# Assertion (software development)

From Wikipedia, the free encyclopedia

In computer programming, an **assertion** is a predicate (a true–false statement) placed in a program to indicate that the developer *thinks* that the predicate is always true at that place. If an assertion evaluates to false at run-time, an assertion failure results, which typically causes execution to abort.

## Details

The following code contains two assertions, `x > 0` and `x > 1`, and they are indeed true at the indicated points during execution:

```
x = 1;
assert (x > 0);
x++;
assert (x > 1);
```

Programmers can use assertions to help specify programs and to reason about program correctness. For example, a precondition—an assertion placed at the beginning of a section of code—determines the set of states under which the programmer expects the code to execute. A postcondition—placed at the end—describes the expected state at the end of execution. For example: `x > 0 { x++ } x > 1`

The example above uses the notation for including assertions used by C.A.R. Hoare in his 1969 paper.[1] That notation cannot be used in existing mainstream programming languages. However, programmers can include unchecked assertions using the comment feature of their programming language. For example, in C:

```
x = 5;
x = x + 1;
// {x > 1}
```

The braces included in the comment help distinguish this use of a comment from other uses.

Libraries may provide assertion features as well. For example, in C using glibc with C99 support:

```
#include <assert.h>
x = 5;
x = x + 1;
assert(x > 1);
```

Several modern programming languages include checked assertions - statements that are checked at runtime or sometimes statically. If an assertion evaluates to false at run-time, an assertion failure results, which typically causes execution to abort. This draws attention to the location at which the logical inconsistency is detected and can be preferable to the behaviour that would otherwise result.

The use of assertions helps the programmer design, develop, and reason about a program.

## Usage

In languages such as Eiffel, assertions form part of the design process; other languages, such as C and Java, use them only to check assumptions at runtime. In both cases, they can be checked for validity at runtime but can usually also be suppressed.

### Assertions in design by contract

Assertions can function as a form of documentation: they can describe the state the code expects to find before it runs (itspreconditions), and the state the code expects to result in when it is finished running (postconditions); they can also specifyinvariants of a class. Eiffel integrates such assertions into the language and automatically extracts them to document the class. This forms an important part of the method of design by contract.

This approach is also useful in languages that do not explicitly support it: the advantage of using assertion statements rather than assertions in comments is that the program can check the assertions every time it runs; if the assertion no longer holds, an error can be reported. This prevents the code from getting out of sync with the assertions.

## Assertions for run-time checking

An assertion may be used to verify that an assumption made by the programmer during the implementation of the program remains valid when the program is executed. For example, consider the following Java code:

```
int total = countNumberOfUsers();
if (total % 2 == 0) {
    // total is even
} else {
    // total is odd and non-negative
    assert(total % 2 == 1);
}
```

In Java, `%` is the *remainder* operator (or *modulus*): if its first operand is negative, the result can also be negative. Here, the programmer has assumed that `total` is non-negative, so that the remainder of a division with 2 will always be 0 or 1. The assertion makes this assumption explicit: if `countNumberOfUsers` does return a negative value, the program may have a bug.

A major advantage of this technique is that when an error does occur it is detected immediately and directly, rather than later through its often obscure side-effects. Since an assertion failure usually reports the code location, one can often pin-point the error without further debugging.

Assertions are also sometimes placed at points the execution is not supposed to reach. For example, assertions could be placed at the `default` clause of the `switch` statement in languages such as C, C++, and Java. Any case which the programmer does not handle intentionally will raise an error and the program will abort rather than silently continuing in an erroneous state. In D such an assertion is added automatically when a `switch` statement doesn't contain a `default` clause.

In Java, assertions have been a part of the language since version 1.4. Assertion failures result in raising an `AssertionError` when the program is run with the appropriate flags, without which the assert statements are ignored. In C, they are added on by the standard header `assert.h` defining `assert (assertion)` as a macro that signals an error in the case of failure, usually terminating the program. In standard C++ the header `cassert` is required instead. However, some C++ libraries still have the `assert.h` available.

The danger of assertions is that they may cause side effects either by changing memory data or by changing thread timing. Assertions should be implemented carefully so they cause no side effects on program code.

Assertion constructs in a language allow for easy test-driven development (TDD) without the use of a third-party library.

## Assertions during the development cycle

During the development cycle, the programmer will typically run the program with assertions enabled. When an assertion failure occurs, the programmer is immediately notified of the problem. Many assertion implementations will also halt the program's execution: this is useful, since if the program continued to run after an assertion violation occurred, it might corrupt its state and make the cause of the problem more difficult to locate. Using the information provided by the assertion failure (such as the location of the failure and perhaps a stack trace, or even the full program state if the environment supports core dumps or if the program is running in a debugger), the programmer can usually fix the problem. Thus assertions provide a very powerful tool in debugging.

## Static assertions

Assertions that are checked at compile time are called static assertions.

Static assertions are particularly useful in compile time template metaprogramming, but can also be used in low-level languages like C by introducing illegal code if (and only if) the assertion fails. For example, in C a static assertion can be implemented like this:

```
#define SASSERT(pred) switch(0){case 0:case pred:;}

SASSERT( BOOLEAN CONDITION );
```

If the `(BOOLEAN CONDITION)` part evaluates to false then the above code will not compile because the compiler will not allow two case labels with the same constant. The boolean expression must be a compile-time constant value, for example `(sizeof(int)==4)` would be a valid expression in that context. This construct does not work at file scope (i.e. not inside a function), and so it must be wrapped inside a function.

Another popular [2] way of implementing assertions in C is:

```
static char const static_assertion[ (BOOLEAN CONDITION)
                                    ? 1 : -1
                                  ] = {'!'};
```

If the `(BOOLEAN CONDITION)` part evaluates to false then the above code will not compile because arrays may not have a negative length. If in fact the compiler allows a negative length then the initialization byte (the `'!'` part) should cause even such over-lenient compilers to complain. The boolean expression must be a compile-time constant value, for example `(sizeof(int)==4)` would be a valid expression in that context.

Both of these methods require a method of constructing unique names. Modern compilers support a __COUNTER__ preprocessor define that facilitates the construction of unique names, by returning monotonically increasing numbers for each compilation unit.[3]

D provides static assertions through the use of `static assert`,[4] C11 and C++11 through `static_assert`.

## Disabling assertions

Most languages allow assertions to be enabled or disabled globally, and sometimes independently. Assertions are often enabled during development and disabled during final testing and on release to the customer. Not checking assertions avoids the cost of evaluating the assertions while (assuming the assertions are free of side effects) still producing the same result under normal conditions. Under abnormal conditions, disabling assertion checking can mean that a program that would have aborted will continue to run. This is sometimes preferable.

Some languages, including C and C++, completely remove assertions at compile time using the preprocessor. Java requires an option to be passed to the run-time engine in order to enable assertions. Absent the option, assertions are bypassed, but they always remain in the code unless optimised away by a JIT compiler at run-time or excluded by an if(false) condition at compile time, thus they need not have a run-time space or time cost in Java either.

Programmers can build checks into their code that are always active by bypassing or manipulating the language's normal assertion-checking mechanisms.

## Comparison with error handling

Assertions are distinct from routine error-handling. Assertions document logically impossible situations and discover programming errors: if the impossible occurs, then something fundamental is clearly wrong. This is distinct from error handling: most error conditions are possible, although some may be extremely unlikely to occur in practice. Using assertions as a general-purpose error handling mechanism is unwise: assertions do not allow for recovery from errors; an assertion failure will normally halt the program's execution abruptly. Assertions also do not display a user-friendly error message.

Consider the following example of using an assertion to handle an error:

```
int *ptr = malloc(sizeof(int) * 10);
assert(ptr);
// use ptr
...
```

Here, the programmer is aware that `malloc` will return a `NULL` pointer if memory is not allocated. This is possible: the operating system does not guarantee that every call to `malloc` will succeed. If an out of memory error occurs the program will immediately abort. Without the assertion, the program would continue running until `ptr` was dereferenced, and possibly longer, depending on the specific hardware being used. So long as assertions are not disabled, an immediate exit is assured. But if a graceful failure is desired, the program has to handle the failure. For example, a server may have multiple clients, or may hold resources that will not be released cleanly, or it may have uncommitted changes to write to a datastore. In such cases it is better to fail a single transaction than to abort abruptly.

Another error is to rely on side effects of expressions used as arguments of an assertion. One should always keep in mind that assertions might not be executed at all, since their sole purpose is to verify that a condition which should always be true does in fact hold true. Consequently, if the program is considered to be error-free and released, assertions may be disabled and will no longer be evaluated.

Consider another version of the previous example:

```
int *ptr;
// Statement below fails if malloc() returns NULL,
// but is not executed at all when compiling with -NDEBUG!
assert(ptr = malloc(sizeof(int) * 10));
// use ptr: ptr isn't initialised when compiling with -NDEBUG!
...
```

This might look like a smart way to assign the return value of `malloc` to `ptr` and check if it is `NULL` in one step, but the `malloc` call and the assignment to `ptr` is a side effect of evaluating the expression that forms the `assert` condition. When the `NDEBUG` parameter is passed to the compiler, as when the program is considered to be error-free and released, the `assert()` statement is removed, so `malloc()` isn't called, rendering `ptr` uninitialised. This could potentially result in a segmentation fault or similar null pointer error much further down the line in program execution, causing bugs that may besporadic and/or difficult to track down. Programmers sometimes use a similar VERIFY(X) define to alleviate this problem.

## History

The assertional method for proving correctness of programs was advocated by Alan Turing. In a talk "Checking a Large Routine" at Cambridge, June 24, 1949 Turing suggested: "How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite *assertions* which can be checked individually, and from which the correctness of the whole program easily follows".[5]

## References


1. **Jump up^** C.A.R. Hoare, An axiomatic basis for computer programming, *Communications of the ACM*, 1969.
2. **Jump up^** Jon Jagger, *Compile Time Assertions in C*, 1999.
3. **Jump up^** GNU, "GCC 4.3 Release Series — Changes, New Features, and Fixes"
4. **Jump up^** Static assertions in D*http://dlang.org/version.html#StaticAssert*
5. **Jump up^** Alan Turing, Checking a Large Routine, 1949; quoted in C.A.R Hoare, "The Emperor's Old Clothes", 1980 Turing Award lecture


## External links[edit]


- *Programming With Assertions in Java* by Qusay H. Mahmoud, (Oracle Corp.), 2005
- *A historical perspective on runtime assertion checking in software development* by Lori A. Clarke, David S. Rosenblum in: ACM SIGSOFT Software Engineering Notes 31(3):25-37, 2006
- *The benefits of programming with assertions* by Philip Guo (Stanford University), 2008
- *Assertions: a personal perspective* by C.A.R. Hoare in: Annals of the History of Computing, IEEE, Volume: 25 , Issue: 2 (2003), Page(s): 14 - 25
- *My Compiler Does Not Understand Me* by Poul-Henning Kamp in: ACM Queue 10(5), May 2012