Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 343–350.

How #includes Affect Build Time in Large Systems

József Mihalicza

Department of Programming Languages and Compilers, Eötvös Loránd University

Abstract

The #include concept, present in numerous mainstream languages like C, C++, Objective C, has unexpectedly bad effects on build times. Many current large systems using the #include technique suffer from unacceptable long build procedure. Long builds waste many valuable manhours or even man months, elongate development and as a result make keeping deadlines harder. Using a different approach in a large system is proven to result in even 10 times faster builds. This paper compares various widely used freeware software packages and shows both the overhead the #includes cause and the gain achieved by applying the mentioned approach.

Keywords: C++, build time

1. Introduction

C++ [1] is a widely spread programming language, many industrial projects are developed in it. Though it has numerous advantages, the long compilation time is one of its drawbacks, especially if we compare it to other languages.

The total build time of a software system affects the time a compilation error is identified in continuous integration [2] environments. From a developer's perspective the time of incremental builds is what rather counts, but after a synchronisation with the source repository a full build might be necessary. The probability of that grows with the number of developers working on the same source base. If the QA team works based on packages made by a build automation tool (e.g. CruiseControl) the minimum time a new package can get to them is also determined by the full build time.

Bad build performance leads to various consequences. The long waits during a develop-test-develop-test cycle can distract the developer's focus. In case of widely used headers (e.g. one included in the precompiled header) sometimes a worse solution is chosen intentionally merely to avoid long compilation. These choices are rarely replaced by the optimal ones afterwards.

With template [3] metaprograms one can make the compiler run even quite complex algorithms [8], C++ templates are proven to be Turing complete [4]. In [6] a profiler framework is presented to detect slow components in these metaprograms.

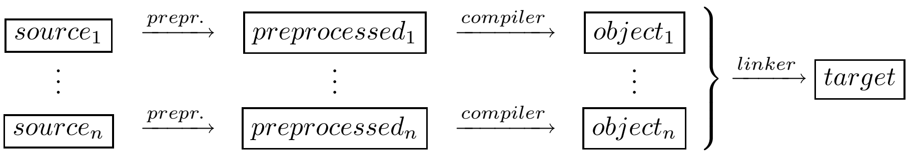
In [7] the `#include` mechanism is proven to be one of the main sources of bad build performance. There a program transformation is presented which can help in radically reducing the full compilation time.

The purpose of this paper is the evaluation of that program transformation method by applying it to three open source C++ libraries. In section 2 I briefly show the method itself. Then in section 3 I describe what exact libraries I chose for the test and present the steps of their transformation, mainly focusing on the problematic points. Section 4 summarizes and discusses the results in numbers. Section 5 contains a brief conclusion and some ideas for future work.

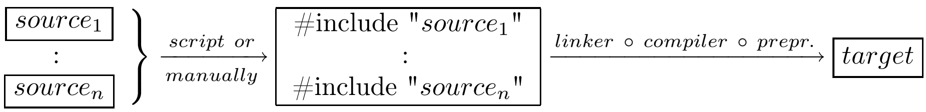
2. The method

The idea is the following: instead of compiling numerous `.cpp` files one by one separately, we concatenate them together using the `#include` directive and compile at once. This way if there is a header file included in many original source files, it will be processed only once.

The build flow changes from:



to:



For example, instead of compiling

```
lib_1_file_a.cpp
lib_1_file_b.cpp
lib_1_file_c.cpp
```

separately, we will have a file, to say `lib_1_all.cpp` with the following contents:

```
#include "lib_1_file_a.cpp"
#include "lib_1_file_b.cpp"
#include "lib_1_file_c.cpp"
```

This transformation can bring in new errors because of different reasons. Section 3 discusses them in details. Moreover, this new configuration can hide serious dependency violations. As the code evolves, obviously erroneous code fragments can remain unrecognized and cause unforeseen trouble later. To avoid this side effect it is advisable to maintain also a configuration that does not apply the described transformation.

Integrated development environments (IDE) raise the question of whether to remove the original source files from the project. Many of them allows exclusion of source modules piecewise within a configuration. If this is not possible, having both the `..._all.cpp` files and the original sources in the project would result in double compilation and double definition of all symbols later at link time. Removing the sources, however, may prevent the IDE from helping the developer navigating in the source files. Fortunately most modern IDEs parse also the included files for symbols regardless whether they are added to the project. As discussed before, maintaining a transformationless configuration is recommended. Either this can be achieved by excluding the original and the `..._all.cpp` files in two separate build configurations respectively, or by using preprocessor directives as shown below:

```
lib_1_all.cpp:
#ifdef FAST_COMPILATION
#undef FAST_COMPILATION
#include "lib_1_file_a.cpp"
#include "lib_1_file_b.cpp"
#define FAST_COMPILATION
#endif
```

```
lib_1_file_a.cpp:
#ifndef FAST_COMPILATION
original contents
of lib_1_file_a.cpp

#endif
```

This latter approach has the advantage that it works in all environments and does not need a separate build configuration. Though in this case the compiler should preprocess all source files even in the fast configuration, it will remain fast since the whole file is disabled and will not include further sources.

In `Makefile` based projects we can either create a rule for the composition of the `..._all.cpp` files, or just create them manually.

The procedure has an interesting side effect. Having all files included together into a single file, we get a centralized place where each compilation unit of the library can be enabled or disabled easily with precompiler directives. Suppose, for example, that `lib_1` has some independent subcomponents, which are not big enough to be separate libraries, but still we would like to handle them together. In this case we can dedicate preprocessor directives to these subcomponents and control their inclusion in the resulting library:

```
#include "lib_1_base_file_a.cpp"
#include "lib_1_base_file_b.cpp"
#ifdef LIB_1_SUBCOMPONENT1
#include "lib_1_subcomponent_1_file_a.cpp"
#include "lib_1_subcomponent_1_file_b.cpp"
#endif
#ifdef LIB_1_SUBCOMPONENT2
#include "lib_1_subcomponent_2_file_a.cpp"
...
```

3. Transformation case studies

The program transformation described in section 2 has been applied to three open source libraries:

OpenSceneGraph 2.8.2 3D graphics toolkit

wxWidgets 2.8.10 Widget toolkit for creating GUIs

Xerces 3.0.1 XML parser library

C++ units are not typically designed for being included together, they can use arbitrary local symbols that are not unique among the source files. Therefore our transformation easily leads to compilation errors. Based on the experience with the libraries above, this section presents what modifications may be needed to make the code compile again after the transformation. We will see what constructs we have to be careful with when using this technique.

unwanted sources If scripts are used to gather the source files to be included together, be careful not to add a file that is otherwise not part of the project. In OpenSceneGraph for example, there is a `Matrix_implementation.cpp` which acts like a template. It contains a generic implementation of a matrix type, where the actual class type is everywhere `Matrix_implementation`, a type which is not defined earlier in that file. `Matrixd.cpp` and `Matrixf.cpp` use this generic implementation, defining the `Matrix_implementation` before including the `.cpp`:

```
// specialise Matrix_implementaiton to be Matrixd
#define Matrix_implementation Matrixd
...
// now compile up Matrix via Matrix_implementation
#include "Matrix_implementation.cpp"
```

This is a pattern [9] for simulating templates. Here I had just to remove the inclusion of the generic implementation from the `..._all.cpp` file.

double definition of inline function The generic matrix implementation, see above, uses an inline function defined in the `.cpp` file. After including together `Matrixd.cpp` and `Matrixf.cpp` this inline function was defined twice in the same compilation unit, which is an error. The solution is a header guard-like prevention from double definition:

```
#ifndef Matrix_implementation_cpp_included
#define Matrix_implementation_cpp_included
template <class T> inline T SGL_ABS(T a)
{ return (a >= 0 ?  a :  -a); }
#endif
```

local symbols with identical names This is the most frequent conflict type. Originally different compilation units chose the same identifier to denote a local element. In OpenSceneGraph for example these were mainly typedefs, static variables and complete classes. I had to assign distinct names for these variables. In `abc.cpp` for example I changed

```
typedef buffered_value< ref_ptr<abc::Extensions> >
BufferedExtensions;
static BufferedExtensions s_extensions;
```

to

```
typedef buffered_value< ref_ptr<abc::Extensions> >
abc_BufferedExtensions;
static abc_BufferedExtensions s_abc_extensions;
```

where `abc` was one of `BlendEquation`, `BlendFunc`, `BufferObject` etc.

multiply defined macros As the compilation no longer ends at the end of the source file, we have to add `#undef` directives for each corresponding `#define` introduced in the given unit. Each original source file assumes that no custom macros (except for those coming from the environment or make file) are defined at the 0. position of the file. It can happen that we do not even notice that a macro from a previous file remained active in the next included file. These bugs are very difficult to find afterwards, so the best is to follow the simple rule of thumb: `#undef` every macro at the end. This step can easily be automated though it is not that exhaustive to do manually.

macro redefinition In some files I had to put `#ifndef` guards around the definition of an otherwise standard macro. Probably the original intent was not to include the whole world because of this sole macro definition.

conflict with a header included in a preceding unit This is an evil one. It happens when some previously included header file contains such definitions that cause conflicts in another file later on. If the problematic symbol is a macro, a well placed `#undef` should solve it, otherwise renaming [10] may be necessary. In my case the `min` and `max` macros (of `windef.h`) conflicted with `numeric_limits::min` and `numeric_limits::max` respectively.

In some cases the conflict is not, or not easily resolvable. We always have the option to put these conflicting sources into separate `_all.cpp` files. In the Xerces library some sources include `windows.h`, which in turn includes `winsock.h`. Another source later includes `winsock2.h` which conflicts with the definitions coming from `winsock.h`. Though this concrete conflict could have been resolved in numerous other ways, I chose to let this case be a demonstrative example. I put the sources including `winsock2.h` into a separate `_all.cpp` file, `NetAccessor_all.cpp`.

using namespace vs. local symbol The usage of `using namespace` can easily lead to name clashes in our method. The solution is preferably removing the using directive or alternatively renaming the conflicting local symbol.

The following table shows what amount of change was needed to make the transformated libraries compile:

Library	Files	Modified files	Characters added	Average bytes per modified file
OpenSceneGraph	1661	31	2418	78
wxWidgets	825	82	12100	148
Xerces	811	50	5082	102

As the numbers show, 2-10% of the source files has to be adapted to the new compilation method. These changes are safe and simple, in most cases only renames.

4. Results

Now let us see the effects of the transformation in numbers. The following tables will show the change both in the compilation time and in the summed up preprocessed source size in LOC metric, for all the three libraries:

Target	original time	transformed time	ratio	original LOC	transformed LOC	ratio
osg plugin	626	18	2.9%	5388153	75279	1.4%
Osg	669	41	6.1%	5344590	150242	2.8%
osgUtil	229	53	23.1%	1814422	83677	4.6%
osgDB	143	19	13.3%	1373930	93451	6.8%
osgGA	95	11	11.6%	877402	66304	7.6%
osgViewer	114	24	21.1%	872151	124964	14.3%
osgText	58	15	25.9%	461226	68671	14.9%
OpenThreads	7	2	28.6%	166959	72085	43.2%
osgviewer app	13	13	100.0%	67380	67380	100.0%
all	1954	196	10.0%	16366213	802053	4.9%

Table 1: OpenSceneGraph

We can see that though the source size compression was maximal at wxWidgets, it is not reflected in the time ratio. The reason is the heavy usage of precompiled headers. Similarly, though Xerces has twice as big size ratio, still the time ratio is almost the same as of OpenSceneGraph.

Table **??** shows the overhead of preprocessing before and after the transformation. It seems in Xerces either there is less coupling between the modules, or the `#include` dependencies were consciously kept minimal.

Target	original time	transformed time	ratio	original LOC	transformed LOC	ratio
core	58	24	41.4%	19394804	192829	1.0%
base	24	11	45.8%	6534236	129696	2.0%
xrc	26	11	42.3%	4868302	99163	2.0%
adv	16	9	56.3%	2171443	109069	5.0%
html	18	9	50.0%	2111128	100901	4.8%
wxtiff	20	2	10.0%	1937586	69857	3.6%
net	9	4	44.4%	895406	80094	8.9%
richtext	16	9	56.3%	824870	109411	13.3%
aui	11	7	63.6%	532399	97514	18.3%
media	9	5	55.6%	450098	95189	21.1%
xml	5	4	80.0%	148692	75148	50.5%
qa	7	5	71.4%	147088	85731	58.3%
odbc	4	2	50.0%	147088	73544	50.0%
gl	10	5	50.0%	85731	85731	100.0%
dbgrid	6	4	66.7%	85731	85731	100.0%
wxjpeg	7	2	28.6%	76217	12665	16.6%
wxexpat	1	1	100.0%	69164	31842	46.0%
wxpng	4	1	25.0%	45500	14843	32.6%
wxregex	1	0	0.0%	13345	7693	57.6%
wxzlib	1	0	0.0%	11213	5810	51.8%
all	253	115	45.5%	40550041	1562461	3.9%

Table 2: wxWidgets

Target	original time	transformed time	ratio	original LOC	transformed LOC	ratio
XercesLib	204	22	10.8%	2287482	181125	7.9%
NetAccessor			10.8%	111402	55685	50.0%
all	204	22	10.8%	2398884	236810	9.9%

Table 3: Xerces

Library	original LOC	preprocessed LOC	ratio	ratio
OpenSceneGraph	91205	16366213	17944.43%	4.90%
OpenSceneGraph transformed		802053	879.40%	
wxWidgets	357642	40550041	11338.17%	3.85%
wxWidgets transformed		1562461	436.88%	
Xerces	122396	2398884	1959.94%	9.87%
Xerces transformed		236810	193.48%	

Table 4: Preprocessing overhead

5. Conclusion and future works

The results showed that the presented method for reducing full compilation time can lead to as much as even 10 times faster builds. The ratio of the preprocessed and the original source size gets dropped down radically (to 4-10%), under 1000% in all cases, 200% for Xerces, which means that the preprocessed source does not reach the double of the original size. On the other hand, manual code modifications may be necessary to make the transformed source work again, and some rules and limitations are to be followed in order to keep the method working.

Future works may include the automation of an error free transformation, the development of a coding style to help remaining compatible with the approach, or further studies on eliminating the overhead of `#include`s.

References

[1] Stroustrup, B., The C++ Programming Language, *Addison-Wesley* (2000)

[2] Duvall, P., Matyas, S., Glover, A., Continuous integration: improving software quality and reducing risk *Addison-Wesley Professional* (2007)

[3] Vandevoorde, D., Josuttis, N.M., C++ Templates: The Complete Guide *Addison-Wesley Professional* (2002)

[4] Veldhuizen, T.L., C++ templates are turing complete. *Technical Report* (2003)

[5] Abrahams, D., Gurtovoy, A., C++ Template Metaprogramming: Concept, Tools, and Techniques from Boost and Beyond *Addison-Wesley Professional* (2004)

[6] Porkolab, Z., Mihalicza, J., Patakai, N., Sipos, A. Analysis of profiling techniques for C++ template metaprograms *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica, 30:97-116* (2009)

[7] Mihalicza, J., Compile C++ systems in quarter time *Proceedings of 10th International Scientific Conference on Informatics* (2009)

[8] Alexandrescu, A., Modern C++ design: generic programming and design patterns applied *Addison-Wesley* (2001)

[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design patterns: elements of reusable object-oriented software *Addison-Wesley Professional* (1995)

[10] Fowler, M. Refactoring: Improving the Design of Existing Code *Addison-Wesley* (1999)

József Mihalicza
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
e-mail: `jmihalicza@gmail.com`