Proposal of File Literals

Document No.: P0373R0
Project: Programming Language C++
Audience: Evolution Working Group
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2016-05-21

Summary

We propose a new kind of literal called a *file literal*. A file literal contains a source file path:

```
bF"myfile"
```

During translation, a copy of the specified source file (`myfile`) is taken, and its content becomes the value of the file literal. It's like a #include, but the included file contains compile-time information other than C++ code. Another way of looking at it is that it's like a string literal but the information is stored in a separate dedicated source file, rather than in-line in an enclosing source file.

Example

The content of datafile.txt:

```
The quick brown fox jumps over the lazy dog.
```

The content of main.cc:

```
#include <iostream>
int main() {
  std::cout << tF"datafile.txt";
}
```

The program outputs:

```
The quick brown fox jumps over the lazy dog.
```

without accessing the filesystem at run-time (the content of datafile.txt has been "compiled-in").

Motivation

There is a large set of use cases for “compiling-in” files into C++ programs. Consequently there is a widespread existing practice of non-standard techniques to achieve this. We propose file literals as a standardization of this existing practice, to realize the usual portability, teachability and performance benefits.

Use Cases For Files

A file contains some set of related information that is packaged together and encoded (through one or more logical layers) into binary. This is true of all files. For example:

- An English UTF-8 text file contains information that is encoded in a natural language (English in this case) as text (sequence of characters) and then that text is encoded to binary in a text encoding (UTF-8 in this case).
- A C++ source file is an “abstract semantic tree” that is encoded in C++ as text and then that text is encoded to binary in the source character encoding for some particular implementation.
- A PNG image file is a grid of pixels that is encoded to binary according to an image format (PNG in this case).

Files and filesystems are fundamental components of almost all computational systems and information systems, with use cases too numerous to mention.

Use Cases For Compiling-In Files

Files can either be compiled-in to programs by the implementation on the build system during translation, or they can be accessed at run-time on the target system.

For a given file that contains information needed by a program, if that information will not change for the programs lifecycle (the union of all executions of the program), and the file is available at compile-time on the build system, then it is a candidate for being compiled-in to the program.

With constexpr programming it is even possible to decode a compiled-in file during translation and then use or manipulate that information at compile-time. Such decoding/processing is paid for once during compilation of the program, as opposed to N times where N is the number of times the program is executed.

Some specific examples of such candidate files and use cases include:

- Code files in non-C++ programming languages, such as shader programs for OpenGL and DirectX - or scripts of various kinds.
- “Static” program configuration files in declarative formats like XML or JSON.
- Mock/fake input data for unit test programs.
- Natural language files such as string lists for translation.
- Other kinds of static resource files, such as images.

Existing Practice

There are many non-standard mechanisms used to compile-in files into C++ programs. They largely involve executing a transcoding tool as a separate “pre-build” step that converts the file to be compiled-in into a C++ source file. The generated C++ source file is then translated as usual in the build proper.

For simple public examples there are tools like XPM, incbin, "xxd -i". There are many more proprietary ones. There is a spectrum of solutions starting with these simple cases going up to the sophisticated “asset” management and installer systems (used in the games industry for example).

These non-standard mechanisms suffer a lot of problems. In particular they are difficult to use and configure. The build system needs to have novel configuration and then the name and format of the generated header file needs to be determined. This layer of indirection is unnecessary. A standard mechanism can perform the operation directly during translation in a simple portable way.

Design Goals

Given the previously mentioned spectrum of possible solutions ranging from simple dumb systems to intricate “full stack” resource management systems, we want to choose a reasonable balance to standardize.

As with all compile-time features of C++, it is always possible to develop an arbitrarily-complex alternate custom solution that exists outside the phases of translation of C++ and doesn’t impact the standard at all. This can be achieved through pre-build and installer systems.

It is also possible to do sophisticated things with constexpr programming and a library solution that builds upon a simple raw core language feature.

In light of these two points, our goal is not to try and compete with the entire spectrum of existing non-standard practices, and only offer reasonable coverage of the relatively simple and common use cases, at least for version 1 of this feature.

Design

We start with source file inclusion:

```
#include “Q”
```

The Q in the above is called a q-char-sequence. There is already a mapping between such paths and the source files of a C++ program. Developing some sort of alternative way of describing the paths of files to be compiled-in doesn’t seem significantly beneficial, especially given the benefit of reuse of this existing mechanism. Therefore we propose that the mapping between Q and the file in the source file system should be the same as for #include as it is for file literals. File literals should use the existing source file inclusion mechanism.

We choose the base prefix “F for File” to distinguish syntactically the new kind of literal from other literals:

```
F”Q”
```

Next, what should the value of the literal be? It should reflect the content of the file. There are a tremendous number of file formats. We can’t support them all, at least directly. We propose to support two families of the most primitive formats. More complex formats can then be decoded from them using higher level library features. The first is called binary format:

```
bF”Q”
```

Such a file literal is called a binary file literal. A binary file literal is an array of N const unsigned char, where N is the length of the file. It is an object similar to a string literal, except it is not null terminated (this is to avoid accidental use of the null terminator as an end delimiter, because in general binary files can contain embedded nulls), and is of type unsigned char, not char (this is to avoid accidental error-prone conversion to a C string “const char*”). The value is the same as what would be obtained if the file were copied to the target system and then fopened at run-time in “rb” mode and fgetc were called N times (converting the return value from int to unsigned char as like fputc):

Example:

The file named `foo` is 4 bytes long and contains the bytes 0x81, 0x82, 0x83 and 0x84.

Therefore:

```
constexpr auto f =  bF”foo”;
```

```
static_assert(is_same_v<const unsigned char[4], decltype(f)>);
static_assert(f[0] == 0x81);
static_assert(f[1] == 0x82);
static_assert(f[2] == 0x83);
static_assert(f[3] == 0x84);
```

As binary file literals are constant expressions like string literals, it is possible to do library-based constexpr decoding to support any file format.

In addition to these general-purpose binary file literals, we also propose the more specialized text file literals:

```
tF"Q"
u8F"Q"
uF"Q"
UF"Q"
LF"Q"
```

The input file of a text file literal is a text file, encoded in the same source character encoding as other source files. It is as if the input file is interpreted as the body of a raw string literal. A text file literal has either a prefix t or an encoding-prefix. If it has the prefix t, the type and value are the same as an ordinary raw string literal `R"(content)"` where content is the content of the input file. If it has an encoding-prefix E, the type and value are the same as ER"(content)".

In short, the prefix of a text file literal specifies the execution character encoding.

Wording Sketch

Add new section 16.10:

16.10 File literals [cpp.filelit]

file-encoding-prefix:
b
t
encoding-prefix

A token of the form:

file-literal:
file-encoding-prefix **F "** *q-char-sequence* **"**

is a *file literal*. [Note: A file literal holds the same content as the source file identified by the *q-char-sequence*. --end note]

A file literal **EF"Q"**, where **E** is the *file-encoding-prefix*, **Q** is the *q-char-sequence* - is defined as follows:

Let the file that would be opened by a preprocessing directive:

```
# include "Q"
```

be known as the file literal input file. The file literal input file shall exist, or the program is ill-formed.

If **E** is **b**, the type of the file literal shall be array of N const unsigned char. The value of the file literal shall be the same value that would be observed if the file literal input file was copied to the execution environment, opened with std::fopen in mode `"rb"`, and read completely and successfully with repeated calls to std::fgetc. N is the number of would-be calls to std::fgetc (excluding the final EOF). The value of the ith element of the file literal is the would-be return value of the ith call to fgetc converted to unsigned char.

Otherwise, the file literal shall be equivalent by definition to a raw string literal **ER"D(C)D"** where **C** is the content of the file literal input file interpreted as an included source file, and **D** is a valid *d-char-sequence* that does not appear in **C**.

FAQ

Why would you use a text file literal instead of a raw string literal?

Raw string literals have become a popular kind of string literal to use to define a literal sequence of multi-line text "in-line" in a C++ source file - because of their verbatim nature they don't need to be obfuscated by using preprocessor string literal concatenation and redundantly encoding newlines with "\n".

Sometimes it is useful instead to dedicate a whole source file to effectively hold the content of such a string literal. Organizing a codebase using a filesystem of small source files is accepted good practice as it is generally easier to navigate than scrolling through long source files. Further, having a dedicated source file for a string literal means that tools (for example language-aware editors) can be used on that source file that know about the language of the string literal, but don't know about C++.

Doesn’t using a syntax like bF”myfile” cause parsing problems for non-compiler tools that need to identify source file dependencies from #includes?

No. For a tool to accurately do dependency scanning, it already has to do near full tokenization and preprocessing:

```
#include MACRO_REPLACE_ME

auto x = R"(
#include "skip_me"
)";

#if some_expression_that_is_false
#include "skip_me_too"
#endif
```

For approximate dependency scanning, which is of dubious benefit anyway, it is easy to add the proposed file string literal token pattern to the scanner.

File literals are replaced during “preprocessing”, what is the output of a preprocess-only compilation?

The output of a “preprocess-only” translation is unspecified and has always been outside the scope of the standard. Because of this, the output of a preprocess-only translation of a file literal is likewise unspecified.

What about other source formats other than binary or text in source encoding?

Let us first notice that all kinds of source formats, execution formats and transcodings can be supported by using binary file literals in combination with a library feature:

```
template<size_t N> constexpr ExecutionFormat custom_transform(const
unsigned char input_file[N]) {
  // transform input_file into whatever object you want
  return result;
}

constexpr ExecutionFormat output = custom_transform(bF”myfile”);
```

Consequently, for simplicity, at least for version 1, we think the two source formats of binary and source encoded text are sufficient to have as built-in. In practice, the most common source encoding is UTF-8 and this matches the most common text format.

That said, our proposal is extensible by future extensions that would add more file-encoding-prefixes, and could include built-in support for other kinds of source formats, execution formats and transcodings.

What about user-defined file literals?

There are some minor design complexities added by providing user-defined **binary** file literals, due in part to the use of the size of the array rather than null-termination. Therefore we have decided to leave user-defined file literals as a possible future extension, and to see if there is demand for them. The proposed version 1 feature is extensible to user-defined file literals.

References

https://groups.google.com/a/isocpp.org/forum/#!msg/std-proposals/b6ncBojU8wI/blx1GlLqUAUJ

https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/CDbPC2YgiHE

https://groups.google.com/a/isocpp.org/d/topic/std-proposals/tKioR8OUiAw/discussion