# Secure Coding in C

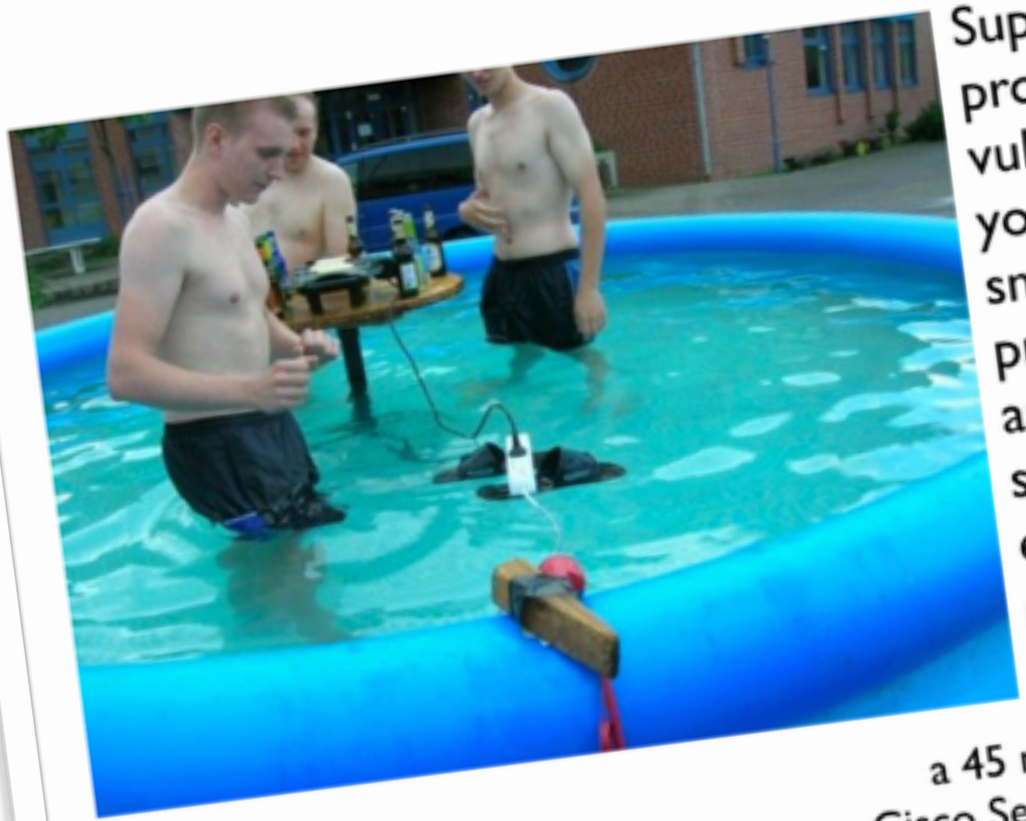
Olve Maudal

# Secure Coding ~~~~ in C

with integers

Olve Maudal

## Insecure Coding in C (and C++)



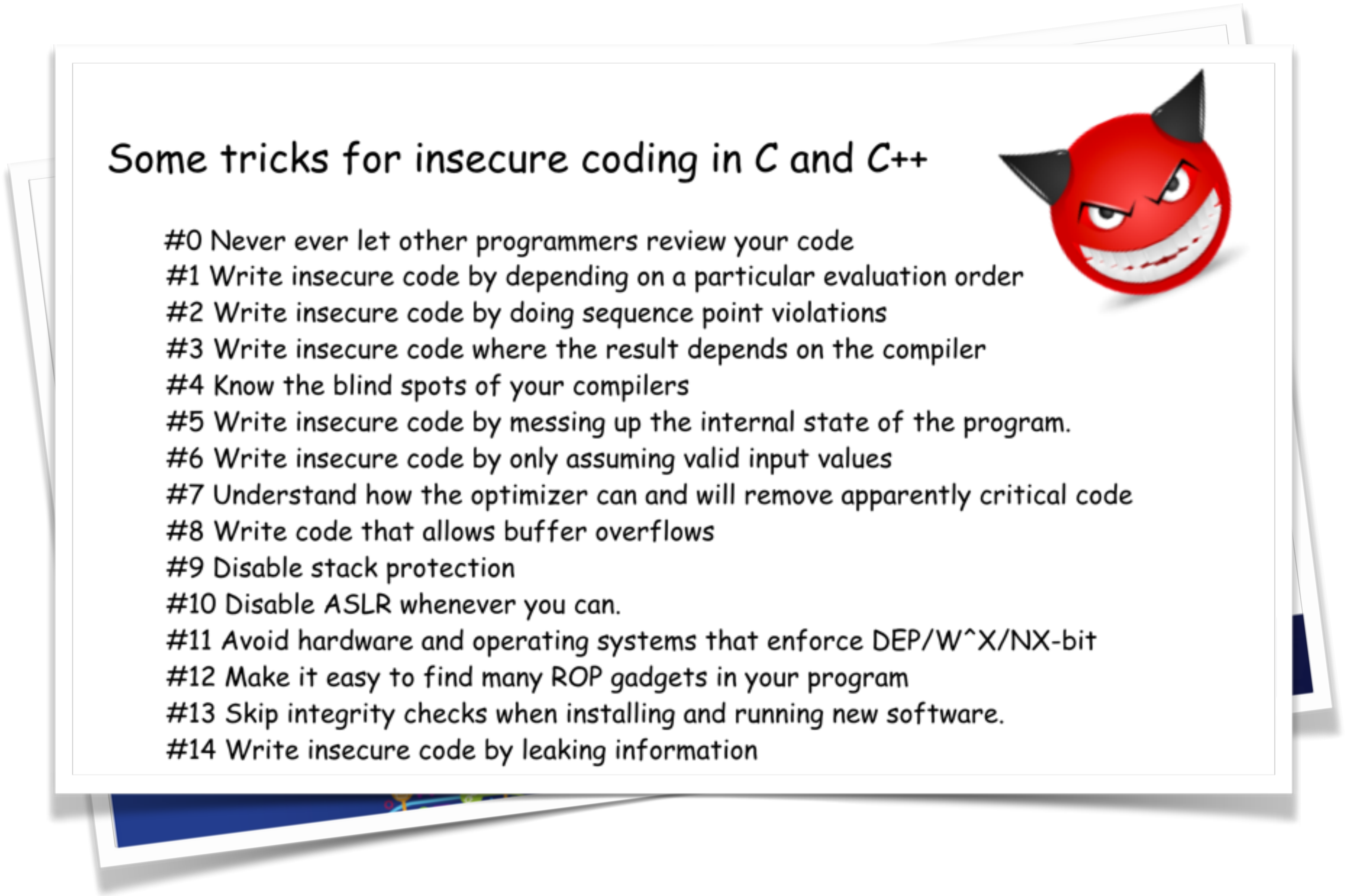
Suppose your goal is to deliberately create buggy programs in C and C++ with serious security vulnerabilities that can be "easily" exploited. Then you need to know about things like stack smashing, shellcode, arc injection, return-oriented programming. You also need to know about annoying protection mechanisms such as address space layout randomization, stack canaries, data execution prevention, and more. (This session is really about how to write secure code, I just use negative examples to illustrate my points.)

a 45 minute presentation
Cisco SecConX Bangalore, India
November 12, 2014

SecConX India **2014**

## Some tricks for insecure coding in C and C++

#0 Never ever let other programmers review your code
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by doing sequence point violations
#3 Write insecure code where the result depends on the compiler
#4 Know the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Understand how the optimizer can and will remove apparently critical code
#8 Write code that allows buffer overflows
#9 Disable stack protection
#10 Disable ASLR whenever you can.
#11 Avoid hardware and operating systems that enforce DEP/W^X/NX-bit
#12 Make it easy to find many ROP gadgets in your program
#13 Skip integrity checks when installing and running new software.
#14 Write insecure code by leaking information



## Some tricks for insecure coding in C and C++

#0 Never ever let other programmers review your code
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by doing sequence point violations
#3 Write insecure code where the result depends on the compiler
#4 Know the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Understand how the optimizer can and will remove apparently critical code
#8 Write code that allows buffer overflows
#9 Disable stack protection
#10 Disable ASLR whenever you can.
#11 Avoid hardware and operating systems that enforce DEP/W^X/NX-bit
#12 Make it easy to find many ROP gadgets in your program
#13 Skip integrity checks when installing and running new software.
#14 Write insecure code by leaking information

http://www.pvv.org/~oma/InsecureCodingC_Bangalore_Nov2014.pdf

just a quick recap of the most important thing I covered at SecConX Bangalore last year

On undefined behavior **anything** can happen!

On undefined behavior **anything** can happen!

## On undefined behavior **anything** can happen!

When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose” [comp.std.c]

you can not reason about undefined behavior!

## In practice, what do you think happens if you run this program on your machine?

foo.c

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

In practice, what do you think happens if you run this program on your machine?

foo.c

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Here is what happens on my machine:

## In practice, what do you think happens if you run this program on your machine?

foo.c

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Here is what happens on my machine:

icc

```
13
```

## In practice, what do you think happens if you run this program on your machine?

foo.c

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Here is what happens on my machine:

icc

```
13
```

clang

```
11
```

## In practice, what do you think happens if you run this program on your machine?

foo.c

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Here is what happens on my machine:

icc

```
13
```

clang

```
11
```

gcc

```
12
```

## In practice, what do you think happens if you run this program on your machine?

foo.c

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Here is what happens on my machine:

icc

```
13
```

clang

```
11
```

gcc

```
12
```

Here is what "could" also happen:

## In practice, what do you think happens if you run this program on your machine?

foo.c

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Here is what happens on my machine:

icc

```
13
```

clang

```
11
```

gcc

```
12
```

Here is what "could" also happen:

x

```
core dump
```

## In practice, what do you think happens if you run this program on your machine?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Here is what happens on my machine:

icc

```
13
```

clang

```
11
```

gcc

```
12
```

Here is what "could" also happen:

x

```
core dump
```

y

```
3.14169265
```

## In practice, what do you think happens if you run this program on your machine?

foo.c

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Here is what happens on my machine:

icc

```
13
```

clang

```
11
```

gcc

```
12
```

Here is what "could" also happen:

x

```
core dump
```

y

```
3.14169265
```

z

```
launching
42 missiles
```

## In practice, what do you think happens if you run this program on your machine?

foo.c

```c
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

Here is what happens on my machine:

icc

```
13
```

clang

```
11
```

gcc

```
12
```

Here is what "could" also happen:

x

```
core dump
```

y

```
3.14169265
```

z

```
launching
42 missiles
```

see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

## you can not reason about undefined behavior!

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine: *

* if the uninitialized internal representation for b is a positive even number

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:*

icc

```
true
```

* if the uninitialized internal representation for b is a positive even number

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:*

icc

```
true
```

clang

```
false
```

* if the uninitialized internal representation for b is a positive even number

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:*

icc

```
true
```

clang

```
false
```

gcc

* if the uninitialized internal representation for b is a positive even number

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:*

icc

```
true
```

clang

```
false
```

gcc

```
true
```

* if the uninitialized internal representation for b is a positive even number

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:*

icc

```
true
```

clang

```
false
```

gcc

```
true
false
```

* if the uninitialized internal representation for b is a positive even number

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:*

icc

```
true
```

clang

```
false
```

gcc

```
true
false
```

* if the uninitialized internal representation for b is a positive even number

see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:*

icc

```
true
```

clang

```
false
```

gcc

```
true
false
```

Here is what "could" also happen:

* if the uninitialized internal representation for b is a positive even number

see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:*

icc

```
true
```

clang

```
false
```

gcc

```
true
false
```

Here is what "could" also happen:

x

```
core dump
```

* if the uninitialized internal representation for b is a positive even number

see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:*

icc

```
true
```

clang

```
false
```

gcc

```
true
false
```

Here is what "could" also happen:

x

```
core dump
```

y

```
3.14169265
```

* if the uninitialized internal representation for b is a positive even number

see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

## In practice, what do you think are possible outcomes when this function is called?

foo.c

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

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Here is what I can observe on my machine:*

icc

```
true
```

clang

```
false
```

gcc

```
true
false
```

Here is what "could" also happen:

x

```
core dump
```

y

```
3.14169265
```

z

```
launching
42 missiles
```

* if the uninitialized internal representation for b is a positive even number

see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

## YOU CAN NOT REASON ABOUT UNDEFINED BEHAVIOR!

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
```

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
```

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
```

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
2147483647
```

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
2147483647
```

clang

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
2147483647
```

clang

```
2147483644
2147483645
2147483646
2147483647
```

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
2147483647
```

clang

```
2147483644
2147483645
2147483646
2147483647
-2147483648
```

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
2147483647
```

clang

```
2147483644
2147483645
2147483646
2147483647
-2147483648
```

gcc

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
2147483647
```

clang

```
2147483644
2147483645
2147483646
2147483647
-2147483648
```

gcc

```
2147483644
2147483645
2147483646
2147483647
-2147483648
-2147483647
-2147483646
...
```

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
2147483647
```

clang

```
2147483644
2147483645
2147483646
2147483647
-2147483648
```

gcc

```
2147483644
2147483645
2147483646
2147483647
-2147483648
-2147483647
-2147483646
...
```

Here is what "could" also happen:

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
2147483647
```

clang

```
2147483644
2147483645
2147483646
2147483647
-2147483648
```

gcc

```
2147483644
2147483645
2147483646
2147483647
-2147483648
-2147483647
-2147483646
...
```

Here is what "could" also happen:

x

```
core dump
```

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
2147483647
```

clang

```
2147483644
2147483645
2147483646
2147483647
-2147483648
```

gcc

```
2147483644
2147483645
2147483646
2147483647
-2147483648
-2147483647
-2147483646
...
```

Here is what "could" also happen:

x

```
core dump
```

y

```
3.14169265
```

* with optimization

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

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

Here is what happens on my machine:*

icc

```
2147483644
2147483645
2147483646
2147483647
```

clang

```
2147483644
2147483645
2147483646
2147483647
-2147483648
```

gcc

```
2147483644
2147483645
2147483646
2147483647
-2147483648
-2147483647
-2147483646
...
```

Here is what "could" also happen:

x

```
core dump
```

y

```
3.14169265
```

z

```
launching
42 missiles
```

* with optimization

repeat after me:

repeat after me:

you can not reason about undefined behavior!

repeat after me:

you can not reason about undefined behavior!
**you can not reason about undefined behavior!**

repeat after me:

you can not reason about undefined behavior!
**you can not reason about undefined behavior!**
**YOU CAN NOT REASON ABOUT UNDEFINED BEHAVIOR!**

## about integers

```
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

What is the potential problem
with this function?

```
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

What is the potential problem with this function?

Signed integer overflow is undefined behavior.

```
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int a = 3;
int b = 7;
int m = midpoint(a, b);
printf("%d\n", m);
```

```c
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int a = 3;
int b = 7;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
5
```

```c
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int a = 3;
int b = 7;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
5
```

```c
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int a = -3;
int b = -7;
int m = midpoint(a, b);
printf("%d\n", m);
```

```c
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int a = -3;
int b = -7;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
-5
```

```c
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int a = -3;
int b = -7;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
-5
```

```
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
-147483643
```

```c
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
core dump
```

```c
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
launching 42 missiles
```

```
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
launching 42 missi
```

you might want to consider the -ftrapv flag to signal failure on sign integer overflow

```c
int midpoint(int a, int b)
{
    return (a + b) / 2;
}
```

```c
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
launching 42 missiles
```

```
int midpoint(int a, int b)
{
    return a/2 + b/2;
}
```

```
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```c
int midpoint(int a, int b)
{
    return a/2 + b/2;
}
```

```c
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
2000000004
```

```
int midpoint(int a, int b)
{
    return a/2 + b/2;
}
```

```
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
2000000004
```

```
int midpoint(int a, int b)
{
    return a + (b - a) / 2;
}
```

```
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```c
int midpoint(int a, int b)
{
    return a + (b - a) / 2;
}
```

```c
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
2000000005
```

```
int midpoint(int a, int b)
{
    return a + (b - a) / 2;
}
```

```
int a = 2000000003;
int b = 2000000007;
int m = midpoint(a, b);
printf("%d\n", m);
```

```
2000000005
```

```c
int midpoint(int a, int b)
{
    return a + (b - a) / 2;
}
```

```c
int a = 2000000000;
int b = -2000000000;
int m = midpoint(a, b);
printf("%d\n", m);
```

```c
int midpoint(int a, int b)
{
    return a + (b - a) / 2;
}
```

```c
int a = 2000000000;
int b = -2000000000;
int m = midpoint(a, b);
printf("%d\n", m);
```

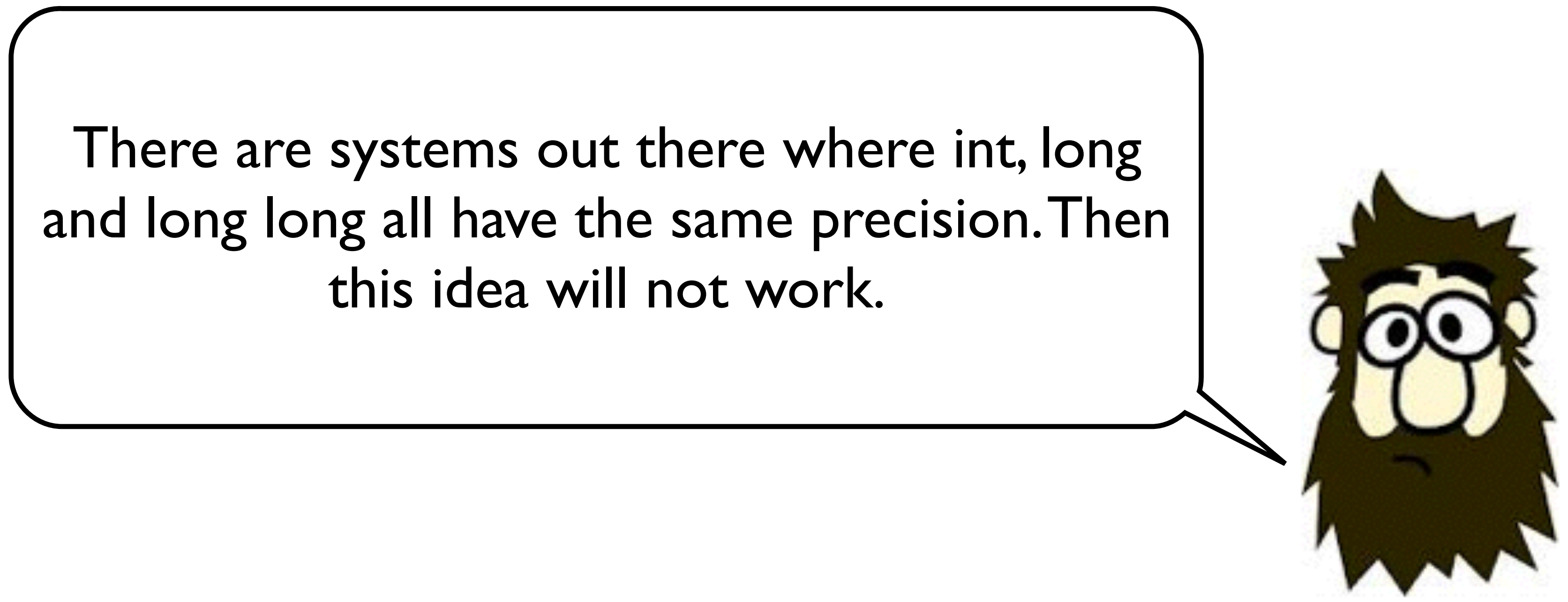
```
launching 42 missiles
```

```
int midpoint(int a, int b)
{
    return ((long)a + (long)b) / 2;
}
```

```
int midpoint(int a, int b)
{
    return ((long long)a + (long long)b) / 2;
}
```

```
int midpoint(int a, int b)
{
    return ((intmax_t)a + (intmax_t)b) / 2;
}
```

```
int midpoint(int a, int b)
{
    return ((intmax_t)a + (intmax_t)b) / 2;
}
```


There are systems out there where int, long and long long all have the same precision. Then this idea will not work.

```
bool is_pos(int a) { return a >= 0; }
bool is_neg(int a) { return a < 0; }
bool is_even(int a) { return !(a % 2); }
bool is_odd(int a) { return a % 2; }

int midpoint(int a, int b) {
    if (a > b) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    if (is_pos(a) && is_pos(b))
        return a + (b - a) / 2;
    if (is_neg(a) && is_pos(b))
        return (b + a) / 2;
    if (is_odd(a) != is_odd(b))
        return a + (b - a) / 2 + 1;
    return a + (b - a) / 2;
}
```

I look forward to someone sending me a better solution a * b

```
a * b
```

or

```
a * b
```

or

```
signed int result;
if (si_a > 0) {  /* si_a is positive */
  if (si_b > 0) {  /* si_a and si_b are positive */
    if (si_a > (INT_MAX / si_b)) {
      /* Handle error */
    }
  } else { /* si_a positive, si_b nonpositive */
    if (si_b < (INT_MIN / si_a)) {
      /* Handle error */
    }
  } /* si_a positive, si_b nonpositive */
} else { /* si_a is nonpositive */
  if (si_b > 0) { /* si_a is nonpositive, si_b is positive */
    if (si_a < (INT_MIN / si_b)) {
      /* Handle error */
    }
  } else { /* si_a and si_b are nonpositive */
    if ( (si_a != 0) && (si_b < (INT_MAX / si_a))) {
      /* Handle error */
    }
  } /* End if si_a and si_b are nonpositive */
} /* End if si_a is nonpositive */

result = si_a * si_b;
```

```
a * b
```

or

```
signed int result;
if (si_a > 0) {  /* si_a is positive */
  if (si_b > 0) {  /* si_a and si_b are positive */
    if (si_a > (INT_MAX / si_b)) {
      /* Handle error */
    }
  } else { /* si_a positive
    if (si_b < (INT_MIN
      /* Handle
    }

                           e, si_b is positive */
                              {

        /* si_a and si_b are nonpositive */
    ( (si_a != 0) && (si_b < (INT_MAX / si_a))) {
      /* Handle error */
    }
  } /* End if si_a and si_b are nonpositive */
} /* End if si_a is nonpositive */

result = si_a * si_b;
```

Tip:

Ensure that operations on signed integers do not result in overflow (INT32-C)

```
a * b
```

or

```
signed int result;
if (si_a > 0) {  /* si_a is positive */
  if (si_b > 0) {  /* si_a and si_b are positive */
    if (si_a > (INT_MAX / si_b)) {
      /* Handle error */
    }
  } else { /* si_a positive
    if (si_b < (INT MIN
      /* Handle
    }
                                      e, si_b is positive */
                                      {

          si_a and si_b are nonpositive */
      ( (si_a != 0) && (si_b < (INT_MAX / si_a))) {
      /* Handle error */
    }
  } /* End if si_a and si_b are nonpositive */
} /* End if si_a is nonpositive */

result = si_a * si_b;
```

Tip:

Ensure that operations on signed integers do not result in overflow (INT32-C)

https://www.securecoding.cert.org/

```c
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

```
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

What is the potential problem with this function?

```
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

What is the potential problem with this function?

Unsigned integers can not overflow but even well defined wrapping can give surprising results.

```
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

```c
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

```c
unsigned int a = 2;
unsigned int b = 4;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```c
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

```c
unsigned int a = 2;
unsigned int b = 4;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```
3
```

```
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

```
unsigned int a = 2;
unsigned int b = 4;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```
3
```

```c
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

```c
unsigned int a = 2;
unsigned int b = 4;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

```
unsigned int a = 4000000002;
unsigned int b = 4000000004;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```c
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

```c
unsigned int a = 4000000002;
unsigned int b = 4000000004;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```c
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

```c
unsigned int a = 4000000002;
unsigned int b = 4000000004;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```
1852516355
```

```
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return (a + b) / 2;
}
```

```
unsigned int a = 4000000002;
unsigned int b = 4000000004;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```
1852516355
```

```c
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return a + (b - a) / 2;
}
```

```c
unsigned int a = 4000000002;
unsigned int b = 4000000004;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return a + (b - a) / 2;
}
```

```
unsigned int a = 4000000002;
unsigned int b = 4000000004;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```
4000000003
```

```c
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return a + (b - a) / 2;
}
```

```c
unsigned int a = 4000000002;
unsigned int b = 4000000004;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```
4000000003
```

```c
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return a + (b - a) / 2;
}
```

```c
unsigned int a = 4000000004;
unsigned int b = 4000000002;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```
unsigned int midpoint(unsigned int a, unsigned int b)
{
    return a + (b - a) / 2;
}
```

```
unsigned int a = 4000000004;
unsigned int b = 4000000002;
unsigned int m = midpoint(a,b);
printf("%u\n", m);
```

```
1852516355
```

```
unsigned int midpoint(unsigned int a, unsigned int b)
{
    if (a < b)
        return a + (b - a) / 2;
    else
        return b + (a - b) / 2;
}
```

```
unsigned int midpoint(unsigned int          gned int b)
{
    if (a < b)
        retu
    else
        retu                  / 2;
}
```

Tip:

Ensure that unsigned integer operations do not wrap (INT30-C)

```c
void foo(void)
{
    unsigned int a = 2;
    if (a > -1)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
void foo(void)
{
    unsigned int a = 2;
    if (a > -1)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

What do you think this code snippet will print?

```
void foo(void)
{
    unsigned int a = 2;
    if (a > -1)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
Bar
```

What do you think this code snippet will print?

```
void foo(void)
{
    unsigned int a = 2;
    if (a > -1)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
Bar
```

What do you think this code snippet will print?

```c
void foo(void)
{
    unsigned int a = 2;
    if (a > -1L)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
void foo(void)
{
    unsigned int a = 2;
    if (a > -1L)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
Foo
```

```
void foo(void)
{
    unsigned int a = 2;
    if (a > -1L)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
Foo
```

on systems where sizeof(long) > sizeof(int)

```
void foo(void)
{
    unsigned int a = 2;
    if (a > -1L)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
Foo
```

on systems where sizeof(long) > sizeof(int)

```
Bar
```

```
void foo(void)
{
    unsigned int a = 2;
    if (a > -1L)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
Foo
```

on systems where sizeof(long) > sizeof(int)

```
Bar
```

on systems where sizeof(long) == sizeof(int)

```
void foo(void)
{
                 oo\n");

             printf("Bar\n");
}
```

Tip:

Understand the data model used by your implementation(s) (INT00-C)

```
Foo
```

on systems where sizeof(long) > sizeof(int)

```
Bar
```

on systems where sizeof(long) == sizeof(int)

```
void foo(void)
{
                             oo\n");
          printf(
}
```

Tip:

Understand the data model used by your implementation(s) (INT00-C)

Tip:

Understand integer conversion rules (INT02-C)

on systems where sizeof(long) > sizeof(int)

```
Bar
```

on systems where sizeof(long) == sizeof(int)

```
void foo(void)
{
    char a = -2;
    if (42 + a > 42)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
void foo(void)
{
    char a = -2;
    if (42 + a > 42)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
void foo(void)
{
    char a = -2;
    if (42 + a > 42)
        printf("Foo\n");
    else
        printf("Bar\n");
}
```

```
Foo
```

on systems where default char is unsigned

```
Bar
```

on systems where default char is signed

```
void foo(void)
{
    char
                          );
        printf("Bar\n");
}
```

Tip:

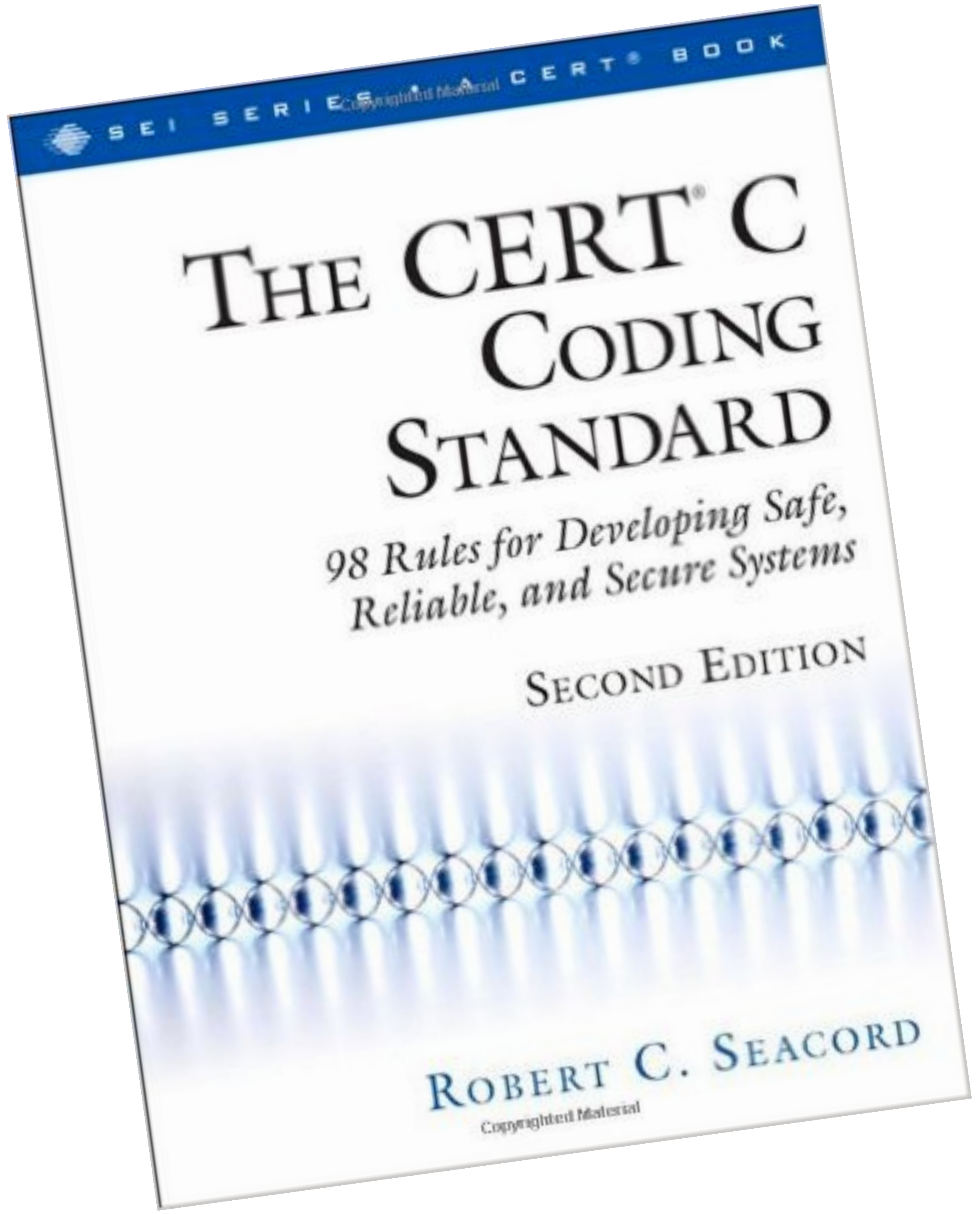
Use only explicitly signed or unsigned char type for numeric values (INT07-C)

```
Foo
```

on systems where default char is unsigned

```
Bar
```

on systems where default char is signed



https://www.securecoding.cert.org
(~250 rules and recommendations)

## CERT C Rules and Recommendations

- Preprocessor (PRE)
- Declarations and Initialization (DCL)
- Expressions (EXP)
- Integers (INT)
- Floating Point (FLP)
- Arrays (ARR)
- Characters and Strings (STR)
- Memory Management (MEM)
- Input Output (FIO)
- Environment (ENV)
- Signals (SIG)
- Error Handling (ERR)
- Application Programming Interfaces (API)
- Concurrency (CON)

## CERT C Rules and Recommendations

- Preprocessor (PRE)
- Declarations and Initialization (DCL)
- Expressions (EXP)
- **Integers (INT)**
- Floating Point (FLP)
- Arrays (ARR)
- Characters and Strings (STR)
- Memory Management (MEM)
- Input Output (FIO)
- Environment (ENV)
- Signals (SIG)
- Error Handling (ERR)
- Application Programming Interfaces (API)
- Concurrency (CON)

## CERT C Rules and Recommendations about integers

- **Understand the data model used by your implementation(s) (INT00-C)**
- Use rsize_t or size_t for all integer values representing the size of an object (INT01-C)
- **Understand integer conversion rules (INT02-C)**
- Enforce limits on integer values originating from tainted sources (INT04-C)
- Do not use input functions to convert character data if they cannot handle all possible inputs (INT05-C)
- Use strtol() or a related function to convert a string token to an integer (INT06-C)
- **Use only explicitly signed or unsigned char type for numeric values (INT07-C)**
- Verify that all integer values are in range (INT08-C)
- Ensure enumeration constants map to unique values (INT09-C)
- Do not assume a positive remainder when using the % operator (INT10-C)
- Do not make assumptions about the type of a plain int bit-field when used in an expression (INT12-C)
- Use bitwise operators only on unsigned operands (INT13-C)
- Avoid performing bitwise and arithmetic operations on the same data (INT14-C)
- Use intmax_t or uintmax_t for formatted IO on programmer-defined integer types (INT15-C)
- Do not make assumptions about representation of signed integers (INT16-C)
- Define integer constants in an implementation-independent manner (INT17-C)
- Evaluate integer expressions in a larger size before comparing or assigning to that size (INT18-C)
- **Ensure that unsigned integer operations do not wrap (INT30-C)**
- **Ensure that integer conversions do not result in lost or misinterpreted data (INT31-C)**
- **Ensure that operations on signed integers do not result in overflow (INT32-C)**
- Ensure that division and remainder operations do not result in divide-by-zero errors (INT33-C)
- Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand (INT34-C)
- Use correct integer precisions (INT35-C)
- Converting a pointer to integer or integer to pointer (INT36-C)

## beware of unspecified behavior

beware of unspecified behavior

and

beware of unspecified behavior

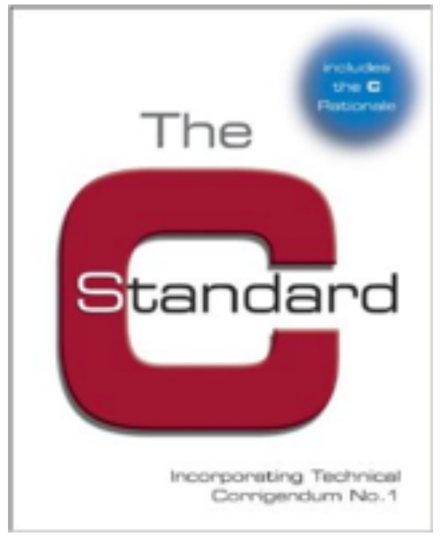
and

you can not reason about undefined behavior!

!

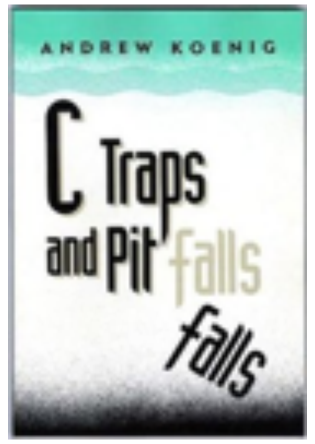
# Resources

"C Programming Language" by Kernighan and Ritchie is a book that you need to read over and over again. Security vulnerabilities and bugs in C are very often just a result of not using the language correctly. Instead of trying to remember everthing as it is formally written in the C standard, it is better to try to understand the spirit of C and try to understand why things are designed as they are in the language. Nobody tells this story better than K&R.

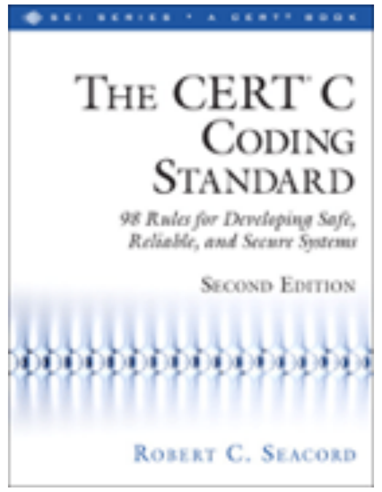


All professional C programmers should have a copy of the C standard and they should get used to regularly look up terms and concepts in the standard. It is easy to find cheap PDF-version of the standard ($30), but you can also just download the latest draft and they are usually 99,93% the same as the real thing. I also encourage everyone to read the Rationale for C99 which is available for free on the WG14 site. http://www.open-std.org/jtc1/sc22/wg14/

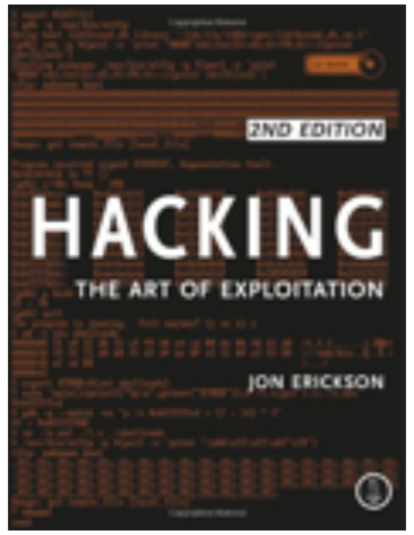
I got my first serious journey into deeper understanding of C came when I read Peter van der Linden wonderful book "Expert C programming" (1994). I consider it as one of the best books ever written about C.



"C traps and pitfalls" by Andrew Koenig (1988) is also a very good read.



The CERT C Coding Standard contains a lot of good advice and insightful recommendations. While I don't recommend anyone to blindly follow all the guidelines here (some of them are rather silly), but there is a lot of wisdom in most of the guidelines.



This is a really nice book about how to hack into systems and programs written in C. The book also has a surprisingly concise and well written introduction to C as a programming language.