Why the compiler broke your program

Peter Brett, LiveCode

Six impossible things before breakfast

```
/**
 * Returns the first EntList not of type join, starting from this.
 */
EntList * EntList::firstNot( JoinType j ) {
    EntList * sibling = this;

    while( sibling != NULL && sibling->join == j ) {
        sibling = sibling->next;
    }
    return sibling;  // (may = NULL)
}
```

sibling can't be null…

…so why do I get a null pointer dereference here?

```
#define NULL (__null)
typedef int JoinType;
class EntList {
    EntList* next;
    JoinType join;
public:
    EntList* firstNot(JoinType j);
};

EntList *EntList::firstNot(JoinType j)
{
    EntList * sibling = this;
    while (sibling != NULL) {
        if (sibling->join != j)
            break;
        sibling = sibling->next;
    }
    return sibling;
}
```

First

Loop

```
EntList::firstNot(int):
        test    rdi, rdi
        je      .L2
        mov     edx, DWORD PTR [rdi+8]
        mov     rax, rdi
        cmp     edx, esi
        je      .L3
        jmp     .L2
.L5:
        cmp     DWORD PTR [rax+8], edx
        jne     .L4
.L3:
        mov     rax, QWORD PTR [rax]
        test    rax, rax
        jne     .L5
        rep
        ret
.L2:
        mov     rax, rdi
.L4:
        rep
        ret
```

GCC 4.4.7 (pre C++11): -O3

```
#define NULL (nullptr)
enum class JoinType : int;
class EntList {
    EntList* next;
    JoinType join;
public:
    EntList* firstNot(JoinType j);
};

EntList * EntList::firstNot(JoinType j)
{
    EntList * sibling = this;
    while (sibling != NULL) {
        if (sibling->join != j)
            break;
        sibling = sibling->next;
    }
    return sibling;
}
```

```
EntList::firstNot(JoinType):
        mov     rax, rdi
.L3:
        cmp     DWORD PTR [rax+8], esi
        jne     .L1
        mov     rax, QWORD PTR [rax]
        test    rax, rax
        jne     .L3
.L1:
        rep ret
```

GCC 6.3: -O3

What does the C++ standard say?

"If a non-static member function of a class `X` is called for an object that is not of type `X`, or of a type derived from `X`, the behavior is undefined."

— C++17 draft standard §12.2.2

"In the body of a non-static member function, the keyword `this` is a prvalue expression whose value is the address of the object for which the function is called."

— C++17 draft standard §12.2.2.1

Undefined behaviour is magic!

1. If `EntList::firstNot()` is called for an object that is not of type `EntList`, the behaviour is undefined.
2. `nullptr` is not an object of type `EntList`.
3. Therefore if `EntList::firstNot()` is called for `nullptr`, the behaviour is undefined.
4. Therefore it can be assumed that `this` is never `nullptr`.
5. Therefore the check can be optimised out.

```
#define NULL (nullptr)
enum class JoinType : int;
class EntList {
    EntList* next;
    JoinType join;
public:
    EntList* firstNot(JoinType j);
};

EntList * EntList::firstNot(JoinType j)
{
    EntList * sibling = this;
    while (sibling != NULL) {
        if (sibling->join != j)
            break;
        sibling = sibling->next;
    }
    return sibling;
}
```

```
EntList::firstNot(JoinType):
        test    rdi, rdi
        je      .L6
        cmp     esi, DWORD PTR [rdi+8]
        mov     rax, rdi
        je      .L4
        jmp     .L1
.L5:
        cmp     DWORD PTR [rax+8], esi
        jne     .L1
.L4:
        mov     rax, QWORD PTR [rax]
        test    rax, rax
        jne     .L5
        rep ret
.L1:
        rep ret
.L6:
        xor     eax, eax
        ret
```

GCC 6.3: -O3 **-fno-delete-null-pointer-checks**

What’s the actual problem here?

- The standard is wrong!
  - The C++ standard should define what happens when calling methods on an invalid object
- The compiler is wrong!
  - A compiler shouldn’t include new optimisations that might break previously-working code
  - …or, at least, they shouldn’t be enabled by default
- The program is wrong!
  - The program should use STL collection types & algorithms
  - The program shouldn’t expect a specific realization of undefined behaviour

Working with a legacy codebase

- Know the C++ spec & be able to recognize common problematic UB patterns
  - `this` vs. `nullptr`
  - Signed overflow
  - Out-of-bounds access
  - Uninitialised scalar variables
  - Access to dead pointers, e.g. after passing to `realloc()`
- Become friends with your disassembler and debugger
- Disable optimisations that cause problems
  - Use lower optimisation level
  - -fno-delete-null-pointer-checks, -fno-strict-overflow, -fno-strict-aliasing
- Use UndefinedBehaviorSanitizer (-fsanitize=undefined)
  - Requires excellent test coverage
  - Sometimes UB is required for fast code, e.g. array offsets

Developing new code

- Avoid implementing your own data structures & algorithms
  - Modern STL implementations are really good (libc++, libstdc++, MSVC 2017)
- Design APIs not to use raw pointers
- Be a pedantic language lawyer
  - Avoid UB if possible
  - If UB is necessary, document it carefully
- Know your compiler & platform ISA

Sanity-check the assembly generated by the compiler

Thank you!

Resources:

- My Little Optimizer: Undefined Behavior is Magic (Michael Spencer, CppCon)
- Garbage In, Garbage Out: Arguing about Undefined Behavior with Nasal Demons (Chandler Carruth, CppCon)
- C++ Draft Standard
- Compiler Explorer