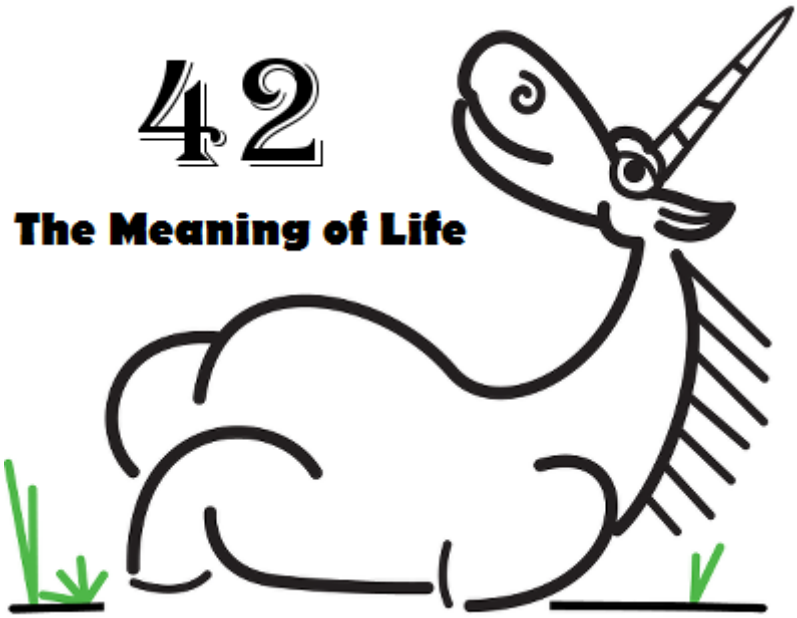
The Ultimate Question of Programming, Refactoring, and Everything

Author: Andrey Karpov

Date: 14.04.2016

Yes, you've guessed correctly - the answer is "42". In this article you will find 42 recommendations about coding in C++ that can help a programmer avoid a lot of errors, save time and effort. The author is Andrey Karpov - technical director of "Program Verification Systems", a team of developers, working on PVS-Studio static code analyzer. Having checked a large number of open source projects, we have seen a large variety of ways to shoot yourself in the foot; there is definitely much to share with the readers. Every recommendation is given with a practical example, which proves the currentness of this question. These tips are intended for C/C++ programmers, but usually they are universal, and may be of interest for developers using other languages.



Preface

About the author. My name is Andrey Karpov. The scope of my interests – the C/C++ language and the promotion of code analysis methodology. I have been Microsoft MVP in Visual C++ for 5 years. The main aim of my articles and work in general - is to make the code of programs safer and more secure. I'll be really glad if these recommendations help you write better code, and avoid typical errors. Those who write code standards for companies may also find some helpful information here.

A little bit of history. Not so long ago I created a resource, where I shared useful tips and tricks about programming in C++. But this resource didn't get the expected number of subscribers, so I don't see the point in giving a link to it here. It will be on the web for some time, but eventually, it will be deleted. Still, these tips are worth keeping. That's why I've updated them, added several more and combined them in a single text. Enjoy reading!

1. Don't do the compiler's job

Consider the code fragment, taken from **MySQL** project. The code contains an error that PVS-Studio analyzer diagnoses in the following way: V525 The code containing the collection of similar blocks. Check items '0', '1', '2', '3', '4', '1', '6' in lines 680, 682, 684, 689, 691, 693, 695.

```
static int rr_cmp(uchar *a,uchar *b)
{
  if (a[0] != b[0])
    return (int) a[0] - (int) b[0];
  if (a[1] != b[1])
    return (int) a[1] - (int) b[1];
  if (a[2] != b[2])
    return (int) a[2] - (int) b[2];
  if (a[3] != b[3])
    return (int) a[3] - (int) b[3];
  if (a[4] != b[4])
    return (int) a[4] - (int) b[4];
  if (a[5] != b[5])
    return (int) a[1] - (int) b[5];      <<<<====
  if (a[6] != b[6])
    return (int) a[6] - (int) b[6];
  return (int) a[7] - (int) b[7];
}
```

Explanation

This is a classic error, related to copying fragments of code (Copy-Paste). Apparently, the programmer copied the block of code "if (a[1] != b[1]) return (int) a[1] - (int) b[1];". Then he started changing the indices and forgot to replace "1" with "5". This resulted in the comparison function occasionally returning an incorrect value; this issue is going to be difficult to notice. And it's really hard to detect since all the tests had not revealed it before we scanned MySQL with PVS-Studio.

Correct code

```
if (a[5] != b[5])
  return (int) a[5] - (int) b[5];
```

Recommendation

Although the code is neat and easy-to-read, it didn't prevent the developers from overlooking the error. You can't stay focused when reading code like this because all you see is just similar looking blocks, and it's hard to concentrate the whole time.

These similar blocks are most likely a result of the programmer's desire to optimize the code as much as possible. He "unrolled the loop" manually. I don't think it was a good idea in this case.

Firstly, I doubt that the programmer has really achieved anything with it. Modern compilers are pretty smart, and are very good at automatic loop unrolling if it can help improve program performance.

Secondly, the bug appeared in the code because of this attempt to optimize the code. If you write a simpler loop, there will be less chance of making a mistake.

I'd recommend rewriting this function in the following way:

```
static int rr_cmp(uchar *a,uchar *b)
{
  for (size_t i = 0; i < 7; ++i)
  {
    if (a[i] != b[i])
      return a[i] - b[i];
  }
  return a[7] - b[7];
}
```

Advantages:

- The function is easier to read and comprehend.
- You are much less likely to make a mistake writing it.

I am quite sure that this function will work no slower than its longer version.

So, my advice would be - write simple and understandable code. As a rule, simple code is usually correct code. Don't try to do the compiler's job - unroll loops, for example. The compiler will most definitely do it well without your help. Doing such fine manual optimization work would only make sense in some particularly critical code fragments, and only after the profiler has already estimated those fragments as problematic (slow).

2. Larger than 0 does not mean 1

The following code fragment is taken from **CoreCLR** project. The code has an error that PVS-Studio analyzer diagnoses in the following way: V698 Expression 'memcmp(....) == -1' is incorrect. This function can return not only the value '-1', but any negative value. Consider using 'memcmp(....) < 0' instead.

```
bool operator( )(const GUID& _Key1, const GUID& _Key2) const
  { return memcmp(&_Key1, &_Key2, sizeof(GUID)) == -1; }
```

Explanation

Let's have a look at the description of *memcmp()* function:

int memcmp (const void * ptr1, const void * ptr2, size_t num);

Compares the first num bytes of the block of memory pointed by ptr1 to the first num bytes pointed by ptr2, returning zero if they all match, or a value different from zero representing which is greater, if they do not.

Return value:

- < 0 - the first byte that does not match in both memory blocks has a lower value in ptr1 than in ptr2 (if evaluated as unsigned char values).
- == 0 - the contents of both memory blocks are equal.
- > 0 - the first byte that does not match in both memory blocks has a greater value in ptr1 than in ptr2 (if evaluated as unsigned char values).

Note that if blocks aren't the same, then the function returns values greater than or less than zero. Greater or less. This is important! You cannot compare the results of such functions as *memcmp()*, *strcmp()*, *strncmp()*, and so on with the constants 1 and -1.

Interestingly, the wrong code, where the result is compared with the 1/ -1 can work as the programmer expects for many years. But this is sheer luck, nothing more. The behavior of the function can unexpectedly change. For example, you may change the compiler, or the developers will optimize *memcmp()* in a new way, so your code will cease working.

Correct code

```
bool operator( )(const GUID& _Key1, const GUID& _Key2) const
  { return memcmp(&_Key1, &_Key2, sizeof(GUID)) < 0; }
```

Recommendation

Don't rely on the way the function works now. If the documentation says that a function can return values less than or greater than 0, it does mean it. It means that the function can return -10, 2, or 1024. The fact that you always see it return -1, 0, or 1 doesn't prove anything.

By the way, the fact that the function can return such numbers as 1024, indicates, that the result of *memcmp()* execution cannot be stored in the variable of *char* type. This is one more wide-spread error, whose consequences can be really serious. Such a mistake was the root of a serious vulnerability in MySQL/MariaDB in versions earlier than 5.1.61, 5.2.11, 5.3.5, 5.5.22. The thing is, that when a user connects to MySQL/MariaDB, the code evaluates a token (SHA from the password and hash) that is then compared with the expected value of *memcmp()* function. But on some platforms the return value can go beyond the range [-128..127] As a result, in 1 out of 256 cases the procedure of comparing hash with an expected value always returns *true*, regardless of the hash. Therefore, a simple command on bash gives a hacker root access to the volatile MySQL server, even if the person doesn't know the password. The reason for this was the following code in the file 'sql/password.c':

```
typedef char my_bool;
...
my_bool check(...) {
  return memcmp(...);
}
```

A more detailed description of this issue can be found here: Security vulnerability in MySQL/MariaDB.

3. Copy once, check twice

The fragment is taken from **Audacity** project. The error is detected by the following PVS-Studio diagnostic: V501 There are identical sub-expressions to the left and to the right of the '-' operator.

```
sampleCount VoiceKey::OnBackward (....) {
```

```
  ...
  int atrend = sgn(buffer[samplesleft - 2]-
                   buffer[samplesleft - 1]);
  int ztrend = sgn(buffer[samplesleft - WindowSizeInt-2]-
                   buffer[samplesleft - WindowSizeInt-2]);
  ...
}
```

Explanation

The "buffer[samplesleft - WindowSizeInt-2]" expression is subtracted from itself. This error appeared because of copying a code fragment (Copy-Paste): the programmer copied a code string but forgot to replace 2 with 1.

This a really banal error, but still it is a mistake. Errors like this are a harsh reality for programmers, and that's why there will speak about them several times here. I am declaring war on them.

Correct code

```
int ztrend = sgn(buffer[samplesleft - WindowSizeInt-2]-
                 buffer[samplesleft - WindowSizeInt-1]);
```

Recommendation

Be very careful when duplicating code fragments.

It wouldn't make sense to recommend rejecting the copy-paste method altogether. It's too convenient, and too useful to get rid of such an editor functionality.

Instead, just be careful, and don't hurry - forewarned is forearmed.

Remember that copying code may cause many errors. Here, take a look at some examples of bugs detected with the V501 diagnostic. Half of these errors are caused by using Copy-Paste.

If you copy the code and then edit it - check what you've got! Don't be lazy!

We'll talk more about Copy-Paste later. The problem actually goes deeper than it may seem, and I won't let you forget about it.

4. Beware of the ?: operator and enclose it in parentheses

Fragment taken from the **Haiku** project (inheritor of BeOS). The error is detected by the following PVS-Studio diagnostic: V502 Perhaps the '?:' operator works in a different way than it was expected. The '?:' operator has a lower priority than the '-' operator.

```
bool IsVisible(bool ancestorsVisible) const
{
  int16 showLevel = BView::Private(view).ShowLevel();
  return (showLevel - (ancestorsVisible) ? 0 : 1) <= 0;
}
```

Explanation

Let's check the C/C++ operation precedence. The ternary operator ?: has a very low precedence, lower than that of operations /, +, <, etc; it is also lower than the precedence of the minus operator. As a result, the program doesn't work in the way the programmer expected.

The programmer thinks that the operations will execute in the following order:

```
(showLevel - (ancestorsVisible ? 0 : 1) ) <= 0
```

But it will actually be like this:

```
((showLevel - ancestorsVisible) ? 0 : 1) <= 0
```

The error is made in very simple code. This illustrates how hazardous the ?: operator is. It's very easy to make a mistake when using it; the ternary operator in more complex conditions is pure damage to the code. It's not only that you are very likely to make and miss a mistake; such expressions are also very difficult to read.

Really, beware of the ?: operator. I've seen a lot of bugs where this operator was used.

Correct code

```
return showLevel - (ancestorsVisible ? 0 : 1) <= 0;
```

Recommendation

In previous articles, we've already discussed the problem of a ternary operator, but since then I've become even more paranoid. The example given above shows how easy it is to make an error, even in a short and simple expression, that's why I'll modify my previous tips.

I don't suggest rejecting the ?: operator completely. It may be useful, and even necessary sometimes. Nevertheless, please do not overuse it, and if you have decided to use it, here is my recommendation:

ALWAYS enclose the ternary operator in parentheses.

Suppose you have an expression:

A = B ? 10 : 20;

Then you should write it like this:

A = (B ? 10 : 20);

Yes, the parentheses are excessive here...

But, it will protect your code later when you or your colleagues add an X variable to 10 or 20 while doing code refactoring:

A = X + (B ? 10 : 20);

Without the parentheses, you could forget that the ?: operator has low precedence, and accidentally break the program.

Of course, you can write "X+" inside the parentheses, but it will still lead to the same error, although it is additional protection that shouldn't be rejected.

5. Use available tools to analyze your code

The fragment is taken from **LibreOffice** project. The error is detected by the following PVS-Studio diagnostic: V718 The 'CreateThread' function should not be called from 'DllMain' function.

```
BOOL WINAPI DllMain( HINSTANCE hinstDLL,
```

```
                    DWORD fdwReason, LPVOID lpvReserved )
{
  ....
  CreateThread( NULL, 0, ParentMonitorThreadProc,
                (LPVOID)dwParentProcessId, 0, &dwThreadId );
  ....
}
```

Explanation

I used to have a side job as a freelancer long time ago. Once I was given a task I failed to accomplish. The task itself was formulated incorrectly, but I didn't realise that at the time. Moreover, it seemed clear and simple at first.

Under a certain condition in the *DllMain* I had to do some actions, using Windows API functions; I don't remember which actions exactly, but it wasn't anything difficult.

So I spent loads of time on that, but the code just wouldn't work. More than that, when I made a new standard application, it worked; but it didn't when I tried it in the *DllMain* function. Some magic, isn't it? I didn't manage to figure out the root of the problem at the time.

It's only now that I work on PVS-Studio development, so many years later, that I have suddenly realized the reason behind that old failure. In the DllMain function, you can perform only a very limited set of actions. The thing is that some DLL may be not loaded yet, and you cannot call functions from them.

Now we have a diagnostic to warn programmers when dangerous operations are detected in *DllMain* functions. So it was this, which was the case with that old task I was working on.

Details

More details about the usage of *DllMain* can be found on the MSDN site in this article: Dynamic-Link Library Best Practices. I'll give some abstracts from it here:

DllMain is called while the loader-lock is held. Therefore, significant restrictions are imposed on the functions that can be called within *DllMain.* As such, *DllMain* is designed to perform minimal initialization tasks, by using a small subset of the Microsoft Windows API. You cannot call any function in *DllMain* which directly, or indirectly, tries to acquire the loader lock. Otherwise, you will introduce the possibility that your application deadlocks or crashes. An error in a *DllMain* implementation can jeopardize the entire process and all of its threads.

The ideal *DllMain* would be just an empty stub. However, given the complexity of many applications, this is generally too restrictive. A good rule of thumb for *DllMain* is to postpone the initialization as for as long as possible. Slower initialization increases how robust the application is, because this initialization is not performed while the loader lock is held. Also, slower initialization enables you to safely use much more of the Windows API.

Some initialization tasks cannot be postponed. For example, a DLL that depends on a configuration file will fail to load if the file is malformed or contains garbage. For this type of initialization, the DLLs should attempt to perform the action, and in the case of a failure, exit immediately rather than waste resources by doing some other work.

You should never perform the following tasks from within *DllMain*:

- Call LoadLibrary or LoadLibraryEx (either directly or indirectly). This can cause a deadlock or a crash.
- Call GetStringTypeA, GetStringTypeEx, or GetStringTypeW (either directly or indirectly). This can cause a deadlock or a crash.
- Synchronize with other threads. This can cause a deadlock.
- Acquire a synchronization object that is owned by code that is waiting to acquire the loader lock. This can cause a deadlock.
- Initialize COM threads by using CoInitializeEx Under certain conditions, this function can call *LoadLibraryEx*.
- Call the registry functions. These functions are implemented in Advapi32.dll. If Advapi32.dll is not initialized before your DLL, the DLL can access uninitialized memory and cause the process to crash.
- Call *CreateProcess*. Creating a process can load another DLL.
- Call *ExitThread*. Exiting a thread during DLL detach can cause the loader lock to be acquired again, causing a deadlock or a crash.
- Call *CreateThread*. Creating a thread can work if you do not synchronize with other threads, but it is risky.
- Create a named pipe or other named object (Windows 2000 only). In Windows 2000, named objects are provided by the Terminal Services DLL. If this DLL is not initialized, calls to the DLL can cause the process to crash.
- Use the memory management function from the dynamic C Run-Time (CRT). If the CRT DLL is not initialized, calls to these functions can cause the process to crash.
- Call functions in User32.dll or Gdi32.dll. Some functions load another DLL, which may not be initialized.
- Use managed code.

Correct code

The code fragment from the LibreOffice project cited above may or may not work - it all a matter of chance.

It's not easy to fix an error like this. You need refactor your code in order to make the DllMain function as simple, and short, as possible.

Recommendation

It's hard to give recommendations. You can't know everything; everyone may encounter a mysterious error like this. A formal recommendation would sound like this: you should carefully read all the documentation for every ~~program~~ entity you work with. But you surely understand that one can't foresee every possible issue. You'd only spend all your time reading documentation then, have no time for programming. And even having read N pages, you couldn't be sure you haven't missed some article that could warn you against some trouble.

I wish I could give you somewhat more practical tips, but there is unfortunately only one thing I can think of: use static analyzers. No, it doesn't guarantee you will have zero bugs. Had there been an analyzer all those years ago, which could have told me that I couldn't call the *Foo* function in *DllMain*, I would have saved a lot of time and even more nerves: I really was angry, and going crazy, because of not being able to solve the task.

6. Check all the fragments where a pointer is explicitly cast to integer types

The fragment is taken from **IPP Samples** project. The error is detected by the following PVS-Studio diagnostic: V205 Explicit conversion of pointer type to 32-bit integer type: (unsigned long)(img)

```
void write_output_image(...., const Ipp32f *img,
                        ...., const Ipp32s iStep) {
  ...
  img = (Ipp32f*)((unsigned long)(img) + iStep);
  ...
}
```

Note. Some may say that this code isn't the best example for several reasons. We are not concerned about why a programmer would need to move along a data buffer in such a strange way. What matters to us is the fact that the pointer is explicitly cast to the "unsigned long" type. And only this. I chose this example purely because it is brief.

Explanation

A programmer wants to shift a pointer at a certain number of bytes. This code will execute correctly in Win32 mode because the pointer size is the same as that of the *long* type. But if we compile a 64-bit version of the program, the pointer will become 64-bit, and casting it to *long* will cause the loss of the higher bits.

Note. Linux uses a different data model. In 64-bit Linux programs, the 'long' type is 64-bit too, but it's still a bad idea to use 'long' to store pointers there. First, such code tends to get into Windows applications quite often, where it becomes incorrect. Second, there are special types whose very names suggest that they can store pointers - for example, intptr_t. Using such types makes the program clearer.

In the example above, we can see a classic error which occurs in 64-bit programs. It should be said right off that there are lots of other errors, too, awaiting programmers in their way of 64-bit software development. But it is the writing of a pointer into a 32-bit integer variable that's the most widespread and insidious issue.

This error can be illustrated in the following way:

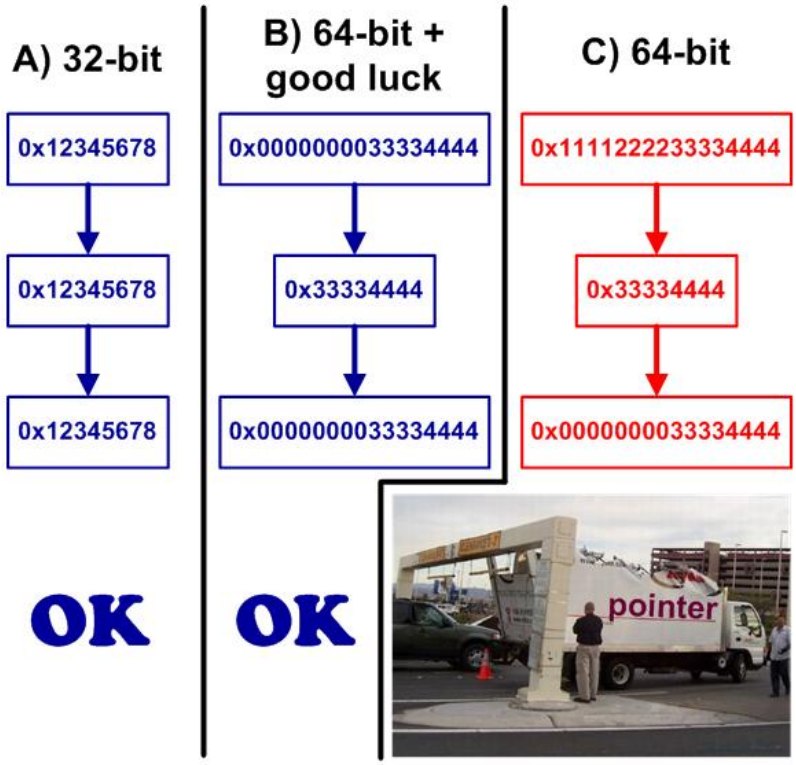


Figure 1. A) 32-bit program. B) 64-bit pointer refers to an object that is located in the lower addresses. C) 64-bit pointer is damaged.

Speaking about its insidiousness, this error is sometimes very difficult to notice. The program just "almost works". Errors causing the loss of the most significant bits in pointers may only show up in a few hours of intense use of the program. First, the memory is allocated in the lower memory addresses, that's why all the objects and arrays are stored in the first 4 GB of memory. Everything works fine.

As the program keeps running, the memory gets fragmented, and even if the program doesn't use much of it, new objects may be created outside those first 4 GB. This is where the troubles start. It's extremely difficult to purposely reproduce such issues.

Correct code

You can use such types as *size_t, INT_PTR, DWORD_PTR, intrptr_t*, etc. to store pointers.

```
img = (Ipp32f*)((uintptr_t)(img) + iStep);
```

Actually, we can do it without any explicit casting. It is not mentioned anywhere that the formatting is different from the standard one, that's why there is no magic in using *__declspec(align(#))* and so on. So, the pointers are shifted by the number of bytes that is divisible by Ipp32f; otherwise we will have undefined behavior (see EXP36-C)

So, we can write it like this:

```
img += iStep / sizeof(*img);
```

Recommendation

Use special types to store pointers - forget about *int* and *long*. The most universal types for this purpose are *intptr_t* and *uintptr_t.* In Visual C++, the following types are available: *INT_PTR*, *UINT_PTR*, *LONG_PTR*, *ULONG_PTR*, *DWORD_PTR*. Their very names indicate that you can safely store pointers in them.

A pointer can fit into the types *size_t* and *ptrdiff_t* too, but I still wouldn't recommend using them for that, for they are originally intended for storing sizes and indices.

You cannot store a pointer to a member function of the class in *uintptr_t.* Member functions are slightly different from standard functions. Except for the pointer itself, they keep hidden value of *this* that points to the object class. However, it does not matter - in the 32-bit program, you can not assign such a pointer to *unsigned int*. Such pointers are always handled in a special way, that's why there aren't many problems in 64-bit programs. At least I haven't seen such errors.

If you are going to compile your program into a 64-bit version, first, you need to review and fix all the code fragments where pointers are cast into 32-bit integer types. Reminder - there will be more troublesome fragments in the program, but you should start with the pointers.

For those who are creating or planning to create 64-bit applications, I suggest studying the following resource: Lessons on development of 64-bit C/C++ applications.

7. Do not call the alloca() function inside loops

This bug was found in **Pixie** project. The error is detected by the following PVS-Studio diagnostic: V505 The 'alloca' function is used inside the loop. This can quickly overflow stack.

```
inline  void  triangulatePolygon(....) {
  ...
  for (i=1;i<nloops;i++) {
    ...
    do {
      ...
      do {
        ...
        CTriVertex *snVertex =
          (CTriVertex *) alloca(2*sizeof(CTriVertex));
        ...
      } while(dVertex != loops[0]);
      ...
    } while(sVertex != loops[i]);
    ...
  }
  ...
}
```

Explanation

The *alloca(size_t)* function allocates memory by using the stack. Memory allocated by alloca() is freed when leaving the function.

There's not much stack memory usually allocated for programs. When you create a project in Visual C++, you may see that the default setting is just 1 megabyte for the stack memory size, this is why the *alloca()* function can very quickly use up all the available stack memory if used inside a loop.

In the example above, there are 3 nested loops at once. Therefore, triangulating a large polygon will cause a stack overflow.

It is also unsafe to use such macros as *A2W* in loops as they also contain a call of the *alloca()* function.

As we have already said, by default, Windows-programs use a stack of 1 Megabyte. This value can be changed; in the project settings find and change the parameters 'Stack Reserve Size', and 'Stack Commit Size'. Details: "/STACK (Stack Allocations)". However, we should understand that making the stack size bigger isn't the solution to the problem - you just postpone the moment when the program stack will overflow.

Recommendation

Do not call the *alloca()* function inside loops. If you have a loop and need to allocate a temporary buffer, use one of the following 3 methods to do so:

1. Allocate memory in advance, and then use one buffer for all the operations. If you need buffers of different sizes every time, allocate memory for the biggest one. If that's impossible (you don't know exactly how much memory it will require), use method 2.
2. Make the loop body a separate function. In this case, the buffer will be created and destroyed right off at each iteration. If that's difficult too, there's only method N3 left.
3. Replace *alloca()* with the *malloc()* function or *new* operator, or use a class such as *std::vector*. Take into account that memory allocation will take more time in this case. In the case of using *malloc/new* you will have to think about freeing it. On the other hand, you won't get a stack overflow when demonstrating the program on large data to the customer.

8. Remember that an exception in the destructor is dangerous

This issue was found in **LibreOffice** project. The error is detected by the following PVS-Studio diagnostic: V509 The 'dynamic_cast<T&>' operator should be located inside the try..catch block, as it could potentially generate an exception. Raising exception inside the destructor is illegal.

```
virtual ~LazyFieldmarkDeleter()

{

  dynamic_cast<Fieldmark&>

    (*m_pFieldmark.get()).ReleaseDoc(m_pDoc);

}
```

Explanation

When an exception is thrown in a program, the stack begins to unroll, and objects get destroyed by calling their destructors. If the destructor of an object being destroyed during stack unrolling throws another exception which leaves the destructor, the C++ library will immediately terminate the program by calling the *terminate()* function. What follows from this is the rule that destructors should never let exceptions out. An exception thrown inside a destructor must be handled inside the same destructor.

The code cited above is rather dangerous. The *dynamic_cast* operator will generate a *std::bad_cast* exception if it fails to cast an object reference to the required type.

Likewise, any other construct that can throw an exception is dangerous. For example, it's not safe to use the *new* operator to allocate memory in the destructor. If it fails, it will throw a *std::bad_alloc* exception.

Correct code:

The code can be fixed using the *dynamic_cast* not with a reference, but with the pointer. In this case, if it's impossible to convert the type of the object, it won't generate an exception, but will return *nullptr*.

```
virtual ~LazyFieldmarkDeleter()
{
  auto p = dynamic_cast<Fieldmark*>m_pFieldmark.get();
  if (p)
    p->ReleaseDoc(m_pDoc);
}
```

Recommendation

Make your destructors as simple as possible. Destructors aren't meant for memory allocation and file reading.

Of course, it's not always possible to make destructors simple, but I believe we should try to reach that. Besides that, a destructor being complex is generally a sign of a poor class design, and ill-conceived solutions.

The more code you have in your destructor, the harder it is to provide for all possible issues. It makes it harder to tell which code fragment can or cannot throw an exception.

If there is some chance that an exception may occur, a good solution is usually to suppress it by using the *catch(...)*:

```
virtual ~LazyFieldmarkDeleter()
{
  try
  {
    dynamic_cast<Fieldmark&>
      (*m_pFieldmark.get()).ReleaseDoc(m_pDoc);
  }
  catch (...)
  {
    assert(false);
  }
}
```

True, using it may conceal some error in the destructor, but it may also help the application to run more stably in general.

I'm not insisting on configuring destructors to never throw exceptions - it all depends on the particular situation. Sometimes it's rather useful to generate an exception in the destructor. I have seen that in specialized classes, but these were rare cases. These classes are designed in such a way that the objects generate an exception upon the destruction, but if it is a usual class like "own string","dot", "brush" "triangle", "document" and so on, in these cases the exceptions shouldn't be thrown from the destructor.

Just remember that double exception on end cause a program termination, so it's up to you to decide if you want this to happen in your project or not.

9. Use the '\0' literal for the terminal null character

The fragment is taken from **Notepad++** project. The error is detected by the following PVS-Studio diagnostic: The error text: V528 It is odd that pointer to 'char' type is compared with the '\0' value. Probably meant: *headerM != '\0'.

```
TCHAR headerM[headerSize] = TEXT("");

...

size_t Printer::doPrint(bool justDoIt)

{

  ...

  if (headerM != '\0')

  ...

}
```

Explanation

Thanks to this code's author, using the '\0' literal to denote the terminal null character, we can easily spot and fix the error. The author did a good job, but not really.

Imagine this code were written in the following way:

```
if (headerM != 0)
```

The array address is verified against 0. The comparison doesn't make sense as it always *true*. What's that - an error or just a redundant check? It's hard to say, especially if it is someone else's code or code written a long time ago.

But since the programmer used the '\0' literal in this code, we can assume that the programmer wanted to check the value of one character. Besides, we know that comparing the *headerM* pointer with *NULL* doesn't make sense. All of that taken into account, we figure that the programmer wanted to find out if the string is empty or not but made a mistake when writing the check. To fix the code, we need to add a pointer dereferencing operation.

Correct code

```
TCHAR headerM[headerSize] = TEXT("");

...

size_t Printer::doPrint(bool justDoIt)

{
```

```
  ...
  if (*headerM != _T('\0'))
  ...
}
```

Recommendation

The number 0 may denote *NULL*, *false*, the null character '\0', or simply the value 0. So please don't be lazy - avoid using 0 for shorter notations in every single case. It only makes the code less comprehensible, and errors harder to find.

Use the following notations:

- 0 - for integer zero;
- *nullptr* - for null pointers in C++*;*
- *NULL* - for null pointers in C*;*
- '\0', L'\0', _T('\0') - for the terminal null;
- 0.0, 0.0f - for zero in expressions with floating-point types;
- *false, FALSE* - for the value 'false'.

Sticking to this rule will make your code clearer, and make it easier for you and other programmers to spot bugs during code reviews.

10. Avoid using multiple small #ifdef blocks

The fragment is taken from **CoreCLR** project. The error is detected by the following PVS-Studio diagnostic: V522 Dereferencing of the null pointer 'hp' might take place.

```
heap_segment* gc_heap::get_segment_for_loh (size_t size
#ifdef MULTIPLE_HEAPS
                                            , gc_heap* hp
#endif //MULTIPLE_HEAPS
                                            )
{
#ifndef MULTIPLE_HEAPS
    gc_heap* hp = 0;
#endif //MULTIPLE_HEAPS
    heap_segment* res = hp->get_segment (size, TRUE);
    if (res != 0)
    {
#ifdef MULTIPLE_HEAPS
        heap_segment_heap (res) = hp;
#endif //MULTIPLE_HEAPS
  ....
```

```
}
```

Explanation

I believe that *#ifdef/#endif* constructs are evil - an unavoidable evil, unfortunately. They are necessary and we have to use them. So I won't urge you to stop using *#ifdef,* there's no point in that. But I do want to ask you to be careful to not "overuse" it.

I guess many of you have seen code literally stuffed with *#ifdefs*. It's especially painful to deal with code where *#ifdef* is repeated every ten lines, or even more often. Such code is usually system-dependent, and you can't do without using *#ifdef i*n it. That doesn't make you any happier, though.

See how difficult it is to read the code sample above! And it is code reading which programmers have to do as their basic activity. Yes, I do mean it. We spend much more time reviewing and studying existing code than writing new one. That's why code which is hard to read reduces our efficiency so much, and leaves more chance for new errors to sneak in.

Getting back to our code fragment, the error is found in the null pointer dereferencing operation, and occurs when the MULTIPLE_HEAPS macro is not declared. To make it easier for you, let's expand the macros:

```
heap_segment* gc_heap::get_segment_for_loh (size_t size)
{
  gc_heap* hp = 0;
  heap_segment* res = hp->get_segment (size, TRUE);
  ....
```

The programmer declared the *hp* variable, initialized it to *NULL*, and dereferenced it right off. If MULTIPLE_HEAPS hasn't been defined, we'll get into trouble.

Correct code

This error is still living in CoreCLR (12.04.2016) despite a colleague of mine having reported it in the article "25 Suspicious Code Fragments in CoreCLR", so I'm not sure how best to fix this error.

As I see it, since (hp == nullptr), then the 'res' variable should be initialized to some other value, too - but I don't know what value exactly. So we'll have to do without the fix this time.

Recommendations

Eliminate small *#ifdef/#endif* blocks from your code - they make it really hard to read and understand! Code with "woods" of *#ifdefs* is harder to maintain and more prone to mistakes.

There is no recommendation to suit every possible case - it all depends on the particular situation. Anyway, just remember that *#ifdef* is a source of trouble, so you must always strive to keep your code as clear as possible.

Tip N1. Try refusing *#ifdef*.

#ifdef can be sometimes replaced with constants and the usual *if* operator. Compare the following 2 code fragments: A variant with macros:

```
#define DO 1
```

```
#ifdef DO
static void foo1()
{
  zzz();
}
#endif //DO

void F()
{
#ifdef DO
  foo1();
#endif // DO
  foo2();
}
```

This code is hard to read; you don't even feel like doing it. Bet you've skipped it, haven't you? Now compare it to the following:

```
const bool DO = true;

static void foo1()
{
  if (!DO)
    return;
  zzz();
}

void F()
{
  foo1();
  foo2();
}
```

It's much easier to read now. Some may argue the code has become less efficient since there is now a function call and a check in it. But I don't agree with that. First, modern compilers are pretty smart and you are very likely to get the same code without any extra checks and function calls in the release version. Second, the potential performance losses are too small to be bothered about. Neat and clear code is more important.

Tip N2. Make your *#ifdef* blocks larger.

If I were to write the *get_segment_for_loh()* function, I wouldn't use a number of *#ifdefs* there; I'd make two versions of the function instead. True, there'd be a bit more text then, but the functions would be easier to read, and edit too.

Again, some may argue that it's duplicated code, and since they have lots of lengthy functions with *#ifdef* in each, having two versions of each function may cause them to forget about one of the versions when fixing something in the other.

Hey, wait! And why are your functions lengthy? Single out the general logic into separate auxiliary functions - then both of your function versions will become shorter, ensuring that you will easily spot any differences between them.

I know this tip is not a cure-all. But do think about it.

Tip N3. Consider using templates - they might help.

Tip N4. Take your time and think it over before using *#ifdef*. Maybe you can do without it? Or maybe you can do with fewer *#ifdefs*, and keep this "evil" in one place?

11. Don't try to squeeze as many operations as possible in one line

The fragment is taken from **Godot Engine** project. The error is detected by the following PVS-Studio diagnostic: V567 Undefined behavior. The 't' variable is modified while being used twice between sequence points.

```
static real_t out(real_t t, real_t b, real_t c, real_t d)
{
  return c * ((t = t / d - 1) * t * t + 1) + b;
}
```

Explanation

Sometimes, you can come across code fragments where the authors try to squeeze as much logic as possible into a small volume of code, by means of complex constructs. This practice hardly helps the compiler, but it does make the code harder to read and understand for other programmers (or even the authors themselves). Moreover, the risk of making mistakes in such code is much higher, too.

It is in such fragments, where programmers try to put lots of code in just a few lines, that errors related to undefined behavior are generally found. They usually have to do with writing in and reading from one and the same variable within one sequence point. For a better understanding of the issue, we need to discuss in more detail the notions of "undefined behavior" and "sequence point".

Undefined behavior is the property of some programming languages to issue a result that depends on the compiler implementation or switches of optimization. Some cases of undefined behavior (including the one being discussed here) are closely related to the notion of a "sequence point".

A sequence point defines any point in a computer program's execution at which it is guaranteed that all side effects of previous evaluations will have been performed, and no side effects from subsequent evaluations have yet been revealed. In C/C++ programming languages there are following sequence points:

- sequence points for operators "&&", "||", ",". When not overloaded, these operators guarantee left-to-right execution order;

- sequence point for ternary operator "?:";
- sequence point at the end of each full expression (usually marked with ';');
- sequence point in place of the function call, but after evaluating the arguments;
- sequence point when returning from the function.

Note. *The new C++ standard has discarded the notion of a "sequence point", but we'll be using the above given explanation to let those of you unfamiliar with the subject, grasp the general idea easier and faster. This explanation is simpler than the new one, and is sufficient for us to understand why one shouldn't squeeze lots of operations into one "pile".*

In the example we have started with, there is none of the above mentioned sequence points, while the '=' operator, as well as the parentheses, can't be treated as such. Therefore, we cannot know which value of the *t* variable will be used when evaluating the return value.

In other words, this expression is one single sequence point, so it is unknown in what order the *t* variable will be accessed. For instance, the "t * t" subexpression may be evaluated before or after writing into the " t = t / d - 1" variable.

Correct code

```
static real_t out(real_t t, real_t b, real_t c, real_t d)
{
  t = t / d - 1;
  return c * (t * t * t + 1) + b;
}
```

Recommendation

It obviously wasn't a good idea to try to fit the whole expression in one line. Besides it being difficult to read, it also made it easier for an error to sneak in.

Having fixed the defect and split the expression into two parts, we have solved 2 issues at once - made the code more readable, and gotten rid of undefined behavior by adding a sequence point.

The code discussed above is not the only example, of course. Here's another:

```
*(mem+addr++) =
    (opcode >= BENCHOPCODES) ? 0x00 : ((addr >> 4)+1) << 4;
```

Just as in the previous case, the error in this code has been caused by unreasonably complicated code. The programmer's attempt to increment the *addr* variable within one expression has led to undefined behavior as it is unknown which value the *addr* variable will have in the right part of the expression - the original or the incremented one.

The best solution to this problem is the same as before - do not complicate matters without reason; arrange operations in several expressions instead of putting them all in one:

```
*(mem+addr) = (opcode >= BENCHOPCODES) ? 0x00 : ((addr >> 4)+1) << 4;
addr++;
```

There is a simple yet useful conclusion to draw from all of this - do not try to fit a set of operations in as few lines if possible. It may be more preferable to split the code into several fragments, thus making it more comprehensible, and reducing the chance errors occuring.

Next time you're about to write complex constructs, pause for a while and think what using them will cost you, and if you are ready to pay that price.

12. When using Copy-Paste, be especially careful with the last lines

This bug was found in **Source SDK** library. The error is detected by the following PVS-Studio diagnostic: V525 The code containing the collection of similar blocks. Check items 'SetX', 'SetY', 'SetZ', 'SetZ'.

```
inline void SetX( float val );

inline void SetY( float val );

inline void SetZ( float val );

inline void SetW( float val );


inline void Init( float ix=0, float iy=0,

                  float iz=0, float iw = 0 )

{

  SetX( ix );

  SetY( iy );

  SetZ( iz );

  SetZ( iw );

}
```

Explanation

I'm 100% sure this code was written with the help of Copy-Paste. One of the first lines was copied several times, with certain letters changed in its duplicates. At the very end, this technique failed the programmer: his attention weakened, and he forgot to change letter 'Z' to 'W' in the last line.

In this example, we are not concerned about the fact of a programmer making a mistake; what matters is that it was made at the end of a sequence of monotonous actions.

I do recommend reading the article "The Last Line Effect". Due to public interest a scientific version of it also got published.

Put briefly, when copying code fragments through the Copy-Paste method, it is highly probable that you will make a mistake at the very end of the sequence of copied lines. It's not my guess, it's statistical data.

Correct code

```
{

  SetX( ix );

  SetY( iy );

  SetZ( iz );

  SetW( iw );

}
```

Recommendation

I hope you have already read the article I've mentioned above. So, once again, we are dealing with the following phenomenon. When writing similarly looking code blocks, programmers copy and paste code fragments with slight changes. While doing so, they tend to forget to change certain words or characters, and it most often happens at the end of a sequence of monotonous actions because their attention weakens.

To reduce the number of such mistakes, here are a few tips for you:

1. Arrange your similar looking code blocks in "tables": it should make mistakes more prominent. We will discuss the "table" code layout in the next section. Perhaps in this case the table layout wasn't of much help, but still it's a very useful thing in programming.
2. Be very careful and attentive when using Copy-Paste. Stay focused, and double-check the code you have written - especially the last few lines.
3. You have now learned about the last line effect; try to keep this in mind, and tell your colleagues about it. The very fact of you knowing how such errors occur, should help you avoid them.
4. Share the link to the "The Last Line Effect" article with your colleagues.

13. Table-style formatting

Fragment taken from the **ReactOS** project (open-source operating system compatible with Windows). The error is detected by the following PVS-Studio diagnostic: V560 A part of conditional expression is always true: 10035L.

```
void adns__querysend_tcp(adns_query qu, struct timeval now) {
  ...
  if (!(errno == EAGAIN || EWOULDBLOCK ||
        errno == EINTR || errno == ENOSPC ||
        errno == ENOBUFS || errno == ENOMEM)) {
  ...
}
```

Explanation

The code sample given above is small and you can easily spot the error in it. But when dealing with real-life code, bugs are often very hard to notice. When reading code like that, you tend to unconsciously skip blocks of similar comparisons and go on to the next fragment.

The reason why it happens has to do with the fact that conditions are poorly formatted and you don't feel like paying too much attention to them because it requires certain effort, and we assume that since the checks are similar, there are hardly any mistakes in the condition and everything should be fine.

One of the ways out is formatting the code as a table.

If you felt too lazy to search for an error in the code above, I'll tell you: "errno ==" is missing in one of the checks. It results in the condition always being true as the EWOULDBLOCK is not equal to zero.

Correct code

```
if (!(errno == EAGAIN || errno == EWOULDBLOCK ||
      errno == EINTR || errno == ENOSPC ||
      errno == ENOBUFS || errno == ENOMEM)) {
```

Recommendation

For a start, here's a version of this code formatted in the simplest "table" style. I don't like it actually.

```
if (!(errno == EAGAIN  || EWOULDBLOCK     ||
      errno == EINTR   || errno == ENOSPC ||
      errno == ENOBUFS || errno == ENOMEM)) {
```

It's better now, but not quite.

There are two reasons why I don't like this layout. First, the error is still not much visible; second, you have to insert too many spaces to align the code.

That's why we need to make two improvements in this formatting style. The first one is we need to use no more than one comparison per line: it makes errors easy to notice. For example:

```
a == 1 &&
b == 2 &&
c      &&
d == 3 &&
```

The second improvement is to write operators &&, ||, etc., in a more rational way, i.e. on the left instead of on the right.

See how tedious it is to align code by means of spaces:

```
x == a          &&
y == bbbbb      &&
z == ccccccccc &&
```

Writing operators on the left makes it much faster and easier:

```
   x == a
&& y == bbbbb
&& z == ccccccccc
```

The code looks a bit odd, but you'll get used to it very soon.

Let's combine these two improvements to write our code sample in the new style:

```
if (!(   errno == EAGAIN
      || EWOULDBLOCK
      || errno == EINTR
      || errno == ENOSPC
      || errno == ENOBUFS
      || errno == ENOMEM)) {
```

Yes, it's longer now - yet the error has become clearly seen, too.

I agree that it looks strange, but nevertheless I do recommend this technique. I've been using it myself for half a year now and enjoy it very much, so I'm confident about this recommendation.

I don't find it a problem at all that the code has become longer. I'd even write it in a way like this:

```
const bool error =    errno == EAGAIN
                   || errno == EWOULDBLOCK
                   || errno == EINTR
                   || errno == ENOSPC
                   || errno == ENOBUFS
                   || errno == ENOMEM;
if (!error) {
```

Feel disappointed with the code being too lengthy and cluttered? I agree. So let's make it a function!

```
static bool IsInterestingError(int errno)
{
  return    errno == EAGAIN
         || errno == EWOULDBLOCK
         || errno == EINTR
         || errno == ENOSPC
         || errno == ENOBUFS
         || errno == ENOMEM;
}
....
if (!IsInterestingError(errno)) {
```

You may think that I'm dramatizing things, being too much of a perfectionist. But I assure you that errors are very common in complex expressions, and I wouldn't ever bring them up weren't they 'so frequent. They are everywhere. And they are very difficult to notice.

Here's another example from WinDjView project:

```
inline bool IsValidChar(int c)
{
  return c == 0x9 || 0xA || c == 0xD ||
         c >= 0x20 && c <= 0xD7FF ||
         c >= 0xE000 && c <= 0xFFFD ||
         c >= 0x10000 && c <= 0x10FFFF;
}
```

The function consists of just a few lines, but it still has an error. The function always returns *true*. The reason, in the long run, has to do with poor formatting and programmers maintaining the code for many years being unwilling to read it carefully.

Let's refactor this code in the "table" style, I'd also add some parentheses:

```
inline bool IsValidChar(int c)
{
  return
       c == 0x9
    || 0xA
    || c == 0xD
    || (c >= 0x20    && c <= 0xD7FF)
    || (c >= 0xE000  && c <= 0xFFFD)
    || (c >= 0x10000 && c <= 0x10FFFF);
}
```

You don't have to format your code exactly the way I suggest. The aim of this post is to draw your attention to typos in "chaotically" written code. By arranging it in the "table" style, you can avoid lots of silly typos, and that's already great. So I hope this post will help you.

Note

Being completely honest, I have to warn you that "table" formatting may sometimes cause harm. Check this example:

```
inline
void elxLuminocity(const PixelRGBi& iPixel,
                   LuminanceCell< PixelRGBi >& oCell)
{
  oCell._luminance = 2220*iPixel._red +
                     7067*iPixel._blue +
                     0713*iPixel._green;
  oCell._pixel = iPixel;
}
```

It's taken from the eLynx SDK project. The programmer wanted to align the code, so he added 0 before the value 713. Unfortunately, he forgot that 0 being the first digit in a number means that this number is octal.

An array of strings

I hope that the idea about the table formatting of the code is clear, but I feel like giving couple more examples. Let's have a look at one more case. By bringing it here, I am saying that the table formatting should be used not only with conditions, but also with other various constructions of a language.

The fragment is taken from Asterisk project. The error is detected by the following PVS-Studio diagnostic: V653 A suspicious string consisting of two parts is used for array initialization. It is possible that a comma is missing. Consider inspecting this literal: "KW_INCLUDES" "KW_JUMP".

```
static char *token_equivs1[] =
{
  ....
  "KW_IF",
  "KW_IGNOREPAT",
  "KW_INCLUDES"
  "KW_JUMP",
  "KW_MACRO",
  "KW_PATTERN",
  ....
};
```

There is a typo here - one comma is forgotten. As a result two strings that have completely different meaning are combined in one, i.e. we actually have:

```
  ....
  "KW_INCLUDESKW_JUMP",
  ....
```

The error could be avoided if the programmer used the table formatting. Then, if the comma is omitted, it will be easy to spot.

```
static char *token_equivs1[] =
{
  ....
  "KW_IF"        ,
  "KW_IGNOREPAT" ,
  "KW_INCLUDES"  ,
  "KW_JUMP"      ,
  "KW_MACRO"     ,
  "KW_PATTERN"   ,
  ....
};
```

Just like last time, pay attention, that if we put the delimiter to the right (a comma in this case), you have to add a lot of spaces, which is inconvenient. It is especially inconvenient if there is a new long line/phrase: we will have to reformat the entire table.

That's why I would again recommend formatting the table in the following way:

```
static char *token_equivs1[] =
{
```

```
  ....
  , "KW_IF"
  , "KW_IGNOREPAT"
  , "KW_INCLUDES"
  , "KW_JUMP"
  , "KW_MACRO"
  , "KW_PATTERN"
  ....
};
```

Now it's very easy to spot a missing comma and there is no need to use a lot of spaces - the code is beautiful and intuitive. Perhaps this way of formatting may seem unusual, but you quickly get used to it - try it yourself.

Finally, here is my short motto. **As a rule, beautiful code is usually correct code.**

14. A good compiler and coding style aren't always enough

We have already spoken about good styles of coding, but this time we'll have a look at an anti-example. It's not enough to write good code: there can be various errors and a good programming style isn't always a cure-all.

The fragment is taken from **PostgreSQL.** The error is detected by the following PVS-Studio diagnostic: V575 The 'memcmp' function processes '0' elements. Inspect the third argument.

Cppcheck analyzer can also detect such errors. It issues a warning: Invalid memcmp() argument nr 3. A non-boolean value is required.

```
Datum pg_stat_get_activity(PG_FUNCTION_ARGS)
{
  ....
  if (memcmp(&(beentry->st_clientaddr), &zero_clientaddr,
             sizeof(zero_clientaddr) == 0))
  ....
}
```

Explanation

A closing parenthesis is put in a wrong place. It's just a typo, but unfortunately it completely alters the meaning of the code.

The sizeof(zero_clientaddr) == 0 expression always evaluates to 'false' as the size of any object is always larger than 0. The *false* value turns to 0, which results in the *memcmp()* function comparing 0 bytes. Having done so, the function assumes that the arrays are equal and returns 0. It means that the condition in this code sample can be reduced to *if (false)*.

Correct code

```
if (memcmp(&(beentry->st_clientaddr), &zero_clientaddr,
```

```
            sizeof(zero_clientaddr)) == 0)
```

Recommendation

It's just the case when I can't suggest any safe coding technique to avoid typos. The only thing I can think of is "Yoda conditions", when constants are written to the left of the comparison operator:

```
if (0 == memcmp(&(beentry->st_clientaddr), &zero_clientaddr,
                sizeof(zero_clientaddr)))
```

But I won't recommend this style. I don't like and don't use it for two reasons:

First, it makes conditions less readable. I don't know how to put it exactly, but it's not without reason that this style is called after Yoda.

Second, they don't help anyway if we deal with parentheses put in a wrong place. There are lots of ways you can make a mistake. Here's an example of code where using the Yoda conditions didn't prevent the incorrect arrangement of parentheses:

```
if (0 == LoadStringW(hDllInstance, IDS_UNKNOWN_ERROR,
        UnknownError,
        sizeof(UnknownError) / sizeof(UnknownError[0] -
        20)))
```

This fragment is taken from the ReactOS project. The error is difficult to notice, so let me point it out for you: sizeof(UnknownError[0] - 20).

So Yoda conditions are useless here.

We could invent some artificial style to ensure that every closing parenthesis stands under the opening one. But it will make the code too bulky and ugly, and no one will be willing to write it that way.

So, again, there is no coding style I could recommend to avoid writing closing parentheses in wrong places.

And here's where the compiler should come in handy and warn us about such a strange construct, shouldn't it? Well, it should but it doesn't. I run Visual Studio 2015, specify the /Wall switch... and don't get any warning. But we can't blame the compiler for that, it has enough work to do as it is.

The most important conclusion for us to draw from today's post is that good coding style and compiler (and I do like the compiler in VS2015) do not always make it. I sometimes hear statements like, "You only need to set the compiler warnings at the highest level and use good style, and everything's going to be OK" No, it's not like that. I don't mean to say some programmers are bad at coding; it's just that every programmer makes mistakes. Everyone, no exceptions. Many of your typos are going to sneak past the compiler and good coding style.

So the combo of good style + compiler warnings is important but not sufficient. That's why we need to use a variety of bug search methods. There's no silver bullet; the high quality of code can be only achieved through a combination of several techniques.

The error we are discussing here can be found by means of the following methods:

- code review;
- unit-tests;
- manual testing;

- static code analysis;
- etc.

I suppose you have already guessed that I am personally interested in the static code analysis methodology most of all. By the way, it is most appropriate for solving this particular issue because it can detect errors at the earliest stage, i.e. right after the code has been written.

Indeed, this error can be easily found by such tools as Cppcheck or PVS-Studio.

Conclusion. Some people don't get it that having skill isn't enough to avoid mistakes. Everyone makes them - it's inevitable. Even super-guru make silly typos every now and then. And since it's inevitable, it doesn't make sense blaming programmers, bad compilers, or bad style. It's just not going to help. Instead, we should use a combination of various software quality improving techniques.

15. Start using enum class in your code, if possible

All the examples of this error I have are large. I've picked one of the smallest, but it's still quite lengthy. Sorry for that.

This bug was found in **Source SDK** library. The error is detected by the following PVS-Studio diagnostic: V556 The values of different enum types are compared: Reason == PUNTED_BY_CANNON.

```
enum PhysGunPickup_t
{
  PICKED_UP_BY_CANNON,
  PUNTED_BY_CANNON,
  PICKED_UP_BY_PLAYER,
};

enum PhysGunDrop_t
{
  DROPPED_BY_PLAYER,
  THROWN_BY_PLAYER,
  DROPPED_BY_CANNON,
  LAUNCHED_BY_CANNON,
};

void CBreakableProp::OnPhysGunDrop(...., PhysGunDrop_t Reason)
{
  ....
  if( Reason == PUNTED_BY_CANNON )
  {
    PlayPuntSound();
```

```
  }
  ....
}
```

Explanation

The *Reason* variable is an enumeration of the PhysGunDrop_t type. This variable is compared to the named constant PUNTED_BY_CANNON belonging to another enumeration, this comparison being obviously a logical error.

This bug pattern is quite widespread. I came across it even in such projects as Clang, TortoiseGit, and Linux Kernel.

The reason why it is so frequent is that enumerations are not type safe in the standard C++; you may get easily confused about what should be compared with what.

Correct code

I don't know for sure what the correct version of this code should look like. My guess is that PUNTED_BY_CANNON should be replaced with DROPPED_BY_CANNON or LAUNCHED_BY_CANNON. Let it be LAUNCHED_BY_CANNON.

```
if( Reason == LAUNCHED_BY_CANNON )
{
  PlayPuntSound();
}
```

Recommendation

Consider yourself lucky if you write in C++; I recommend that you start using *enum class* right now and the compiler won't let you compare values, that refer to different enumerations. You won't be comparing pounds with inches anymore.

There are certain innovations in C++ I don't have much confidence in. Take, for instance, the *auto* keyword. I believe it may be harmful when used too often. Here's how I see it: programmers spend more time reading the code rather than writing it, so we must ensure that the program text is easy-to-read. In the C language, variables are declared in the beginning of the function, so when you edit the code in the middle or at the end of it, it's not always easy to figure what some Alice variable actually means. That's why there exists a variety of variable naming notations. For instance, there is a prefix notation, where *pfAlice* may stand for a "pointer to float".

In C++, you can declare variables whenever you need, and it is considered a good style. Using prefixes and suffixes in variable names is no longer popular. And here the *auto* keyword emerges, resulting in programmers starting to use multiple mysterious constructs of the "auto Alice = Foo();" kind again. Alice, who the fuck is Alice?!

Sorry for digressing from our subject. I wanted to show you that some of the new features may do both good and bad. But it's not the case with *enum class*: I do believe it does only good.

When using *enum class*, you must explicitly specify to which enumeration a named constant belongs to. It protects the code from new errors. That is, the code will look like this:

```
enum class PhysGunDrop_t
```

```
{
  DROPPED_BY_PLAYER,
  THROWN_BY_PLAYER,
  DROPPED_BY_CANNON,
  LAUNCHED_BY_CANNON,
};


void CBreakableProp::OnPhysGunDrop(...., PhysGunDrop_t Reason)
{
  ....
  if( Reason == PhysGunDrop_t::LAUNCHED_BY_CANNON )
  {
    PlayPuntSound();
  }
  ....
}
```

True, fixing old code may involve certain difficulties. But I do urge you to start using *enum class* in new code right from this day on. Your project will only benefit from it.

I don't see much point in introducing *enum class*. Here's a few links for you to learn all the details about this new wonderful feature of the C++11 language:

1. Wikipedia. C++11. Strongly typed enumerations.
2. Cppreference. Enumeration declaration.
3. StackOverflow. Why is enum class preferred over plain enum?

16. "Look what I can do!" - Unacceptable in programming

This section will be slightly similar to "Don't try to squeeze as many operations as possible in one line", but this time I want to focus on a different thing. Sometimes it feels like programmers are competing against somebody, trying to write the shortest code possible.

I am not speaking about complicated templates. This is a different topic for discussion, as it is very hard to draw a line between where these templates do harm, and where they do good. Now I am going to touch upon a simpler situation which is relevant for both C and C++ programmers. They tend to make the constructions more complicated, thinking, "I do it because I can".

The fragment is taken from **KDE4** project. The error is detected by the following PVS-Studio diagnostic: V593 Consider reviewing the expression of the 'A = B == C' kind. The expression is calculated as following: 'A = (B == C)'.

```
void LDAPProtocol::del( const KUrl &_url, bool )
{
  ....
```

```
  if ( (id = mOp.del( usrc.dn() ) == -1) ) {
    LDAPErr();
    return;
  }
  ret = mOp.waitForResult( id, -1 );
  ....
}
```

Explanation

After looking at this code, I always have questions such as: What was the point of doing it? Did you want to save a line? Did you want to show that you can combine several actions in one expression?

As a result we have a typical error pattern - using expressions of the if (A = Foo() == Error) kind.

The precedence of the comparison operation is higher than that of the assignment operation. That's why the "mOp.del(usrc.dn()) == -1" comparison is executed first, and only then the *true* (1) or *false* (0) value is assigned to the *id* variable.

If mOp.del() returns '-1', the function will terminate; otherwise, it will keep running and the 'id' variable will be assigned an incorrect value. So it will always equal 0.

Correct code

I want to emphasize: adding extra parentheses is not a solution to the problem. Yes, the error can be eliminated. But it's the wrong way.

There were additional parentheses in the code - have a closer look. It's difficult to say what they were meant for; perhaps the programmer wanted to get rid of the compiler warnings. Perhaps he suspected that the operation priority may be not right, and wanted to fix this issue, but failed to do so. Anyway, those extra brackets don't help.

There is a deeper problem here. If it is a possible not to make the code more complicated, don't. It is better to write:

```
id = mOp.del(usrc.dn());
if ( id == -1 ) {
```

Recommendation

Don't be so lazy as not to write an extra code line: complex expressions are hard to read, after all. Do the assignment first, and only then, the comparison. Thus you will make it easier for programmers who will be maintaining your code later, and also it will reduce the chances of making a mistake.

So my conclusion is - don't try to show off.

This tip sounds trivial, but I hope it will help you. It's always better to write clear and neat code, instead of in a "see how cool I am" style.

17. Use dedicated functions to clear private data

The fragment is taken from the **Apache HTTP Server** project. The error is detected by the following PVS-Studio diagnostic: V597 The compiler could delete the 'memset' function call, which is used to flush 'x' buffer. The RtlSecureZeroMemory() function should be used to erase the private data.

```
static void MD4Transform(
  apr_uint32_t state[4], const unsigned char block[64])
{
  apr_uint32_t a = state[0], b = state[1],
               c = state[2], d = state[3],
               x[APR_MD4_DIGESTSIZE];
  ....
  /* Zeroize sensitive information. */
  memset(x, 0, sizeof(x));
}
```

Explanation

In this code the programmer uses a call of the *memset()* function to erase private data. But it's not the best way to do that because the data won't actually be erased. To be more exact, whether or not they will be erased depends on the compiler, its settings, and the Moon phase.

Try to look at this code from the compiler's viewpoint. It does its best to make your code work as fast as possible, so it carries out a number of optimizations. One of them is to remove the calls of functions which don't affect the program's behavior, and are therefore excessive from the viewpoint of the C/C++ language. This is exactly the case with the *memset()* function in the code sample above. True, this function changes the 'x' buffer, but this buffer is not used anywhere after that, which means the call of the *memset()* function can - and ought to - be deleted.

Important! What I'm telling you now is not a theoretical model of the compiler's behavior - it's a real-life one. In such cases, the compiler does remove the calls of the *memset()* function. You can do a few experiments to check it for yourself. For more details and examples on this issue, please see the following articles:

1. Security, security! But do you test it?
2. Safe Clearing of Private Data.
3. V597. The compiler could delete the 'memset' function call, which is used to flush 'Foo' buffer. The RtlSecureZeroMemory() function should be used to erase the private data
4. Zero and forget -- caveats of zeroing memory in C (see also the discussion of this article).
5. MSC06-C. Beware of compiler optimizations.

What makes this error with removed *memset()* calls especially tricky, is its being very hard to track. When working in the debugger, you will most likely be dealing with un-optimized code, with the function call still there. You can only find the error when studying the assembler listing, generated when building the optimized application version.

Some programmers believe that it has to do with a bug in the compiler, and that it has no right to throw away the calls of such an important function as *memset()*. But this is not the case. This function is by no means more, or less, important than any other, so the compiler has full right to optimize the code where it is called. After all, such code may turn out to be excessive indeed.

Correct code

```
memset_s(x, sizeof(x), 0, sizeof(x));
```

or

```
RtlSecureZeroMemory(x, sizeof(x));
```

Recommendation

You should use special memory clearing functions that the compiler is not allowed to remove for its optimization purposes.

Visual Studio, for instance, offers the *RtlSecureZeroMemory* function; and starting with C11, you can use the *memset_s* function. If necessary, you can even create a safe function of your own - there are lots of examples on the Internet. Here is a couple of them.

Version No.1.

```
errno_t memset_s(void *v, rsize_t smax, int c, rsize_t n) {
  if (v == NULL) return EINVAL;
  if (smax > RSIZE_MAX) return EINVAL;
  if (n > smax) return EINVAL;
  volatile unsigned char *p = v;
  while (smax-- && n--) {
    *p++ = c;
  }
  return 0;
}
```

Version No.2.

```
void secure_zero(void *s, size_t n)
{
    volatile char *p = s;
    while (n--) *p++ = 0;
}
```

Some programmers even go further, and implement functions to fill the array with pseudo-random values, these functions running at different times to ensure better protection from time-measuring attacks. You can find the implementations of such functions on the internet, too.

18. The knowledge you have, working with one language isn't always applicable to another language

The fragment is taken from **Putty** project. Ineffective code is detected by the following PVS-Studio diagnostic: V814 Decreased performance. Calls to the 'strlen' function have being made multiple times when a condition for the loop's continuation was calculated.

```
static void tell_str(FILE * stream, char *str)
{
```

```
  unsigned int i;
  for (i = 0; i < strlen(str); ++i)
    tell_char(stream, str[i]);
}
```

Explanation

There's no actual error here, but such code can be extremely inefficient when we deal with long strings, as the *strlen()* function is called in every loop iteration. So the error, if there is one here, is one of inefficiency.

As a rule, this kind of thing is typically found in code written by those that have previously worked with the Pascal language (or Delphi). In Pascal, the evaluation of the terminating condition of the loop is computed just once, thus this code is suitable and quite commonly used.

Let's have a look at an example of code written in Pascal. The word *called* will be printed only once, because the *pstrlen()* is called only once.

```
program test;
var
  i   : integer;
  str : string;

function pstrlen(str : string): integer;
begin
  writeln('called');
  pstrlen := Length(str);
end;

begin
  str := 'a pascal string';
  for i:= 1 to pstrlen(str) do
    writeln(str[i]);
end.
```

Effective code:

```
static void tell_str(FILE * stream, char *str)
{
  size_t i;
  const size_t len = strlen(str);
  for (i = 0; i < len; ++i)
    tell_char(stream, str[i]);
}
```

Recommendation

Don't forget that in C/C++, loop termination conditions are re-computed at the end of each and every iteration. Therefore it's not a good idea to call inefficient slow functions as part of this evaluation, especially if you can compute it just the once, before the loop is entered.

In some cases the compiler might be able to optimize the code with *strlen()*. For instance, if the pointer always refers to the same string literal, but we shouldn't rely on that in any way.

19. How to properly call one constructor from another

This issue was found in **LibreOffice** project. The error is detected by the following PVS-Studio diagnostic: V603 The object was created but it is not being used. If you wish to call constructor, 'this->Guess::Guess(....)' should be used.

```
Guess::Guess()
{
  language_str = DEFAULT_LANGUAGE;
  country_str = DEFAULT_COUNTRY;
  encoding_str = DEFAULT_ENCODING;
}


Guess::Guess(const char * guess_str)
{
  Guess();
  ....
}
```

Explanation

Good programmers hate writing duplicate code. And that's great. But when dealing with constructors, many shoot themselves in the foot, trying to make their code short and neat.

You see, a constructor can't simply be called like an ordinary function. If we write "A::A(int x) { A(); }", it will lead to creating a temporary unnamed object of the *A* type, instead of calling a constructor without arguments.

This is exactly what happens in the code sample above: a temporary unnamed object *Guess()* is created and gets immediately destroyed, while the class member *language_str* and others remain uninitialized.

Correct code:

There used to be 3 ways to avoid duplicate code in constructors. Let's see what they were.

The first way is to implement a separate initialization function, and call it from both constructors. I'll spare you the examples - it should be obvious as it is.

That's a fine, reliable, clear, and safe technique. However, some bad programmers want to make their code even shorter. So I have to mention two other methods.

They are pretty dangerous, and require you to have a good understanding of how they work, and what consequences you may have to face.

The second way:

```
Guess::Guess(const char * guess_str)
```

```
{
  new (this) Guess();
  ....
}
```

Third way:

```
Guess::Guess(const char * guess_str)
{
  this->Guess();
  ....
}
```

The second and the third variant are rather dangerous because the base classes are initialized twice. Such code can cause subtle bugs, and do more harm than good. Consider an example where such a constructor call is appropriate, where it's not.

Here is a case where everything is fine:

```
class SomeClass
{
  int x, y;
public:
  SomeClass() { new (this) SomeClass(0,0); }
  SomeClass(int xx, int yy) : x(xx), y(yy) {}
};
```

The code is safe and works well since the class only contains simple data types, and is not derived from other classes. A double constructor call won't pose any danger.

And here's another example where explicitly calling a constructor will cause an error:

```
class Base
{
public:
 char *ptr;
 std::vector vect;
 Base() { ptr = new char[1000]; }
 ~Base() { delete [] ptr; }
};


class Derived : Base
{
```

```
  Derived(Foo foo) { }
  Derived(Bar bar) {
     new (this) Derived(bar.foo);
  }
  Derived(Bar bar, int) {
     this->Derived(bar.foo);
  }
}
```

So we call the constructor using the expressions "new (this) Derived(bar.foo);" or "this->Derived(bar.foo)".

The Base object is already created, and the fields are initialized. Calling the constructor once again will cause double initialization. As a result, a pointer to the newly allocated memory chunk will be written into *ptr*, which will result in a memory leak. As for double initialization of an object of the *std::vector* type, the consequences of it are even harder to predict. One thing is clear: code like that is not permissible.

Do you need all that headache, after all? If you can't utilize C++11's features, then use method No. 1 (create an initialization function). An explicit constructor call may be only needed on very rare occasions.

Recommendation

And now we have a feature to help us with the constructors, at last!

C++11 allows constructors to call other peer constructors (known as delegation). This allows constructors to utilize another constructor's behavior with a minimum of added code.

For example:

```
Guess::Guess(const char * guess_str) : Guess()
{
  ....
}
```

To learn more about delegating constructors, see the following links:

1. Wikipedia. C++11. Object construction improvement.
2. C++11 FAQ. Delegating constructors.
3. MSDN. Uniform Initialization and Delegating Constructors.

20. The End-of-file (EOF) check may not be enough

The fragment is taken from **SETI@home** project. The error is detected by the following PVS-Studio diagnostic: V663 Infinite loop is possible. The 'cin.eof()' condition is insufficient to break from the loop. Consider adding the 'cin.fail()' function call to the conditional expression.

```
template <typename T>
std::istream &operator >>(std::istream &i, sqlblob<T> &b)
{
```

```
  ....
  while (!i.eof())
  {
    i >> tmp;
    buf+=(tmp+' ');
  }
  ....
}
```

Explanation

The operation of reading data from a stream object is not as trivial as it may seem at first. When reading data from streams, programmers usually call the *eof()* method to check if the end of stream has been reached. This check, however, is not quite adequate as it is not sufficient and doesn't allow you to find out if any data reading errors or stream integrity failures have occurred, which may cause certain issues.

Note. The information provided in this article concerns both input and output streams. To avoid repetition, we'll only discuss one type of stream here.

This is exactly the mistake the programmer made in the code sample above: in the case of there being any data reading error, an infinite loop may occur as the *eof()* method will always return *false*. On top of that, incorrect data will be processed in the loop, as unknown values will be getting to the *tmp* variable.

To avoid issues like that, we need to use additional methods to check the stream status: *bad()*, *fail()*.

Correct code

Let's take advantage of the fact that the stream can implicitly cast to the *bool* type. The *true* value indicates that the value is read successfully. More details about the way this code works can be found on StackOverflow.

```
template <typename T>
std::istream &operator >>(std::istream &i, sqlblob<T> &b)
{
  ....
  while (i >> tmp)
  {
    buf+=(tmp+' ');
  }
  ....
}
```

Recommendation

When reading data from a stream, don't use the *eof()* method only; check for any failures, too.

Use the methods *bad()* and *fail()* to check the stream status. The first method is used to check stream integrity failures, while the second is for checking data reading errors.

However, it's much more convenient to use *bool()* operator, as it is shown in the example of the correct code.

21. Check that the end-of-file character is reached correctly (EOF)

Let's continue the topic of working with files. And again we'll have a look at EOF. But this time we'll speak about a bug of a completely different type. It usually reveals itself in localized versions of software.

The fragment is taken from **Computational Network Toolkit**. The error is detected by the following PVS-Studio diagnostic: V739 EOF should not be compared with a value of the 'char' type. The 'c' should be of the 'int' type.

```
string fgetstring(FILE* f)
{
  string res;
  for (;;)
  {
    char c = (char) fgetc(f);
    if (c == EOF)
      RuntimeError("error reading .... 0: %s", strerror(errno));
    if (c == 0)
      break;
    res.push_back(c);
  }
  return res;
}
```

Explanation

Let's look at the way EOF is declared:

```
#define EOF (-1)
```

As you can see, the EOF is nothing more than '-1 ' of *int* type. *Fgetc()* function returns a value of *int* type. Namely, it can return a number from 0 to 255 or -1 (EOF). The values read are placed into a variable of *char* type. Because of this, a symbol with the 0xFF (255) value turns into -1, and then is handled in the same way as the end of file (EOF).

Users that use Extended ASCII Codes, may encounter an error when one of the symbols of their alphabet is handled incorrectly by the program.

For example in the Windows 1251 code page, the last letter of Russian alphabet has the 0xFF code, and so, is interpreted by the program as the end-of-file character.

Correct code

```
for (;;)
{
```

```
  int c = fgetc(f);
  if (c == EOF)
    RuntimeError("error reading .... 0: %s", strerror(errno));
  if (c == 0)
    break;
  res.push_back(static_cast<char>(c));
}
```

Recommendation

There is probably no particular recommendation here, but as we are speaking about EOF, I wanted to show an interesting variant of an error, that some people aren't aware of.

Just remember, if the functions return the values of *int* type, don't hasten to change it into *char*. Stop and check that everything is fine. By the way, we have already had a similar case discussing the function *memcmp()* in Chapter N2 - "Larger than 0 does not mean 1" (See the fragment about a vulnerability in MySQL)

22. Do not use #pragma warning(default:X)

The fragment is taken from **TortoiseGIT** project. The error is detected by the following PVS-Studio diagnostic: V665 Possibly, the usage of '#pragma warning(default: X)' is incorrect in this context. The '#pragma warning(push/pop)' should be used instead.

```
#pragma warning(disable:4996)
LONG result = regKey.QueryValue(buf, _T(""), &buf_size);
#pragma warning(default:4996)
```

Explanation

Programmers often assume that warnings disabled with the "pragma warning(disable: X)" directive earlier will start working again after using the "pragma warning(default : X)" directive. But it is not so. The 'pragma warning(default : X)' directive sets the 'X' warning to the DEFAULT state, which is quite not the same thing.

Suppose that a file is compiled with the /Wall switch used. The C4061 warning must be generated in this case. If you add the "#pragma warning(default : 4061)" directive, this warning will not be displayed, as it is turned off by default.

Correct code

```
#pragma warning(push)
#pragma warning(disable:4996)
LONG result = regKey.QueryValue(buf, _T(""), &buf_size);
#pragma warning(pop)
```

Recommendation

The correct way to return the previous state of a warning is to use directives "#pragma warning(push[,n])" and "#pragma warning(pop)". See the Visual C++ documentation for descriptions of these directives: Pragma Directives. Warnings.

Library developers should pay special attention to the V665 warning. Careless warning customization may cause a whole lot of trouble on the library users' side.

A good article on this topic: So, You Want to Suppress This Warning in Visual C++

23. Evaluate the string literal length automatically

The fragment is taken from the **OpenSSL** library. The error is detected by the following PVS-Studio diagnostic: V666 Consider inspecting the third argument of the function 'strncmp'. It is possible that the value does not correspond with the length of a string which was passed with the second argument.

```
if (!strncmp(vstart, "ASCII", 5))
  arg->format = ASN1_GEN_FORMAT_ASCII;
else if (!strncmp(vstart, "UTF8", 4))
  arg->format = ASN1_GEN_FORMAT_UTF8;
else if (!strncmp(vstart, "HEX", 3))
  arg->format = ASN1_GEN_FORMAT_HEX;
else if (!strncmp(vstart, "BITLIST", 3))
  arg->format = ASN1_GEN_FORMAT_BITLIST;
else
  ....
```

Explanation

It's very hard to stop using magic numbers. Also, it would be very unreasonable to get rid of such constants as 0, 1, -1, 10. It's rather difficult to come up with names for such constants, and often they will make reading of the code more complicated.

However, it's very useful to reduce the number of magic numbers. For example, it would be helpful to get rid of magic numbers which define the length of string literals.

Let's have a look at the code given earlier. The code was most likely written using the Copy-Paste method. A programmer copied the line:

```
else if (!strncmp(vstart, "HEX", 3))
```

After that "HEX" was replaced by "BITLIST", but the programmer forgot to change 3 to 7. As a result, the string is not compared with "BITLIST", only with "BIT". This error might not be a crucial one, but still it is an error.

It's really bad that the code was written using Copy-Paste. What's worse is that the string length was defined by a magic constant. From time to time we come across such errors, where the string length does not correspond with the indicated number of symbols because of a typo or carelessness of a programmer. So it's quite a typical error, and we have to do something about it. Let's look closely at the question of how to avoid such errors.

Correct code

First it may seem that it's enough to replace *strncmp()* call with *strcmp()*. Then the magic constant will disappear.

```
else if (!strcmp(vstart, "HEX"))
```

Too bad-we have changed the logic of the code work. The *strncmp()* function checks if the string starts with "HEX", and the function *strcmp()* checks if the strings are equal. There are different checks.

The easiest way to fix this is to change the constant:

```
else if (!strncmp(vstart, "BITLIST", 7))
  arg->format = ASN1_GEN_FORMAT_BITLIST;
```

This code is correct, but it is very bad because the magic 7 is still there. That's why I would recommend a different method.

Recommendation

Such an error can be prevented if we explicitly evaluate the string length in the code. The easiest option is to use the *strlen()* function.

```
else if (!strncmp(vstart, "BITLIST", strlen("BITLIST")))
```

In this case it will be much easier to detect a mismatch if you forget to fix one of the strings:

```
else if (!strncmp(vstart, "BITLIST", strlen("HEX")))
```

But the suggested variant has two disadvantages:

1. There is no guarantee that the compiler will optimize the *strlen()* call and replace it with a constant.
2. You have to duplicate the string literal. It does not look graceful, and can be the subject of a possible error.

The first issue can be dealt with by using special structures for literal length evaluation during the compilation phase. For instance, you can use a macro such as:

```
#define StrLiteralLen(arg) ((sizeof(arg) / sizeof(arg[0])) - 1)
....
else if (!strncmp(vstart, "BITLIST", StrLiteralLen("BITLIST")))
```

But this macros can be dangerous. The following code can appear during the refactoring process:

```
const char *StringA = "BITLIST";
if (!strncmp(vstart, StringA, StrLiteralLen(StringA)))
```

In this case *StrLiteralLen* macro will return some nonsense. Depending on the pointer size (4 or 8 byte) we will get the value 3 or 7. But we can protect ourselves from this unpleasant case in C++ language, by using a more complicated trick:

```
template <typename T, size_t N>
char (&ArraySizeHelper(T (&array)[N]))[N];
#define StrLiteralLen(str) (sizeof(ArraySizeHelper(str)) - 1)
```

Now, if the argument of the *StrLiteralLen* macro is a simple pointer, we won't be able to compile the code.

Let's have a look at the second issue (duplicating of the string literal). I have no idea what to say to C programmers. You can write a special macro for it, but personally I don't like this variant. I am not a fan of macros. That's why I don't know what to suggest.

In C++ everything is fabulously awesome. Moreover, we solve the first problem in a really smart way. The template function will be of a great help to us. You can write it in different ways, but in general it will look like this:

```
template<typename T, size_t N>
int mystrncmp(const T *a, const T (&b)[N])
{
  return _tcsnccmp(a, b, N - 1);
}
```

Now the string literal is used only once. The string literal length is evaluated during the compilation phase. You cannot accidentally pass a simple pointer to the function and incorrectly evaluate the string length. Presto!

Summary: try to avoid magic numbers when working with strings. Use macros or template functions; the code will become not only safer, but more beautiful and shorter.

As an example, you can look at the declaration of a function *strcpy_s ()*:

```
errno_t strcpy_s(
   char *strDestination,
   size_t numberOfElements,
   const char *strSource
);
template <size_t size>
errno_t strcpy_s(
   char (&strDestination)[size],
   const char *strSource
); // C++ only
```

The first variant is intended for the C language, or in the case of a buffer size not being known in advance. If we work with the buffer, created on the stack, then we can use the second variant in C++:

```
char str[BUF_SIZE];
strcpy_s(str, "foo");
```

There are no magic numbers, there is no evaluation of the buffer size at all. It's short and sweet.

24. Override and final specifiers should become your new friends

The fragment is taken from the **MFC** library. The error is detected by the following PVS-Studio diagnostic: V301 Unexpected function overloading behavior. See first argument of function 'WinHelpW' in derived class 'CFrameWndEx' and base class 'CWnd'.

```
class CWnd : public CCmdTarget {
```

```
  ....
  virtual void WinHelp(DWORD_PTR dwData,
                       UINT nCmd = HELP_CONTEXT);
  ....
};
class CFrameWnd : public CWnd {
  ....
};
class CFrameWndEx : public CFrameWnd {
  ....
  virtual void WinHelp(DWORD dwData,
                       UINT nCmd = HELP_CONTEXT);
  ....
};
```

Explanation

When you override a virtual function it's quite easy to make an error in the signature and to define a new function, which won't be in any way connected with the function in the base class. There can be various errors in this case.

1. Another type is used in the parameter of the overridden function.
2. The overridden function has a different number of parameters, this can be especially crucial when there are many parameters.
3. The overridden function differs in *const* modifier.
4. The base class function is not a virtual one. It was assumed that the function in the derived class would override it in the base class, but in reality it hides it.

The same error can occur during the change of types or parameter quantity in the existing code, when the programmer changed the virtual function signature in almost the entire hierarchy, but forgot to do it in some derived class.

This error can appear particularly often during the porting process to the 64-bit platform when replacing the *DWORD* type with *DWORD_PTR, LONG* with *LONG_PTR* and so on. Details. This is exactly our case.

Even in the case of such an error the 32-bit system will work correctly, as both *DWORD* and *DWORD_PTR* are synonyms of *unsigned long*; but in 64-bit version there will be an error because *DWORD_PTR* is a synonym of *unsigned __int64* there.

Correct code

```
class CFrameWndEx : public CFrameWnd {
  ....
  virtual void WinHelp(DWORD_PTR dwData,
                       UINT nCmd = HELP_CONTEXT) override;
```

```
  ....
};
```

Recommendation

Now we have a way to protect ourselves from the error we described above. Two new specifiers were added in C++11:

- *Override* - to indicate that the method is overriding a virtual method in a base class
- *Final* - to indicate that derived classes do not need to override this virtual method.

We are interested in the *override* specifier. This is an indication for the compiler to check if the virtual function is really overriding the base class function, and to issue an error if it isn't.

If *override* was used when determining the function WinHelp in the CFrameWndEx class, we would have an error of compilation on a 64-bit version of an application. Thus the error could have been prevented at an early stage.

Always use the *override* specifier (or *final*), when overriding virtual functions. More details about *override* and *final* can be seen here:

- Cppreference.com. override specifier (since C++11)
- Cppreference.com. final specifier (since C++11)
- Wikipedia.org. Explicit overrides and final.
- Stackoverflow.com. 'override' in c++11.

25. Do not compare 'this' to nullptr anymore

The fragment is taken from **CoreCLR** project. This dangerous code is detected by the following PVS-Studio diagnostic: V704 'this == nullptr' expression should be avoided - this expression is always false on newer compilers, because 'this' pointer can never be NULL.

```
bool FieldSeqNode::IsFirstElemFieldSeq()
{
  if (this == nullptr)
    return false;
  return m_fieldHnd == FieldSeqStore::FirstElemPseudoField;
}
```

Explanation

People used to compare *this* pointer with 0 / *NULL* / *nullptr.* It was a common situation when C++ was only in the beginning of its development. We have found such fragments doing "archaeological" research. I suggest reading about them in an article about checking Cfront. Moreover, in those days the value of *this* pointer could be changed, but it was so long ago that it was forgotten.

Let's go back to the comparison of *this* with *nullptr*.

Now it is illegal. According to modern C++ standards, *this* can NEVER be equal to *nullptr*.

Formally the call of the IsFirstElemFieldSeq() method for a null-pointer *this* according to C++ standard leads to undefined behavior.

It seems that if this==0, then there is no access to the fields of this class while the method is executed. But in reality there are two possible unfavorable ways of such code implementation. According to C++ standards, *this* pointer can never be null, so the compiler can optimize the method call, by simplifying it to:

```
bool FieldSeqNode::IsFirstElemFieldSeq()
{
  return m_fieldHnd == FieldSeqStore::FirstElemPseudoField;
}
```

There is one more pitfall, by the way. Suppose there is the following inheritance hierarchy.

```
class X: public Y, public FieldSeqNode { .... };
....
X * nullX = NULL;
X->IsFirstElemFieldSeq();
```

Suppose that the Y class size is 8 bytes. Then the source pointer *NULL* (0x00000000) will be corrected in such a way, so that it points to the beginning of FieldSeqNode sub object. Then you have to offset it to sizeof(Y) byte. So *this* in the IsFirstElemFieldSeq() function will be 0x00000008. The "this == 0" check has completely lost its sense.

Correct code

It's really hard to give an example of correct code. It won't be enough to just remove this condition from the function. You have to do the code refactoring in such a way that you will never call the function, using the null pointer.

Recommendation

So, now the "if (this == nullptr)" is outlawed. However, you can see this code in many applications and libraries quite often (MFC library for instance). That's why Visual C++ is still diligently comparing *this* to 0. I guess the compiler developers are not so crazy as to remove code that has been working properly for a dozen years.

But the law was enacted. So for a start let's avoid comparing *this* to null. And once you have some free time, it will be really useful to check out all the illegal comparisons, and rewrite the code.

Most likely the compilers will act in the following way. First they will give us comparison warnings. Perhaps they are already giving them, I haven't studied this question. And then at some point they'll fully support the new standard, and your code will cease working altogether. So I strongly recommend that you start obeying the law, it will be helpful later on.

P.S. When refactoring you may need the Null object pattern.

Additional links on the topic:

1. Still Comparing "this" Pointer to Null?
2. Diagnostic V704.

26. Insidious VARIANT_BOOL

The fragment is taken from **NAME** project. The code contains an error that PVS-Studio analyzer diagnoses in the following way: V721 The VARIANT_BOOL type is utilized incorrectly. The true value (VARIANT_TRUE) is defined as -1. Inspect the first argument.

```
virtual HRESULT __stdcall
  put_HandleKeyboard (VARIANT_BOOL pVal) = 0;
....
pController->put_HandleKeyboard(true);
```

Explanation:

There is quite a witty quote:

We all truck around a kind of original sin from having learned Basic at an impressionable age. (C) P.J. Plauger

And this hint is exactly on the topic of evil. VARIANT_BOOL type came to us from Visual Basic. Some of our present day programming troubles are connected with this type. The thing is that "true" is coded as -1 in it.

Let's see the declaration of the type and the constants denoting true/false:

typedef short VARIANT_BOOL;

#define VARIANT_TRUE ((VARIANT_BOOL)-1)

#define VARIANT_FALSE ((VARIANT_BOOL)0)

It seems like there is nothing terrible in it. False is 0, and truth is not 0. So, -1 is quite a suitable constant. But it's very easy to make an error by using *true* or *TRUE* instead of *VARIANT_TRUE*.

Correct code

```
pController->put_HandleKeyboard(VARIANT_TRUE);
```

Recommendation

If you see an unknown type, it's better not to hurry, and to look up in the documentation. Even if the type name has a word *BOOL*, it doesn't mean that you can place 1 into the variable of this type.

In the same way programmers sometimes make mistakes, when they use *HRESULT* type, trying to compare it with *FALSE* or *TRUE* and forgetting that:

```
#define S_OK      ((HRESULT)0L)
#define S_FALSE   ((HRESULT)1L)
```

So I really ask you to be very careful with any types which are new to you, and not to hasten when programming.

27. Guileful BSTR strings

Let's talk about one more nasty data type - *BSTR* (Basic string or binary string).

The fragment is taken from **VirtualBox** project. The code contains an error that PVS-Studio analyzer diagnoses in the following way: V745 A 'wchar_t *' type string is incorrectly converted to 'BSTR' type string. Consider using 'SysAllocString' function.

```
....
HRESULT EventClassID(BSTR bstrEventClassID);
....
hr = pIEventSubscription->put_EventClassID(
                    L"{d5978630-5b9f-11d1-8dd2-00aa004abd5e}");
```

Explanation

Here's how a *BSTR* type is declared:

```
typedef wchar_t OLECHAR;
typedef OLECHAR * BSTR;
```

At first glance it seems that "wchar_t *" and *BSTR* are one and the same things. But this is not so, and this brings a lot of confusion and errors.

Let's talk about *BSTR* type to get a better idea of this case.

Here is the information from MSDN site. Reading MSDN documentation isn't much fun, but we have to do it.

A BSTR (Basic string or binary string) is a string data type that is used by COM, Automation, and Interop functions. Use the *BSTR* data type in all interfaces that will be accessed from script. BSTR description:

1. **Length prefix.** A four-byte integer that contains the number of bytes in the following data string. It appears immediately before the first character of the data string. This value does not include the terminating null character.
2. **Data string.** A string of Unicode characters. May contain multiple embedded null characters.
3. **Terminator.** Two null characters.

A *BSTR* is a pointer. The pointer points to the first character of the data string, not to the length prefix. *BSTRs* are allocated using COM memory allocation functions, so they can be returned from methods without concern for memory allocation. The following code is incorrect:

```
BSTR MyBstr = L"I am a happy BSTR";
```

This code builds (compiles and links) correctly, but it will not function properly because the string does not have a length prefix. If you use a debugger to examine the memory location of this variable, you will not see a four-byte length prefix preceding the data string. Instead, use the following code:

```
BSTR MyBstr = SysAllocString(L"I am a happy BSTR");
```

A debugger that examines the memory location of this variable will now reveal a length prefix containing the value 34. This is the expected value for a 17-byte single-character string that is converted to a wide-character string through the inclusion of the "L" string modifier. The debugger will also show a two-byte terminating null character (0x0000) that appears after the data string.

If you pass a simple Unicode string as an argument to a COM function that is expecting a *BSTR*, the COM function will fail.

I hope this is enough to understand why we should separate the *BSTR* and simple strings of *"wchar_t *"* type.

Additional links:

1. MSDN. BSTR.
2. StackOverfow. Static code analysis for detecting passing a wchar_t* to BSTR.
3. StackOverfow. BSTR to std::string (std::wstring) and vice versa.
4. Robert Pittenger. Guide to BSTR and CString Conversions.
5. Eric Lippert. Eric's Complete Guide To BSTR Semantics.

Correct code

```
hr = pIEventSubscription->put_EventClassID(
        SysAllocString(L"{d5978630-5b9f-11d1-8dd2-00aa004abd5e}"));
```

Recommendation

The tip resembles the previous one. If you see an unknown type, it's better not to hurry, and to look it up in the documentation. This is important to remember, so it's not a big deal that this tip was repeated once again.

28. Avoid using a macro if you can use a simple function

The fragment is taken from **ReactOS** project. The code contains an error that PVS-Studio analyzer diagnoses in the following way: V640 The code's operational logic does not correspond with its formatting. The second statement will always be executed. It is possible that curly brackets are missing.

```
#define stat64_to_stat(buf64, buf)    \
    buf->st_dev   = (buf64)->st_dev;    \
    buf->st_ino   = (buf64)->st_ino;    \
    buf->st_mode  = (buf64)->st_mode;   \
    buf->st_nlink = (buf64)->st_nlink;  \
    buf->st_uid   = (buf64)->st_uid;    \
    buf->st_gid   = (buf64)->st_gid;    \
    buf->st_rdev  = (buf64)->st_rdev;   \
    buf->st_size  = (_off_t)(buf64)->st_size;   \
    buf->st_atime = (time_t)(buf64)->st_atime;  \
    buf->st_mtime = (time_t)(buf64)->st_mtime;  \
    buf->st_ctime = (time_t)(buf64)->st_ctime;  \


int CDECL _tstat(const _TCHAR* path, struct _stat * buf)
{
  int ret;
  struct __stat64 buf64;
```

```
  ret = _tstat64(path, &buf64);
  if (!ret)
    stat64_to_stat(&buf64, buf);
  return ret;
}
```

Explanation

This time the code example will be quite lengthy. Fortunately it's rather easy, so it shouldn't be hard to understand.

There was the following idea. If you manage to get file information by means of *_tstat64()* function, then put these data into the structure of *_stat* type. We use a *stat64_to_stat* macro to save data.

The macro is incorrectly implemented. The operations it executes are not grouped in blocks with curly brackets { }. As a result the conditional operator body is only the first string of the macro. If you expand the macro, you'll get the following:

```
if (!ret)
  buf->st_dev   = (&buf64)->st_dev;
buf->st_ino   = (&buf64)->st_ino;
buf->st_mode  = (&buf64)->st_mode;
```

Consequently the majority of the structure members are copied regardless of the whether the information was successfully received or not.

This is certainly an error, but in practice it's not a fatal one. The uninitialized memory cells are just copied in vain. We had a bit of luck here. But I've come across more serious errors, connected with such poorly written macros.

Correct code

The easiest variant is just to add curly brackets to the macro. To add do { .... } while (0) is a slightly better variant. Then after the macro and the function you can put a semicolon ';'.

```
#define stat64_to_stat(buf64, buf)   \
  do { \
    buf->st_dev   = (buf64)->st_dev;   \
    buf->st_ino   = (buf64)->st_ino;   \
    buf->st_mode  = (buf64)->st_mode;  \
    buf->st_nlink = (buf64)->st_nlink; \
    buf->st_uid   = (buf64)->st_uid;   \
    buf->st_gid   = (buf64)->st_gid;   \
    buf->st_rdev  = (buf64)->st_rdev;  \
    buf->st_size  = (_off_t)(buf64)->st_size;  \
```

```
    buf->st_atime = (time_t)(buf64)->st_atime; \
    buf->st_mtime = (time_t)(buf64)->st_mtime; \
    buf->st_ctime = (time_t)(buf64)->st_ctime; \
  } while (0)
```

Recommendation

I cannot say that macros are my favorite. I know there is no way to code without them, especially in C. Nevertheless I try to avoid them if possible, and would like to appeal to you not to overuse them. My macro hostility has three reasons:

- It's hard to debug the code.
- It's much easier to make an error.
- The code gets hard to understand especially when some macros use another macros.

A lot of other errors are connected with macros. The one I've given as an example shows very clearly that sometimes we don't need macros at all. I really cannot grasp the idea of why the authors didn't use a simple function instead. Advantages of a function over a macro:

- The code is simpler. You don't have to spend additional time writing it and, aligning some wacky symbols \.
- The code is more reliable (the error given as an example won't be possible in the code at all)

Concerning the disadvantages, I can only think of optimization. Yes, the function is called but it's not that serious at all.

However, let's suppose that it's a crucial thing to us, and meditate on the topic of optimization. First of all, there is a nice keyword *inline* which you can use. Secondly, it would be appropriate to declare the function as *static*. I reckon it can be enough for the compiler to build in this function and not to make a separate body for it.

In point of fact you don't have to worry about it at all, as the compilers have become really smart. Even if you write a function without any *inline/static*, the compiler will build it in; if it considers that it's worth doing it. But don't really bother going into such details. It's much better to write a simple and understandable code, it'll bring more benefit.

To my mind, the code should be written like this:

```
static void stat64_to_stat(const struct __stat64 *buf64,
                           struct _stat *buf)
{
  buf->st_dev   = buf64->st_dev;
  buf->st_ino   = buf64->st_ino;
  buf->st_mode  = buf64->st_mode;
  buf->st_nlink = buf64->st_nlink;
  buf->st_uid   = buf64->st_uid;
  buf->st_gid   = buf64->st_gid;
  buf->st_rdev  = buf64->st_rdev;
```

```
  buf->st_size  = (_off_t)buf64->st_size;
  buf->st_atime = (time_t)buf64->st_atime;
  buf->st_mtime = (time_t)buf64->st_mtime;
  buf->st_ctime = (time_t)buf64->st_ctime;
}
```

Actually we can make even more improvements here. In C++ for example, it's better to pass not the pointer, but a reference. The usage of pointers without the preliminary check doesn't really look graceful. But this is a different story, I won't talk about it in a section on macros.

29. Use a prefix increment operator (++i) in iterators instead of a postfix (i++) operator

The fragment is taken from the **Unreal Engine 4** project. Ineffective code is detected by the following PVS-Studio diagnostic: V803 Decreased performance. In case 'itr' is iterator it's more effective to use prefix form of increment. Replace iterator++ with ++iterator.

```
void FSlateNotificationManager::GetWindows(....) const
{
  for( auto Iter(NotificationLists.CreateConstIterator());
       Iter; Iter++ )
  {
    TSharedPtr<SNotificationList> NotificationList = *Iter;
    ....
  }
}
```

Explanation

If you hadn't read the title of the article, I think it would've been quite hard to notice an issue in the code. At first sight, it looks like the code is quite correct, but it's not perfect. Yes, I am talking about the postfix increment - 'Iter++'. Instead of a postfix form of the increment iterator, you should rather use a prefix analogue, i.e. to substitute 'Iter++' for '++Iter'. Why should we do it, and what's the practical value of it? Here is the story.

Effective code:

```
for( auto Iter(NotificationLists.CreateConstIterator());
     Iter; ++Iter)
```

Recommendation

The difference between a prefix and a postfix form is well known to everybody. I hope that the internal structure distinctions (which show us the operational principles) are not a secret as well. If you have ever done the operator overloading, then you must be aware of it. If not - I'll give a brief explanation. (All the others can skip this paragraph and go to the one, which follows the code examples with operator overloading)

The prefix increment operator changes an object's state, and returns itself in the changed form. No temporary objects required. Then the prefix increment operator may look like this:

```
MyOwnClass& operator++()
{
  ++meOwnField;
  return (*this);
}
```

A postfix operator also changes the object's state but returns the previous state of the object. It does so by creating a temporary object, then the postfix increment operator overloading code will look like this:

```
MyOwnClass operator++(int)
{
  MyOWnCLass tmp = *this;
  ++(*this);
  return tmp;
}
```

Looking at these code fragments, you can see that an additional operation of creating a temporary object is used. How crucial is it in practice?

Today's compilers are smart enough to do the optimization, and to not create temporary objects if they are of no use. That's why in the Release version it's really hard to see the difference between 'it++' and '++it'.

But it is a completely different story when debugging the program in the Debug-mode. In this case the difference in the performance can be really significant.

For example, in this article there are some examples of estimation of the code running time using prefix and postfix forms of increment operators in the Debug-version. We see that is almost 4 times longer to use the postfix forms.

Those, who will say, "And? In the Release version it's all the same!" will be right and wrong at the same time. As a rule we spend more time working on the Debug-version while doing the Unit-tests, and debugging the program. So quite a good deal of time is spent working with the Debug version of software, which means that we don't want to waste time waiting.

In general I think we've managed to answer the question - "Should we use the prefix increment operator (++i) instead a of postfix operator (i++) for iterators". Yes, you really should. You'll get a nice speed-up in the Debug version. And if the iterators are quite "heavy", then the benefit will be even more appreciable.

References (reading recommendation):

- Is it reasonable to use the prefix increment operator ++it instead of postfix operator it++ for iterators?
- Pre vs. post increment operator - benchmark

30. Visual C++ and wprintf() function

The fragment is taken from **Energy Checker SDK.** The code contains an error that PVS-Studio analyzer diagnoses in the following way: V576 Incorrect format. Consider checking the second actual argument of the 'wprintf' function. The pointer to string of wchar_t type symbols is expected.

```
int main(void) {
  ...
  char *p = NULL;
  ...
  wprintf(
    _T("Using power link directory: %s\n"),
    p
  );
  ...
}
```

Explanation

Note: The first error is in the usage of _T for specifying a string in wide-character format. To use L prefix will be the correct variant here. However this mistake is not a crucial one and is not of a big interest to us. The code simply won't be compiled if we don't use a wide-character format and _T will expand into nothing.

If you want a *wprintf()* function to print a *char** type string, you should use "%S" in the format string.

Many Linux programmers don't see where the pitfall is. The thing is that Microsoft quite strangely implemented such functions as *wsprintf*. If we work in Visual C++ with the *wsprintf* function, then we should use "%s" to print wide-character strings, at the same time to print *char** strings we need "%S". So it's just a weird case. Those who develop cross platform applications quite often fall into this trap.

Correct code

The code I give here as a way to correct the issue is really not the most graceful one, but I still want to show the main point of corrections to make.

```
char *p = NULL;
...
#ifdef defined(_WIN32)
wprintf(L"Using power link directory: %S\n"), p);
#else
wprintf(L"Using power link directory: %s\n"), p);
#endif
```

Recommendation

I don't have any particular recommendation here. I just wanted to warn you about some surprises you may get if you use functions such as *wprintf()*.

Starting from Visual Studio 2015 there was a solution suggested for writing a portable code. For compatibility with ISO C (C99), you should point out to the preprocessor a _CRT_STDIO_ISO_WIDE_SPECIFIERS macro.

In this case the code:

```
const wchar_t *p = L"abcdef";

const char *x = "xyz";

wprintf(L"%S %s", p, x);
```

is correct.

The analyzer knows about _CRT_STDIO_ISO_WIDE_SPECIFIERS and takes it into account when doing the analysis.

By the way, if you turn on the compatibility mode with ISO C (the _CRT_STDIO_ISO_WIDE_SPECIFIERS macro is declared), you can get the old behavior, using the specifier of "%Ts" format.

In general the story about the wide - character symbols is quite intricate, and goes beyond the frames of one short article. To investigate the topic more thoroughly, I recommend doing some reading on the topic:

- Bug 1121290 - distinguish specifier s and ls in the printf family of functions
- MBCS to Unicode conversion in swprintf
- Visual Studio swprintf is making all my %s formatters want wchar_t * instead of char *

31. In C and C++ arrays are not passed by value

The fragment is taken from the game '**Wolf**'. The code contains an error that PVS-Studio analyzer diagnoses in the following way: V511 The sizeof() operator returns size of the pointer, and not of the array, in 'sizeof (src)' expression.

```
ID_INLINE mat3_t::mat3_t( float src[ 3 ][ 3 ] ) {
  memcpy( mat, src, sizeof( src ) );
}
```

Explanation

Sometimes programmers forget that in C/C++ you cannot pass an array to a function by value. This is because a pointer to an array is passed as an argument. Numbers in square brackets mean nothing, they only serve as a kind of hint to the programmer, which array size is supposed to be passed. In fact, you can pass an array of a completely different size. For example, the following code will be successfully compiled:

```
void F(int p[10]) { }

void G()

{

  int p[3];

  F(p);

}
```

Correspondingly, the *sizeof(src)* operator evaluates not the array size, but the size of the pointer. As a result, *memcpy()* will only copy part of the array. Namely, 4 or 8 bytes, depending on the size of the pointer (exotic architectures don't count).

Correct code

The simplest variant of such code can be like this:

```
ID_INLINE mat3_t::mat3_t( float src[ 3 ][ 3 ] ) {
  memcpy(mat, src, sizeof(float) * 3 * 3);
}
```

Recommendation

There are several ways of making your code more secure.

The array size is known. You can make the function take the reference to an array. But not everyone knows that you can do this, and even fewer people are aware of how to write it. So I hope that this example will be interesting and useful:

```
ID_INLINE mat3_t::mat3_t( float (&src)[3][3] )
{
  memcpy( mat, src, sizeof( src ) );
}
```

Now, it will be possible to pass to the function an array only of the right size. And most importantly, the *sizeof()* operator will evaluate the size of the array, not a pointer.

Yet another way of solving this problem is to start using *std::array* class.

The array size is not known. Some authors of books on programming advise to use std::vector class, and other similar classes, but in practice it's not always convenient.

Sometimes you want to work with a simple pointer. In this case you should pass two arguments to the function: a pointer, and the number of elements. However, in general this is bad practice, and it can lead to a lot of bugs.

In such cases, some thoughts given in "C++ Core Guidelines" can be useful to read. I suggest reading "Do not pass an array as a single pointer". All in all it would be a good thing to read the "C++ Core Guidelines" whenever you have free time. It contains a lot of useful ideas.

32. Dangerous printf

The fragment is taken from **TortoiseSVN** project. The code contains an error that PVS-Studio analyzer diagnoses in the following way: V618 It's dangerous to call the 'printf' function in such a manner, as the line being passed could contain format specification. The example of the safe code: printf("%s", str);

```
BOOL CPOFile::ParseFile(....)
{
  ....
  printf(File.getloc().name().c_str());
  ....
}
```

Explanation

When you want to print or, for example, to write a string to the file, many programmers write code that resembles the following:

```
printf(str);

fprintf(file, str);
```

A good programmer should always remember that these are extremely unsafe constructions. The thing is, that if a formatting specifier somehow gets inside the string, it will lead to unpredictable consequences.

Let's go back to the original example. If the file name is "file%s%i%s.txt", then the program may crash or print some rubbish. But that's only a half of the trouble. In fact, such a function call is a real vulnerability. One can attack programs with its help. Having prepared strings in a special way, one can print private data stored in the memory.

More information about these vulnerabilities can be found in this article. Take some time to look through it; I'm sure it will be interesting. You'll find not only theoretical basis, but practical examples as well.

Correct code

```
printf("%s", File.getloc().name().c_str());
```

Recommendation

Printf()-like functions can cause a lot of security related issues. It is better not to use them at all, but switch to something more modern. For example, you may find *boost::format* or *std::stringstream* quite useful.

In general, sloppy usage of the functions *printf()*, *sprintf()*, *fprintf()*, and so on, not only can lead to incorrect work of the program, but cause potential vulnerabilities, that someone can take advantage of.

33. Never dereference null pointers

This bug was found in GIT's source code. The code contains an error that PVS-Studio analyzer diagnoses in the following way: V595 The 'tree' pointer was utilized before it was verified against nullptr. Check lines: 134, 136.

```
void mark_tree_uninteresting(struct tree *tree)

{

  struct object *obj = &tree->object;

  if (!tree)

    return;

  ....

}
```

Explanation

There is no doubt that it's bad practice to dereference a null pointer, because the result of such dereferencing is undefined behavior. We all agree about the theoretical basis behind this.

But when it comes to practice, programmers start debating. There are always people who claim that this particular code will work correctly. They even bet their life for it - it has always worked for them! And

then I have to give more reasons to prove my point. That's why this article topic is another attempt to change their mind.

I have deliberately chosen such an example that will provoke more discussion. After the *tree* pointer is dereferenced, the class member isn't just using, but evaluating, the address of this member. Then if (tree == nullptr), the address of the member isn't used in any way, and the function is exited. Many consider this code to be correct.

But it is not so. You shouldn't code in such a way. Undefined behavior is not necessarily a program crash when the value is written at a null address, and things like that. Undefined behavior can be anything. As soon as you have dereferenced a pointer which is equal to null, you get an undefined behavior. There is no point in further discussion about the way the program will operate. It can do whatever it wants.

One of the signs of undefined behavior is that the compiler can totally remove the "if (!tree) return;" - the compiler sees that the pointer has already been dereferenced, so the pointer isn't null and the compiler concludes that the check can be removed. This is just one of a great many scenarios, which can cause the program to crash.

I recommend having a look at the article where everything is explained in more details: http://www.viva64.com/en/b/0306/

Correct code

```
void mark_tree_uninteresting(struct tree *tree)
{
  if (!tree)
    return;
  struct object *obj = &tree->object;
  ....
}
```

Recommendation

Beware of undefined behavior, even if it seems as if everything is working fine. There is no need to risk that much. As I have already written, it's hard to imagine how it may show its worth. Just try avoiding undefined behavior, even if it seems like everything works fine.

One may think that he knows exactly how undefined behavior works. And, he may think that this means that he is allowed to do something that others can't, and everything will work. But it is not so. The next section is to underline the fact that undefined behavior is really dangerous.

34. Undefined behavior is closer than you think

This time it's hard to give an example from a real application. Nevertheless, I quite often see suspicious code fragments which can lead to the problems described below. This error is possible when working with large array sizes, so I don't know exactly which project might have arrays of this size. We don't really collect 64-bit errors, so today's example is simply contrived.

Let's have a look at a synthetic code example:

```
size_t Count = 1024*1024*1024; // 1 Gb
if (is64bit)
```

```
  Count *= 5; // 5 Gb
char *array = (char *)malloc(Count);
memset(array, 0, Count);

int index = 0;
for (size_t i = 0; i != Count; i++)
  array[index++] = char(i) | 1;

if (array[Count - 1] == 0)
  printf("The last array element contains 0.\n");

free(array);
```

Explanation

This code works correctly if you build a 32-bit version of the program; if we compile the 64-bit version, the situation will be more complicated.

A 64-bit program allocates a 5 GB buffer and initially fills it with zeros. The loop then modifies it, filling it with non-zero values: we use "| 1" to ensure this.

And now try to guess how the code will run if it is compiled in x64 mode using Visual Studio 2015? Have you got the answer? If yes, then let's continue.

If you run a **debug** version of this program, it'll crash because it'll index out of bounds. At some point the index variable will overflow, and its value will become −2147483648 (INT_MIN).

Sounds logical, right? Nothing of the kind! This is an undefined behavior, and anything can happen.

To get more in-depth information, I suggest the following links:

- Integer overflow
- Understanding Integer Overflow in C/C++
- Is signed integer overflow still undefined behavior in C++?

An interesting thing - when I or somebody else says that this is an example of undefined behavior, people start grumbling. I don't know why, but it feels like they assume that they know absolutely everything about C++, and how compilers work.

But in fact they aren't really aware of it. If they knew, they would't say something like this (group opinion):

This is some theoretical nonsense. Well, yes, formally the 'int' overflow leads to an undefined behavior. But it's nothing more but some jabbering. In practice, we can always tell what we will get. If you add 1 to INT_MAX then we'll have INT_MIN. Maybe somewhere in the universe there are some exotic architectures, but my Visual C++ / GCC compiler gives an incorrect result.

And now without any magic, I will give a demonstration of UB using a simple example, and not on some fairy architecture either, but a Win64-program.

It would be enough to build the example given above in the **Release** mode and run it. The program will cease crashing, and the warning "the last array element contains 0" won't be issued.

The undefined behavior reveals itself in the following way. The array will be completely filled, in spite of the fact that the index variable of *int* type isn't wide enough to index all the array elements. Those who still don't believe me, should have a look at the assembly code:

```
  int index = 0;
  for (size_t i = 0; i != Count; i++)
000000013F6D102D  xor         ecx,ecx
000000013F6D102F  nop
    array[index++] = char(i) | 1;
000000013F6D1030  movzx       edx,cl
000000013F6D1033  or          dl,1
000000013F6D1036  mov         byte ptr [rcx+rbx],dl
000000013F6D1039  inc         rcx
000000013F6D103C  cmp         rcx,rdi
000000013F6D103F  jne         main+30h (013F6D1030h)
```

Here is the UB! And no exotic compilers were used, it's just VS2015.

If you replace *int* with *unsigned,* the undefined behavior will disappear. The array will only be partially filled, and at the end we will have a message - "the last array element contains 0".

Assembly code with the *unsigned*:

```
  unsigned index = 0;
000000013F07102D  xor         r9d,r9d
  for (size_t i = 0; i != Count; i++)
000000013F071030  mov         ecx,r9d
000000013F071033  nop         dword ptr [rax]
000000013F071037  nop         word ptr [rax+rax]
    array[index++] = char(i) | 1;
000000013F071040  movzx       r8d,cl
000000013F071044  mov         edx,r9d
000000013F071047  or          r8b,1
000000013F07104B  inc         r9d
000000013F07104E  inc         rcx
000000013F071051  mov         byte ptr [rdx+rbx],r8b
000000013F071055  cmp         rcx,rdi
000000013F071058  jne         main+40h (013F071040h)
```

Correct code

You must use proper data types for your programs to run properly. If you are going to work with large-size arrays, forget about *int* and *unsigned*. So the proper types are *ptrdiff_t*, *intptr_t*, *size_t*, *DWORD_PTR*, *std::vector::size_type* and so on. In this case it is *size_t*:

```
size_t index = 0;
for (size_t i = 0; i != Count; i++)
  array[index++] = char(i) | 1;
```

Recommendation

If the C/C++ language rules result in undefined behavior, don't argue with them or try to predict the way they'll behave in the future. Just don't write such dangerous code.

There are a whole lot of stubborn programmers who don't want to see anything suspicious in shifting negative numbers, comparing *this* with null or signed types overflowing.

Don't be like that. The fact that the program is working now doesn't mean that everything is fine. The way UB will reveal itself is impossible to predict. Expected program behavior is one of the variants of UB.

35. Adding a new constant to enum don't forget to correct switch operators

The fragment is taken from the **Appleseed** project. The code contains an error that PVS-Studio analyzer diagnoses in the following way: V719 The switch statement does not cover all values of the 'InputFormat' enum: InputFormatEntity.

```
enum InputFormat
{
    InputFormatScalar,
    InputFormatSpectralReflectance,
    InputFormatSpectralIlluminance,
    InputFormatSpectralReflectanceWithAlpha,
    InputFormatSpectralIlluminanceWithAlpha,
    InputFormatEntity
};

switch (m_format)
{
  case InputFormatScalar:
    ....
  case InputFormatSpectralReflectance:
  case InputFormatSpectralIlluminance:
    ....
```

```
  case InputFormatSpectralReflectanceWithAlpha:
  case InputFormatSpectralIlluminanceWithAlpha:
    ....
}
```

Explanation

Sometimes we need to add a new item to an existing enumeration (*enum*), and when we do, we also need to proceed with caution - as we will have to check where we have referenced the *enum* throughout all of our code, e.g., in every *switch* statement and *if* chain. A situation like this can be seen in the code given above.

InputFormatEntity was added to the InputFormat - I'm making that assumption based on the fact that the constant has been added to the end. Often, programmers add new constants to the end of enum, but then forget to check their code to make sure that they've dealt with the new constant properly throughout, and corrected the switch operator.

As a result we have a case when "m_format==InputFormatEntity" isn't handled in any way.

Correct code

```
switch (m_format)
{
  case InputFormatScalar:
  ....
  case InputFormatSpectralReflectance:
  case InputFormatSpectralIlluminance:
  ....
  case InputFormatSpectralReflectanceWithAlpha:
  case InputFormatSpectralIlluminanceWithAlpha:
  ....
  case InputFormatEntity:
  ....
}
```

Recommendation

Let's think, how can we reduce such errors through code refactoring? The easiest, but not a very effective solution is to add a "default:", that will cause a message to appear, e.g.:

```
switch (m_format)
{
  case InputFormatScalar:
  ....
  ....
```

```
  default:
    assert(false);
    throw "Not all variants are considered"
}
```

Now if the *m_format* variable is *InputFormatEntity*, we'll see an exception. Such an approach has two big faults:

1. As there is the chance that this error won't show up during testing (if during the test runs, *m_format* is not equal to *InputFormatEntity*), then this error will make its way into the Release build and would only show up later - during runtime at a customer's site. It's bad if customers have to report such problems!

2. If we consider getting into *default* as an error, then you have to write a *case* for all of the enum's possible values. This is very inconvenient, especially if there are a lot of these constants in the enumeration. Sometimes it's very convenient to handle different cases in the *default* section.

I suggest solving this problem in the following way; I can't say that it's perfect, but at least it's something.

When you define an *enum*, make sure you also add a special comment. You can also use a keyword and an enumeration name.

Example:

```
enum InputFormat
{
  InputFormatScalar,
  ....
  InputFormatEntity
  //If you want to add a new constant, find all ENUM:InputFormat.
};


switch (m_format) //ENUM:InputFormat
{
  ....
}
```

In the code above, when you change the InputFormat enum, you are directed to look for "ENUM:InputFormat" in the source code of the project.

If you are in a team of developers, you would make this convention known to everybody, and also add it to your coding standards and style guide. If somebody fails to follow this rule, it will be very sad.

36. If something strange is happening to your PC, check its memory

I think you got pretty tired looking at numerous error patterns. So this time, let's take a break from looking at code.

A typical situation - your program is not working properly. But you have no idea what's going on. In such situations I recommend not rushing to blame someone, but focus on your code. In 99.99% of cases, the root of the evil is a bug that was brought by someone from your development team. Very often this bug is really stupid and banal. So go ahead and spend some time looking for it!

The fact that the bug occurs from time to time means nothing. You may just have a Heisenbug.

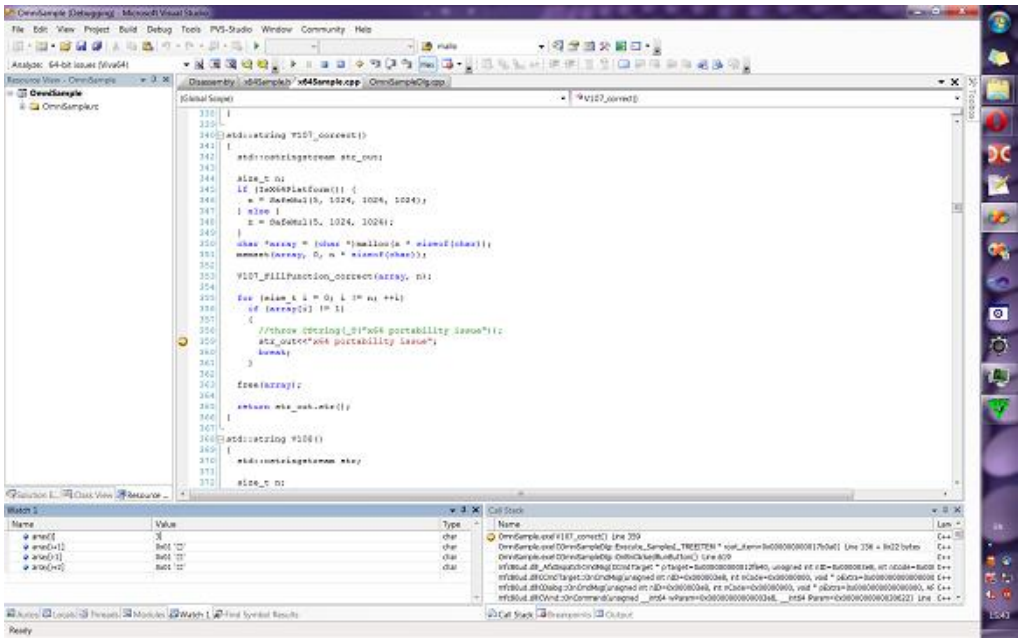
Blaming the compiler would be an even worse idea. It may do something wrong, of course, but very rarely. It will be very awkward if you find out that it was an incorrect use of *sizeof()*, for example. I have a post about that in my blog: The compiler is to blame for everything

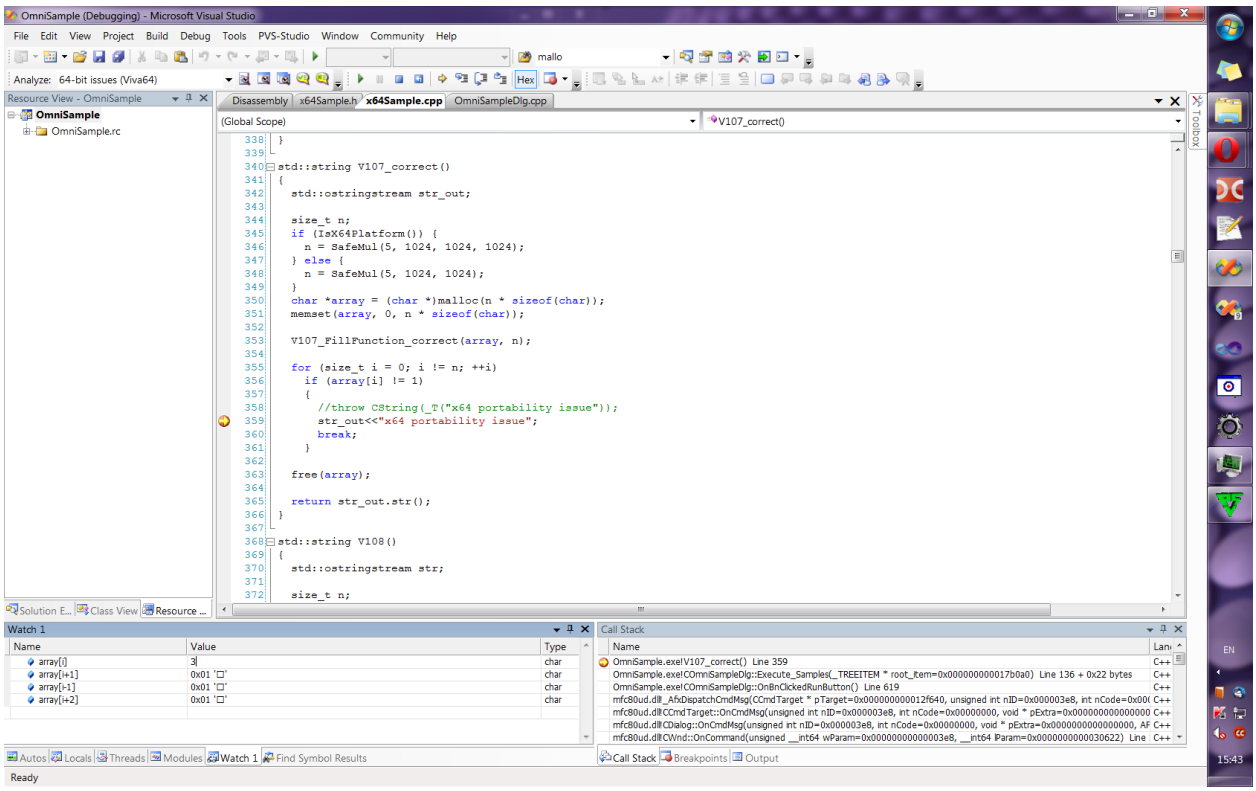
But to set the record straight, I should say that there are exceptions. Very seldom the bug has nothing to do with the code. But we should be aware that such a possibility exists. This will help us to stay sane.

I'll demonstrate this using an example of a case that once happened with me. Fortunately, I have the necessary screenshots.

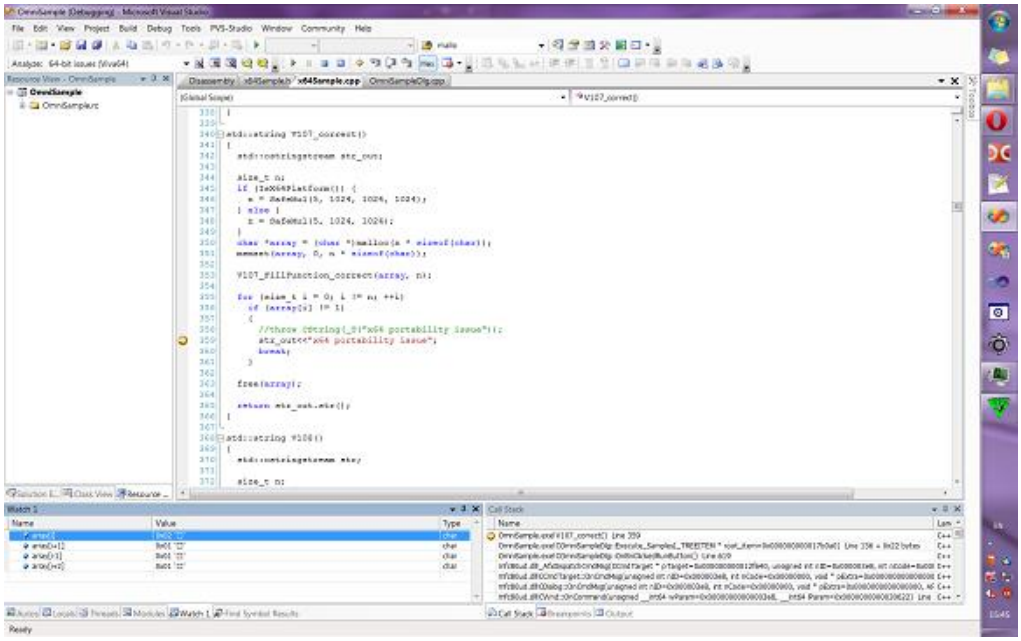
I was making a simple test project that was intended to demonstrate the abilities of the Viva64 analyzer (the predecessor of PVS-Studio), and this project was refusing to work correctly.

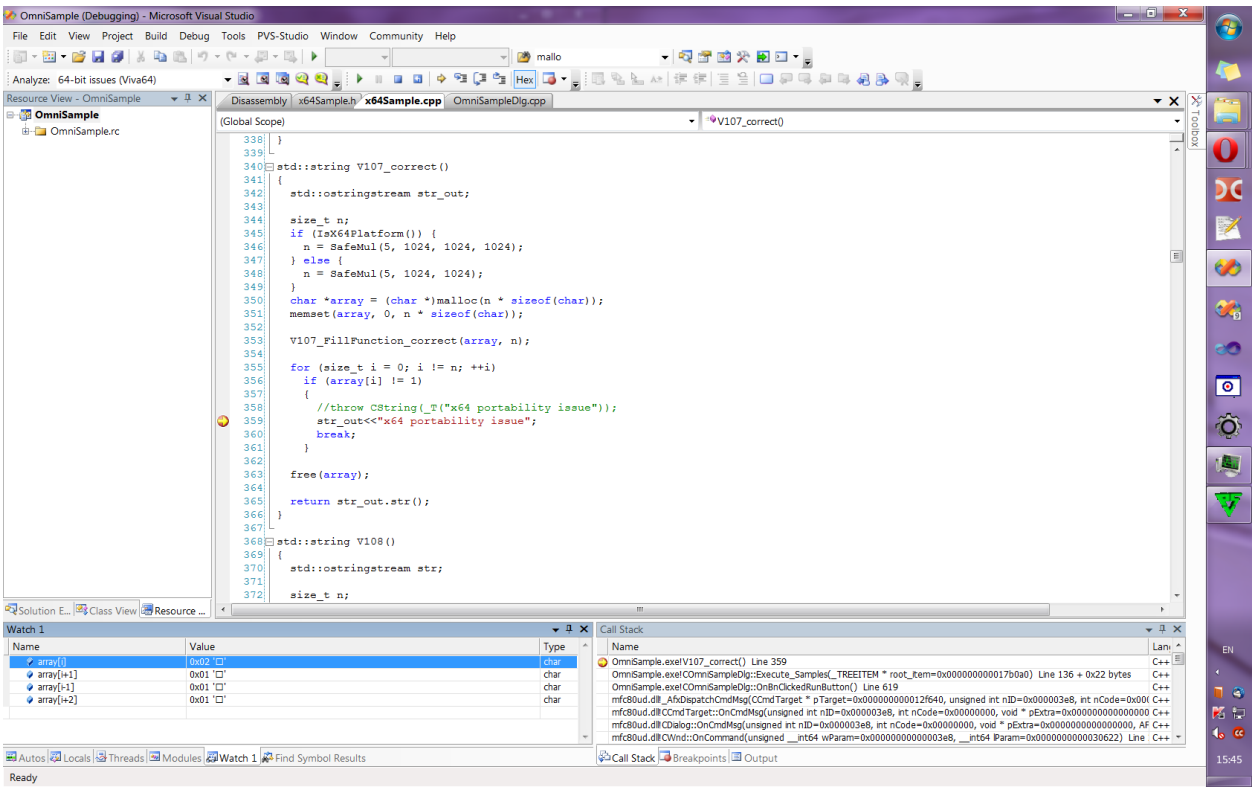
After long and tiresome investigations, I saw that one memory slot is causing all this trouble. One bit, to be exact. You can see on the picture that I am in debug mode, writing the value "3" in this memory cell.





After the memory is changed, the debugger reads the values to display in the window, and shows number 2: See, there is 0x02. Although I've set the "3" value. The low-order bit is always zero.





A memory test program confirmed the problem. It's strange that the computer was working normally without any problems. Replacement of the memory bank finally let my program work correctly.

I was very lucky. I had to deal with a simple test program. And still I spent a lot of time trying to understand what was happening. I was reviewing the assembler listing for more than two hours, trying to find the cause of the strange behavior. Yes, I was blaming the compiler for it.

I can't imagine how much more effort it would take, if it were a real program. Thank God I didn't have to debug anything else at that moment.

Recommendation

Always look for the error in your code. Do not try to shift responsibility.

However, if the bug reoccurs only on your computer for more than a week, it may be a sign that it's not because of your code.

Keep looking for the bug. But before going home, run an overnight RAM test. Perhaps, this simple step will save your nerves.

37. Beware of the 'continue' operator inside do {...} while (...)

Fragment taken from the **Haiku** project (inheritor of BeOS). The code contains an error that PVS-Studio analyzer diagnoses in the following way: V696 The 'continue' operator will terminate 'do { ... } while (FALSE)' loop because the condition is always false.

```
do {

  ....

  if (appType.InitCheck() == B_OK

    && appType.GetAppHint(&hintRef) == B_OK

    && appRef == hintRef)
```

```
  {
    appType.SetAppHint(NULL);
    // try again
    continue;
  }
  ....
} while (false);
```

Explanation

The way *continue* works inside the do-while loop, is not the way some programmers expect it to. When *continue* is encountered, there will always be a check of loop termination condition. I'll try to explain this in more details. Suppose the programmer writes code like this:

```
for (int i = 0; i < n; i++)
{
  if (blabla(i))
    continue;
  foo();
}
```

Or like this:

```
while (i < n)
{
  if (blabla(i++))
    continue;
  foo();
}
```

Most programmers by intuition understand that when continue is encountered, the controlling condition (i < n) will be (re)evaluated, and that the next loop iteration will only start if the evaluation is true. But when a programmer writes code:

```
do
{
  if (blabla(i++))
    continue;
  foo();
} while (i < n);
```

the intuition often fails, as they don't see a condition above the *continue*, and it seems to them that the *continue* will immediately trigger another loop iteration. This is not the case, and *continue* does as it always does - causes the controlling condition to be re-evaluated.

It depends on sheer luck if this lack of understanding of continue will lead to an error. However, the error will definitely occur if the loop condition is always false, as it is in the code snippet given above, where the programmer planned to carry out certain actions through subsequent iterations. A comment in the code "//try again" clearly shows their intention to do so. There will of course be no "again", as the condition is always false, and so once *continue* is encountered, the loop will terminate.

In other words, it turns out that in the construction of this do {...} while (false), the *continue* is equivalent to using *break*.

Correct code

There are many options to write correct code. For example, create an infinite loop, and use *continue* to loop, and *break* to exit.

```
for (;;) {
  ....
  if (appType.InitCheck() == B_OK
    && appType.GetAppHint(&hintRef) == B_OK
    && appRef == hintRef)
  {
    appType.SetAppHint(NULL);
    // try again
    continue;
  }
  ....
  break;
};
```

Recommendation

Try to avoid *continue* inside do { ... } while (...). Even if you really know how it all works. The thing is that you could slip and make this error, and/or that your colleagues might read the code incorrectly, and then modify it incorrectly. I will never stop saying it: a good programmer is not the one who knows and uses different language tricks, but the one who writes clear understandable code, that even a newbie can comprehend.

38. Use nullptr instead of NULL from now on

New C++ standards brought quite a lot of useful changes. There are things which I would not rush into using straight away, but there are some changes which need to be applied immediately, as they will bring with them, significant benefits.

One such modernization is the keyword *nullptr*, which is intended to replace the *NULL* macro.

Let me remind you that in C++ the definition of *NULL* is 0, nothing more.

Of course, it may seem that this is just some syntactic sugar. And what's the difference, if we write *nullptr* or *NULL*? But there is a difference! Using *nullptr* helps to avoid a large variety of errors. I'll show this using examples.

Suppose there are two overloaded functions:

```
void Foo(int x, int y, const char *name);

void Foo(int x, int y, int ResourceID);
```

A programmer might write the following call:

```
Foo(1, 2, NULL);
```

And that same programmer might be sure that he is in fact calling the first function by doing this. It is not so. As *NULL* is nothing more than 0, and zero is known to have *int* type, the second function will be called instead of the first.

However, if the programmer had used *nullptr* no such error would occur and the first function would have been called. Another common enough use of *NULL* is to write code like this:

```
if (unknownError)
  throw NULL;
```

To my mind, it is suspicious to generate an exception passing the pointer. Nevertheless sometimes people do so. Apparently, the developer needed to write the code in this way. However, discussions on whether it is good or bad practice to do so, go beyond the scope of this note.

What is important, is that the programmer decided to generate an exception in the case of an unknown error and "send" a null pointer into the outer world.

In fact it is not a pointer but *int*. As a result the exception handling will happen in a way that the programmer didn't expect.

"throw nullptr;" code saves us from misfortune, but this does not mean that I believe this code to be totally acceptable.

In some cases, if you use *nullptr*, the incorrect code will not compile.

Suppose that some WinApi function returns a *HRESULT* type. The *HRESULT* type has nothing to do with the pointer. However, it is quite possible to write nonsensical code like this:

```
if (WinApiFoo(a, b, c) != NULL)
```

This code will compile, because *NULL* is 0 and of *int* type, and *HRESULT* is a *long* type. It is quite possible to compare values of *int* and *long* type. If you use *nullptr*, then the following code will not compile:

```
if (WinApiFoo(a, b, c) != nullptr)
```

Because of the compiler error, the programmer will notice and fix the code.

I think you get the idea. There are plenty such examples. But these are mostly synthetic examples. And it is always not very convincing. So are there any real examples? Yes, there are. Here is one of them. The only thing - it's not very graceful or short.

This code is taken from the **MTASA** project.

So, there exists *RtlFillMemory()*. This can be a real function or a macro. It doesn't matter. It is similar to the *memset()* function, but the **2nd and 3rd argument switched their places**. Here's how this macro can be declared:

```
#define RtlFillMemory(Destination,Length,Fill) \
  memset((Destination),(Fill),(Length))
```

There is also *FillMemory()*, which is nothing more than *RtlFillMemory()*:

```
#define FillMemory RtlFillMemory
```

Yes, everything is long and complicated. But at least it is an example of real erroneous code.

And here's the code that uses the *FillMemory* macro.

```
LPCTSTR __stdcall GetFaultReason ( EXCEPTION_POINTERS * pExPtrs )
{
  ....
  PIMAGEHLP_SYMBOL pSym = (PIMAGEHLP_SYMBOL)&g_stSymbol ;
  FillMemory ( pSym , NULL , SYM_BUFF_SIZE ) ;
  ....
}
```

This code fragment has even more bugs. We can clearly see that at least the 2 and 3 arguments are confused here. That's why the analyzer issues 2 warnings V575:

- V575 The 'memset' function processes value '512'. Inspect the second argument. crashhandler.cpp 499
- V575 The 'memset' function processes '0' elements. Inspect the third argument. crashhandler.cpp 499

The code compiled because *NULL* is 0. As a result, 0 array elements get filled. But in fact the error is not only about this. *NULL* is in general not appropriate here. The memset() function works with bytes, so there's no point in trying to make it fill the memory with *NULL* values. This is absurd. Correct code should look like this:

```
FillMemory(pSym, SYM_BUFF_SIZE, 0);
```

Or like this:

```
ZeroMemory(pSym, SYM_BUFF_SIZE);
```

But it's not the main point, which is that this meaningless code compiles successfully. However, if the programmer had gotten into the habit of using *nullptr* instead of *NULL* and written this instead:

```
FillMemory(pSym, nullptr, SYM_BUFF_SIZE);
```

the complier would have emitted a error message, and the programmer would realize that they did something wrong, and would pay more attention to the way they code.

Note. I understand that in this case *NULL* is not to blame. However, it is because of *NULL* that the incorrect code compiles without any warnings.

Recommendation

Start using *nullptr*. Right now. And make necessary changes in the coding standard of your company.

Using *nullptr* will help to avoid stupid errors, and thus will slightly speed up the development process.

39. Why incorrect code works

This bug was found in **Miranda NG's** project. The code contains an error that PVS-Studio analyzer diagnoses in the following way: V502 Perhaps the '?:' operator works in a different way than was expected. The '?:' operator has a lower priority than the '|' operator..

```
#define MF_BYCOMMAND 0x00000000L

void CMenuBar::updateState(const HMENU hMenu) const

{

  ....

  ::CheckMenuItem(hMenu, ID_VIEW_SHOWAVATAR,

    MF_BYCOMMAND | dat->bShowAvatar ? MF_CHECKED : MF_UNCHECKED);

  ....

}
```

Explanation

We have seen a lot of cases that lead to incorrect working of the program, this time I would like to raise a different thought-provoking topic for discussion. Sometimes we see that totally incorrect code happens, against all odds, to work just fine! Now, for experienced programmers this really comes as no surprise (another story), but for those that have recently started learning C/C++, well, it might be a little baffling. So today, we'll have a look at just such an example.

In the code shown above, we need to call *CheckMenuItem()* with certain flags set; and, on first glance we see that if *bShowAvatar* is true, then we need to bitwise *OR MF_BYCOMMAND* with *MF_CHECKED* - and conversely, with MF_UNCHECKED if it's false. Simple!

In the code above the programmers have chosen the very natural ternary operator to express this (the operator is a convenient short version of if-then-else):

MF_BYCOMMAND | dat->bShowAvatar ? MF_CHECKED : MF_UNCHECKED

The thing is that the priority of |operator is higher than of ?: operator. (see Operation priorities in C/C++). As a result, there are two errors at once.

The first error is that the condition has changed. It is no longer - as one might read it - "dat->bShowAvatar", but "MF_BYCOMMAND | dat->bShowAvatar".

The second error - only one flag gets chosen - either MF_CHECKED or MF_UNCHECKED. The flag MF_BYCOMMAND is lost.

But despite these errors the code works correctly! Reason - sheer stroke of luck. The programmer was just lucky that the MF_BYCOMMAND flag is equal to 0x00000000L. As the MF_BYCOMMAND flag is equal to 0, then it doesn't affect the code in any way. Probably some experienced programmers have already gotten the idea, but I'll still give some comments in case there are beginners here.

First let's have a look at a correct expression with additional parenthesis:

MF_BYCOMMAND | (dat->bShowAvatar ? MF_CHECKED : MF_UNCHECKED)

Replace macros with numeric values:

0x00000000L | (dat->bShowAvatar ? 0x00000008L : 0x00000000L)

If one of the operator operands | is 0, then we can simplify the expression:

dat->bShowAvatar ? 0x00000008L : 0x00000000L

Now let's have a closer look at an incorrect code variant:

MF_BYCOMMAND | dat->bShowAvatar ? MF_CHECKED : MF_UNCHECKED

Replace macros with numeric values:

0x00000000L | dat->bShowAvatar ? 0x00000008L : 0x00000000L

In the subexpression "0x00000000L | dat->bShowAvatar" one of the operator operands | is 0. Let's simplify the expression:

dat->bShowAvatar ? 0x00000008L : 0x00000000L

As a result we have the same expression, this is why the erroneous code works correctly; another programming miracle has occurred.

Correct code

There are various ways to correct the code. One of them is to add parentheses, another - to add an intermediate variable. A good old *if* operator could also be of help here:

```
if (dat->bShowAvatar)
  ::CheckMenuItem(hMenu, ID_VIEW_SHOWAVATAR,
                  MF_BYCOMMAND | MF_CHECKED);
else
  ::CheckMenuItem(hMenu, ID_VIEW_SHOWAVATAR,
                  MF_BYCOMMAND | MF_UNCHECKED);
```

I really don't insist on using this exact way to correct the code. It might be easier to read it, but it's slightly lengthy, so it's more a matter of preferences.

Recommendation

My recommendation is simple - try to avoid complex expressions, especially with ternary operators. Also don't forget about parentheses.

As it was stated before in chapter N4, the ?: is very dangerous. Sometimes it just slips your mind that it has a very low priority and it's easy to write an incorrect expression. People tend to use it when they want to clog up a string, so try not to do that.

40. Start using static code analysis

It is strange to read such big pieces of text, written by a developer of a static code analyzer, and not to hear recommendations about the usage of it. So here it is.

Fragment taken from the **Haiku** project (inheritor of BeOS). The code contains an error that PVS-Studio analyzer diagnoses in the following way: V501 There are identical sub-expressions to the left and to the right of the '<' operator: lJack->m_jackType < lJack->m_jackType

```
int compareTypeAndID(....)
{
  ....
  if (lJack && rJack)
  {
    if (lJack->m_jackType < lJack->m_jackType)
    {
      return -1;
    }
    ....
}
```

Explanation

It's just a usual typo. Instead of rJack it was accidentally written lJack in the right part of the expression.

This typo is a simple one indeed, but the situation is quite complicated. The thing is that the programming style, or other methods, are of no help here. People just make mistakes while typing and there is nothing you can do about it.

It's important to emphasize that it's not a problem of some particular people or projects. No doubt, all people can be mistaken, and even professionals involved in serious projects can be. Here is the proof of my words. You can see the simplest misprints like A == A, in such projects as: Notepad++, WinMerge, Chromium, Qt, Clang, OpenCV, TortoiseSVN, LibreOffice, CoreCLR, Unreal Engine 4 and so on.

So the problem is really there and it's not about students' lab works. When somebody tells me that experienced programmers don't make such mistakes, I usually send them this link.

Correct code

```
if (lJack->m_jackType < rJack->m_jackType)
```

Recommendation

First of all, let's speak about some useless tips.

- Be careful while programming, and don't let errors sneak into your code (Nice words, but nothing more)
- Use a good coding style (There isn't s a programming style which can help to avoid errors in the variable name)

What can really be effective?

- Code review
- Unit tests (TDD)
- Static code analysis

I should say right away, that every strategy has its strong and weak sides. That's why the best way to get the most efficient and reliable, code is to use all of them together.

Code reviews can help us to find a great deal of different errors, and on top of this, they help us to improve readability of the code. Unfortunately shared reading of the text is quite expensive, tiresome and doesn't give a full validity guarantee. It's quite hard to remain alert, and find a typo looking at this kind of code:

```
qreal l = (orig->x1 - orig->x2)*(orig->x1 - orig->x2) +
          (orig->y1 - orig->y2)*(orig->y1 - orig->y1) *
          (orig->x3 - orig->x4)*(orig->x3 - orig->x4) +
          (orig->y3 - orig->y4)*(orig->y3 - orig->y4);
```

Theoretically, **unit tests** can save us. But it's only in theory. In practice, it's unreal to check all the possible execution paths; besides that, a test itself can have some errors too :)

Static code analyzers are mere programs, and not artificial intelligence. An analyzer can skip some errors and, on the contrary, display an error message for code which in actuality, is correct. But despite all these faults, it is a really useful tool. It can detect a whole lot of errors at an early stage.

A static code analyzer can be used as a cheaper version of Code Review. The program examines the code instead of a programmer doing it, and suggests checking certain code fragments more thoroughly.

Of course I would recommend using PVS-Studio code analyzer, which we are developing. But it's not the only one in the world; there are plenty of other free and paid tools to use. For example you can start with having a look at a free open Cppcheck analyzer. A good number of tools is given on Wikipedia: List of tools for static code analysis.

Attention:

- A static analyzer can hurt your brain if not used correctly. One of the typical mistakes is to "get the maximum from the check mode options, and drown in the stream of warnings messages". That's one of many recommendations I could give, so to get a bigger list, could be useful to go to A, B.
- A static analyzer should be used on a regular basis, not just from time to time, or when everything gets really bad. Some explanations: C, D.

Really, try using static code analyzers, you'll like them. It's a very nice sanitary tool.

Finally I would recommend reading an article by John Carmack: Static Code Analysis.

41. Avoid adding a new library to the project

Suppose you need to implement an X functionality in your project. Theorists of software development will say that you have to take the already existing library Y, and use it to implement the things you need. In fact, it is a classic approach in software development - reusing your own or others' previously created libraries (third-party libraries). And most programmers use this way.

However, those theorists in various articles and books, forget to mention what hell it will become to support several dozen third-party libraries in about 10 years.

I strongly recommend avoiding adding a new library to a project. Please don't get me wrong. I am not saying that you shouldn't use libraries at all, and write everything yourself. This would be insufficient, of course. But sometimes a new library is added to the project at the whim of some developer, intending to add a little cool small "feature" to the project. It's not hard to add a new library to the project, but then the whole team will have to carry the load of its support for many years.

Tracking the evolution of several large projects, I have seen quite a lot of problems caused by a large number of third-party libraries. I will probably enumerate only some of the issues, but this list should already provoke some thoughts:

1. Adding new libraries promptly increases the project size. In our era of fast Internet and large SSD drives, this is not a big problem, of course. But, it's rather unpleasant when the download time from the version control system turns into 10 minutes instead of 1.
2. Even if you use just 1% of the library capabilities, it is usually included in the project as a whole. As a result, if the libraries are used in the form of compiled modules (for example, DLL), the distribution size grows very fast. If you use the library as source code, then the compile time significantly increases.
3. Infrastructure connected with the compilation of the project becomes more complicated. Some libraries require additional components. A simple example: we need Python for building. As a result, in some time you'll need to have a lot of additional programs to build a project. So the probability that something will fail increases. It's hard to explain, you need to experience it. In big projects something fails all the time, and you have to put a lot of effort into making everything work and compile.
4. If you care about vulnerabilities, you must regularly update third-party libraries. It would be of interest to violators, to study the code libraries to search for vulnerabilities. Firstly, many libraries are open-source, and secondly, having found a weak point in one of the libraries, you can get a master exploit to many applications where the library is used.
5. One the libraries may suddenly change the license type. Firstly, you have to keep that in mind, and track the changes. Secondly, it's unclear what to do if that happens. For example, once, a very widely used library softfloat moved to BSD from a personal agreement.
6. You will have troubles upgrading to a new version of the compiler. There will definitely be a few libraries that won't be ready to adapt for a new compiler, you'll have to wait, or make your own corrections in the library.
7. You will have problems when moving to a different compiler. For example, you are using Visual C++, and want to use Intel C++. There will surely be a couple of libraries where something is wrong.
8. You will have problems moving to a different platform. Not necessarily even a totally different platform. Let's say, you'll decide to port a Win32 application to Win64. You will have the same problems. Most likely, several libraries won't be ready for this, and you'll wonder what to do with them. It is especially unpleasant when the library is lying dormant somewhere, and is no longer developing.
9. Sooner or later, if you use lots of C libraries, where the types aren't stored in *namespace*, you'll start having name clashes. This causes compilation errors, or hidden errors. For example, a wrong *enum* constant can be used instead of the one you've intended to use.
10. If your project uses a lot of libraries, adding another one won't seem harmful. We can draw an analogy with the broken windows theory. But consequently, the growth of the project turns into uncontrolled chaos.
11. And there could be a lot of other downsides in adding new libraries, which I'm probably not aware of. But in any case, additional libraries increase the complexity of project support. Some issues can occur in a fragment where they were least expected to.

Again, I should emphasize; I don't say that we should stop using third-party libraries at all. If we have to work with images in PNG format in the program, we'll take the LibPNG library, and not reinvent the wheel.

But even working with PNG we need to stop and think. Do we really need a library? What do we want to do with the images? If the task is just to save an image in *.png file, you can get by with system

functions. For example, if you have a Windows application, you could use WIC. And if you're already using an MFC library, there is no need to make the code more sophisticated, because there's a CImage class (see the discussion on StackOverflow). Minus one library - great!

Let me give you an example from my own practice. In the process of developing the PVS-Studio analyzer, we needed to use simple regular expressions in a couple of diagnostics. In general, I am convinced that static analysis isn't the right place for regular expressions. This is an extremely inefficient approach. I even wrote an article regarding this topic. But sometimes you just need to find something in a string with the help of a regular expression.

It was possible to add existing libraries, but it was clear that all of them would be redundant. At the same time we still needed regular expressions, and we had to come up with something.

Absolutely coincidentally, exactly at that moment I was reading a book "Beautiful Code" (ISBN 9780596510046). This book is about simple and elegant solutions. And there I came across an extremely simple implementation of regular expressions. Just a few dozen strings. And that's it!

I decided to use that implementation in PVS-Studio. And you know what? The abilities of this implementation are still enough for us; complex regular expressions are just not necessary for us.

Conclusion: Instead of adding a new library, we spent half an hour writing a needed functionality. We suppressed the desire to use one more library. And it turned out to be a great decision; the time showed that we really didn't need that library. And I am not talking about several months, we have happily used it for more than five years.

This case really convinced me that the simpler solution, the better. By avoiding adding new libraries (if possible), you make your project simpler.

Readers may be interested to know what the code for searching regular expressions was. We'll type it here from the book. See how graceful it is. This code was slightly changed when integrating to PVS-Studio, but its main idea remains unchanged. So, the code from the book:

```
// regular expression format
// c Matches any "c" letter
//.(dot) Matches any (singular) symbol
//^ Matches the beginning of the input string
//$ Matches the end of the input string
# Match the appearance of the preceding character zero or
// several times

int matchhere(char *regexp, char *text);
int matchstar(int c, char *regexp, char *text);

// match: search for regular expression anywhere in text
int match(char *regexp, char *text)
{
  if (regexp[0] == '^')
```

```
    return matchhere(regexp+1, text);
  do { /* must look even if string is empty */
   if (matchhere(regexp, text))
     return 1;
  } while (*text++ != '\0');
  return 0;
}

// matchhere: search for regexp at beginning of text
int matchhere(char *regexp, char *text)
{
   if (regexp[0] == '\0')
     return 1;
   if (regexp[1] == '*')
     return matchstar(regexp[0], regexp+2, text);

   if (regexp[0] == '$' && regexp[1] == '\0')
     return *text == '\0';
   if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
     return matchhere(regexp+1, text+1);
   return 0;
}

// matchstar: search for c*regexp at beginning of text
int matchstar(int c, char *regexp, char *text)
{
  do {   /* * a * matches zero or more instances */
           more instances */
    if (matchhere(regexp, text))
      return 1;
  } while (*text != '\0' && (*text++ == c || c == '.'));
  return 0;
}
```

Yes, this version is extremely simple, but for several years there was need to use more complex solutions. It really has got limited functionality, but there was no need to add anything more complicated, and I don't think there will be. This is a good example of where a simple solution turned out to be better than a complex one.

Recommendation

Don't hurry to add new libraries to the project; add one only when there is no other way to manage without a library.

Here are the possible workarounds:

1) Have a look if the API of your system, or one of the already used libraries has a required functionality. It's a good idea to investigate this question.
2) If you plan to use a small piece of functionality from the library, then it makes sense to implement it yourself. The argument to add a library "just in case" is no good. Almost certainly, this library won't be used much in the future. Programmers sometimes want to have universality that is actually not needed.
3) If there are several libraries to resolve your task, choose the simplest one, which meets your needs. As I have stated before, get rid of the idea "it's a cool library - let's take it just in case"
4) Before adding a new library, sit back and think. Maybe even take a break, get some coffee, discuss it with your colleagues. Perhaps you'll realsie that you can solve the problem in a completely different way, without using third-party libraries.

P.S. The things I speak about here may not be completely acceptable to everyone. For example, the fact that I'm recommending the use of WinAPI, instead of a universal portable library. There may arise objections based on the idea that going this way "binds" this project to one operating system. And then it will be very difficult to make a program portable. But I do not agree with this. Quite often the idea "and then we'll port it to a different operating system" exists only in the programmer's mind. Such a task may even be unnecessary for managers. Another option - the project will kick the bucket due to the complexity and universality of it before gaining popularity and having the necessity to port. Also don't forget about point (8) in the list of problems, given above.

42. Don't use function names with "empty"

The fragment is taken from **WinMerge** project. The code contains an error that PVS-Studio analyzer diagnoses in the following way: V530 The return value of function 'empty' is required to be utilized.

```
void CDirView::GetItemFileNames(
  int sel, String& strLeft, String& strRight) const
{
  UINT_PTR diffpos = GetItemKey(sel);
  if (diffpos == (UINT_PTR)SPECIAL_ITEM_POS)
  {
    strLeft.empty();
    strRight.empty();
  }
  ....
}
```

Explanation

A programmer wanted to clean the *strLeft* and *strRight* strings. They have *String* type, which is nothing else than *std::wstring*.

For this purpose he called the *empty()* function. And this is not correct. The *empty()* function doesn't change the object, but returns the information if the string is empty or not.

Correct code

To correct this error you should replace the *empty()* function with *clear()* or *erase ()*. WinMerge developers preferred *erase()* and now the code looks like this:

```
if (diffpos == (UINT_PTR)SPECIAL_ITEM_POS)
{
  strLeft.erase();
  strRight.erase();
}
```

Recommendation

In this case the name "empty()" is really inappropriate. The thing is that in different libraries, this function can mean two different actions.

In some libraries the *emply()* function clears the object. In other ones, it returns the information if the object is empty or not.

I would say that the word "empty" is lame in general, because everybody understands it differently. Some think it's an "action", others that it's "information inquiry". That's the reason for the mess we can see.

There is just one way out. Do not use "empty" in the class names.

- Name the function for cleaning as "erase" or "clear". I would rather use "erase", because "clear" can be quite ambiguous.
- Choose another name for the function which gets information, "isEmpty" for instance.

If you for some reason think that it's not a big deal, then have a look here. It's quite a widespread error pattern. Of course it's slightly late to change such classes as *std::string*, but at least let's try not to spread the evil any longer.

Conclusion

I hope you enjoyed this collection of tips. Of course, it is impossible to write about all the ways to write a program incorrectly, and there is probably no point in doing this. My aim was to warn a programmer, and to develop a sense of danger. Perhaps, next time when a programmer encounters something odd, he will remember my tips and won't haste. Sometimes several minutes of studying the documentation or writing simple/clear code can help to avoid a hidden error that would make the life of your colleagues and users miserable for several years.

I also invite everybody to follow me on Twitter @Code_Analysis

Bugless coding!

Sincerely, Andrey Karpov.