Undefined Behavior

CS 211

Initial code setup

The code in this course is available in your Unix shell account. You can get your own copy like this:

```
% cd cs211
% tar -xvkf ~cs211/lec/07_ub.tgz
⋮
% cd 07_ub
```

Road map

The awful truth about `int`

Road map

The awful truth about `int`

WTF

Road map

The awful truth about `int`

WTF

Examples of undefined behavior

Up next

The awful truth about `int`

WTF

Examples of undefined behavior

Something funny about `int`

Not every mathematical integer can fit in a C `int`.

Something funny about `int`

Not every mathematical integer can fit in a C `int`.

- An `int` is stored in a finite number of bits (like 16 or 32 or 64)

Something funny about `int`

Not every mathematical integer can fit in a C `int`.

- An `int` is stored in a finite number of bits (like 16 or 32 or 64)
- This means that it has a finite range

Something funny about `int`

Not every mathematical integer can fit in a C `int`.

- An `int` is stored in a finite number of bits (like 16 or 32 or 64)
- This means that it has a finite range
- For example, 32-bit `int`s (usually) range from -2^{31} to $2^{31} - 1$ (inclusive)

Something funny about `int`

Not every mathematical integer can fit in a C `int`.

- An `int` is stored in a finite number of bits (like 16 or 32 or 64)
- This means that it has a finite range
- For example, 32-bit `int`s (usually) range from -2^{31} to $2^{31} - 1$ (inclusive)
- The actual values are defined in `<limits.h>` as *`INT_MIN`* and *`INT_MAX`*

Something funny about `int`

Not every mathematical integer can fit in a C `int`.

- An `int` is stored in a finite number of bits (like 16 or 32 or 64)
- This means that it has a finite range
- For example, 32-bit `int`s (usually) range from -2^{31} to $2^{31}-1$ (inclusive)
- The actual values are defined in `<limits.h>` as *`INT_MIN`* and *`INT_MAX`*
- An `int` operation whose mathematical result is out of range causes

UNDEFINED BEHAVIOR

Let’s see these limits

src/limits.c

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

#define SHOW_ME(Type, Fmt, Min, Max) \
    printf("%-19s %2zu bytes %21" Fmt " to %-21" Fmt "\n", \
           #Type, sizeof(Type), (Type)Min, (Type)Max)

int main(void)
{
    SHOW_ME(int,           "d",   INT_MIN,    INT_MAX);
    SHOW_ME(long,          "ld",  LONG_MIN,   LONG_MAX);

    SHOW_ME(unsigned int,  "u",   0,          UINT_MAX);
    SHOW_ME(unsigned long, "lu",  0L,         ULONG_MAX);
}
```

Up next

The awful truth about `int`

WTF

Examples of UNDEFINED BEHAVIOR

WTF IS UNDEFINED BEHAVIOR?!?!

It’s like a kind of error…

WTF IS UNDEFINED BEHAVIOR?!?!

It’s like a kind of error…

But the computer doesn’t necessarily notice…

WTF IS UNDEFINED BEHAVIOR?!?!

It’s like a kind of error…

But the computer doesn’t necessarily notice…

Your program might just keep running and produce nonsense!

WTF IS UNDEFINED BEHAVIOR?!?!

It’s like a kind of error...

But the computer doesn’t necessarily notice...

Your program might just keep running and produce nonsense!

Technically, a program with UB has no meaning. It’s allowed to do anything:

WTF IS UNDEFINED BEHAVIOR?!?!

It's like a kind of error...

But the computer doesn't necessarily notice...

Your program might just keep running and produce nonsense!

Technically, a program with UB has no meaning. It's allowed to do anything:

- Crash

WTF IS UNDEFINED BEHAVIOR?!?!

It's like a kind of error...

But the computer doesn't necessarily notice...

Your program might just keep running and produce nonsense!

Technically, a program with UB has no meaning. It's allowed to do anything:

- Crash
- Keep going

WTF IS UNDEFINED BEHAVIOR?!?!

It’s like a kind of error…

But the computer doesn’t necessarily notice…

Your program might just keep running and produce nonsense!

Technically, a program with UB has no meaning. It’s allowed to do anything:

- Crash
- Keep going
- Reformat your hard disk

WTF IS UNDEFINED BEHAVIOR?!?!

It’s like a kind of error...

But the computer doesn’t necessarily notice...

Your program might just keep running and produce nonsense!

Technically, a program with UB has no meaning. It’s allowed to do anything:

- Crash
- Keep going
- Reformat your hard disk
- Launch the missiles

No Traveling

From Prof. John Regehr, an expert on C compilation:

It is very common for people to say—or at least think—something like this:

> *The x86 ADD instruction is used to implement C's signed add operation, and it has two's complement behavior when the result overflows. I'm developing for an x86 platform, so I should be able to expect two's complement semantics when 32-bit signed integers overflow.*

No Traveling

From Prof. John Regehr, an expert on C compilation:

It is very common for people to say—or at least think—something like this:

> *The x86 ADD instruction is used to implement C's signed add operation, and it has two's complement behavior when the result overflows. I'm developing for an x86 platform, so I should be able to expect two's complement semantics when 32-bit signed integers overflow.*

THIS IS WRONG. You are saying something like this:

> *Somebody once told me that in basketball you can't hold the ball and run. I got a basketball and tried it and it worked just fine. He obviously didn't understand basketball.*

https://blog.regehr.org/archives/213

Up next

The awful truth about `int`

WTF

Examples of UNDEFINED BEHAVIOR

Some examples of UB

- Uninitialized memory access
- Integer division by 0
- Integer result out of range (“overflow”)

Some examples of UB

- Uninitialized memory access
- Integer division by 0
- Integer result out of range (“overflow”)

Example of all three:

```
int x, y;
scanf("%d%d", &x, &y);
printf("%d\n", x / y);
```

Some examples of UB

- Uninitialized memory access
- Integer division by 0
- Integer result out of range (“overflow”)

Example of all three:

```
int x, y;
scanf("%d%d", &x, &y);
printf("%d\n", x / y);
```

Fix for all three:

```
int x, y;
if (scanf("%d%d", &x, &y) == 2 &&
        y != 0 &&
        !(x == INT_MIN && y == -1))
    printf("%d\n", x / y);
```

UB is really weird

src/int_max.c

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

void how_about_this_int(int z)
{
    puts(z < z + 1 ? "math" : "C.S.");
}

int main(void)
{
    how_about_this_int(0);
    how_about_this_int(INT_MAX);
}
```

The results depend on the optimization level

%

The results depend on the optimization level

```
% make int_max
```

The results depend on the optimization level

```
% make int_max
cc -o int_max src/int_max.c -std=c11 -pedanti…
%
```

The results depend on the optimization level

```
% make int_max
cc -o int_max src/int_max.c -std=c11 -pedanti…
% ./int_max
```

The results depend on the optimization level

```
% make int_max
cc -o int_max src/int_max.c -std=c11 -pedanti…
% ./int_max
math
C.S.
%
```

The results depend on the optimization level

```
% make int_max
cc -o int_max src/int_max.c -std=c11 -pedanti…
% ./int_max
math
C.S.
% make int_max.opt
```

The results depend on the optimization level

```
% make int_max
cc -o int_max src/int_max.c -std=c11 -pedanti…
% ./int_max
math
C.S.
% make int_max.opt
cc -O2 -o int_max.opt src/int_max.c -std=c11 …
%
```

The results depend on the optimization level

```
% make int_max
cc -o int_max src/int_max.c -std=c11 -pedanti…
% ./int_max
math
C.S.
% make int_max.opt
cc -O2 -o int_max.opt src/int_max.c -std=c11 …
% ./int_max.opt
```

The results depend on the optimization level

```
% make int_max
cc -o int_max src/int_max.c -std=c11 -pedanti…
% ./int_max
math
C.S.
% make int_max.opt
cc -O2 -o int_max.opt src/int_max.c -std=c11 …
% ./int_max.opt
math
math
%
```

The results depend on the optimization level

```
% make int_max
cc -o int_max src/int_max.c -std=c11 -pedanti…
% ./int_max
math
C.S.
% make int_max.opt
cc -O2 -o int_max.opt src/int_max.c -std=c11 …
% ./int_max.opt
math
math
%
```

(This is very, very bad.)

A few more things that are UNDEFINED in C

- Dereferencing a null or freed pointer

A few more things that are UNDEFINED in C

- Dereferencing a null or freed pointer
- Pointer arithmetic that goes out of bounds

A few more things that are UNDEFINED in C

- Dereferencing a null or freed pointer
- Pointer arithmetic that goes out of bounds (except for one-past-the-end, which is okay)

A few more things that are UNDEFINED in C

- Dereferencing a null or freed pointer
- Pointer arithmetic that goes out of bounds (except for one-past-the-end, which is okay)
- Comparing unrelated pointers with `<`, `<=`, `>`, or `>=`

A few more things that are UNDEFINED in C

- Dereferencing a null or freed pointer
- Pointer arithmetic that goes out of bounds (except for one-past-the-end, which is okay)
- Comparing unrelated pointers with `<`, `<=`, `>`, or `>=`
- Any arithmetic operation on a *signed* integral type whose mathematical result is out of range for that type

A few more things that are UNDEFINED in C

- Dereferencing a null or freed pointer
- Pointer arithmetic that goes out of bounds (except for one-past-the-end, which is okay)
- Comparing unrelated pointers with `<`, `<=`, `>`, or `>=`
- Any arithmetic operation on a *signed* integral type whose mathematical result is out of range for that type
- Passing a function the wrong number or types of arguments

A few more things that are UNDEFINED in C

- Dereferencing a null or freed pointer
- Pointer arithmetic that goes out of bounds (except for one-past-the-end, which is okay)
- Comparing unrelated pointers with `<`, `<=`, `>`, or `>=`
- Any arithmetic operation on a *signed* integral type whose mathematical result is out of range for that type
- Passing a function the wrong number or types of arguments
- Reaching the end of a non-`void` function without returning a result, if the caller uses the result

A few more things that are UNDEFINED in C

- Dereferencing a null or freed pointer
- Pointer arithmetic that goes out of bounds (except for one-past-the-end, which is okay)
- Comparing unrelated pointers with `<`, `<=`, `>`, or `>=`
- Any arithmetic operation on a *signed* integral type whose mathematical result is out of range for that type
- Passing a function the wrong number or types of arguments
- Reaching the end of a non-`void` function without returning a result, if the caller uses the result
- Performing two side-effecting operations on the same object in an indeterminate order (*e.g.*, `++x + ++x`)

A few more things that are UNDEFINED in C

- Dereferencing a null or freed pointer
- Pointer arithmetic that goes out of bounds (except for one-past-the-end, which is okay)
- Comparing unrelated pointers with `<`, `<=`, `>`, or `>=`
- Any arithmetic operation on a *signed* integral type whose mathematical result is out of range for that type
- Passing a function the wrong number or types of arguments
- Reaching the end of a non-`void` function without returning a result, if the caller uses the result
- Performing two side-effecting operations on the same object in an indeterminate order (*e.g.*, `++x + ++x`)
- ...and many more!

– Next time: Linked Data Structures (for real) –