CSO-1

Undefined Behavior

Daniel G. Graham Ph.D

UNDEFINED BEHAVIOR

```
  GNU nano 6.3          example0.c          Modified
#include <limits.h>
#include <stdio.h>

int main(){
        printf("%d\n", 1<<32);
        return 0;
}
```

```
dgg6b@portal09:~/CSO-Code-Examples/Undefined$ gcc exam
ple0.c
example0.c: In function ‘main’:
example0.c:5:25: warning: left shift count >= width of
 type [-Wshift-count-overflow]
    5 |         printf("%d\n", 1<<32);
      |                         ^~
dgg6b@portal09:~/CSO-Code-Examples/Undefined$
```

UNDEFINED BEHAVIOR

The comp.lang.c C FAQ defines “undefined behavior” like this:

Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.

If any step in a program’s execution has undefined behavior, then the entire execution is without meaning. This is important: it’s not that evaluating (1<<32) has an unpredictable result, but rather that the entire execution of a program that evaluates this expression is meaningless. --- Embedded in Academia

OFFICAL C DOCUMENTION

2 EXAMPLE An example of locale-specific behavior is whether the **`islower`** function returns true for characters other than the 26 lowercase Latin letters.

3.4.3

1 **undefined behavior**

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

2 NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

3 EXAMPLE An example of undefined behavior is the behavior on integer overflow.

GNU nano 6.3 example0.c

```c
#include <limits.h>
#include <stdio.h>

int main(){
        printf("%d \n", 1 << 32);
        return 0;
}
```

GNU nano 6.3 example0.s

```
        .text
        .file   "example0.c"
        .globl  main                                    # -- Be
        .p2align        4, 0x90
        .type   main,@function
main:                                                   # @main
        .cfi_startproc
# %bb.0:
        pushq   %rax
        .cfi_def_cfa_offset 16
        movl    $.L.str, %edi
        xorl    %eax, %eax
        callq   printf
        xorl    %eax, %eax
        popq    %rcx
        .cfi_def_cfa_offset 8
        retq
.Lfunc_end0:
```

What is missing? What is the value in edi
The second parameter.

GNU nano 6.3 example0.c

```c
#include <limits.h>
#include <stdio.h>

int main(){
        printf("%d \n", 1 << 31);
        return 0;
}
```

GNU nano 6.3 example0.s

```
        .text
        .file   "example0.c"
        .globl  main                                    #
        .p2align        4, 0x90
        .type   main,@function
main:                                                   # @main
        .cfi_startproc
# %bb.0:
        pushq   %rax
        .cfi_def_cfa_offset 16
        movl    $.L.str, %edi
        movl    $-2147483648, %esi                      #
        xorl    %eax, %eax
        callq   printf
        xorl    %eax, %eax
        popq    %rcx
        .cfi_def_cfa_offset 8
        retq
```

UNDEFINED BEHAVIOR

```
#include <limits.h>
#include <stdio.h>

int main (void)
{
  printf ("%d\n", (INT_MAX+1) < 0);
  return 0;
}
```

What should this program out?

MAX INT two complement: 0x7FFFFFFF

Should it be negative number, 0, or Max INT

The C Standard doesn't say so it is undefined behavior

LET SEE WHAT CLANG DOES

```
  GNU nano 6.3          example0.c
#include <limits.h>
#include <stdio.h>

int main(){
        printf("%d\n", INT_MAX +1);
        return 0;
}
```

```
dgg6b@portal09:~/CSO-Code-Examples/Undefined$ clang ex
ample0.c
example0.c:5:25: warning: overflow in expression; resu
lt is -2147483648 with type 'int' [-Winteger-overflow]
        printf("%d\n", INT_MAX +1);
                               ^
1 warning generated.
dgg6b@portal09:~/CSO-Code-Examples/Undefined$ ./a.out
-2147483648
dgg6b@portal09:~/CSO-Code-Examples/Undefined$
```

INSERT A CONSTANT

GNU nano 6.3 example0.c

```c
#include <limits.h>
#include <stdio.h>

int main(){
        printf("%d\n", INT_MAX +1);
        return 0;
}
```

GNU nano 6.3 example0.s

```
        .text
        .file   "example0.c"
        .globl  main
        .type   main,@function
main:
        .cfi_startproc
# %bb.0:
        pushq   %rax
        .cfi_def_cfa_offset 16
        movl    $.L.str, %edi
        movl    $-2147483648, %esi
        xorl    %eax, %eax
        callq   printf
        xorl    %eax, %eax
        popq    %rcx
        .cfi_def_cfa_offset 8
        retq
```

UNDEFINED BEHAVIOR

x= (x*2)/2

Should the compiler be allowed to optimize this away.

Just like before what if X wraps. Should we return the negative result

GNU nano 6.3 example1.c

```
#include <stdio.h>

int ex_function(int x){
        x = (x*2)/2;
        return x;
        }

int main(){
        int input;
        scanf("%d", &input);
        return input;
```

Compiled with -O3

GNU nano 6.3 example1.s

```
        .text
        .file   "example1.c"
        .globl  ex_function                     # --
        .p2align        4, 0x90
        .type   ex_function,@function
ex_function:                                    # @ex_functi
        .cfi_startproc
# %bb.0:
        movl    %edi, %eax
        retq
.Lfunc_end0:
        .size   ex_function, .Lfunc_end0-ex_function
        .cfi_endproc
                                                # -- End fun
        .globl  main                                    # --
        .p2align        4, 0x90
        .type   main,@function
main:                                           # @main
        .cfi_startproc
# %bb.0:
        pushq   %rax
        .cfi_def_cfa_offset 16
        leaq    4(%rsp), %rsi
        movl    $.L.str, %edi
        xorl    %eax, %eax
        callq   __isoc99_scanf
        movl    4(%rsp), %eax
        popq    %rcx
        .cfi_def_cfa_offset 8
        retq
```

UNDEFINED BEHAVIOR

x= (x*2)/2

Should the compiler be allowed to optimize this away.

But if X is signed and can wrap around. (Let's check out a video)

Is wrapping behavior defined in the c language?

--Sadly no

MORE UNDEFINED BEHAVIOR

```
  GNU nano 6.3          example1.c
#include <stdio.h>
#include <limits.h>
int ex_function(int x){
        x = (x*2)/2;
        return x;
        }

int main(){
        printf("%d \n", ex_function(INT_MAX));
        return 0;
}
```

```
dgg6b@portal09:~/CSO-Code-Examples/Undefined$ clang
example1.c
dgg6b@portal09:~/CSO-Code-Examples/Undefined$ ./a.ou
```

UNDEFINED BEHAVIOR

x= (x*2)/2

INT_MAX * 2 = negative
01111…11 == INT MAX
11111…10 == INT_MAX*2 (shift 2)
11111…111 == Result >> 1 (-1)

Should the compiler be allowed to optimize this away.

But if X is unsigned and can wrap around. (Let's check out a video)

Is wrapping behavior defined in the c language?

--Sadly no

MORE UNDEFINED BEHAVIOR

```
int a[5];
a[x] = 0;
If(x >= 5){
   printf("Do we need this?");
}
```

This printf should never happen. If it happens the program is already broken, so it doesn't mater.

So, the compiler will optimize it away.

UNREACHABNLE

```
a /= x
if(x==0){
  print("Unreachable")
}
```

Dividing by zero is an undefined behavior

So compile will optimize away the if statement

NOT ALL OPTIMIZATION ARE IMPLEMENTED

```
a <<= x
If(x >= 32)
  print("Unreachable")
```

Can't shift int by greater than 32.

Some compilers don't have this this optimization built in.

```
int my_function(int a){
     int x = 0;
  a/=x;
}
```

It one part of the function is has undefined behavior the compiler ass

So now the compiler can optimize other code with assumption that this function will not be called. And this can start to cause strange behavior

___buitin__unreachable()

assume()

WRAPPING

```
int a;
if (a + 1 >a)
a++
```

The compiler will remove the if statement so it difficult to see if something wraps.

YOUR COMPILER BROKE MY CODE

Many optimizations are possible, but they would break too much code.

UNINITIALIZED VALUES

```
int function(){
  int a; //not initialized
   if(a ==0)
    return 0;
      If( a!= 0)
       return 0;
}
```

What does this compile to?

UNINITIALIZED VALUES

```
int function(){
  int a; //not initialized
   if(a ==0)
    return 0;
      If( a!= 0)
       return 0;
}
```

What does this compile to?

ret

Yes. Just ret.

Uninitialized values are in an indeterminate state.

MEMORY IS NOT REALLY YOUR UNTIL YOU WRITE TO IT.

```
volatile char * buf = malloc(1);
if (buf == NULL)
    return
char c1 = buf[0];
char c2 = buf[0];
assert(c1 == c2)
```

Don't assume that the values that you have not initialized are static or stable.

The operating system doesn't always reserve that section of memory for you until you write to it.

The volatile keyword tells the compiler that the variable may change, at any time with-out action begin taken in code. (Think Memory mapped IO)

Draw picture of illustrating memory mapped IO

+

NOW LET TALK A BIT ABOUT PADDING

```
struct name_tag{
    int y;
    char * x;
    char z;
};
```

Take the largest element size and uses that as the default for all other members

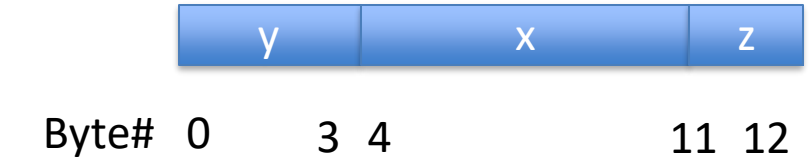
Byte# 0 3 4 11 12

This would be most compressed representation. But NOT how it get layout in the C standard.

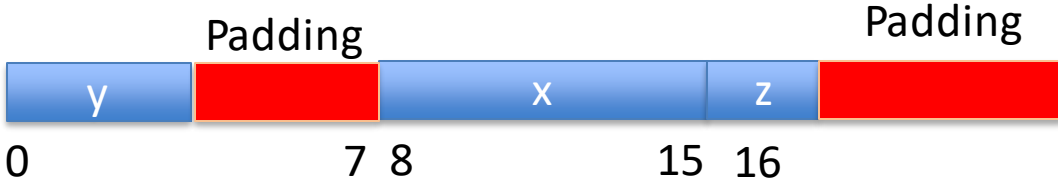
NOW LET TALK A BIT ABOUT PADDING

```
struct name_tag{
    int y;
    char * x;
    char z;
};
```

Take the largest element size and uses that as the default for all other members



Byte# 0 3 4 11 12



Instead, the compiler adds padding

Also get alignment ☺