Abstraction example - Integers

Variable semantics

- What does the following line of code do?

```
unsigned k = 0;
```


- Allocates space for the variable
- Associates the space with the variable
- Defines how the [illegible] interpreted
- Initialises the sto[illegible]

Let's test this

Example

```c
#include <stdio.h>

int main() {
  int i;
  int sum = 0;

  for (i = 0; i < 10; i++)
    sum += i;

  printf("The sum is %d\n", sum);
}
```

```
_main:
    pushq   %rbp
    movq    %rsp, %rbp
    leaq    L_.str(%rip), %rdi
    movl    $45, %esi
    xorl    %eax, %eax
    callq   _printf
    xorl    %eax, %eax
    popq    %rbp
    retq
L_.str:
    .asciz  "The sum is %d\n"
```

Compiler abstractions

- The compiler generates a ***concrete implementation*** of an ***abstract program*** description
- Preserves semantics (to some extent)
- Changes implementation details.
  - May not match intuition

- We need to understand the abstraction

Unsigned integers

- What is the output from:

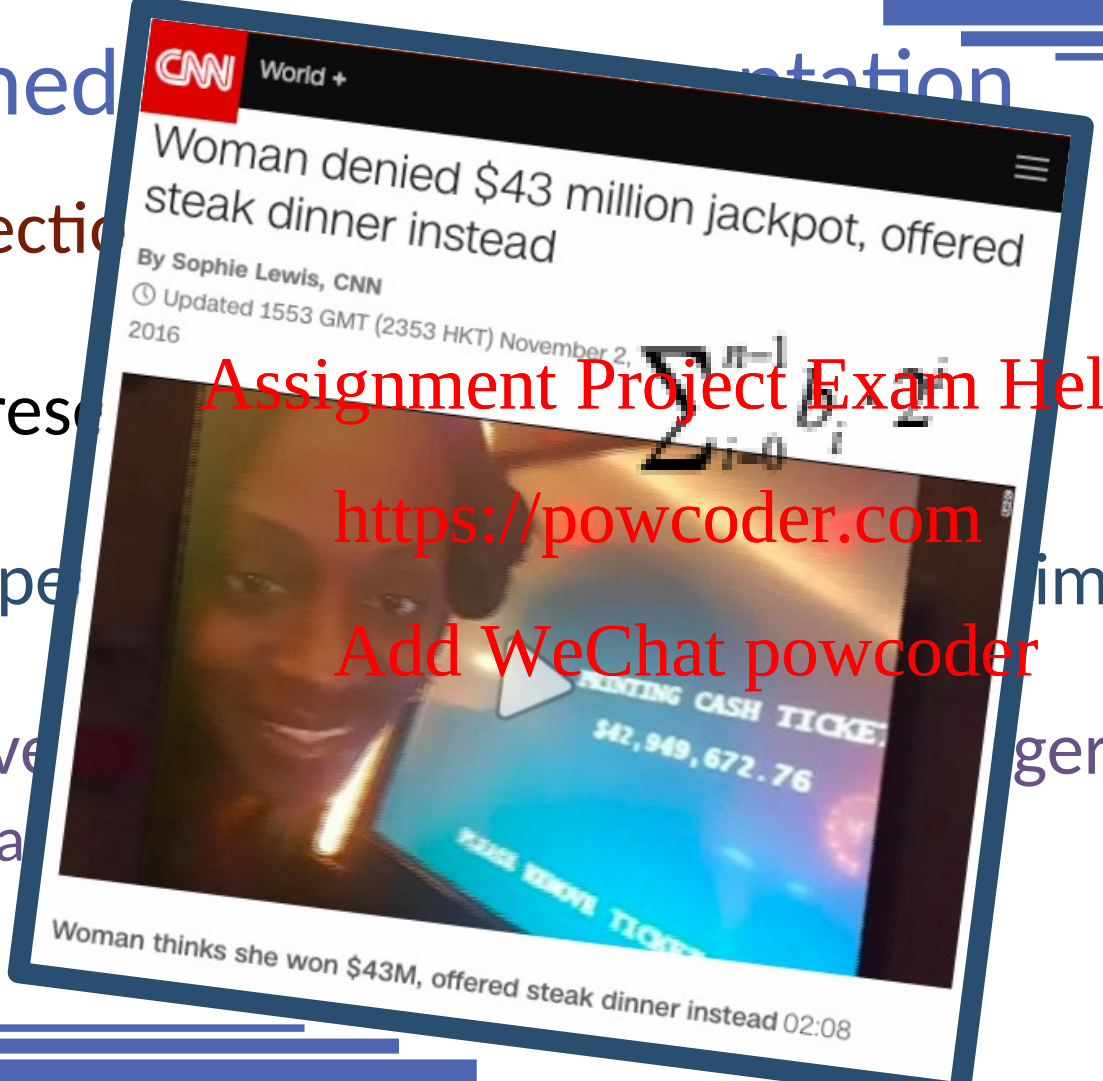
```
$ count 4
$ count 2949
$ count 67295
```



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

int main(int c, char **v) {
  unsigned i, n;

  n = strtoul(v[1], NULL, 0);

  for (i = n; i < n+10; i++)
    printf("%u\n", i);
}
```

Unsigned

- Collectio
- Represe
- n depe implementation
- An ove ger than 2^n
  - All a

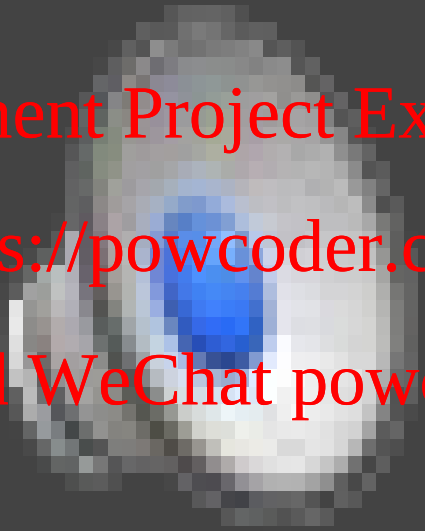


Signed Integers

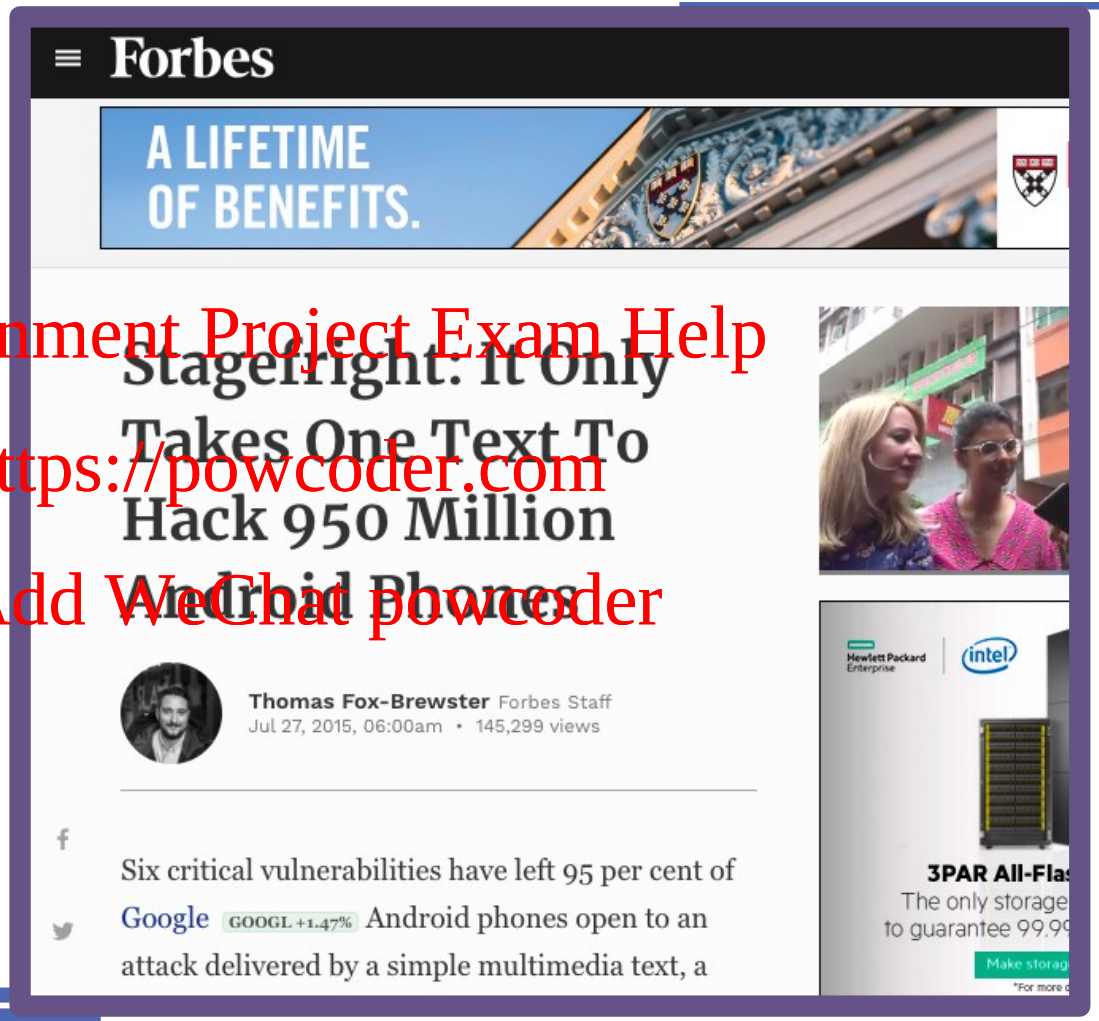
- One sign bit s and n-1 value bits $b_0 \ldots b_{n-2}$
- Three possible interpretations:
  - Sign magnitude $(1-2s)\sum_{i=0}^{n-2} b_i \cdot 2^i$
  - Ones' complement $(1-2s)\sum_{i=0}^{n-2} (b_i \oplus s) \cdot 2^i$
  - Two's complement $\sum_{i=0}^{n-2} b_i \cdot 2^i - s \cdot 2^{n-1}$
- Most modern processors use two's complement

Signed integer overflow

- Undefined behaviour
  - Use modulo 2^{n-1} arithmetic
  - Return maximums or minimum values
  - Return zero
  - Do nothing
  - Cause a trap
  - Launch $500M fireworks



Integer overflow vulnerabilities



Stagefright

- Before

```
uint8_t *buffer = new (std::nothrow) uint8_t[size + chunk_size];
if (buffer == NULL) {
    return ERROR_MALFORMED;
}
```

- After

```
if (SIZE_MAX - chunk_size <= size) {
    return ERROR_MALFORMED;
}

uint8_t *buffer = new uint8_t[size + chunk_size];
if (buffer == NULL) {
```

Why not use `chunk_size+size >= SIZE_MAX` ?

Type Conversion

```
bool isValidAddition(uint16_t x, uint16_t y)
{
  if (x + y < x)
    return false;
  return true;
}
```


```
if ((uint16_t)(x + y) < x)
```

CVE-2017-7602 (LibTIFF)

```
ma=(tmsize_t)td->td_stripoffset[strip];
mb=ma+size;
if ((td->td_stripoffset[strip] > (uint64)TIFF_TMSIZE_T_MAX)||(ma>tif->tif_size))
        n=0;
else if ((mb<ma)||(mb<size)||(mb>tif->tif_size))
        n=tif->tif_size-ma;
else
        n=size;
```

ma is positive

mb >= size
(overflow ignored)

Test removed!

Fix: test for overflow

```
if ((td->td_stripoffset[strip] > (uint64)TIFF_TMSIZE_T_MAX)||
     ((ma=(tmsize_t)td->td_stripoffset[strip])>tif->tif_size))
 {
     n=0;
 }
 else if( ma > TIFF_TMSIZE_T_MAX - size )
 {
     n=0;
 }
 else
```

Best practices

- Know the language
  - Undefined behaviours are dangerous
- Test user input for overflow
  - Special attention to input that affects allocation
- Use safe tests
  - Subtract from maximum
- Use explicit casts when using types smaller than `int`

Language Support

- Java:
  - `Math.multiplyExact`, `Math.addExact`, etc.
- C/C++ compilers:
  - `-fwrapv`, `-ftrapv`
  - `-fsanitize`
- C#
  - `checked`