Programmation
La mémoire, adresses et pointeurs

Régis Barbanchon

L1 Info-Math, Semestre 2

Obtenir le nombre de bits dans un byte

Un `char` est par définition stocké sur un byte,
qui est la plus petite unité de mémoire manipulable.

La macro `CHAR_BIT` définie dans `<limits.h>` indique le nombre de bits dans un `char` et donc dans un byte.

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

int main (void)
{
  printf ("bits per byte: %d\n", CHAR_BIT);
  return 0;
}
```

En C, un byte est garanti d'avoir au moins 8 bits.
En pratique, sur les machines modernes, un byte = 8 bits (octet) :

```
bits per byte: 8
```

Obtenir le nombre de bytes dans un type

La primitive `sizeof(Type)` donne la taille du type `Type`.
Cette taille, de type `size_t` est un nombre entier de bytes :

```c
#include <stdio.h>

int main (void)
{
  printf ("char : %d byte\n",  (int) sizeof(char));
  printf ("short: %d bytes\n", (int) sizeof(short));
  printf ("int  : %d bytes\n", (int) sizeof(int));
  printf ("long : %d bytes\n", (int) sizeof(long));
  return 0;
}
```

Sur une architecture 32-bits :

```
char : 1 byte
short: 2 bytes
int  : 4 bytes
long : 4 bytes
```

Sur une architecture 64-bits :

```
char : 1 byte
short: 2 bytes
int  : 4 bytes
long : 8 bytes
```

Obtenir le nombre de bits dans un type

Pour connaître la taille en bits d'un type `Type`,
il faut multiplier `sizeof(Type)` par `CHAR_BIT` :

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

int main (void)
{
  printf ("char : %d bits\n", (int) (CHAR_BIT * sizeof(char)));
  printf ("short: %d bits\n", (int) (CHAR_BIT * sizeof(short)));
  printf ("int  : %d bits\n", (int) (CHAR_BIT * sizeof(int)));
  printf ("long : %d bits\n", (int) (CHAR_BIT * sizeof(long)));
  return 0;
}
```

Sur une architecture 32-bits :

```
char : 8 bits
short: 16 bits
int  : 32 bits
long : 32 bits
```

Sur une architecture 64-bits :

```
char : 8 bits
short: 16 bits
int  : 32 bits
long : 64 bits
```

La limite des entiers non-signés

famille	*garantie en C*	*archi 32-bits*	*archi 64-bits*
`char`	$\geq$ 8 bits	= 8 bits	= 8 bits
`short`	$\geq$ 16 bits	=16 bits	=16 bits
`int`	$\geq$ 16 bits	=32 bits	=32 bits
`long`	$\geq$ 32 bits	=32 bits	=64 bits

Limites supérieures des entiers non-signés sur architecture 64-bits :

`UCHAR_MAX`	255	$= 2^8 - 1$
`USHRT_MAX`	65 535	$= 2^{16} - 1$
`UINT_MAX`	4 294 967 295 U	$= 2^{32} - 1$
`ULONG_MAX`	18 446 744 073 709 551 615 UL	$= 2^{64} - 1$

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

int main (void)
{
  printf ("%u\n", UCHAR_MAX); printf ( "%u\n", USHRT_MAX);
  printf ("%u\n", UINT_MAX);  printf ("%lu\n", ULONG_MAX);
  return 0;
}
```

La représentation signée en complément à 2

Les machines modernes représentent les nombres signés sur n bits en complément à 2 (pour $n = 8,\ 16,\ 32,\ 64$) :

- les nombres positif sont repr. par les motifs de bits $< 2^{n-1}$,
- les nombres négatifs sont repr. par les motifs de bits $\geq 2^{n-1}$,
- $-x$ est représenté par le motif de bits de $2^n - x \pmod{2^n}$.

Exemple fictif pour $n = 3$ bits :

motif de bits	non-signé	signé	remarques
000	0U	$+0$	repr. unique pour zéro
001	1U	$+1$	
010	2U	$+2$	
011	3U	$+3$	
100	4U	-4	pas d'opposé repr. pour -2^{n-1}
101	5U	-3	
110	6U	-2	
111	7U	-1	

La limite des entiers signés

Limites inférieures des entiers signés sur architecture 64-bits :

SCHAR_MIN	-128	$= -2^{7}$
SHRT_MIN	-32 768	$= -2^{15}$
INT_MIN	-2 147 483 648	$= -2^{31}$
LONG_MIN	-9 223 372 036 854 775 808 L	$= -2^{63}$

Limites supérieures des entiers signés sur architecture 64-bits :

SCHAR_MAX	+127	$= +2^{7} - 1$
SHRT_MAX	+32 767	$= +2^{15} - 1$
INT_MAX	+2 147 483 647	$= +2^{31} - 1$
LONG_MAX	+9 223 372 036 854 775 807 L	$= +2^{63} - 1$

Le type `char` peut être soit `signed char` soit `unsigned char`, au choix du compilateur, et donc l'intervalle `CHAR_MIN..CHAR_MAX` peut être soit `SCHAR_MIN..SCHAR_MAX`, soit `0..UCHAR_MAX`.

Endianness : Big-Endian vs Little-Endian

L'endianness est l'ordre des bytes d'un entier en mémoire.

Ex : Si `0xDEADBEEF` occupe 4 bytes à partir d'une adresse `addr`, la machine y ordonne les 4 bytes `0xDE`, `0xAD`, `0xBE`, `0xEF` (`0xDE` est le byte de poids fort, et `0xEF` le byte de poids faible).

En Big-Endian, il sont stockés du poids fort vers le poids faible :

adresses	addr	addr+1	addr+2	addr+3
unsigned char	DE	AD	BE	EF
unsigned short	DEAD		BEEF	
unsigned int	DEADBEEF			

En Little-Endian, ils sont stockés du poids faible vers le poids fort :

adresses	addr	addr+1	addr+2	addr+3
unsigned char	EF	BE	AD	DE
unsigned short	BEEF		DEAD	
unsigned int	DEADBEEF			

L'endianness des processeurs Intel

Ce programme permet de voir comment les bytes sont stockés :

```
int main (void)
{
  // fields of a union start at same address and overlap in memory.
  union {
    unsigned int   uints  [1];
    unsigned short ushorts[2];
    unsigned char  uchars [4];
  } mem;

  mem.ints[0]= 0xDEADBEEFU;

  printf ("mem.uints  []: %08X\n", mem.uints[0]);

  printf ("mem.ushorts[]: %04X %04X\n", mem.ushorts[0],
                                        mem.ushorts[1]);

  printf ("mem.uchars []: %02X %02X %02X %02X\n", mem.uchars[0],
                                                  mem.uchars[1],
                                                  mem.uchars[2],
                                                  mem.uchars[3]);

  return 0;
}
```

Les PC sont Little-Endian, on y obtient donc la sortie :

```
mem.uints  []: DEADBEEF
mem.ushorts[]: BEEF DEAD
mem.uchars []: EF BE AD DE
```

L'opérateur de référencement & (esperluette)

L'adresse d'une variable `var` s'obtient par l'expression `& var`.
Si `var` est de type `TypeName`, alors `& var` est de type `TypeName *`, où `TypeName *` est le type *pointeur sur* `TypeName`.

```
int main (void)
{
  int    age    = 42  ; int    * addr_of_age    = & age;
  double weight= 63.5; double * addr_of_weight= & weight;
  ...
```

`void *` est le type *pointeur sur type inconnu*.
On peut convertir tout type de pointeur sur donnée en `void *`, et `printf("%p", ...)` peut afficher un pointeur `void *` :

```
   ...
   printf ("addr_of_age    : %p\n", (void *) addr_of_age);
   printf ("addr_of_weight: %p\n", (void *) addr_of_weight);
   return 0;
}
```

```
addr_of_age    : 0x7ffe115ee128
addr_of_weight: 0x7ffe115ee120
```

L'opérateur de déréférencement * (étoile), ou indirection

Étant donné un pointeur `addr_of_var` de type `TypeName *` contenant l'adresse d'une variable `var` de type `TypeName`, on atteint cette variable par l'expression `* addr_of_var` :

```
int main (void) {
  int    age    = 42;   int    * addr_of_age    = & age;
  double weight= 63.5; double * addr_of_weight= & weight;
  printf ("age    : %d\n", * addr_of_age);
  printf ("weight: %f\n", * addr_of_weight);
  return 0;
}
```

```
age    : 42
weight: 63.5
```

Modifier `* addr_of_var`, c'est modifier `var` (et vice-versa) :

```
int main (void) {
  int age= 42; int * addr_of_age= & age;
  (* addr_of_age)++; printf ("age: %d is %d\n", age, * addr_of_age);
  age++;             printf ("age: %d is %d\n", age, * addr_of_age);
  return 0;
}
```

```
age: 43 is 43
age: 44 is 44
```

Pointeurs en paramètre de fonction (1/2)

Lors d'une invocation de fonction :

- les valeurs des arguments sont recopiés dans les paramètres.
- Modifier un paramètre ne modifie donc pas l'argument original (lorsque ce dernier est une variable).

```
void Int_SwapValues (int left, int right) // cannot behave as expected
{
  int old_left= left;
  left = right;
  right= old_left;
}
```

```
int main (void)
{
  int first= 1, last= 2;
  Int_SwapValues (first, last);
  printf ("values not swapped: %d %d\n", first, last);
  return 0;
}
```

```
values not swapped: 1 2
```

En modifiant `left` et `right`, on n'a pas modifié `first` et `last`.

Pointeurs en paramètre de fonction (2/2)

Un pointeur en paramètre d'une fonction permet à celle-ci de modifier une variable extérieure pointée.

```
void Int_Swap (int * addr_of_left, int * addr_of_right) // behaves as expected
{
  int old_left   = * addr_of_left;
  * addr_of_left = * addr_of_right;
  * addr_of_right= old_left;
}
```

```
int main (void)
{
  int first= 1, last= 2;
  Int_Swap (& first, & last);
  printf ("values swapped: %d %d\n", first, last);
  return 0;
}
```

```
values swapped: 2 1
```

En modifiant `* addr_of_left` et `* addr_of_right`, on a modifié les variables pointées `first` et `last`.

Remarque : c'est ce même mécanisme qui utilisé par `scanf()`.

Adresse d'un tableau (1/2)

L'adresse d'un tableau est l'adresse de sa 1ère case (d'indice 0).
L'identificateur du tableau s'évalue en son adresse.

```
int main (void) {
  int array[3]= { 10, 20, 30 };
  int * addr_of_first_cell= & array[0];
  int * addr_of_array= array;
  printf ("%p is %p\n", (void *) addr_of_array, (void *) addr_of_first_cell);
  return 0;
}
```

```
0x7ffe30c13670 is 0x7ffe30c13670
```

On utilise les crochets d'accès aux cases sur une adresse de tableau de la même manière que sur le tableau lui-même :

```
int main (void) {
  int array[3]= { 10, 20, 30 };
  int * addr_of_array= array;
  addr_of_array[1]= 666; // same as: array[1]= 666;
  return 0;
}
```

Adresse d'un tableau (2/2)

Lorsqu'on utilise les crochets pour le paramètre d'une fonction...

```
void ArrayOfInt_Init (int array[], int length, int value) {
  for (int k= 0; k < length; k++) {
     array [k]= value;
  }
}
```

Il s'agit de sucre syntaxique masquant un pointeur sur le tableau :

```
void ArrayOfInt_Init (int * addr_of_array, int length, int value) {
  for (int k= 0; k < length; k++) {
     addr_of_array [k]= value;
  }
}
```

Le tableau n'est pas copié, c'est son adresse qui est copiée.
C'est la raison pour laquelle il est modifiable dans la fonction.

```
int main (void) {
  int devil_numbers[3];
  ArrayOfInt_Init (devil_numbers, 3, 666);
  return 0;
}
```

Pointeurs en retour de fonction (1/4)

Soit une structure représentant un point du plan :

```
typedef struct Point {
  double x, y;
} Point;

Point Point_Make (double x, double y) {
  return (Point) { .x= x, .y= y };
}
```

Écrivons une fonction retournant sa représentation en chaîne :

```
int main (void) {
  Point p= Point_Make (5.0, 8.0);
  printf ("%s\n", Point_ToString (p));
  return 0;
}
```

L'implémentation ci-dessous ne fonctionnera pas :

```
char * Point_ToString (Point point) {
  char local_string [128];
  sprintf (local_string, "<%f, %f>", point.x, point.y);
  return local_string; // illegal, local_string[] does not survive the function
}
```

Car il est interdit de retourner l'adresse d'une variable locale.

Pointeurs en retour de fonction (2/4)

On peut s'en sortir en fournissant la chaîne de sortie en argument :

```
int main (void) {
  Point p= Point_Make (5.0, 8.0);
  char text [128];
  printf ("%s\n", Point_ToString (p, text));
  return 0;
}
```

`Point_ToString()` retourne alors l'adresse de son paramètre :

```
char * Point_ToString (Point point, char provided_string[]) {
  sprintf (provided_string, "<%f, %f>", point.x, point.y);
  return provided_string; // legal, provided_string[] lives outside the function
}
```

Cela fonctionne car la durée de vie de l'argument `text[]`
ne dépend pas de la fonction `Point_ToString()`.

Pointeurs en retour de fonction (3/4)

Si l'on ne souhaite pas fournir la chaîne en argument...

```
int main (void) {
  Point p= Point_Make (5.0, 8.0);
  printf ("%s\n", Point_ToString (p));
  return 0;
}
```

on peut utiliser une variable `static` (= var globale à nom local) :

```
char * Point_ToString (Point point) {
  static char static_string [128];
  sprintf (static_string, "<%f, %f>", point.x, point.y);
  return static_string; // legal, static_string[] is a global variable
}
```

```
int main (void) {
  Point p1= Point_Make (5.0, 8.0), p2= Point_Make (6.0, 9.0);

  // same global variable written twice before entering printf()
  printf ("%s %s\n", Point_ToString (p1), Point_ToString (p2));

  // behaves as expected, second write occurs after first printf() completed
  printf ("%s " , Point_ToString (p1));
  printf ("%s\n", Point_ToString (p2));
  return 0;
}
```

Pointeurs en retour de fonction (4/4)

On peut mixer les deux approches en fournissant :

- soit l'adresse d'un tableau pre-alloué :

```
int main (void) {
  Point p1= Point_Make (5.0, 8.0), p2= Point_Make (6.0, 9.0);
  char text1 [128], text2 [128];
  printf ("%s %s\n", Point_ToString(p1, text1), Point_ToString(p2, text2));
  return 0;
}
```

- soit l'adresse NULL de <stdlib.h>, qui ne pointe sur rien :

```
int main (void) {
  Point p= Point_Make (5.0, 8.0);
  printf ("%s\n", Point_ToString (p, NULL));
  return 0;
}
```

et dans ce dernier cas la fonction utilise une variable static :

```
char * Point_ToString (Point point, char optional_string[]) {
  static char static_string [128];
  char * result= (optional_string != NULL) ? optional_string : static_string;
  sprintf (result, "<%f, %f>", point.x, point.y);
  return result;
}
```

Pointeurs sur structures, opérateur -> (flèche)

Supposons que l'on veuille initialiser un point via une fonction :

```
void Point_Init (Point * addr_of_point, double x, double y) { ... }

int main (void) {
  Point point;
  Point_Init ( & point, 5.0, 8.0);
  printf ("<%f, %f>", point.x, point.y);
  return 0;
}
```

Il faut déréférencer `addr_of_point` avant d'accèder aux champs :

```
void Point_Init (Point * addr_of_point, double x, double y) {
  (* addr_of_point).x= x;  // should use operator -> below instead
  (* addr_of_point).y= y;  // should use operator -> below instead
}
```

Cette écriture étant lourde, on utilise l'opérateur `->` (flèche) :

```
void Point_Init (Point * addr_of_point, double x, double y) {
  addr_of_point->x= x;
  addr_of_point->y= y;
}
```

`pointer->field` est equivalent à `(* pointer).field`

Qualificateur const sur le type pointé

Lorsqu'une fonction ne souhaite pas modifier la donnée pointée, elle le signale en qualifiant le type pointé par `const` :
Exemple pour une structure :

```
void
Point_Print (Point const * addr_of_point, FILE * file) {
  fprintf (file, "<%f,%f>", addr_of_point->x, addr_of_point->y);
}
```

Exemple pour un tableaux dont les cases ne sont pas modifiées :

```
bool
ArrayOfInt_IsSorted (int const array[], int length) {
  for (int k= 1; k < length; k++) {
    if (array[k] < array[k-1]) return false;
  }
  return true;
}
```

Rappelons que derrière le sucre syntaxique `int array[]`,
se cache le pointeur `int * array`, pointant sur la première case :

```
bool
ArrayOfInt_IsSorted (int const * array, int length) {
  // same code...
}
```