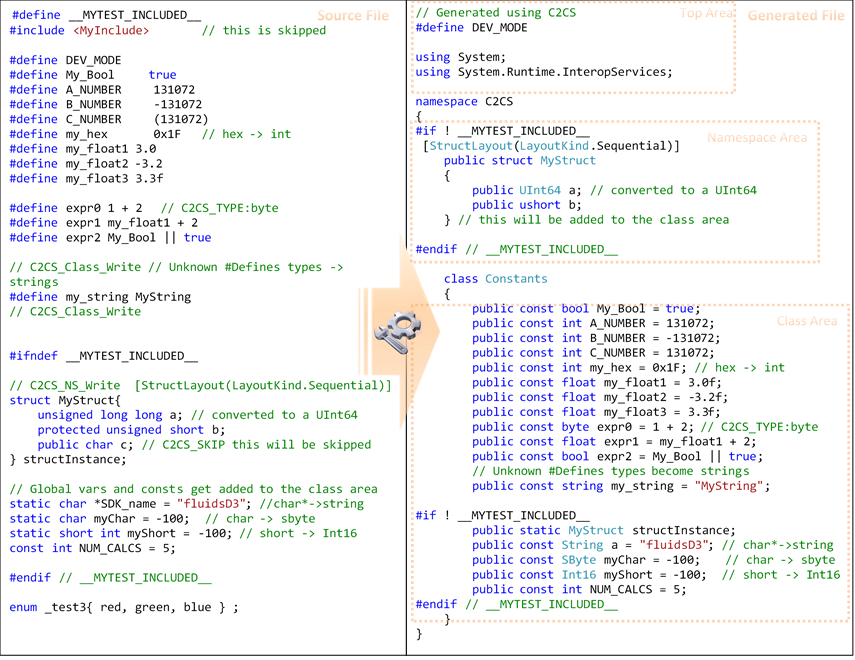
* **[Download Source - 15.2 KB](http://www.codeproject.com/KB/cs/800111/cppHeader2CS_source.zip)**
* [**Download Executable - 9 KB**](http://www.codeproject.com/KB/cs/800111/cppHeader2CS_Executable.zip)

 Introduction

This command line based tool extracts C/C++ constants, predefinitions, structs, and enums from a C/C++ header file and then saves them to a C# file during compile time using Visual Studio's pre-build command line events.  This happens very quickly so the compile time hit is minimal.  The goal is to keep one set of constants and enums when a solution has both C/C++ and C# projects.  These constants and enums are extracted from a given C/C++ file, converted, and are then written to a generated C# file that is usually already added to a C# project.

Since C# does not support predefinitions at the same level as C/C++, the program converts #define values to C# constants in a default class.  Since the C/C++ #define values are typeless, *CppHeader2CS* tries to figure out the best constant type to use for the C# output. In a nutshell, it first tries parsing an int, followed by a float, followed by bool, and if all else fails it just saves it as a string.  Example:*#Define cat 5* is translated into *public const int cat = 5;* whereas *#define cat 5.0* is translated into *public const float cat = 5;*.  Any type can also be forced by appending the C2CS\_TYPE command in a comment following the #define.  For Example, *#Define cat 5 // C2CS\_TYPE: float* is translated to *public const float cat = 5;.*

To accomplish the conversions RegEx is used to gather the information, then the data is converted into a c# equivalent and then saved into one of three destination areas. The three C# destination areas are the top, namespace, and class area. (see orange boxes in diagram) StringBuilders are used to gather the converted data.  The first StringBuilder writes to the very top of the output - this can be something like "*using system.IO;".*A *"using system.IO;"* can be added using a special command in the C/C++ source file, "*//C2CS\_Top\_Write using system.IO;".* The second StringBuilder outputs to the namespace area and would typically consist of structs and enums. The final location is in a default class area and this mainly holds constants. In the end, these are merged into one document and are outputted either to a file or to the console.  Below you will find an output image with some orange boxes on it. Each orange box represents a StringBuilder.



Inspiration

In the past, I have had many instances where I had a need to pass constants, enums or simple structures from a C/C++ to a C# project.  Searching online, I could not find anything that would do what I wanted.  Other solutions were full C/C++ to C# conversions that were too slow to run on each compile.  I did find a nice T2 templates project that did some of the stuff I wanted but overall the T2 template was hard to read and do not support the functionality I needed. I decided on a regular expressions approach.  RegEx can be a little annoying and hard to read but they work great for some problems.

Using the code

Usage Instructions

1. Copy the *CppHeader2CS.exe* file to some location
2. Open the Project Properties for the C# project and then navigate to the Build Events Section.
3. In the Pre-build event command line enter something like the following:   
   *C:\[tool location]\CppHeader2CS.exe "$(ProjectDir)MyInput.h" "$(ProjectDir)myCppSharedItems.cs"*  
   Depending on your needs you will need to adjust the file names and locations above.  Visual Studio helps with this using the macros button.

Command Line Usage

*CppHeader2CS.exe input\_file [output\_file]*  
*input\_file* - this is the C/C++ source header file.  It should be a simple file meant for the C/C++ to C# sharing.  
*output\_file* - this is an optional C# output file that is typically a file in the C# project.

Command List (located in c/c++ source file in comments)

The program supports a some commands that can be entered in comments in the source file.  These can be used to write or adjust the output.  The three C2CS\_\*\_Write commands can be used to pretty much put anything in the c# file through comments. The other options can be used to set a custom namespace or class name as well as force a type or skip a line in the source all together.

**// *C2CS\_TOP\_Write text to write*** - C2CS\_TOP\_Write is a direct passthrough that writes text above the namespace. This can be useful whenever there is a need to print at the top of the output file. Example:  *//C2CS\_TOP\_Write using System.IO;*  
**// *C2CS\_NS\_Write text to write*** - C2CS\_NS\_Write is a direct passthrough that writes text above the default class but in the namespace area. This area usually contains enums and structs.  
**// *C2CS\_Class\_Write text to write*** - C2CS\_Class\_Write is a direct passthrough that writes text in the class area.  *CppHeader2CS* writes only to a single class.  
**// *C2CS\_Set\_Namespace MyNsName*** - C2CS\_Set\_Namespace sets the namespace name. This is optional and if unspecified defaults to C2CS.  
**// *C2CS\_Set\_ClassName MyClass*** - C2CS\_Set\_ClassName sets the class name to use. This is optional and if unspecified defaults to Constants.  
**// *C2CS\_TYPE MyType*** - C2CS\_TYPE is used in the comments after a #define to specify what type to use.  This is required if the automatic detection does not work as expected or the programmer wants to force a type. Example: *#Default mySum (2+1) //C2CS\_TYPE int;*  
**// *C2CS\_SKIP*** - Adding C2CS\_SKIP in any comment forces *CppHeader2CS* to ignore the current line.

Examples

**Example 1 - #define ->int**

#define Cars 10

...gets added to the class area as...

public const int Cars = 10;

Points of Interest

* #automatically is recognized as an int.

**Example 2 - #define ->float**

#DEFINE MyDouble 3.2

...gets added to the class area as...

public const double MyDouble = 3.2f;

Points of Interest

* "#DEFINE" will always get translated to lower case
* *#DEFINE MyFloat 3.2f*   with the “f” will result in a float output.
* *#DEFINE MyFloat 3.2d*  with the “d” will result in a double output.
* when no suffix is given the output will be double
* scientific notation is supported

**Example 3 - #define -> handles very simple expressions**

Given my\_float is an existing float...

#define expr1 my\_float + 2

...gets added to the class area as...

public const float expr1 = my\_float + 2;

Points of Interest

* The converter recognized that this must be a float. It is very limited however as it only works with +,-,/, \* and for logic true, false, &&, ||
* If it does not auto-detect as expected, the command // *C2CS\_TYPE MyType* can always be used.

**Example 4 - #define -> string**

#DEFINE MyString New World Test!

...gets added to the class area as...

public const string MyString = "New World Test!";

Points of Interest

* If int.TryParse, float.TryParse, and bool.TryParse all fail then type string is used

**Example 5 - simple #define with no value**

#DEFINE FullDebug

is slightly modified and moved to the top...

#define FullDebug

Points of Interest

* not shown, but #define is moved near the top of the c# file as required by C#
* case was change to lowercase

**Example 6 - Struct**

// C2CS\_NS\_Write  [StructLayout(LayoutKind.Sequential)]

struct SomeStruct1{unsigned char  a; unsigned char b;};

...gets added to the namespace area as...

[StructLayout(LayoutKind.Sequential)]

public struct SomeStruct1

{

    public byte a;

    public byte b;

}

Points of Interest

* Here we prefix a ” // C2CS\_NS\_Write [StructLayout(LayoutKind.Sequential)]” This will cause the struct to be layed out more like a C/C++ struct. Only add this if you need it.
* Single line structs are supported but they get converted to standard format.
* Only basic items are supported here like int, short, long, long long, char, char\*, unsigned, floats, and doubles.

**Example 7 - Struct #2**

struct SomeStruct1{

   unsigned long long test123; //some comments

   public long test124;

} someStruct1Instance; // my instance notes

...gets added to the namespace area as...

public struct  SomeStruct1

{

  public ulong  test123; //some comments

  public long  test124;

}

...and the following also get added to the class section...

public SomeStruct1  someStruct1Instance;  // my instance notes

Points of Interest

* “//” style comments are brought over when on the same line
* Bit fields are not supported at this time

**Example 8 - Enum**

enum MyEnum {

        test1 = 33,

        test2 = 5,

        test3 = test2 };

...gets added to the namespace area as...

[Flags]

enum MyEnum

{

    test1 = 33,

    test2 = 5,

   test3 = test2,

};

Points of Interest

* "[Flags]" is prefixed only if there are assignments.  This was done so that the enum meaning would be the same. This might be changed in the future.

**Example 9 - constants/static**

protected static char myChar = -100;  // my notes

...gets added to the class area as...

public const SByte myChar = -100;  // my notes

Points of Interest

* the output is always  "public const..."
* protected and static are ignored (also private and const specifiers are ignored)
* Values are written to the class section.

Performance

Performance was an important component for this project. If this compile time tool took too long it would not be user friendly.  I used some RegEx tips that I found, StringBuilders, and a Dictionary to help with this. The conversion is relatively fast. The demo/testing input file read+convert+write takes between 10 to 30ms depending on what system I use. During the build, I had a couple opportunities to increase the performance slightly, however, I gave those up for a more readable and simpler code.  One thing I learned in some online 'RegEx Tips' is to make sure there is not any catastrophic backtracking.  In a nutshell it’s a good idea to be more specific in the RegEx expressions and to avoid the ketch-all ".\*".  I had no major issues but I did find being more specific did help by about 50% for the demo file.  There were some other tips I implemented also that can easily be found online.

The file is also only 37kb and this should help out with its performance.  39 copies would fit on one floppy disk. =)

Limitations

* Only works with preprocessor directives, constants,  structures,  and enums.
* Does not convert any functions (it’s not a full C/C++ to C# converter)
* / \*  ... \* / style of comments are not brought over.  In the first pass of the program they are simply removed. They can be used anywhere in the source file but just note that they will never be in the output file.
* typedef is not supported
* bitfields are partially supported
* char array are not supported

Future Wish list

* Add support for simple function conversions such as:   int add(int a, int b){ return (a+b);}
* add typedef, char array, and additional bitfield support

If you would like to contribute, please feel free to email me the changes and I’ll try to update the project when I can. I’ll create a change log file with the changes and contributors in there. Please note that I might choose to modify the changes or choose not to add them.

Points of Interest

Being able to recognize and parse a C/C++ file so easily and quickly is a great testament to Regular Expressions. While a non-RegEx C/C++/C# parser would have probably had better performance and control, the RegEx solution was much easier and quicker to get running.  RegEx is a great tool when the programmer wants to get any pattern matching done quickly and reliably.  Without RegEx, the code would have been taken much longer to build and it’s possible that I would have lost interest in the project before I finished it! =)

Other Tools to Try

**P/Invoke Interop Assistant** - This is a tool that was brought to my attention that does something similar to what cppHeader2CS does.  The P/Invoke Interop Assistant is a tool that generates C# or VB  P/Invoke code from a c++ file but it also emits structs, enums, #defines, and other items.   Depending on your needs, this tool might be a better fit.  Here are some advantages of each:

P/Invoke Interop Assistant advantages over cppHeader2CS:

* windows.h types
* p/invoke header generation(the main purpose of the tool)
* it supports char arrays, typedef, and include files
* output to VB
* output in general is better for p/invoke usage
* there is probably more that I am missing (it's a more powerful/complex tool in general)

Summary: Use Interop Assistant if any of the above requirements are needed.  This tool can be run at build time but it is slower and will always update the destination file.  Having the destination file updated can be annoying if the file is open as visual studio will prompt you to reload it.

cppHeader2CS pluses:

* supports more predefs, IA only supports #define
* supports global constants
* support custom namespace and class names
* allows guided output through commands in comments
* type auto-detection of #define -> const works a little better
* is faster, around 0.1 sec vs 0.7-3.0 sec (on my system)
* will not overwrite the output if there are no changes
* small 26kb file and no installation needed

Summary: Use cppHeader2CS for simpler header files that get modified often and are updated during the build process. This tool does not support char arrays, typedef, custom types, and include files.    
    
Instructions for those that might want to try this tool in place of cppHeader2CS:

1. go to <http://clrinterop.codeplex.com/releases/view/14120>[[^](http://clrinterop.codeplex.com/releases/view/14120)] and download the program
2. Install the program
3. In the install folder 'something like C:\Program Files (x86)\InteropSignatureToolkit' there is a file named sigimp.exe and this is the file that is needed. In the 'pre-build event command line' area use something like "C:\Program Files (x86)\InteropSignatureToolkit\sigimp.exe" /genPreProc:yes /lang:cs /out:"$(ProjectDir)MyOutput.cs" "$(ProjectDir)MyInput.h"

**Using T4 Templetes** - This is another way to perform build time C/C++ to C# conversions.  It is just a simple T4 file.  The example in the link below just handles #define to constent conversions and will auto-detect simple integers.   I am including this because this is the tool I used before cppHeader2CS.  <http://stackoverflow.com/a/5638890>

History

5/30/2014 - First working program, #define support only  
6/20/2014 - Added more preprocessor support, structs and constant support  
6/27/2014 - Added support for source file commands  
7/12/2014 - Added #define auto-detect type support  
7/21/2014 - Added enum support  
7/26/2014 - Renamed from C2CS to CppHeader2CS (C2CS can be misinterpreted as a full C to C# converter)  
7/27/2014 - Initial post  
10/15/2014 - Major Update

* same-file custom types are not supported.  "struct A { ... }; struct B { A a;...}" will convert.
* decimals without suffix now default to double
* added 'd' suffix support,
* added additional double support
* added void\* support
* bitfield structs are accepted but bit portion is removed