Document Number: P0235R0
Date: 2016-02-05
Audience: EWG, SG14
Reply To: Guy Somberg
gsomberg@blizzard.com
Brian Fitzgerald
bfitz@blizzard.com

A Packaging System for C++

Guy Somberg

Brian Fitzgerald

Abstract

We present a design and specification of a packaging system for C++. This system differs from modules in that it is all about source code distribution, rather than the mechanics of compiling. A useful packaging system is important to unify software packages, and to make it trivial (or, at least, as easy as possible) both to use new C++ libraries and to package libraries for distribution.

Contents

1 Introduction

The success of many programming languages such as Perl, Java, and Ruby can be attributed, in large part, to their packaging systems. C++ is successful despite the lack of any centralized packaging system or

standardized source code package format[1]. However, that success does not mean that we should ignore the benefits of a packaging system.

We have performed a survey of existing packaging systems in other languages, published on GitHub[2], which we have drawn upon extensively as reference. The system we describe in this paper defines standards for layout and file format of packages. It is as simple as it is possible for it to be, while still providing features both for library authors and for library users.

Ultimately, the goal of this packaging system is different whether you are a library author or a library user:

- As a library user, the time it takes me from going on a website and seeing a C++ library to using it in my own project should be on the order of seconds, and should not require me to do anything other than add a single line to my source code.
- As a library author, I have the ability to create source code packages for consumption by this packaging system with minimal (or, ideally, no) changes to my project. Furthermore, I can define a set of parameters to the build

As a side-benefit, there is a related feature (which can be split out into its own proposal if so desired) that will allow a single translation unit to reference a set of others to compile in sequence, thus allowing you to define an entire program in a single file.

There has, for a long time, been a desire for this sort of system to exist in C++. At his BoostCon 2008 keynote presentation entitled "A C++ library wish list", Bjarne Stroustrup described a world of easily-installed and packaged libraries. And, in fact, in slide #33 of that presentation, he shows a system that is similar in spirit to what we are proposing here in this document.

2 The Problem

When I, as a C++ programmer, want to use a C++ library that is not built into my compiler or distribution, it is a big distraction. I will have to:

- Acquire the library. Hopefully they have source code, and, if not, hopefully they have a compiled binary that is compatible with my chosen compiler and command-line settings.
- Build the library. If I'm lucky, they've included a makefile or project file. Sometimes they don't. Sometimes it's worse, and I have to install Perl or some other third-party tool in order to generate header or source files.
- Integrate the library into my build system. Sometimes this is as easy as copying files into my project, but I may end up having to figure out the proper subset of files to include. Oh, and don't forget build flags…

Finally, if I've done all of that right, then I'll have a library that I can start to use. Wait, what was I working on? I've now forgotten why I needed this library in the first place.

---

[1] C++ is not unique in this regard. Python is another example of a language that became successful before a standardized package system. However, as successful as Python was without standardized packages, it became even more popular once a packaging system was in place.

[2] See Section D (References) "A Survey of Programming Language Package Systems" for the URL.

What we need, therefore, is a way that I can trivially tell the compiler “hey, use that library” and have confidence that the library will be acquired, built, and integrated without any extra work on my part. On the other side of the coin, we need a way for library authors to easily create source packages that can be used by C++ developers using the standard tooling.

3 Packages Design

We want it to be as trivial as possible to use packages from the get-go. Anything more than a single line of code is too much. Let’s take a look at how we might use a library like zlib:

```
#using package "zlib"
void decompress(array_view<byte> buffer) {
  // ...
  inflate(state, Z_NO_FLUSH);
  // ...
}
```

When the “zlib” package is imported, it is opened up (we’ll see how in a minute), its contents are added to a set of files to compile, and a defaulted set of headers and module imports is included into the current translation unit. The compiler then resets itself, then compiles each of the package’s files as its own translation unit.

It is, of course, expected (but not required) that the compiler will cache the results of these compiles, so if another translation unit requests the same files with the same compile settings, it will already have them available.

This document has settled on the preprocessor directive '#using' as its mechanism for using packages. We know that the C++ language is averse to adding new features to the preprocessor, so consider the directive as a straw-man for the purposes of exposition. For a discussion of the different syntax options, see Section 5.

3.1 Syntax Overview

There is one new preprocessor directive that we are adding for this proposal: #using. That directive, then, has three different keywords that trigger different behavior: package, option, and path. See section 6 for a discussion of why we chose a preprocessor keyword, and other alternatives.

```
// Default package import
#using package "my_package"

// package import with version selection
#using package "my_package" version "1.2.3"

// package import, overriding the default list of headers with an empty list
// the 'version' syntax works in addition to this, but is omitted for brevity
#using package "my_package" ()

// package import with specific headers included
#using package "my_package" ("foo.h", "bar.h")

// package import with specific modules imported
```

```
// This list can be empty, and you can have both modules and headers defined
#using package "my_package" [foo, bar.baz]

// set package option MY_KEY to "my_value"
#using option MY_KEY = "my_value"

// add foo.cpp to the set of files to compile
#using path "foo.cpp"

// add all files in the path foo/ to the set of files to compile
#using path "foo"

// add all files in the path foo/ and all of its subdirectories
// to the set of files to compile
#using path "foo" recursive
```

Details of the semantics for '#using package' are in Section 3.5.1, '#using option' in Section 3.5.2, and '#using path' in section 3.5.3.

Let's first examine the details of what is a package.

3.2 Packages

A package is a specific layout of files and directories, which is typically packaged as a zip file, but can also be a directory in the filesystem. If it is a zip file, then it must conform to ISO/IEC 21320-1:2015 (the ISO standard describing the zip file format, with some restrictions on compression and settings).

The layout of a package file must look like this:

Entry	Meaning
/	Root of the path
/MANIFEST	Manifest file. Optional.
/include	Include files go here
/source	Source files go here
/obj	Object files – reserved pathname for the system, but usage details are implementation defined. Will typically be used to store .o or .obj precompiled versions of the contents of the /source directory.
/lib	Library archive files – reserved pathname for the system, but usage details are implementation defined. Will typically be used to store .a or .lib precompiled versions of the contents of the /source directory.
/bin	Binary tool files – reserved pathname for the system, but usage details are implementation defined. Will typically be used to store tools that need to be run. For example, for a Google Protobuf package, the /bin directory could contain copies of the protoc protobuf compiler.

See section 3.6 for a description of the MANIFEST file format.

3.3 What the Compiler Does with Packages

Once it's been told to load a package, the compiler needs to know what to do with it. Here is what the compiler does in order to use a package:

1) Find the package
   a) If no version is selected, then use an implementation-defined algorithm to discover the package's location at any version.
   b) If a version is selected, then use an implementation-defined algorithm to discover that specific version.
   c) If the package cannot be found, then it is an error.
2) If the package has already been loaded
   a) Jump to step 5
3) Look for the MANIFEST file in the root of the package's directory structure and parse it.
4) Starting with the root directory, perform the following procedure for each directory that matches:
   a) If there is a directory called “include”, then add it to the Package Include Path List non-recursively.
   b) If there is a directory called “source”, then add all of its contents recursively to the File Set.
      i) If there are modules in the package, then appropriate module metadata is generated at this time.
   c) For every other directory in the list:
      i) If it matches one of the other reserved names (“obj”, “lib”, or “bin”) then perform an implementation-defined task on it. The default is to ignore it.
      ii) Apply special package option rules, as defined in section 3.5.2.1.
      iii) If it is not reserved and the special package option rules don't apply, then ignore the directory.
5) Include the default headers from the Package Include Path List, and import the default modules.

In short, when using a package, it adds the source files to the File Set, includes headers from inside of the package, and imports modules from inside of the package. Of course, the “as if” rule applies here, which will allow compiler vendors to cache the results of package compiles, or to use compatible object or library files from the reserved directories.

Step 5 deserves a bit more explanation. Let's say that we have a package called "foo", which contains a MANIFEST file that declares that the default includes are 'foo.h' and 'public/bar.h', and that the default imports are 'foo.containers' and 'bar.algorithms'. See section 3.6 for a description of the MANIFEST file format and how these defaults are defined.

You would use this package thus:

```
#using package "foo"
```

When the compiler sees this, it will go through the whole procedure described above, and then behave ***as if*** the '#using package' were replaced with:

```
#include "foo.h"
#include "public/bar.h"
import foo.containers;
import bar.algorithms;
```

3.4 The File Set

We have mentioned this mysterious “set of files to compile” or "File Set" a few times. Let’s formalize what we actually mean by it.

As the compiler is operating on a translation unit that uses one or more packages, it stores a set of files to compile after it has finished this translation unit. Once it has successfully completed building this translation unit, it resets its internal state, loads each file in turn, and compiles it as its own independent translation unit. When it is building each translation unit from inside of a package, the include search path is amended to include the Package Include Path List for that package.

The File Set can be modified in two ways:

1) **#using path** – This syntax allows the programmer to explicitly insert into the File Set. Note that there is no way to query, remove, or reorder the set – only insert. If a file is added into the set more than once, it is only compiled once.
2) **#using package** – This syntax loads a package, and inserts all of its contents into the File Set. Other than by using a package option, there is no way to control which files get chosen.

Here's one very important statement: if the file is not a C++ file (as determined in an implementation-defined manner), then the compiler should treat the file in an implementation-defined manner. This bunch of weasel-wording is there to allow the compiler to deal gracefully with files in other programming languages. We expect it to be common for C++ packages to have components written in C, assembly, or even Fortran. The compiler should do its best to compile each input file as a source file in its native language, and output an error if it cannot do so.

Now that we've got an idea of what a package is and how the compiler deals with it, let's take a look at the actual syntax.

3.5 Syntax of the #using Directive

3.5.1 #using package

The #using package syntax is the way in which you tell the compiler that you would like to use a package. There are many optional parts of the syntax, but the defaults are all reasonable. There is only one required piece of data: the package name.

```
#using package "foo"
```

This syntax loads the package with the default semantics, which are: default #includes and module imports, and an auto-selected version.

After that, there are optional header and module overrides. By default, the #using package directive will #include a default list of files, and import a default list of modules, both of which are defined in the MANIFEST file. (See section 3.6 for MANIFEST file format.) If you want to override this list with your own, then you can do so with this syntax:

```
#using package "foo" ("foo/foo.h", "bar/bar.h") [foo, bar.baz]
```

This will load the "foo" package, then include from the package the contents of "foo/foo.h" and "bar/bar.h" in that order. Next, it will import the modules: the moral equivalent of "import foo" and "import bar.baz".

After the header and module overrides, there is an optional version selector, which allows you to select a particular version of a package to use:

```
#using package "foo" version "1.2.3"
```

This syntax will load version 1.2.3 of the "foo" package. You can use this syntax with the header and module overrides, but we've left them out for brevity.

Finally, let's see the full '#using package' directive in all of its glory:

```
#using package "foo" \
  ("foo/foo.h", "bar/bar.h") \
  [foo, bar.baz] \
  version "1.2.3"
```

Whew! That's a lot of options, but not too bad. There are two things that make this syntax clean:

1.) As a code reviewer or a person looking at the code, you can stop reading wherever you want, and the rest of the line are all details that are less interesting.
2.) As a code writer, you can usually leave most of this line out because there are sensible defaults.

3.5.2 #using option

The '#using option' syntax allows the customer of a package to control subsets of a package that have been designated by the package author. The syntax is:

```
#using option identifier = string
```

Internally, a compiler should keep a mapping between identifiers and values (map<string, string>). The '#using option' syntax allows values to be set, modified, or removed (by setting the value equal to the empty string). Importantly, this is a write-only operation. These values CANNOT be queried in any way, except by the package loader.

So what does the package loader do with these values? At any location within the package directory hierarchy, you may have a directory with a name that matches the following pattern, which we are calling the “package option pattern”:

```
IDENTIFIER=string
```

Having a directory with a name that follows this pattern activates the Special Package Option Rules.

3.5.2.1 Special Package Option Rules

In Section 3.3, we have an entry 4.c.ii, which talks about special package option rules, which we will lay out here.

1.) If the directory name matches a package option patter
    a. Split the directory name on the equal token ('=') into an identifier and a value.
    b. Check the internal option map for the given identifier. If the value exists and matches exactly the value in the map (as set by a '#using option' directive), then recurse into this directory and perform the entire sequence starting with entry 4.a on this directory.

A few examples of directory names that match the package option pattern:

```
ARCHITECTURE=x86-64
ARCHITECTURE=ARM8
OS=Linux
CONFIGURATION=DebugOptimized
```

3.5.3 #using path

The #using path syntax allows the programmer to add files to the File Set. The behavior is as follows:

- If the path referenced is a filename, then that file is inserted into the File Set.
- If the path referenced is a directory, then:
  - All of the files in that directory are inserted into the File Set.
  - If the "recursive" modifier is present, then the directory is scanned recursively, and all files in all subdirectories, recursively, are inserted into the File Set.

This particular functionality is, in some ways, orthogonal to the main specification. The packages feature can be shipped without this syntax, and this syntax can be shipped without the packages feature. However, we believe that the two are sufficiently close together, and work together particularly well, so we present them together.

3.6 MANIFEST File Format

The MANIFEST file is literally called "MANIFEST" with no extension. It is a JSON file, or, more formally, a file that can be opened by any program that conforms to either ECMA-404 or IETF RFC 7159 (whichever one is more palatable to the committee). The MANIFEST file will be used to contain a set of key/value pairs. There are currently a number of keys that are defined by the standard:

- **VERSION** – A string representing the version of the package. This string can be in any format that is meaningful to the package, although there can be some optional restrictions placed upon it if the library author chooses to do so. These will be described later in this document.
- **INCLUDE** – An array of strings representing the default headers from inside of the package to include by default in a translation unit when the package is used.
- **IMPORT** – An array of strings representing the names of modules to import by default in a translation unit when the package is used. The strings must be valid module names.

Here is an example MANIFEST file:

```
{
  "VERSION": "1.2.3-alpha",
  "INCLUDE": ["foo.h", "public/bar.h"],
  "IMPORT": ["foo.containers", "bar.algorithms"]
}
```

3.7 A Note on "Creative Misuse"

There is great potential with all of these new packaging features to replace large portions of existing build systems. By "creatively misusing" the #using package and #using path syntaxes, you can create single files that include all of the contents of your project:

```
// main.cpp
#using package "plugin" version "1.5.8"
#using path "plugins.cpp"
#using path "plugins"
#using path "extensions.cpp"
#using path "extensions" recursive

int main() {
```

```
  // ...
}
```

Now when I get this software package, all I need to do is compile main.cpp and the entirety of the project, spread out over multiple files and directories, will be compiled and linked together with a single invocation of the compiler. All of a sudden the build system can do a lot less work in terms of file management, and a lot more work in terms of build management.

4 A Standing Document

Packages are of no use if you can't use them. People are free to build their packages however they want, and compiler vendors are free to implement this specification however they want. However, if they are given too much freedom, then we will end up in a situation where using packages is a nightmare due to platform or package differences.

There are some pieces of the packaging specification that we need to have, but which we cannot or should not put into the standard. Therefore, we will write, as part of this process, a Standing Document akin to SD-6 (SG10 Feature Test Recommendations). This document will provide best practices both for organizing packages, as well as recommendations for features that compilers should implement. A few examples of the contents of this document:

1. A list of recommended default built-in #using option key/value pairs for architecture and platform. Compilers should implement these according to this spec, and package authors should use them rather than invent their own.
2. A standards-compliant mechanism for package authors to reroute package options for dependent packages that do not follow the recommendations.
3. A pair of #pragmas for defining where to get a package, and how the package versioning should work.
4. A description of how build settings for packages should work.
5. A requirement that builds be consistent with regard to package versions.

Let's examine each of those in turn.

4.1 Built-in #using options

Each compiler should have a number of built-in options that are defaulted for its own configuration and build settings. The idea is that users of libraries should not have to add any #using option directives, and package authors should not have to guess at the correct default options to use. Here is an example of what the key/value pairs would be:

Key	**Available Values**
Platform	Windows, Linux, MacOS X, BSD
Architecture	X86, x86-64, ARM, SPARC, MIPS, POWER8
Compiler	Clang, GCC, ICC, MSVC

We would, of course, solicit inputs from the various partners and come up with a set of options that are amenable to all.

4.2 Rerouting Package Options

Package authors are not obligated to use the standards in this standing document, and you may want to use a package that itself uses a package that has a non-standard option set. As the author of the outer package that itself uses standard options, but which uses an inner package that uses the non-standard option set, how do you translate your standard options into non-standard versions?

The solution is to have one or more header files laid out in option-specific directories, and then include those headers before #using the package. For example, let's say that I'm following the standard in my package and using **Platform=Windows**, but I need to use a package from within my package that uses **OS=Win32** to mean the same thing. I need to map my options to the other options.

So, create a package option directory within my package for my platform, and put a header in there called options.h:

\Platform=Windows\include\options.h

In options.h, I add the non-standard #using option:

```
// options.h
#using option OS = "Win32"
```

Now, when I use my non-standard package, I simply include "options.h" ahead of time:

```
#include "options.h"
#using package "nonstandard-layout"
```

Hopefully, most packages will follow the standard and this trick won't be needed much, but it's there for when people need it.

4.3 Pragmas

We are also recommending (but not requiring) that compilers implement two #pragmas in order to make working with packages that much more awesome.

4.3.1 #pragma package_source

The purpose of the package_source pragma is to allow the system to retrieve packages from somewhere other than the local hard disk. Examples of such locations could be: http, ftp, or a source control repository such as git or subversion. Here is an example of the syntax:

```
#pragma package_source("package", "location", "protocol", "options")
```

The parameters to package_source are:

- **Package** – The name of the package that we're talking about
- **Location** – The location to get the package from. This string is expected to be a URI, as defined by IETF RFC 3986.
- **Protocol** – A string that is matched against an implementation-defined list of available protocols. There is a short list of recommended-supported protocols, but none are required.
- **Options** – An implementation-defined string protocol-specific options. These may be things like git branch/tag, svn revision number, username/password, etc. This string is parsed, and if it is not parsed as either an empty string, or as a valid set of protocol options, then it is an error.

The details of what the compiler does with this option are implementation-defined. However, there are two restrictions:

- If the protocol is not supported by the compiler, it should emit a warning.
- If the compiler doesn't understand one or more of the options for the given protocol, then it should emit a warning.

In general, the compiler will be expected to retrieve the given package from the specified location (and very likely cache it). Here is example usage:

```
#pragma package_source("example", "http://example.com/svn/trunk", \
                       "svn", \
                       "revision=1266; username=guest; password=guest")
```

List of recommended protocols to support:

- http:// and https://
- ftp:// and ftps://
- file://
- Source control-specific providers (such as git, TFS, Perforce, and Subversion)

This list is not canonical, though, and compiler writers are free to not support any of them. It's also probably a good idea for compiler writers to allow these protocols to be extensible through some sort of plugin model, but, again, that won't be a requirement.

4.3.2 #pragma package_version_scheme

Sometimes you don't want to specify the down-and-dirty specific version of a package that you want to use – you just want to specify "I'm using version 1 of this package or any compatible version". If the package is numbered 1.2.3, it should satisfy that constraint, assuming that the package is using a semantic versioning scheme.

So, in order to make it easier to use packages by version number, we are adding one last pragma:

```
#pragma package_version_scheme("package", "scheme")
```

The package_version_scheme pragma enables certain rules for evaluating which version of a package to use, when multiple versions are available. While compiler vendors are free to implement any schemes that they want, there is one required scheme that is defined by the standing document: semantic versioning, as defined by the Semantic Versioning specification version 2.0.0.

When you enable semantic versioning, the version numbers declared in the #using package directive are checked against the semantic version specification and an error is produced if they don't meet the specification, with the following exceptions:

- If a version is listed as "X" or "X.", then it will be as though it were "X.0.0"
- If a version is listed as "X.Y" or "X.Y." then it will be as though it were "X.Y.0"

If semantic versioning is enabled for a package then the compiler, when selecting among multiple available packages, must select the highest-numbered compatible package. For example, if package

versions 1.2.8-beta, 1.2.8, 1.5.66, and 2.1.2 are available, then the following examples indicate which version will be selected:

```
#pragma package_version_scheme("my_package", "semantic")

// All of these will select version 1.5.66
#using package "my_package" version "1"
#using package "my_package" version "1.2"
#using package "my_package" version "1.4.8-dev"

// All of these will select version 2.1.2
#using package "my_package" version "2."
#using package "my_package" version "2.1.1"

// This will generate an error "no compatible package available"
#using package "my_package" version "2.1.3"
```

As stated before, the compiler is allowed to define other versioning schemes, with rules established by the compiler vendor.

4.4 Package Build Settings

When the compiler sees a package, it is ultimately building a set of extra translation units. The standing document will declare that compilers should use an implementation-defined set of build configurations that we will call the compiler's package build flags. This distinction is important, because the compiler settings that I use to build my own code are likely to be different from the compiler settings that I want to use to build third-party code (which packages are generally going to be). In general I will want to compile third-party code with optimizations turned on since I will not be editing it or debugging into it.

The Standing Document will actually make these declarations:

- Package build flags should have optimizations turned on by default.
- Package build flags should be exposed to the developer, in case they want to tweak them by turning optimizations off, or to change other settings.

One great example of why this is important is hashing algorithms, such as MD5 or SHA-1. It can be devastating to my application's performance if my hash functions are built in debug mode. Instead, it should compile the hashing package with optimizations turned on. The same is generally true for any package – in general, I'm not going to be editing the code for a third-party package, and I want it to run as fast as possible.

4.5 Consistent Builds

If you don't specify a version number for your package, then the compiler is free to select any version that it deems appropriate. However, imagine how horrified you would be if you built your program, then did a simple rebuild, and all of a sudden got a completely different version of your dependency, simply because somebody checked in a change into source control, or somebody else compiled a program that brought a newer version of a package into a shared package repository.

That would be bad.

So, the standing document will include a recommendation that compilers use some mechanism to ensure that, if package versions are not specified, that they do not change without informing the programmer.

5 Syntax Discussion

Why are the keywords on the preprocessor, rather than part of the language? Couldn't you have done something like hijack the "using" keyword and have a syntax like 'using package "foo"'?

The C++ language has been moving in the direction of removing the need for the preprocessor, pretty much from day 1. Templates, for example, were initially implemented as a way of creating generic objects without having to use macros. Bjarne Stroustrup has expressed a dislike for the preprocessor in many places, including the C++ Super-FAQ on isocpp.org[3]. While this paper has selected a preprocessor token for its operation, we are also cognizant of the fact that this will not be the most popular choice with the committee.

5.1 Design Perspectives

We can look at this from a number of different perspectives:

1.) As a **customer** of a package, I don't care what the details are. I just want to use it. So if the compiler is doing things to bring in the contents of header files, I don't care, so long as the symbols are available to me and the package is compiled and linked properly.
2.) As an **author** of a package, my customers shouldn't have to care whether I've implemented my package using headers or modules, and they shouldn't need to know which header files to include. They should just be able to use it. This desire is very much in line with the customers' desire.
3.) As a **compiler developer**, whatever semantics we come up with must be implementable (c.f. the "export" keyword fiasco). Furthermore, they must not place a sufficient burden on the development of the compiler that it hinders features. However, it is also true that "All Things Are Possible"[4], and that the most complexity can be dealt with elegantly.
4.) As a **language author**, I want the design to be consistent and explainable under the current rules.

Clearly, we have a number of different factors here. And we haven't even talked about one of the biggest motivators yet: the C language. Packages are a very exciting feature, and we would like to make sure that we build semantics and syntax that are compatible with WG14.

There are four ways that we can go with this feature:

1.) Packages are only for modules – We can declare that packages are effectively dependent on Modules, and that packages are the vehicle by which modules are delivered. There is no way to use headers under this model.
2.) #using as a preprocessing directive – This is the specification as this document describes it. The preprocessor does the work in this case.
3.) using keyword (without '#') that can do preprocessing magic – We can simply remove the octothorpe ('#') from the keyword, but retain the semantics. This has implications for compiler complexity.

[3] https://isocpp.org/wiki/faq/newbie#preprocessor-is-evil
[4] Reference: me, every time a sound designer comes to me asking for a new feature.

4.) **Separate syntax for header packages** – Packages are packages, but we must access their contents using a different syntax depending on whether the package is authored using modules or headers.

Let's examine each of these from the perspectives of the various stakeholders.

5.2 Packages Only for Modules

In this option, the syntax is:

```
using package "package" [module] version "abc"
using package_option IDENTIFIER = "value"
using path "foo" recursive
```

Headers are not supported, so the syntax for defining them is removed. The module list is optional, and it is allowed to be empty.

Stakeholder	Effect
Package Customer	When using a package, there is no difference to me, because I just have to add a 'using package' directive to my code. However, there will be fewer packages available at the beginning of the process, because there are no modules currently in existence. Bootstrapping is, therefore, problematic, and it may drive developers to use non-standard mechanisms simply because a library doesn't have modules implemented.
Package Author	I have to implement my library as a module. Hopefully, this will be a small delta, but it is still a burden on me, especially if I am taking a large existing library and porting it. It requires a lot of extra up-front work to publish my existing library as a package. However, once that work is done, it's done. If I'm writing a new package, I must use modules, even if I would prefer not to. Also, if I am taking third-party source and assembling a package from it, then that package is not going to be usable unless I write a module wrapper for it.
Compiler Developer	No problems here. The semantics described in this paper simply move into the translation phase instead of the preprocessing phase.
Language Author	Again, no problem. And, as a bonus, there is no preprocessor voodoo.
WG14	This is problematic. The C language has no modules, which means that C programs cannot use packages at all. If, however, WG14 gets its own compatible module specification, then they could adopt packages at the same time. There is also the problem of keywords: C doesn't currently have a "using" keyword, so we would either have to introduce it to C, or come up with a keyword that both C and C++ accept.

Basically, this option is certainly very clean from the perspective of C++, but it has two problems:

1.) It is a big hindrance to adoption. Packages must be modules, so the modules specification has to be standardized and broadly implemented in order to use this. Existing packages have to be converted or wrapped.
2.) There is an impediment to WG14 accepting this, because they do not currently have module support. Therefore, modules will have to be ported to C before packages can be brought to them.

5.3 #using Syntax

This is the syntax as defined in section 4 of this document:

```
#using package "foo" ("header.h") [module] version "abc"
#using option IDENTIFIER="value"
#using path "foo" recursive
```

Stakeholder	Effect
Package Customer	I can just add a '#using package' directive to my code to start using a package. I don't have to do anything special. There should be plenty of packages available to me, because the syntax supports both headers and modules.
Package Author	If my package is already in the right directory layout, I can simply zip it up. If not, then it is either a simple one-time process to reorganize it, or I can write a script to assemble a package.
Compiler Developer	No problems here. This is exactly the semantics described in this paper. All of the work is done during the preprocessing phase.
Language Author	Technically, there is no problem, because we're just adding a new preprocessor directive to Section 16. However, it's also unfortunate, because this is all preprocessor magic, which the committee is trying to eliminate.
WG14	Syntactically, there should not be any problems. C already uses the preprocessor, so we're just adding another preprocessing directive. Obviously if a package contains a module, then a C program can't use it (unless the C language adopts a compatible module specification, and also if the package is written using 'extern "C"').

This option is described in great detail in this document. It's fairly clean, although it does live in the preprocessor, which we are trying to avoid.

5.4 'using' with Preprocessing Magic

The syntax here is identical to 6.3, except without the octothorpe:

```
using package "foo" ("header.h") [module] version "abc"
using option IDENTIFIER="value"
using path "foo" recursive
```

Stakeholder	Effect
Package Customer	From my perspective, there is no difference between this syntax and the #using syntax. It's just a matter of whether or not I have an octothorpe. I can use any package with a single line of code.
Package Author	Again, there is no difference from my perspective between this and the #using syntax.
Compiler Developer	Here we have some complexity. We are injecting the preprocessor into the middle of the translation step, which requires extra work beyond the basic semantics. The preprocessor "engages in a tight little dance with the lexer"[5], and that dance is now going to be far more complicated.
Language Author	It is counterintuitive to me that any preprocessing should happen outside of the preprocessor step. Section 2.2 of the standard talks about the stages of compilation, and all of a sudden in the middle of step 7, we're recursing back to step 1 for a portion of the file under one very specific circumstance. Also, there is verbiage in Section 16 about the preprocessor, and it is made very clear that the octothorpe ('#') is the indicator for a preprocessor directive.

[5] From the clang source code, Preprocessor.h

WG14	C does not have the 'using' keyword, so either we will have to get WG14 to embrace the new keyword, or we will have to come up with some syntax that uses a keyword that is common to both languages.

This option is every bit as clean as '#using' from the perspective of the package users and authors, but it is far more contentious from the perspective of the compiler developers and the language specification.

5.5 Separate Syntax for Headers and Modules

We will use '#include package' as a straw-man for this separate syntax in this section. We recommend against having '#using' and 'using' in this option because they are too similar.

```
#include package "foo" ("header.h") version "abc"
using package "foo" [module] version "abc"
#include package_option IDENTIFIER=value
using package_option IDENTIFIER=value
#include path "foo" recursive
using path "foo" recursive
```

Stakeholder	Effect
Package Customer	As a customer, this is the most complex of the options. The complexity is not big, but it is mysterious and arbitrary. If a package is written using headers, I use '#include package', and if it is written using modules, then I use 'using package'. If it has both headers and modules, then I must have both '#include package' and 'using package'. Furthermore, I have to know how the package was authored in order to know which syntax to use. This creates opportunities for confusion ("why didn't using this package work?") and frustration ("if the compiler knows enough to give me an error message that I used the wrong syntax, why couldn't it do the right thing?").
Package Author	If I have a library that is packaged using headers, I may be reluctant to convert it to use modules because then all of my customers will have to change their code. There is very little that I have to do different from the other options other than to document the appropriate method for using this package.
Compiler Developer	There is some complexity because now we have two different syntaxes to support. However, both of those syntaxes will use the same back-end data structures, so the complexity is mostly on the front-end.
Language Author	While I am unhappy about adding anything to the preprocessor, I am more pleased that the preprocessor magic is confined to the C way of doing things. For C++, we have just C++ things (modules), and the changes to the preprocessor are just there to support the C language.
WG14	When we bring this to WG14, we just have to remove the module stuff and keep the '#include package' syntax – a clean slice. New syntax for the #include keyword, but no new keywords otherwise.

This option is the cleanest from the language and compiler perspective, but it can cause some end-user confusion, and very likely a slower adoption of modules.

5.6 Alternate Syntax Summary

Each option has benefits for some subset of the stakeholders and problems for others. However, no matter which option is chosen, the semantics of packages remain the same: translation units are added

to the File Set, headers are included, modules are imported, and the end-user has to do nothing more than add a line of code in order to use the package. It's all a matter of how it is spelled.

6 Goals and Principles

In building this system, we have a number of goals that we want to accomplish and non-goals that we want to avoid. It is important to be explicit about them, so that the design choices that we have made can be evaluated against the goals.

Goal: Standard Package Layout

Packages are useless if different compiler vendors have different standards. If Clang uses a different package layout than Microsoft Visual Studio, then it's a losing situation for everybody. Library authors have to author two different packages using two different toolchains, and library users risk getting the wrong package format.

We already live in this world. There are a million different distributions of Boost (other than the canonical source-only distribution, of course), one for each compiler/standard library/compiler setting combination. This is one of the drivers for putting things into the standard library, so that implementations will be guaranteed to be available for everybody. But, of course, this doesn't scale well to large numbers of libraries, or to libraries that genuinely shouldn't be in the standard.

The only way around this problem is to require that packages strictly adhere to a particular layout. This layout can also be contained within a file, and we must therefore select a file format that is standardized, and which has all of the features that we require.

Goal: Source-Code Packages

We only want to standardize how the compiler interacts with source code in a package. The reason for this is multi-fold. First of all, having a standard for pre-built or binary object files or libraries would require that we standardize on file formats for these, which is a non-starter due to the differences among the various operating systems vendors. Also, it would discourage innovation, since the format would be fixed. Finally, including requirements for binary files would delve into the territory of compiler settings, and ensuring that the built binaries are compatible with the source that you are building[6].

Therefore, C++ packages are all about distributing source code and including it in your build.

Goal: Minimal New Syntax

We need the minimum amount of new language features to build real packages of real code. We should, not shy away from new syntax where we need it, but we should also take care that the syntax we are including solves the real problems that we have. New syntax should be easy to write, easy to read, and as close to "do what I mean" as possible.

Goal: Build Real Packages

[6] Settings such as Visual Studio's /MD, /MT, /MDd, and /MTd control whether the standard library is linked statically or dynamically, and whether or not the debug version is used. Similarly, for GCC and Clang, -fPIC forces generation of position-independent code. These settings must match exactly among all built libraries, or you can have serious problems.

Real Code™ is big and complex. Real Code™ has thousands of files, hundreds of headers, and millions of lines of code. More importantly, Real Code™ is not all C++. Technically, even C is another language. C++ Packages should allow Real Code™ to be built and included with C++ programs.

Of course, C++ cannot dictate what other languages do or how they work, but we should have provisions for building packages that include source written in other languages.

Goal: Interoperate with C

C++ is, for the most part, a superset of C. We should embrace that, and build a specification that the C language can adopt. Or, more precisely, where we can quite trivially draw a line and give everything on one side of the line to C without loss of generality.

Goal: Allow Package Users to Specify Versions

Each program should be able to specify which version of a particular package it is asking for. This is important, because if I depend on "whatever version I have installed", then I will have two C++ programs that each want a different incompatible major version of a particular package, and then you will have to juggle the dependency yourself.

The idea of what is a "version" is fluid, and can refer to anything from a proper Semantic Version to a git checkin hash or a branch/tag combination. No matter what a version is, we should be able to specify it.

Goal: Allow Packages to Themselves Use Packages

If I'm writing a C++ package to be used by others, I should be able to use packages as well. This feature is for all users of C++, not just end-users. This means that packages need to work recursively, and that there is some way to disambiguate incompatible package versions. If I'm included package A and package B version 1, and package B includes package A version 2, there needs to be some way to disambiguate which version I should actually use.

Goal: Work with Both the Preprocessor and the Module System

Packages are about getting code into your project and using it. We don't want to mess with the mechanics of the preprocessor, and we don't want to mess with the upcoming module system.

Non-goal: Disallow Non-Source Packages

While source packages are a good default position, we recognize that this is not possible or desirable in some cases. Some vendors are not in a position to (or do not wish to) ship their libraries in source format, but there is still tremendous value in enabling them to use the package system to deliver their libraries in a uniform way. We do not want to place any requirements on the behavior of the compiler in the face of binary files, and we also do not want to disallow vendors from shipping implementations that support precompiled packages.

Non-Goal: Implement a Build System

If we take this feature too far, we're in danger of implementing an entire build system. That's far too much. We want to enable people to use library packages easily, and to make it easy to build their own packages.

7 Interim Paths and the Glorious Future

How real is this feature? It's all good and fine to write words on the page, but if there's no implementation, then it's not really real. Is this implementable?

7.1 Clang Implementation

We have been working on the Clang compiler to implement a working proof-of-concept of this feature. It is available at https://github.com/Blizzard/clang. All indications are that this feature is imminently implementable with a relatively small amount of work. After all, we already live in a world where the compiler has to build multiple translation units in sequence, and then link them all together. This is not changing that model – it's just generating the list of files to compile in a different way.

One of the things that is required for this feature to be truly meaningful is for there to be a critical mass of packages already available. But now we have a chicken-and-egg problem. The feature needs to get into the standard and the compiler vendors need to implement the feature before it can be truly widely used. But, if there's no compiler support, then there is no incentive for people to make standard packages.

7.2 The cppget Package Tool

What we need in order for package authors to generate packages in the standard format is a way for users to use these packages, even if their compiler doesn't support it yet. Enter the `cppget` tool, which is a standalone utility used to acquire, install, and integrate standard C++ packages. It is meant to be used manually by users; it can be scripted by make systems, but in truth make systems should directly incorporate C++ package support once compilers support it following the standard.

By default, `cppget` is used in a very straightforward way, with no command-line options. You run `cppget` in the directory containing your top-level project (a solution file for Visual Studio, an xcodeproj for XCode, and so on), with the name of a package you want installed.

```
> cppget google_protobuf
Downloading google_protobuf
Installing vesion 3.0.0-beta-1
```

The `cppget` tool has the address of the main package repositories hard-coded into it. It will look up the identifier `google_protobuf`, pick the most recent release version, install it to a directory at this level, create a project file for it appropriate for your build system, and update your main project (solution, makefile, etc.) to add it.

The auto-guessing works for small projects. For large projects, the project maintainer can add a `cppget.config` file. This file contains instructions on where to place downloaded projects, what change to make to generated project files (targets, etc.), what top-level project files to update (there may be several, especially in a cross-platform project), etc. This would also be how you handle cross-compiling – e.g. building Android projects or iOS projects, since auto-configuration based on the current system would be inadequate. Also, the `cppget.config` file could point to more or different package repositories.

This tool is expected to be interim; once we have C++ package support in the standard, we expect that only a very few projects will use the cppget tool. Likely only large projects that are building on systems without IDEs that offer equivalent functionality.

The cppget tool will be available at https://github.com/Blizzard/cppget.

7.3 The Glorious Future

Once C++ packages are standardized and there is compiler support for them, people still need to get their packages from somewhere. Other languages have centralized package repositories:

Language	Package Repository
TeX	CTAN
Perl	CPAN
Python	PyPI
Ruby	Ruby Gems
Node.js JavaScript	Npm

Each of these repositories has its own strengths and weaknesses in terms of its management and curation policies. The details of those policies are outside the scope of this document.

However, one thing that should happen is that we should have a centralized, curated repository of C++ packages that people can assume will be there. Imagine if I didn't have to go looking all over the Internet for this package or that – any package that I'm 90% likely to want or need is already in the default public repository. And, of course, I can always get packages from my own company's repository, or from source control.

It's a glorious future.

8 Questions (And Some Answers)

There are many ways to implement a packaging system. Plenty of other languages have built-in systems, each with their own ideas and concepts and backwards-compatibility burdens. Let's examine some of the decisions that we made and some of the alternatives.

Why Do This in the First Place? Why bother? What's wrong with what we have now? After all, C++ is popular, getting more popular, and people are already able to use packages.

All true, but just because the status quo is "fine" doesn't mean that we couldn't shake it up to make the world better. Yes, we can already use packages, but it is a tedious and manual process. As we just noted, acquiring a new library is error prone and time consuming. We want to make using C++ a joy, and consuming C++ libraries as easy as humanly possible.

Don't We Already Have This? After all, there are already packages out there for C++ code. Red Hat has their distributions, as do Debian and Microsoft. And you can find precompiled Boost binaries pretty easily.

Yes, these things are true, but they're not a solution to the problem, they're actually a part of the problem. Why is it that I have to go out of my way to get Boost binaries? Why can't the subset of Boost that I use just compile as part of my build process? And what if my compile settings aren't compatible with the prebuilt packages that are distributed? Obviously, I can build my own Boost, but now all of that packaging work is wasted.

Similarly, it is a big waste of manpower for Debian and Red Hat to package libraries. They're each packaging the same content in slightly different ways, for a different subset of developers. C++ is, by its nature, a cross-platform language – not tied to a specific version of an operating system family. By depending on operating system vendors to package these libraries, we're they're introducing problems

where we wouldn't ordinarily have them, because they're creating variation axes based on host operating system distribution, rather than based on the contents of the package. The best way to distribute these packages is to have a single canonical distribution that all compilers must adhere to.

Why Do This in the Compiler? Doesn't this feature belong in the build system? After all, the whole point of the build system is to manage which files I'm compiling, and to organize them. Shouldn't Visual Studio and CMake and Premake solve these things for me?

The question actually is its own answer. CMake can't, won't, and shouldn't use the same package format as Visual Studio. They're just very different beasts. Ditto any combination of build systems. Plus, any other newer better systems that appear in the future. By putting this into the compiler, into the C++ standard, we are creating a single universal specification that ALL build systems are required to understand and use.

Now I can choose whichever build system meets my personal needs best, without having to worry about whether there is a package in the specific format that I need.

Also, many C++ users cross-compile for other platforms, and end up having to implement various kludges in order to get the appropriate header/source/library file for the appropriate build platform. Having packages built into the compiler would make these kludges go away, because the files are imported in a uniform fashion.

What's the difference between modules and packages?

Packages are about getting code into your executable across multiple translation units. Modules are about importing symbols into your existing translation unit. Packages have a standard defined file format. Modules are a more abstract concept, and the file format is a compiler-dependent implementation detail.

While the Modules and Packages specifications are orthogonal, the best places for modules to live are inside of packages.

Isn't it going to make compiling more expensive and time-consuming to use packages?

That's, ultimately, a tooling issue. It should be a pretty quick and trivial check for the compiler to determine that it's already acquired and/or built a package and not do so again.

This seems like a radical change. Shouldn't this be outside of the standard?

In fact, the change is relatively small. It's a small amount of new syntax, and some very simple rules for processing. Crucially, we are not proposing anything here that compilers *are not already doing*. Compilers already keep track of header file paths. Compilers already keep track of a list of files to compile. Compilers already perform checks to see if they've already built a file. All we are doing here is codifying this practice, and embracing the codification in order to enable both features and community.

The question of whether this should be in the standard is one we have grappled with. Ultimately, we have to ask: if this doesn't go into the C++ standard, where should it go? We could create a secondary specification, but without built-in language support, what good is it? Each build-system would have to implement support itself, and then we have to duplicate the work for every single build system.

No, there's no better place for this work than the C++ standard.

What about <language-feature>? How will that work with packages?

That's the nice thing about packages. They don't change the semantics of the language at all. If a language feature already works, it will continue to work in the same fashion. Remember: all we're doing is adding a File Set. We're still building C++ translation units and linking them together just like we already were before – it's just where some subset of those files come from is different now.

Why did we choose zip files? And why did we have to pick a file format to begin with? Couldn't we just have left it implementation-defined or unspecified?

We must have a standard layout for packages; without one, the entire feature is neutered and ineffectual. Without the ability to store and transfer packages as single files, the feature is less useful. People will be far more likely to share, distribute, and consume libraries if packages are easy to get.

Ultimately, the C++ community will decide whether folders or zip files are the better option, but we want to make sure that the option is there. We selected the zip format as a matter of convenience. Zip files are an ISO standard, just about every operating system under the sun can open them, there are liberally-licensed open-source libraries for reading them, and they support directory hierarchies and multiple files.

There are plenty of ways in which directories could be laid out. Why did we choose this one?

It comes down to two things. First, the layout that we have described here is sufficiently common that it is either already how many projects lay their files out, or it is a small and unobjectionable change to do so. Also, even if package authors want a different layout in their source, they can construct the package zip file according to the spec. Second, each directory matches a specific operation that the algorithm for parsing packages performs. The 'include' directories are added to the search path, and the 'source' directories are added recursively to the File Set.

Why isn't there more information in the manifest? Couldn't we have organized the directories in an arbitrary fashion, and placed a mapping of some sort into the MANIFEST file? Why did we pick JSON instead of XML or ini files or some custom format?

The more work you have to do in order to create a package, the less likely you are to a) make a package in the first place, and b) get it right while doing so. We tried very hard to create a design that did not require any sort of manifest at all, but were ultimately forced to, because otherwise there is no place to put the version number or the default headers and modules.

We certainly could have allowed arbitrary organization, with a mapping defined in the MANIFEST file. However, that would have been contrary to the simplicity of the current system, and would have required that a MANIFEST file exist. The data structures would have been far more complicated, and packages would be more complicated to use and to examine. Parsing a package would require more code in the compiler, be more likely to have bugs, and be slower. The simpler the better.

We picked JSON for convenience. JSON is both an ECMA standard and an IETF RFC, although it is not an ISO standard, which is unfortunate. JSON is easy for humans to read and easy for machines to parse. It is also significantly less verbose than XML. We could have specified our own file format, but that's a gigantic load of verbiage and specification just so that we can have a place to put a single standardized key/value pair. Also, JSON is, by definition, extensible, so if compiler vendors want to put more features into their MANIFEST files, then they have a very expressive language with which they can do so.

What if I want to have a filter for the files in my path for a '#using path' directive? Can I just include a subset of the files in my application's source directory?

According to this specification, no. However, if this feature is desired, then we can make the string parameter to #using package be a regular expression. That way you can write a regular expression that defines which files you want to add, and the compiler will select only files that match the expression.

Why do the version semantics look the way that they do? Why semantic versions? Why bother solving the version problem to begin with? After all, other languages (like Go and Python) haven't solved it, and they're doing just fine.

Go and Python chose to ignore the versioning issue in slightly different ways, and in both cases, the community came together and solved it on their own. In python's case, there is a non-standard but common __version__ attribute which the community has standardized upon. In Go's case, there are entirely separate toolchains that solve the versioning problem.

And in both cases, the solution is not perfect. For Python, you can end up with libraries that don't have the __version__ attribute, which means that you can't query their version. Also, the Python environment is global, which means that you can't have different versions of the package for different programs. For Go, you have to both use a non-default toolchain *and* manage the version of the dependencies yourself.

So how did we end up with the version semantics that we have? We don't want to require a specific version scheme to be the "one true standard". That is, each package maintainer can come up with their own scheme, whatever makes sense for their distribution. We only require that the version be expressible as a string.

Without any version scheme, we cannot understand the contents of a version string – all we can do is check it character-for-character against the package that we have. If they don't match, we can tell you that, and we can show you the different version strings, but we can't fix it for you or do anything about it. Even if it's quote-unquote "obvious" how to fix it.

So that's where version schemes come in in the standing document. We want to be able to put some intelligence into the version number, and allow the compiler vendors to define rules for sensible version schemes so that they can help you to both use a package, and to use the best version of a package. We require semantic versioning, because it is one of the most common version schemes around, and it has simple, sensible, easy-to-implement rules for how it operates.

Let's say that I have package versions 1.2.8, 1.2.10, and 1.5.12. I want the latest version of 1.2, but I don't care which one it is. Importantly, I don't want version any version that is later than 1.2 (such as 1.5.12). How can I use semantic versioning to make that happen?

Unfortunately, the rules of semantic versioning don't allow this sort of scheme. However, compilers are free to implement a "semantic-minor " version scheme that implements these semantics. If there is enough desire for this, we can add it to the standing document.

Why have options in the first place? Why implement options as a simple key/value pair? Why are they write-only? Why not use the preprocessor?

We needed to have options so that a single package could target different outputs. Common axes are platform, architecture, and compile target. For example, you may have slightly different logging code for Windows vs. Linux. Similarly, your logging may be more verbose in Debug vs. Release modes. You may have architecture-specific optimizations for x86-64 vs. ARM. There are no standard axes, but package authors need to be able to allow the customers to configure their targets without having to author a million packages.

Imagine if we didn't have package option settings. The only way to create customizations would be to create one extra package for each combination of settings. Now my #usings become very complicated:

```
#using package "mypackage"
#if PLATFORM == WINDOWS
  #using package "mypackage.windows"
  #if BUILD == DEBUG
    #using package "mypackage.windows.debug"
  #else
    #using package "mypackage.windows.release"
  #endif
#elif PLATFORM == LINUX
  #using package "mypackage.linux"
  #if BUILD == DEBUG
    #using package "mypackage.linux.debug"
  #else
    #using package "mypackage.linux.release"
  #endif
#else
  #error Unknown platform
#endif
```

Now compare that to the equivalent code with options:

```
#if PLATFORM == WINDOWS
  #using option PLATFORM = "Win32"
#elif PLATFORM == LINUX
  #using option PLATFORM = "Linux"
#else
  #error Unknown platform
#endif

#if BUILD == DEBUG
  #using option TARGET = "Debug"
#else
  #using option TARGET = "Release"
#endif

#using package "mypackage"
```

The latter is much clearer and more extensible.

As for why they are write-only: what would be the benefit of reading them? What decisions could you make? The purpose of these keys is to configure which contents of a package are brought in. If you need

to make decisions in your code, use the preprocessor! The only legitimate use case for reading these values is to translate them into equivalent options for other packages. The mechanism for doing so is described in section 6.2.

Speaking of which, why not use the preprocessor to implement options? The package reader does run as part of the preprocessor, which means that it will have access to all of the preprocessor symbols. The answer here is twofold: we don't want to pollute the namespace of the preprocessor with symbols that are only used for the packaging system, and we don't want to accidentally overwrite any existing preprocessor symbols. Also, we don't want to accidentally cause troubles by overwriting built-in symbols.

What is the purpose of getting packages from external systems? Why is there both a location URL and a protocol?

The purpose of the package_source() pragma is to allow you to include a package from an arbitrary location. That's just double-speak, though, for "we want to get packages directly from github". If a package author has authored their source in the package format, then you should be able to just check it out and use it directly and immediately.

URLs, by definition, have a protocol. However, the protocol of the URL is not necessarily the language that you speak to get the package. For example, most source control systems allow you to access their repositories by http, for example. For git, the URL will typically end with a ".git" file extension, but this is not true for Subversion (for example). So, the protocol field actually tells the compiler what language it should speak, and the location tells it where to go to get the contents of the package.

Who will host the central package repository of the Glorious Future?

An excellent question. We hope to speak with people about it, but we don't have an answer for this.

What about data integrity? If I grab a package, how do I know it's correct and uncorrupted?

That's a good question. We could add some sort of digital signature to the package, but to some degree, this is more of a feature of the Glorious Future from section 6.2. The centralized, curated package repository will include some sort of facility for digital signatures, and the protocol when communicating with the repository will, similarly, have some data integrity checks.

Will packages help me to ensure that a library has a license that is compatible with my program?

This would be a neat feature! The compiler could, for example, be given knowledge of a number of licenses (GPL, LGPL, BSD, MIT, Commercial, etc.). If packages embed that information into the MANIFEST file, then the compiler could perform a check to make sure that the licenses are compatible.

9 Further Bikeshedding

Obviously there's plenty of room for bikeshedding here. I'm going to present a few options that I come up with off the top of my head, but these lists are by no means complete, and we are open to other spellings.

- **#using** –#package, #import, #use, or we could hijack #include
- **#using package** –#package use, #using pkg
- **#using option** –#package option, #using opt

- **#using path –** #using import, #using add, #file add, #path add
- **package_source** –pkg_src
- **package_version_scheme** – scheme, version_type, package_version_type
- **MANIFEST** – manifest (lowercase), package.manifest, pkg.man, package, c++.manifest, cpp.manifest
- **include (directory inside of a package)** – inc, headers, header, hfiles
- **source (directory inside of a package)** – src, cpp, cppfiles

And, of course, plenty more where that came from.

A Standardese

Here is a first pass at a formal syntax for the preprocessor #using directive:

pp-using-directive:

using *pp-using-command*

pp-using-command:

pp-using-package-command

pp-using-option-command

pp-using-path-command

pp-using-package-command:

package *preprocessing-token header-list*$_{opt}$ *module-list*$_{opt}$ *package-version-selector*$_{opt}$

header-list:

(*header-list-entry*$_{opt}$)

header-list-entry:

preprocessing-token

preprocessing-token, header-list-entry

module-list:

[*module-list-entry*$_{opt}$]

module-list-entry:

module-name

module-name, module-list-entry

package-version-selector:

version *preprocessing-token*

pp-using-option-command:

option *identifier* = *preprocessing-token*

pp-using-path-command:

path *preprocessing-token using-path-recursive*$_{opt}$

using-path-recursive:

recursive

Other than that first-pass attempt, this document doesn't have any standardese in it. However, we will note a few places that will probably need to get modified:

- Section 1.2 Normative References. We're adding a few new normative references.
- Section 2.2 Phases of Translation. We're going to change this slightly to include the File Set.
- Section 3.5 Program and Linkage. Update this to describe package contents.
- Section 16. Preprocessing Directives. This is where all of the package stuff will live.
- Annex A. Update to include #using syntax.
- Annex B. Update to include any quantities used by packages.

And, of course, we will have to add a new top-level section talking about packages.

B Interoperating with WG14

We have taken great care in developing this standard that it should work for C as well as C++. We hope that once C++ packages become standard, then WG14 will embrace them as well. As a result, we must be cognizant of our syntax choices

C Acknowledgements

We would like to thank the following people for their help and assistance in putting together this paper:

Michael Wong for early feedback, and for helping us to navigate the C++ standardization process.

Bjarne Stroustrup for early feedback, and for creating C++ in the first place.

Ben Deane for valuable thoughts as we iterated on the design.

Matt Versluys for authoring early packaging systems used at Blizzard, and for suggesting that we do this at all.

The Authors of the Module Specification (N4465) for providing me with a good paper to model.

D References

D.1 Papers and Presentations

A C++ library wish list
https://github.com/boostcon/2008_presentations/raw/master/wed/Keynote_LibrayWishList.pdf

A Survey of Programming Language Packaging Systems
https://neurocline.github.io/papers/survey-of-programming-language-packaging-systems.html

D.2 Normative References

ISO/IEC 21320-1:2015 Information technology -- Document Container File (Zip files)
http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=60101

ECMA-404 The JSON Data Interchange Format
http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

IETF RFC 7159 The JavaScript Object Notation (JSON) Data Interchange Format
https://tools.ietf.org/html/rfc7159

IETF RFC 3986 Uniform Resource Identifier (URI): Generic Syntax
http://tools.ietf.org/html/rfc3986

Semantic Versioning 2.0.0
http://semver.org/spec/v2.0.0.html

D.3 Existing Packaging Systems

Language	Package System Name	URL
TeX	CTAN	https://www.ctan.org/?lang=en
Perl	CPAN	http://www.cpan.org/
Python	PyPI	https://pypi.python.org/pypi
Ruby	RubyGems	https://rubygems.org/
Java	Maven	http://mvnrepository.com/
JavaScript (Node.js)	NPM	https://www.npmjs.com/
.Net, WiX, C++	NuGet	https://www.nuget.org/
C++	Biicode	https://www.biicode.com/
C++, Go	Conan	https://www.conan.io/
C++	Build2	https://build2.org/