

Assert Use



INTRODUCTION

The core idea behind using asserts is to specify a condition, and trigger an event based on the case where the condition is false. In other words we assume that everything is OK, but if this is not the case, we want to signal the problem.

This simple concept is very powerful concerning debugging and testing. The more asserts we place into our code, the more sure we are about states, and what is going on inside the application.

The core problem with asserts is usually either laziness to implement these, or the worry that the application will slow down too much due to assert functions¹, or then there's no coherent way of defining asserts.

Now, ANSI C libraries has a function called `assert()` that will take a statement and error. However we want to have a much better control of what is happening when an assert is triggered. For instance, we would like to drop to a high level or a low level debugger, log it into a file, maybe present a power-user Alert with low level information that could be sent back to the developer during an alpha/beta test, or maybe even forward the information to another process or machine using Apple Events.

ASSERT Macros

A set of assert macros in the `DTSC++Library.h` is addressing this need. The use of these macros are very straight forward:

```
fError = ::AESend(&fAE, &fReply, fSendMode, fPriority, fTimeout,
nil, nil);
ASSERT(fError == noErr, ("\pProblems with AESend = %d");
```

or

```
fError = ::AESend(&fAE, &fReply, fSendMode, fPriority, fTimeout,
nil, nil);
VASSERT(fError == noErr, ("Problems with AESend = %d", fError));
```

¹This was one of the major reasons the early Eiffel compilers got a bad reputation, because asserts were turned on by default for the final builds :-)

The `ASSERT` takes two arguments, a statement that should evaluate to true, if false it will print out a message (`Str255`) defined in the second parameter.

`VASSERT` has also a condition in the first parameter, and accepts `printf` style variable arguments in the second parameter. Note that we are using C style, or actually `printf` style definitions instead of `Str255s`². Note also that we need to place this statement inside parenthesis because `VASSERT` will expand to a macro, and unless we clearly mark that this whole block is one single part, we will end up in trouble.

Labels

We have the following labels defined inside the header file:

```
// GLOBAL DEFINITIONS

// TESTING, used for flagging that we use test code inside the classes and
// frameworks
// TESTLEVEL0      - user friendly alerts as part of the application
// TESTLEVEL1      - minor test level (detailed error message in alert //
//                  box)
// TESTLEVEL2      - semi-major test level (high level debuggers)
// TESTLEVEL3      - major test level (low level debuggers)
// TESTLOG1        - trigger a file log of information
// TESTLOG2        - trigger a file log of information from a low level //
//                  debugger

#define TESTLEVEL3
```

The core idea is that a developer will specify how asserts will be printed. If none of these labels are defined, then both macros will expand to `((void)0)`, in other words no code.

`TESTLEVEL0` could be defined as a user friendly alert that either terminates the program with a friendly message, or otherwise does something that is part of the friendly look and feel of a Macintosh.

`TESTLEVEL1` will trigger an alert box with the message stated in the `ASSERT` part. This level could be used to create alpha/beta level applications that could be sent to end user testers, and they could dutifully report the messages back to the developer.

`TESTLEVEL2` will trigger a break into a high level debugger (`SADE`, `SourceBug`). In other words we are using `SysBreakStr`. Note that the Think debugger hates this trap (crashes), as well as the machine usually crashes if not high level debugger is running., so be careful. In the case of Think, use:

²We would have used C strings in `ASSERT` as well, but Think C does not have `debugstr` implemented, sigh...

TESTLEVEL3 will trigger a break into a low level debugger, we are using `DebugStr`. In the case of ASSERT we will drop directly into the next executable line. In the case of VASSERT we are not that lucky, because due to `va_arg` use for variable argument parsing VASSERT is calling various libraries, and we end up further down the chain of functions.

TESTLOG1 will define that we will print an entry into a file placed on the top level of the boot volume, with a time/date stamp.

TESTLOG2 will additionally do a low level debugger dump of the current register use, calling chain of functions and the currently executed instructions.

Other Flags

The FILEINFO flag defines if we want to add file/line information to the message. This information is taken from the `__FILE__` and `__LINE__` ANSI definitions. By default this flag is enabled.

Source Code

The macros are defined in the `DTSC++Library.h` file.

Future Enhancements

We are open for suggestions, one direction is to create a feature flagged with `TESTAELOG` that will send out Apple Events with the string attached to. This would make it possible to log the information into another process residing either inside the same Mac, or over the network to another system.