Programmation
Autour des chaînes de caractères
DRY (Don't Repeat Yourself)
DOT (Do One Thing)

Régis Barbanchon

L1 Info-Math, Semestre 2

Comparaison de chaînes (1/6)

On ne peut pas tester si 2 tableaux (et donc 2 chaînes) `array1` et `array2` ont même contenu avec `array1 == array2`.

En effet, un identificateur de tableau s'évaluant en son adresse, l'opérateur `==` teste si ses opérandes pointent sur la même adresse :

```
int main (void)
{
  char s1[]= "blabla";
  char s2[]= "blabla";

  printf ("s1 == s1 : %s\n", (s1 == s1) ? "true" : "false");
  printf ("s2 == s2 : %s\n", (s2 == s2) ? "true" : "false");
  printf ("\n");
  printf ("s1 == s2 : %s\n", (s1 == s2) ? "true" : "false");
  printf ("s2 == s1 : %s\n", (s2 == s1) ? "true" : "false");
  return 0;
}
```

```
s1 == s1 : true
s2 == s2 : true

s1 == s2 : false
s2 == s1 : false
```

Comparaison de chaînes (2/6)

Il faut donc passer par une fonction de comparaison.
Pour les chaînes à encodage monobyte (1 caractère par byte) tel l'ASCII (sur 7 bits) ou l'ISO-8859-1 (latin-1, sur 8 bits), il y a la fonction `strcmp()` de `<string.h>` :

```
$ 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.
```

Comparaison de chaînes (3/6)

Reprenons l'exemple précédent et ses deux chaînes égales :

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

int main (void)
{
  char s1[]= "blabla";
  char s2[]= "blabla";

  printf ("strcmp (s1, s1) : %d\n", strcmp (s1, s1));
  printf ("strcmp (s2, s2) : %d\n", strcmp (s2, s2));
  printf ("\n");
  printf ("strcmp (s1, s2) : %d\n", strcmp (s1, s2));
  printf ("strcmp (s2, s1) : %d\n", strcmp (s2, s1));
  return 0;
}
```

La fonction `strcmp()` retourne bien `0` dans tous les cas ici :

```
strcmp (s1, s1) : 0
strcmp (s2, s2) : 0

strcmp (s1, s2) : 0
strcmp (s2, s1) : 0
```

Comparaison de chaînes (4/6)

Testons `strcmp()` sur des chaînes variées :

```
int main (void)
{
  printf ("%+d\n", strcmp ("blabla", "blibli"));
  printf ("%+d\n", strcmp ("blabla", "blabla"));
  printf ("%+d\n", strcmp ("bla",    "blabla"));
  printf ("%+d\n", strcmp ("blabla", "bla"));
  return 0;
}
```

La sortie de son implémentation par clang est :

```
-8
+0
-98
+98
```

En fait, l'implémentation par clang de `strcmp(s1, s2)` :

- recherche la longueur `k` du prefixe commun de `s1` et `s2`,
- et retourne la comparaison de `s1[k]` et `s2[k]`,
  qui est la différence `s1[k] - s2[k]` des codes caractères.

Comparaison de chaînes (5/6)

Charset ISO-8859-1 (Latin 1)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			32	0 48	@ 64	P 80	‘ 96	p 112			32	º 176	À 192	Đ 208	à 224	ð 240
1			! 33	1 49	A 65	Q 81	a 97	q 113			¡ 161	± 177	Á 193	Ñ 209	á 225	ñ 241
2			" 34	2 50	B 66	R 82	b 98	r 114			¢ 162	² 178	Â 194	Ò 210	â 226	ò 242
3			# 35	3 51	C 67	S 83	c 99	s 115			£ 163	³ 179	Ã 195	Ó 211	ã 227	ó 243
4			$ 36	4 52	D 68	T 84	d 100	t 116			¤ 164	´ 180	Ä 196	Ô 212	ä 228	ô 244
5			% 37	5 53	E 69	U 85	e 101	u 117			¥ 165	µ 181	Å 197	Õ 213	å 229	õ 245
6			& 38	6 54	F 70	V 86	f 102	v 118			¦ 166	¶ 182	Æ 198	Ö 214	æ 230	ö 246
7			’ 39	7 55	G 71	W 87	g 103	w 119			§ 167	· 183	Ç 199	× 215	ç 231	÷ 247
8			(40	8 56	H 72	X 88	h 104	x 120			“ 168	¸ 184	È 200	Ø 216	è 232	ø 248
9			) 41	9 57	I 73	Y 89	i 105	y 121			© 169	¹ 185	É 201	Ù 217	é 233	ù 249
A			* 42	: 58	J 74	Z 90	j 106	z 122			ª 170	° 186	Ê 202	Ú 218	ê 234	ú 250
B			+ 43	; 59	K 75	[91	k 107	{ 123			« 171	» 187	Ë 203	Û 219	ë 235	û 251
C			, 44	< 60	L 76	\ 92	l 108	| 124			¬ 172	¼ 188	Ì 204	Ü 220	ì 236	ü 252
D			- 45	= 61	M 77	] 93	m 109	} 125			– 173	½ 189	Í 205	Ý 221	í 237	ý 253
E			. 46	> 62	N 78	^ 94	n 110	~ 126			® 174	¾ 190	Î 206	Þ 222	î 238	þ 254
F			/ 47	? 63	O 79	_ 95	o 111				¯ 175	¿ 191	Ï 207	ß 223	ï 239	ÿ 255

www.langbox.com - LangBox International

Les codes Latin-1
des caractères concernés
par les exemples précédents :

- `'a' == 0x61 == 97`
- `'b' == 0x62 == 98`
- `'i' == 0x69 == 105`
- `'l' == 0x6C == 108`
- `'\0' == 0x00 == 0`

Comparaison de chaînes (6/6)

Vérifions les résultats de nos exemples :

`strcmp("blabla","blibli") == 'a'-'i' == 97-105 == -8`

'b'	'l'	'a'●	'b'	'l'	'a'	'\0'
[0]	[1]	[2]●	[3]	[4]	[5]	[6]
'b'	'l'	'i'●	'b'	'l'	'i'	'\0'

`strcmp("blabla","blabla") == '\0'-'\0' == 0-0 == 0`

'b'	'l'	'a'	'b'	'l'	'a'	'\0'●
[0]	[1]	[2]	[3]	[4]	[5]	[6]●
'b'	'l'	'a'	'b'	'l'	'a'	'\0'●

`strcmp("bla","blabla") == '\0'-'b' == 0-98 == -98`

'b'	'l'	'a'	'\0'●			
[0]	[1]	[2]	[3]●	[4]	[5]	[6]
'b'	'l'	'a'	'b'●	'l'	'a'	'\0'

`strcmp("blabla","bla") == 'b'-'\0' == 98-0 == +98`

'b'	'l'	'a'	'b'●	'l'	'a'	'\0'
[0]	[1]	[2]	[3]●	[4]	[5]	[6]
'b'	'l'	'a'	'\0'●			

Comparaison de chaînes et principe DOT (1/6)

Tentons d'implémenter notre version personnelle de `strcmp()`.

Voici un 1er jet de notre fonction `String_Compare()` :

```
int String_Compare (char const s1[], char const s2[]) {
  int k;
  for (k= 0; s1 [k] != '\0'; k++) { // find common prefix length k...
    if (s1[k] != s2[k]) break;
  }
  return s1[k] - s2[k]; // compare characters at common prefix length k...
}
```

Testons ce premier jet (`\xE9` est le code Latin-1 du caractère `é`) :

```
void Test_String_Compare (void) {
  assert (String_Compare ("blabla",  "blibli") ==  97 - 105);
  assert (String_Compare ("blabla",  "blabla") ==   0 -   0);
  assert (String_Compare ("bla",     "blabla") ==   0 -  98);
  assert (String_Compare ("blabla",  "bla"   ) ==  98 -   0);
  assert (String_Compare ("R\xE9gis", "Regis") == 233 - 101);
}

int main (void) {
  Test_String_Compare();
  return 0;
}
```

L'exécution montre que le dernier test échoue :

```
void Test_String_Compare():
Assertion ‘String_Compare ("R\xE9gis", "Regis") == 233 - 101’ failed.
Aborted (core dumped)
```

Comparaison de chaînes et principe DOT (2/6)

Le résultat n'est pas correct pour `"R\xE9gis"` et `"Regis"`.
La cause profonde de l'erreur est que le programmeur concentre trop de sources d'erreurs dans une même fonction.

En effet, `String_Compare()` se charge à la fois :

1. de définir le calcul de la longueur `k` du préfixe commun,
2. de définir le calcul de comparaison de deux caractères,
3. d'articuler les deux pour comparer `s1[k]` et `s2[k]`.

En se concentrant sur l'item 1, le programmeur a négligé l'item 2.

Si `String_Compare()` se bornait à **ne faire qu'une seule chose** :

- on n'y verrait que l'item 3 articulant les autres items,
- l'item 1 serait extrait vers `String_CommonPrefixLength()`,
- l'item 2 serait extrait vers `Char_Compare()`.

Comparaison de chaînes et principe DOT (3/6)

Après extraction fonctionnelle, on obtient le découpage suivant, où chaque fonction ne fait qu'une seule chose :

```
int String_CommonPrefixLength (char const s1[], char const s2[]) {
  int k;
  for (k= 0; s1 [k] != '\0'; k++) {
    if (s1 [k] != s2 [k]) break;
  }
  return k;
}
```

```
int Char_Compare (char c1, char c2) {
  return c1 - c2;
}
```

```
int String_Compare (char const s1[], char const s2[]) {
  int length= String_CommonPrefixLength (s1, s2);
  return Char_Compare (s1 [length], s2 [length]);
}
```

On jongle avec moins de *balles mentales* dans une même fonction, et chaque fonction d'un niveau d'abstraction donné s'exprime en termes d'abstraction immédiatement inférieurs définis ailleurs. C'est le principe **Do One Thing** (abrégé par **DOT** dans ce cours).

Comparaison de chaînes et principe DOT (4/6)

Les fonctions ainsi découplées peuvent être testées séparément.

```
void Test_String_CommonPrefixLength (void) {
  assert (String_CommonPrefixLength ("blabla",   "blibli") == 2);
  assert (String_CommonPrefixLength ("blabla",   "blabla") == 6);
  assert (String_CommonPrefixLength ("bla",      "blabla") == 3);
  assert (String_CommonPrefixLength ("blabla",   "bla"   ) == 3);
  assert (String_CommonPrefixLength ("R\xE9gis", "Regis" ) == 1);
}
```

```
void Test_Char_Compare (void) {
  assert (Char_Compare ('a'   , 'i' ) ==  97 - 105);
  assert (Char_Compare ('\0'  , '\0') ==   0 -   0);
  assert (Char_Compare ('\0'  , 'a' ) ==   0 -  98);
  assert (Char_Compare ('a'   , '\0') ==  98 -   0);
  assert (Char_Compare ('\xE9', 'e' ) == 233 - 101);
}
```

```
int main (void) {
  Test_String_CommonPrefixLength();
  Test_Char_Compare();
  return 0;
}
```

On isole ainsi mieux la cause de l'erreur :

```
void Test_Char_Compare():
Assertion 'Char_Compare ('\xE9', 'e' ) == 233 - 101' failed.
Aborted (core dumped)
```

Comparaison de chaînes et principe DOT (5/6)

```
void Test_Char_Compare():
Assertion ‘Char_Compare (’\xE9’, ’e’ ) == 233 - 101’ failed.
Aborted (core dumped)
```

L'erreur est donc isolée dans `Char_Compare()` :

```
int Char_Compare (char c1, char c2) {
  return c1 - c2;
}
```

Ici, on manipule les valeurs numériques du type `char`
dont la *signedness* est dépendante du compilateur.

Clang sur Linux utilise la convention que `char` est `signed char`.
Un nombre au dessus de 127 allume donc son bit de signe,
qui porte la valeur $-2^7 = -128$ au lieu de $+2^7 = +128$,
et le total des bits bascule ainsi en valeur négative.

Comparaison de chaînes et principe DOT (6/6)

En particulier, la valeur non-signée `0xE9 == 233`
devient la valeur signée `-23` :

numéro du bits	7	6	5	4	3	2	1	0	total
bits de 233 et -23	1	1	1	0	1	0	0	1	= ?
poids des bits en non signé	128	64	32		8			1	= 233
poids des bits en signé	-128	64	32		8			1	= -23

On doit donc comparer les caractères en tant que `unsigned char`
si l'on souhaite comparer leurs codes numériques :

```
int Char_Compare (char c1, char c2) {
  return (unsigned char) c1 - (unsigned char) c2;
}
```

Et ainsi, les tests passent.

Chaînes et constantes littérales (1/3)

On a vu qu'il n'y pas de différence entre :

```
void Function (char string []) { }
```

et

```
void Function (char * string) { }
```

Cela n'est valable que pour les paramètres formels d'une fonction.
En C, il y a une différence entre les deux définitions suivantes :

```
char array1[]= "bla";
char array2[]= "bla";
```

et

```
char * ptr1= "bla";
char * ptr2= "bla";
```

À gauche, il n'y a pas de mauvaise surprise :

- on alloue 2 tableaux `array1` et `array2` de longueur 3+1,
- ils sont initialisés avec `{ 'b', 'l', 'a', '\0' }`,
- on peut modifier les cases de ces tableaux,
- les deux tableaux sont bien deux objets distincts
  - leurs adresses sont différentes, c-à-d `array1 != array2`,
  - la modification d'une case de l'un ne modifie rien dans l'autre.

Chaînes et constantes littérales (2/3)

On a vu qu'il n'y pas de différence entre :

```
void Function (char string []) { }
```

et

```
void Function (char * string) { }
```

Cela n'est valable que pour les paramètres formels d'une fonction.
En C, il y a une différence entre les deux définitions suivantes :

```
char array1[]= "bla";
char array2[]= "bla";
```

et

```
char * ptr1= "bla";
char * ptr2= "bla";
```

À droite, en revanche, c'est plus compliqué :

- `"bla"` est un littéral, indépendant de toute variable, dont on mémorise ensuite l'adresse dans un pointeur,
- les deux littéraux `"bla"` peuvent être distincts ou non (c-à-d, on peut avoir `ptr1 == ptr2` ou `ptr1 != ptr2`),
- ces littéraux peuvent être dans une zone interdite en écriture (il faut considérer `ptr1` et `ptr2` comme des `char const *`).

Chaînes et constantes littérales (3/3)

Pour les tableaux de caractères, on aura toujours la même sortie :

```
int main (void) {
  char array1[]= "bla", array2[]= "bla";
  printf ("strcmp (array1, array2) == %d\n", strcmp (array1, array2));
  printf ("array1 %s array2\n", (array1 == array2) ? "==" : "!=");
  array1[0]= 'g';
  printf ("after change: <%s> and <%s>\n", array1, array2);
  return 0;
}
```

```
strcmp (array1, array2) == 0
array1 != array2
after change: <gla> and <bla>
```

Pour les pointeurs sur chaînes littérales, avec gcc ou clang :

```
int main (void) {
  char * ptr1= "bla", * ptr2= "bla";
  printf ("strcmp (ptr1, ptr2) == %d\n", strcmp (ptr1, ptr2));
  printf ("ptr1 %s ptr2\n", (ptr1 == ptr2) ? "==" : "!="); // unspecified
  ptr1[0]= 'g';                                            // undefined
  printf ("after change: <%s> and <%s>\n", ptr1, ptr2);
  return 0;
}
```

```
strcmp (ptr1, ptr2) == 0
ptr1 == ptr2                          # literals are shared in memory by clang
Segmentation fault (core dumped)      # literals are write-protected  by clang
```

Recherche d'un caractère : version prédicat

Codons le prédicat testant si un caractère appartient à une chaîne.
Remarque : Toujours utiliser le paradigme *Early-Exit*
en sortant de la fonction avec `return` dès que l'on peut.

```
bool String_ContainsChar (char const text [], char searched) {
  for (int k= 0; text[k] != '\0'; k++) {
    if (text[k] == searched) return true;            // early-exit (GOOD!)
  }
  return false;
}
```

Le paradigme inverse (*Single Entry/Single Exit*) obfusque le code en exigeant artificiellement de ne sortir qu'à l'accolade fermante :

- par rajout de variables de contrôle de flux ou de résultat,
- par rajout de conditionnelles

```
bool String_ContainsChar (char const text [], char searched) {
  bool found= false;                                 // extra variable.
  for (int k= 0; text[k] != '\0' && ! found; k++) { // extra condition.
    if (text[k] == searched) found= true;            // failure to return.
  }
  return found;                                      // single exit (BAD!)
}
```

Recherche d'un caractère : version index de l'occurrence

Modifions le prédicat afin de retourner l'index de l'occurrence, ou l'index invalide `-1` en cas de non-occurrence :

```
int String_CharIndex (char const text[], char searched)
{
  for (int k= 0; text[k] != '\0'; k++) {
    if (text[k] == searched) return k;                    // early-exit (GOOD!)
  }
  return -1;
}
```

Là encore, toujours utiliser le paradigme *Early-Exit* ci-dessus, et non le paradigme obsolète *Single Entry/Single Exit* ci-dessous :

```
int String_CharIndex (char const text[], char searched) {
  int result= -1;                                         // extra variable.
  for (int k= 0; text[k] != '\0' && result == -1; k++) {  // extra condition.
    if (text[k] == searched) result= k;                   // failure to return.
  }
  return result;                                          // single exit (BAD!)
}
```

Recherche d'un caractère : version adresse de l'occurrence

Modifions la fonction afin de retourner l'adresse de l'occurrence, ou `NULL` (le pointeur vers rien) en cas de non-occurrence :

```
char * String_FindChar (char const text [], char searched)
{
  for (int k= 0; text[k] != '\0'; k++) {
    if (text[k] == searched)
      return (char *) & text [k]; // explict cast to lose const qualifier.
  }
  return NULL;
}
```

Là encore, toujours utiliser le paradigme *Early-Exit* ci-dessus, et non le paradigme obsolète *Single Entry/Single Exit* ci-dessous :

```
char * String_FindChar (char const text [], char searched)
{
  char * result= NULL;                                          // extra variable.
  for (int k= 0; text[k] != '\0' && result == NULL; k++) {      // extra condition.
    if (text[k] == searched) result= (char *) & text [k];       // failure to return.
  }
  return result;                                                // single exit (BAD!)
}
```

Recherche d'un caractère et principe DRY (1/5)

Noter que l'on peut :

- réduire `ContainsChar()` et `FindChar()` à `CharIndex()`.
  En effet, si l'on a index de l'occurrence :
  - la présence est attestée par un index différent de `-1`,
  - on peut déduire son adresse par addition de pointeur.

- réduire `ContainsChar()` et `CharIndex()` à `FindChar()`.
  En effet, si l'on a l'adresse de l'occurrence :
  - la présence est attestée par une addresse différente de `NULL`,
  - on peut déduire son index par soustraction de deux pointeurs.

En revanche, on ne peux rien réduire à `ContainsChar()`.

Application du principe DRY (Don't Repeat Yourself) :
Supposons que l'on a déjà codé `CharIndex()` ou `FindChar()`,
et codons les autres fonctions en **réutilisant ce qui est déjà codé**.

Recherche d'un caractère et principe DRY (2/5)

Supposons que l'on a déjà codé `CharIndex()`, alors :

- en appliquant DRY, le prédicat `ContainsChar()` s'écrit :

```
bool String_ContainsChar (char const text[], char searched) {
  return String_CharIndex (text, searched) != -1;
}
```

Remarque : Ne pas écrire d'étude de cas fictive telle que...

```
bool String_ContainsChar (char const text[], char searched) {
  int index= String_CharIndex (text, searched);
  if (index != -1) return  true; // fictive case study (BAD!)
  else             return false;
}
```

- en appliquant DRY, la fonction `FindChar()` s'écrit :

```
char * String_FindChar (char const text[], char searched)  {
  int index= String_CharIndex (text, searched);
  if (index == -1) return NULL; // guard clause for exceptional flow
  return (char *) text + index; // regular flow at main indent level
}
```

Remarque : avec l'opérateur ternaire, on peut aussi écrire :

```
char * String_FindChar (char const text[], char searched)  {
  int index= String_CharIndex (text, searched);
  return (index != -1) ? (char *) text + index : NULL;
}
```

Recherche d'un caractère et principe DRY (3/5)

Supposons que l'on a déjà codé `FindChar()`, alors :

- en appliquant DRY, le prédicat `ContainsChar()` s'écrit :

```
bool String_ContainsChar (char const text[], char searched) {
  return String_FindChar (text, searched) != NULL;
}
```

Remarque : Ne pas écrire d'étude de cas fictive telle que...

```
bool String_ContainsChar (char const text [], char searched) {
  char * found= String_FindChar (text, searched);
  if (found != NULL) return  true; // fictive case study (BAD!)
  else               return false;
}
```

- en appliquant DRY, la fonction `CharIndex()` s'écrit :

```
int String_CharIndex (char const text[], char searched) {
  char * found= String_FindChar (text, searched);
  if (found == NULL) return -1; // guard clause for exceptional flow
  return found - text;          // regular flow at main indent level
}
```

Remarque : avec l'opérateur ternaire, on peut aussi écrire :

```
int String_CharIndex (char const text[], char searched) {
  char * found= String_FindChar (text, searched);
  return (found != NULL) ? found - text : -1;
}
```

Recherche d'un caractère et principe DRY (4/5)

L'équivalent de la fonction `FindChar()` existe en standard.
Elle s'appelle `strchr()` et est déclarée dans `<string.h>` :

```
$ man 3 strchr
```

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

NAME
       strchr, strrchr - locate character in string

SYNOPSIS
       #include <string.h>
       char *strchr (const char *s, int c);
       char *strrchr(const char *s, int c);

DESCRIPTION
       The  strchr() function returns a pointer to the first occurrence of the
       character c in the string s.
       The strrchr() function returns a pointer to the last occurrence of  the
       character c in the string s.
       Here "character" means "byte"; these functions do not work with wide or
       multibyte characters.

RETURN VALUE
       The strchr() and strrchr() functions return a pointer  to  the  matched
       character  or NULL if the character is not found.  The terminating null
       byte is considered part of the string, so that if  c  is  specified  as
       '\0', these functions return a pointer to the terminator.
```

Recherche d'un caractère et principe DRY (5/5)

Cependant `FindChar()` renvoie `NULL` si l'on cherche `'\0'`, alors que `strchr()` renvoie l'adresse du `'\0'` dans ce cas.
De sorte que `strchr()` correspond au code ci-dessous :

```
char * strchr (char const text[], char searched) {
  for (int k= 0; /* no condition here */; k++) {
    if (text[k] == searched) return (char *) text + k;
    if (text[k] == '\0')     return NULL;
  }
}
```

Donc, si l'on réutilise `strchr()` pour écrire nos 3 fonctions :

```
char * String_FindChar (char const text[], char searched) {
  if (searched == '\0') return NULL; // additional guard clause
  return strchr (text, searched);
}
```

```
bool String_ContainsChar (char const text[], char searched) { // unchanged
  char * found= String_FindChar (text, searched);
  return found != NULL;
}
```

```
int String_CharIndex (char const text[], char searched) { // unchanged
  char * found= String_FindChar (text, searched);
  return (found != NULL) ? found - text : -1;
}
```

En conclusion, 2 principes à retenir : DRY et DOT (1/2)

DRY et DOT sont les deux principes fondamentaux de la prog.

DRY : Don't Repeat Yourself.
Éliminer les redondances, les répétitions de mise en oeuvre,
en réutilisant des fonctions au lieu de recoder ce qu'elles font déjà.

DOT : Do One Thing.
Une fonction ne doit faire qu'une seule chose,
articuler des concepts de niveaux d'abstraction inférieurs,
et non pas définir également la mise en oeuvre de ceux-ci,
qui doivent faire l'objet de leur propres fonctions.

Tous deux reposant sur l'extraction fonctionnelle,
Ils sont impossibles à respecter sans méthodologie de nommage,
l'identificateur de chaque fonction exprimant son but unique.

En conclusion, 2 principes à retenir : DRY et DOT (2/2)

Parfois, lorsqu'une fonction viole le principe DOT,
les symptômes d'une violation de DRY apparaît également,
car le niveau d'abstraction non-exprimé se trouve alors dupliqué :

```
bool GregorianYear_IsLeap (int year) {
  if (year %   4 != 0) return false;
  if (year % 100 != 0) return true;
  if (year % 400 != 0) return false;
  return true;
}
```

L'absence du prédicat testant si un entier est multiple d'un autre
est ici révélé par la duplication de sa mise en oeuvre.
La violation de DRY révèle donc ici la violation de DOT :

```
bool Int_IsMultipleOf (int number, int factor) {
  return number % factor == 0;
}
```

```
bool GregorianYear_IsLeap (int year) {
  if ( ! Int_IsMultipleOf (year,   4)) return false;
  if ( ! Int_IsMultipleOf (year, 100)) return true;
  if ( ! Int_IsMultipleOf (year, 400)) return false;
  return true;
}
```