# Relevant Information

Info that isn’t specific to any implementation strategy

## C’s Foreign Function Interface and ABI

<https://www.p6r.com/articles/2014/07/20/binary-compatible-c-interfaces/>

A foreign function interface is an interface designed to interact with other languages. .Net’s P/Invoke functions and python’s ctypes module are two examples of a foreign function interface specifically for C in practice. C++ *extern ‘C’* attribute serves as its foreign function interface for external C code, which essentially uses C conventions. This is outlined in the [C++ Core Guidelines I.26: **If you want a cross-compiler ABI, use a C-style subset**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-abi)**.**

[This](https://www.p6r.com/articles/2014/07/20/binary-compatible-c-interfaces/#footnote_plugin_reference_11) is a fantastic article with sources for C++ binary compatibility. Here’s a brief summary:

*“One of C++’s greatest shortcomings (at least with regards to interoperability) is the lack of an Application Binary Interface (ABI) as part of the standard. C++ is a language specification after all, and many would argue that is out of the scope of the standard to define specific implementations.* ***The result is that binaries produced with one compiler, are not necessarily compatible with those produced by another compiler (or even by different versions of the same compiler!)****. This simple fact greatly hampers our ability as developers to easily create C++ libraries for distribution to a broad audience. This leaves developers with only two distribution options: (1) provide source for their libraries, or (2) provide multiple binaries, one for each version of each compiler that a customer might use. The matrix for that can get out of hand quickly.*

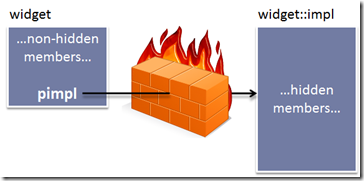
*A solution to this is to attempt to export all public APIs using a mechanism that creates a stable and cross-compiler compatible binary interface. Doing this does come with compromises, however you will have to decide if they are worth it or not for your specific situation*“

The following can differ between compiler implementations

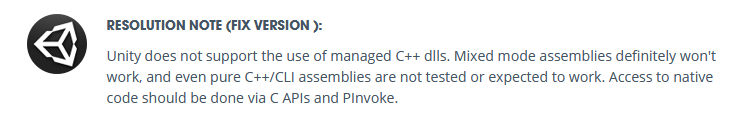
1. Runtime type information
2. Object Layout
3. Non-POD Types
4. Virtual Methods
5. Memory Allocation / Heap Implementation
6. Standard library implementation
7. Resource utilization
8. Padding
9. Exception handling
10. Name mangling

The standard even acknowledges the lack of an ABI by suggesting that the PIMPL idiom is used n these cases, covered in the C++ ISO’s Core Guidelines section I.27: **For stable library ABI, consider the Pimpl idiom, a.k.a. A** [**Compilation Firewall**](https://herbsutter.com/gotw/_100/).

[*“Because private data members participate in class layout and private member functions participate in overload resolution, changes to those implementation details require recompilation of all users of a class that uses them. A non-polymorphic interface class holding a pointer to implementation (Pimpl) can isolate the users of a class from changes in its implementation at the cost of an indirection.”*](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-pimpl)

[](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-pimpl)

## Unity Support for C++/CLI



[Unity Issue Tracker :](https://issuetracker.unity3d.com/issues/unity-editor-silently-crashes-when-calling-a-function-inside-a-c-plus-plus-dll?_ga=2.266736262.909723550.1580761060-1853579396.1576616414) **[Unity Editor silently crashes when calling a function inside a C++ DLL](https://issuetracker.unity3d.com/issues/unity-editor-silently-crashes-when-calling-a-function-inside-a-c-plus-plus-dll?_ga=2.266736262.909723550.1580761060-1853579396.1576616414)**

Unity does not support the modern usage [Mixed Assemblies](https://docs.microsoft.com/en-us/cpp/dotnet/mixed-native-and-managed-assemblies?view=vs-2019), leaving [PInvoke](https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke) with a C-Style interface as the only option for communication between Unity and our external C++ codebase. Mono, the version of the .Net Framework Unity uses, has an [exhaustive albeit outdated summary of PInvoke](https://www.mono-project.com/docs/advanced/pinvoke/) on their website. Python may have a variety of options, the Unity interface must be strictly C-Style with a wrapper on the unity side to handle m[arshalling](https://docs.microsoft.com/en-us/dotnet/standard/native-interop/type-marshaling) between managed and native types. This is similar to using ctypes in python to interact with c-style interfaces.

## Static Libraries

C++ compiles code down to .obj files. These .obj files are then linked together during the linking compile process into a usable DLL or exe. [**Static libraries are simply collections of .obj files with indexing**, making them “...no different from no different than executables generated from individual source or object files”](https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-creating-libraries) *- Intel C++ Compiler reference*.

This means that you have *zero* performance cost from using a static library, however if the consuming program uses a different version of the compiler than the compiled binary, then errors may occur that cannot be caught at compile time *unless a C-Style Interface is used*.

[*“If you have an object file that has external symbols with C++ linkage, that object file may not link correctly with object files produced by a different major version of the toolset. There are many possible outcomes: the link may fail entirely (for example, if name decoration changed). The link may succeed, and things may not work at runtime (for example, if type layout changed). Or in many cases, things may happen to work and nothing will go wrong.”*](https://docs.microsoft.com/en-us/cpp/porting/overview-of-potential-upgrade-issues-visual-cpp?view=vs-2019) *- Microsoft MSDN*

[*When your module[dll] calls new or malloc, the memory can only be freed by your module calling delete or free. If another module calls delete or free, that will use the C/C++ runtime of* ***that other module*** *which is not the same as yours.*](https://devblogs.microsoft.com/oldnewthing/20060915-04/?p=29723) *- MSDN Raymond Chen*

## Link Time vs Compile Time Dependencies

*“Our task is to devise a physical structure (set of files) for the program that represents the logical components in a consistent, comprehensible, and flexible manner. In particular, we aim for a clean separation of interfaces (e.g.,function declarations) and implementations (e.g., function definitions).* ***A file is the traditional unit of storage (in a file system) and the traditional unit of compilation****.” - Bjarne Stroustrup* The C++ Programing Language 4th Edition

In C++ , there are two steps to building a program: Compiling, and Linking. During Compilation, all cpp files that have been changed since last build are compiled into .o files. Changes to any headers used by the cpp file will also cause a recompile, since during pre-processing headers are appended to the top of the cpp’s code. Changes within one CPP will not trigger a recompile within another CPP, and multiple techniques such as forward declarations can be used to avoid including headers in headers to break this compile time dependency.

The second phase, linking, is where the final product of the code is created, be it a DLL, EXE, or static library. During link time, the linker checks for functions defined within each translation unit that have no matching definition, then locates and maps the definition within another translation unit. Generally when changes are made to cpp files, the entire program will be relinked, however several compilers have settings such as microsoft’s [**/Incremental**](https://docs.microsoft.com/en-us/cpp/build/reference/incremental-link-incrementally?view=vs-2019)switch that allow a program to be partially relinked as needed.

*Implicitly Linked DLLs* are linked to programs during link time using .lib files, which can between builds.

## Zero Build Time Dependencies

We seperate every chunk of functionality into a unique DLL that explicitly links with all other dlls in the codebase and can exist as a standalone program. This removes all build time dependency between modules. Each file can be built independently, however changes to *Public* interfaces will require recompiles by all users. Modules can be swapped out or called by filepath.

**Pros:**

* *Zero build time dependencies*

**Cons:**

* Every DLL will require extra code to manage its interface its C-Style calls, creating more points for failure and increasing the complexity of adding new features
* Loss of type safety past DLL boundaries, causing bugs that cannot be caught at compile time, and may not even throw exceptions
* Users will be forced to use new/delete and manually manage memory or wrapper classes will need to be designed solely to clean up memory once deleted
* All communications will be inherently platform specific and require new interface code in order to be compiled on another platform

## Allowed Build Time Dependencies

The entire program is developed as a set of C++ packages that are allowed to have hierarchical build time dependencies. Packages can be added or removed from the project as long as the packages that are dependent on them receive some valid substitute or modification. Separate deployment strategies can be implemented considering each deployment strategy has a specified project and any code necessary to provide interfaces between packages.

**Pros:**

* C++ linker handles all cross module communication optimally, clearly communicates cross-module errors during compile time, and automatically relink or recompile modules when any headers they include change.
* Never need to use raw pointers internally, leaving the external wrappers as the only potential for a memory leak.
* Separate deployment strategies can be developed with the same core code and additional wrappers to handle cross module communication independently.
* Internal logical and physical structure of the codebase has no impact on external callers, allowing for changes to be made internally without any impact on the design of C#/Python wrappers

**Cons:**

* Developers for this codebase will require access to the source code