPort, pageNumber);
            PrClosePage(thePrPort);
            nrequire(theError = PrError(), PrClosePage);

            ++pageNumber;
         }
         PrCloseDoc(thePrPort);
         nrequire(theError = PrError(), PrCloseDoc);
      }
```

```
            if ((**thePrRecHdl).prJob.bJDocLoop == bSpoolLoop) {
               PrPicFile(thePrRecHdl, nil, nil, nil, &theStatus);
               nrequire(theError = PrError(), PrPicFile);
            }
         }
      }

      PrClose();
      ncheck(PrError());

      DisposeHandle((Handle)thePrRecHdl);
      DisposeDialog(printingStatusDialog);
      SetPort(oldPort);
      return;

PrOpenPage:
      PrClosePage(thePrPort);
PrClosePage:
PrOpenDoc:
      PrCloseDoc(thePrPort);
PrPicFile:
PrCloseDoc:
      DisposeDialog(printingStatusDialog);
GetNewDialog:
PurgeSpace:
PrintDefault:
PrOpen:
      PrClose();
      DisposeHandle((Handle)thePrRecHdl);
NewHandle:
      SetPort(oldPort);
      PostPrintingErrors(theError);
} /* PrintStuff */
```

PREFLIGHTING CALLS

Preflighting a call is the process of ensuring that the preconditions are met. Usually this isn't necessary since the preconditions will be satisfied by handling the exceptions of prior calls or will be implicit in the caller's preconditions. For example, there's no need to ensure that the TEHandle being passed to TEKey isn't nil if the exceptional case of the previous TENew returning nil was handled.

In case preconditions haven't been satisfied by handling the exceptions of previous calls, **require** can be used to check the precondition and invoke a handler if it's not being met. This is especially useful for routines that have strong preconditions or

preconditions that are difficult to determine. Earlier, TEKey was used as an example of a routine with strong preconditions. To ensure the preconditions for TEKey, **require** could be used as follows:

```
OSErr SafeTEKey(short key, TEHandle hTE) {
   enum { teSlop = 1024 };

   OSErr     error;
   TEPtr     w           = *hTE;
   Handle    hText       = w->hText;
   short     teLength    = w->teLength;

   SetHandleSize(hText, teLength + teSlop);
   nrequire(error = MemError(), SetHandleSize);
   SetHandleSize(hText, teLength);
   TEKey(key, hTE);
   return(noErr);

SetHandleSize:
   return error;
}
```

The constant teSlop is used instead of 1 just to be safe. Adding some slop for routines with implied, rather than stated, preconditions is always a good idea.

For some routines the preconditions are too complex or subject to change to accurately state as an assertion. This is the case for GetNewDialog, as shown in version 3. GetNewDialog can fail when there isn't enough memory for one of the numerous QuickDraw elements to be allocated, to load the WDEF, or, if the dialog contains TextEdit items, to create the TEHandle. About all that can be done to guarantee that GetNewDialog succeeds is to ensure that there's a reasonable amount of memory available. It's fairly safe to rely on the Process Manager in System 7 to make sure there's space in the system heap for the WDEF. This is what's done in version 3. The assertion is based on contiguous memory instead of total memory in case the heap is too fragmented to allocate some of the larger blocks required. Sometimes all that can be done is to increase the chances of survival.

WHEN TO CLEAN UP

Just as preconditions can sometimes be tricky to determine, post-conditions can be hazardous as well. It's important to understand the post-conditions of the routines being called, so that the machine can be returned to a known state, ensuring valid post-conditions for the calling routine. Normally, if an exception is being raised, a routine should dispose of everything it *successfully* allocated, close everything it *successfully* opened, and release everything it *successfully* locked. So, if NewHandle is

CouldDialog, which was intended as a preflight tool for GetNewDialog, is a no-op in System 7 (it's been broken since the Macintosh II was introduced).•

called successfully, DisposeHandle is called in the handler. If OpenFile is called successfully, CloseFile is called in the handler.

But this rule isn't always true. One counterexample is the Printing Manager. Even if PrOpen flags an exception PrClose *must* be called. The same is true for PrOpenDocument and PrOpenPage.

Shared resources present another potential problem. If GetResource is successfully called on a system resource, it's a bad idea to release it, because it may also be in use by another routine. SetResLoad(false) and GetResource can be called to determine whether the resource is already in memory before loading it, and then it can be released only if it was loaded. This, however, is taking things to an extreme. It may be better to document that these resources may be loaded even if the routine flags an exception. Since this is determinable by the caller, it suffices as a valid post-condition.

FUTURE DIRECTIONS

The routines and macros provided in Exceptions.h lay the foundation for writing robust software. There are more sophisticated exception-handling mechanisms, such as the proposed **catch** and **throw** implementation for C++. Ada has a reasonable exception-handling mechanism, as does CLU and Eiffel. However, these mechanisms don't lend themselves to dealing with exceptions from routines that were not written using the same mechanism and so are difficult to use on the Macintosh when dealing with the OS and Toolbox. The **check** and **require** macros are flexible enough to be useful in most situations and are implemented in C, so they can be of value for many (if not most) existing projects. They are also C++ friendly and can be of great use to C++ programmers as well.

After you read the code in version 3 that uses these macros it should be fairly simple to answer the questions asked about version 2 at the beginning of the article. This is left as an exercise.

Turn the page if you want even more detail . . .

RELATED READING

- Macintosh Technical Note "A Printing Loop That Cares . . ." (formerly #161).
- *Object-Oriented Software Construction* by Bertrand Meyer (Prentice-Hall, 1988). Contains more information on programming by contract.
- *Debugging Macintosh Software with MacsBug* by Konstantin Othmer and Jim Straus (Addison-Wesley, 1991). Contains additional MacsBug tips.

MORE DETAIL THAN MOST FOLKS NEED

The **require** and **check** macro implementation is shown in Figure 2. To ensure that there aren't any side effects, any macro that's larger than a single statement is enclosed in **do { } while(false)**. This ensures that the macro behaves as a simple statement and can be used anyplace a simple statement would be (such as after an **if**). The **do { } while(false)** does not generate any object code. In some of the macros, **if** statements appear in the form

```
if (assertion) ; /* Do nothing. */
else { /* Do something. */ }
```

Under some conditions, this will generate more efficient code than

```
if (!assertion) { /* Do something. */ }
```

(This was especially true back in the days of the MPW 3.1 compiler.) There are variables declared within the scope of the macros when debugging is on. This avoids side effects caused by evaluating *assertion* multiple times (once in the condition and once to display it). For example:

```
nrequire(ReadCharacters(), Fail);
```

If ReadCharacters returned a value other than nil, MacsBug would be invoked to display the result before executing the handler Fail. Without the local variable, ReadCharacters would be executed a second time to display the value. The second execution may cause side effects like increasing a file pointer as well as reading in a different set of characters.

When *assertion* is an error code returned by a function, it can be assigned to a variable to preserve the error. This also keeps the exception-handling code enclosed within the **require** statement. For example:

```
nrequire(error = GetError(), Fail);
```

However, with warnings set to full, this invokes a warning because the assignment takes place as part of an **if** statement. Using

```
error = GetError();
nrequire(error, Fail);
```

generates identical code (at least with MPW 3.2) and doesn't cause any warnings.

A macro, **resume**, is provided for recovering from exceptions. It's used within a handler and takes the form

resume(*exception***);**

where *exception* corresponds to *exception* used in a **require** statement. The **resume** macro simply transfers control to the point immediately following the **require** statement. Because of the **resume** feature, multiple **require** statements cannot share the same exception handler. Sometimes sharing a handler is convenient, so **resume** can be disabled with a statement:

```
#define resumeLabel(exception)
```

To reenable **resume**, use

```
#define resumeLabel(exception)\
   resume_ ## exception:
```

There's also a **check_action** macro which, like **require_action**, allows a statement to be executed when *assertion* fails. The **check_action** macro compiles out like all **check** macros and should be viewed as a development-time tool only. Being able to execute a statement allows for the exit of a routine if the preconditions aren't met.

Seriously insane cycle counters take note that the MPW 3.2 C compiler doesn't reuse a register to store a variable in a local scope when the register was used in a prior scope containing a **goto** statement (**require** generates a **goto**). This can lead to code that isn't as efficient as it should be but can usually be coded around (it's difficult to generate in the first place). Hopefully this will be fixed in a future compiler.•

```
#define require(assertion, exception)          \
   do {                                        \
      if (assertion) ;                         \
      else {                                   \
         dprintf(notrace,                      \
            "Assertion \"%s\" Failed\n"        \
            "Exception \"%s\" Raised",         \
            #assertion, #exception);           \
         goto exception;                       \
         resumeLabel(exception);               \
      }                                        \
   } while (false)

#define check(assertion)                       \
   do {                                        \
      if (assertion) ;                         \
      else {                                   \
         dprintf(notrace,                      \
            "Assertion \"%s\" Failed",         \
            #assertion);                       \
      }                                        \
   } while (false)
```

Figure 2
Implementing **require** and **check**

THANKS TO OUR TECHNICAL REVIEWERS
Scott Boyd, Konstantin Othmer, Sam Weiss. Also, special thanks to everyone in the Print Shop (present and former members) for using this stuff and suggesting numerous improvements during the past few years.•