# Clang vs GCC vs MSVC: Diagnostics

**Now with more tests and MSVC diagnostics!**

GCC and Clang have always been trying to prove who has the better error diagnostics.

Clang first dissed GCC's error in their "Expressive Diagnostics" article. GCC improved their diagnostics and released their comeback article, titled "Clang Diagnostics Comparison".

Let's see who is really better by testing common errors in Clang 6.0.0, GCC 7.3.0, and, via the Compiler Explorer, MSVC 2017 19.10.25107. Note that GCC 8 appears to have improved some messages, but it isn't a stable release yet.

I am counting out the static analyzers in MSVC and Clang, as it wouldn't be fair to compare it to GCC's lack of one. Only -Wall or /W3 will be used, unless no errors are found, then I will try -Weverything, -Wextra -Wpedantic, or /Wall.

## Round 1: Missing Semicolons

Forgetting semicolons. *You do it all the time, and if you don't, shut up, you're lying.*

**semicolon.c**

```
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n") // no semicolon
    return 0 // no semicolon
}
```

Here are your average missing semicolons, after the *printf* statement and the return statement. For such a common error, the compiler should be smart enough to pick that up, right?

```
~ $ gcc-7 -Wall semicolon.c
semicolon.c: In function 'main':
semicolon.c:5:5: error: expected ';' before 'return'
     return 0 // no semicolon
     ^~~~~~
     ;

C:\> cl /W3 semicolon.c
semicolon.c(5): error C2143: syntax error: missing ';' before 'return'
semicolon.c(6): error C2143: syntax error: missing ';' before '}'
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

...right?

~ $ clang-6.0 -Wall semicolon.c
semicolon.c:4:30: error: expected ';' after expression
    printf("Hello, world!\n") // no semicolon
                             ^
                             ;

semicolon.c:5:13: error: expected ';' after return statement
    return 0 // no semicolon
            ^
            ;
2 errors generated.
```

Clang clearly won here, with MSVC in second. GCC didn't recognize the second error, and the "expected ';' before 'return'" errors from MSVC and GCC are like saying that to climb a tree, I must put the tree below me. It is *technically* true, but it is stupid.

**Score:** Clang: 2, GCC: 0, MSVC: 1

## Round 2: The Missing Brace

Forgetting a brace at the end of a function is another common error, although not as common as the former.

**missingbrace.c**

```
int main(void) {
    return 0;
// no closing brace
```

Hopefully, GCC or MSVC can make up for this one.

```
~ $ gcc-7 -Wall missingbrace.c
missingbrace.c: In function 'main':
missingbrace.c:2:5: error: expected declaration or statement at end of input
     return 0;
     ^~~~~~

God damn it, GCC.

C:\> cl /W3 missingbrace.c
missingbrace.c(1): fatal error C1075: the left brace '{' was unmatched at the end of the file
Internal Compiler Error in Z:\vpt\compiler-explorer\windows\19.10.25017\lib\native\bin\cl.exe.  You will be prompted to
send an error report to Microsoft later.
INTERNAL COMPILER ERROR in 'Z:\vpt\compiler-explorer\windows\19.10.25017\lib\native\bin\amd64_x86\cl.exe'
    Please choose the Technical Support command on the Visual C++
    Help menu, or open the Technical Support help file for more information
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

I might be wrong, but it looks like MSVC crashed. Nice job, Microsoft. Crash aside, MSVC did better.

~ $ clang-6.0 -Wall missingbrace.c
missingbrace.c:2:14: error: expected '}'
    return 0;
             ^
missingbrace.c:1:16: note: to match this '{'
int main(void) {
               ^
1 error generated.
```

Yet again, two points to Clang.

**Score:** Clang: 4, GCC: 0, MSVC: 2

## Round 3: Out of bounds

Another common error:

**outofbounds.c**

```
#include <stdio.h>

static const int array[4] = { 1, 2, 3, 4 };

int main(void) {
  for (int i = 0; i <= 4 /* should be < */; i++) {
      printf("%d ", array[i]);
  }
  return 0;
}
```

Interestingly, even with -Warray-bounds or /Wall, this isn't picked up in either Clang or MSVC.

However, with -O2, GCC actually says something right for a change!

```
~ $ gcc-7 -Wall -O2 outofbounds.c
outofbounds.c: In function 'main':
outofbounds.c:7:9: warning: iteration 4 invokes undefined behavior [-Waggressive-loop-optimizations]
       printf("%d ", array[i]);
       ~~~~~~~~~~~~~~~~~~~~~~~
outofbounds.c:6:5: note: within this loop
   for (int i = 0; i <= 4 /* should be < */; i++) {
   ^~~

GCC only gets one point here, though, because it doesn't always show this error.
```

**Score:** Clang: 4, GCC: 1, MSVC: 2

## Round 4: Ifs without Braces

Ifs without braces. While they can be convenient, they often cause more harm than good, such as the infamous goto fail bug.

**if-else-bug.c**

```
#include <stdio.h>

int main(int argc, char*argv) {
  if (argc > 1) // needs braces
      argc--;
      argv++;
  else
      printf("Usage: %s <arguments>\n", *argv); // (this would theoretically be UB because of the argv++)
  return 0;
}
```

Naturally, being Apple's compiler, Clang should pick up on this error.

```
~ $ clang-6.0 -Wall if-else-bug.c
if-else-bug.c:8:5: error: expected expression
    else
    ^
1 error generated.

...That is a very useless error. No wonder Apple didn't pick up on that bug.

C:\> cl /W3 if-else-bug.c
if-else-bug.c(7): error C2181: illegal else without matching if
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

MSVC at least makes some sense, unlike the nonsense Clang spits out.

~ $ gcc-7 -Wall if-else-bug.c
if-else-bug.c: In function 'main':
if-else-bug.c:5:5: warning: this 'if' clause does not guard... [-Wmisleading-indentation]
     if (argc > 1) // needs braces
     ^~
if-else-bug.c:7:9: note: ...this statement, but the latter is misleadingly indented as if it were guarded by the 'if'
       argv++;
       ^~~~
if-else-bug.c:8:5: error: 'else' without a previous 'if'
     else
     ^~~~

Wow. For once, GCC nailed it!
```

**Score:** Clang: 4, GCC: 3, MSVC: 2

## Round 5: Java-style string concatenation

Java, JavaScript, C++ (to a point), and a few other languages let you join strings and other things with a '+'. C doesn't do what you might expect.

**string-concat.c**

```
#include <stdio.h>

int main(void) {
    int value = 4;
    const char *string = "value = " + value; // This isn't Java!
    printf("%s\n", string);
    return 0;
}
```

```
~ $ gcc-7 -Wall -pedantic string-concat.c
~ $ clang-6.0 -Wall string-concat.c
string-concat.c:5:37: warning: adding 'int' to a string does not append to the string [-Wstring-plus-int]
    const char *string = "value = " + value; // This isn't Java!
                         ~~~~~~~~~~ ^
string-concat.c:5:37: note: use array indexing to silence this warning
    const char *string = "value = " + value; // This isn't Java!
                                    ^
                         &          [    ]
1 warning generated.

GCC and MSVC didn't pick it up at all, but Clang gave a very helpful error.
```

**Score:** Clang: 6, GCC: 3, MSVC: 2

## Round 6: Forgetting to return a value

Sometimes, you forget that a function needs to return a value, or you forget to put a return statement after that switch statement, or whatever.

**no-return.c**

```
#include <stdlib.h>

int doesNotReturnAValue(void) {
    // no return value
}

int mightNotReturnAValue(void) {
    if (rand() % 2 == 0) {
        return 2;
    }
    // if rand() is odd, there is no return value
}
```

```
~ $ gcc-7 -Wall no-return.c
no-return.c: In function 'doesNotReturnAValue':
no-return.c:5:1: warning: control reaches end of non-void function [-Wreturn-type]
 }
 ^
no-return.c: In function 'mightNotReturnAValue':
no-return.c:12:1: warning: control reaches end of non-void function [-Wreturn-type]
 }
 ^
~ $ clang-6.0 -Wall no-return.c
no-return.c:5:1: warning: control reaches end of non-void function [-Wreturn-type]
}
^
no-return.c:12:1: warning: control may reach end of non-void function [-Wreturn-type]
}
^
2 warnings generated.

Whaaaaaaat... Zero points for zero sense!

C:\> cl /W3 no-return.c
no-return.c(5) : warning C4716: 'doesNotReturnAValue': must return a value
no-return.c(12) : warning C4715: 'mightNotReturnAValue': not all control paths return a value
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

That's more like it, MSVC!
```

**Score:** Clang: 6, GCC: 3, MSVC: 4

## Round 7: Forgetting your namespace

Time for some C++!

Forgetting to either add a "using namespace" or put the namespace before your calls is an error that I always make.

**no-namespace.cpp**

```
#include <iostream>

int main() {
    cout << "Hello, world!\n"; // should be std::cout
    return 0;
}
```

Let's see what the compilers have to say, shall we?

```
C:\> cl /W3 no-namespace.cpp
no-namespace.cpp(4): error C2065: 'cout': undeclared identifier
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Thanks for nothing, Microsoft.

~ $ g++-7 -Wall no-namespace.cpp
no-namespace.cpp: In function 'int main()':
no-namespace.cpp:4:5: error: 'cout' was not declared in this scope
     cout << "Hello, world!\n"; // should be std::cout
     ^~~~
no-namespace.cpp:4:5: note: suggested alternative:
In file included from no-namespace.cpp:1:0:
/usr/include/c++/7.3.0/iostream:61:18: note:   'std::cout'
   extern ostream cout;  // Linked to standard output
                  ^~~~

I guess that is better. GCC understands that we meant std::cout, although the message is kind of confusing. Let's see Clang's version.

~ $ clang++-6.0 -Wall no-namespace.cpp
no-namespace.cpp:4:5: error: use of undeclared identifier 'cout'; did you mean 'std::cout'?
    cout << "Hello, world!\n"; // should be std::cout
    ^~~~
    std::cout
/usr/include/c++/v1/iostream:54:33: note: 'std::cout' declared here
extern _LIBCPP_FUNC_VIS ostream cout;
                        ^
1 error generated.

There we go. Same information as GCC, but, unlike GCC, it goes directly to the point and asks if we meant "std::cout", then shows the implementation. Two points to Clang, one point to GCC.
```

**Score:** Clang: 8, GCC: 4, MSVC: 4

## Round 8: dynamic_casting a class itself

The C++ dynamic_cast is supposed to be used on a pointer to a class, not on the class itself. It is weird.

**casting-a-class.cpp**

```
class Base {};
class Derived : public Base {};

int main() {
    Base base;
    Derived derived = dynamic_cast<Derived>(base); // should be used on a pointer
    return 0;
}
```

```
~ $ clang++-6.0 -Wall casting-a-class.cpp
casting-a-class.cpp:6:23: error: 'Derived' is not a reference or pointer
    Derived derived = dynamic_cast<Derived>(base); // should be a pointer
                      ^~~~~~~~~~~~~~~~~~~~~

1 error generated.

Huh? Well, no duh, Clang. Why are you erroring, though?

~ $ g++-7 -Wall casting-a-class.cpp
casting-a-class.cpp: In function 'int main()':
casting-a-class.cpp:6:49: error: cannot dynamic_cast 'base' (of type 'class Base') to type 'class Derived' (target is not pointer or reference)
     Derived derived = dynamic_cast<Derived>(base); // should be a pointer
                                             ^~~~

GCC makes it a bit clearer, although I don't know what it's pointing to.

C:\> cl /W3 casting-a-class.cpp
casting-a-class.cpp(6): error C2680: 'Derived': invalid target type for dynamic_cast
casting-a-class.cpp(6): note: target type must be a pointer or reference to a defined class
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

MSVC is the winner here. It explained the issue quite well.
```

**Score:** Clang: 8, GCC: 5, MSVC: 6

# The winner is Clang!

Note that I am not saying that one compiler sucks. All three compilers have their strengths and weaknesses. But Clang has proven itself to be stronger in the diagnostics department.