Chapter 33: Assertion

Parameter	Details
expression	expression of scalar type.
message	string literal to be included in the diagnostic message.

An ***assertion*** is a predicate that the presented condition must be true at the moment the assertion is encountered by the software. Most common are ***simple assertions***, which are validated at execution time. However, ***static assertions*** are checked at compile time.

Section 33.1: Simple Assertion

An assertion is a statement used to assert that a fact must be true when that line of code is reached. Assertions are useful for ensuring that expected conditions are met. When the condition passed to an assertion is true, there is no action. The behavior on false conditions depends on compiler flags. When assertions are enabled, a false input causes an immediate program halt. When they are disabled, no action is taken. It is common practice to enable assertions in internal and debug builds, and disable them in release builds, though assertions are often enabled in release. (Whether termination is better or worse than errors depends on the program.) Assertions should be used only to catch internal programming errors, which usually means being passed bad parameters.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int main(void)
{
    int x = -1;
    assert(x >= 0);

    printf("x = %d\n", x);
    return 0;
}
```

Possible output with `NDEBUG` undefined:

```
a.out: main.c:9: main: Assertion `x >= 0' failed.
```

Possible output with `NDEBUG` defined:

```
x = -1
```

It's good practice to define `NDEBUG` globally, so that you can easily compile your code with all assertions either on or off. An easy way to do this is define `NDEBUG` as an option to the compiler, or define it in a shared configuration header (e.g. `config.h`).

Section 33.2: Static Assertion

`Version ≥ C11`

Static assertions are used to check if a condition is true when the code is compiled. If it isn't, the compiler is required to issue an error message and stop the compiling process.

A static assertion is one that is checked at compile time, not run time. The condition must be a constant expression, and if false will result in a compiler error. The first argument, the condition that is checked, must be a constant expression, and the second a string literal.

Unlike assert, `_Static_assert` is a keyword. A convenience macro `static_assert` is defined in **`<assert.h>`**.

```
#include <assert.h>

enum {N = 5};
_Static_assert(N == 5, "N does not equal 5");
static_assert(N > 10, "N is not greater than 10");  /* compiler error */
```

Version = C99

Prior to C11, there was no direct support for static assertions. However, in C99, static assertions could be emulated with macros that would trigger a compilation failure if the compile time condition was false. Unlike `_Static_assert`, the second parameter needs to be a proper token name so that a variable name can be created with it. If the assertion fails, the variable name is seen in the compiler error, since that variable was used in a syntactically incorrect array declaration.

```
#define STATIC_MSG(msg, l) STATIC_MSG2(msg, l)
#define STATIC_MSG2(msg,l) on_line_##l##__##msg
#define STATIC_ASSERT(x, msg) extern char STATIC_MSG(msg, __LINE__) [(x)?1:-1]

enum { N = 5 };
STATIC_ASSERT(N == 5, N_must_equal_5);
STATIC_ASSERT(N > 5, N_must_be_greater_than_5); /* compile error */
```

Before C99, you could not declare variables at arbitrary locations in a block, so you would have to be extremely cautious about using this macro, ensuring that it only appears where a variable declaration would be valid.

Section 33.3: Assert Error Messages

A trick exists that can display an error message along with an assertion. Normally, you would write code like this

```
void f(void *p)
{
    assert(p != NULL);
    /* more code */
}
```

If the assertion failed, an error message would resemble

> Assertion failed: p != NULL, file main.c, line 5

However, you can use logical AND (`&&`) to give an error message as well

```
void f(void *p)
{
    assert(p != NULL && "function f: p cannot be NULL");
    /* more code */
}
```

Now, if the assertion fails, an error message will read something like this

> Assertion failed: p != NULL && "function f: p cannot be NULL", file main.c, line 5

The reason as to why this works is that a string literal always evaluates to non-zero (true). Adding `&& 1` to a Boolean expression has no effect. Thus, adding `&& "error message"` has no effect either, except that the compiler will display the entire expression that failed.

Section 33.4: Assertion of Unreachable Code

During development, when certain code paths must be prevented from the reach of control flow, you may use `assert(0)` to indicate that such a condition is erroneous:

```
switch (color) {
    case COLOR_RED:
    case COLOR_GREEN:
    case COLOR_BLUE:
        break;

    default:
        assert(0);
}
```

Whenever the argument of the `assert()` macro evaluates false, the macro will write diagnostic information to the standard error stream and then abort the program. This information includes the file and line number of the `assert()` statement and can be very helpful in debugging. Asserts can be disabled by defining the macro `NDEBUG`.

Another way to terminate a program when an error occurs are with the standard library functions `exit`, `quick_exit` or `abort`. `exit` and `quick_exit` take an argument that can be passed back to your environment. `abort()` (and thus `assert`) can be a really severe termination of your program, and certain cleanups that would otherwise be performed at the end of the execution, may not be performed.

The primary advantage of `assert()` is that it automatically prints debugging information. Calling `abort()` has the advantage that it cannot be disabled like an assert, but it may not cause any debugging information to be displayed. In some situations, using both constructs together may be beneficial:

```
if (color == COLOR_RED || color == COLOR_GREEN) {
   ...
} else if (color == COLOR_BLUE) {
   ...
} else {
   assert(0), abort();
}
```

When asserts are *enabled*, the `assert()` call will print debug information and terminate the program. Execution never reaches the `abort()` call. When asserts are *disabled*, the `assert()` call does nothing and `abort()` is called. This ensures that the program *always* terminates for this error condition; enabling and disabling asserts only effects whether or not debug output is printed.

You should never leave such an `assert` in production code, because the debug information is not helpful for end users and because `abort` is generally a much too severe termination that inhibit cleanup handlers that are installed for `exit` or `quick_exit` to run.

Section 33.5: Precondition and Postcondition

One use case for assertion is precondition and postcondition. This can be very useful to maintain invariant and design by contract. For a example a length is always zero or positive so this function must return a zero or positive value.

```c
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int length2 (int *a, int count)
{
    int i, result = 0;

    /* Precondition: */
    /* NULL is an invalid vector */
    assert (a != NULL);
    /* Number of dimensions can not be negative.*/
    assert (count >= 0);

    /* Calculation */
    for (i = 0; i < count; ++i)
    {
        result = result + (a[i] * a[i]);
    }

    /* Postcondition: */
    /* Resulting length can not be negative. */
    assert (result >= 0);
    return result;
}

#define COUNT 3

int main (void)
{
    int a[COUNT] = {1, 2, 3};
    int *b = NULL;
    int r;
    r = length2 (a, COUNT);
    printf ("r = %i\n", r);
    r = length2 (b, COUNT);
    printf ("r = %i\n", r);
    return 0;
}
```