Input Files Are Source Files

Clearing up some core vocabulary

Document #:	P3556R0
Date:	2025-03-15
Project:	Programming Language C++
Audience:	Core Working Group
Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Abstract

The C++ Standard uses the term *input file* without definition and appears to use the defined term *source file* for the same purpose. This paper examines the history of this wording to understand its intent, proposes a new term, *source text*, to represent the source of a program during translation, and then unifies the terms *input file* and *source file* as just *source file*.

2 Revision History

R0 March 2025 (post-Hagenberg mailing)

Initial draft of this paper

3 Introduction

The act of translating C++ code is broken down into nine phases in 5.2 [lex.phases]. The origin of this paper was an editorial pull request changing the undefined term *input file* to the clearly specified term *source file* in translation phase 1.

The CWG chair raised concerns that the existing terminology, having been extensively reviewed in the process of adopting [P2295R6], should not be changed lightly and suggested that a paper be written to persuade the Core Working Group that a change would be helpful and is necessary. Subsequent research suggests that, at that time, wider concerns regarding which parts of our specification directly consume the input passed into the translator and which parts are acting on the source that has been input into translation were not actively addressed but seem to be the basis for confusion of the *input file* vs. *source file* terminology.

This proposal is intended to have no functional change but to directly address the question of when the specification is dealing with the immutable (to the translator) source file that is fed into translation and when it is in the process of consuming and manipulating its copy of those inputs during the act of translation.

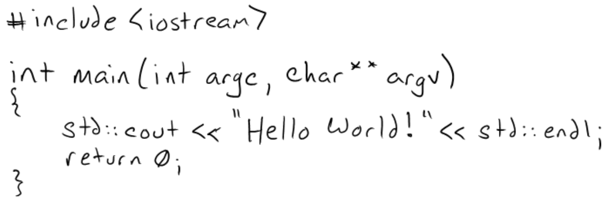
4 Categories of Files

4.1 What is a source file?

The Standard very clearly defines the term *source file* in [lex.separate]p1:

> The text of the program is kept in units called *source files* in this document.

Consider the example program, posted to Stack Overflow[1] by James McNellis, that highlights that source files need not be text files. McNellis claims, correctly, that the following image is a well-formed C++ program and complains that all his compilers reject it.



```
#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Figure 1: Hello World!

In McNellis's example, the source file is a piece of paper containing handwritten code, an image of which he posted. Is this piece of paper a valid source file? I posit the answer is *yes*.

Imagine a curious college student from the maker community[2] takes up the challenge of implementing a C++ compiler that accepts source files written on paper. The student feeds those programs into a physical machine that is the translator, which then scans the supplied source file (i.e., a sheet of paper storing a program), runs the scanned image through some OCR algorithms (influenced by current AI trends to better handle ambiguous characters such as `o`, `O`, and `0`), and turns the processed image into a sequence of translation character-set elements, representing end-of-line indicators as new-line characters, thus completing the implementation-defined source-file mapping of translation phase 1. This resulting sequence of translation-set characters is fed directly into phase 2 of an open-source compiler, such as a fork of Clang. Note that this source-to-translation-character-set-mapping is one way; the translator is not expected to issue diagnostics in the author's handwriting. An automated form feed would allow the consumption of programs comprising multiple sheets of paper, reinventing punch cards for the twenty-first century.

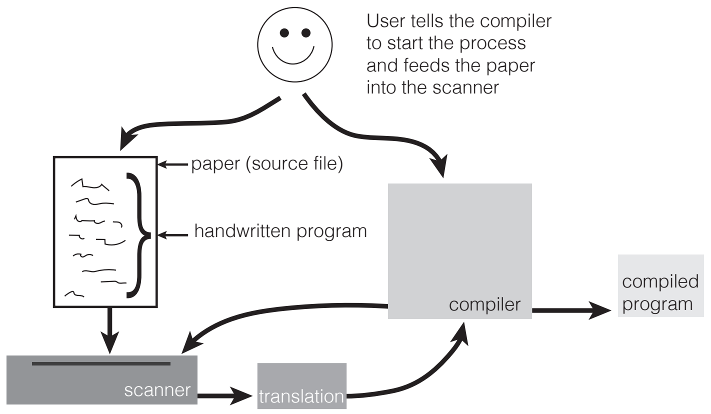

User tells the compiler to start the process and feeds the paper into the scanner
paper (source file)
handwritten program
compiler
compiled program
scanner
translation

Figure 2: Process

[1]https://stackoverflow.com/questions/5508110

[2]Wikipedia states, "The maker culture is a contemporary subculture representing a technology-based extension of DIY culture." Note that Wikipedia references are likely to change over time.

At no point in this hypothetical pipeline are the contents of the paper source file turned into a text file on disk; all traces of the data throughout the whole process are stored in working memory by the translation process.

In this scenario, the text of the program is clearly stored in a source file that is scrawled in McNellis's handwriting onto a piece of paper and, using the maker's machine and with the correct implementation-defined mapping from paper source file to the translation character set, is input as a stream of UTF-8 codepoints for the translator to process, completing phase 1 of translation. It is a valid and well-formed program.

4.2 What is an input file?

Input files are fed to the translator in phase 1 of translation. The term is never defined by the Standard but was introduced relatively recently in C++23 by paper [P2295R6] as a replacement for the similarly undefined adjective *physical* when appended to the term source file, i.e., *physical source file*.

CWG discussed, at length, the introduction of this new term, and that conversation formed the majority of the review of [P2295R6] at CWG's virtual meetings on 2022-02-25 and 2022-03-11. CWG failed to reach a consensus on the terminology. Perhaps surprisingly, when the paper was next reviewed on 2022-07-01, the topic was not raised, and the paper was promptly approved.

No clear connection exists between source files and input files, although input files seem to be some unspecified subset of source files; if input files do not hold programs, then what are they feeding to the translator?

4.3 What is a physical source file?

Physical *source file* is an obsolete term no longer used by the Standard. C++20 is the last Standard to use it, and the term is used only as the input to the start of phase 1 of translation and in an Annex C compatibility note describing how we changed phase 1 compared to a previous Standard.

However, closer examination shows that the term used by phase 1 is not simply *physical source file*, but rather the larger, compound term *physical source file* ***character***, which appears to be the intended term since the index includes *physical source file character* and references only this clause and omits *physical source file*.

4.4 What does a source file contain?

Regardless of the terminology being *source file* or *input file*, the file's contents are mapped in an implementation-defined manner to a sequence of translation character set elements, representing end-of-line indicators as new-line characters. However, how should we describe its contents? Should we even try, if we want to support an open set of representations of programs?

The current wording presumes the contents of an input file are characters, whereas this proposal suggests we use the less prescriptive term *contents*, which allows for arbitrary initial representation, such as images that are mapped to characters during phase 1.

5 Proposed Solution

For the purposes of the Standard, I believe we are dealing with two artifacts: first, the *source file* that is the immutable source of truth where programs are stored and that cannot be manipulated or updated by the translator and, second, the *source text* that is mapped from the source file into memory and then repeatedly processed and manipulated during translation phases 1–6 before emitting a token stream in phase 7.

The current wording has the notion of *input file* and *source file*, which appear to be indistinguishable in this context, and treats *source file* according to my notion of *source text* above; the current wording makes no distinction between the file that is read and the contents of that file.

I propose that input files *are* source files and that we rename them accordingly. We would then define *source text* as the sequence of translation character-set elements produced at the end of phase 1.

With the new vocabulary established, review every occurrence of *source file* in the Standard and either affirm the term is used correctly or replace *file* with *text* if the contents are being manipulated or the specification in question is clearly referring to processing text after translation phase 1.

The other defining characteristic of a source file is that many source files — but not all, i.e., only those that are included or imported — have names by which include directives and import statements locate them. No clear method exists to name *source text*, so any specification referring to a name is clearly referring to a source *file*, not source *text*.

6 Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N5008], the latest draft at the time of writing.

In addition to the usual insert and ~~remove~~ markup, I will use **bold** to indicate uses of **source file** that remain unchanged.

All paragraphs of the Core Language part of the Standard that use the term *source file* or *input file* are cited, even if there are no changes, to provide context and a consistent audit. However, almost all occurrences of *source file* in the Standard Library's clauses are unaffected by this paper (after audit), so only the one paragraph that is worth exploring is quoted. Note that some paragraphs are quoted only to include a footnote that uses the term *source file*.

5.1 [lex.separate] Separate translation

1 The text of the program is kept in units called ***source files*** in this document. A **source file** together with all the headers (16.4.2.3 [headers]) and **source files** included (15.3 [cpp.include]) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (15.2 [cpp.cond]) preprocessing directives, as modified by the implementation-defined behavior of any conditionally-supported-directives (15.1 [cpp.pre]) and pragmas (15.9 [cpp.pragma]), if any, is called a *preprocessing translation unit*.

[*Note 1:* A C++ program need not all be translated at the same time. —*end note*]

5.2 [lex.phases] Phases of translation

1 An implementation shall support ~~input~~source files that are a sequence of UTF-8 code units (UTF-8 files). It may also support an implementation-defined set of other kinds of ~~input~~source files, and, if so, the kind of ~~an input~~a source file is determined in an implementation-defined manner that includes a means of designating ~~input~~source files as UTF-8 files, independent of their content.

[*Note 1:* In other words, recognizing the U+FEFF BYTE ORDER MARK is not sufficient. —*end note*]

If ~~an input~~a source file is determined to be a UTF-8 file, then it shall be a well-formed UTF-8 code unit sequence and it is decoded to produce a sequence of Unicode scalar values. A sequence of translation character set elements (5.3.1 [lex.charset]) is then formed by mapping each Unicode scalar value to the corresponding translation character set element. In the resulting sequence, each pair of characters in the input sequence consisting of U+000D CARRIAGE RETURN followed by U+000A LINE FEED, as well as each U+000D CARRIAGE RETURN not immediately followed by a U+000A LINE FEED, is replaced by a single new-line character.

For any other kind of ~~input~~source file supported by the implementation, ~~characters~~the contents are mapped, in an implementation-defined manner, to a sequence of translation character set elements, representing end-of-line indicators as new-line characters.

2 If the first translation character is U+FEFF BYTE ORDER MARK, it is deleted. Each sequence of a backslash character (\) immediately followed by zero or more whitespace characters other than new-line followed by a new-line character is deleted, splicing physical source lines to form *logical source lines*. Only the last backslash on any physical source line shall be eligible for being part of such a splice.

[*Note 2:* Line splicing can form a *universal-character-name* (5.3.1 [lex.charset]). —*end note*]

~~A source file~~Source text that is not empty and that (after splicing) does not end in a new-line character shall be processed as if an additional new-line character were appended to the ~~file~~text.

3 The source ~~file~~text is decomposed into preprocessing tokens (5.5 [lex.pptoken]) and sequences of whitespace characters (including comments). ~~A source file~~Source text shall not end in a partial preprocessing token or in a

partial comment.[3] Each comment (5.4 [lex.comment]) is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of whitespace characters other than new-line is retained or replaced by one space character is unspecified. As characters from the source ~~file~~text are consumed to form the next preprocessing token (i.e., not being consumed as part of a comment or other forms of whitespace), except when matching a *c-char-sequence*, *s-char-sequence*, *r-char-sequence*, *h-char-sequence*, or *q-char-sequence*, *universal-character-names* are recognized (5.3.2 [lex.universal.char]) and replaced by the designated element of the translation character set (5.3.1 [lex.charset]). The process of dividing a source ~~file~~text's characters into preprocessing tokens is context-dependent.

[*Example 1:* See the handling of `<` within a `#include` preprocessing directive (15.3 [cpp.include]). —*end example*]

4 The source ~~file~~text is analyzed as a *preprocessing-file* (15.1 [cpp.pre]). Preprocessing directives (Clause 15) are executed, macro invocations are expanded (15.6 [cpp.replace]), and `_Pragma` unary operator expressions are executed (15.12 [cpp.pragma.op]). A `#include` preprocessing directive (15.3 [cpp.include]) causes the named header or **source file** to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

7 Each preprocessing token…

…

…are required to be available.

[*Note 4:* **Source files**, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. —*end note*]

5.6 [lex.header] Header names

header-name:
`<` *h-char-sequence* `>`
`"` *q-char-sequence* `"`
h-char-sequence:
h-char h-char-sequence$_{opt}$
h-char:
any member of the translation character set except new-line and U+003E GREATER-THAN SIGN
q-char-sequence:
q-char q-char-sequence$_{opt}$
q-char:
any member of the translation character set except new-line and U+0022 QUOTATION MARK

1 The sequences in both forms of *header-name*s are mapped in an implementation-defined manner to headers or to external **source file** names as specified in 15.3 [cpp.include].

[*Note 1:* Header name preprocessing tokens appear only within a `#include` preprocessing directive, a `__has_include` preprocessing expression, or after certain occurrences of an import token (see 5.5 [lex.pptoken]). —*end note*]

10.3 [module.import] Import declaration

5 A *module-import-declaration* that specifies a *header-name* `H` imports a synthesized *header unit*, which is a translation unit formed by applying phases 1 to 7 of translation (5.2 [lex.phases]) to the **source file** or header nominated by `H`, which shall not contain a *module-declaration*.

[*Note 2:* A header unit is a separate translation unit with an independent set of defined macros. All declarations within a header unit are implicitly exported (10.2 [module.interface]), and are attached to the global module

[3][FN 9] A partial preprocessing token would arise from ~~a~~ source ~~file~~text ending in the first portion of a multi-character token that requires a terminating sequence of characters, such as a *header-name* that is missing the closing `"` or `>`. A partial comment would arise from ~~a~~ source ~~file~~text ending with an unclosed `/*` comment.

(10.1 [module.unit]). —*end note*]

An importable header is a member of an implementation-defined set of headers that includes all importable C++ library headers (16.4.2.3 [headers]). `H` shall identify an importable header. Given two such *module-import-declarations*:

(5.1) — if their *header-name*s identify different headers or **source files** (15.3 [cpp.include]), they import distinct header units;

(5.2) — otherwise, if they appear in the same translation unit, they import the same header unit;

(5.3) — otherwise, it is unspecified whether they import the same header unit.

[*Note 3:* It is therefore possible that multiple copies exist of entities declared with internal linkage in an importable header. —*end note*]

[*Note 4:* A *module-import-declaration* nominating a *header-name* is also recognized by the preprocessor, and results in macros defined at the end of phase 4 of translation of the header unit being made visible as described in 15.5. Any other *module-import-declaration* does not make macros visible. —*end note*]

15 [cpp] Preprocessing directives

15.1 [cpp.pre] Preamble

1 A *preprocessing directive* consists of a sequence of preprocessing tokens that satisfies the following constraints: At the start of translation phase 4, the first preprocessing token in the sequence, referred to as a *directive-introducing* token, begins with the first character in the source ~~file~~text (optionally after whitespace containing no new-line characters) or follows whitespace containing at least one new-line character, and is

(1.1) — a `#` preprocessing token, or

(1.2) — an `import` preprocessing token immediately followed on the same logical source line by a *header-name*, `<`, *identifier*, *string-literal*, or `:` preprocessing token, or

(1.3) — a `module` preprocessing token immediately followed on the same logical source line by an *identifier*, `:`, or `;` preprocessing token, or

(1.4) — an `export` preprocessing token immediately followed on the same logical source line by one of the two preceding forms.

The last preprocessing token in the sequence is the first preprocessing token within the sequence that is immediately followed by whitespace containing a new-line character.

7 The implementation can process and skip sections of source ~~files~~text conditionally, include other **source files**, import macros from header units, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

15.2 [cpp.cond] Conditional inclusion

4 The header or **source file** identified by the parenthesized preprocessing token sequence in each contained *has-include-expression* is searched for as if that preprocessing token sequence were the *pp-tokens* in a `#include` directive, except that no further macro expansion is performed. If such a directive would not satisfy the syntactic requirements of a `#include` directive, the program is ill-formed. The *has-include-expression* evaluates to `1` if the search for the **source file** succeeds, and to `0` if the search fails.

15 ... If none of the conditions evaluates to true, and there is a `#else` directive, the group controlled by the `#else` is processed; lacking a `#else` directive, all the groups until the `#endif` are skipped.[4]

15.3 [cpp.include] Source file inclusion

1 A `#include` directive shall identify a header or **source file** that can be processed by the implementation.

3 A preprocessing directive of the form

[4][FN 125] As indicated by the syntax, a preprocessing token cannot follow a `#else` or `#endif` directive before the terminating new-line character. However, comments can appear anywhere in ~~a~~ source ~~file~~text, including within a preprocessing directive.

```
# include " q-char-sequence " new-line
```

causes the replacement of that directive by the ~~entire contents~~ source text of the **source file** identified by the specified sequence between the " delimiters. The named **source file** 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

```
# include < h-char-sequence > new-line
```

with the identical contained sequence (including `>` characters, if any) from the original directive.

6 A `#include` preprocessing directive may appear in a **source file** that has been read because of a `#include` directive in another file, up to an implementation-defined nesting limit.

8 [*Note 1:* An implementation can provide a mechanism for making arbitrary **source files** available to the `< >` search. However, using the `< >` form for headers provided with the implementation and the `" "` form for sources outside the control of the implementation achieves wider portability. For instance:

```
#include <stdio.h>
#include <unistd.h>
#include "usefullib.h"
#include "myprog.h"
```

—*end note*]

15.5 [cpp.import] Module directive

3 If a *pp-import* is produced by **source file** inclusion (including by the rewrite produced when a `#include` directive names an importable header) while processing the *group* of a *module-file*, the program is ill-formed.

15.6.5 [cpp.rescan] Rescanning and further replacement

1 After all parameters in the replacement list have been substituted and `#` and `##` processing has taken place, all placemarker preprocessing tokens are removed. Then the resulting preprocessing token sequence is rescanned, along with all subsequent preprocessing tokens of the source ~~file~~text, for more macro names to replace.

3 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source ~~file~~text's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

15.7 [cpp.line] Line control

2 The line number of the current source line is the line number of the current physical source line, i.e., it is one greater than the number of new-line characters read or introduced in translation phase 1 (5.2 [lex.phases]) while processing the source ~~file~~text to the current preprocessing token.

4 A preprocessing directive of the form

```
# line digit-sequence " s-char-sequence_opt " new-line
```

sets the presumed line number similarly and changes the presumed name of the **source file** to be the contents of the character string literal.

15.11 [cpp.predefined] Predefined macro names

1 The following macro names shall be defined by the implementation:

```
__DATE__
```

The date of translation of the **source file**: a character string literal of the form `"Mmm dd yyyy"`, where the names of the months are the same as those generated by the `asctime` function, and the first character of `dd` is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

```
__FILE__
```

The presumed name of the current **source file** (a character string literal).[5]

```
__LINE__
```

The presumed line number (within the current source ~~file~~text) of the current source line (an integer literal).

```
__TIME__
```

The time of translation of the **source file**: a character string literal of the form `"hh:mm:ss"` as in the time generated by the `asctime` function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

19.3.3 [assertions.assert] The assert macro

1 If `NDEBUG` is defined as a macro name at the point in the source ~~file~~text where `<cassert>` is included, the `assert` macro is defined as

```
#define assert(...) ((void)0)
```

[5] [FN 131] The presumed **source file** name can be changed by the #line directive.

7 Acknowledgements

Thanks to Michael Park for the pandoc-based framework used to transform this document's source from Markdown.

Thanks to James McNellis for the most excellent example and to Corentin Jabot for the original work specifying UTF-8 as a portable file format for C++.

Thanks to Lori Hughes for reviewing this paper and for providing editorial feedback.

8 References

[N5008] Thomas Köppe. Working Draft, Programming Languages — C++.
https://wg21.link/n5008

[P2295R6] Corentin Jabot, Peter Brett. 2022-07-01. Support for UTF-8 as a portable source file encoding.
https://wg21.link/p2295r6