Langage de programmation SAFE-C

Tutorial pour développeurs connaissant déjà le C

Voici un programme Safe-C :

```
// date.c

from std use calendar, console;

void main()
{
  DATE_TIME now;

  get_datetime (out now);

  printf ("Nous sommes le %02d/%02d/%04d ", now.day, now.month, now.year);
  printf ("et il est %02d:%02d:%02d.\n", now.hour, now.min, now.sec);
}
```

Compilation

Comme le C, un programme Safe-C utilise les extensions de fichier .h pour les interfaces et .c pour les corps de programme. Le make est intégré au compilateur : il suffit de donner au compilateur le nom du fichier .c principal et il suit automatiquement les chemins des inclusions de libraries (from xxx use yyy;) ou de fichiers locaux (use yyy;).

Initialisation des variables

Toutes les variables doivent être initialisées lors de la première utilisation, y compris les tableau et les structures qui doivent recevoir une première valeur complète. Cela peut se faire soit avec l'instruction **clear** qui remplace le "**memset (&v, 0x00, sizeof(v));**" du C, ou par une assignation d'un aggregat complet tel que :

```
void main()
{
  int tab[3];

  clear tab;          // tous les éléments à 0

  tab = {all => 5};   // tous les éléments à 5

  tab = {5, 6, 7};    // assignation d'un aggregat
}
```

Les structures doivent être initialisées de manière identique :

```
void main()
{
  struct KEY
  {
    int  nr;
```

```
    char c;
  }

  KEY k;

  clear k;              // tous les éléments à 0

  k = {1, 'a'};         // aggregat simple

  k = {nr=>1, c=>'a'};  // aggregat avec noms
}
```

Types de données

Voici un aperçu rapide des types de données :
- les entiers : signés: int1, int2, int4 et int8; non-signés: uint1, uint2 et uint4; et leur synonymes: tiny, short, int, long, byte, ushort et uint.
- les types énumération : char, wchar, bool.
- les types flottant : float et double.
- les types tableau, struct, union, pointeur safe(^) ou unsafe(*), pointeur vers fonction, opaque, générique.

Les tableaux se déclarent comme en C, avec cependant une subtilité :

```
void main()
{
  char     t1[10], t2[10];
  char[10] t1, t2;
}
```

Les deux lignes de déclarations ci-dessus sont identiques car ce qui est spécifié à gauche avec le type s'applique à tous les identifiants de droite. On peut combiner les deux syntaxes.

On peut donner un nom à un tableau de longueur non-précisée, ainsi **string** est prédéfini comme :

```
typedef char[] string;
```

ce qui a pour effet que les déclarations suivantes sont toutes identiques :

```
void main()
{
  char[100]   buffer1;
  string(100) buffer2;
  char        buffer3[100];
  string      buffer4(100);
}
```

Passage de Paramètres

Il y a 3 modes de passage pour les paramètres:
- in (read-only; les types simples sont passés par valeur, les tableaux et struct par adresse)
- ref (par adresse)

- out (par adresse aussi mais le paramètre est considéré comme non-initialisé au départ et doit recevoir une valeur complète avant chaque fin de fonction possible)

```
void foo (int i, ref int j, out int k)
{
  k = i + j;
}
```

L'appel de fonction se fait comme ceci, en répétant le mode :

```
void main ()
{
  int i, j, k;

  i = 1;
  j = 2;

  foo (i, ref j, out k);
}
```

Donc vous voyez que contrairement au C, on n'utilise aucun symbole & ou *.

Les tableaux sont passés comme ceci :

```
void foo1 (char[10] str);
void foo2 (char[] str);
void foo3 (string str);
```

La fonction foo1 accepte uniquement les tableaux de char de longueur 10 : à l'exécution on ne passe qu'une adresse sur la pile.

La fonction foo2 accepte des tableaux de char de n'importe quelle longueur : à l'exécution on passe sur la pile l'adresse ainsi qu'un champ longueur ce qui permet d'interroger la longueur du tableau à l'intérieur de la fonction.

La fonction foo3 est équivalente à foo2 mais plus agréable à lire.

les attributs

L'attribute 'length permet de prendre la longueur de n'importe quel tableau :

```
void main()
{
  char tab[3];
  int  i;

  i = tab'length;   // 3
```

Les attributes 'min et 'max permettent de prendre le minimum/maximum d'un entier.

```
void main()
{
  int i, petit, grand;

  petit = i'min;   // -2_147_483_648
```

```
  grand = i'max;    // +2_147_483_647
}
```

Les attributes 'first et 'last permettent de prendre la première/dernière valeur d'un type énumération.

```
void main()
{
  enum COLOR {RED, GREEN, BLUE};

  COLOR a, b;

  a = COLOR'first;    // RED
  b = a'last;         // BLUE
}
```

L'attribut 'string permet de convertir une valeur énumération en string représentant son litéral, ce qui est assez utile quand on ajoute des printf lors du débugging ...

```
void main()
{
  enum COLOR {RED, GREEN, BLUE};
  COLOR c = RED;
  printf ("c = %s\n", c'string);
  printf ("première couleur = %s\n", COLOR'first'string);
}
```

Les slices

Un slice est une tranche de tableau avec un début et une longueur :

```
  string(5) s;
  string(2) t;

  s = "Hello";
  t = s[3:2];         // copie "lo" (début=3, longueur=2)
  s[1:4] = "ELLO";    // garde le H majuscule mais change le reste
```

Les indices et slices de tableaux sont vérifiés et génèrent une erreur fatale en cas de dépassement.

Les strings

Le composant **'strings'** contient les fonctions bien connues : strcpy, strcat, sprintf, etc ..
Il est à noter que, contrairement au C, le caractère nul qui termine les strings est optionnel. Donc vous pouvez faire un strcpy() de "Hello" dans un string de longueur 5, il n'y aura pas de caractère de terminaison nul dans ce cas.

```
from std use strings;

void main()
{
  string(64) str, str2;
  int        i = 2, j = 3, len;
```

```
  sprintf (out str, "la valeur de i est : %d", i);
  sprintf (out str2, " et j vaut : %d", j);
  strcat (ref str, str2);
  len = strlen (str);
}
```

Si vous faites un strcpy d'un string de longueur 6 dans un tableau de 5, le programme s'arrêtera sur une erreur fatale.

Les constantes

Les déclarations C suivantes :

```
  #define MAX   100
  #define TITLE "programme.c"
```

s'écriront en Safe-C :

```
  const int    MAX   = 100;
  const string TITLE = "programme.c";
```

Les tableaux 'jagged'

La déclaration C suivante :

```
  char *table[] = {"This", "is", "an", "example"};
```

s'écrira en Safe-C :

```
  const string table[] = {"This", "is", "an", "example"};
```

On peut connaitre le nombre de strings par table'length.

Les struct

Les struct se déclarent quasiment comme en C :

```
  struct PERSON
  {
    char[20]  name;
    int       age;
  }

  PERSON per;
```

Il existe en outre des structs avec un discriminant de type énumération :

```
    enum TypeShape {POINT, SQUARE, CIRCLE, TRIANGLE};

    struct Shape (TypeShape kind)
    {
      int x, y;

      switch (kind)
      {
```

```
      case POINT:
        null;

      case SQUARE:
        int side;

      case CIRCLE:
        int radius;

      case TRIANGLE:
        int base, height;
    }
  }

  Shape(SQUARE) s = {x=>1, y=>2, side=>3};
```

La taille de la structure dépend de la valeur du discriminant lors de la création de la variable. On ne réserve donc pas la taille maximum mais uniquement la taille pour la variante donnée.

Les structures variantes peuvent être passées en paramètre :

```
void foo1 (Shape(POINT) p)
{
  // ...
}

void foo2 (Shape s)
{
  switch (s.kind)
  {
    case POINT:
      // ...
      break;
  }
}
```

La fonction foo1 n'accepte que les Shape de type POINT, alors que foo2 accepte n'importe quelle variante.
foo2 reçoit le discriminant s.kind dans un paramètre caché et peut donc savoir de quelle variante il s'agit.

Les types paqués

```
packed struct PERSON
{
  char[20]  name;
  int       age;
}

PERSON per;
```

Le mot-clé **packed** indique au compilateur de ne pas aligner les champs de la structure. La structure devient donc 'portable' et peut être passée par une fonction d'entrée-sortie vers l'extérieur. Une structure packed ne peut pas contenir de pointeur ^ (sinon on pourrait lire une valeur quelconque de pointeur à partir du disque dur et corrompre la mémoire).
Il y a une règle qui convertit implicitement tous les types paqués en tableau de byte lors du passage par paramètre.

read() et write() étant déclarés comme ceci :

```
  int read (int fd, out byte[] buffer);
  int write (int fd, byte[] buffer);
```

On peut donc écrire ceci :

```
  rc = read (fd, out per);
ou
  rc = write (fd, per);
```

Par ailleurs, n'importe quelle variable peut être convertie en tableau de byte grâce à l'attribut 'byte :

```
  byte tab[4];
  float f = 1.2;

  tab = f'byte;   // copie 4 bytes
```

ce qui permet de copier le contenu de n'importe quel variable vers n'importe quelle autre variable (sauf si la type de la variable contient un pointeur safe^, ceux-là étant exclus de ces conversions) :

```
  int   i;
  float f = 1.2;

  i'byte      = f'byte;       // copie 4 bytes
  i'byte[0]   = f'byte[0];    // copie le premier byte
  i'byte[2:2] = f'byte[2:2];  // copie les 2 derniers bytes
```

Le type object

Le type **object** est prédéfini comme un tableau de byte :

```
    typedef byte[] object;    // open array of byte
```

Le type **object**[] est utilisé dans la définition de fonctions ayant un nombre variable de paramètres, telles que celles-ci :

```
  int sprintf (out string buffer, string format, object[] arg);
  int sscanf  (string buffer,     string format, out object[] arg);
```

Dans le corps de ces fonctions, on peut connaitre le nombre de paramètres par arg'length, et chaque paramètre est accessible par arg[i] et a le type d'un tableau de byte. En fonction du string de format, il suffit alors à la fonction de le convertir vers le type souhaité à l'aide de l'attribut 'byte.

Les références

Une référence permet de renommer une variable en un nom plus court.
En pratique une référence désigne l'adresse de la variable.

```
  ref string s = p^.line[i]^;
  printf ("%s\n", s);
```

Les types pointeurs

Un pointeur est déclaré par le symbole ^. Le mot-clé **new** permet d'allouer des variables de taille dynamique sur le heap, avec ou sans spécification d'une valeur initiale.

Il y a les types simples :

```
int^  p  = new int;         // objet initialisé à zéro
int^  p2 = new int ' (1);   // initialisation explicite à 1.
```

ou

```
struct NODE
{
  int   nr;
  NODE^ next;
}

NODE^ p  = new NODE;              // objet initialisé à zéro
NODE^ p2 = new NODE ' {1, null};  // initialisation explicite par aggregate
NODE^ p3 = new NODE ' (p^);       // initialisé avec valeur d'un autre objet
```

Il y a deux types d'objet tableau : ceux à longueur constante (qui ont une constante spécifiée dans la déclaration du pointeur) :

```
int[3]^  p = new int[3];           // pointe toujours vers un tableau de longueur 3.
int[3]^  q = new int ' {1, 2, 3};  // idem
```

et ceux à longueur dynamique (qui n'ont aucune longueur spécifiée dans la déclaration du pointeur) :

```
int[]^  p = new int[3];
int[]^  q = new int ' {1, 2, 3};
```

remarquez bien la différence : les derniers ont un champ longueur stocké dans un header au début de l'objet heap; ils ne sont pas compatibles avec les premiers.

Finalement il y a les structures à discriminant contenant la valeur du discriminant dans un header au début de l'objet heap:

```
Shape^  p = new Shape(POINT);
Shape^  q = new Shape(POINT) ' {x=>1, y=>2};
Shape^  r = new Shape ' (q^);
```

Implementation des types pointeurs

Le type pointeur avec ^ est sécurisé par un mécanisme de 'tombstone' : chaque pointeur pointe vers une structure interne appelée Tombstone qui contient un pointeur vers le vrai objet alloué sur le heap ainsi qu'un compteur de références. Ce mécanisme, géré de manière thread-safe, permet de prévenir toute opération qui risquerait de corrompre la mémoire.

Si le système est à court de mémoire lors d'un **new** le programme s'arrête sur une erreur fatale exactement comme quand votre pile est pleine suite à un trop grand nombre d'appels récursifs. C'est à vous à gérer votre consommation mémoire. Tout comme en C, tout block mémoire alloué avec **new** doit être libéré avec l'instruction **free** après utilisation.

```
  free p;
  free q;
```

L'utilisation de **free** sur un objet toujours référencé par n'importe quel thread ou déjà libéré par **free** précédemment va génèrer une erreur fatale.

En revanche, le langage ne vous préviendra pas si vous oubliez des **free**, car cela ne corrompt pas la mémoire.

Les pointeurs vers fonction

Les pointeurs vers fonction existent comme en C, sans surprise. Voici un exemple :

```
  void treat_node (Shape s);     // function declaration

  typedef void TREAT (Shape s);  // function pointer type

  void treatment ()
  {
    TREAT treat;    // function pointer variable

    treat = null;
    treat = treat_node;   // parameter modes and types must match

    if (treat != null)
      treat (s);
  }
```

Les pointeurs unsafe

Pour interfacer les libraries avec le système d'exploitation, les anciens pointeurs du C sont disponibles en Safe-C, avec notamment l'opérateur & pour prendre l'adresse d'un objet, l'indexation tableau d'un pointeur p[i], la sélection d'un champ p->field, ou bien les opérateurs ++ et -- sur un pointeur.
Tout cela n'est cependant disponible qu'à l'intérieur d'une section unsafe :

```
#begin unsafe
  const string filename = "Test\0";
  char *p = &filename;
  p++;
#end unsafe
```

Les threads

L'opérateur **run** permet de démarrer un thread de façon très simple.

```
  void mon_thread ()
  {
  }

  void main()
  {
    int rc;
```

```
  rc = run mon_thread ();    // starts a thread (rc: 0=OK, -1=error)
}
```

La fonction mon_thread peut avoir au maximum un paramètre.

Les types opaques

Les types opaques permettent de créer une forme très simplifiée de classe dans laquelle les champs d'une structure ne sont accessible que dans le fichier .c correspondant au .h où est déclaré le type opaque. En outre, toutes les opérations permettant de prendre une copie (clone) du type opaque sont interdites.

```
// drawing.h

struct DRAW_CONTEXT;     // opaque type

void init   (out DRAW_CONTEXT d);
void circle (ref DRAW_CONTEXT d, int x, int y, int radius);
```

```
// drawing.c

struct DRAW_CONTEXT      // full struct type
{
  int    x, y, dx, dy;
  IMAGE^ image;
}

void init (out DRAW_CONTEXT d)
{
  // ..
}

void circle (ref DRAW_CONTEXT d, int x, int y, int radius)
{
  // ..
}
```

```
// main.c

use drawing;

void main()
{
  DRAW_CONTEXT a, b;
  init (out a);
  b = a;        // ERROR : assignment not allowed for limited types
}
```

Les package génériques

Le Safe-C permet de déclarer des packages génériques, ce qui permet d'écrire des algorithmes qu'on peut instancier pour un type donné. C'est le même effet que les macros du C, sauf que le compilateur ne remplace pas mécaniquement le type ELEMENT par int; il vérifie tout le package syntaxiquement.

A noter qu'on peut aussi définir des packages non-génériques, ou des packages imbriqués.

Voici un exemple de tri à bulle qu'on instantie pour le type int :

```
// bubble.h

generic <ELEMENT>           // generic type ELEMENT
  int compare (ELEMENT a,
               ELEMENT b);  // return -1 if a<b, 0 if a==b, +1 if a>b
package BubbleSort
  void sort (ref ELEMENT table[]);
end BubbleSort;
```

```
// bubble.c

package body BubbleSort
  public void sort (ref ELEMENT table[])
  {
    int     i, j;
    ELEMENT temp;

    for (i=1; i<table'length; i++)
    {
      for (j=i; j>0; j--)
      {
        if (compare (table[j-1], table[j]) <= 0)
          break;

        temp       = table[j-1];
        table[j-1] = table[j];
        table[j]   = temp;
      }
    }
  }
end BubbleSort;
```

```
int compare_int (int a, int b)
{
  if (a < b) return -1;
  if (a > b) return +1;
  return 0;
}

package Sort_int = new BubbleSort (ELEMENT => int,
                                   compare => compare_int);

void main()
{
  int table[5] = {2, 19, 3, 9, 4};

  sort (ref table);    // must be written Sort_int.sort if ambiguous
}
```

Pour terminer

Pour terminer, voici encore quelques informations en bref :

- avec le type **wchar** sur 16-bits, le Safe-C supporte l'UTF-16 qui permet les caractères chinois, japonais, etc .. et cela même au niveau du code source, donc on peut écrire des strings constants japonais "".

- l'instruction **assert b;** permet de vérifier une assertion à la compilation ou à l'exécution;

- l'instruction **abort;** arrête le programme sur une erreur fatale;

- l'instruction **sleep n;** permet de suspendre un thread pendant une période donnée. sleep prend un argument int ou float qui exprime des secondes.

- l'instruction **_unused v;** permet de spécifier une variable inutilisée pour supprimer un warning;

- en cas d'erreur fatale, l'utilisation du composant **'exception'** permet de générer un fichier **CRASH-REPORT.TXT** permettant au programmeur de localiser l'erreur.

Voilà vous connaissez maintenant l'essentiel du langage Safe-C.

Tout le reste (opérateurs, instructions) devrait vous être familier si vous connaissez déjà le C.
