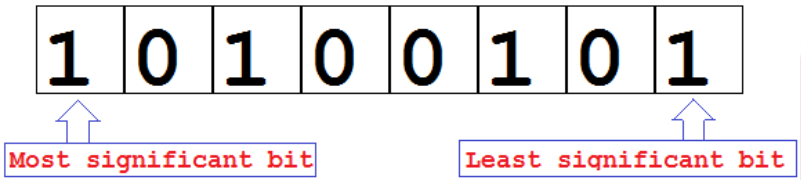
Bits & Bytes

Patricio Domingues

BINARY, OCTAL AND HEXADECIMAL BASES

The binary base

- Bit
  - Value of the binary base: `0` or `1`
- With N bits, we can have 2^N different states
  - Examples
    - 2 bits = 2^2: `00`, `01`, `10` and `11`
    - 3 bits = 2^3: `000`, `001`, `010`, `011`, `100`, `101`, `110`, `111`
  - 16 bits
    - 2^{16} distinct integer values
    - Unsigned: 0, 1, 2, …, 2^{16}-1 (65535)
    - Signed: -2^{15}(-32768)… 0 …2^{15}-1 (32767)



Octal base

- Octal
  - Numerical base which has 8 symbols: `0,1,2,3,4,5,6,7`
  - Conversion to decimal
    - Ex: $413_8 = 4 \times 8^2 + 1 \times 8^1 + 3 \times 8^0 = 4x64 + 1x8 + 3x1 = 267_{10}$
  - Conversion to binary
    - Ex: 413_8 = `100.001.011`
      - Each octal digit is maped to three bits, since we need 3 bits to represents 8 symbols
  - In C (and many other languages), a leading `0` means the number is in the octal base
    - Example: `0701`
  - In python 3.x, octals have leading `0o`

Octal	Binário
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Hexadecimal base

- Hexadecimal
  - Numerical base with 16 symbols:
    `0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F`
  - Conversion to decimal
    - Ex: $413_{16} = 4 \times 16^2 + 1 \times 16^1 + 3 \times 16^0 =$
      $4x256 + 1x16 + 3x1 = 1043_{10}$
  - Conversion to binary
    - Ex: $413_{16} =$ `0100.0001.0011`
      - Each hexadecimal digit is maped to **four** bits, since we need **four** bits to represents 16 symbols
  - In C (and many other languages), a leading `0x` means the number is in the hexadecimal base
    - Example: `0x413`

Hexadecimal	Binário
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Bits fields in C

- In C, we can define bit-field in structs
  - See exemple below

```
typedef struct example1{
        int field01:2;                    ➔ 2-bit wide
        unsigned int field02:4;           ➔ 4-bit wide
        float value_float;
}example1_t;

example1_t  exampleA;
exampleA.field01 = 1;
exemplеA.field02 = 0xA;
printf("field01=%d\n", exampleA.field01);
printf("field02=%d\n", exampleA.field02);
```

Example – *bit fields*

```c
#include <stdio.h>
// Define a structure with bit fields for RGB color
struct RGBColor {
    unsigned int red : 5; // 5 bits for the red component (0-31)
    unsigned int green : 5; // 5 bits for the green component (0-31)
    unsigned int blue : 5; // 5 bits for the blue component (0-31)
};
int main() {
    struct RGBColor pixel;
    // Assign RGB color values
    pixel.red = 15; // Red: 15 (01111 in binary)
    pixel.green = 31; // Green: 31 (11111 in binary)
    pixel.blue = 5; // Blue: 5 (00101 in binary)

    // Print the RGB color values
    printf("Red: %u\n", pixel.red);
    printf("Green: %u\n", pixel.green);
    printf("Blue: %u\n", pixel.blue);
    return 0;
}
```

https://pastebin.com/LeCDvZh7

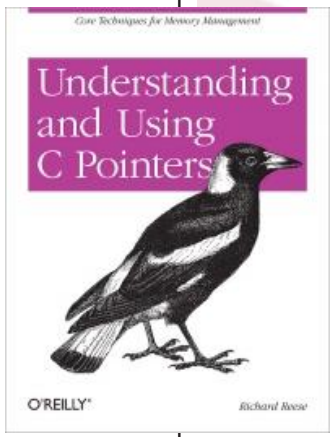
Finite size

- Computers are finite state machines
  - Memory is finite
    - Variables have finite length
      - We always need to be aware of the size of a variable
      - We always need to be aware of the signedness
  - Examples
    - unsigned char: 8 bits
      - An unsigned char can holds integer values between:
        - » 0 and 2^8-1 (i.e., 255)
    - signed char: 8 bits
      - A signed char can holds integer values between:
        - » -2^7 (-128) and 2^7-1 (+127)

Memory models (#1)

- Memory models
  - Size of I: integer; L:long; P:pointer
- Examples
  - ILP32: Integer=32 bits; Long=32 bits; Pointer=32 bits
  - ILP64: Integer=64 bits; Long=64 bits; Pointer=64 bits
  - LP64: Integer=32 bits; Long=64 bits; Pointer=64 bits

C Data Type	LP64	ILP64	LLP64	ILP32	LP32
char	8	8	8	8	8
short	16	16	16	16	16
_int32		32			
int	32	64	32	32	16
long	64	64	32	32	32
long long			64		
pointer	64	64	64	32	32



Memory models (#2)

- Size of datatype in C (depends on the ILP)

```
#include <stdio.h>
int main(void){
        printf("sizeof(char)=%lu bytes\n", sizeof(char));
        printf("sizeof(short)=%lu bytes\n", sizeof(short));
        printf("sizeof(int)=%lu bytes\n", sizeof(int));
        printf("sizeof(long)=%lu bytes\n", sizeof(long));
        printf("sizeof(long long int)=%lu bytes\n", sizeof(long long int));
        printf("sizeof(float)=%lu bytes\n", sizeof(float));
        printf("sizeof(double)=%lu bytes\n", sizeof(double));
        printf("sizeof(long double)=%lu bytes\n", sizeof(long double));
        printf("sizeof(char*)=%lu bytes\n", sizeof(char*));
        printf("sizeof(short*)=%lu bytes\n", sizeof(short*));
        printf("sizeof(long double*)=%lu bytes\n", sizeof(long double*));

        return 0;
}
```

Size of basic datatypes

- Compiled with gcc 5.4 in a 32-bit lubuntu 16.04
  - uname -a
    - Linux ubuntu 4.4.0-21-generic #37-Ubuntu SMP Mon Apr 18 18:34:49 UTC 2016 **i686 i686 i686** GNU/Linux

- `sizeof(char)=1 bytes`
- `sizeof(short)=2 bytes`
- `sizeof(int)=4 bytes`
- `sizeof(long)=4 bytes`
- `sizeof(long long int)=8 bytes`
- `sizeof(float)=4 bytes`
- `sizeof(double)=8 bytes`
- `sizeof(long double)=12 bytes`
- **`sizeof(char*)=4 bytes`**
- **`sizeof(short*)=4 bytes`**
- **`sizeof(long double*)=4 bytes`**

Size of basic datatypes

- Compiled with gcc 9.3 in a 64-bit lubuntu 20.04
  - uname -a
    - Linux so-vm 5.4.0-40-generic #44-Ubuntu SMP Tue Jun 23 00:01:04 UTC 2020 **x86_64 x86_64 x86_64** GNU/Linux
    - `sizeof(char)=1 bytes`
    - `sizeof(short)=2 bytes`
    - `sizeof(int)=4 bytes`
    - `sizeof(long)=8 bytes`
    - `sizeof(long long int)=8 bytes`
    - `sizeof(float)=4 bytes`
    - `sizeof(double)=8 bytes`
    - `sizeof(long double)=16 bytes`
    - **`sizeof(char*)=8 bytes`**
    - **`sizeof(short*)=8 bytes`**
    - **`sizeof(long double*)=8 bytes`**

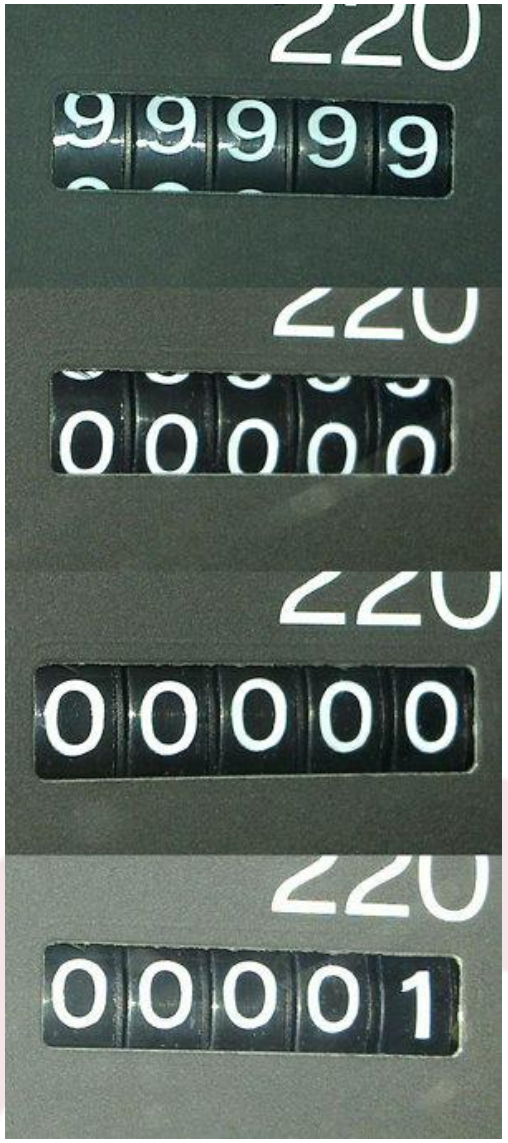
64-bit version

Addresses have 64 bits: 8 bytes

OVERFLOW AND UNDERFLOW

Overflow of integer variables (1)

- Integer variables have finite size
  - Overflow
  - This is similar to what happen to a (old) car odometer
    - 99999 kms ➔ 0 kms



http://i.imgur.com/deeV8.jpg

Overflow of integer variables (2)

- Integer variables have finite size: overflow

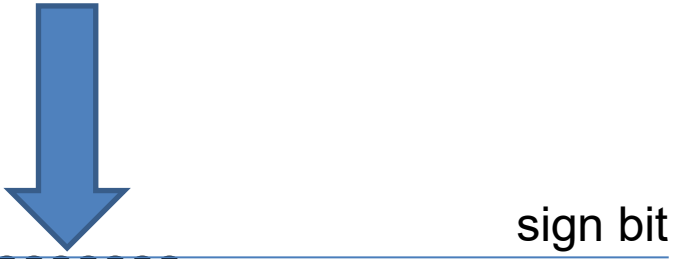
```
#include <stdio.h>
#include <limits.h>
int main(void){
        int Overflow = INT_MAX;  /* INT_MAX: valor máximo de um INT */
        printf("Overflow at max.:%d\n", Overflow);
        Overflow++;
        printf("Overflow beyond max.:%d\n", Overflow);
        return 0;
}
```

- Output of the program
  - Overflow at max: 2147483647
    - (2^31)-1
  - Overflow beyond max: -2147483648
    - -(2^31)

- The overflow changes the most significant bit (MSb), thus changing the bit signal to 1
- It goes from the maximum value (INT_MAX) to the lowest value (INT_MIN)

Overflow of integer variables (3)

- Output of the program
  - Overflow at max: 2147483647
    - $(2^{31})-1$
    - Binary: **01111111 11111111 11111111 11111111**
    - Hexadecimal: **0x7FFFFFFF**
  - Overflow beyond max: -2147483648
    - -2^{31}
    - Binary: **10000000 00000000 00000000 00000000**
    - Hexadecimal: **0x80000000**



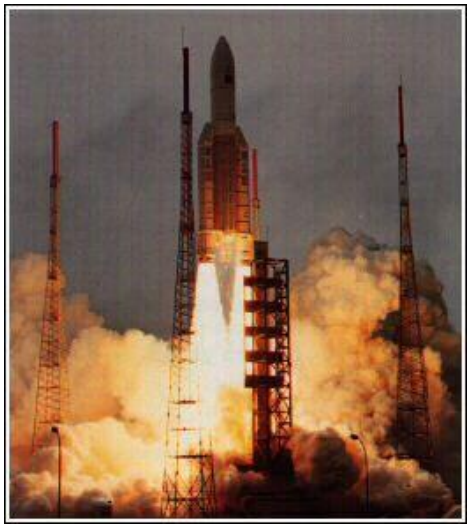
- The Most Significant bit (MSb) in a signed integer corresponds to the sign bit
  - 1: negative; 0 positive

- The overflow changes the most significant bit (MSb), thus changing the bit signal to 1
- It goes from the maximum value (INT_MAX) to the lowest value (INT_MIN)

OVERFLOW AND UNDERFLOW CASE-STUDIES (ARIANE 5, GRUB2,DEEP IMPACT, B787)

Overflow of integer variables - Ariane 5 (1)

- Maiden flight of spacecraft Ariane 5
  - https://www.youtube.com/watch?v=gp_D8r-2hwk
- *On June 4th, 1996, only 30 seconds after the launch, the Ariane 5 rocket began to disintegrate slowly and exploded.*
- *In the rocket's software (which came from Ariane 4), a 64-bit floating point variable with decimals called Horizontal Bias (BH) was transformed into a 16-bit signed integer.*
  - *Horizontal Bias is much higher in Ariane 5 than in Ariane 4 due to different trajectory at launch*
- *This variable, taking different sizes in memory, triggered a series of bugs that affected all the on-board computers and hardware, paralyzing the entire ship and triggering its self-destruct sequence.*



Overflow of integer variables - Ariane 5 (2)

- What did happen to Ariane 5?

source: “A Bug and a Crash” (https://pvs-studio.com/en/blog/posts/cpp/0426/)

- Guidance system's own computer tried to convert one piece of data -- the sideways velocity of the rocket, *Horizontal Bias* (BH) -- from a 64-bit floating-point format to a 16-bit integer signed format
- The number was too big for an 16-bit signed integer (>32767), and **an overflow error resulted**. The guidance system shutdown with an error code.
- The **error code was interpreted as valid data** from the inertial guidance system by the on-board computer, which thought that correction was needed
- The rocket made an abrupt course correction that was not needed, compensating for a wrong turn that had not taken place. Self-destruction was triggered automatically because aerodynamic forces were ripping the boosters from the rocket

```
L_M_BV_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_BV) *
                               G_M_INFO_DERIVE(T_ALG.E_BV));
if L_M_BV_32 > 32767 then
   P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
   P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
   P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M
end if;

P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                               ((1.0/C_M_LSB_BH) *
                               G_M_INFO_DERIVE(T_ALG.E_BH)))
end LIRE_DERIVE;
```

ADA code of Ariane 5 highligthing the E_BH variable

Fonte:
https://import.viva64.com/docx/blog/0426_Space_error/image15.png

Underflow in grub2 (#1)

- Bug found in December 2015
  - It existed since 2009
- *Underflow* in the function `grub_username_get`

```
static int grub_username_get (char buf[], unsigned buf_size){
  unsigned cur_len = 0;
  int key;
  while (1){
      key = grub_getkey ();
      if (key == '\n' || key == '\r')
        break;
      if (key == '\e'){
          cur_len = 0;
          break;
        }
```

(continue)

- Info: http://hmarco.org/bugs/CVE-2015-8370-Grub2-authentication-bypass.html

Underflow in grub2 (#2)

- *Underflow* in `grub_username_get` **(continued)**

```
      if (key == '\b') {  // Does not checks underflows !!
         cur_len--; // Integer underflow !!
         grub_printf ("\b");
         continue;
       }
     if (!grub_isprint (key))
       continue;
     if (cur_len + 2 < buf_size){
         buf[cur_len++] = key; // Off-by-two !!
         grub_printf ("%c", key);
       }
   }
 // Out of bounds overwrite
 grub_memset( buf + cur_len, 0, buf_size - cur_len);
 grub_xputs ("\n");
 grub_refresh ();
 return (key != '\e');
}
```

Correction: if (key == '\b' **&& cur_len**)

- Info: http://hmarco.org/bugs/CVE-2015-8370-Grub2-authentication-bypass.html

Unix time_t

- Time origin ("zero time") is
  - 1 jan 1970 00:00 GMT ("UNIX EPOCH")
- The datatype `time_t` is used to hold a `signed integer` value to represent time
  - Number of seconds elapsed since EPOCH
  - Example: `time_t now = time(NULL);`
- In system that uses a **32-bit signed integer**, overflow will occur when time > 2^{31}-1
  - 19 jan 2038 03:14:07 GMT
  - *UNIX Year 2038 problem*

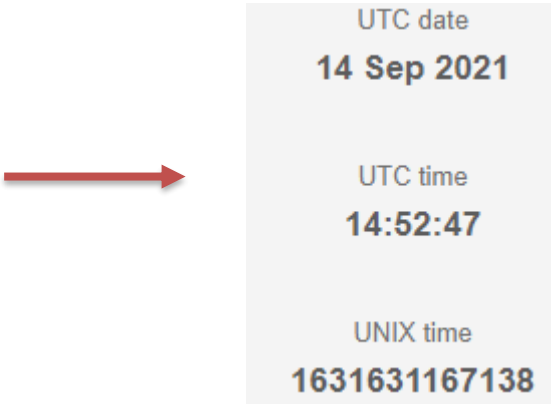
Binary : 01111111 11111111 11111111 11110000
Decimal : 2147483632
Date : 2038-01-19 03:13:52 (UTC)
Date : 2038-01-19 03:13:52 (UTC)

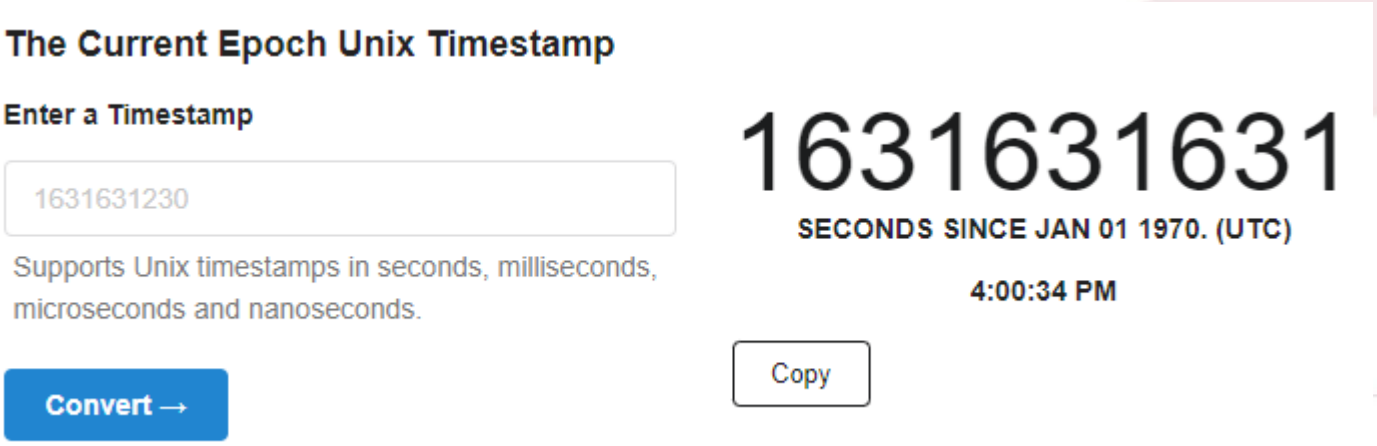
Unix time_t

- New variants of Unix time
  - EPOCH is still Jan 1st 1970 00:00:00 UTC
- 64 bits integer
  - Number of milliseconds since EPOCH

  OR

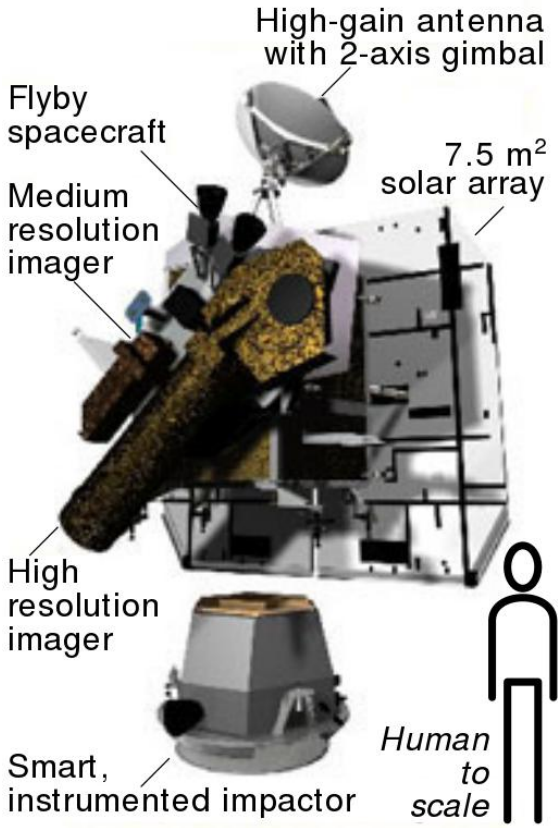
  - Number of microseconds since EPOCH



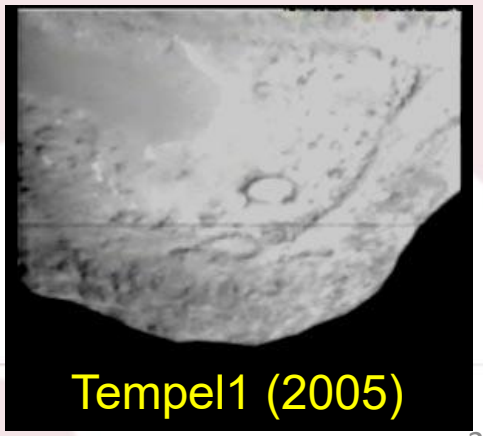


Deep Impact mission *bug*

- Deep Impact mission
  - Send a probe to comet Tempel1
    - One part of the probe is an impactor (4th July 2005)
  - Mission extended to:
    - flyby (700 kms) comet Hartley2 (2010)
    - Observe comet Garradd (2012)
    - Observe comet C/2012 (2013)
  - But…communication lost on 11th August 2013
  - Overflow of time variable
    - EPOCH for Deep Impact was 1 Jan 2000, 00:00 GMT
      - at August 11, 2013, 00:38:49, it was 2^{32} of one-tenth seconds from January 1, 2000
    - *Deep Impact's chief mission scientist Mike A'Hearn said, "Basically, it was a Y2K problem, where some software didn't roll over the calendar date correctly." The fault-protection software misread any dates after 2013-08-11, and the misreads triggered an endless series of computer reboots.*



http://bit.ly/2gsQra0



B787 – Overflow

- «This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode. **This condition is caused by a software counter internal to the GCUs that will overflow after 248 days of continuous power.** We are issuing this AD to prevent loss of all AC electrical power, which could result in loss of control of the airplane» (2015)
  - Source: https://bit.ly/2o4CPtj
- 2^{31} in 1/100 seconds =~ 248,5 days
- Integer overflow of 31-bit value

HP SSD firmware – overflow?

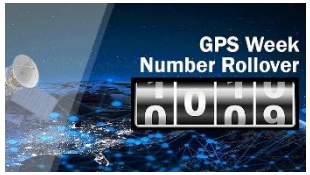
- 2019.11 - HPE SAS Solid State Drives
  - "Critical Firmware Upgrade Required for Certain HPE SAS Solid State Drive Models to Prevent Drive Failure at 32,768 Hours of Operation"
    - 3 years, 270 days 8 hours
  - Problem in firmware
    - 32768 = 2^15, that is, **INT16MAX+1**
    - Someone used a (signed) `int16_t` variable for counting hours…
- Drive needs a firmware update (HPD8) before reaching 32768 hours of operations
  - After 32768 hours, it becomes bricked (supposedly)

Hardware Platforms Affected: HPE Synergy 480 Gen9 Compute Module, HPE Synergy 660 Gen9 Compute Module, HPE Synergy D3940 Storage Module, HPE 400GB 12G SAS Mixed Use-3 SFF 2.5-in SC 3yr Wty MO0400JFFCF Solid State Drive, HPE 800GB 12G SAS Mixed Use-3 SFF 2.5-in SC 3yr Wty MO0800JFFCH Solid State Drive, HPE 1.6TB 12G SAS Mixed Use-3 SFF 2.5-in SC 3yr Wty MO1600JFFCK Solid State Drive, HPE 3.2TB 12G SAS Mixed Use-3 SFF 2.5-in SC 3yr Wty MO3200JFFCL Solid State Drive, HPE 480GB 12G SAS Read Intensive-3 SFF 2.5-in SC 3yr Wty VO0480JFDGT Solid State Drive, HPE 960GB 12G SAS Read Intensive-3 SFF 2.5-in SC 3yr Wty VO0960JFDGU Solid State Drive, HPE 3.84TB 12G SAS Read Intensive-3 SFF 2.5-in SC 3yr Wty VO3840JFDHA Solid State Drive, HPE Synergy 620 Gen9 Compute Module, HPE Synergy 680 Gen9 Compute Module, HPE ProLiant XL270d Gen9 Server, HPE D6020 Disk Enclosure, HPE StoreVirtual 3000 Storage, HPE ProLiant DL360 Gen10 Server, HPE ProLiant BL460c Gen10 Server Blade, HPE ProLiant DL380 Gen10 Server, HPE ProLiant DL388 Gen10 Server, HPE ProLiant DL160 Gen10 Server, HPE ProLiant DL180 Gen10 Server, HPE ProLiant DL580 Gen10 Server, HPE ProLiant ML110 Gen10 Server, HPE ProLiant ML350 Gen10 Server, HPE ProLiant DL385 Gen10 Server, HPE ProLiant DL325 Gen10 Server, HPE ProLiant DL20 Gen10 Server, HPE ProLiant SL230s Gen8 Server, HPE ProLiant BL460c Gen8 Server Blade, HPE ProLiant BL465c Gen8 Server Blade, HPE ProLiant DL160 Gen8 Server, HPE ProLiant BL420c Gen8 Server Blade, HPE ProLiant DL320e Gen8 Server, HPE ProLiant WS460c Gen8 Graphics Server Blade, HPE ProLiant BL660c Gen8 Server Blade, HPE ProLiant DL560 Gen8 Server, HPE D6000 Disk Enclosure, HPE StoreEasy 1000 Storage, HPE D2220sb Storage Blade, HPE ProLiant SL210t Gen8 Server, HPE StoreVirtual 4335 Hybrid Storage, HPE ProLiant DL580 Gen8 Server, HPE D3000 Disk Enclosures, HPE ProLiant DL180 Gen9 Server, HPE ProLiant DL360 Gen9 Server, HPE ProLiant BL460c Gen9 Server Blade, HPE ProLiant DL380 Gen9 Server, HPE ProLiant ML350 Gen9 Server, HPE ProLiant XL230a Gen9 Server, HPE ProLiant DL388 Gen9 Server, HPE ProLiant DL120 Gen9 Server, HPE ProLiant WS460c Gen9 Graphics Server Blade, HPE ProLiant DL580 Gen9 Server, HPE ProLiant BL660c Gen9 Server Blade, HPE ProLiant DL560 Gen9 Server, HPE Apollo 4200 Gen9 Server, HPE Apollo 4500 System, HPE ProLiant XL450 Gen9 Server

https://bit.ly/35loKct

"Critical: Drive failure occurs after 32768 hours of use"

GPS rollover week



- GPS systems have a rollover every 1024 weeks
  - The week number is kept in a 10-bit register
  - $1024 = 2^{10}$
  - So every 1024 weeks, there is a GPS rollover
    - 1024 weeks = 19.7 years
- GPS is not only used for positioning, but also for time accuracy
  - Last occurrence 6/7 April 2019 (2^{nd} rollover occurrence)
- Some effects of the 2019 rollover
  - Honeywell flight management/navigation software caused a KLM flight to be cancelled because technician failed to patch
  - Some GPS navigation devices
- The new protocol CNAV uses 13-bit week identifier
  - $2^{13} = 8192$ weeks = 157.5 years

THE HEADER FILE <LIMITS.H>

The <limits.h> file

- The C programming language has constants for the max. and min. values of integer datatypes
  - File <limits.h>

```
#define CHAR_BIT      8
/* Minimum and maximum values a `signed char' can hold.  */
#define SCHAR_MIN     (-128)
#define SCHAR_MAX     127
/* Maximum value an `unsigned char' can hold.  (Minimum is 0.)  */
#define UCHAR_MAX     255
/* Minimum and maximum values a `signed short int' can hold.  */
#define SHRT_MIN      (-32768)
#define SHRT_MAX      32767
/* Maximum value an `unsigned short int' can hold.  (Minimum is 0.)  */
#define USHRT_MAX     65535
/* Minimum and maximum values a `signed int' can hold.  */
#define INT_MIN       (-INT_MAX - 1)
#define INT_MAX       2147483647
/* Maximum value an `unsigned int' can hold.  (Minimum is 0.)  */
#define UINT_MAX      4294967295U
(…)
```

BINARY OPERATORS IN C

Binary operators in C

- Binary operators
  - NOT: ~
  - AND: &
  - OR: |
  - XOR: ^
  - left shift: <<
  - right shift: >>
- Binary operations are efficiently executed by CPUs
  - Direct support through dedicated CPU instructions

Binary *NOT* ~ operator

- NOT operator
  - Symbol: ~
  - Unary operator
    - It only has one operand
  - It negates each bit
- Example
  - A= 01010001_2
  - B = ~A
  - B ← 10101110_2

```
#include <stdio.h>
 int main(void){
    unsigned int in = 0x01234567;
    unsigned int out;
    out = ~in;
    printf("in: %08x\n", in);
    printf("out: %08x\n", out);
    return 0;
}
== OUTPUT ==
in: 01234567 (0000.0001.0010.0011.0101.0110.0111)
out:fedcba98 (1111.1110.1101.1100.1010.1001.1000)
```

Binary *AND* & (#1)

- AND operator
  - Symbol: &
  - Binary operator
    - a & b
- Example

Binary AND (&)	0	1
0	0	0
1	0	1

```
int main(void){
    int a = 0x12; /* 0001.0010b, 18 base 10 */
    int b = 0x0F; /* 0000.1111b, 15 base 10 */
    int c;
    c = a & b;  /* binary AND */
    /* 0001.0010 & 0000.1111 => 0000.0010 */
    printf("c = %d & %d => %x\n", a, b, c);
    return 0;
}
```

Binary *AND* & (#2)

- Do not confuse ***binary and*** with ***logical and***
  - ***binary and: &***
  - ***logical and: &&***
- **Logical and** is used with logical conditions

```
if((a==20) && (b==10)){…
…}
if(a==20) & (b==10)){…
}
```

Binary AND (&)	0	1
0	0	0
1	0	1

Logical AND (&&)	False	True
False	False	False
True	False	True

Logical AND &&

- Logical AND - example

```
#include <stdio.h>
int main(void){
    int a = 0;
    int b = 2;
    int result;
    /* true */
    result = ((a==0) && (b==2));
    printf("TRUE => %d\n", result);
    /* false */
    result = ((a==0) && (b==3));
    printf("FALSE => %d\n", result);
    return 0;
}
== OUTPUT ==
TRUE => 1
FALSE => 0
```

Binary *OR* | (#1)

- OR operator
  - Symbol: |
  - Binary operator
    - a | b

Binary OR (|)	0	1
0	0	1
1	1	1

- Example

```
int main(void){
  int a = 0x003;/* 0000.0000.0011b, 3 base10 */
  int b = 0x120;/* 0001.0010.0000b, 288 base10 */
  int c;
  c = a | b; /* binary OR */
  /* 0000.0000.0011 | 0001.0010.0000
          => 0001.0010.0011 */
  printf("c = %d | %d => %d\n", a, b, c);
  return 0;
}
```

== output ==

c = 3 | 288 => 291

Binary *OR* | (#2)

- Do not confuse ***binary OR*** with ***logical OR***
  - ***binary or: |***
  - ***logical OR: ||***
- **Logical and** is used with logical conditions

```
if((a==20) || (b==10)){…
…}
if(a==20) | (b==10)){…
}
```

Binary OR (|)	0	1
0	0	1
1	1	1

Logical OR (||)	False	True
False	False	True
True	True	True

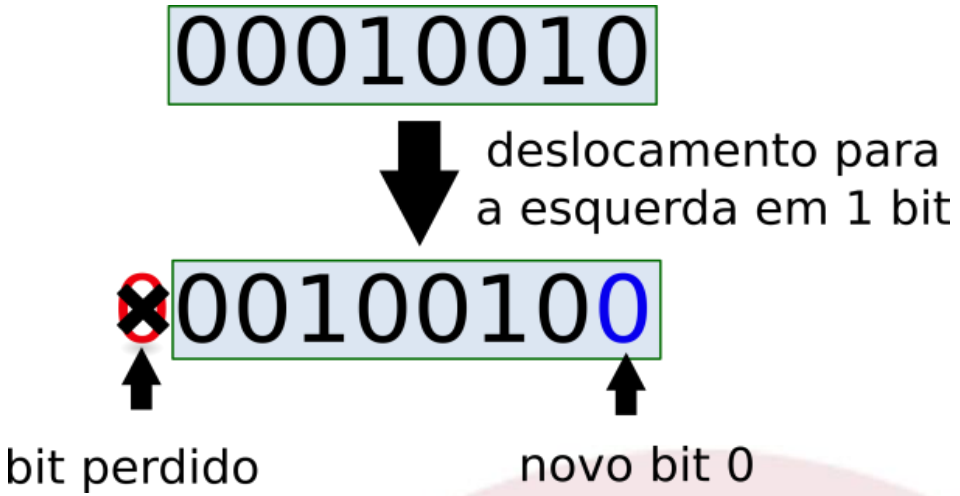
XOR (eXclusive OR)

- XOR operator
  - Symbol: ^
  - Binary operator
    - a ^ b
- Question
  - What's the result?
    - `int c = c^c;`

XOR (^)	0	1
0	0	1
1	1	0

Left shift << (#1)

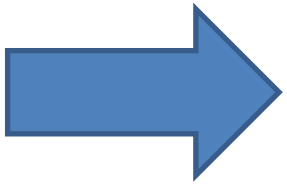
- Left shift operator
  - Symbol: <<
  - Binary operator
    - value << N
    - N is the number of left-shifted bits
    - If N >= sizeof(type)*8
      - Undefined behavior
- Example
  - A = 2;
  - A << 3? ➔ 16
  - X << N
    - multiply X by 2^N



Left shift << (#2)

- Example
  - Left shift is a fast way to multiply by 2
  - But, watch out for overflow!

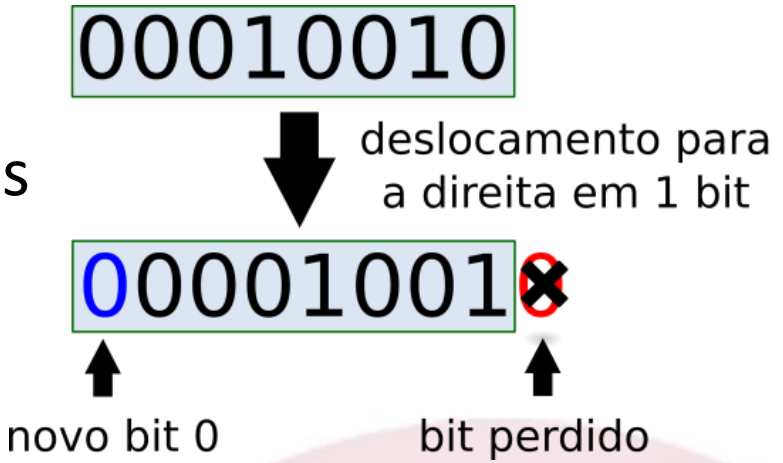
```
#include <stdio.h>
int main(void){
    unsigned int value = 1;
    unsigned int value_shift;
    size_t size_bits=sizeof(value)*8;
    unsigned int i;
    for(i=0;i<size_bits;i++){
        valor_shift = value << i;
        printf("[shift (value << %02u)]%u\n",
                i, value_shift);
    }
    return 0;
}
```



```
[shift (value << 00)]1
[shift (value << 01)]2
[shift (value << 02)]4
[shift (value << 03)]8
[shift (value << 04)]16
(…)
[shift (value << 29)]536870912
[shift (value << 30)]1073741824
[shift (value << 31)]2147483648
```

Right shift >> (#1)

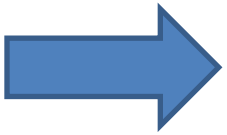
- Right shift operator
  - Symbol: >>
  - Binary operator
    - value >> N
    - N is the number of right-shifted bits
- Example
  - A = 32;
  - A >> 3? ➔ 4
    - A / 2^3 = A / 8



Right shift >> (#2)

- Example

```
int main(void){
    int positive = 998;
    unsigned int sem_sinal = 998;
    int positive_shift_R;
    unsigned int sem_sinal_shift_R;
    int i;
    for(i=0; i < 4; i++){
        positive_shift_R = positive >> i;
        printf("===[i=%d]===\n", i);
        printf("positive_shift_R=%d\n",
                positive_shift_R);
        sem_sinal_shift_R = sem_sinal >> i;
        printf("sem_sinal_shift_R=%d\n",
            sem_sinal_shift_R);
    }
    return 0;
}
```



```
===[i=0]===
  positive_shift_R=998
  sem_sinal_shift_R=998
===[i=1]===
  positive_shift_R=499
  sem_sinal_shift_R=499
===[i=2]===
  positive_shift_R=249
  sem_sinal_shift_R=249
===[i=3]===
  positive_shift_R=124
  sem_sinal_shift_R=124
```

Right shift >> (#3)

- The behaviour of a right shift operation on a **signed integer** is system dependent
  - The C language **doesn't state** which bit (0 ou 1) should be inserted as the most significant bit on a right shift operation
  - Don't use >> with negative numbers …

```
#include <stdio.h>
int main(void){
     int positive = 998;
     int negative = -998;
     int positive_shift, negative_shift;
     int i;
     for(i=0; i < 4; i++){
          printf("===[shift right %d]===\n",i);
          positive_shift = positive >> I;
          negative_shift = negative >> I;
     printf("positive_shift=%d\n",positive_shift);
     printf("negative_shift=%d\n",negative_shift);
     }
     return 0;
}
```

```
===[shift right 0]===
positive_shift=998
negative_shift=-998
===[shift right 1]===
positive_shift=499
negative_shift=-499
===[shift right 2]===
positive_shift=249
negative_shift=-250
===[shift right 3]===
positive_shift=124
negative_shift=-125
```

-998/4 = -249.5
The programs returns -250 instead of -249

Right shift in JAVA

- Java
  - Besides the >> right shift, JAVA has another right shift operator
    - >>>
    - It is called “right shift with no sign extension”
    - It always inserts a 0 as the most significant bit

Precedence	Operator	Operand type	Description
1	++,	Arithmetic	Increment and decrement
1	+, -	Arithmetic	Unary plus and minus
1	~	Integral	Bitwise complement
1	!	Boolean	Logical complement
1	(*type*)	Any	Cast
2	*, /, %	Arithmetic	Multiplication, division, remainder
3	+, -	Arithmetic	Addition and subtraction
3	+	String	String concatenation
4	<<	Integral	Left shift
4	>>	Integral	Right shift with sign extension
4	>>>	Integral	Right shift with no extension
5	<, <=, >, >=	Arithmetic	Numeric comparison
5	instanceof	Object	Type comparison
6	==, !=	Primitive	Equality and inequality of value
6	==, !=	Object	Equality and inequality of reference
7	&	Integral	Bitwise AND
7	&	Boolean	Boolean AND
8	^	Integral	Bitwise XOR
8	^	Boolean	Boolean XOR
9	|	Integral	Bitwise OR
9	|	Boolean	Boolean OR
10	&&	Boolean	Conditional AND
11	||	Boolean	Conditional OR
12	?:	N/A	Conditional ternary operator
13	=	Any	Assignment

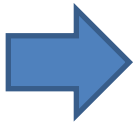
http://www.w3processing.com/java/images/operators.png

Usage of bitwise operators (#1)

- Shift operator
  - Left shift
    - Build a binary mask with a single bit set to 1
    - `A = 0x1 << 3` ➔ `00...001000`
    - `A = 0x1 << 5` ➔ `00..0100000`
    - A integer number which has only one bit set to 1 is a power of 2
      - E.g.: `0010` ➔ `2; 010000` ➔ `16`
  - We can invert a mask with the NOT operator
    - `B = ~A` ➔ `11..1011111`

Usage of bitwise operators (#2)

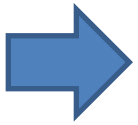
- How to extract the value of a given bit?
  - Is it 0 ou 1?
- Example
  - `int A= some_value;`
  - What's the value of the 3rd bit of A?
    - use a mask and the AND operator
    - `mask = 0…0100`



```
int A = …; /* What's the 3rd bit? */
int mask_3rd_bit = 0x1 << 2;
int value_3rd_bit = A & mask_3rd_bit;
if( value_3rd_bit ){
    /* 3rd bit is 1 */
}else{
    /* 3rd bit is 0 */
}
```

Usage of bitwise operators (#3)

- How to set to 1 the value of a given bit?
- Example
  - Set to 1 the value of the 4th bit, without changing the other bits
  - `int A=some_value;`
    - use a mask and the OR operator
    - `mask = 0…01000`



```
int A = …; /* set 4th bit to 1 */
int mask_4rd_bit = 0x1 << 3;
int A_4th_bit1 = A | mask_4rd_bit;
```

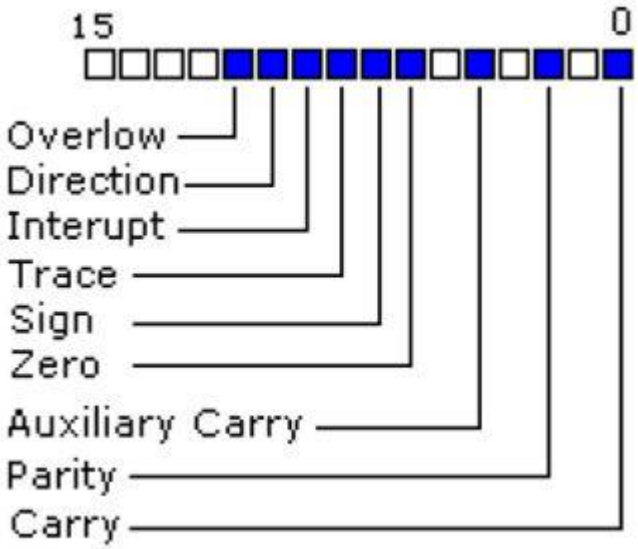
Bit recipes

- Check if the n^{th} bit is set:
  - `(flags & (1 << n)) != 0`
- Set the n^{th} bit:
  - `flags |= (1 << n)`
- Clear the n^{th} bit:
  - `flags &= ~(1 << n)`
- Flip the n^{th} bit:
  - `flags ^= (1<<n)`

FLAGS IN PROGRAMMING

Binary flags

- In informatics, a flag represents an ON/OFF value
  - It can be represented by a single bit
    - 0 – OFF
    - 1 – ON
- In a 16-bit (short) integer we can have…16 different flags
  - We need to use | to activate bits / flags
  - We need to use & to extract bits/flags



http://bit.ly/2ecWf7y

Flags in the C library (#1)

- Several functions and structs in C have a “flags” parameter
  - int shmget(key_t key, size_t size, int **shmflg**);
  - int mkostemp(char *template, int **flags**);
  - int open(const char *pathname, int **flags**, mode_t mode);
- The flag parameter is set by OR-ing some preprocessor constants
  - Example

```
char *filename;
int fd;
 do {
   filename = tempnam (NULL, "foo");
   fd = open (filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWR, 0600);
   free (filename);
 } while (fd == -1);
```

Continue >>

Flags in the C library (#2)

- Example

```
(…)
   fd = open (filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWR, 0600);
(…)
```

Binary OR

- What are the values of O_CREAT, O_EXCL, …?
  - They are power of 2
    - Only one bit is 1
      - O_RDWR=2 ➔ 2nd bit is O_RDWR flag
      - O_CREAT=64➔ 7th bit is O_CREAT flag
      - O_EXCL=128 ➔ 8th bit is O_EXCL flag
      - O_TRUNC=512➔ 10th bit is O_TRUNC flag

```c
#include <stdio.h>
#include <fcntl.h>
int main(void){
    printf("O_RDWR=%d\n", O_RDWR);
    printf("O_CREAT=%d\n", O_CREAT);
    printf("O_EXCL=%d\n", O_EXCL);
    printf("O_TRUNC=%d\n", O_TRUNC);
    return 0;
}
```

  - Question
    - How to test if the variable holding the flags has a given flag?
      - Use AND operator and the flag: variable & FLAG

C23’S NEW BITWISE FUNCTIONS

C23 standard (#1)

- The `_BitInt(x)` datatype
  - Bit-precise integer
  - New type introduced in the C23 standard
  - Use Two's complement representation
  - Represents an integer with a specific bit width.
    - X can be within the range [1, BITINT_MAXWIDTH]
  - Examples
    - _BitInt(8) A;
    - unsigned _BitInt(12) B;
    - _BitInt(32) C;
    - unsigned _BitInt(64) D;
- Note: gcc14 or plus is required for C23

C23 standard (#2)

- C23 supports *binary constants* and *digital separator*
  - Prefix is **`0b`** for binary constants
    - GCC, CLANG and other compilers already supports binary constants
  - Allows for a digit separator (see below)
    - The digit separator can be used with other numerical bases (octal, decimal, hexadecimal)
  - Examples

```
// Binary with digit separators
_BitInt(16) precise_bits = 0b1100'1010'1111'0000;
// Decimal numbers
long distance = 384'400'000;
// Hexadecimal
unsigned long color = 0xFF'FF'FF'FF;
```

C23 standard (#3)

- Some new bit utility functions in the new header **<stdbit.h>**
  - All start with stdc_ to minimize conflict with legacy code
  - stdc_count_ones*: **returns the number of bits in value set to 1**
    - int stdc_count_onesuc(unsigned char value);
    - int stdc_count_onesus(unsigned short value);
    - int stdc_count_onesui(unsigned int value);
    - int stdc_count_onesul(unsigned long value);
    - int stdc_count_onesull(unsigned long long value);
    - generic_return_type stdc_count_ones(generic_value_type value)
  - `stdc_count_zero*:` returns the number of bits set to 0
  - Functions resort to the CPU's "POPCNT" instruction
    - "Population Count": computes the number of bits=1 in a given number

Generic macro

Example – C23

- int stdc_count_onesuc(unsigned char value);

```c
#include <stdio.h>
#include <stdbit.h>
int main(void) {
    unsigned char value = 0b10101010;  // 170 decimal
    // Count the number of 1-bits (population count)
    int ones_count = stdc_count_ones_uc(value);
    printf("Value: 0x%02X (0b10101010)\n", value);
    printf("Number of 1-bits: %d\n", ones_count);  // Output: 4
    return 0;
}
```

```
Value: 0xAA (0b10101010)
Number of 1-bits: 4
```

https://pastebin.com/NcY9RNNR

C23 standard (#3)

- The **stdc_bit_width** functions compute the smallest number of bits needed to store **value**
- The **stdc_bit_width** functions return 0 if value is 0. Otherwise, they return 1 + ⌊log2(value)⌋.
  - #include <stdbit.h>
  - unsigned int stdc_bit_width_uc(unsigned char value);
  - unsigned int stdc_bit_width_us(unsigned short value);
  - unsigned int stdc_bit_width_ui(unsigned int value);
  - unsigned int stdc_bit_width_ul(unsigned long value);
  - unsigned int stdc_bit_width_ull(unsigned long long value);
  - /* generic_return_type stdc_bit_width(generic_value_type value); */

Generic macro

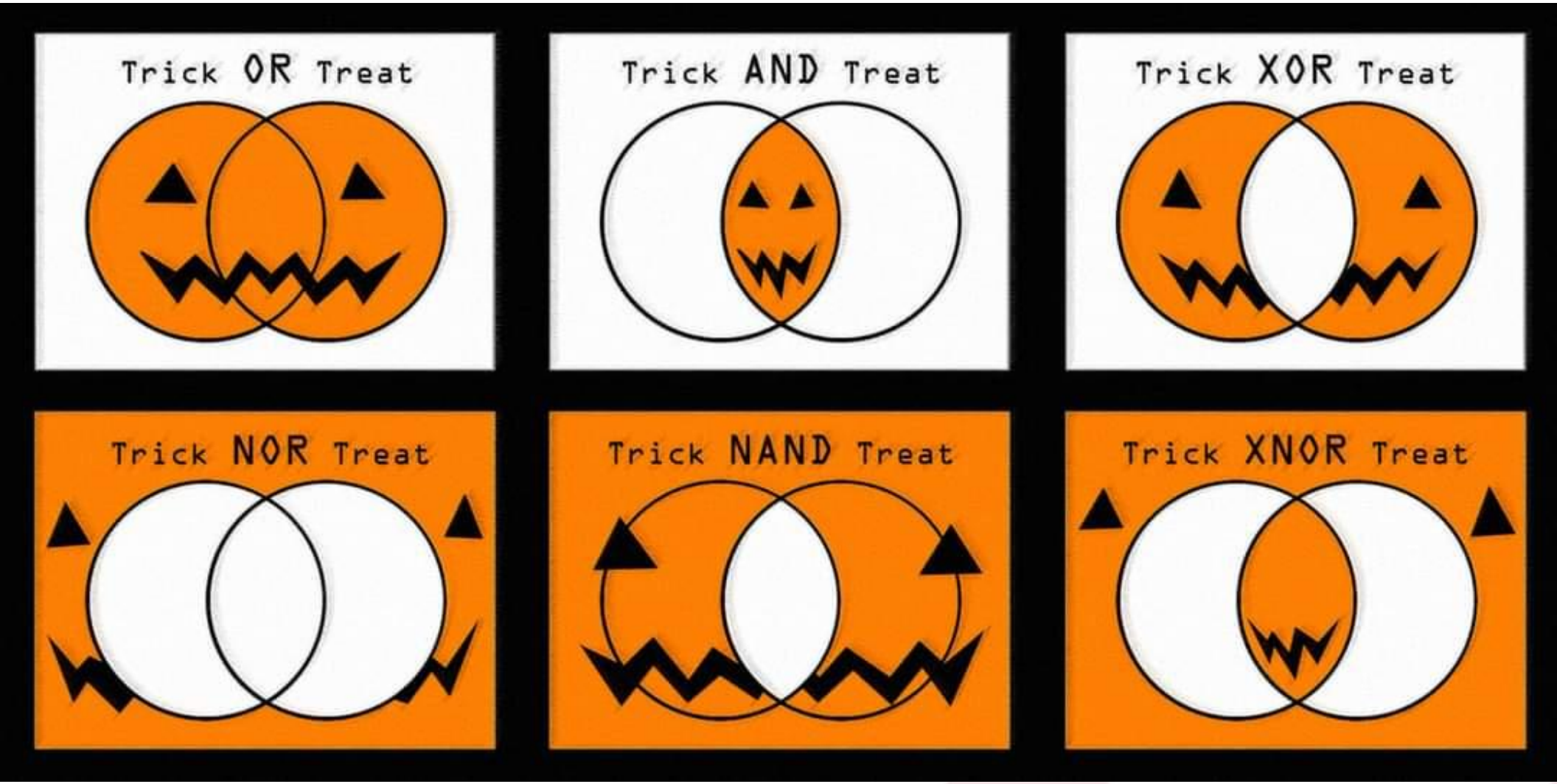
Example

- unsigned int stdc_bit_width_ui(unsigned int value);

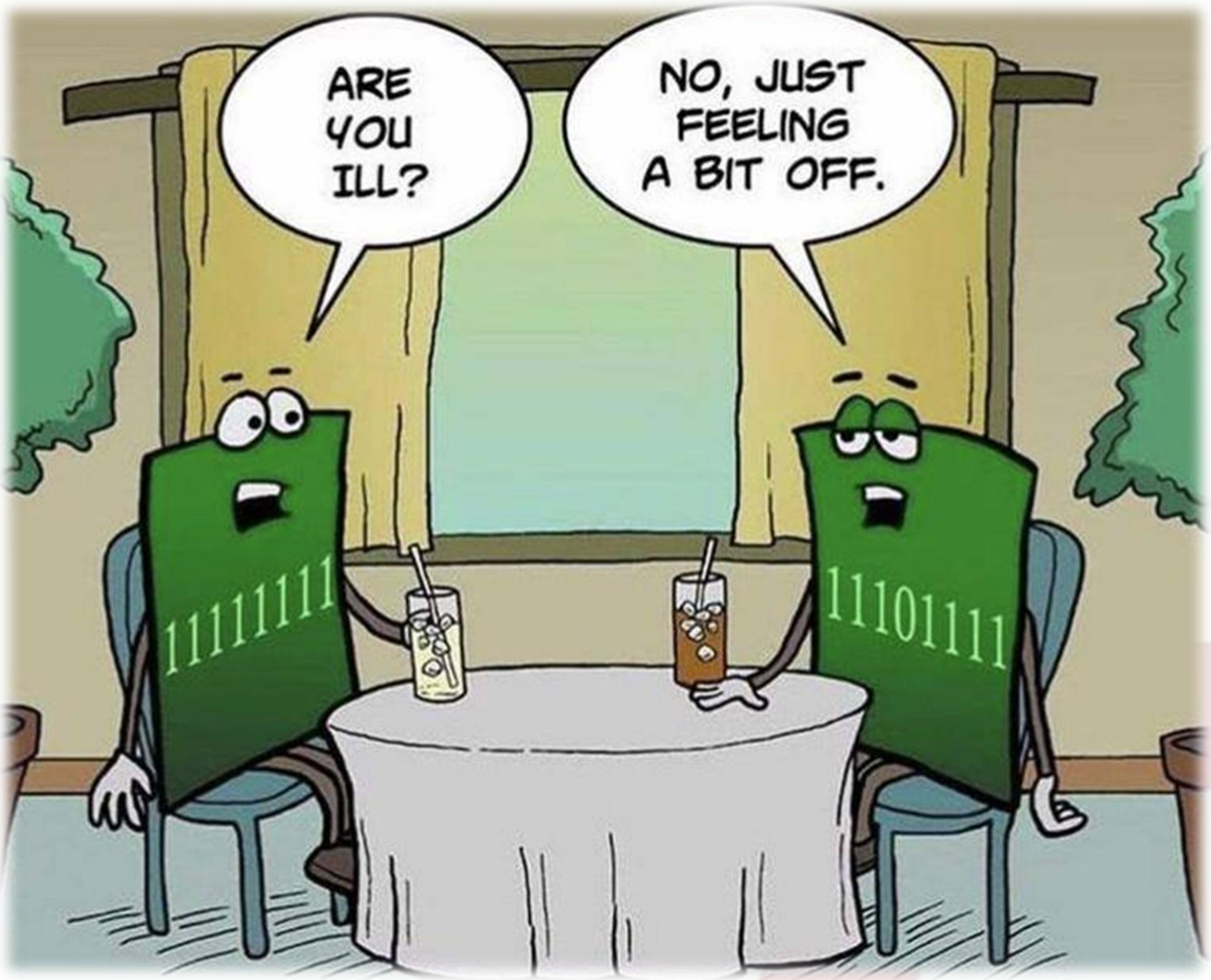
```c
#include <stdio.h>
#include <stdbit.h>
int main(void) {
    printf("Value 0:  %u bits needed\n", stdc_bit_width_ui(0));     // 0
    printf("Value 1:  %u bits needed\n", stdc_bit_width_ui(1));     // 1
    printf("Value 15: %u bits needed\n", stdc_bit_width_ui(15));    // 4
    printf("Value 16: %u bits needed\n", stdc_bit_width_ui(16));    // 5
    printf("Value 255: %u bits needed\n", stdc_bit_width_ui(255));  // 8
    printf("Value 256: %u bits needed\n", stdc_bit_width_ui(256));  // 9
    return 0;
}
```

```
Value 0:  0 bits needed
Value 1:  1 bits needed
Value 15: 4 bits needed
Value 16: 5 bits needed
Value 255: 8 bits needed
Value 256: 9 bits needed
```

https://pastebin.com/dHyvn89f



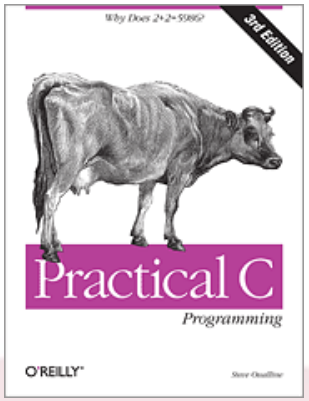
https://twitter.com/DarkLiterata/status/1583562256788631552/photo/1


ARE YOU ILL?
NO, JUST FEELING A BIT OFF.
11111111
11101111

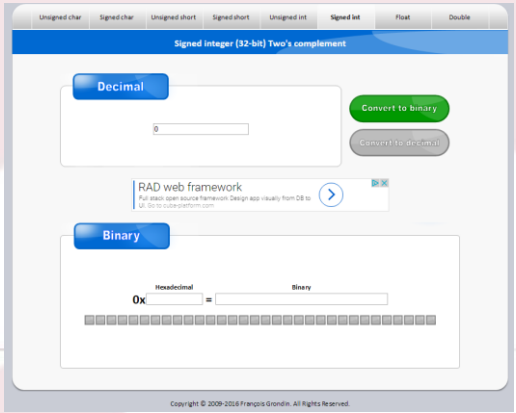
Bibliography

- Patrício Domingues. “Manipulação ao nível do bit na Linguagem C”. Revista Programar, Número 50, pp. 26-35, Setembro 2015, ISSN 1647 0710. http://bit.ly/2dmD74H

- Steve Oualline, “Practical C Programming – Chapter 11: Bit Operations”, O'Reilly Media, Inc.", 1997, ISBN: 1-56592-306-5



- Online resources
  - Converter: decimal & datatypes http://www.binaryconvert.com/convert_signed_int.html



Vim Escape :q!