# Preprocessor embed - Binary Resource Inclusion

JeanHeyd Meneide <phdofthehouse@gmail.com>

October 31st, 2020



***Document***: WG14 n2592 | WG21 p1967r3
***Previous Revisions***: WG14 n2470 | WG21 p1967r0, p1967r1, p1967r2
***Audience***: WG14, WG21
***Proposal Category***: New Features
***Target Audience***: General Developers, Application Developers, Compiler/Tooling Developers
***Latest Revision***: https://thephd.github.io/vendor/future_cxx/papers/source/C - embed.html

***Abstract:***

Pulling binary data into a program often involves external tools and build system coordination. Many programs need binary data such as images, encoded text, icons and other data in a specific format. Current state of the art for working with such static data in C includes creating files which contain solely string literals, directly invoking the linker to create data blobs to access through carefully named extern variables, or generating large brace-delimited lists of integers to place into arrays. As binary data has grown larger, these approaches have begun to have drawbacks and issues scaling. From parsing 5 megabytes worth of integer literal expressions into AST nodes to arbitrary string literal length limits in compilers, portably putting binary data in a C program has become an arduous task that taxes build infrastructure and compilation memory and time.

This proposal provides a flexible preprocessor directive for making this data available to the user in a straightforward manner.

# 1 Changelog

## 1.1 Revision 3 - October 25th, 2020

— Added post C++ meeting notes and discussion.
— Removed type or bit specifications from the `#embed` directive.
— Moved “Type Flexibility” section and related notes to the Appendix as they are now unpursued.

## 1.2 Revision 2 - April 10th, 2020

— Added post C meeting notes and discussion.
— Added discussion of potential endianness.

## 1.3 Revision 1 - March 5th, 2020

— Improved wording section at the end to be more detailed in handling preprocessor (which does not understand types).

## 1.4 Revision 0 - January 5th, 2020

— Initial release.

# 2 Polls & Votes

The votes for the C++ Committee are as follows:

— SF: Strongly in Favor
— F: In Favor
— N: Neutral
— A: Against
— SA: Strongly Against

The votes for the C Committee are as follows:

— Y: Ye
— N: Nay
— A: Abstain

## 2.1 September 2020 Virtual C++ EWG Meeting

“We want `#embed [optional limit] header-name` (no type name, no other specification) as a feature.”

| SF | F | N | A | SA |
|---|---|---|---|---|
| 2 | 16 | 3 | 0 | 1 |

This vote gained the most consensus in the Committee. While there were some

individuals who wanted to be able to specify a type, there was stronger interest in not specifying a type at all and always producing a list of integer literals suitable to be used anywhere an `initializer-list` was valid.

"We want to explore allowing an optional sequence of tokens to specify a type to `#embed`."

| SF | F | N | A | SA |
|---|---|---|---|---|
| 1 | 9 | 4 | 4 | 3 |

Further need was also expressed for `constexpr` of different types of variables, so we would rather focus that ability into a sister feature, `std::embed`. There was also an expression to augment `std::bitcast< ... >( ... )` to handle arrays of data, which would be a follow-on proposal. There was a great amount of interest in the `std::bitcast` direction, which means a paper should be written to follow up on it.

## 2.2 April 2020 Virtual C Meeting

"We want to have a proper preprocessor `#embed ...` over a `#pragma _STDC embed ...` -based directive."

This had UNANIMOUS CONSENT to pursue a proper preprocessor directive and NOT use the `#pragma` syntax. It is noted that the author deems this to be the best decision!

"We want to specify embed as using `#embed [bits-per-element] header-name` rather than `#embed [pp-tokens-for-type] header-name`." (2-way poll.)

| Y | N | A |
|---|---|---|
| 10 | 2 | 3 |

— Y: 10 bits-per-element (Ye)
— N: 2 type-based (Nay)
— A: 4 Abstain (Abstain)

This poll will be a bit harder to accommodate properly. Using a *`constant-expression`* that produces a numeric constant means that the max-length specifier is now ambiguous. The syntax of the directive may need to change to accommodate further exploration.

# 3 Introduction

For well over 40 years, people have been trying to plant data into executables for varying reasons. Whether it is to provide a base image with which to flash hardware in a hard reset, icons that get packaged with an application, or scripts that are intrinsically tied to the program at compilation time, there has always been a strong need to couple and ship binary data with an application.

C does not make this easy for users to do, resulting in many individuals reaching for utilities such as `xxd`, writing python scripts, or engaging in highly platform-specific

linker calls to set up `extern` variables pointing at their data. Each of these approaches come with benefits and drawbacks. For example, while working with the linker directly allows injection of vary large amounts of data (5 MB and upwards), it does not allow accessing that data at any other point except runtime. Conversely, doing all of these things portably across systems and additionally maintaining the dependencies of all these resources and files in build systems both like and unlike `make` is a tedious task.

Thusly, we propose a new preprocessor directive whose sole purpose is to be `#include`, but for binary data: `#embed`.

## 3.1 Motivation

The reason this needs a new language feature is simple: current source-level encodings of "producing binary" to the compiler are incredibly inefficient both ergonomically and mechanically. Creating a brace-delimited list of numerics in C comes with baggage in the form of how numbers and lists are formatted. C's preprocessor and the forcing of tokenization also forces an unavoidable cost to lexer and parser handling of values.

Therefore, using arrays with specific initialized values of any significant size becomes borderline impossible. One would think this old problem would be work-around-able in a succinct manner. Given how old this desire is (that comp.std.c thread is not even the oldest recorded feature request), proper solutions would have arisen. Unfortunately, that could not be farther from the truth. Even the compilers themselves suffer build time and memory usage degradation, as contributors to the LLVM compiler ran the gamut of the biggest problems that motivate this proposal in a matter of a week or two earlier this very year. Luke is not alone in his frustrations: developers all over suffer from the inability to include binary in their program quickly and perform exceptional gymnastics to get around the compiler's inability to handle these cases.

C developer progress is impeded regarding the inability to handle this use case, and it leaves both old and new programmers wanting.

## 3.2 But *How* Expensive Is This?

Many different options as opposed to this proposal were seriously evaluated. Implementations were attempted in at least 2 production-use compilers, and more in private. To give an idea of usage and size, here are results for various compilers on a machine with the following specification:

— Intel Core i7 @ 2.60 GHz
— 24.0 GB RAM
— Debian Sid or Windows 10
— Method: Execute command hundreds of times, stare extremely hard at `htop`/Task Manager

While `time` and `Measure-Command` work well for getting accurate timing information and can be run several times in a loop to produce a good average value, tracking memory consumption without intrusive efforts was much harder and thusly relied on OS reporting with fixed-interval probes. Memory usage is therefore approximate and

may not represent the actual maximum of consumed memory. All of these are using the latest compiler built from source if available, or the latest technology preview if available. Optimizations at `-O2` (GCC & Clang style)/`/O2 /Ob2` (MSVC style) or equivalent were employed to generate the final executable.

### 3.2.1 Speed Size

| Strategy | 40 kilobytes | 400 kilobytes | 4 megabytes | 40 megabytes |
|---|---|---|---|---|
| `#embed` GCC | 0.236 s | 0.231 s | 0.300 s | 1.069 s |
| `xxd`-generated GCC | 0.406 s | 2.135 s | 23.567 s | 225.290 s |
| `xxd`-generated Clang | 0.366 s | 1.063 s | 8.309 s | 83.250 s |
| `xxd`-generated MSVC | 0.552 s | 3.806 s | 52.397 s | Out of Memory |

### 3.2.2 Memory Size

| Strategy | 40 kilobytes | 400 kilobytes | 4 megabytes | 40 megabytes |
|---|---|---|---|---|
| `#embed` GCC | 17.26 MB | 17.96 MB | 53.42 MB | 341.72 MB |
| `xxd`-generated GCC | 24.85 MB | 134.34 MB | 1,347.00 MB | 12,622.00 MB |
| `xxd`-generated Clang | 41.83 MB | 103.76 MB | 718.00 MB | 7,116.00 MB |
| `xxd`-generated MSVC | ~48.60 MB | ~477.30 MB | ~5,280.00 MB | Out of Memory |

### 3.2.3 Analysis

The numbers here are not reassuring that compiler developers can reduce the memory and compilation time burdens with regard to large initializer lists. Furthermore, privately owned compilers and other static analysis tools perform almost exponentially worse here, taking vastly more memory and thrashing CPUs to 100% for several minutes (to sometimes several hours if e.g. the Swap is engaged due to lack of main memory). Every compiler must always consume a certain amount of memory in a relationship directly linear to the number of tokens produced. After that, it is largely implementation-dependent what happens to the data.

The GNU Compiler Collection (GCC) uses a tree representation and has many places where it spawns extra “garbage”, as its called in the various bug reports and work items from implementers. There has been a 16+ year effort on the part of GCC to reduce its memory usage and speed up initializers (C Bug Report and C++ Bug Report). Significant improvements have been made and there is plenty of room for GCC to improve here with respect to compiler and memory size. Somewhat unfortunately, one of the current changes in flight for GCC is the removal of all location information beyond the 256th initializer of large arrays in order to save on space. This technique is not viable for static analysis compilers that promise to recreate source code exactly as was written, and therefore discarding location or token information for large initializers is not a viable cross-implementation strategy.

LLVM’s Clang, on the other hand, is much more optimized. They maintain a much better scaling and ratio but still suffer the pain of their token overhead and Abstract Syntax Tree representation, though to a much lesser degree than GCC. A bug report

was filed but talk from two prominent LLVM/Clang developers made it clear that optimizing things any further would require an extremely large refactor and functionality add of parser internals, with potentially dubious gains. As part of this proposal, the implementation provided does attempt to do some of these optimizations, and follows some of the work done in this post to try and prove memory and file size savings. (The savings in trying to optimize parsing large array literals were "around 10%", compared to the order-of-magnitude gains from `#embed` and similar techniques).

Microsoft Visual C (MSVC) scales the worst of all the compilers, even when given the benefit of being on its native operating system. Both Clang and GCC outperform MSVC on Windows 10 or WINE as of the time of writing.

Linker tricks on all platforms perform better with time (though slower than `#embed` implementation), but force the data to be optimizer-opaque (even on the most aggressive "Link Time Optimization" or "Whole Program Optimization" modes compilers had). Linker tricks are also exceptionally non-portable: whether it is the `incbin` assembly command supported by certain compilers, specific invocations of `rc.exe`/`objcopy` or others, non-portability plagues their usefulness in writing Cross-Platform C (see Appendix for listing of techniques). This makes C decidedly unlike the "portable assembler" advertised by its proponents (and my Professors and co-workers).

# 4 Design

There are two design goals at play here, sculpted to specifically cover industry standard practices with build systems and C programs. The first is to enable developers to get binary content quickly and easily into their applications. This can be icons/images, scripts, tiny sound effects, hardcoded firmware binaries, and more. In order to support this use case, this feature was designed for simplicity and builds upon widespread existing practice.

## 4.1 First Principle: Simplicity and Familiarity

Providing a directive that mirrors `#include` makes it natural and easy to understand and use this new directive. It accepts both chevron-delimited (`<>`) and quote-delimited (`""`) strings like `#include` does. This matches the way people have been generating files to `#include` in their programs, libraries and applications: matching the semantics here preserves the same mental model. This makes it easy to teach and use, since it follows the same principles:

```
#include <limits.h>

/* default is unsigned char */
const unsigned char icon_display_data[] = {
    #embed "art.png"
};

/* specify a type-name to change array type */
```

```
const char reset_blob[] = {
    #embed "data.bin"
};
```

Because of its design, it also lends itself to being usable in a wide variety of contexts and with a wide variety of vendor extensions. For example:

```
#include <limits.h>

/* attributes work just as well */
const signed char aligned_data_str[] __attribute__ ((aligned (8))) =
        {
    #embed "attributes.xml"
};
```

The above code obeys the alignment requirements for an implementation that understands GCC directives, without needing to add special support in the `#embed` directive for it: it is just another array initializer, like everything else.

### 4.1.1 Existing Practice - Search Paths

It follows the same implementation experience guidelines as `#include` by leaving the search paths implementation defined, with the understanding that implementations are not monsters and will generally provide `-fembed-path`/`-fembed-path=` and other related flags as their users require for their systems. This gives implementers the space they need to serve the needs of their constituency.

### 4.1.2 Existing Practice - Discoverable and Distributable

Build systems today understand the make dependency format, typically through use of the compiler flags `-(M)MD` and friends. This sees widespread support, from CMake, Meson and Bazel to ninja and make. Even VC++ has a version of this flag – `/showIncludes` – that gets parsed by build systems.

This preprocessor directive fits perfectly into existing build architecture by being discoverable in the same way with the same tooling formats. It also blends perfectly with existing distributed build systems which preprocess their files with `-frewrite-includes` before sending it up to the build farm, as `distcc` and `icecc` do.

## 4.2 Second Principle: Efficiency

The second principle guiding the design of this feature is facing the increasing problems with `#include` and typical source-style rewriting of binary data. Array literals do not scale. Processing large comma-delimited, *braced-init-lists* of data-as-numbers produces excessive compilation times. Compiler memory usage reaches extraordinary levels that are often ten to twenty times (or more) of the original desired data file (see above tables in the Motivation section). Part of this is endemic to the compiler: the preprocessor demands that tokens be

String literals do not suffer the same compilation times or memory scaling issues, but the C Standard has limits on the maximum size of string literals (§5.2.4.1, "— 4095 characters in a string literal (after concatenation)"). One implementation takes the C Standard quite almost exactly at face value: it allows 4095 bytes in a single string *piece*, so multiple quoted pieces each no larger than 4095 bytes must be used to create large enough string literals to handle the work.

`#embed`'s specification is such that it behaves "as if" it expands to an *initializer-list*, comma-separated sequence of integral literals. This means an implementation does not have to run the full gamut of producing an abstract syntax tree of an expression. Nor does it need to generate a token sequence that must be manipulated by the preprocessor either. Most compilers do not need a fully generic expression list that spans several AST nodes for what is logically just a sequence of numeric literals. A more direct representation can be used internally in the compiler, drastically speeding up processing and embedding of the binary data into the translation unit for use by the program. One of the test implementations uses such a direct representation and achieves drastically reduced memory and compile time footprint, making large binary data accessible in C programs in an affordable manner.

### 4.2.1 Infinity Files

The earliest adopters and testers of the implementation reported problems when trying to access POSIX-style `char` devices and pseudo-files that do not have a logical limitation. These "infinity files" served as the motivation for introducing the "limit" parameter; there are a number of resources which are logically infinite and thusly having a compiler read all of the data would result an Out of Memory error, much like with `#include` if someone did `#include "/dev/urandom"`.

The limit parameter is specified before the resource name in `#embed`, like so:

```
const int please_dont_oom_kill_me[] = {
    #embed 32 "/dev/urandom"
};
```

This prevents locking compilers in an infinite loop of reading from potentially limitless resources. Note the parameter is a hard upper bound, and not an exact requirement. A resource may expand to 16 elements, which is fine, and not the maximum of 32.

## 5 Implementation Experience

An implementation of this functionality is available in branches of both GCC and Clang, accessible right now with an internet connection through the online utility Compiler Explorer. The Clang compiler with this functionality is called "x86-64 clang (std::embed)" and the GCC compiler is called "x86-64 gcc (std::embed)" in the Compiler Explorer UI.

# 6 Alternative Syntax

There were previous concerns about the syntax. WG14 voted to keep the syntax as a plain `#embed` preprocessor directive.

# 7 Wording - C

This wording is relative to C's latest working draft.

## 7.1 Intent

The intent of the wording is to provide a preprocessing directive that:

— takes a string literal identifier – potentially from the expansion of a macro or expression that produces a macro – and uses it to find a unique resource in some implementation-specific manner;
— it produces a list of integral constant values suitable for an *initializer-list*, where each element of that list is bound by the width and size of `unsigned char` as found in `<limits.h>`;
— errors if the size of the resource does not have enough bits to fully and properly initialize all the values generated by the directive;
— allows a limit parameter limiting the number of elements to be specified;
— and, present such contents as if by a list of values, such that it can be used to initialize arrays of known and unknown bound even if additional elements of the whole initialization list come before or after the directive.

## 7.2 Proposed Language Wording

Note: The ❑ is a stand-in character to be replaced by the editor.

**Add another *control-line* production and a new *parenthesized-non-header-digits* to §6.10 Preprocessing Directives, Syntax, paragraph 1:**

> *control-line:*
> ...
> **# embed** *pp-tokens new-line*

**Add a new sub clause as §6.10.❑ to §6.10 Preprocessing Directives, preferably after §6.10.2 Source file inclusion:**

> ### §6.10.❑ Resource embedding
>
> **Constraints**
>
> [1]A **`#embed`** directive shall identify a resource that can be processed by the implementation as a binary data sequence, of an optionally specified type, that results in a part of or a whole of an *initializer-list*.
>
> **Semantics**

[2] A preprocessing directive of the form

**# embed** *constant-expression*$_{opt}$ **<** *h-char-sequence* **>***new-line*

searches a sequence of implementation-defined places for a resource identified uniquely by the specified sequence between the < and >. The named resource is searched for in an implementation-defined manner.

[3] A preprocessing directive of the form

**# embed** *constant-expression*$_{opt}$ **"** *q-char-sequence* **"** *new-line*

searches a sequence of implementation-defined places for a resource identified uniquely by the specified sequence between the ", or < and >, delimiters. The named resource is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

**# embed** *constant-expression*$_{opt}$ **<** *h-char-sequence* **>** *new-line*

with the identical contained *q-char-sequence* (including > characters, if any) from the original directive.

[4] If either form of the **`#embed`** directive specified is not preceded by an **`#include`** directive for **`<limits.h>`**, then an implementation may issue a diagnostic.

[5] Either form of the **`#embed`** directive specified previously, for the following token sequence of **`unsigned char`**, behave as if it were expanded to an *initializer-list*. Specifically, each element of the *initializer-list* behaves as if the characters from the resource were read and mapped in an implementation-defined manner to a sequence of bits. The sequence of bits is used to produce an *integer constant expression* of exactly `UCHAR_WIDTH` bits and must have a value between `0` and `UCHAR_MAX`, inclusive.

[6] If a *constant-expression* is specified, it shall result in an integral constant and be suitable for use in an **`#if`** preprocessing directive. The mapping from the contents of the resource to the elements of the *initializer-list* shall contain up to *constant-expression* elements according to the above. The implementation shall issue a diagnostic if the implementation-defined bit size is not a multiple of the `UCHAR_WIDTH`; and, the implementation-defined bit size is less than *constant-expression* * `UCHAR_WIDTH`. The program is well-formed if the implementation-defined bit size is greater than or equal to *constant-expression* * `UCHAR_WIDTH`.

[7] If a *constant-expression* is not specified, the implementation shall issue a diagnostic if the implementation-defined bit size is not a multiple of the `UCHAR_WIDTH`.

[8] If the resulting *initializer-list* is used in a place where a constant

expression (6.6) is valid, then the *initializer-list* must be a constant expression. If the resulting *initializer-list* is used as part of the initialization of an array of incomplete type, then the *initializer-list* will contribute to the size of the completed array type at the end of the initializer (6.7.9).

[9] A preprocessing directive of the form

**# embed** *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **embed** in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms[18]. The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single resource name preprocessing token is implementation-defined.

**Add 4 new Example paragraphs below the above text in §6.10.2 Resource embedding:**

[10] **EXAMPLE 1** Placing a small image resource.

```
#include <stddef.h>
#include <limits.h>

void have_you_any_wool(const unsigned char*,
        size_t);

int main (int, char*[]) {
    const unsigned char baa_baa[] = {
#embed "black_sheep.ico"
    };

    have_you_any_wool(baa_baa,
        sizeof(baa_baa) / sizeof(*baa_baa));

    return 0;
}
```

[11] **EXAMPLE 2** Checking the first 4 elements of a sound resource.

```
#include <assert.h>
#include <limits.h>

int main (int, char*[]) {
    const char sound_signature[] = {
#embed 4 <sdk/jump.wav>
    };
```

```
    // verify PCM WAV resource
    assert(sound_signature[0] == 'R');
    assert(sound_signature[1] == 'I');
    assert(sound_signature[2] == 'F');
    assert(sound_signature[3] == 'F');
    assert((sizeof(sound_signature) /
        sizeof(*sound_signature)) == 4);

    return 0;
}
```

[12] **EXAMPLE 3** Diagnostic for resource which is too small.

```
#include <limits.h>

int main (int, char*[]) {
    const unsigned char coefficients[] = {
#embed unsigned char "only_3_bits.bin"
    };

    return 0;
}
```

An implementation must produce a diagnostic where 3 bits (i.e., the implementation-defined bit size) is less than `UCHAR_WIDTH`, or the implementation-defined bit size modulo `UCHAR_WIDTH` is not 0. [13] **EXAMPLE 4** Extra elements added to array initializer.

```
#include <limits.h>

#include <string.h>

#ifndef SHADER_TARGET
#define SHADER_TARGET "phong.glsl"
#endif

extern char* null_term_shader_data;

void fill_in_data () {
    const char internal_data[] = {
#embed SHADER_TARGET
    , 0 };

    strcpy(null_term_shader_data, internal_data);
}
```

---

18) Note that adjacent string literals are not concatenated into a single

string literal (see the translation phases in 5.1.1.2); thus, an expansion that results in two string literals is an invalid directive.

**Forward references:** macro replacement (6.10.).

# 8 Wording - C++

This wording is relative to C++'s latest working draft.

## 8.1 Intent

The intent of the wording is to provide a preprocessing directive that:

— takes a string literal identifier – potentially from the expansion of a macro – and uses it to find a unique resource in an implementation-defined manner;
— behaves as if it produces a list of values suitable for the initialization of an array as well as initializes each `unsigned char` element according to the specific environment limits found in `<climits>`;
— errors if the size of the resource does not have enough bits to fully and properly initialize all the values generated by the directive;
— allows a limit parameter limiting the number of elements to be specified (but allowing less than the limit);
— produces a core constant expression that can be used to initialize `constexpr` arrays;
— and, present such contents as if it is a list of values, such that it can be used to initialize arrays of known and unknown bound even if additional elements of the whole initialization list come before or after the directive.

## 8.2 Proposed Feature Test Macro

The proposed feature test macro is `__cpp_pp_embed` for the preprocessor functionality, and `__cpp_pp_embed_classes` for the extended functionality.

## 8.3 Proposed Language Wording

Append to §14.8.1 Predefined macro names [**cpp.predefined**]'s **Table 16** with two additional entries:

| Macro name | Value |
|---|---|
| __cpp_pp_embed | ???? |
| __cpp_pp_embed_classes | ???? |

Add a new *control-line* production to §15.1 Preamble [**cpp.pre**] and a new grammar production:

*control-line*:
...
`#__embed` *pp-tokens new-line*

Add a new sub-clause §15.4 Resource inclusion [**cpp.res**]:

**15.4 Resource inclusion [cpp.res]**

[1] A `#embed` directive shall identify a resource file that can be processed by

the implementation.

[2] A preprocessing directive of the form

`#` **`embed`** *constant-expression*$_{opt}$ `<` *h-char-sequence* `>` *new-line*

or

`#` **`embed`** *constant-expression*$_{opt}$ `"` *q-char-sequence* `"` *new-line*

searches a sequence of implementation-defined places for a resource identified uniquely by the specified sequence between the `<` and `>` or the `"` and `"` delimiters. How the places are specified or the resource identified is implementation-defined.

[3] If either form of the **`#embed`** directive specified is not preceded by an **`#include`** directive for **`<climits>`**, then an implementation may issue a diagnostic.

[4] Either form of the **`#embed`** directive specified previously behave as if it were expanded to an *initializer-list*. Specifically, each element of the *initializer-list* behaves as if the characters from the resource were read and mapped in an implementation-defined manner to a sequence of bits. The sequence of bits is used to produce an *integer constant expression* of exactly `UCHAR_WIDTH` bits and must have a value between `0` and `UCHAR_MAX`, inclusive.

[5] If a *constant-expression* is specified, it shall result in an integral constant and be suitable for use in an **`#if`** preprocessing directive. The mapping from the contents of the resource to the elements of the *initializer-list* shall contain up to *constant-expression* elements according to the above. If the implementation-defined bit size is not a multiple of the `UCHAR_WIDTH`; and, the implementation-defined bit size is less than *constant-expression* multiplied by `UCHAR_MAX`, then the program is ill-formed. The program is well-formed if the implementation-defined bit size is greater than or equal to *constant-expression* multiplied by `UCHAR_WIDTH`.

[6] If a *constant-expression* is not specified, the implementation shall issue a diagnostic if the implementation-defined bit size is not a multiple of the `UCHAR_WIDTH`.

[ Example:

```
#include <cstddef>
#include <climits>

void have_you_any_wool(const unsigned char*, std::size_t);

int main (int, char*[]) {
  constexpr const unsigned char baa_baa[] = {
```

```
#embed "black_sheep.ico"
  };

  have_you_any_wool(baa_baa,
      sizeof(baa_baa) / sizeof(*baa_baa));

  return 0;
}
```

– end Example ]

[ Example:

```
#include <cassert>

int main (int, char*[]) {
  constexpr const char sound_signature[] = {
#embed 4 <sdk/jump.wav>
  };

  // verify PCM WAV resource
  assert(sound_signature[0] == 'R');
  assert(sound_signature[1] == 'I');
  assert(sound_signature[2] == 'F');
  assert(sound_signature[3] == 'F');
  assert((sizeof(sound_signature) / sizeof(*sound_signature))
         == 4);

  return 0;
}
```

– end Example ]

[ Example:

```
#include <climits>

int main (int, char*[]) {
  const unsigned char coefficients[] = {
// may produce diagnostic: 16 bits (i.e., implementation-
         defined bit size)
// is not enough for an unsigned long long (e.g.
         ULLONG_WIDTH)
#embed "only_3_bits.bin"
  };

  const unsigned char byte_factors[] = {
// may produce diagnostic: 12 bits % UCHAR_WIDTH may not be 0
```

```
// on a system where the resource with an implementation-
        defined
// bit size of 12 bits
#embed "12_bits.bin"
  };

  const unsigned char scalar[] = {
// if the bit size of unsigned char is less than
// an implementation-defined bit size of 24,
// then this does not produce a diagnostic.
#embed unsigned char 1 "24_bits.bin"
  };

  return 0;
}
```

– end Example ]

[ Example:

```
#include <algorithm>
#include <iterator>

#ifndef SHADER_TARGET
#define SHADER_TARGET "phong.glsl"
#endif

extern char* null_term_shader_data;

void init_data () {
  constexpr const char internal_data[] = {
#embed SHADER_TARGET
  , 0 }; // additional element to null terminate content

  std::copy_n(internal_data, std::size(internal_data),
      null_term_shader_data);
}
```

– end Example ]

# 9 Acknowledgements

Thank you to Alex Gilding for bolstering this proposal with additional ideas and motivation. Thank you to Aaron Ballman, David Keaton, and Rhajan Bhakta for early feedback on this proposal. Thank you to the #include<C++> for bouncing lots of ideas off the idea in their Discord.

Thank you to the Lounge<C++> for their continued support, and to Robot M. F. for the valuable early implementation feedback.

# 10 Appendix

## 10.1 Existing Tools

This section categorizes some of the platform-specific techniques used to work with C++ and some of the challenges they face. Other techniques used include pre-processing data, link-time based tooling, and assembly-time runtime loading. They are detailed below, for a complete picture of today's landscape of options. They include both C and C++ options.

### 10.1.1 Pre-Processing Tools

1. Run the tool over the data (`xxd -i xxd_data.bin > xxd_data.h`) to obtain the generated file (`xxd_data.h`) and add a null terminator if necessary:

```
unsigned char xxd_data_bin[] = {
  0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x2c, 0x20, 0x57, 0x6f, 0x72, 0x6c,
        0x64,
  0x0a, 0x00
};
unsigned int xxd_data_bin_len = 13;
```

2. Compile `main.c`:

```
#include <stdlib.h>
#include <stdio.h>

// prefix as const,
// even if it generates some warnings in g++/clang++
const
#include "xxd_data.h"

#define SIZE_OF_ARRAY (arr) (sizeof(arr) / sizeof(*arr))

int main() {
    const char* data = reinterpret_cast<const char*>(xxd_data_bin);
    puts(data); // Hello, World!
    return 0;
}
```

Others still use python or other small scripting languages as part of their build process, outputting data in the exact C++ format that they require.

There are problems with the `xxd -i` or similar tool-based approach. Tokenization and Parsing data-as-source-code adds an enormous overhead to actually reading and making that data available.

Binary data as C(++) arrays provide the overhead of having to comma-delimit every single byte present, it also requires that the compiler verify every entry in that array is

a valid literal or entry according to the C++ language.

This scales poorly with larger files, and build times suffer for any non-trivial binary file, especially when it scales into Megabytes in size (e.g., firmware and similar).

### 10.1.2 python

Other companies are forced to create their own ad-hoc tools to embed data and files into their C++ code. MongoDB uses a custom python script, just to format their data for compiler consumption:

```
import os
import sys

def jsToHeader(target, source):
    outFile = target
    h = [
        '#include "mongo/base/string_data.h"',
        '#include "mongo/scripting/engine.h"',
        'namespace mongo {',
        'namespace JSFiles{',
    ]
    def lineToChars(s):
        return ','.join(str(ord(c)) for c in (s.rstrip() + '\n')) +
         ','
    for s in source:
        filename = str(s)
        objname = os.path.split(filename)[1].split('.')[0]
        stringname = '_jscode_raw_' + objname

        h.append('constexpr char ' + stringname + "[] = {")

        with open(filename, 'r') as f:
            for line in f:
                h.append(lineToChars(line))

        h.append("0};")
        # symbols aren't exported w/o this
        h.append('extern const JSFile %s;' % objname)
        h.append('const JSFile %s = { "%s", StringData(%s, sizeof(%s)
        - 1) };' %
                 (objname, filename.replace('\\', '/'), stringname,
         stringname))

    h.append("} // namespace JSFiles")
    h.append("} // namespace mongo")
    h.append("")
```

```
    text = '\n'.join(h)

    with open(outFile, 'wb') as out:
        try:
            out.write(text)
        finally:
            out.close()


if __name__ == "__main__":
    if len(sys.argv) < 3:
        print "Must specify [target] [source] "
        sys.exit(1)
    jsToHeader(sys.argv[1], sys.argv[2:])
```

MongoDB were brave enough to share their code with me and make public the things they have to do: other companies have shared many similar concerns, but do not have the same bravery. We thank MongoDB for sharing.

### 10.1.3 ld

A complete example (does not compile on Visual C++):

1. Have a file ld_data.bin with the contents `Hello, World!`.
2. Run `ld -r binary -o ld_data.o ld_data.bin`.
3. Compile the following `main.cpp` with `c++ -std=c++17 ld_data.o main.cpp`:

```
#include <stdlib.h>
#include <stdio.h>

#define STRINGIZE_(x) #x
#define STRINGIZE(x) STRINGIZE_(x)

#ifdef __APPLE__
#include <mach-o/getsect.h>

#define DECLARE_LD_(LNAME) extern const unsigned char
        _section$__DATA__##LNAME[];
#define LD_NAME_(LNAME) _section$__DATA__##LNAME
#define LD_SIZE_(LNAME) (getsectbyLNAME("__DATA", "__"
        STRINGIZE(LNAME))->size)
#define DECLARE_LD(LNAME) DECLARE_LD_(LNAME)
#define LD_NAME(LNAME) LD_NAME_(LNAME)
#define LD_SIZE(LNAME) LD_SIZE_(LNAME)

#elif (defined __MINGW32__) /* mingw */
```

```
#define DECLARE_LD(LNAME)                                       \
  extern const unsigned char binary_##LNAME##_start[];    \
  extern const unsigned char binary_##LNAME##_end[];
#define LD_NAME(LNAME) binary_##LNAME##_start
#define LD_SIZE(LNAME) ((binary_##LNAME##_end) -
        (binary_##LNAME##_start))
#define DECLARE_LD(LNAME) DECLARE_LD_(LNAME)
#define LD_NAME(LNAME) LD_NAME_(LNAME)
#define LD_SIZE(LNAME) LD_SIZE_(LNAME)

#else /* gnu/linux ld */

#define DECLARE_LD_(LNAME)                                       \
  extern const unsigned char _binary_##LNAME##_start[];     \
  extern const unsigned char _binary_##LNAME##_end[];
#define LD_NAME_(LNAME) _binary_##LNAME##_start
#define LD_SIZE_(LNAME) ((_binary_##LNAME##_end) -
        (_binary_##LNAME##_start))
#define DECLARE_LD(LNAME) DECLARE_LD_(LNAME)
#define LD_NAME(LNAME) LD_NAME_(LNAME)
#define LD_SIZE(LNAME) LD_SIZE_(LNAME)
#endif

DECLARE_LD(ld_data_bin);

int main() {
    const char* p_data = reinterpret_cast<const char*>
        (LD_NAME(ld_data_bin));
    // impossible, not null-terminated
    //puts(p_data);
    // must copy instead
    return 0;
}
```

This scales a little bit better in terms of raw compilation time but is shockingly OS, vendor and platform specific in ways that novice developers would not be able to handle fully. The macros are required to erase differences, lest subtle differences in name will destroy one's ability to use these macros effectively. We omitted the code for handling VC++ resource files because it is excessively verbose than what is present here.

N.B.: Because these declarations are `extern`, the values in the array cannot be accessed at compilation/translation-time.

### 10.1.4 incbin

There is a tool called incbin which is a 3rd party attempt at pulling files in at "assembly time". Its approach is incredibly similar to `ld`, with the caveat that files must

be shipped with their binary. It unfortunately falls prey to the same problems of cross-platform woes when dealing with Visual C, requiring additional pre-processing to work out in full.

## 10.2 Type Flexibility

**Note:** As per the vote in the September C++ Evolution Working Group Meeting, Type Flexibility is not being pursued in the preprocessor for various implementation and support splitting concerns.

A type can be specified after the `#embed` to view the data in a very specific manner. This allows data to initialized as exactly that type.

Type flexibility was not pursued for various implementation concerns. Chief among them was single-purpose preprocessors that did not have access to frontend information. This meant it was very hard to make a system that was both preprocessor conformant but did not require e.g. `sizeof( ... )` information at the point of preprocessor invocation. Therefore, the type flexibility feature was pulled from `#embed` and will be conglomerated in other additions such as `std::bitcast` or `std::embed`.

```
#include <limits.h>

/* specify a type-name to change array type */
const int shorten_flac[] = {
    #embed int "stripped_music.flac"
};
```

The contents of the resource are mapped in an implementation-defined manner to the data, such that it will use `sizeof(type-name) * CHAR_BIT` bits for each element. If the file does not have enough bits to fill out a multiple of `sizeof(type-name) * CHAR_BIT` bits, then a diagnostic is required. Furthermore, we require that the type passed to `#embed` that must one of the following fundamental types, signed or unsigned, spelled exactly in this manner:

- `char`, `unsigned char`, `signed char`
- `short`, `unsigned short`, `signed short`
- `int`, `unsigned int`, `signed int`
- `long`, `unsigned long`, `signed long`
- `long long`, `unsigned long long`, `signed long long`

More types can be supported by the implementation if the implementation so chooses (both the GCC and Clang prototypes described below support more than this). The reason exactly these types are required is because these are the only types for which there is a suitable way to obtain their size at pre-processor time. Quoting from §5.2.4.2.1, paragraph 1:

> The values given below shall be replaced by constant expressions suitable for use in `#if` preprocessing directives.

This means that the types above have a specific size that can be properly initialized by

a preprocessor entirely independent of a proper C frontend, without needing to know more than how to be a preprocessor. This does require that every use of `#embed` is accompanied by a `#include <limits.h>` (or, in the case of C++, `#include <climits>`).

### 10.2.1 Endianness

> what would happen if you did fread into an int? that's my answer 🙂
> – The Cursed Bruja of the Great Sages, Isabella Muerte

**Note: Endianness is not an issue for single bytes. This section is kept for historical reasons.**

It's a simple answer. A compiler-magic based implementation like the ones provided below have no endianness issues, but an implementation which writes out integer literals may need to be careful of host vs. target endianness to make sure it serializes correctly to the final binary. As a litmus test, the following code – given a suitable sized `"foo.bin"` resource – must pass:

```c
#include <limit.h>
#include <string.h>
#include <assert.h>

int main() {
    const unsigned char foo0[] = {
#embed "foo.bin"
    };
    const int foo1[] = {
#embed int "foo.bin"
    };
    const unsigned int foo2[] = {
#embed unsigned int "foo.bin"
    };
    // additionally, if the implementation supports
    // extended types (see the "Type Flexibility" section)
    const float foo3[] = {
#embed float "foo.bin"
    };

    assert(memcmp(&foo0[0], &foo1[0], sizeof(foo0)) == 0);
    assert(memcmp(&foo1[0], &foo2[0], sizeof(foo0)) == 0);
    assert(memcmp(&foo0[0], &foo2[0], sizeof(foo0)) == 0);
    // additionally, if the implementation supports
    // extended types (see the "Type Flexibility" section)
    assert(memcmp(&foo0[0], &foo3[0], sizeof(foo0)) == 0);

    return 0;
}
```

This implies that the bits are always overlaid into the elements properly. Note that this does not imply the following must pass:

```
#include <limit.h>
#include <string.h>
#include <assert.h>

int main() {
    const unsigned char foo0[] = {
#embed int "foo.bin"
    };
    const int foo1[] = {
#embed unsigned char "foo.bin"
    };
    const unsigned long long foo2[] = {
#embed int "foo.bin"
    };
    // additionally, if the implementation supports
    // extended types (see the "Type Flexibility" section)
    const int foo2[] = {
#embed float "foo.bin"
    };

    // NONE of these are guaranteed!
    assert(memcmp(&foo0[0], &foo1[0], sizeof(foo0)) == 0);
    assert(memcmp(&foo1[0], &foo2[0], sizeof(foo0)) == 0);
    assert(memcmp(&foo0[0], &foo2[0], sizeof(foo0)) == 0);
    // additionally, if the implementation supports
    // extended types (see the "Type Flexibility" section)
    assert(memcmp(&foo0[0], &foo3[0], sizeof(foo0)) == 0);

    return 0;
}
```

This may truncate elements (the initialization of `foo0`), put the integer value into a larger type (the initialization of `foo1`), underflow the target representation (the initialization of `foo2`), or simply fail to compile (the implementation-defined `foo2` can reject initializing `int`s from `float`s). We see this as okay: the user has deliberately not matched the type being initialized with the type being viewed with the directive.