Improving safety of C++26 contracts.

Document #:	P3910R0
Date:	2025-11-02
Project:	Programming Language C++
Audience:	SG21, EWG

Reply-to:	Bengt Gustafsson
	<bengt.gustafsson@beamways.com>

Contents

1 Improving safety of C++26 contracts.

The C++26 standard is set to contain a new *contracts* facility. Much work has been devoted to this feature which is in line with the growing demand for making C++ more safe. Still a lot of objections are being raised against standardizing contracts in its current form. One major objection is that as written the standard offers no means to guarantee that all contracts are checked when mulltiple TUs are linked together. This problem is especially hard when libraries delivered in object form are part of the link. This paper tries to find ways to provide such guarantees without touching the standard text.

What this paper suggests are purely implementation strategies that allow users better control of the evaluation semantics (**ES**) of their contracts. The corner stone of these strategies is to include the evaluation semantic into the name mangling of function names. To get this to work requires changes to the linker and compiler. Changes to the linker are not to be taken lightly but I think that the fears of affecting the linker have been detrimental to the language several times, and this is another of those situations.

There are other objections to the contracts facility, related to constification, exceptions etc. These are not addressed by this paper.

1.1 What are the benefits of name mangling based on ES

The obvious result of making the ES part of the name mangling is that code for functions compiled with different ES can co-exist in the same executable. This immediately guarantees that inline functions that are not actually inlined will have the same ES as if they had actually been inlined. This is as each TU compiles the inline functions with its own ES and thus the corresponding mangled name. When the linker then randomly selects one of these compiled functions it uses the mangled name and will select one with the ES of the call site.

This requires no changes to the linker, but has the consequence that mixed ES builds are impossible! The mangled name will mismatch when calling a non-inline function between TUs compiled with different ESes, causing linker errors. For some use cases this guarantee is a good thing but for many it is unacceptable. In most cases the link will contain some libraries that are not compiled with name mangling based on ES, for instance OS provided libraries and C libraries.

1.2 Giving the linker ES name mangling knowledge

To enable linking between TUs compiled with different ESes the linker must have knowledge about the subset of the name mangling scheme related to ES.

When the linker fails to find the function using the ES requested by the object module it must be able to select another ES to link to instead. I propose that the linker gets a prioritized list of allowed ESes as a command line option and looks for the symbol in this order if the ES requested is not available. If there is no match the link fails. This can be used to for instance prevent any code with ignore semantics to be linked in, while observe or enforce are both allowed. A linker should provide a reasonable default which may be: *quick_enforce*, *enforce*, *observe*, *ignore*, *no_contracts*, where *no_contracts* is the same as pre-C++26 code but also used by new compilers for functions without contracts.

With this linker switch and fairly simple logic to handle it in the linker we can guarantee that no functions with contracts compiled with an excluded ES are in the executable.

There is however an issue lurking in the shadows. With this system the address of an inline function may differ when taken in TUs compiled with different ESes. This is not allowed by the standard, the address of a function must be the same throughout the executable, even if it is an inline function.

1.3 The undefined ES

To preserve the address equality for inline functions compilers can generate a mangled name with an extra, *undefined*, ES name mangling. This means that the linker will never find a function with this ES as none are compiled. Thus the linker will follow its list of allowed ESes and select the first inline function address found. As this is done by the linker the address of a certain function will be the same in all TUs, preserving the address equality guarantee.

This operation comes with a cost: The ES of an inline function whose address is taken may not be that of the compilation of the code where the address is taken. This re-opens a window where it is not known which semantic is actually used, but with the default priority list described above the semantic will at least be the same or stricter than the one of the code where the address is taken.

As many (most?) code bases don't care about address equality for function pointers a compiler switch could be implemented which instead of the *undefined* ES uses the ES of the current compilation when the address of a function is taking. This reinstates the ES guarantee. Note however that some uses of address equality may be hidden in libraries. For instance if the program relies on replacing a callback pointer with itself being quick performance degradation may result from foregoing the address equality guarantee.

1.4 One step beyond: Multiple compilations

The measures described above are enough to be able to enforce that inline functions evaluate the contracts in the same way whether they are actually inlined or not, and that only code of certain ESes is linked in to an executable. This resolves two main objections to the P2900 approach to contracts.

However, with the name mangling of non-inline functions in place it is not that far fetched to compile a source code file for multiple ESes and link all of the object modules. This is very useful for libraries that are to be distributed to users which may want different ESes. Instead of shipping separate libraries for each ES with the risk of mixing them up (all contain the same set of symbols) you can now ship one library with all ESes, ready to be linked. This is similar to the *fat binary* option available when compiling CUDA code. Fat binaries include pre-compiled object code for multiple GPU generations, while regular binaries just contain intermediate code which is jit-compiled for the GPU at hand on each system.

There are two problems with linking multiple ESes. One is that if there are static or global variables defined in the source code being compiled more than once the linking will result in multiple defined symbols. Static and global variables as well as their initializer code can't be name mangled with ES as we can't have one instance of a global variable per ES in a mixed ES build or initialize it more than once. Similarly `extern "C"`functions and the function `main` can only exist once with the regular name mangling (*no_contracts*) even if they have contracts. Thus compiling a source code file with such functions in more than one ES invariably causes linker errors.

The other problem is that as `main` can only be compiled once causes that *the entire executable* will run with the same ES as `main` was compiled with. This renders the idea of compiling the same source code file with multiple ESes worthless. A way to call a function compiled with another ES is clearly needed.

1.5 Preventing multiply defined symbols

One way to prevent multiply defined symbols for static and global variables, their initialization code, `extern "C"` functions and `main` when linking TUs compiled with multiple ESes is to provide a compiler switch to turn off code generation for these entities. This however makes configuring the build system unnecessarily complicated and compile times long.

A more robust solution would be to implement a compiler option that lists the ESes to compile all functions in the source code for, except that the code for static and global variable initializers, `extern "C"` functions and `main` is only emitted for the first ES in the list. This reduces the compile time overhead for the multiple ESes as the front end compilation is only done once. It can also reduce code bloat at least for functions without `contract_assert` statements by calling the ignore implementation from the checked implementation(s). For functions without contracts the name can use the *no_contracts* mangling and only one compilation be made. This means that for TUs without contracts the result is the same as today.

Initializers for `inline` global variables still pose a problem but this can be solved by the same system as inline functions, using the *undefined* ES combined with emitting code for the initializer mangled with each of the ESes in the list of the compilation. The `.init_array` section (Linux) or `.CRT$XCU` section (Windows) contains pointers to the *undefined* ES which forces the linker to select in its prioritized ES list as when taking the address of a function.

Another possibility is to tag initializer code for *all* static and global variables with the *undefined* ES name mangling and let the linker decide which ES of the static initializer code to link in. this has the advantage that all initializers have the same ES and especially for libraries shipped in object form it would be an advantage to let the library user select the ES also for initializers. Both of these options could be made available.

1.6 Specifying ES when using functions

As described above the `main` function ES will control the ES of all functions in the executable unless there is some way to change the ES when calling a function. Standardizing such a feature will be a task for a future standard iteration. For the time being I will use a strawman syntax similar to a cast of the function name:

```
// Extern function compiled in one or more ESes in another TU.
void myFunction();

void callMF() {
    // Call the function with the ES used to compile callMF
    myFunction();
    // Call the enforce compiled myFunction in all compilations of callMF.
    evaluation_semantic_cast<enforce>(myFunction)();

    // Call any myFunction compilation that the linker can find
    evaluation_semantic_cast<undefined>(myFunction)();

    // Set ptr to the highest priority ES compilation of myFunction
```

```
    // the linker can find.
    void(*ptr)() = myFunction;

    // Set ptrl to the ES of each callMF compilation.
    void(*ptrl)() = evaluation_semantic_cast(myFunction);

    // Set ptro to point to the observe compiled myFunction.
    void(*ptro)() = evaluation_semantic_cast<observe>(myFunction);
}
```

Here I additionally suggest that `evaluation_semantic_cast` without a template argument list defaults to the ES of the code being compiled, offering a way to opt out of the address equality requirement locally, in preference of guaranteeing that the ES is preserved between the code being compiled and the code pointed to.

Note that the defaults for direct calls and for taking function addresses are opposite: For direct calls the same ES as is being compiled is the default while, to preserve address equality, the Linker's top priority function is selected. Providing the `evaluation_semantic_cast<undefined>` possibility for the direct call case is maybe not necessary but makes sure all possiibilities exist.

1.7 Cosntructors, virtual functions and vtables

Virtual functions are compiled just like any function. For inline virtual functions the address is still taken and put in the vtable. Constructors, which are responsible for setting up the vtable pointer in the constructed objects, can be compiled with one or more ESes just like any functions. The question is whether the vtable pointer should point to the same vtable for all objects or if it should point to a vtable with pointers to functions compiled with the same ES as the constructor.

To make sure that the vtable pointer is set up equally in all objects the *undefined* name mangling is used when referring to the vtable in the constructor code, while a weak symbol for the vtable of the ES being compiled is provided in the object module. In this way all objects will use the most prioritized virtual function compilations selected by the linker.

To let vtable pointers be set up according to the ES of the constructor compilation is just a matter of instead referring to the vtable using the same name mangling as the constructor is being compiled for.

It is not clear which of these strategies is *best*, regardless of which one is chosen an object may potentially be passed between code compiled with different ESes and thus not match with the calling code. It will not be possible to allow using `evaluation_semantic_cast` on virtual function calls as the actual implementation for a certain ES for the subclass at hand is not known. The same limitation goes for taking the address of a virtual member function and specifying a specific ES.

Note that apart from being awkward a system where the vtable is extended to contain pointers to different ES compilations of the same virtual function is an ABI break as soon as code compiled with a older compiler is called.

1.7.1 Optimization opportunities

If the compiler can select which outlined implementation of an inline function to call when not actually inlining the function it will often be possible to call the implementation with ignore semantic as the compiler may be able to deduce that some contracts are redundant (such as checking the index towards the size each turn in a loop). As the compiler has always seen the source code for inline functions it can make this decision based on whether the function contains any `contract_assert` statements that have to be executed (preventing calls to the ignore semantic function).

For non-inline functions the compiler is not able to see the source code but can instead remove caller side checking as it can now rest assured that if a function is called it will actually perform the checks if the name mangling

for that semantic is used in the call. A problem with this idea is that this may cause linker errors if the called function was only compiled without checks.

1.8 Exploring LTCG and Modules

With link time code generation or modules it would be possible to only compile to intermediate representation in the compile step and then compile to machine code at link time. This removes a lot of compiles of functions that are never called with a certain ES and removes possibilities for linker errors. However, for tool chains that don't support these technologies the standard should not add features that require them as it is a massive undertaking to implement.

For tool chains supporting any of these technologies it however seems like the first choice for implementation strategy. The semantics of the program will be the same, down to which ES is selected in each case, the only difference is that compilation of machine code for certain ESes is on demand.

1.9 Exploring weak symbols and trampolines

Is it possible to use the existing linker features *weak symbols* and *alternate names* together with forwarding so called trampoline functions to implement the functionality here without altering linkers?

To some extent yes: It seems perfectly possible to use weak symbols make sure that global variables defined in a TU that is compiled for multiple ESes do not cause multiply defined symbol errors. The same goes for their initializers, but with lost control over which ES the initializers actually run with.

`extern "C"` functions and `main` can always have their names mangled using the *no_contracts* name mangling without affecting linking.

However I can't see a way to ensure address equality when taking the address of functions without altering linkers. This is as the *alternate name* features in both Windows and Linux only accepts one fallback name, while with four ESes and *no_contracts* the function address to use may be one of up to five different mangled function names.

Likewise, handling vtables is problematic if we want to guarantee that all objects have the same ES for their virtual functions.

1.10 Why having both *ignore* and *no_contracts*

The reason for differentiating between these is that often you want to prevent *ignore* when linking an executable for a safety critical application, while functions compiled before C++26 must be possible to link to. As for functions which don't have contracts the ES makes no difference anyway.

1.11 Real world use case

In the company where I am employed we produce software to do image enhancement for medical Ultrasound and X-Ray equipment. This requires doing compute heavy operations for each pixel in the images at rates of several megapixels per second, often on constrained hardware. The code that does the image processing is heavily vectorized with SIMD and multithreaded over the processor cores. Adding any form of contract checks would be devastating to this performance.

However there is a significant part of the software, more than half of the code lines, that is not time critical but has to do with setup and interfacing to customer code, file handling, license management etc.

The source code is organized by Operation, which is a smallish part of the entire image processing data flow. In each TU there is code both for the setup, management and running of the image processing for a certain Operation. This means that to be able to use the TU based granularity of evaluation semantic selection we would need to reorganize our code so that all setup and auxiliary functionality is located in different files from the performance critical processing code. This is not only a big refactorization job, it also reduces code quality significantly by forcing unnatural subdivision of code that belongs together into different files. The possibilities

for making errors in this subdivision or the setting up of the compiler switches for each file causing either poor performance or missed checks. The refactorization effort itself is a deterrent both budget-wise and risk-wise.

I don't think this is a particularly unusual use case, so for the next standard iteration I think we need to prioritize some other way of specifying parts of call-graphs that should have different evaluation semantics, and to do this we must start with at least being able to compile and link different implementations of non-inline functions into the same program as described above. Adding a way to select which ES of a function to call as exemplified by `evluation_semantic_cast` above provides a way to explicitly escape the fully checked parts of the code when the time critical computation is to be run.

1.12 Summary

It seems that these types of relatively small changes to compilers and linkers could reduce the anxiety that contracts are not checked when they should. It addresses both out of line functions in object modules and inline functions that are not inlined at some call sites. With some linker support the address equality guarantee can be preserved.

To allow different part of the program call graph to have different semantics will require extensions beyond C++26, which can be prototyped as implementation specific extensions, but eventually need to be standardized for portability.