How to use assertions in C

John Reekie

This document describes a method for using assertions in C. It is based on Bertrand Meyer's paper ``Applying `Design by Contract'," in *IEEE Computer,* October 1992, pages 40-51. The document is essentially a cut from some notes I once wrote for a C programming course.

Assertions

An *assertion* specifies that a program satisfies certain conditions at particular points in its execution. There are three types of assertion:

Preconditions

: Specify conditions at the start of a function.

Postconditions

: Specify conditions at the end of a function.

Invariants

: Specify conditions over a defined region of a program.

An assertion violation indicates a bug in the program. Thus, assertions are an effective means of improving the reliability of programs-in other words, they are a systematic debugging tool. In this document, I mainly consider preconditions, and invariants not at all. (This will need to be fixed - hjr.)

Assertions in C

In C, assertions are implemented with the standard *assert* macro. The argument to *assert* must be true when the macro is executed, otherwise the program aborts and prints an error message. For example, the assertion

```
    assert( size <= LIMIT );
```

will abort the program and print an error message like this:

```
    Assertion violation: file tripe.c, line 34: size <= LIMIT
```

if *size* is greater than *LIMIT*.

Preconditions

Preconditions specify the input conditions to a function. Here is an example of a function with preconditions:

```
int
magic( int size, char *format )
{
    int maximum;

    assert( size <= LIMIT );
    assert( format != NULL );
    ...
```

These pre-conditions have two consequences:

1. *magic* is only required to perform its task if the pre- conditions are satisfied. Thus, as the writer of *magic*, you are not required to make *magic* do anything sensible if *size* or *format* are not as stated in the assertions.

2. The caller is certain of the conditions under which *magic* will perform its task correctly. Thus, if your code is calling *magic*, you must ensure that the *size* or *format* arguments to the call are as specified by the assertions.

Consider the following analogy. Suppose you (the function) are employed as an apple-packer. One of the conditions of your contract is that the temperature in the warehouse will be no greater than 30C. If the temperature exceeds 30C, you are not obliged to do anything: you can keep packing apples if you want to, or you can choose to go to the beach. Your employer (the caller), however, knows that you are not required to pack apples if the temperature exceeds 30C, so he or she makes sure that the air-conditioning in the warehouse is operating correctly.

Postconditions

Postconditions specify the output conditions of a function. They are used much less frequently than preconditions, partly because implementing them in C can be a little awkward. Here is an example of a postcondition in *magic*:

```
    ...
    assert( result <= LIMIT );
    return result;
}
```

The postcondition also has two consequences:

1. *magic* guarantees that the stated condition will hold when it completes execution. As the writer of *magic*, you must make certain that your code never produces a value of result that is greater than *LIMIT*.

2. The caller is certain of the task that *magic* will perform (provided its preconditions are satisfied). If your program is calling *magic*, then you know that the result returned by *magic* can be no greater than *LIMIT*.

Compare this with the apple-picker analogy. Another part of your contract states that you will not bruise the apples. It is therefore your responsibility to ensure that you do not (and if you do, you have failed.) Your employer is thus relieved of the need to check that the apples are not bruised before shipping them.

Recommended practice

Writing preconditions

The simplest and most effective use of assertions is as preconditions-that is, to specify and check input conditions to functions. Two very common uses are to assert that:

1. Pointers are not NULL.

2. Indexes and size values are non-negative and less than a known limit.

Each assertion must be listed in the Asserts section of the function description comment in the corresponding header file. For example, the comment describing *magic* will include:

```
 *  Asserts:
 *      'size' is no greater then LIMIT.
 *      'format' is not NULL.
 *      The function result is no greater than LIMIT.
 */
```

If there are no assertions, write ``Nothing'':

```
 *  Asserts:
 *      Nothing
 */
```

Satisfying preconditions

When your code calls a function with preconditions, you must ensure that the function's preconditions are satisfied. This does not mean that you have to include code to check the argument to very function that you call! For example, in the following code, resize does not need to check that the argument to measure is NULL, since its own assertion ensures this:

```
void
resize( int *value )
{
    assert( value != NULL );
    ...
    measure( value, 0 );
    ...
```

In other words, you need to decide for yourself when and where values must explicitly be checked to avoid violating preconditions.

Assertion violations

If a precondition is violated during program testing and debugging, then there is a bug in the code that called the function containing the precondition. The bug must be found and fixed.

If a postcondition is violated during program testing and debugging, then there is a bug in the function containing the precondition. The bug must be found and fixed.

Assertions and error-checking

It is important to distinguish between program errors and run- time errors:

1. A program error is a bug, and should never occur.

2. A run-time error can validly occur at any time during program execution.

Assertions are not a mechanism for handling run-time errors. For example, an assertion violation caused by the user inadvertently entering a negative number when a positive number is expected is poor program design. Cases like this must be handled by appropriate error-checking and recovery code (such as requesting another input), not by assertions.

Realistically, of course, programs of any reasonable size do have bugs, which appear at run-time. Exactly what conditions are to be checked by assertions and what by run-time error- checking code is a design issue. Assertions are very effective in reusable libraries, for example, since i) the library is small enough for it to be possible to guarantee bug-free operation, and ii) the library routines cannot perform error- handling because they do not know in what environment they will be used. At higher levels of a program, where operation is more complex, run-time error-checking must be designed into the code.

Turning assertions off

By default, ANSI C compilers generate code to check assertions at run-time. Assertion-checking can be turned off by defining the NDEBUG flag to your compiler, either by inserting

```
#define NDEBUG
```

in a header file such as stdhdr.h, or by calling your compiler with the -dNDEBUG option:

```
cc -dNDEBUG ...
```

This should be done only you are confident that your program is operating correctly, and only if program run-time is a pressing concern.

About this document ...

The command line arguments were:
latex2html `-nolatex -split 0 assertions.tex`.

The translation was initiated by John Reekie on Thu Dec 7 17:25:37 PST 1995

John Reekie
Thu Dec 7 17:25:37 PST 1995