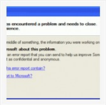
# Mixing Multiple Visual Studio Versions in a Program is Evil

Home → Programming → C/C++ → **Mixing Multiple Visual Studio Versions in a Program is Evil**



as encountered a problem and needs to close.
ience.

middle of something, the information you were working on

rosoft about this problem.
an error report that you can send to help us improve Som
t as confidential and anonymous.

his error report contain?
ort to Microsoft?

Like 0 Tweet Save Share 40

Many times I have seen programmers complain that they see some strange behavior during testing; something that they cannot explain. Either their program crashes randomly or it simply crashes for no apparent reason. They scratch their head trying to debug their code. Everything seems fine, inputs are OK, memory allocations seem fine, code does not seem to contain any bugs yet it fails spectacularly.

I have seen countless number of hours being lost trying to get to the bottom of such failures, only to find out that the cause of the unexplained behavior was that the program used DLLs compiled against multiple Visual Studio versions. For example: a program using `LibA.dll` that is compiled against Visual Studio-2005 and `LibB.dll` that is compiled against Visual Studio-2008.

Very often, in various programming forums, I see questions about mixing multiple Visual Studio versions in the program. I also see responses such as – "this should not be a problem" or "I do this all the time without any issues".

Let's explore in detail various issues surrounding the topic of using multiple Visual Studio versions in the program. We will also see what can be done to avoid potential issues if multiple Visual Studio versions can't be avoided. I have not explored this topic in detail for .NET languages (i.e. managed code), so I will limit this discussion to code developed in native C/C++ programming language only. In particular, we will explore this topic in following context:

- Dynamic Memory Allocations.
- Global and Static Data Variables.
- Function Calls taking/Returning C/C++ Objects.
- Template Classes, In-Line & Template Functions.
- File I/O.
- Environment Variables.

## Visual Studio Product Names and Versions

Visual Studio C/C++ run time libraries contain Visual Studio internal version numbers. Mapping between Visual Studio product names and corresponding internal version numbers is as follows:

| Product Name | Version Number |
|---|---|
| Visual Studio .NET | 7.0 |
| Visual Studio .NET 2003 | 7.1 |
| Visual Studio 2005 | 8.0 |
| Visual Studio 2008 | 9.0 |
| Visual Studio 2010 | 10.0 |
| Visual Studio 2012 | 11.0 |

## Visual Studio Runtime Environment Explained

Visual Studio C/C++ run time environment (CRT environment) consists of the following dynamic libraries:

- **`MSVCR*.DLL`** – This DLL contains C run-time routines – such as `printf`, `scanf`, `fgetc`, `ceil`, `getenv`, `putenv`, `isdigit`, etc. When C code is linked dynamically to CRT, Visual Studio will link it against appropriate version of `MSVCR*.DLL`.
- **`MSVCP*.DLL`** – This DLL contains C++ run-time routines, classes and templates, etc. – such as standard template classes (`basic_string`, `vector`, `set`, `iostream`, etc.), C++ implementation of math library, etc. When C/C++ code is linked dynamically to CRT, Visual Studio will link it against appropriate version of `MSVCR*.DLL` and `MSVCP*.DLL`.
- **`MSVCM*.DLL`** – This DLL contains C/C++ run-time routines used for mixed mode (managed and native) programming.

There is also a `LIBCMT.LIB` which is used if C code is linked statically to the CRT environment.

Visual Studio run-time environment libraries are named after the internal version number. For example:

- For Visual Studio .NET 2003, these libraries are: `MSVCR71.DLL`, `MSVCP71.DLL`, and `MSVCM71.DLL`
- For Visual Studio 2005, these libraries are: `MSVCR80.DLL`, `MSVCP80.DLL`, and `MSVCM80.DLL`.
- For Visual Studio 2008, these libraries are: `MSVCR90.DLL`, `MSVCP90.DLL`, and `MSVCM90.DLL`.
- For Visual Studio 2010, these libraries are: `MSVCR100.DLL`, `MSVCP100.DLL`, and `MSVCM100.DLL`.

## Basics of how Libraries are Loaded at Run-Time

Without going too much into technical detail, I will try to explain how dynamic and static libraries are loaded into the program memory space at the time of execution. This explanation may not be technically accurate, but it provides enough explanation for basic understanding.

### Dynamic Link Libraries (DLLs)

DLL are designed to be shared across all processes that use them unlike a static library (explained later) where the library is private (not shared) to the process that uses it. In general, Dynamic Link Libraries can contain code, data, and resources in any combination.

Code segment is for the executable code and functions that the program calls. The code segment is shared across multiple processes that use the DLL. The code segment occupies a single place in the physical memory and is read-only.

Archives

- June 2014
- February 2014
- April 2013
- March 2013
- July 2012

Categories

- Corporate
- How To
- More Commentary
- Programming
  - C/C++
- Software Solutions
- Teamcenter
  - Administration
  - Workflow
- Visual Studio

Data segment is internal data that the DLL uses – such as static variables, internal data storage, etc. This data represents the state of the DLL. The data segment is generally private to each process that uses the DLL. Shared data segments are possible for *inter-process communication*, but that discussion is not relevant for the current topic. The data segment is read-write for the process that owns the data segment. Although the data segment is private to the process, it is still managed by the DLL. So in essence data segment is private per process, per DLL.

### Static Libraries

Static libraries are designed to become part of the executable during linking. Unlike DLLs, Static libraries are not shared. At the time of linking, all appropriate code segment and data segment elements from static library are included in the executable. Thus that code and data is loaded in the same physical memory location where executable code segment and data segment are loaded.

The most important thing to remember is that the data segment is private to the process whereas code segment is shared (in the case of DLL) or private (in the case of static library).

## Passing CRT Object across CRT Boundaries

For this discussion please refer to the CRT Environments below (Figure 1). This figure shows how the CRT environment is loaded depending upon if the library was dynamically compiled or statically compiled. Let's see some scenarios when a program might use multiple CRT instances.

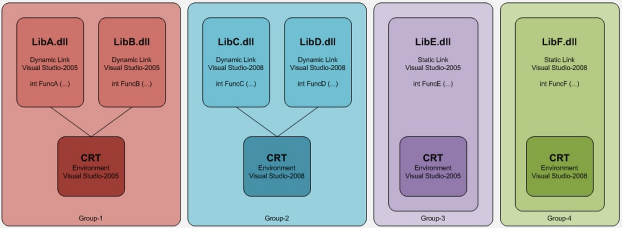


Figure 1: CRT Environment

Libraries from Group-1, namely `LibA.dll` (which contains the function `FuncA`) and `LibB.dll` (which contains the function `FuncB`), are dynamically linked against Visual Studio-2005. As a result these DLLs will share a common version 8.0 CRT instance.

Similarly, libraries from Group-2, namely `LibC.dll` (which contains the function `FuncC`) and `LibD.dll` (which contains the function `FuncD`), are also dynamically linked but against Visual Studio-2008. As a result these DLLs will share a common version 9.0 CRT instance.

However, the library from Group-3, namely `LibE.dll` (which contains the function `FuncE`), is statically linked against Visual Studio-2005. As a result it contains its own version 8.0 CRT instance. This CRT instance will **not** be shared with any other library or program.

Similarly, the library from Group-4, namely `LibF.dll` (which contains the function `FuncF`), is also statically linked but against Visual Studio-2008. As a result it contains its own version 9.0 CRT instance. This CRT instance will **not** be shared with any other library or program.

As long as a program links against libraries from only one of these four groups, your program will have only one CRT instance. However, if your program links against libraries from more than one group, then your program will have multiple CRT instances.

There is an additional layer of complexity depending upon how the program itself is linked (dynamically or statically) to CRT environment. If the program itself is linked dynamically then it will share the appropriate CRT instance with other programs/DLLs. On the other hand if the program is linked statically then it too will have its own CRT instance that will not be shared with any other program/DLL. But for the sake of simplicity, let's focus this discussion on only DLLs and their CRT instances.

CRT objects – such as file handles, environment variables, allocated memory, etc. are the data objects created and managed by a CRT instance.

Are multiple CRT instances in the program memory space good or bad? In general passing CRT objects across CRT instance boundary (within which they are created and managed) is a bad practice and should be avoided. Let's examine some cases and see why. For rest of the discussion, please refer to the CRT Environments above (Figure 1).

### Dynamically Allocated Memory

Dynamic memory is allocated when you do `new` (C++) or `malloc` (C/C++). Dynamic memory is allocated on heap. Each CRT instance manages its own heap. Whereas local variables, function parameters, and return values are created on the stack. The program, not the CRT instance, manages the stack.

When `FuncA` inside `LibA` allocates dynamic memory, it calls `new` or `malloc` method from its associated CRT instance. This dynamic memory is allocated on the heap and is managed by that CRT instance. Keep in mind that sometimes dynamic memory is allocated in the background. For example – the CRT function `strdup` will allocate memory, copy input string and return the allocated memory. Let's explore various scenarios and see what could happen if this dynamically allocated memory is passed as function parameter to other functions inside other libraries.

| **`FuncA` Calls** | **Called Function Performs** | **Notes** |
|---|---|---|
| `FuncB` | • Memory Free<br>• Memory Reallocation<br>• Memory Access | • Function call does not cross CRT boundary.<br>• `FuncB` will access same CRT instance as that of `FuncA` for performing memory operations.<br>• Free or reallocation of memory will be performed on the same heap that was used by `FuncA` to allocate memory. |
| `FuncC` `FuncD` `FuncF` | • Memory Free<br>• Memory Reallocation | • Function call crosses CRT boundary.<br>• The CRT instance allocating the memory is not same as the one attempting to free it. Furthermore they are of different versions. |

| | | |
|---|---|---|
| | | • Function call will result into memory access error or program crash because the heap where memory is allocated is not same as the one from where it is being freed. The CRT instance of Group-2 or Group-4 does not have any record of this memory allocation. |
| `FuncC` `FuncD` `FuncF` | • Memory Access | • Function call crosses CRT boundary.<br>• Accessing the allocated memory may not result into an error. However, interpreting the memory content may result into exceptions, data corruption, memory access error, or program crash.<br>• See subsection on "Passing C/C++ Objects" later in the blog. |
| `FuncE` | • Memory Free<br>• Memory Reallocation | • Function call crosses CRT boundary.<br>• The CRT instance allocating the memory is not same as the one attempting to free it. It does not matter that the CRT instance versions are same.<br>• Function call will result into error or program crash. |
| `FuncE` | • Memory Access | • Function call crosses CRT boundary.<br>• Accessing the allocated memory may not result into an error. This is different than `FuncC` accessing memory. In this case versions of CRT instances are same. However, the CRT instance of `FuncE` is statically compiled and hence could be at a lower build level than the CRT instance of `FuncA`. This difference in build level may result into exceptions, data corruption, memory access error, or program crash while interpreting the memory content.<br>• See subsection on "Passing C/C++ Objects" later in the blog. |

Here are some suggestions to avoid this potential error:

- Make sure that the executable and all dependent libraries are linked dynamically and using same Visual Studio version. This will ensure that at run time your program will use one and only one CRT instance. Any library can perform `malloc` and `free`, but all those libraries will access the same CRT instance to do those memory operations.
- Another potential solution is to statically link the program to the CRT environment. This solution may not work if your program depends on any 3$^{rd}$ party DLLs.
- If multiple CRT instances can't be avoided then the library responsible for allocating memory must provide means of freeing the allocated memory too. This is exactly what Teamcenter ITK does. ITK provides `MEM_alloc` and `MEM_free` functions. Now, regardless of which library allocates the memory and which library frees it, the actual allocation and freeing happens inside single CRT instance, as long as memory is allocated using `MEM_alloc` and freed using `MEM_free`.

## Global and Static Data

Static data is persistent but its scope is limited to certain block of code, a function, or a file. Global data, on the other hand, has global scope. Any function from any DLL can access global data if the variable is exported. If it is not exported then any function from the same DLL which declares the global data can access it.

One well-known CRT function using static data is – `strtok`, which find the next token in string. The first call to `strtok` requires the input string to be tokenized. Subsequent calls, however, require `NULL` as the input. This function uses static data to store the state of the function and that's how it knows how to return next token when input is `NULL`. Any non-`NULL` input resets the internal state of the function. If the program has multiple CRT instances then each instance has its own `strtok` function and each `strtok` function has its own internal state.

Imagine what would happen if first call to `strtok` happens inside `FuncA`, second call to `strtok` (with `NULL` input) happens inside `FuncB`, and third call to `strtok` (with `NULL` input) happens inside `FuncC`. The first and second calls would work fine because both calls to `strtok` access the same CRT instance. But the third call, on the other hand, would result in the wrong token being returned if there were any prior calls to `strtok` from that CRT instance. Or worst yet, program would result in memory access error or program crash if it was the first ever call to `strtok` from that CRT instance.

Here are some suggestions to avoid this potential error, if program has multiple CRT instances:

- Check Microsoft documentation to see if the function has any specific warning. Also see if there is any "*safe*" replacement version available. Starting Visual Studio-2005, Microsoft provided many replacement functions to legacy CRT functions with potential of buffer overrun errors. Refer to Visual Studio-2005 documentation regarding Security Enhancement in the CRT and Deprecated CRT Functions. For example the "*safe*" replacement version of `strtok` is `strtok_s`, which uses the context parameter to store state of the function, rather than rely on internal static data.
- Even if there is a *safe* replacement version available, it is still a bad practice to have multiple CRT instances in your program.
- One another potential solution is to provide set of functions inside a single DLL to do all static and global data manipulation and use those function instead of direct CRT calls. This solution may not work if your program depends on any 3$^{rd}$ party DLLs.

## Passing C/C++ Objects (Non-Primitive Data Objects)

`int`, `float`, `double`, `char`, and `bool` (along with arrays of those) are all primitive data types. Their definition is not subject to the CRT version. It is safe to pass these objects of these data types from one function another even across a CRT boundary. But nowadays program rarely depends on these data types alone. Programs use derived data types such as – structures and classes. The following discussion refers to such derived data types.

C and C++ objects, such as class instances and structure variables can be passed from one function another by value or by reference. When passed by value, data is duplicated on the stack prior to executing a function call. Similarly, when returning by value, data is duplicated on the stack prior to returning. Data is copied either performing a bitwise copy in case of C or calling a copy constructor in case of C++. Let's explore pitfalls of multiple CRT instances in this context.

When `FuncA` calls `FuncB` and passes CRT objects by value, everything works fine because both functions use same CRT instance. But when `FuncA` calls `FuncC` by passing CRT objects by value, results are not guaranteed. This is because the passed CRT object may be interpreted differently in different CRT instance. Let's say that `FuncA` creates a C++ class instance of `std::vector<int>` and passes it by value to `FuncC`. The CRT instance of `LibA` performs the copy of that `vector` object on the stack using its own (Visual Studio-2005) interpretation of `std::vector` implementation. When the program enters into `FuncC`, the CRT instance of `LibC` interprets the `vector` object from stack using its own (Visual Studio-2008) implementation. It is very likely that the `std::vector` implementation of Visual Studio-2008 is much different than that from Visual Studio-2005. A similar situation would arise when returning CRT Objects by value. Also keep in mind that most C++ class instances use dynamically allocated memory, which might be freed or reallocated in background. Refer to the subsection on "In-Line functions, Template Classes, and Template Functions" later in the blog.

Well, then the next question is – is it safer to pass CRT objects by reference across a CRT boundary? And the answer is – absolutely not. It does not matter how you pass an object. It is the object interpretation, data alignment, and order of data elements that causes this issue. In my opinion passing CRT objects as references across CRT boundary are even more error prone than passing by values. One could at least, employ serialization/de-serialization techniques to safeguard CRT objects when passed by value.

The same holds true in case of simple C structures too. Although it is unlikely that, say for an example, the order of `struct tm` or `struct FILE`, would change from one version of Visual Studio to another, one can't reply on such assumptions.

Here are some suggestions to avoid this potential error, if program has multiple CRT instances:

- Rather than relying on C++ copy constructors or C bitwise copy, write your own serialization and de-serialization routines and use them consistently. Again, this solution may not work if your program depends on any 3rd party DLLs. In that case you may have to develop your own wrapper functions for serialization and de-serialization routines in multiple Visual Studio versions.

## In-Line Functions, Template Classes, and Template Functions

In-line functions are not compiled rather are replaced by its definition at the point of use, much like C/C++ macros. It is rather a bit more complicated than that. Visual C++ compiler has its own logic to decide which functions are to be in-lined and which functions are not. Using in-line keyword is merely a suggestion to the Visual C++ compiler that the function is a potential candidate for in-lining. Compiler, at its own discretion, may still decide to compile the function rather than in-lining.

On the other hand, the actual class definition of template classes and actual function definitions of template functions are always created at the time of use.

This situation adds another layer to the whole issue of multiple CRT instances. The actual class definition or function definition or in-line function may differ from one CRT version to another. This would result in interpretation errors of CRT objects, especially C++ class instances.

Here are some suggestions to avoid this potential error, if program has multiple CRT instances:

- Rather than relying on C++ copy constructors or C bitwise copy, write your own serialization and de-serialization routines and use them consistently. Again, this solution may not work if your program depends on any 3rd party DLLs. In that case you may have to develop your own wrapper functions for serialization and de-serialization routines in multiple Visual Studio versions.

## File I/O and Operations

Just as mentioned in discussion about C++ class objects and template classes, file I/O is also subject to potential errors when performed from functions across CRT boundaries.

In C, file I/O is performed by various CRT function using the `FILE` structure. This in and of itself may not cause an error. However using `fopen` in one CRT instance and `fclose` or `freopen` in another would certainly cause a memory access error, or will crash the program.

In C++ file I/O is performed using `iostream` library of template classes. These classes and methods are subject to same limitation discussed before in subsection “In-Line functions, Template Classes, and Template Functions”.

Here are some suggestions to avoid this potential error, if a program has multiple CRT instances:

- Short of making sure that the program does not have multiple CRT instances, there is not much you can do about this.

## Environment Variables

Even reading and writing environment variables in multiple CRT environments would cause a memory access error, or will crash the program.

An environment variable value is read using `getenv` CRT function. And it is updated or created using `putenv` CRT function. Just like heap management, each CRT instance manages its own environment variables. Each CRT instance is initialized when program starts. At the time of initialization, the CRT instance copies the current environment space into its own buffers. Multiple CRT instances mean multiple such environment space buffers. And guess what…they don't talk to each other. Let's explore this in detail.

Let's say that `FuncA` calls `putenv` to set `MY_ENV` environment variable. Subsequently, `FuncB` reads that same environment variable using `getenv`. This all works fine because both these calls, `putenv` and `getenv`, are executed in same CRT instance. If `FuncC`, `FuncD`, `FuncE`, or `FuncF` tries to read that same environment variable using `getenv`, it may not find it defined or it may not have correct value in their respective environment variable space.

In this case, however, memory access error or program crash may not be direct result of calls to `putenv` or `getenv`. It would rather be due to the expectation that `getenv` would not return `NULL` and hence result of using `getenv` results without checking for `NULL`.

Here are some suggestions to avoid this potential error, if program has multiple CRT instances:

- Just like the suggestion for dynamic memory allocation, provide your own wrapper implementation of `putenv` and `getenv`. Use these wrapper functions consistently. This is exactly what Teamcenter ITK does too. ITK provides its own `ss_putenv` and `ss_getenv` wrapper functions.

## Resources

Just like dynamic memory CRT resources are managed by CRT instance. Hence, CRT resources created, allocated and passed over CRT boundary are problematic. If you must do this, then make sure your own library provides wrapper functions for complete implementation of various management functions of the resource – including allocating, destroying, reallocating, etc.

# Final Thoughts

As I have explained in this blog, using multiple CRT instances in a program is absolutely a bad idea. It is extremely problematic. Not to mention extremely hard to debug.

I tried to give the best explanation and possible workarounds in case if you must have multiple CRT instances in your program space. There is, however, no guarantee that those workarounds will work in every situation. Especially if you have 3rdparty libraries, you can't control how those libraries will handle those problematic areas.  You can't certainly force them to use your wrapper functions. Although, you could develop set of those wrapper functions for each version of CRT instance in your environment and use those wrapper functions as a go-between interface.

Here are my recommendations:

- Avoid using multiple CRT instances in your program space at all costs.
- Avoid using multiple CRT instances in your program space at all costs.
- Did I mention, avoid using multiple CRT instances in your program space at all costs.

## Next Steps

I think I have beaten this subject to death. I don't think I need to provide any more examples to convince you not to let your program use multiple CRT instances. If you are convinced, I have done my job and I would certainly appreciate your comments/feedback. If you are still not convinced, I am out of examples and can only say that you are on your own.

Please feel free to comment on this topic. We also encourage you to explore our website to learn about products and services we offer. If you have any questions or wish to discuss this topic or any other topic with us in private, please contact us.

Like 0 Tweet Save Share 40

By Yogesh Fegade on April 9, 2013 / C/C++, How To, Programming, Software Solutions, Visual Studio / 42 Comments

## 42 Comments

**Suresh Amberkar**
5:11 pm on April 25, 2013

Great and very helpful article. I do a lot of coding with C and C++ in NX. This blog explains why we should stick to the Visual Studio version supported for each version of NX. Thanks.

Reply

**Blaine Bell**
10:16 am on July 2, 2013

Yes, very impressive article. Does anyone know whether this includes the Java JRE? We have an Applet that uses JNI into a library that was compiled against VS2010. Since not all Windows machines have the 2010 redistributables, we include them (msvcr100.dll and msvcp100.dll) in the jar so that we can load them. However, some JREs (e.g., 1.6) depend on other versions of Visual Studio (the JRE only distributes msvcr.dll), including a possible new version of the JRE might depend on VS2013. Does this mean that our Java Applet is evil if it runs on that older JRE version? Do we need to compile different versions for VS versions, then dynamically figure out which libraries exist on the machine, and use the right version? Or has JAVA figured out a way to minimally use the msvcr so that it does not have the problems described above. Thanks again for a great article!

Reply

**Lalit Wagh**
6:48 am on August 4, 2013

Great explanation!!!

Reply

**Sachin**
4:27 pm on March 10, 2014

Thanks Yogesh!!!
very helpful article.

Reply

**Alex**
7:13 am on March 13, 2014

Thanks
Very helpful article.

Reply

**Marco**
8:10 am on May 14, 2014

Like a mantra: Avoid using multiple CRT instances in your program space at all costs.

Thank you.

Reply

**MICHAEL**

7:36 pm on June 5, 2014

Thank you for this article as it makes perfect sense. The "pain I am feeling though," is that our organization has different project teams that for a time for various reasons must stay the course in building with their VS version. Since their libraries are shared amongst other projects, we who develope in C++ are forced to stay with their version, 2010. it would be sure nice if there was some sort of backwards compatibility feature or switch that could be set, until a future time when dlls from that team could be built with the latest version of VS. No other but for this reason my project team must stay back in 2010, and I am missing out on all the awesome teamware features of 2013. Any thoughts? Thanks again for your explination.

Reply

**YOGESH FEGADE**

10:58 pm on June 5, 2014

Michael,

Unfortunately there is no easy or simple workaround to your problem:-(

Reply

**BAJI**

12:23 am on September 27, 2014

could you please share the ITK code to read a BOM and its children

Reply

**SYED**

12:33 am on May 27, 2015

This is the exact thing i was looking for, Recently we integrated 3rd Party C++ library in our C library and we started seeing the backward compatibility issues.

So i am now sure why Microsoft break the backward compatibility between two version of Visual Studio.

Reply

**SYED**

12:45 am on May 27, 2015

I have one question though!!

You said issues will be at runtime, what i am seeing is i can't use C++ library compiled with VS2013 with the program which are still using 2010.

I am assuming, whatever you mentioned above is the reason for this behavior.

Reply

**YOGESH FEGADE**

9:30 am on May 27, 2015

Syed, Most of these issues will be observed during run-time. There is very little chance that you will see any issues during compile time.

For example: If you have 3rd party DLL compiled using VS2013 and you are using that DLL to compile your project in VS2010, then you will be linking to the 3rd party DLL and all your CRT calls (from your project in VS2010) will link against CRT V10 libraries installed on your machine.

Reply back, if I did not understand your question properly.

Reply

**KAMAL**

1:51 am on July 16, 2015

As you have specified that you were using C++ so are you using template libraries (STL)? If so then there is definitely mix of inline and non-inlined calls as its templated which will give compile time errors.

Reply

**SYED**

1:00 am on May 28, 2015

Thanks for your reply!!!

Yes we are seeing issue at runtime also.

But the answer i am looking about this the forward/Backward compatibility of C++ libraries.
If i have a C library( compiler on VS 2010) which is linking against 3rd Party C++ library ( Also compiler on VS 2010) and if i want to use this library with application which is compiled on VS 2013. Will this is possible? If we can compiler & link properly then will there be runtime issues?
Thanks

Reply

**YOGESH FEGADE**

11:23 am on May 28, 2015

Even if you are able to compile and link successfully does not guarantee that there will not be any run-time issues.

In the scenario that you explained, where those two DLLs (one custom and one 3rd party) linked against VS2010 are used by an application linked against VS2013, you may see run-time issues depending upon what CRT resource are being used and if they cross the CRT boundary.

It's also likely that the run-time issue that you see may not be related to resources crossing CRT boundary. They may be entirely due to programming bugs.

Reply

**SYED**

8:48 am on May 29, 2015

Thanks..please answer my last question for static libraries.

Will static library has compatibility issues.?

Reply

**YOGESH FEGADE**

9:56 am on May 29, 2015

In your situation, linking to static library may also cause issues because you can't control how your 3rd party DLL is linked. Plus you are dealing with multiple versions of visual studio. Even if you link your DLL and application statically, you will have static version of VS2010 libraries and static version of VS2013 libraries.

If you are certain that the run-time issues are because different versions of Visual Studio, then your options depends on what those errors are and what resources are affected (assuming that you can't link everything against VS2013 and you can't ask your vendor to link their 3rd party DLL against VS2013).

If your run-time issues are NOT because of different versions of Visual Studio, then it comes down to good old debugging!!

Reply

**SYED**

6:56 am on June 1, 2015

Thanks for your prompt reply!!!
Highly appreciate your inputs.
Thanks!!!!

Reply

**HAN**

3:38 am on June 4, 2015

Great article. This article solved my problem. Thanks!

Reply

**Zailor**

1:22 am on July 22, 2015

Hi,

Thank you for the very good article.

But I have a little problem with this sentence.

"As long as a program links against libraries from only one of these four groups, your program will have only one CRT instance."

My understanding is

if a program is linked against libraries from one of these group, AND the program and the library are using the same compiler version, then the program will have have only one CRT instance.

Otherwise, it will have multiple CRT instances.

Is my understanding correct ?

Reply

**Yogesh Fegade**

9:28 am on July 22, 2015

You are correct, however, Visual Studio version will dictate which CRT library version you are going to link against.

So, as long as compiler is same you will link against same version of the CRT library; and conversely if you link against same version of the CRT library implies that you are using same compiler.

Reply

**Ganesh Koli**

12:53 pm on August 5, 2015

Hi Yogesh,
Thanks for the article, it really helpful.
I am stuck in following issue, can you please help me :
i write a makefile to compile the source code, using third party dll (odbc-dll)
in building the binary . i am using Visual studio 8 and 10 to build binary , in both cases binary generated . but if i tested both binary on another machine then only binary (using visual studio 10) is working fine . but binary (using visual studio 8) is not able to start.
i checked the DLL and their path used by binary using 'dependency walker' software , and i can see that , both binary are using different dll, as per you mention i.e. :
binary build on VS 10 use : msvcr100.dll and msvcp100.dll
binary build on VS 10 use : msvcr90.dll and msvcp90.dll.

but i am not getting why the binary (VS 8) is failing ?

Reply

**Yogesh Fegade**

1:48 pm on August 5, 2015

If the binary built using VS8 fails to start on the machine – then you should check the binary using dependency walker on the machine where it fails (not on the machine where you generate the binary). Dependency Walker should tell you if any dependent dll is missing.

It it fails (as in crashes) then there is a chance that a crash dump (.dmp) is generated. You can analyse that dump file to see why it failed.

Else, you will have to debug the binary built using VS8 on the machine where it fails.

Reply

**Bjorn**

6:35 am on November 27, 2015

Hello!
Thanks alot for this article, as a third party developer this is very interesting. But I'm still don't understand this fully, so please bear with me if the question has already been answered.
If I build a DLL with everything it needs statically linked, (including c/c++ runtime). The only functions exposed are of "extern C" nature, almost exclusively using primitive types, and pointers to primitive types. There is one exception, a struct created from inside the DLL, that contains primitive types, (including a const char* and a couple of void*). (The struct is "typedef struct structName" if that matters). Can this be a problem?
The other case where I'm unsure of is that we pass in char* (char arrays), and a int saying how

large the "array" is. Is this valid? Internally we use c++ functions, but no c++ is ever exposed outwards.

No heap allocations will ever happen on opposite sides of the boundry from what I understand. However we do expose the ability to set two callback functions, one for "malloc" and one for "free". Can this cause a problem?

Considering that we statically link the runtime, is there a risk that when using our DLL the linker may use one or the other statically linked version, (ex: use part of ours in their code, causing corruption?)? If this could be a problem, is there a way around it? (Hiding the symbols outwards of the CRT functions we use?)
Thanks!

Reply

**Yogesh Fegade**
11:32 am on November 27, 2015

Yes, it is quite interesting and challenging to maintain 3rd party libraries and avoid the DLL hell:-)

> If you build your DLL with all required run-time libraries statically linked, don't expose objects other than of primitive types, and provide malloc/free callbacks for your users to allocate/free memory for the objects that your DLL needs and manages then you should be fine as long as your users exclusively use your malloc/free callbacks to manage the memory for the objects that your DLL needs and manages. Using structures made up of primitive types should be fine as well.
>
> Accessing memory across the DLL boundary is not a problem. So, if you pass in char* (char arrays), and a int saying how large the "array" is, then as long as the receiving function only tries to read the memory there is no risk. Even updating memory content is OK to. For example if the first element in the character array is "Bjorn" and the receiving function wants to change it to "BJORN". What the receiving function should avoid doing is – free the memory or reallocate. They can still do that if they use your malloc/free callbacks.
>
> if your DLL links run-time library statically then there is no risk of linker using your static version of run-time into some other DLL.

Keep in mind that if you build statically linked DLL, then size of your DLL is going to increase.

Reply

**Tao**
9:30 pm on February 17, 2016

Hi Yogesh,

Thanks for the article which is really helpful. By the way, I observed that when we use dlls built from multiple VS versions in a same program (say the caller built from vs2013, and it calls a func in a dll built from vs2010), if all the dlls are release version (built with option /MD), then there will be no issue on heap memory management. I searched online, somebody says in that situation, when multiple dlls work in /MD way, at runtime only one crt instance will be used. If possible, could you explain the rationale behind this?

Reply

**Yogesh Fegade**
1:55 pm on February 18, 2016

Very good question Tao.

I don't claim to be expert in memory management mechanisms, but your question prompted me to do some research. Thanks.

It seems that prior to Visual Studio 2010, CRT memory allocation functions (malloc, etc.) used custom allocator and hence each malloc required corresponding CRT call to free the allocated memory. Starting Visual Studio 2010, malloc simply uses the HeapAlloc to allocate. I believe that this memory allocated using malloc is allocated on processes's default heap (see later). That's the reason why if you mix multiple DLLs linking Visual Studio 2010 and above, there is no issue with heap management.

There are two types of heaps that process can create/access – default heap and dynamic heap. Both these are available only to the process that creates them. However, dynamic heap requires explicit call to create heap using HeapCreate and requires heap handle to allocate memory on that dynamic heap. Default heap also has a handle but can be accessed using GetProcessHeap.

So, it would seem that if memory is allocated on processes's default heap using HeapAlloc, then any CRT call should be able to free that memory using HeapFree. Starting Visual Studio 2010 malloc and free use HeapAlloc and HeapFree to allocate/free memory. So as long as allocating DLL and freeing DLL are linked with Visual Studio 2010 and above there will be no issue with heap memory management.

If you write your custom allocator to allocate heap on dynamic heap, then that custom allocator will need to keep track of memory allocations and the dynamic heap handle where the memory is allocated so that it can be freed properly. Obviously the DLL that creates the custom allocator knows what's where. This was the case with malloc for Visual Studio versions prior to 2010. In this case only the DLL that allocated the memory could free it for other DLLs would not know which heap has the memory.

Thanks very much for the comment. I also learned something new today.

Here are some links for more information:

MSDN – About Memory Management
How many CRT instances exist in a process?

Reply

**Tarmo Pikaro**
3:51 pm on April 24, 2016

Cross linked from here:

http://stackoverflow.com/questions/36811679/visual-studio-code-binary-compatibility

With a prototype for solution.

Reply

**philippg**
7:34 am on August 4, 2016

i also ran into this problem with "putenv" and "getenv".
can you please share some more information on the wrapper functions that you suggested (for the process environment variables)?

Reply

**Yogesh Fegade**
9:40 am on August 4, 2016

Thanks Philippg for visiting my webpage.

Refer to the "Figure 1: CRT Environment" image on the page.

Let's say that your code is in LibC (Visual Studio 2008). If you are setting an environment variable by calling putenv from a function inside LibC and reading it using getenv from another function inside LibC, then you are OK.

However, if you are setting an environment variable by calling putenv from a function inside LibC but expect user of LibA to read it using getenv from a function within LibA (or if you expect user of LibA to set an environment variable by calling putenv from a function inside LibA but expect to read it from LibC using getenv from a function within LibC), then you are in trouble. The environment space of LibC is not same as LibA. Environment variable that you set (putenv) inside LibC is not going to be visible (getenv) to methods from LibA.

That's the case when you will need wrapper. And this is how you will use it:

1. Write my_putenv inside LibC. Inside my_putenv, simply call putenv by passing arguments.
2. Write my_getenv inside LibC. Inside my_getenv, simply call getenv and return the output.
3. Instruct all consumers of LibC to use my_putenv to set environment variable and my_getenv to read it.

By doing so you have limited setting and getting environment to CRT instance of LibC only.
Similar logic will apply to all wrapper functions, such as for malloc, calloc, realloc, and free.

Reply

**philippg**
12:07 pm on August 4, 2016

thanks for sharing additional details.

i consume some legacy module (LibA) that is based on the logic to exchange data through environment variables.

i now understood that your "wrapper" concept would require to make changes to this foreign module (LibA) to call into my own code (LibC).
unfortunately this is not possible in my case.

_if_ i had everything under my control i could just as well discard the approach with environment variables and implement interfaces for direct data exchange, instead.

Reply

**Yogesh Fegade**
12:59 pm on August 4, 2016

Well, yes that's why it's critical to think about these details while designing the software.

This is particularly important if one is developing 3rd party libraries for public consumption. You won't know upfront who your consumers are and what environment they are using.

It is also important as a consumer to be aware of what your 3rd party library requirements are.

In your case though, you can still make it work. Although LibA is not in your control, you could develop your own LibZ that will compile against same Visual Studio environment as LibA (hence LibA and LibZ will share same CRT instance). Inside LibZ write my_putenv and my_getenv. Inside LibC (which is in your control) use my_putenv and my_getenv by linking against LibZ.

Now essentially your code (LibC) is setting/reading environment in environment space of LibA via LibZ.

Reply

**Zarian**

7:10 pm on October 12, 2016

Nice article! Things every library writer must know. There is also a good pattern in C++ for writing libraries in a way where you can avoid all the issues that come up when you are mixing dlls/so built with different compilers/ compilers settings etc. The pattern is called "Hourglass Pattern" and there is a very nice CppCon video that covers that in great detail. Here is the link to the video:

https://www.youtube.com/watch?v=PVYdHDm0q6Y

The basic idea is to expose your C++ library with C functions and provide a header only C++ interface on top of the C functions. This still gives the end user a rich C++ interface and avoid all the issues that can arise because of mixing dlls/so from different compilers.

Reply

**Yogesh Fegade**

9:07 am on October 13, 2016

Thanks Zarian, I appreciate the info. I will check that video.

Reply

**Amir Nassiri**

9:34 am on May 19, 2017

Thanks for the comprehensive article.

I have a simple question, do these problems happen to C# projects and managed code? I have projects that are originally developed in VS 2015 in C# version 6.0, but now I am planning to install VS 2017 (with C# version 7.0), should I worry to have bugs appearing in my projects or not?

Thanks in advance.

Reply

**Yogesh Fegade**

8:04 am on June 7, 2017

I don't think these issues will appear in C# or managed code.

Reply

**Andrew**

1:25 pm on September 6, 2017

Thank you for the enlightening article. Our team just ran into an issue with third-party libraries causing us to accidentally mix run-times. I greatly appreciate your detailed break down of the problems this causes, and explanation of potential solutions. You've successfully convinced me that my gut reaction was correct; we are just going to have to rebuild one of the libraries. Mixing versions is indeed evil.
Cheers!

Reply

**Yogesh Fegade**

2:02 pm on September 6, 2017

I am glad that this article helped you. Cheers!!

Reply

**Shree Hari**

6:29 am on January 30, 2018

Great article.

Reply

**PRAKASH VALSANG**

11:05 pm on July 20, 2020

Sir,
I have project in 2015 visual studio which contains 20 projects. Out of which one is v140 and remaining are v90. One of the V90 will load this V140 dll. How to mix them well, Now I have a memory leak in release mode and debug is ok. How to fix it

Thanks

Reply

**Yogesh Fegade**

10:32 am on July 21, 2020

How do you know you have a memory leak? Does you code crash? If your release build crash but debug build does not, then chances are you have unused variables. Unused variables are preserved in debug build but are optimized away in release build. Those unused variables may absorb memory overflow in debug build, which is not possible in release build causing code to crash.

as an example, following code is very likely to crash in release build, but will work nicely in debug build.

```
int
MyFunction()
{
int returnCode = 0;

char value1[1024 + 1];
char value[32 + 1] = ””;
char value2[1024 + 1];

//strcpy(value1, ”Almost 1000 char string”);
strcat(value, ”1234567890123456789012345678901234567890”);
strcat(value, ”1234567890123456789012345678901234567890”);
strcat(value, ”1234567890123456789012345678901234567890”);
strcat(value, ”1234567890123456789012345678901234567890”);
strcat(value, ”1234567890123456789012345678901234567890”);
//strcpy(value2, ”Almost 1000 char string”);

return returnCode;
}
```

Reply

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Submit Comment

SiOM Systems

Expertise in Teamcenter security & access management, configuration management, options & variant configuration, engineering change management, NX CAD data management, document management, workflow/process management, absolute occurrences, appearances, and data migration.

Our Philosopy

'Siddhi' means perfection, mastering a difficult skill, success. 'Om' is the source of all the energy; it is the source of the energy behind the creation, preservation and destruction of the universe. This philosophy guides us in providing perfect, flexible, future proof, high quality solutions to every customer.

Testimonials

... A true professional who can help guide Teamcenter implementations, mentor and teach staff new to Teamcenter, and architect and guide a client to the best practical solution. Read more

Back to Top

Teamcenter and NX are registered trademarks of Siemens Product Lifecycle Management Software Inc.

All other logos, trademarks, registered trademarks or service marks used herein are the property of their respective holders.

SiOM Systems makes no claim of ownership of such marks.