Should there be assertions in release builds
リリースビルドにアサーションはあるべきか

Nobody

The default behavior of `assert` in C++ is to do nothing in release builds

I presume this is done for performance reasons and maybe to prevent users from seeing nasty error messages.

C++ における `assert` のデフォルトの動作は、リリースビルドでは何もしないというものだ。

これはパフォーマンスのためにこうなっているのだと思う。あるいはまた、ユーザに汚いエラーメッセージを見せないためかもしれない。

However, I'd argue that those situations where an `assert` would have fired but was disabled are even more troublesome beacuse the application will then probably crash in an even worse way down the line because some invariant was broken.

しかし思うに、`assert` が発出されたであろう状況においてそれが無効化されていることはむしろより危険ではないか。なぜなら、不変であるものらが壊れてしまい、もっと下の行でアプリケーションはクラッシュするだろうから。

Additinally, the performance argument for me only counts when it is a measurable problem.

Most `assert`s in my code aren't much more complex than

```
assert(ptr != nullptr);
```

which will have small impact on most code.

さらに、パフォーマンスが問題になるのはある程度の大きさがある場合だけだろう。

私のコードの `assert` のほとんどは次のようなもの以上ではないから、大きな影響は生じない。

This leads me to the question: Should assertions (meaning the concept, not the specific implementation) be active in release build?

Why (not)?

よって疑問が浮かんだ。(特定の実装ではなく概念としての) アサーションは、リリースビルドにおいて有効にされてあるべきか。

なぜ (そうではないの) か。

Please note that this question is not about how to enable asserts in release builds (like `#undef _NDEBUG` or using a self defined assert implemen-

この質問は、どうやってリリースビルドで assert を有効にできるかではないことに注意してください。(例えば、`#undef _NDEBUG` とか、自ら定義した

tation).

Furthermore, it is not about enabling asserts in third party/standard library code but in code controlled by me.

アサーションの実装を用いるとか。)

またこの質問は、自分でコントロールできるコードについてのものであり、サードパーティや標準ライブラリのアサーションを有効化したいのではない。

Bernhard Hiller

I know a global player in the health care market who installed a debug version of their hospital information system at customer sites.

They had a special agreement with Microsoft to install the C++ debug libraries there, too.

Well, the quality of their product was ...

ヘルスケア市場の世界的なプレイヤーがデバッグ版をインストールしていることを私は知っている。

Chiristian Hackl

There's an old saying that goes "*removing assertions is a bit like wearing a life-jacket to practice in the harbour, but then leaving the life-jackets behind when your ship leaves for open ocean*" ....

I personally keep them enabled in release biulds by default.

Unfortunately, this practice is not very common in the C++ world, but at least renowned C++ veteran James Kanze always used to argue in favour of keeping assertions in release builds: ...

「アサーションを外すというのは、浜辺で練習する際には救命胴衣を着ていて、海に船が出かけるときには救命胴衣を脱ぎ捨てていくものだ」という古いことわざがある。

私は個人的には、リリースビルドでもアサーションを有効にしておく。

残念だが、C++ の世界ではこの習慣は一般的だとは言えない。有名な C++ のベテランである James Kanze は、リリースビルドでアサーションを用いるべきことを主張している。

Doc Brown

The classic `assert` is a tool from the old standard C library, not from C++.

It is still available in C++, at least for reasons of backwards compatibility.

I have no precise timeline of the C standard libs at hand, but I am pretty sure `assert` was available after the time when K&R C came into live (around 1978).

In classic C, for writing robust programs, adding NULL pointer tests and array bounds checking needs to be done way more frequently than in

C++.

The gross of NULL pointer tests can be avoided by using references and/or smart ointers instead of pointers, and by using `std::vector`, array bounds checking is often unnecessary.

Moreover, the performance hit at 1980 was definitely much more important than today.

So I think that is very likely the reason why "assert" was designed to be active only in debug builds by default.

Moreover, for real error handling in production code, a function which just tests some condition or invariant, and crashes the program if the condition is not fulfilled, is in most cases not flexible enough.

For debugging that is probably ok, since the one who runs the program and observes the error has typically a debugger at hand to analyse what happens.

For production code, however, a sensible solution needs to be a function or mechanism which

- tests some condition (and stops execution at the scope where the condition fails)
- provides a clear error message in case the condition does not hold
- allow the outer scope to take the error message and outputs it to a specific communication channel. This channel might be something like stderr, a standard log file, a message box in a GUI program, a general error handling callback, a network-enabled error channel, or whatever fits best to the particular piece of software.
- allows the outer scope on a per-case base to decide if the program should end gracefully, or if it should continue.

...