Chp 4 : memory management

Arash HABIBI

Assistant Professor and Researcher
in Computer Graphics

Department of computer science

University of Strasbourg

ICube Laboratory

Outline

Numbers can be represented in decimal by a series of n **digits** d_{n-1}, d_{n-2}, ..., d_2, d_1 and d_0, all between 0 and 9.

- d_0 is called the units digit
- d_1 is the tens
- d_2 is the hundreds
- etc

For example the digits of 216 are : d_0=6 (units) d_1=1 (tens) d_2=2 (hundreds)

4.1. Counting in base 2 or 16

How to obtain N+1 ?

For a given number N represented by $d_{n-1}, d_{n-2}, \ldots, d_2, d_1, d_0$.

In order to obtain the digits for N+1 :

Here is the algorithm :

But this algorithm can be used not only for decimal numbers (base 10). It can be used in any other base :

0	**10**	**...**	**90**	**100**
1	11	...	91	101
2	12	...	92	etc
3	13	...	93	
4	14	...	94	
5	15	...	95	
6	16	...	96	
7	17	...	97	
8	18	...	98	
9	19	...	99	

4.1. Counting in base 2 or 16 — digits in base 4

Number N can be represented in ***base 4*** by a series of n digits d_{n-1}, d_{n-2}, ..., d_2, d_1 and d_0, all ***between 0 and 3***.

In order to obtain the digits for N+1, we use the same algorithm :

if $d_0 < 4$ then we increment d_0 otherwise, d_0 becomes 0 and

if $d_1 < 4$ then we increment d_1 otherwise, d_1 becomes 0 and

if $d_2 < 4$ ...

decimal (base 10)	base 4
0	0
1	1
2	2
3	3
4	10
5	11
6	12
7	13
8	20
9	21
10	22
11	23
12	30
13	31
14	32
15	33
16	100
17	101
etc	etc

UFR Mathématique Informatique Strasbourg
département d'informatique

UNIVERSITÉ DE STRASBOURG

digits in base 2

Number N can be represented in ***base 2*** by a series of n digits d_{n-1}, d_{n-2}, ..., d_2, d_1 and d_0, all ***between 0 and 1***.

In order to obtain the digits for N+1, we use the same algorithm :

if $d_0 < 2$ then we increment d_0 otherwise, d_0 becomes 0 and

if $d_1 < 2$ then we increment d_1 otherwise, d_1 becomes 0 and

if $d_2 < 2$ ...

decimal (base 10)	binary (base 2)
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001
etc	etc

4.1. Counting in base 2 or 16

digits in base 16

In base b, all digits $d_i \in [0,b-1]$.
In base b<10, the digits are a subset of {0,1,2,3,4,5,6,7,8,9}.
But for b>10, we have to invent other symbols.

For example in base 16 (hexadecimal) the 16 digits are

{0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}

decimal (base 10)	hexadecimal (base 16)
0	0
1	1
2	2
...	...
9	9
10	a
11	b
12	c
13	d
14	e
15	f
16	10
17	11
..	...
30	1e
31	1f
32	20
33	21
etc	etc

UFR Mathématique Informatique Strasbourg
département d'informatique

4.1. Counting in base 2 or 16 general formula

The general relation that enables to calculate the value of a number described in base b is expressed as follows :

Integer N is expressed in base b by n digits :
$d_{n-1}, d_{n-2}, \ldots, d_2, d_1, d_0$, (with $d_i \in [0,b-1]$) if and only if :

$$N = d_{n-1}.b^{n-1} + d_{n-2}.b^{n-2} + \ldots + d_2.b^2 + d_1.b + d_0$$

With this relation, calculate the decimal values of :

$(1001)_2$, $(11111111)_2$, $(3311)_4$, $(100)_8$, $(ff)_{16}$, $(12)_{16}$

4.1. Counting in base 2 or 16 — What use ?

The only base known by computers for numbers is ***base 2***. In other words, the computers know only ***binary*** numbers. In a binary number, each digit is called a ***bit***.

Addresses have typically 32 or 64 bits. But addresses written with 32 or 64 binary digits are not very readable for humans.

You know that with 4 binary digits, we can write numbers from 0 to 16. In hexadecimal, this is what you can do with ONE digit.

This is why, in order to represented addresses, we group the binary digits 4 by 4 and represent each group of 4 bits by a hexadecimal number.

For example :

$$(0001\ \ 1000\ \ 0110\ \ 1010\ \ 0000)_2 = (186A0)_{16}$$

Questions :

- Write a table with the first 16 integers in binary and in hexadecimal
- How would you write $(0010\ \ 0011\ \ 1010\ \ 1000\ \ 1100)_2$ in hexadecimal ?
- How would you write $(29ae5)_{16}$ in binary ?

4.2. Program and process

What difference ?

You are getting better and better at writing programs. But could anyone tell me the difference between :

- a **program** and
- a **process** ?

Let us look at the following program written in file `nap.c`.

4.2. Program and process

Experiment

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

int main()
{     printf("I am going to sleep !\n");
      sleep(5);
      return 0;
}
```

Function `int sleep(int n);` is declared in `stdlib.h`. It simple waits n seconds without doing anything else. We compile the program in the following way. nap is a ***program***.

```
gcc nap.c -o nap
```

Process Status : ps

Typing `ps -u ahabibi` in a terminal prints the list of *processes* running on the machine and whose owner is ahabibi. (`ps` stands for *Process Status*).

```
  PID TTY          TIME CMD
20854 pts/6    00:00:00 ps
24410 ?        00:00:00 firefox
24420 pts/6    00:00:00 bash
```

which means that user `ahabibi` has 3 running processes : `ps, firefox` and `bash`.

Now remember the program we compiled : let us run it. Type :

./nap

and, in another terminal, print once again the list of running processes :

```
ps -u ahabibi
```

Here is the result :

```
  PID TTY          TIME CMD
24410 ?        00:00:00 firefox
24420 pts/6    00:00:00 bash
30825 pts/6    00:00:00 nap
31560 pts/6    00:00:00 ps
```

With this result in mind, who can tell me the difference between a **program** and a **process** ?

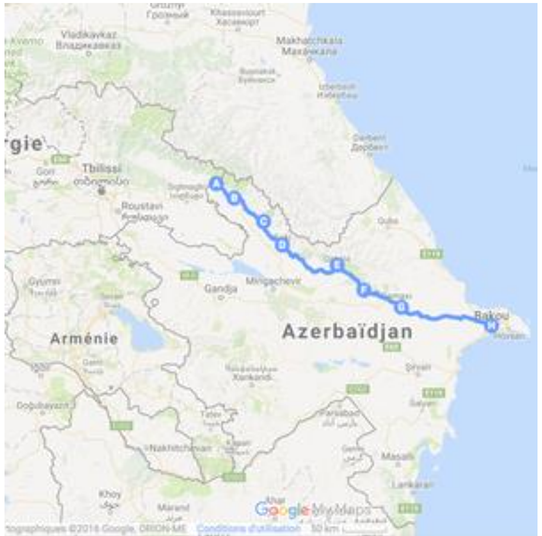
4.2. Program and process

route vs trip

A program is only a file. It does not take any processing time. When you run the program, the machine reads it and executes the program’s instructions.

A process is born. And it needs processor time and memory.

program



process

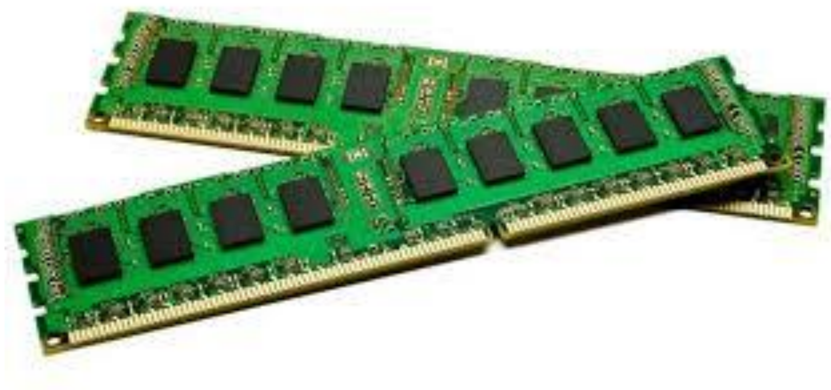
UFR Mathématique Informatique Strasbourg
département d’informatique

UNIVERSITÉ DE STRASBOURG

Variables and addresses

When a process is launched :

- the program is loaded in the main ***memory*** ;
- for each variable involved in the program, some place is allocated in the main memory ;
- In the memory, each variable, is stored at a specific place characterized by an **address** (a number).



How many bits ?

Questions :

- If we use 2 binary digits to write numbers, how many different numbers can we have ? What if we used 3 bits (3 binary digits) ? or 8 bits ?
- If we use 8 bits to write the addresses of a memory, how many elements (numbers with an address) can this memory contain ?
- What if we used 32 bits to write the addresses ?

4.3. Memory and addresses

representation of memory

The memory is composed of ***data elements***, each with a specific ***address***.

This is why we will often represent memory as a ***table*** with two columns, one representing the addresses and the other one the data.

This table has 64 bit addresses. Hexadecimal numbers have often a 0x prefix.

addresses	data
...	...
0x7fff5d029aea	00100101
0x7fff5d029aeb	10100100
0x7fff5d029aec	00000000
0x7fff5d029aed	11101001
0x7fff5d029aee	10101010
0x7fff5d029aef	11010110
0x7fff5d029af0	00101010
...	...

4.3. Memory and addresses

If the following program was launched, the memory allocated to the process could resemble this :

```
#include <stdio.h>

int main()
{       char c=0;
        char d=4;
        return 0;
}
```

	addresses	data
d	0x7fff5d029aea	00000100
c	0x7fff5d029aeb	00000000

The variable names are not stored in each memory location, but we will sometimes add them for more readability.

UFR Mathématique Informatique Strasbourg
département d'informatique

4.3. Memory and addresses — Operator &

The unary* **&** operator returns the address of its operand. For example the following program :

```
#include <stdio.h>

int main()
{   char c=0;
    char d=4;
    printf("%p %p\n",&c,&d);
    return 0;
}
```

outputs :

```
0x7fff5d029aeb 0x7fff5d029aea
```

	addresses	data
d	0x7fff5d029aea	00000100
c	0x7fff5d029aeb	00000000

* There is also a binary & operator which performs bitwise logical AND operation. We will not be using this binary operator.

UFR Mathématique Informatique Strasbourg
département d'informatique

We have talked about addresses so far. Let us talk about the data.

In the memory, the smallest data unit that can be located by an address is an ***octet*** or a ***byte***, which is composed of 8 bits.

Do you remember how many different values can be represented by 8 bits ?

addresses	data
...	...
0x7fff5d029aea	00100101
0x7fff5d029aeb	10100100
0x7fff5d029aec	00000000
0x7fff5d029aed	11101001
0x7fff5d029aee	10101010
0x7fff5d029aef	11010110
0x7fff5d029af0	00101010
...	...

Other types

With 8 bits, we can only have $2^8 = 256$ different values. But there are many variables who can have much more than 256 possible values.

– `char` :	$2^8 = 256$	possible values
– `short int` :	$2^{16} = 65536$	possible values
– `int` :	$2^{32} \approx$ 4 billions	possible values
– `long int` :	$2^{64} \approx 2.10^{19}$	possible values

- You can write `short int` or simply `short`;
- You can write `long int` or simply `long`;

4.4. Different types and sizes

Experiment

Consider the following program (left) and its output (right)

```c
#include <stdio.h>
int main()
{       char a,b;
        short c,d;
        int e,f;
        long g,h;
        printf("chars   : %p %p\n",&b, &a);
        printf("shorts  : %p %p\n",&d, &c);
        printf("ints    : %p %p\n",&f, &e);
        printf("longs   : %p %p\n",&h, &g);
        return 0;
}
```

```
chars   :      0x7fff5d19daea
               0x7fff5d19daeb

shorts  :      0x7fff5d19dae6
               0x7fff5d19dae8

ints    :      0x7fff5d19dadc
               0x7fff5d19dae0

longs   :      0x7fff5d19dac8
               0x7fff5d19dad0
```

Does that give you a hint on the way the memory represents shorts, ints and longs ?

4.4. Different types and sizes

`sizeof` operator

The ***sizeof*** operator returns the size (in bytes) of its argument (type or variable)

```
#include <stdio.h>
struct color {int x, y, z;};
int main()
{       printf("Size of a char : %lu bytes \n",                 sizeof(char));
        printf("Size of an unsigned char : %lu bytes \n",       sizeof(unsigned char));
        printf("Size of a short (int) : %lu bytes \n",          sizeof(short));
        printf("Size of an integer : %lu bytes \n",             sizeof(int));
        printf("Size of a long (int) : %lu bytes \n",           sizeof(long));
        printf("Size of a float : %lu bytes \n",                sizeof(float));
        printf("Size of a double : %lu bytes \n",               sizeof(double));
        // struct color was defined in this program just before the main function.
        printf("Size of a color : %lu bytes \n",                sizeof(struct color));
        return 0;
}
```

A few sizes

Here is the output :

```
Size of a char :                1 bytes
Size of an unsigned char :      1 bytes
Size of a short (int) :         2 bytes
Size of an integer :            4 bytes
Size of a long (int) :          8 bytes
Size of a float :               4 bytes
Size of a double :              8 bytes
Size of a color :               12 bytes
```

signed or unsigned ?

All four integer types (`char, short, int, long`) can be either `signed` or `unsigned`.

`signed char`	:	possible values : from -128 to 127
`unsigned char`	:	possible values : from 0 to 255
`char`		means `signed char` by default
`signed short`	:	possible values : from -32768 to 32768
`unsigned short`	:	possible values : from 0 to 65535
`short`		means `signed short` by default.

The same thing is true for `int` and for `long`.

Overflow experiment

To illustrate what happens when we try to make a variable go out of its scope, let us make an experiment :

```
#include <stdio.h>
int main()
{     unsigned char a=254;
      signed char b=126;
      printf("%d %d\n",a,b);
      a++; b++;
      printf("%d %d\n",a,b);
      a++; b++;
      printf("%d %d\n",a,b);
      return 0;
}
```

The output of this program is :

```
a=254      b=126
a=255      b=127
a=0        b=-128
```

4.1. Different types and sizes

Overflow experiment

To illustrate the fact that, by default, all types are signed, consider the following program :

```
#include <stdio.h>
int main()
{     unsigned short a;
      short b;
      a = b = 32767;
      printf("a=%d b=%d\n",a,b);
      a++; b++;
      printf("a=%d b=%d\n",a,b);
      return 0;
}
```

The output of this program is :

```
a=32767 b=32767
a=32768 b=-32768
```

Which variable experienced an overflow ?

4.1. Different types and sizes

Overflow experiment

Let us change a few details in this program :

```
#include <stdio.h>
int main()
{     unsigned short a;
      short b;
      a = b = 0;
      printf("a=%d b=%d\n",a,b);
      a--; b--;
      printf("a=%d b=%d\n",a,b);
      return 0;
}
```

The output of this program is :

```
a=0      b=0
a=65535  b=-1
```

Which variable experienced an overflow ?

Conclusion

- All your variables are stored in the main memory ;
- Each variable, is characterized by an address which indicates the place in the memory where it has been stored ;
- The data and the adresses are all represented in the machine in base 2 (binary) ;
- In our representations, we often write the addresses by hexadecimal numbers for more readability ;
- The unary & operator returns the address of its operand ;
- The data unit in the memory is the byte
- The different data types can be stored on one or several bytes.