# Programmation en C
# Généricité
# Pointeur `void*` et pointeurs sur fonction

Régis Barbanchon

L1 Info-Math, Semestre 2

# Échange de deux valeurs entières

Supposons que l'on souhaite échanger deux entiers :

```
int main (void) {
  int a= 555, b= 666;
  Int_Swap (& a, & b);
  assert (a == 666 && b == 555);
  return 0;
}
```

alors la fonction d'échange `Int_Swap()` s'écrit comme :

```
void Int_Swap (int * self, int * other)
{
  int self_value= * self;
  * self= * other;
  * other= self_value;
}
```

# Échange de deux valeurs flottantes

Supposons maintenant que l'on souhaite échanger deux flottants :

```
int main (void) {
  double x= 5.5, y= 6.6;
  Double_Swap (& x, & y);
  assert (Double_Equals (x, 6.6) && Double_Equals (y, 5.5));
  return 0;
}
```

où le prédicat `Double_Equals()` est défini comme :

```
#define EPSILON 0.0000001

bool Double_IsZero (double self) {
  return fabs (self) <= EPSILON;
}

bool Double_Equals (double self, double other) {
  return Double_IsZero (self - other);
}
```

alors la fonction d'échange `Double_Swap()` s'écrit comme :

```
void Double_Swap (double * self, double * other) {
  double self_value= * self;
  * self= * other;
  * other= self_value;
}
```

# Échange de deux positions dans une grille

Supposons enfin que l'on souhaite échanger deux positions :

```
typedef struct Pos { int line, col; } Pos;

Pos Pos_Make (int line, int col) {
  return (Pos) { .line= line, .col= col };
}
```

```
int main (void) {
  Pos p= Pos_Make (1, 2), q= Pos_Make (3, 4);
  Pos_Swap (& p, & q);
  assert (Pos_Equals (p, Pos_Make (3, 4)) && Pos_Equals (q, Pos_Make (1, 2)));
  return 0;
}
```

où le prédicat `Pos_Equals()` est défini comme :

```
bool Pos_Equals (Pos self, Pos other) {
  return self.line == other.line && self.col == other.col;
}
```

alors la fonction d'échange `Pos_Swap()` s'écrit comme :

```
void Pos_Swap (Pos * self, Pos * other) {
  Pos self_value= * self;
  * self= * other;
  * other= self_value;
}
```

# Les trois fonctions sont essentiellement identiques

Elles font la même chose sur des types différents :

```
void Int_Swap (int * self, int * other)
{
  int self_value= * self;
  * self= * other;
  * other= self_value;
}
```

```
void Double_Swap (double * self, double * other) {
  double self_value= * self;
  * self= * other;
  * other= self_value;
}
```

```
void Pos_Swap (Pos * self, Pos * other) {
  Pos self_value= * self;
  * self= * other;
  * other= self_value;
}
```

Tentons maintenant d'écrire une fonction d'échange `Data_Swap()` qui soit générique (c-à-d, acceptant n'importe quelle donnée).

# Échange générique de deux données (1/4)

Échanger deux données revient à échanger leurs bytes.
`Data_Swap()` doit alors connaître le nombre de bytes à échanger.
Celui-ci doit être transmis via un argument supplémentaire :

```
int main (void) {
  int a= 55, b= 66;
  Data_Swap (& a, & b, sizeof (int));
  assert (a == 66 && b == 55);

  double x= 5.5, y= 6.6;
  Data_Swap (& x, & y, sizeof (double));
  assert (Double_Equals (x, 6.6) && Double_Equals (y, 5.5));

  Pos p= Pos_Make (1, 2), q= Pos_Make (3, 4);
  Data_Swap (& p, & q, sizeof (Pos));
  assert (Pos_Equals (p, Pos_Make (3, 4)) && Pos_Equals (q, Pos_Make (1, 2)));

  return 0;
}
```

Les types des deux 1$^{ers}$ arguments changeant selon les invocations, on doit les transmettre via le type pointeur générique `void *`, et le prototype de `Data_Swap()` est donc :

```
void Data_Swap (void * self, void * other, int nb_bytes);
```

# Échange générique de deux données (2/4)

Si `Byte_Swap()` est la fonction d'échange de deux bytes :

```
void Byte_Swap (unsigned char * self, unsigned char * other) {
  unsigned char self_value= * self;
  * self= * other;
  * other= self_value;
}
```

alors la fonction `Data_Swap()` s'écrit donc comme :

```
void Data_Swap (void * self, void * other, int nb_bytes) {
  unsigned char * self_bytes = self;
  unsigned char * other_bytes= other;
  for (int k= 0; k < nb_bytes; k++) {
    Byte_Swap (self_bytes + k, other_bytes + k);
  }
}
```

**Remarque :** On a besoin de l'arithmétique de pointeurs.
Celle-ci n'existant pas sur les pointeurs `void *`,
on les a convertis en `unsigned char *`.

# Échange générique de deux données (3/4)

**Pas besoin de cast** dans les conversions suivantes :

```
void Data_Swap (void * self, void * other, int nb_bytes) {
  unsigned char * self_bytes = self;      // void * converted to unsigned char *
  unsigned char * other_bytes= other;     // void * converted to unsigned char *
  for (int k= 0; k < nb_bytes; k++) {
    Byte_Swap (self_bytes + k, other_bytes + k);
  }
}

int main (void) {
  int a= 55, b= 66;
  Data_Swap (& a, & b, sizeof (int));  // int * converted to void *
  return 0;
}
```

En effet :

- On peut convertir **sans cast** un pointeur typé en `void *` :

```
int int_value= 555;
int   *  int_ptr= & int_value;
void * void_ptr= int_ptr;   // (void *) int_ptr
```

- Inversement, on peut reconvertir **sans cast** le `void *` obtenu en le pointeur typé d'origine, ou en un pointeur sur byte :

```
int   * int_ptr2= void_ptr;  // (int *) void_ptr
char * byte_ptr= void_ptr;  // (char*) void_ptr
```

# Échange générique de deux données (4/4)

On peut écrire des wrappers autour de `Data_Swap()` afin de :

- contraindre les deux $1^{ers}$ arguments à avoir le même type ;
- éliminer le $3^{ème}$ argument technique (la taille en bytes).

```
void Int_Swap (int * self, int * other) {
  Data_Swap (self, other, sizeof * self);
}
```

```
void Double_Swap (double * self, double * other) {
  Data_Swap (self, other, sizeof * self);
}
```

```
void Pos_Swap (Pos * self, Pos * other) {
  Data_Swap (self, other, sizeof * self);
}
```

**Remarque :**

En utilisant `sizeof expr` au lieu de `sizeof(TypeName)`,
on obtient toujours l'expression uniforme `sizeof * self`
au lieu de `sizeof(int)`, `sizeof(double)`, et `sizeof(Pos)`.

# Autre exemple : renversement de tableau (1/5)

Supposons que l'on souhaite renverser un tableau d'entiers :

```
int main (void)
{
# define NB_INTEGERS 3
  int integers [NB_INTEGERS]= { 111, 222, 333 };
  ArrayOfInt_Reverse (integers, NB_INTEGERS);

  assert (integers[0] == 333  &&
          integers[1] == 222  &&
          integers[2] == 111);
  return 0;
}
```

Et de même, avec un tableau de flottants :

```
int main (void)
{
# define NB_REALS 3
  double reals [NB_REALS]= { 1.1, 2.2, 3.3 };
  ArrayOfDouble_Reverse (reals, NB_REALS);

  assert (Double_Equals (reals[0], 3.3)  &&
          Double_Equals (reals[1], 2.2)  &&
          Double_Equals (reals[2], 1.1) );
  return 0;
}
```

# Autre exemple : renversement de tableau (2/5)

La fonction `ArrayOfInt_Reverse()` s'écrit :

```
void ArrayOfInt_Reverse (int array[], int length)
{
  for (int left= 0, right= length-1;
       left < right;
       left++, right--) {
    Int_Swap (& array [left], & array [right]);
  }
}
```

De même, la fonction `ArrayOfDouble_Reverse()` s'écrit :

```
void ArrayOfDouble_Reverse (double array[], int length)
{
  for (int left= 0, right= length-1;
       left < right;
       left++, right--) {
    Double_Swap (& array [left], & array [right]);
  }
}
```

Écrivons une fonction de renversement `ArrayOfData_Reverse()` qui soit générique (c-à-d, acceptant tout type de tableau).

# Autre exemple : renversement de tableau (3/5)

On doit pouvoir écrire :

```
int main (void) {
# define NB_INTEGERS 3
  int integers [NB_INTEGERS]= { 111, 222, 333 };
  ArrayOfData_Reverse (integers, NB_INTEGERS, sizeof integers[0]);
  ...
}
```

```
int main (void) {
# define NB_REALS 3
  double reals [NB_REALS]= { 1.1, 2.2, 3.3 };
  ArrayOfData_Reverse (reals, NB_REALS, sizeof reals[0]);
  ...
}
```

Le 3$^{\text{ème}}$ argument est la taille de chaque case, exprimée en bytes.

Le type du 1$^{\text{er}}$ argument changeant selon les invocations,
il doit être de type pointeur générique `void *`.
`ArrayOfData_Reverse()` doit donc avoir le prototype suivant :

```
void ArrayOfData_Reverse (void * array, int length, int cell_size);
```

# Autre exemple : renversement de tableau (4/5)

En supposant que l'on sait obtenir l'adresse d'une case de tableau via la fonction `ArrayOfData_CellAt()`, le renversement s'écrit :

```
void ArrayOfData_Reverse (void * array, int length, int cell_size) {
  for (int left= 0, right= length-1;     left < right;     left++, right--) {
    void *  left_cell= ArrayOfData_CellAt (array,  left, cell_size);
    void * right_cell= ArrayOfData_CellAt (array, right, cell_size);
    Data_Swap (left_cell, right_cell, cell_size);
  }
}
```

L'adressage d'une case s'écrit comme :

```
void * ArrayOfData_CellAt (void const * array, int index, int cell_size) {
  unsigned char const * bytes= array;
  unsigned char const * cell_bytes= bytes + index * cell_size;
  return (void *) cell_bytes;
}
```

- Cette fonction promet de ne pas modifier `array`, donc ce paramètre est qualifié par `void const *`.
- On convertit ce paramètre en `unsigned char const *` afin d'effectuer l'arithmétique de pointeur.
- Le cast au retour permet de perdre le qualificateur `const`.

# Autre exemple : renversement de tableau (5/5)

Là encore, on peut écrire des wrappers spécifiques à chaque type autour de la fonction générique `ArrayOfData_Reverse()` :

```
void ArrayOfInt_Reverse (int array[], int length)
{
  ArrayOfData_Reverse (array, length, sizeof array[0]);
}
```

```
void ArrayOfDouble_Reverse (double array[], int length)
{
  ArrayOfData_Reverse (array, length, sizeof array[0]);
}
```

```
void ArrayOfPos_Reverse (Pos array[], int length)
{
  ArrayOfData_Reverse (array, length, sizeof array[0]);
}
```

**Remarque :**

En utilisant `sizeof expr` au lieu de `sizeof(TypeName)`, on obtient toujours l'expression uniforme `sizeof array[0]` au lieu de `sizeof(int)`, `sizeof(double)`, et `sizeof(Pos)`.

# Pointeurs sur fonctions (1/2)

Dans <math.h>, on trouve (entre autres) les prototypes suivant :

```
double sin   (double angle);  // sine
double cos   (double angle);  // cosine
double tan   (double angle);  // tangent
double ceil  (double number); // round to upper integer
double floor (double number); // round to lower integer
double round (double number); // round to nearest integer
```

Elles ont toutes le même type car :

- elles prennent toutes en argument un `double`
- et elle retournent toutes un `double`.

Nommons par `RealFunc` ce type de fonctions.
On peut définir ce type par la directive `typedef` suivante :

```
typedef double RealFunc (double arg);
```

On peut aussi définir `RealFuncPtr`, son type pointeur :

```
typedef double (* RealFuncPtr) (double arg); // direct definition
```

```
typedef RealFunc * RealFuncPtr; // definition via RealFunc
```

# Pointeurs sur fonctions (2/2)

- L'identificateur d'une fonction s'évalue en son adresse lorsqu'il n'est pas suivi d'une liste d'arguments.
- On peut stocker cette adresse dans une variable de type pointeur sur fonction de même type.
- On peut invoquer la fonction à travers la variable en faisant suivre son identificateur d'une liste d'arguments.

On peut donc stocker l'adresse de `floor()` ou `ceil()` dans une variable `my_round` de type `RealFunc *` ou `RealFuncPtr`, et invoquer ensuite la fonction à travers cette variable :

```
int main (void) {
  double number= 7.3;
  RealFunc * my_round;     // or:   RealFuncPtr my_round;
  my_round= floor; printf ("%f rounded to %f\n", number, my_round (number));
  my_round=  ceil; printf ("%f rounded to %f\n", number, my_round (number));
  return 0;
}
```

```
7.3 rounded to 7.0
7.3 rounded to 8.0
```

# Exemple avec un tableau de pointeurs sur fonctions

Ce programme utilise un tableau de pointeurs sur fonction pour faire passer des nombres par tous les modes d'arrondis :

```
int main (void) {
# define NB_NUMBERS 2
# define NB_FUNCS   3
  double numbers [NB_NUMBERS]= { 2.7182, 3.1415 };
  RealFunc * funcs [NB_FUNCS]= { floor, round, ceil };
  for (int number_id= 0; number_id < NB_NUMBERS; number_id++) {
    double number= numbers [number_id];
    for (int func_id= 0; func_id < NB_FUNCS; func_id++) {
       RealFunc * my_round= funcs [func_id];
       double rounded_number= my_round (number);
       printf ("%f rounded to %f\n", number, rounded_number);
    }
    printf ("\n");
  }
  return 0;
}
```

On obtient la sortie suivante :

```
2.7182 rounded to 2.0
2.7182 rounded to 3.0
2.7182 rounded to 3.0

3.1415 rounded to 3.0
3.1415 rounded to 3.0
3.1415 rounded to 4.0
```

# Pointeur sur fonction. . . en paramètre d'une fonction (1/2)

Chacune de ces trois fonctions prend un tableau de flottants et calcule leurs arrondis pour un mode d'arrondi particulier :

```
void ArrayOfDouble_Floor (double const array[], int length, double images[]) {
  for (int k= 0; k < length; k++)
    images [k]= floor (array[k]);
}
```

```
void ArrayOfDouble_Round (double const array[], int length, double images[]) {
  for (int k= 0; k < length; k++)
    images [k]= round (array[k]);
}
```

```
void ArrayOfDouble_Ceil (double const array[], int length, double images[]) {
  for (int k= 0; k < length; k++)
    images [k]= ceil (array[k]);
}
```

Exemple d'utilisation :

```
int main (void) {
# define NB_NUMBERS 2
  double rounded [NB_NUMBERS], numbers [NB_NUMBERS]= { 2.7182, 3.1415 };
  ArrayOfDouble_Ceil (numbers, NB_NUMBERS, rounded);
  assert (Double_Equals (rounded[0], 3.0));
  assert (Double_Equals (rounded[1], 4.0));
  return 0;
}
```

# Pointeur sur fonction. . . en paramètre d'une fonction (2/2)

On peut généraliser ces trois fonctions en une seule.
Celle-ci prend en arg supplémentaire un pointeur sur fonction :

```
void ArrayOfDouble_Map (double const array[], int length,
                        RealFunc * func, double images[]) {
  for (int k= 0; k < length; k++)
    images [k]= func (array [k]);
}
```

Exemple d'utilisation de la fonction ainsi généralisée :

```
int main (void) {
# define NB_NUMBERS 2
  double rounded [NB_NUMBERS], numbers [NB_NUMBERS]= { 2.7182, 3.1415 };

  ArrayOfDouble_Map (numbers, NB_NUMBERS, floor, rounded);
  assert (Double_Equals (rounded[0], 2.0));
  assert (Double_Equals (rounded[1], 3.0));

  ArrayOfDouble_Map (numbers, NB_NUMBERS, round, rounded);
  assert (Double_Equals (rounded[0], 3.0));
  assert (Double_Equals (rounded[1], 3.0));

  ArrayOfDouble_Map (numbers, NB_NUMBERS, ceil, rounded);
  assert (Double_Equals (rounded[0], 3.0));
  assert (Double_Equals (rounded[1], 4.0));

  return 0;
}
```

# Pointeur sur fonction. . . en retour d'une fonction

Une fonction peut retourner un pointeur sur fonction.

Voici par exemple un mode d'arrondi fantaisiste de notes,
qui varie en fonction de l'humeur du prof et du sexe de l'étudiant :

```
RealFunc * ExamRounding (bool teacher_is_happy, bool student_is_girl)
{
  return student_is_girl ? ceil
     : teacher_is_happy ? round
     : floor;
}
```

Et son utilisation pour un garçon noté par un prof grognon :

```
int main (void)
{
# define NB_MARKS 3
  double boy_marks [NB_MARKS]= { 9.4, 10.2, 12.9 };

  RealFunc * my_rounding= ExamRounding (false, false);
  ArrayOfDouble_Map (boy_marks, NB_MARKS, my_rounding, boy_marks);

  assert (Double_Equals (boy_marks[0],  9.0));
  assert (Double_Equals (boy_marks[1], 10.0));
  assert (Double_Equals (boy_marks[2], 12.0));
  return 0;
}
```

# Pointeur sur fonction prenant `void*` en argument (1/5)

Considérons le prédicat suivant `ArrayOfInt_IsSorted()` testant si un tableau d'entiers est trié par ordre croissant :

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

ainsi que son analogue `ArrayOfInt_IsReverseSorted()` testant si un tableau d'entiers est trié par ordre décroissant :

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

```
int main (void) {
  int a[]= { 1,2,3 }, b[]= { 3,2,1 }, c[]= { 1,2,1 }, d[]= { 2,2,2 };
  assert (   ArrayOfInt_IsSorted (a,3) && ! ArrayOfInt_IsReverseSorted (a,3));
  assert ( ! ArrayOfInt_IsSorted (b,3) &&   ArrayOfInt_IsReverseSorted (b,3));
  assert ( ! ArrayOfInt_IsSorted (c,3) && ! ArrayOfInt_IsReverseSorted (c,3));
  assert (   ArrayOfInt_IsSorted (d,3) &&   ArrayOfInt_IsReverseSorted (d,3));
  return 0;
}
```

# Pointeur sur fonction prenant `void*` en argument (2/5)

Une première généralisation de ces fonctions consiste à introduire le type de fonction `IntPairTest`, prédicat sur deux entiers :

```
typedef bool IntPairTest (int left, int right);
```

. . . ainsi que les deux tests d'ordre de ce type :

```
bool Int_IsLessOrEqual    (int self, int other) { return self <= other; }
bool Int_IsGreaterOrEqual (int self, int other) { return self >= other; }
```

On peut alors écrire la généralisation suivante :

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

. . . ainsi que ses deux wrappers pour les tests de tri :

```
bool ArrayOfInt_IsSorted (int const array [], int length) {
  return ArrayOfInt_HoldsForConsecutive (array, length, Int_IsLessOrEqual);
}

bool ArrayOfInt_IsReverseSorted (int const array [], int length) {
  return ArrayOfInt_HoldsForConsecutive (array, length, Int_IsGreaterOrEqual);
}
```

# Pointeur sur fonction prenant `void*` en argument (3/5)

On peut imaginer la même problématique pour les flottants.
On introduit alors le type de fonction `DoublePairTest` :

```
typedef bool DoublePairTest (double left, double right);
```

. . . ainsi que les deux tests d'ordre de ce type :

```
bool Double_IsLessOrEqual   (double self, double other){ return self <= other; }
bool Double_IsGreaterOrEqual(double self, double other){ return self >= other; }
```

On peut alors écrire la généralisation analogue à la précédente :

```
bool ArrayOfDouble_HoldsForConsecutive (double const array [], int length,
                                        DoublePairTest * test) {
  for (int k= 1; k < length; k++)
    if ( ! test (array [k-1], array [k]) ) return false;
  return true;
}
```

. . . ainsi que ses deux wrappers pour les tests de tri :

```
bool ArrayOfDouble_IsSorted (double const array [], int length) {
  return ArrayOfDouble_HoldsForConsecutive (array, length, Double_IsLessOrEqual);
}

bool ArrayOfDouble_IsReverseSorted (double const array [], int length) {
  return ArrayOfDouble_HoldsForConsecutive (array, length, Double_IsGreaterOrEqual);
}
```

# Pointeur sur fonction prenant `void*` en argument (4/5)

Réunissons ces problématiques (tests de tri croissant/décroissant, sur entiers/sur flottants) à l'intérieur d'une même généralisation.

Le prédicat généralisé sur les paires devient `DataPairTest` :

```
typedef bool DataPairTest (void const * left, void const * right);
```

Il se décline sur les paires d'entiers et de flottants en :

```
bool Data_IsLessOrEqualAsInt (void const * left, void const * right) {
  int const * self= left, * other= right;
  return * self <= * other;
}

bool Data_IsGreaterOrEqualAsInt (void const * left, void const * right) {
  int const * self= left, * other= right;
  return * self >= * other;
}
```

```
bool Data_IsLessOrEqualAsDouble (void const * left, void const * right) {
  double const * self= left, * other= right;
  return * self <= * other;
}

bool Data_IsGreaterOrEqualAsDouble (void const * left, void const * right) {
  double const * self= left, * other= right;
  return * self >= * other;
}
```

# Pointeur sur fonction prenant `void*` en argument (5/5)

La fonction générique de test de tri s'écrit alors :

```
bool ArrayOfData_HoldsForConsecutive (void const * array, int length,
                                      DataPairTest * test, int cell_size) {
  for (int k= 1; k < length; k++) {
    void const * left = ArrayOfData_CellAt (array, k-1, cell_size);
    void const * right= ArrayOfData_CellAt (array, k  , cell_size);
    if ( ! test (left, right) ) return false;
  }
  return true;
}
```

et ses quatre wrappers s'écrivent :

```
bool ArrayOfInt_IsSorted (int const array[], int length) {
  return ArrayOfData_HoldsForConsecutive (array, length,
    Data_IsLessOrEqualAsInt, sizeof array[0]);
}
bool ArrayOfInt_IsReverseSorted (int const array[], int length) {
  return ArrayOfData_HoldsForConsecutive (array, length,
    Data_IsGreaterOrEqualAsInt, sizeof array[0]);
}
bool ArrayOfDouble_IsSorted (double const array[], int length) {
  return ArrayOfData_HoldsForConsecutive (array, length,
    Data_IsLessOrEqualAsDouble, sizeof array[0]);
}
bool ArrayOfDouble_IsReverseSorted (double const array[], int length) {
  return ArrayOfData_HoldsForConsecutive (array, length,
    Data_IsGreaterOrEqualAsDouble, sizeof array[0]);
}
```

# La fonction qsort() de <stdlib.h> (1/2)

La fonction de tri qsort() de la librairie standard est générique :

```
$ man 3 qsort
```

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

NAME
       qsort - sort an array

SYNOPSIS
       #include <stdlib.h>

       void qsort (void *base, size_t nmemb, size_t size,
                   int (*compar)(const void *, const void *));

DESCRIPTION
       The  qsort()  function sorts an array with nmemb elements of size size.
       The base argument points to the start of the array.

       The contents of the array are sorted in ascending order according to  a
       comparison  function  pointed  to  by  compar, which is called with two
       arguments that point to the objects being compared.

       The comparison function must return an integer less than, equal to,  or
       greater  than  zero  if  the first argument is considered to be respec-
       tively less than, equal to, or greater than the second.  If two members
       compare as equal, their order in the sorted array is undefined.

RETURN VALUE
       The qsort() function returns no value.
```

# La fonction qsort() de <stdlib.h> (2/2)

qsort() requiert une fonction de comparaison générique :

```
int Data_CompareAsInt (void const * left, void const * right) {
  int const * self = left;
  int const * other= right;
  return (* self < * other) ? -1
    :    (* self > * other) ? +1
    : 0;
}
```

On peut alors écrire un wrapper pour le tableau du type désiré :

```
void ArrayOfInt_Sort (int array[], int length) {
  qsort (array, length, sizeof array[0], Data_CompareAsInt);
}
```

Pour conclure, voici un exemple d'utilisation :

```
int main (void) {
# define NB_VALUES 7
  int values[NB_VALUES]= { 5, 2, 3, 1, 2, 7, 9 };
  ArrayOfInt_Sort (values, NB_VALUES);
  assert (ArrayOfInt_IsSorted (values, NB_VALUES));
  for (int k= 0; k < NB_VALUES; k++) printf ("%d ", values [k]);
  printf ("\n");
  return 0;
}
```

```
1 2 2 3 5 7 9
```