Preprocessor embed - Binary Resource Inclusion

JeanHeyd Meneide <phdofthehouse@gmail.com>

January 13th, 20120

Document: WG14 n2470 | WG21 p1967
Previous Revisions: n/a
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 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`.

1.1 Motivation

The reason this needs a new language feature is simple: at present we currently poorly indicate the intent of the compiler to “produce”

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` or equivalent were employed to generate the final executable.

1.1.1 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

1.1.2 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

1.1.3 Analysis

The numbers here are not particularly reassuring. Furthermore, privately owned compilers and other static analysis tools perform almost exponentially poorly 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.

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.

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).

2 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.

2.1 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:

```
/* 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 char "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:

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

2.1.1 Type Flexibility

As hinted at in previous sections's code snippets, 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.

```
/* 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.

2.1.2 Existing Practice - Search Paths

It follows the same implementation experience guidelines as `#include` by leaving the search paths implementation defined, with the understand 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.

2.1.3 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.

2.2 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, *brace-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 a brace-delimited, 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. It does 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.

2.2.1 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 int 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 and not the maximum of 32.

3 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.

4 Alternative Syntax

There has been concerns expressed about the form of this feature – whether or not it could be a preprocessor directive itself, or a magical macro introduced in the language, or a special pragma. Each of these has their own specific syntax tradeoffs. The primary choice and the one advocated for is the syntax as shown above: a plain preprocessor directive analogous to `#include`. It is written as `#embed`, but other names (previously recommended by the Community) are `#include_bin`, `#include_binary`, or `#load_binary`.

The syntax can also be adjusted. A preprocessor directive is preferred because that allows it to be findable by the end of Preprocessor.

5 Wording - C

This wording is relative to C's N2454.

5.1 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 on the command line;
- — maps the contents of the file in an implementation-defined manner to a sequence of integer literals, each whose value is no greater than the maximum representable value of a single `unsigned char`;
- — and, present such contents as if by a brace-enclosed list of integer literals, such that it can be used to initialize arrays of known and unknown bound.

5.2 Proposed Language Wording

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

control-line:
  …
  # **embed** *pp-tokens new-line*

parenthesized-non-header:
  ($_{\text{opt}}$ *pp-tokens*)$_{\text{opt}}$

Add a new sub clause as §6.10.� (� is a stand-in character to be replaced by the editor) 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 sequence of binary data.

Semantics

2 A preprocessing directive of the form

# embed *parenthesized-non-header*$_{opt}$ *digit-sequence*$_{opt}$ **<** *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 *parenthesized-non-header*$_{opt}$ *digit-sequence*$_{opt}$ *" 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 *parenthesized-non-header*$_{opt}$ *digit-sequence*$_{opt}$ *< h-char-sequence > new-line*

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

4 If a *parenthesized-non-header* is not specified, then the directive behaves as if the tokens of the *parenthesized-non-header* are `unsigned char`. If a *parenthesized-non-header* is specified, outer parenthesis must be present if it contains one or more of ", < or >.

5 Let the *parenthesized-non-header* tokens be `T`. Either form of the **`#embed`** directive specified previously behave as if it is replaced by the contents of the resource a `{` and `}` delimited *initializer-list*. The *initializer-list* represents an implementation-defined mapping from the contents of the resource to the elements of the *initializer-list*.

6 If a *digit-sequence* is specified, it shall be an unsigned *integer-constant*. The implementation-defined mapping from the contents of the resource to the elements of the *initializer-list* shall produce no more than *digit-sequence* elements.

7 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 2 new Example paragraphs below the above text in §6.10.� Resource embedding:

8 **EXAMPLE 1** Placing a small image resource.

```
#include <stddef.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;
}
```

9 **EXAMPLE 2** Checking the first 4 elements of a sound resource.

```
#include <assert.h>

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

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

    return 0;
}
```

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.�).

6 Wording - C++

This wording is relative to C++'s N4835.

6.1 Intent

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

- — takes a string literal enclosed in <> or `""` – potentially from the expansion of a macro – and use it to find a unique resource on implementation-defined search paths;
- — maps the contents of the file in an implementation-defined manner to a sequence of `type-name` values;
- — produces a diagnostic if the contents do not have enough data to fill out the binary representation of `type-name` values;
- — and, present such contents as if by a brace-enclosed list of integer literals, such that it can be used to initialize arrays of known and unknown bound.

6.2 Proposed Feature Test Macro

The proposed feature test macro is `__cpp_pp_embed` for the preprocessor functionality.

6.3 Proposed Language Wording

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

Macro name	Value
__cpp_pp_embed	202006L

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

control-line:
    …
    `# embed` *pp-tokens new-line*

…

parenthesized-non-header:
    ($_{opt}$ *pp-tokens* $_{opt}$)

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` *parenthesized-non-header*$_{opt}$ *digit-sequence*$_{opt}$ < *h-char-sequence* > *new-line* or

`# embed` *parenthesized-non-header*$_{opt}$ *digit-sequence*$_{opt}$ " *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 there is no *parenthesized-non-header*, then the directive behaves as if the tokens of the *parenthesized-non-header* are `unsigned char`. If a *parenthesized-non-header* is specified, outer parenthesis must be present if the *pp-token*s contains one or more of `"`, `<` or `>`.

[4] An `#embed` directive behaves as-if replaced by the contents of the resource in a *brace-initializer-list*. The brace-delimited *initializer-list* represents an implementation-defined mapping from the contents of the resource to the elements of the *initializer-list*.

[5] If a *digit-sequence* is specified, it shall be an unsigned *integer-literal* and the *brace-initializer-list* will contain no more than *digit-sequence* elements.

[6] Let `T` be the *parenthesized-non-header* tokens. If the implemented-defined bit size of the resource's contents are not a multiple of `sizeof( T ) * CHAR_BIT` or `T` does not denote a trivial type, then the program is ill-formed.

7 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.

8 Appendix

8.1 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.

8.1.1 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).

8.1.2 `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.

8.1.3 `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.

8.1.4 `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.