Dual compiles of functions with contracts

Document #: P3967R0
Date: 2026-01-16
Project: Programming Language C++
Audience: EWGI /SG21
Reply-to: Bengt Gustafsson
<bengt.gustafsson@beamways.com>

Contents

1 Introduction

This proposal solves a number of problems related to contracts by generating code both with and without checked contract-assertions with one compilation command. We call these the checked and unchecked *compiles* and the result the checked and unchecked *code*. The checked compile works exactly like a compile with C++26 contracts, or as amended by other proposals. The unchecked compile works like a C++26 contracts build with *ignore* semantic, except for contract-assertions marked as *non-ignorable* by some mechanism applicable to individual assertions. The two compiles of a function are differentiated by their mangled names and are created by the same compilation command.

The checked compile of the `main` function is called by the startup code. To run any code with unchecked contract assertions a special `unchecked_contracts` block has to be executed. This is syntactically similar to an `extern "C"` or a `consteval` block and causes its contents to be compiled as in the unchecked compile, with function calls directed at unchecked compiles. A `checked_contracts` block is also available which works in the opposite way.

In addition to the two compiles of each function a thunk which jumps to either the checked or unchecked code depending on a *checking mode* variable is compiled. This thunk has the same mangled name as a compiler would emit for the function prior to this proposal. When taking the address of a function the address of this thunk results, and the address of the thunk is used in vtables. By virtue of its legacy name mangling the thunk will also be called by code compiled with an older compiler. The *checking mode* variable is `thread_local` and indicates the current (checked or unchecked) checking mode of the thread and is modified when entering and exiting `unchecked_contracts` and `checked_contracts` blocks.

1.1 Example

This example shows how `unchecked_contracts` and `checked_contracts` blocks can be nested. Note that each compile of `func` is called once from main, and that its first call to `subfunc` is to the corresponding compile. while the two subsequent calls go to the same compiles of `subfunc` for both compiles of `func`. This works the same even if the functions are in different translation units and/or inlined.

```
bool log(const char* msg)
{
    std::cout << msg << std::endl;
    return true;
}

void subfunc() pre(log("Running subfunc"))
{
}

void func() pre(log("Enter func") post(log("Leave func"))
{
    sub_func();
    unchecked_contracts {
        sub_func();
        checked_contracts {
            sub_func();
        }
    }
}

int main()
{
    func();
    unchecked_contracts {
        func();
    }
}
```

Running this program results in the following output:

```
Enter func          // First call of func is with contracts
Running subfunc     // First call to subfunc follows func's checking mode
Running subfunc     // Third call to subfunc unconditionally checks contracts
Leave func          // First call of func ends
Running subfunc     // When calling func in an unchecked block only the third subfunc call logs.
```

Sorry for the side effect of the contract-assertion predicate...

2 Motivation

With the TU-level evaluation semantic of C++26 contracts based on compiler switches performance critical code can't reside in the same TU as safety critical code. This forces unnatural subdivision of code such as `setup` and `run` functions in the same classes into different TUs just to get contract checks on the `setup` parts without getting unacceptable performance degradation in the `run` parts. Furthermore the lack of name mangling difference between functions with different evaluation semantics in C++26 allows the linker to randomly select implementations of outlined inline and template functions, which is a big problem for instance for the standard library.

With this proposal calls to inline and template functions that are not inlined always call the checked or unchecked compile depending on whether the calling code is checked or unchecked, which guarantees that both performance and checking is maintained according to programmer intent.

As unchecked code never executes unless called directly or indirectly from an `unchecked_contracts` block it is relatively easy to find out which parts of a program run are unchecked, and as easy to find out that there is no checking going on in performance critical parts of the code.

Optimizers can elide caller side checks as it is guaranteed that callee side checks are actually performed in checked compiles. With the additional ABI specific entry point suggested below the precondition checking can be moved to the caller side if this is more optimal.

With this proposal it is pointless to compile translation units with the *ignore* semantic as the same code is generated by the unchecked compile anyway. It is expected that the *observe* semantic will most often be set on a specific subset of contract-assertions, so the overall evaluation semantic when compiling will be *enforce* or *quick-enforce*. This minimizes the risk that the checked code of a function actually doesn't check the contract assertions if selected randomly by the linker.

The thunk with the traditionally mangled name makes sure that checking is consistent even when calling virtual functions on objects created in code with opposite checking mode, when calling through function pointers obtained by code with the opposite checking mode and when calling back from code compiled without knowledge of this proposal. Furthermore the address-equality of all function pointers to the same function is preserved, as the address is always that of the thunk.

This proposal also solves the problem of distributing pre-compiled libraries as the same library file contains both the checked and unchecked compiles. This includes the specific problem of linking to a hardened or unhardened standard library as there is only one. Selection of which compile to use is done separately for different parts of the codebase using `unchecked_contracts` and `checked_contracts` blocks even when these call the same function.

3 Proposal

The actual proposal consists of the checked and unchecked compiles of functions, the checking mode variable and the `unchecked_contracts` and `checked_contracts` blocks. Any possibilities beyond this is related to compiler switches etc. and are beyond the scope of the C++ standard. However, it is suggested that compilers should *not* provide a way to compile all contract-assertions in a translation unit as *ignore* or *observe*.

As name mangling is not something that the standard talks about some other nomenclature may have to be introduced, such wording details are not part of this first proposal revision.

3.1 Checked and unchecked compiles

This proposal introduces two *checking modes*: checked and unchecked. Each function in a program is compiled in both these checking modes. Function calls from checked code call checked code of called functions and function calls from unchecked code call unchecked code of called functions. In unchecked code all contract assertions except those especially marked as non-ignorable are ignored. In checked code all contract assertions are checked according to whatever system the compiler implements to control the semantic of each contract assertion, including using in flight proposals such as [P3400R2] or [P3968R0].

The startup code calls the checked compile code of the `main` function.

Special thunks are generated during compilation, which just jump to the checked or unchecked code depending on a `thread_local` *checking mode* variable. The address of this thunk is returned when the address of a function is taken, including when building vtables.

Lambdas are not special in this regard, conceptually there are two compiles of the lambda body. When a captureless lambda is converted to a function pointer the thunk selecting between these two compiles is returned. This follows from the rewrite rules from lambdas to structs with function call operators.

Defining a local class or lambda inside a `unchecked_contracts` or `checked_contracts` block does not prevent the dual compiles of the lambda's function call operator or the generated constructors and destructor.

3.2 Unchecked and checked code blocks

Special `unchecked_contracts` and `checked_contracts` blocks are part of this proposal as shown in the introductory example. Function calls in `unchecked_contracts` blocks don't do caller side contract checks and call unchecked code in both compiles of the surrounding function. Similarly function calls in `checked_contracts` blocks call checked compile code in both compiles of the surrounding function and may do caller side contract checks.

`unchecked_contracts` or `checked_contracts` blocks are only allowed in block scope.

When entering an `unchecked_contracts` block in the checked code of a function the *checking mode* variable is set to unchecked, and when the block ends the variable is set back to checked. When entering a `checked_contracts` block in the unchecked code of a function the *checking mode* variable is set to checked, and when the block ends the variable is set back to unchecked. The checking mode variable for the main thread is initiated to the checked state before main is called. The checking mode variable of other std::threads is initiated from the initiating thread's value.

The braces of `unchecked_contracts` and `checked_contracts` blocks do not introduce a block scope which is to say that local variables are not destroyed at the end of the block. This is consistent with extern "C" blocks.

The checking mode variable is hidden from the programmer and only observed by the thunks.

4 Discussions

Here are some additional discussions and observations regarding the consequences and further details of this proposal.

4.1 Interfacing to code compiled without this proposal

To make sure that both checked and unchecked compiles can call functions compiled with an older C++ compiler an *alternate name* for the function to call using the traditionally mangled name for the function is added to each call site. Alternate names is an object file format feature available in the elf and coff file formats used on Linux, Windows and Mac, but it is possible that such a feature is not available on some less well-known platforms. Another solution may be to include an entry point with a weak symbol with each called unchecked or checked mangled name which just jumps to the traditionally mangled function. This should work on all platforms as weak symbols must be available to link C++ programs.

Functions declared with `extern "C"` don't have mangled names and thus both compiles will call the only existing implementation of the function. For an `extern "C"` function which is defined in a C++ translation unit both compiles are made but the only symbol exposed in the object file refers to the thunk.

4.2 Global variable initializers

All global and static variables are initialized by calling checked code. As initialization of global variables is usually not performance critical this proposal does not include a way to initialize global variables using the unchecked compile code. A future version of this proposal or a follow up proposal may make this possible if needed. One reason for not proposing `unchecked_contracts` blocks in namespace scope is that it is unclear what it would mean to declare or define functions or classes with member functions inside such blocks. Also the possibility of inlined class static variables complicates this considering that the header file containing the inline declaration of the variable may be included from within an `unchecked_contracts` block in some TUs, but not int others.

4.3 Drawbacks

No feature comes without drawbacks, but in this case these are mainly evident during builds.

4.3.1 Compiled code size

One obvious drawback of this proposal is that the generated code size is larger by something approaching a factor of 2 compared to a C++26 build. This is true for the object files and static library files but with function level linking this space premium decreases to a much smaller value, including two compiles only for functions that are actually called from both unchecked and checked code (and not inlined).

With advanced systems such as link time code generation and modules the actual dual code generation can be relegated to link time, when only the code generation that is actually needed in the executable is actually performed.

Note that this drawback can be seen as an advantage: It makes it possible to distribute libraries in binary form that are suitable for both performance critical and safety critical use.

4.3.2 Compile time

Without link time code generation the actual code generation step has to be done twice which affects the compile times. How much this adds depends on how much of the time is spent in the compiler back end generating object code compared to the front end tasks of parsing and semantic analysis that are still only done once. For inline functions that are not inlined as well as for functions with static linkage only compiles called from the compiled TU need to be performed, which alleviates the compile time overhead.

With link time code generation the code generation for regular functions including non-inline functions would be performed at link time. As link time code generation has traditionally been used with high optimization levels in mind the time to link may be high, but in principle the total time could be lower as inline functions are only code-generated once. In most programs only the checked *or* unchecked compile functions are ever called which may create a significant time saving compared to always generate both checked and unchecked code during compilation and then only link one of these compiles.

4.3.3 Runtime overhead

The reading of the *checking mode* variable introduces a small overhead for indirect calls via vtables and function pointers. Modifying this variable during execution of `unchecked_contracts` and `checked_contracts` blocks adds an even smaller overhead. This overhead can be made lower than accessing a regular `thread_local` variable as the checking mode variable can be part of runtime state directly accessible via the `thread_local` pointer register, removing one or two levels of indirection compared to regular `thread_local` variables. For both Intel and ARM architectures checking this variable then consists of loading one byte at an offset relative to the register used for keeping track of `thread_local` data.

The highest risk of measurable overhead is probably when calling virtual functions in hot loops but this adds rather little to the existing overhead, especially on advanced micro-architectures where branch predictors should be extremely good at predicting the jumps in the thunks where the branch condition seldom changes.

4.4 Extra symbols for optimization

So far this proposal has discussed three different entry points for each function, the unchecked compile, the checked compile and the thunk that selects between the two. Checked code calls the checked entry point but this often causes unnecessary contract checks as the caller may be able to statically determine that preconditions are not broken, or at least that within a loop the contract is either broken on every turn or never. In these cases it would be advantageous to move the remaining precondition checks to the caller side. As evaluating preconditions is the first thing that happens in a function it should be possible to provide an entry point after this part of the function. Depending on the ABI and implementation of contract checks a function preamble and a jump instruction may be required to implement this entry point. If the function has only preconditions this entry point is the same as the entry point for the unchecked compile. In an actual implementation the jump instruction would be more efficiently placed at the end of the precondition checking code and jump to after the unchecked entry point's preamble.

This is an optimization that ABIs can implement, but it does not seem it has to be mentioned in the standard as long as the allowance for multiple checking of contracts is retained (along with the allowance to omit checks that can be proven true under the as-if rule).

4.5 Debug builds and always checked contracts

In debug builds it may be desirable to check all contracts regardless of the current checking mode. The easiest way to do this would be to have a compiler flag that disables generation of unchecked code and makes `unchecked_contracts` ineffective. It is also possible to leave this choice to runtime, i.e. to be able to run a built executable with all contracts checked or according to its `unchecked_contracts` blocks. In principle this consists of blocking the setting of the *checking mode* to unchecked by some process wide setting. For this to work all calls in `unchecked_contracts` blocks must call the checking mode selection thunks which carries a small overhead even when this runtime checking enable feature is not used. This overhead should be negligible in debug builds.

4.6 Possible compiler controllability of dual code generation

While the intent of this proposal is that a regular build should do both compiles it is certainly possible to reduce the overhead with knowledge about the use of the code. A compiler switch can be provided that only performs one of the compiles (and sets the thunk to point at this compile). This results in missing symbol errors at link time if calls are made to the non-existing code for a function.

Such compiler switches can be used to ensure that unchecked code of critical parts of a program can never be called, as it doesn't exist if suppressed by a switch. Another use case is to remove the small overhead that is introduced by the thunks, but most really time critical code would not pass through thunks anyway as it is well known that virtual function calls and calls through function pointers are expensive, and those are the only situations that thunks would be executed.

As noted above providing a switch to only generate unchecked code presents a risk of accidentally calling unchecked code. This risk can be mitigated by *not* providing the traditionally mangled function name in this case, which means that to call such code you have to use a `unchecked_contracts` block even with an unchecked-only compilation.

The main drawback with this type of switches is that it brings back the subdivision of enforcement at translation unit borders, which is what this proposal wants to avoid.

4.6.1 Functions without contracts

A function without contracts may or may not call other functions which have contracts. As there may be `contract_assert` type contracts in a called function without observable source code it is not always possible to know if there are contracts in called functions. If it can be determined that there aren't, both compiles of the function will generate the same code so the differently mangled symbols for the function may refer to the same compile (including the thunk name).

If it can't be determined that there are no contracts in called functions the compiler can choose to compile for both checking modes as if the function had contracts. An alternative which reduces code size but has a small performance penalty is to compile functions without contracts with all outgoing calls calling the traditionally mangled function name, thus ending up in thunks when calling functions with contracts. In this case all three mangled names for the function refer to the same compile. This is to say that these compiles are exactly like compiling with an earlier compiler version.

As there is a space/performance trade off a compiler switch and/or attribute on the function itself is advised. This proposal revision does not standardize any *attribute* to control this for individual functions, but it would be possible to add.

4.6.2 Dual entry vtables

It would be possible to get rid of the overhead of calling virtual functions via thunks by doubling the number of function pointers in vtables. This is however an ABI break so it would require a compiler switch. To prevent the linker from linking object modules with traditional and dual entry vtables together a flag in the object file format can be used, but this requires the linker to know about this flag which can be problematic on some platforms.

When loading dynamic libraries the loader would have to check a flag in the executable format to prevent loading libraries linked with or without dual entry vtables. While possible this does not seem worth the effort and risks to end up with a non-starting program in the field for such a miniscule time saving.

4.7 Interaction with P3400 or P3968

With [P3400R2] it becomes possible to individually control the evaluation semantic of each contract-assertion, creating an *effective evaluation semantic* that may differ from the *initial evaluation semantic* from the build system. If the effective evaluation semantic is something else than *ignore* for unchecked compiles (where the initial semantic is *ignore*) the contract-assertion is checked even in the unchecked compiles.

With the alternative [P3968R0] proposal this is instead indicated by the `ignorable` data member of the *assertion object* being false.

4.8 Interaction with library hardening

With this proposal in place it is obvious that library hardening should be based on contracts. This allows using the standard library in programs that contain both performance critical code that can't bounds check for instance every `vector` element access and safety critical code which requires such bounds checking, even for vectors of the same element type or even the same vector.

Presumably library hardening only affects functions defined in headers but some of these may not actually be inlined which makes them susceptible to random selection by the linker if some TUs are compiled without library hardening. This proposal solves that issue and also makes the idea of being able to turn off library hardening less important. However, due to the vast amount of performance critical code relying on the performance of the standard library out there we probably need to be able to turn off library hardening using a compiler define for the near future even though modifying code bases by adding `unchecked_contracts` blocks and turning on library hardening should be encouraged.

Note also that with this proposal there is no problem with *distributing* pre-compiled libraries, including the standard library, as the same library file contains both the checked and unchecked code. For library users there are no awkward choices to make when linking and no risk of selecting the wrong library file.

Implementing library hardening without using contracts makes standard library checking independent from checked/unchecked code which would be bad in all cases. Also, much more preconditions could be checked in the standard library with this proposal in place as there is no performance vs. safety trade-off, you can get both!

4.9 Interaction with implicit contract checks

The proposal [P3100R5] discusses checks for out of bounds accesses to local arrays and other checks which relate to the core language functionality. Such checks are called implicit contract checks and with this proposal they would be performed in the checked compilations only, which is consistent with how all other checks are handled. As the built in operators for which such implicit contract checks would be done are almost always inlined there is no need for name mangling or thunks. For some smaller architectures things like division may be delegated to functions in a support library and then the same system as for all other functions can be used, checking for such things as division by zero in checked compiles only.

5 References

[P3100R5] Timur Doumler, Joshua Berne. 2025-12-15. Implicit contract assertions.
https://wg21.link/p3100r5

[P3400R2] Joshua Berne. 2025-12-14. Controlling Contract-Assertion Properties.
https://wg21.link/p3400r2