Programmation
Arithmétique des pointeurs et tableaux

Régis Barbanchon

L1 Info-Math, Semestre 2

Addition d'un pointeur `char *` et d'un entier (1/3)

Soit :

- `addr` une addresse de type `char *` pointant sur un byte,
- `k` un entier,

alors en C, la somme `addr + k`

- est une addresse de type `char *`,
- dont la valeur est égale à `addr` décalée de `k` bytes.

Pour que l'opération soit valide, la plage de bytes séparant les 2 adresses doit faire partie d'un bloc réservé d'un seul tenant.

(L'adresse générée peut pointer immédiatement après ce bloc mais on ne peut alors pas la déréférencer).

Addition d'un pointeur `char *` et d'un entier (2/3)

Par exemple, étant donné une chaîne `array[]` valant `"Text"`, c-à-d, un tableau de caractères `{ 'T', 'e', 'x', 't', '\0' }`, alors, si `addr` pointe sur le caractère `'x'`, on a :

'T'	'e'	'x'	't'	'\0'	XXX
addr-2	addr-1	addr	addr+1	addr+2	(addr+3)
...b417	...b418	...b419	...b41a	...b41b	(...b41c)

Donc, si `addr` pointe vers l'adresse `& array[index]`
alors `addr+k` pointe vers l'adresse `& array[index+k]` :

```
int main (void) {
  char array []= "Text";
  char * addr= & array[2];
  for (int k= -2; k <= +2; k++)
    printf ("addr %+d = %p : %3d = '%c'\n",
            k,   (void *)(addr + k),   *(addr + k),   *(addr + k));
  return 0;
}
```

```
addr -2 = 0x7fff460db417 :  84 = 'T'
addr -1 = 0x7fff460db418 : 101 = 'e'
addr +0 = 0x7fff460db419 : 120 = 'x'
addr +1 = 0x7fff460db41a : 116 = 't'
addr +2 = 0x7fff460db41b :   0 = ''
```

Addition d'un pointeur `char *` et d'un entier (3/3)

Tout pointeur de type `TypeName *`
peut être converti, avec un cast explicite, en un pointeur sur byte de type `char *`, `signed char *`, ou `unsigned char *`.

Exemple : `unsigned int *` converti en `unsigned char *`

```
int main (void) {
  unsigned int var= 0xDEADBEEFU;

  unsigned int  * addr_of_var = & var;
  unsigned char * addr_of_byte= (unsigned char *) addr_of_var; // explicit cast

  int nb_bytes= sizeof var;
  for (int k= 0; k < nb_bytes; k++) {
    printf ("addr_of_byte %+d = %p : %2X\n",
            k,   (void *)(addr_of_byte + k),   *(addr_of_byte + k));
  }
  return 0;
}
```

Sur une machine Little-Endian comme les PC, on obtient :

```
addr_of_byte +0 = 0x7ffe67305ba8 : EF
addr_of_byte +1 = 0x7ffe67305ba9 : BE
addr_of_byte +2 = 0x7ffe67305baa : AD
addr_of_byte +3 = 0x7ffe67305bab : DE
```

Addition d'un pointeur `TypeName *` et d'un entier (1/2)

De manière plus générale, soit :

- `TypeName` un type de taille `s = sizeof (TypeName)` bytes,
- `addr` une addresse de type pointeur `TypeName *` pointant sur une donnée de type `TypeName`,
- `k` un entier,

alors en C, la somme `addr + k`

- est une addresse de type `TypeName *`,
- dont la valeur est égale à `addr` décalée de `k * s` bytes.

`addr + k` $\Longleftrightarrow$ `(TypeName *) ((char *) addr + k * s)`

Pour que l'opération soit valide, la plage de bytes séparant les 2 adresses doit faire partie d'un bloc réservé d'un seul tenant. (L'adresse générée peut pointer immédiatement après ce bloc mais on ne peut alors pas la déréférencer).

Addition d'un pointeur `TypeName *` et d'un entier (2/2)

Par exemple, étant donné un tableau d'entiers

`int array[]={ 0, 10, 20, 30, 40 };`

alors, si `sizeof(int) == 4` et `addr` pointe sur l'entier `20`, on a :

0	10	20	30	40	XXX
addr-2	addr-1	addr	addr+1	addr+2	(addr+3)
...60d0	...60d4	...60d8	...60dc	...60e0	(...60e4)

Donc, si `addr` pointe vers l'adresse `& array[index]`

alors `addr+k` pointe vers l'adresse `& array[index+k]` :

```
int main (void) {
  int array []= { 0, 10, 20, 30, 40 };
  int * addr= & array[2];
  for (int k= -2; k <= +2; k++)
    printf ("addr %+d = %p : %2d\n",
             k,   (void *)(addr + k),    *(addr + k));
  return 0;
}
```

```
addr -2 = 0x7ffe9e2a60d0 :  0
addr -1 = 0x7ffe9e2a60d4 : 10
addr +0 = 0x7ffe9e2a60d8 : 20
addr +1 = 0x7ffe9e2a60dc : 30
addr +2 = 0x7ffe9e2a60e0 : 40
```

Équivalence entre accès tableau et pointeur+entier

Quelque soit le type `TypeName` et sa taille `sizeof(TypeName)`,
si `array` est l'adresse d'un tableau de ce type,
et `k` un index valide pour ce tableau

on a l'équivalence suivante : $\texttt{array + k} \iff \texttt{\& array[k]}$ et $\texttt{k + array} \iff \texttt{\& k[array]}$

Et comme les opérateurs `&` et `*` sont réciproques
(c-à-d `* & var` $\iff$ `var` et `& * addr` $\iff$ `addr`)

on a l'équivalence suivante : $\texttt{*(array + k)} \iff \texttt{array[k]}$ et $\texttt{*(k + array)} \iff \texttt{k[array]}$

L'opérateur crochets des tableaux est donc du sucre syntaxique
cachant de l'arithmétique sur les pointeurs.

Tableaux en paramètres de fonction (1/3)

Par exemple, la fonction suivante `ArrayOfInt_Print` qui acccède à son paramètre tableau en lecture...

```
void
ArrayOfInt_Print (int const array [], int length, FILE * file)
{
  fprintf (file, "[ ");
  for (int k= 0; k < length; k++)
    fprintf (file, "%3d ", array [k]);
  fprintf (file, "]\n");
}
```

...pourrait être réécrite de façon équivalente comme :

```
void
ArrayOfInt_Print (int const * array, int length, FILE * file)
{
  fprintf (file, "[ ");
  for (int k= 0; k < length; k++)
    fprintf (file, "%3d ", *(array + k) );
  fprintf (file, "]\n");
}
```

Tableaux en paramètres de fonction (2/3)

De même, la fonction suivante `ArrayOfInt_FillWithValue` qui acccède à son paramètre tableau en écriture...

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

...pourrait être réécrite de façon équivalente comme :

```
void
ArrayOfInt_FillWithValue (int * array, int length, int value)
{
  for (int k= 0; k < length; k++) {
    * (array + k)= value;
  }
}
```

Tableaux en paramètres de fonction (3/3)

Dans le manuel (man) documentant les fonctions, on trouve :

- les tableaux exprimés en terme de pointeurs,
- `const` placé à gauche du nom de type plutôt qu'à sa droite.

```
$ man 3 strcmp
```

```
STRCMP(3)                  Linux Programmer's Manual                 STRCMP(3)

NAME
       strcmp, strncmp - compare two strings

SYNOPSIS
       #include <string.h>
       int strcmp(const char *s1, const char *s2);
       int strncmp(const char *s1, const char *s2, size_t n);

DESCRIPTION
       The  strcmp()  function compares the two strings s1 and s2.  It returns
       an integer less than, equal to, or greater than zero if  s1  is  found,
       respectively, to be less than, to match, or be greater than s2.

       The  strncmp()  function  is similar, except it compares only the first
       (at most) n bytes of s1 and s2.

RETURN VALUE
       The strcmp() and strncmp() functions return an integer less than, equal
       to, or greater than zero if s1 (or the first n bytes thereof) is found,
       respectively, to be less than, to match, or be greater than s2.

CONFORMING TO
       POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD.
```

Autres opérateurs entre pointeur et entier

En plus de l'addition `addr + offset`, on a la soustration `addr - offset` ainsi que les opérateurs dérivés habituels :

- les accumulation additive et soustractive
  `addr += offset` $\iff$ `addr= addr + offset`
  `addr -= offset` $\iff$ `addr= addr - offset`
  (la valeur de l'expression est celle de `addr` après sa modif.)
- l'incrémentation et la décrémentation préfixes
  `++ addr` $\iff$ `addr += 1`
  `-- addr` $\iff$ `addr -= 1`
  (la valeur de l'expression est celle de `addr` après sa modif.)
- l'incrémentation et la décrémentation postfixes
  `addr ++` $\iff$ `addr += 1, addr-1`
  `addr --` $\iff$ `addr -= 1, addr+1`
  (la valeur de l'expression est celle de `addr` avant sa modif.)

Soustraction de deux pointeurs de même type (1/2)

Soit :

- un tableau `array` d'éléments de type `TypeName`,
- deux indices `index1` et `index2` de cases dans `array`,
- `addr1` de type `TypeName *` valant `& array[index1]`,
- `addr2` de type `TypeName *` valant `& array[index2]`,

alors :

- la différence `addr1 - addr2` est égale à `index1 - index2`,
- l'ordre `addr1 < addr2` est le même que `index1 < index2`.

```
int main (void) {
  double array []= { 0.0, 10.0, 20.0, 30.0, 40.0, 50.0 };
  double * addr1= & array[4];
  double * addr2= & array[1];
  int diff= addr1 - addr2;
  printf ("at addr1= %p : value %f\n", (void *) addr1, * addr1);
  printf ("at addr2= %p : value %f\n", (void *) addr2, * addr2);
  printf ("%p - %p = %d\n", (void *) addr1, (void *) addr2, diff);
  return 0;
}
```

```
at addr1= 0x7ffca975caa0 : value 40.0
at addr2= 0x7ffca975ca94 : value 10.0
0x7ffca975caa0 - 0x7ffca975ca94 = 3
```

Soustraction de deux pointeurs de même type (2/2)

Exemple : voici une version personnelle de la fonction `strlen()`, qui retourne la longueur d'une chaîne (zéro-terminal exclu) :

```
int String_Length (char const text [])
{
  int length= 0;
  while (text [length] != '\0') {
    length++;
  }
  return length;
}
```

La même fonction recodée avec l'arithmétique de pointeurs :

```
int String_Length (char const * text)
{
  char const * end= text;
  while (* end != '\0') {
    end++;                    // pointer increment
  }
  return end - text;          // difference of two pointrers
}
```

Validité et Invalidité des opérations (1/2)

On ne doit pas soustraire deux pointeurs si les données pointées ne sont pas à l'intérieur d'une même zone d'allocation mémoire, garantissant la contiguité de la plage d'adresses :

```
int main (void)
{
  double array1 []= { 0.0, 10.0, 20.0, 30.0, 40.0 };
  double array2 []= { 0.5, 10.5, 20.5, 30.5, 40.5 };

  double * addr0= & array1 [1];
  double * addr1= & array1 [3];
  double * addr2= & array2 [3];

  int diff10= addr1 - addr0;     // legal, pointers both point inside array1
  int diff21= addr2 - addr1;     // illegal, array1 and array2 not contiguous
  return 0;
}
```

En effet, l'interprétation de la différence de deux pointeurs étant la différence des indices des élements pointés, l'opération n'a de sens que si les deux adresses pointent vers deux éléménts à l'intérieur d'un même tableau.

Validité et Invalidité des opérations (2/2)

En additionnant un entier à une adresse, on ne doit pas générer une adresse qui sort de la zone d'allocation de l'addresse initiale, à la seule exception de l'adresse qui suit immédiatement la zone.

(Dans ce cas, on ne peux pas déréferencer l'adresse obtenue.)

```
int main (void)
{
  double array []= { 0.0, 10.0, 20.0, 30.0, 40.0 };

  double * addr1= & array1 [3];
  addr1++; // legal, still inside array, at last element
  addr1++; // legal, points only one element behind array

  double value= * addr1; // illegal, dereference outside array
  addr1++;               // illegal, points two elements behind array

  double * addr2= & array1 [0];
  addr2--; // illegal, points one element ahead of array
  return 0;
}
```

Validité/invalidité : exemple du test de palindrome (1/9)

Test de palindrome avec des accès tableau :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Le même test codé avec l'arithmétique de pointeurs :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Validité/invalidité : exemple du test de palindrome (2/9)

On va comparer le comportement des deux fonctions,
en les lançant sur différents palindromes :

- la chaîne `"LAVAL"` de longueur impaire,
- la chaîne `"DEED"` de longueur paire,
- la chaîne `"A"` de longueur unitaire (cas limite),
- la chaîne `""` de longueur nulle (cas limite).

Enfin, on va comparer leur fragilité. Que se passe-t-il si :

- on enlève la clause de garde sur `(length == 0)` ?
- on remplace `<` par `<=` dans la condition de boucle ?

Validité/invalidité : exemple du test de palindrome (3a/9)

Test de palindrome avec des accès tableau sur "LAVAL" :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "LAVAL" de longueur impaire :

'L'	'A'	'V'	'A'	'L'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]

Validité/invalidité : exemple du test de palindrome (3a/9)

Test de palindrome avec des accès tableau sur "LAVAL" :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "LAVAL" de longueur impaire :

length= 5

'L'	'A'	'V'	'A'	'L'	'\0'
[0] •	[1]	[2]	[3]	[4] •	[5]

- left
- right

Validité/invalidité : exemple du test de palindrome (3a/9)

Test de palindrome avec des accès tableau sur "LAVAL" :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "LAVAL" de longueur impaire :

length= 5

'L'	'A'	'V'	'A'	'L'	'\0'
[0]	[1] •	[2]	[3] •	[4]	[5]

• left

• right

Validité/invalidité : exemple du test de palindrome (3a/9)

Test de palindrome avec des accès tableau sur "LAVAL" :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "LAVAL" de longueur impaire :

'L'	'A'	'V'	'A'	'L'	'\0'
[0]	[1]	[2] ●● !	[3]	[4]	[5]

length= 5

- left
- right

! STOP

À la sortie de la boucle, les indices left et right sont égaux.

Validité/invalidité : exemple du test de palindrome (3b/9)

Test de palindrome avec l'arithmétique de pointeur sur "LAVAL" :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "LAVAL" de longueur impaire :

'L'	'A'	'V'	'A'	'L'	'\0'
text+0	text+1	text+2	text+3	text+4	text+5

Validité/invalidité : exemple du test de palindrome (3b/9)

Test de palindrome avec l'arithmétique de pointeur sur "LAVAL" :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "LAVAL" de longueur impaire :

length= 5

'L'	'A'	'V'	'A'	'L'	'\0'
text+0 •	text+1	text+2	text+3	text+4 •	text+5

- start
- end

Validité/invalidité : exemple du test de palindrome (3b/9)

Test de palindrome avec l'arithmétique de pointeur sur "LAVAL" :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "LAVAL" de longueur impaire :

length= 5

'L'	'A'	'V'	'A'	'L'	'\0'
text+0	text+1 •	text+2	text+3 •	text+4	text+5

- start
- end

Validité/invalidité : exemple du test de palindrome (3b/9)

Test de palindrome avec l'arithmétique de pointeur sur "LAVAL" :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "LAVAL" de longueur impaire :

'L'	'A'	'V'	'A'	'L'	'\0'
text+0	text+1	text+2 ●● !	text+3	text+4	text+5

length= 5

- start
- end

! STOP

À la sortie de la boucle, les indices start et end sont égaux.

Validité/invalidité : exemple du test de palindrome (4a/9)

Test de palindrome avec des accès tableau sur "DEED" :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "DEED" de longueur paire :

'D'	'E'	'E'	'D'	'\0'
[0]	[1]	[2]	[3]	[4]

Validité/invalidité : exemple du test de palindrome (4a/9)

Test de palindrome avec des accès tableau sur "DEED" :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "DEED" de longueur paire :

length= 4

'D'	'E'	'E'	'D'	'\0'
[0] •	[1]	[2]	[3] •	[4]

- left
- right

Validité/invalidité : exemple du test de palindrome (4a/9)

Test de palindrome avec des accès tableau sur "DEED" :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "DEED" de longueur paire :

length= 4

'D'	'E'	'E'	'D'	'\0'
[0]	[1] •	[2] •	[3]	[4]

- left
- right

Validité/invalidité : exemple du test de palindrome (4a/9)

Test de palindrome avec des accès tableau sur "DEED" :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "DEED" de longueur paire :

'D'	'E'	'E'	'D'	'\0'
[0]	[1] • !	[2] • !	[3]	[4]

length= 4

- left
- right

! STOP

À la sortie de la boucle, les indices left et right se sont croisés.

Validité/invalidité : exemple du test de palindrome (4b/9)

Test de palindrome avec l'arithmétique de pointeur sur "DEED" :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "DEED" de longueur paire :

'D'	'E'	'E'	'D'	'\0'
text+0	text+1	text+2	text+3	text+4

Validité/invalidité : exemple du test de palindrome (4b/9)

Test de palindrome avec l'arithmétique de pointeur sur "DEED" :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "DEED" de longueur paire :

length= 4

'D'	'E'	'E'	'D'	'\0'
text+0 •	text+1	text+2	text+3 •	text+4

- start
- end

Validité/invalidité : exemple du test de palindrome (4b/9)

Test de palindrome avec l'arithmétique de pointeur sur "DEED" :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "DEED" de longueur paire :

length= 4

- start
- end

'D'	'E'	'E'	'D'	'\0'
text+0	text+1 •	text+2 •	text+3	text+4

Validité/invalidité : exemple du test de palindrome (4b/9)

Test de palindrome avec l'arithmétique de pointeur sur "DEED" :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "DEED" de longueur paire :

'D'	'E'	'E'	'D'	'\0'
text+0	text+1 • !	text+2 • !	text+3	text+4

length= 4

- start
- end

! STOP

À la sortie de la boucle, les indices start et end se sont croisés.

Validité/invalidité : exemple du test de palindrome (5a/9)

Test de palindrome avec des accès tableau sur `"A"` :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne `"A"` de longueur unitaire :

'A'	'\0'
[0]	[1]

Validité/invalidité : exemple du test de palindrome (5a/9)

Test de palindrome avec des accès tableau sur `"A"` :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne `"A"` de longueur unitaire :

'A'	'\0'
[0] ●● !	[1]

`length= 1`

- `left`
- `right`

`!` `STOP`

On ne rentre jamais dans le corps de la boucle.

Validité/invalidité : exemple du test de palindrome (5b/9)

Test de palindrome avec l'arithmétique de pointeur sur `"A"` :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne `"A"` de longueur unitaire :

'A'	'\0'
text+0	text+1

Validité/invalidité : exemple du test de palindrome (5b/9)

Test de palindrome avec l'arithmétique de pointeur sur `"A"` :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne `"A"` de longueur unitaire :

'A'	'\0'
text+0 ●● !	text+1

`length= 1`

- `start`
- `end`

`!` `STOP`

On ne rentre jamais dans le corps de la boucle.

Validité/invalidité : exemple du test de palindrome (6a/9)

Test de palindrome avec des accès tableau sur "" :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "" de longueur nulle :

'\0'
[0]

Validité/invalidité : exemple du test de palindrome (6a/9)

Test de palindrome avec des accès tableau sur "" :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "" de longueur nulle :

'\0'
[0]

Early exit : on sort sur la clause de garde `(length == 0)`.

Validité/invalidité : exemple du test de palindrome (6b/9)

Test de palindrome avec l'arithmétique de pointeur sur "" :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "" de longueur nulle :

'\0'
[0]

Validité/invalidité : exemple du test de palindrome (6b/9)

Test de palindrome avec l'arithmétique de pointeur sur "" :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "" de longueur nulle :

'\0'
[0]

Early exit : on sort aussi sur la clause de garde `(length == 0)`.

Validité/invalidité : exemple du test de palindrome (7a/9)

Test de palindrome avec des accès tableau sur "",
sans la clause de garde :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  // if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "" de longueur nulle :

XXX	'\0'
[-1]	[0]

Validité/invalidité : exemple du test de palindrome (7a/9)

Test de palindrome avec des accès tableau sur "",
sans la clause de garde :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  // if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left < right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "" de longueur nulle :

XXX	'\0'
[-1] • !	[0] • !

length= 0

- left
- right

! STOP

Ok : L'index `right` est invalide, mais on n'accède pas à la case.

Validité/invalidité : exemple du test de palindrome (7b/9)

Test de palindrome avec l'arithmétique de pointeur sur "", sans la clause de garde :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  // if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "" de longueur nulle :

XXX	'\0'
text-1	text+0

Validité/invalidité : exemple du test de palindrome (7b/9)

Test de palindrome avec l'arithmétique de pointeur sur "",
sans la clause de garde :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  // if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start < end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "" de longueur nulle :

XXX	'\0'
text-1 • ?	text+0 •

length= 0

- start
- end

? BUG

Erreur : le pointeur end généré est illégal (devant le tableau).

Validité/invalidité : exemple du test de palindrome (8a/9)

Test de palindrome avec des accès tableau sur "A",
utilisant <= dans la condition d'arrêt de boucle :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left <= right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "A" de longueur unitaire :

XXX	'A'	'\0'
[-1]	[0]	[1]

Validité/invalidité : exemple du test de palindrome (8a/9)

Test de palindrome avec des accès tableau sur `"A"`,
utilisant `<=` dans la condition d'arrêt de boucle :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left <= right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne `"A"` de longueur unitaire :

XXX	'A'	'\0'
[-1]	[0] ●●	[1]

`length= 1`

- `left`
- `right`

Validité/invalidité : exemple du test de palindrome (8a/9)

Test de palindrome avec des accès tableau sur "A",
utilisant <= dans la condition d'arrêt de boucle :

```
bool String_IsPalindrome_1 (char const text [])
{
  int length= strlen (text);
  if (length == 0) return true;

  int left= 0;
  int right= length - 1;
  while (left <= right) {
    if (text [left] != text [right]) return false;
    left++; right--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "A" de longueur unitaire :

XXX	'A'	'\0'
[-1]● !	[0]	[1] ● !

length= 1

- ● left
- ● right
- ! STOP

Ok : L'index right est invalide, mais on n'accède pas à la case.

Validité/invalidité : exemple du test de palindrome (8b/9)

Test de palindrome avec l'arithmétique de pointeur sur `"A"`, utilisant `<=` dans la condition d'arrêt de boucle :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start <= end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne `"A"` de longueur unitaire :

XXX	'A'	'\0'
text-1	text+0	text+1

Validité/invalidité : exemple du test de palindrome (8b/9)

Test de palindrome avec l'arithmétique de pointeur sur "A", utilisant <= dans la condition d'arrêt de boucle :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start <= end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne "A" de longueur unitaire :

XXX	'A'	'\0'
text-1	text+0 ●●	text+1

length= 1

- left
- right

Validité/invalidité : exemple du test de palindrome (8b/9)

Test de palindrome avec l'arithmétique de pointeur sur `"A"`, utilisant `<=` dans la condition d'arrêt de boucle :

```
bool String_IsPalindrome_2 (char const * text)
{
  int length= strlen (text);
  if (length == 0) return true;

  char const * start= text;
  char const * end= text + length - 1;
  while (start <= end) {
    if (* start != * end)  return false;
    start++; end--;
  }
  return true;
}
```

Lançons la fonction sur la chaîne `"A"` de longueur unitaire :

XXX	'A'	'\0'
text-1 • ?	text+0	text+1 •

`length= 1`

- `left`
- `right`
- `?` BUG

Erreur : Le pointeur généré `end` est illégal (devant le tableau).

Validité/invalidité : exemple du test de palindrome (9/9)

Conclusion : il faut manier l'arithmétique de pointeur plus prudemment que l'indexation de tableau, car :

- si on peut générer un index hors-tableau
  tant qu'on ne tente pas un accès de tableau sur cet index.
- en revanche, on ne doit pas générer une adresse hors-tableau
  même si on ne tente pas de la déréférencer.

Des bugs subtils peuvent apparaître,
en particulier dans les cas aux limites.