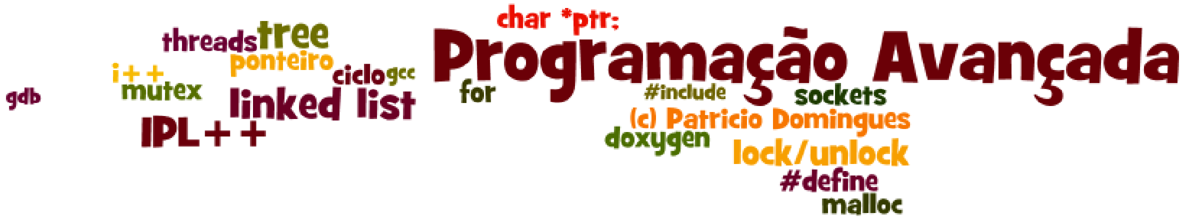
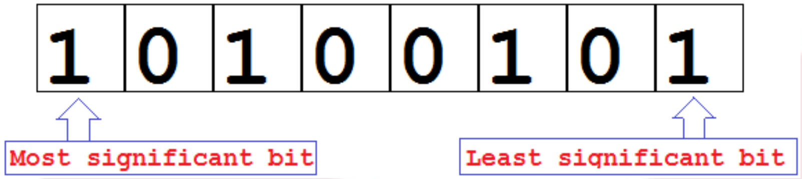
Bits & Bytes



Patricio Domingues

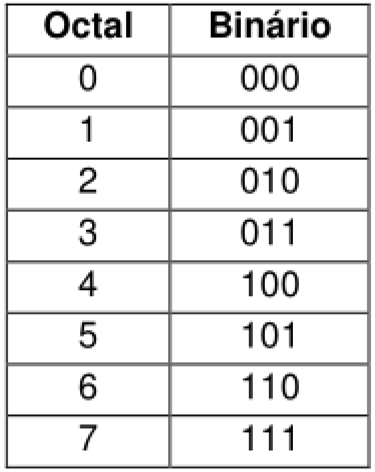
The binary base

- 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

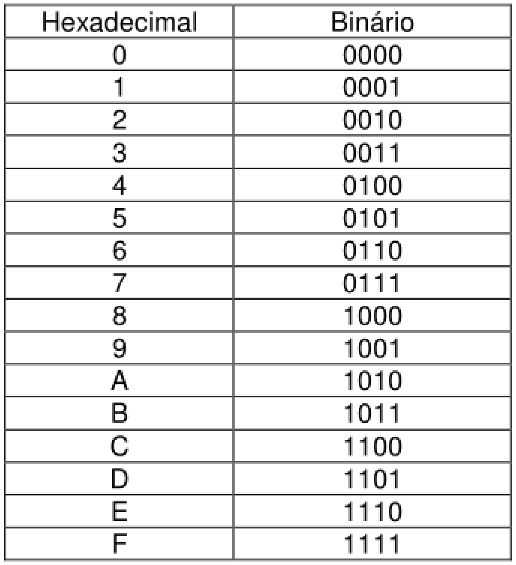
- Octal
  - Numerical base which has 8 symbols: `0,1,2,3,4,5,6,7`
  - Conversion to decimal
    - Ex: $413_8 = 4 \text{ x } 8^2 + 1 \text{ x } 8^1 + 3 \text{ x } 8^0 = 4\text{x}64 \; +1\text{x}8 \; +3\text{x}1=267_{10}$
  - Conversion to binary
    - Ex: 413_8 = `100.001.011`
      - Each octal digit 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: $413_{16} = 4 \times 16^{2} + 1 \times 16^{1} + 3 \times 16^{0} =$ $4\text{x}256 + 1\text{x}16 + 3\text{x}1 = 1043_{10}$
  - 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

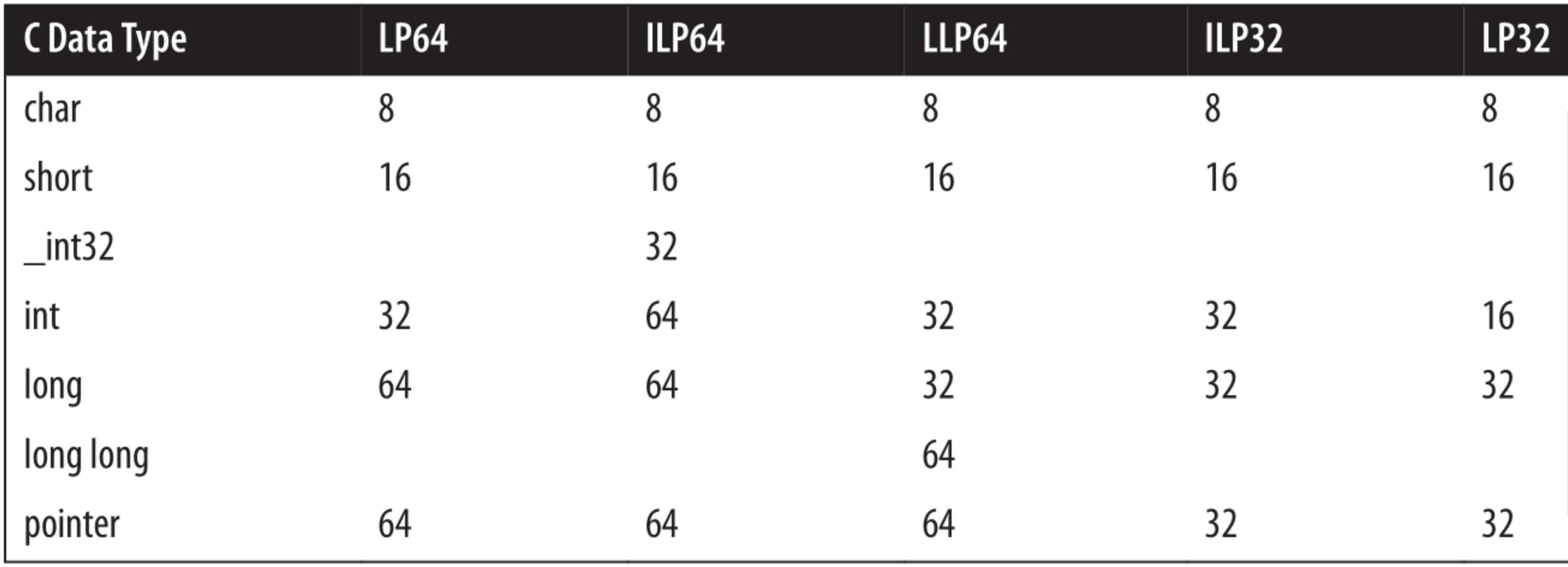
```
typedef struct exemplo1{
        int field01:2;                          ➔ 2-bit wide
        unsigned int field02:4;                 ➔ 4-bit wide
        float value_float;
}example1_t;

example1_t  example1;
example1.field01 = 1;
exemple1.field02 = 0xA;
printf("field01=%d\n", example1.field01);
printf("field02=%d\n", example1.field02);
```

Finite size

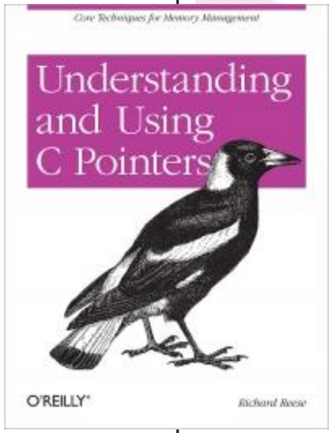
- Computers are finite state machines
  - Memory is finite
    - Variables have finite length
      - We always need to be aware of the size of a variable
      - We always need to be aware of the signedness
  - Examples
    - unsigned char: 8 bits
      - An unsigned char can holds integer values between:
        - » 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
- Exemplo
  - ILP64
    - Integer: 64 bits; Long: 64 bits;Pointer:64



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


Core Techniques for Memory Management
Understanding and Using C Pointers
O'REILLY
Richard Reese

Memory models (1)

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

```
int main(void){
        printf("sizeof(char)=%u bytes\n", sizeof(char));
        printf("sizeof(short)=%u bytes\n", sizeof(short));
        printf("sizeof(int)=%u bytes\n", sizeof(int));
        printf("sizeof(long)=%u bytes\n", sizeof(long));
        printf("sizeof(long long int)=%u bytes\n", sizeof(long long int));
        printf("sizeof(float)=%u bytes\n", sizeof(float));
        printf("sizeof(double)=%u bytes\n", sizeof(double));
        printf("sizeof(long double)=%u bytes\n", sizeof(long double));
        printf("sizeof(char*)=%u bytes\n", sizeof(char*));
        printf("sizeof(short*)=%u bytes\n", sizeof(short*));
        printf("sizeof(long double*)=%u bytes\n", sizeof(long double*));

        return 0;
}
```

Results with gcc 5.4 in a 32-bit virtual machine >>

Size of basic datatypes

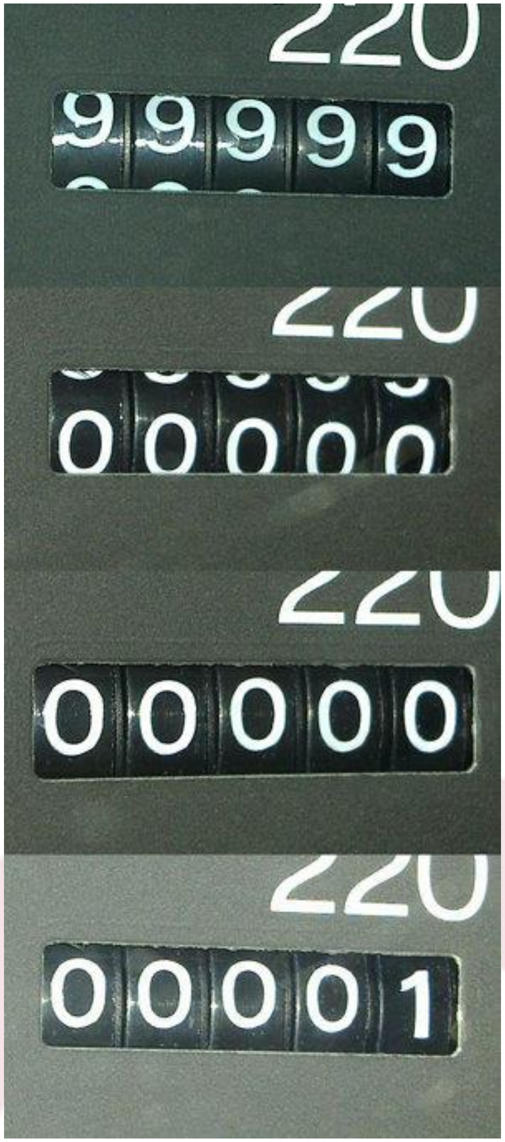
- Compiled with gcc 5.4 in a 32-bit lubuntu 16.04
  - uname -a
    - Linux ubuntu 4.4.0-21-generic #37-Ubuntu SMP Mon Apr 18 18:34:49 UTC 2016 **i686 i686 i686** GNU/Linux
  - `sizeof(char)=1 bytes`
  - `sizeof(short)=2 bytes`
  - `sizeof(int)=4 bytes`
  - `sizeof(long)=4 bytes`
  - `sizeof(long long 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`**

32-bit version

Overflow of integer variables >>

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)

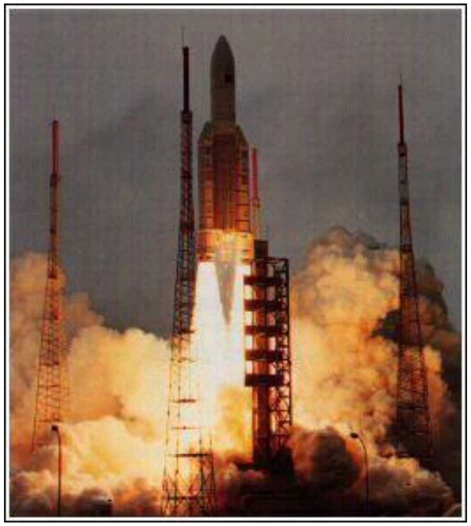
Overflow of integer variables (3)

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

      sign bit
    - 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 of integer variables - Ariane 5 (1)



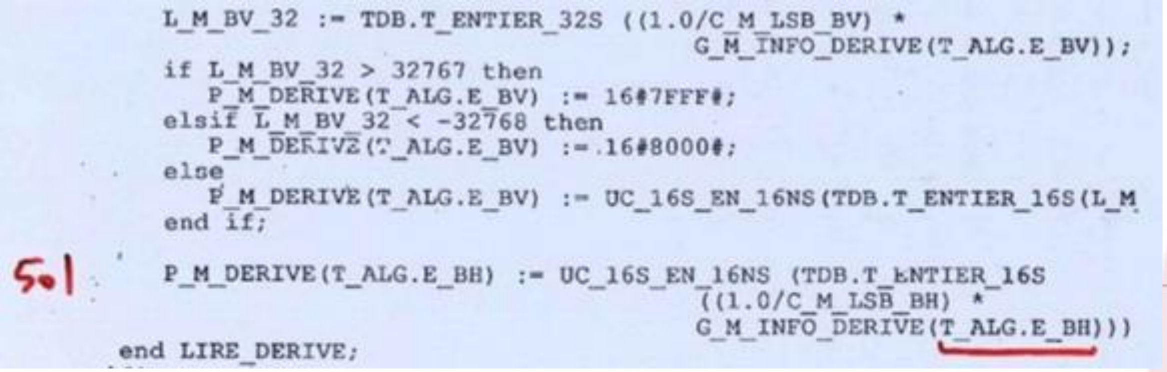
- Maiden flight of spacecraft Ariane 5
  - https://www.youtube.com/watch?v=gp_D8r-2hwk
- *On June 4th, 1996, only 30 seconds after the launch, the Ariane 5 rocket began to disintegrate slowly and exploded.*
- *In the rocket’s software (which came from Ariane 4), a 64-bit floating point variable with 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” (http://www.around.com/ariane.html)

- 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;

501   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;
```

Código ADA do Ariane 5 com indicação da variável E_BH

Fonte: *http://accu.org/content/images/journals/ol120/moene/Ariane-ADA.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)**

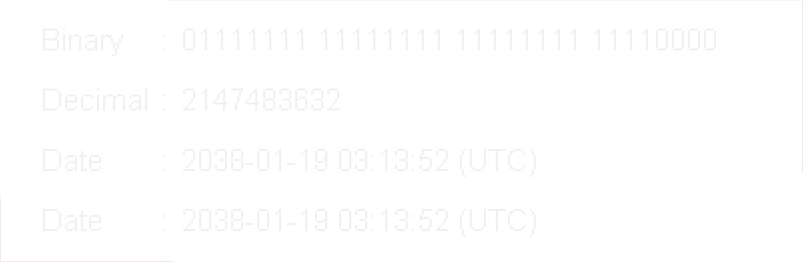
```
    if (key == '\b') {  // Does not checks underflows !!
      cur_len--; // Integer underflow !!
      grub_printf ("\b");
      continue;
     }
   if (!grub_isprint (key))
    continue;
   if (cur_len + 2 < buf_size){
      buf[cur_len++] = key; // Off-by-two !!
      grub_printf ("%c", key);
     }
  }
 // 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

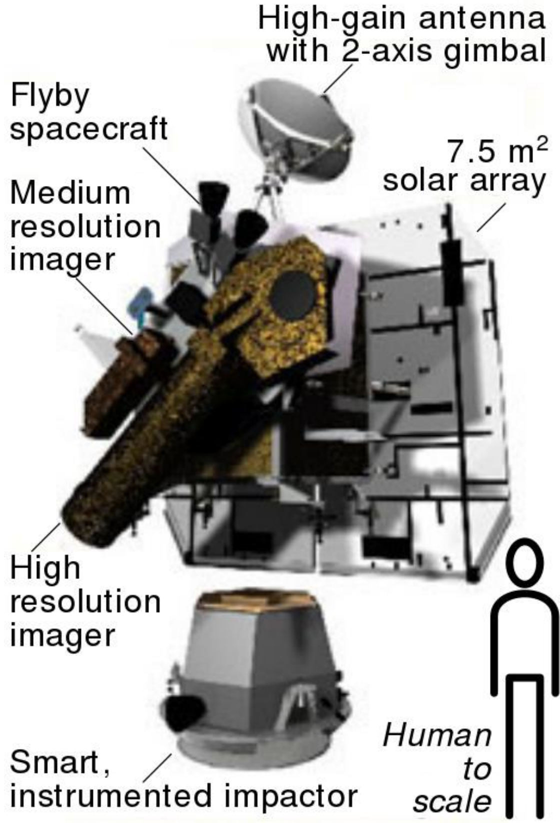
- Time origin (“zero time”) is
  - 1 jan 1970 00:00 GMT (“UNIX EPOCH”)
- The datatype `time_t` is used to hold a `signed integer` value to represent time
  - Number of seconds elapsed since EPOCH
  - Example: `time_t now = time(NULL);`
- In system that uses a **32-bit signed integer**, overflow will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



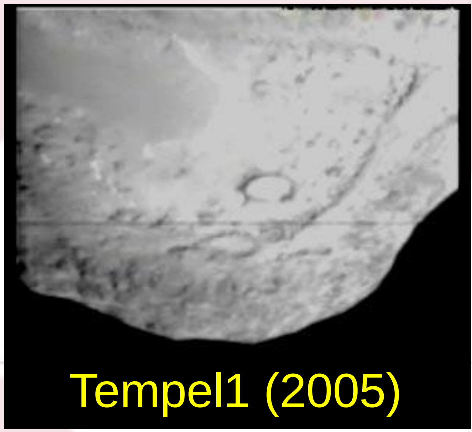
```
Binary  : 01111111 11111111 11111111 11110000
Decimal : 2147483632
Date    : 2038-01-19 03:13:52 (UTC)
Date    : 2038-01-19 03:13:52 (UTC)
```

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 in 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


Tempel1 (2005)

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

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
  - 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 (0000.0001.0010.0011.0100.0101.0111)
out:fedcba98 (1111.1110.1101.1100.1011.1010.1000)
```

Binary *AND* & (#1)

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

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

- Example

```
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

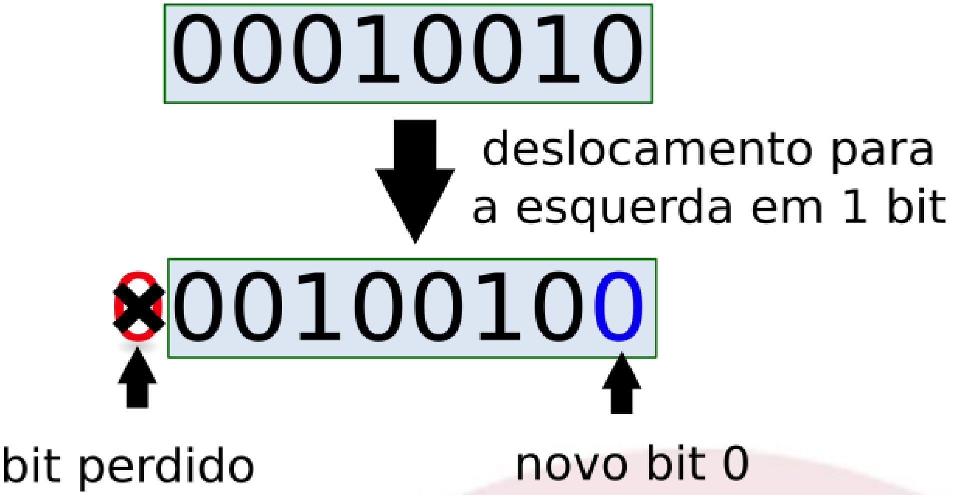
XOR (eXclusive OR)

- 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
- Example
  - A = 2;
  - A << 3? ➔ 16
  - X << N
    - multiply X by 2^N



Left shift << (#2)

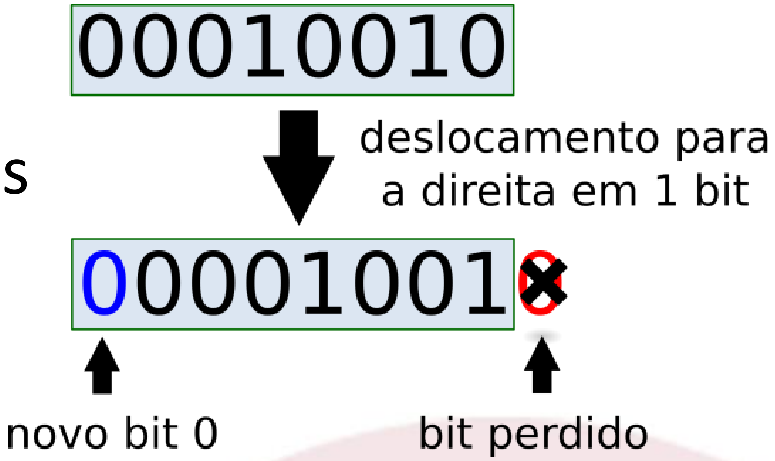
- Example
  - Left shift is a fast way to multiply by 2
  - But, watch out for overflow!

```
#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 >> for divisions…

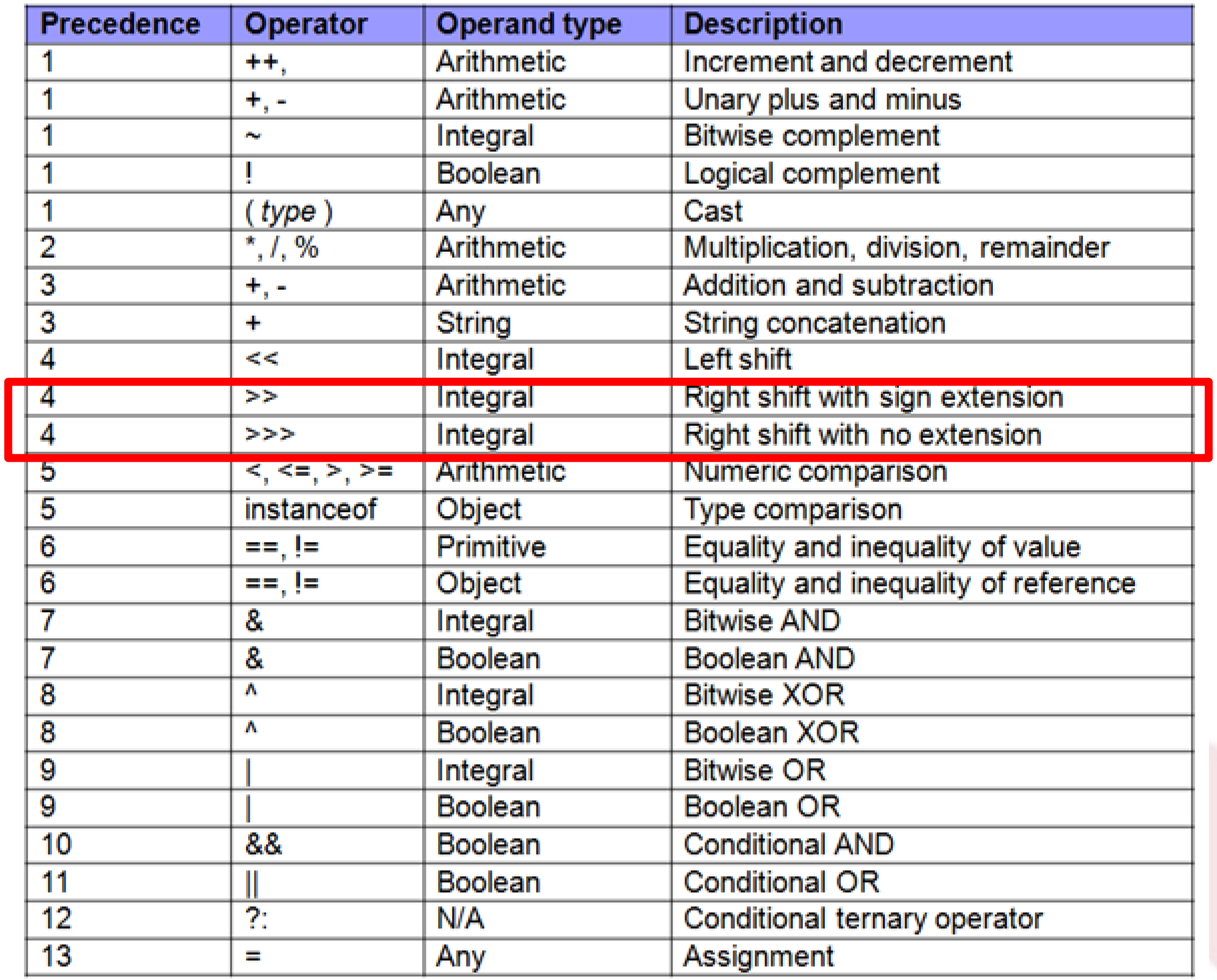
```
#include <stdio.h>
int main(void){
    int positive = 998;
    int negative = -998;
    int positive_shift, negative_shift;
    int i;
    for(i=0; i < 4; i++){
        printf(“===[shift right %d]===\n”,i);
        positive_shift = positive >> I;
        negative_shift = negative >> I;
    printf(“positive_shift=%d\n”,positive_shift);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

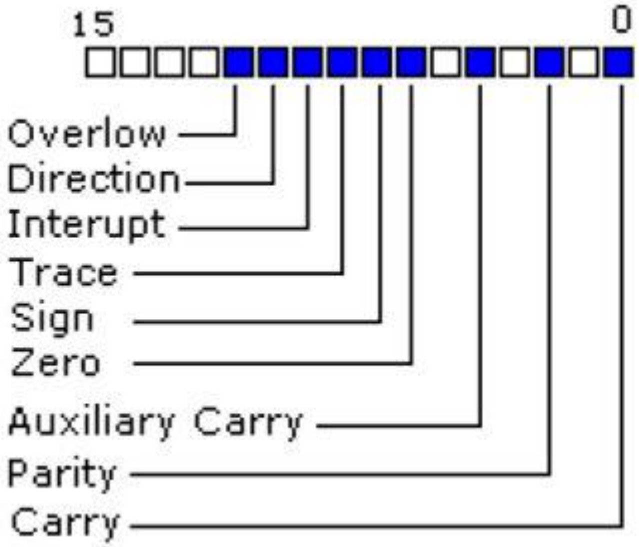
```
int A = …; /* set 4th bit to 1 */
int mask_4rd_bit = 0x1 << 3;
int A_4th_bit1 = A | mask_4rd_bit;
```

Bit recipes

- check if the nth bit is set:
  - `(flags & (1 << n)) != 0`
- set the nth bit:
  - `flags |= (1 << n)`
- clear the nth bit:
  - `flags &= ~(1 << n)`
- Flip the nth bit:
  - `flags ^= (1<<n)`

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);
```

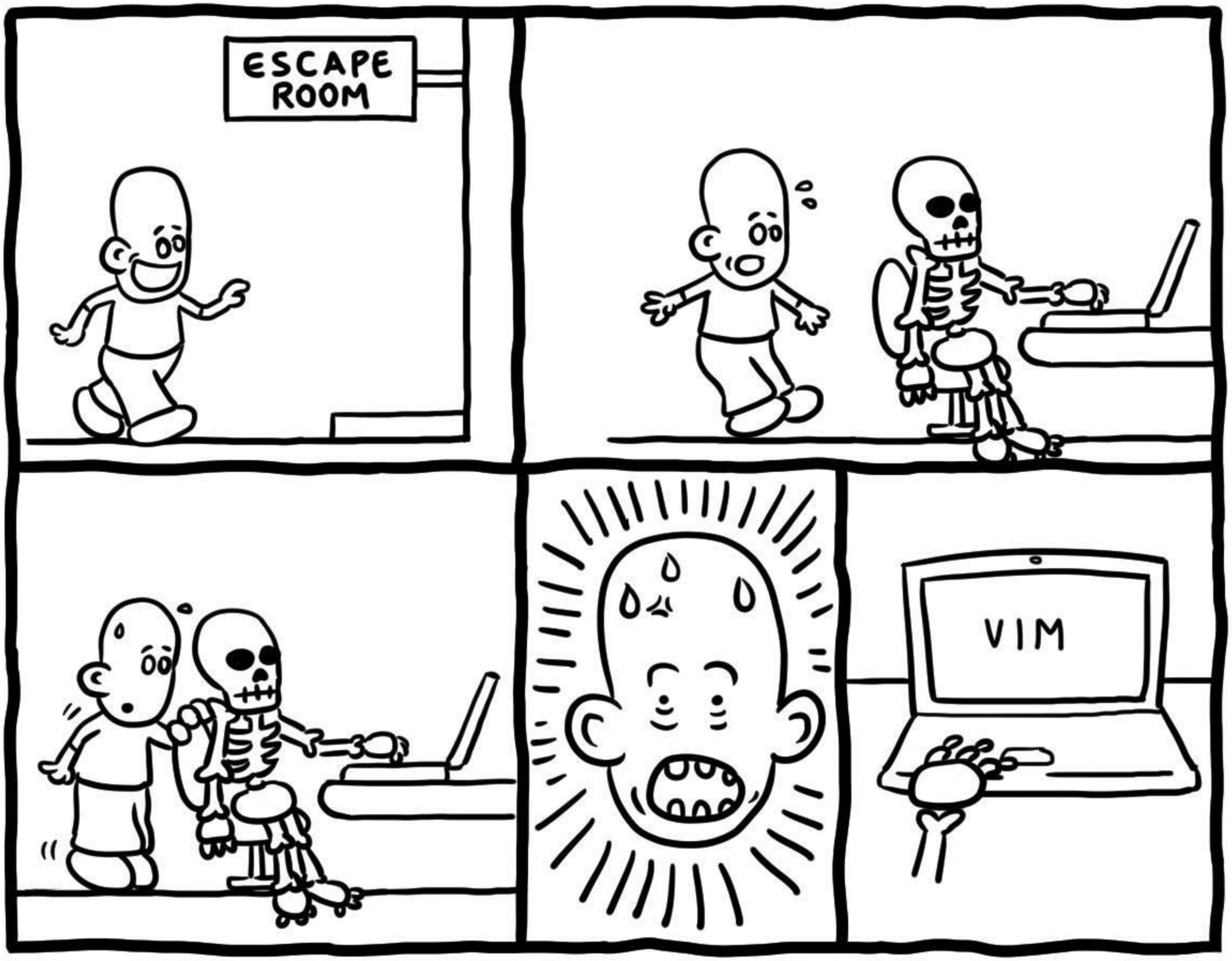
Continue >>

Flags in the C library (#2)

- Example

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

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



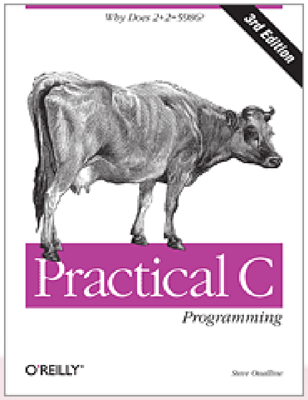
Daniel Stori {turnoff.us}

Bibliography

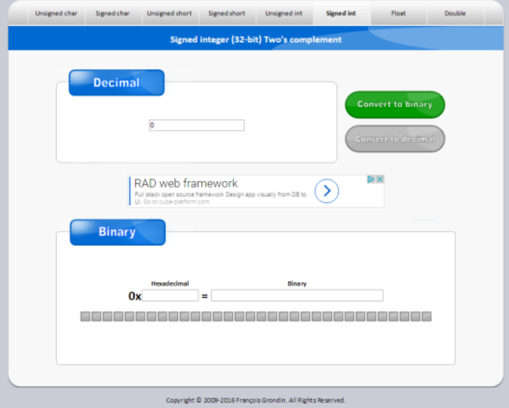
- Patrício Domingues. “Manipulação ao nível do bit na Linguagem C”. Revista Programar, Número 50, pp. 26-35, Setembro 2015, ISSN 1647 0710. http://bit.ly/2dmD74H

PROGRAMAR
REVISTA PORTUGUESA DE PROGRAMAÇÃO • WWW.PORTUGAL-A-PROGRAMAR.PT
AZURE LOGIC APPS
EDIÇÃO 50
O FUTURO DOS BACKENDS?
C#

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



- Online resources
  - Conversor binário / decimal / tipos de dados
    - http://www.binaryconvert.com/convert_signed_int.html


Unsigned char
Signed char
Unsigned short
Signed short
Unsigned int
Signed int
Float
Double
Signed integer (32-bit) Two's complement
Decimal
0
Convert to binary
Convert to decimal
RAD web framework
Full stack open source framework Design app visually from DB to UI. Go to cuba-platform.com
Binary
Hexadecimal
Binary
0x
=
Copyright © 2009-2016 François Grondin. All Rights Reserved.