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.