Manuel d'apprentissage pour le langage de programmation SAFE-C

Ce document décrit le langage de programmation SAFE-C créé par
Samuro pour les gens qui souhaitent apprendre le langage complet.

Résumé:
  le langage de programmation SAFE-C conserve la simplicité, la rapidité
  et le style du langage C, tout en modifiant légèrement les règles afin
  de sécuriser l'accès à la mémoire.

  Le but du Safe-C est que les programmes erronés soient stoppés immédiatement,
  soit à la compilation ou au point d'exécution de l'erreur, afin que la
  la correction soit facile et rapide. L'avantage est une baisse significative
  du coût de maintenance du produit logiciel et une augmentation de la
  satisfaction des clients.

Belgique, Mars 2011, version finale version 1.41, contact: marcsamu@hotmail.com

0. Concepts du Safe-C

Voici un récapitulatif de l'évolution de la famille du langage C :

1972 : C
1983 : C++
1995 : Java
2001 : C#

- C propose les concepts minimum de base pour écrire un programme.
- C++ ajoute les classes et les exceptions.
- Java and C# introduisent une nouvelle syntaxe, et les concepts de
  sécurité mémoire, machine virtuelle, et objets heap libérés automatiquement.

Plus on avance dans le temps, plus on constate que les langages de
programmation deviennent volumineux et complexes à comprendre. La plupart

des nouveaux concepts ont été introduits pour "des raisons de marketing", parce que d'autres langages qui sont en compétition "les proposent aussi".

Si vous jetez un oeil sur les programmes développés par les programmeurs les plus talentueux, par exemple ceux qui développent les systèmes d'exploitation Windows ou Unix, vous serez peut-être surpris d'apprendre qu'ils n'utilisent aucun de ces nouveaux langages. Oui, ils utilisent encore le C de 1972. Pourquoi ? Parce que le C donne la liberté aux programmeurs vraiment talentueux d'écrire les meilleurs programmes dont la vitesse n'est égalable par aucun autre langage actuel.

Oui, les programmeurs professionnels talentueux utilisent tous le C, parce que tous les concepts ajoutés par les autres langages "ne sont pas vraiment nécessaires" !

Alors pourquoi introduire le Safe-C, si le C est tellement meilleur ?

Le C n'est pas un langage strict : un compilateur C permet au programmeur de faire des bêtises sans donner de messages d'erreur. Plus tard, le programme peut alors "crasher" ou il peut avoir des défauts qui permettent aux hackers d'avoir accès à votre ordinateur.

Un programmeur C professionnel le sait, et il est prêt à prendre ce risque s'il peut avoir un programme plus rapide que ses concurrents.

Mais les choses ont changé depuis 1972 : les programmes sont devenus beaucoup plus grands qu'avant en nombre de lignes de code source, et même les programmeurs talentueux qui font très peu d'erreurs vont quand même laisser un certains nombre de bugs non-détectés dans chaque grand projet. Les programmes nécessitent alors des mises-à-jour fréquentes pour corriger ces bugs.

Le Safe-C souhaite supprimer drastiquement ces erreurs par des règles de langage plus strictes, mais sans ajouter des concepts compliqués comme d'autres langages le font, parce que nous voulons garder la rapidité, la simplicité et le style du C.

Le but du Safe-C est que les programmes erronés soient stoppés immédiatement, soit à la compilation ou au point d'exécution de l'erreur, afin que la la correction soit facile et rapide. L'avantage est une baisse significative du coût de maintenance du produit logiciel et une augmentation de la satisfaction des clients.

==========

1. Elements Lexicaux

1.1 Fichier Source

Le langage supporte des fichiers source en ANSI, ainsi qu'en encodage Unicode UTF-8 et UTF-16.

1.2 Commentaires

Les lignes de commentaire en une seule ligne commencent par "//" et continuent jusqu'à la fin de la ligne.
Les commentaires multi-lignes commencent par "/*" et se terminent par "*/".

Les commentaires multi-lignes ne s'imbriquent pas. Le compilateur peut générer

un avertissement si les caractères /* sont trouvés dans un commentaire.

Exemple:

```
  // ceci est un commentaire en une seule ligne

  /* ceci est un commentaire
     multi-ligne */
```

1.3 Identificateurs

Les identificateurs sont composés de lettres, chiffres et du caractère souligné.
Ils ne doivent pas commencer par un chiffre.

Exemple:

```
  montant_total
  MAX
  buffer2
```

Les lettres minuscules et majuscules ne sont pas équivalentes.

1.4 Littéraux entiers

Les littéraux entiers peuvent avoir une forme décimale, hexadécimale ou binaire.

Exemple:

```
  64           // compatible avec n'importe quel nombre entier signé ou non signé
  65_000_000   // souligné pour une meilleure lisibilité
  0xFF         // valeur hexadécimale
  0b1111_0000  // valeur binaire
  176L         // long
```

Le caractère souligné est autorisé dans un littéral entier pour une meilleure
lisibilité.

La valeur entière la plus négative (-2^63) n'est pas représentable par un litéral
entier,
l'expression long'min peut être utilisée à la place.

Si le littéral a un suffixe L, il est de type 'long'.

Les littéraux en octal du C ne sont pas supportés, le compilateur
émet un avertissement pour tout entier littéral qui commence par un chiffre
zéro suivi d'autres chiffres.

1.5 littéraux à virgule flottante

Un littéral à virgule flottante peut avoir les types suivants:
- Si un suffixe f ou F est présent, il a le type 'float',
- Si un suffixe d ou D est présent, il a le type 'double',
- Si aucun suffixe n'est présent, son type dépend du contexte.

Exemple:

```
  1.2f
  3.023_223E+2d
  0x80.0p-1
```

Un littéral hexadécimal avec préfixe 0x est utilisé pour une représentation

machine précise : la mantisse est spécifié dans la base 16,
et l'exposant est spécifié dans la base 10.

1.6 littéraux caractères

Les littéraux caractère sont des valeurs constantes de type char
(ou de type wchar s'ils ont un préfixe 'L').

Exemple:

```
  'A'
  'É'
  '\N'
  L'\x00FF' // préfixe 'L' indique le type wchar
```

Les séquences d'échappement suivantes sont supportées :

```
  \' 0x0027 Single quote
  \" 0x0022 Double quote
  \\ 0x005C Backslash
  \0 0x0000 Null
  \a 0x0007 Alert
  \b 0x0008 Backspace
  \f 0x000C Form feed
  \n 0x000A New line
  \r 0x000D Carriage return
  \t 0x0009 Horizontal tab
  \v 0x000B Vertical tab
```

```
hex_escape_sequence:
  \x  hex_digit  hex_digit                          (pour type char)
  \x  hex_digit  hex_digit  hex_digit  hex_digit    (pour type wchar)
```

1.7 littéraux string

Les littéraux string sont des valeurs constantes de type string
(ou de type wstring en cas de préfixe 'L').

Exemple:

```
  "A"
  "Hello\n"
  "Hello" + " World"
  L"Hallöchen"
  L"\x1234"
```

Les littéraux string de plus de 128 caractères génèrent une erreur de compilation,
des littéraux string plus longs peuvent être construits en utilisant l'opérateur
de concaténation '+'.

1.8 Mots-clés

Les mots-clés suivants sont réservés et ne peuvent pas être utilisés
comme identificateurs:

```
  #begin #define #elif #else #end #endif #error #if #warning
  Lnul

  _asm  abort  assert
  body  bool  break  byte
```

```
  case  char  clear  const  continue
  default  double
  else  end  enum
  false  float  for  free  from
  generic
  if  inline  int  int1  int2  int4  int8
  long
  new  nul  null
  object  out
  package  packed  public
  ref  return  run
  short  sleep  string  struct  switch
  tiny  true  typedef
  _unused  uint  uint1  uint2  uint4  union  use  ushort
  void  volatile
  wchar  while  wstring
```

note: "all" et "as" ne sont pas des mots-clés.

1.9 délimiteurs

Les séquences de caractères suivantes ont une signification particulière comme délimiteurs:

```
  # (début de directive)
  { }  ( )  [ ]   '  :  ,  ;  ?  ~
  .  ..
  +  +=  ++
  -  -=  --  ->
  *  *=
  /  /=    // /*  (début commentaire)
  %  %=
  !  !=
  =  ==  =>
  &  &&  &=
  |  ||  |=
  ^  ^=
  <  <=  <<  <<=
  >  >=  >>  >>=
```

1.10 directives de prétraitement

Les symboles de compilation ont une valeur booléenne constante (vrai ou faux).

Ils ne sont utilisés que dans les directives de prétraitement.

Il existe deux types de symboles de compilation:

a) les symboles globaux
   - doivent être écrits en MAJUSCULES;
   - sont définis dans le fichier de configuration (mk.cfg);
   - sont globaux pour tout le projet.

Exemple:

```
  #if WIN
    from std_windows use io;
  #elif UNIX
    from std_unix use io;
```

```
  #elif MAC
    from std_mac use io;
  #else
    #error os not supported
  #endif
```

b) les symboles locaux
   - doivent être écrits en minuscules;
   - sont définis en utilisant la directive #define
   - n'existent que dans le fichier source où ils sont déclarés.

Exemple:

```
  #define debug  0     // désactiver le débogage pour ce fichier source

  #if debug
    // do this
  #else
    // do that
  #endif


  #if !debug
    // do this
  #endif


  #if 0   // supprimer cette partie du code source

    // ...

    #if true
      // ...
    #endif

  #endif
```

Les directives de prétraitement ne peuvent pas être utilisées pour le remplacement macro comme en C.

1.11 directive unsafe

Les directives "#begin unsafe" et "#end unsafe" entourent du code source non-sûr.

Exemple:

```
  #begin unsafe
    char* p;
  #end unsafe
```

Dans ces régions "unsafe", les opérations suivantes sont possibles:

- déclarer et appeler des fonctions externes et callback.
- déclarer et utiliser des pointeurs unsafe.

1.12. règles de style

Les recommandations de style suivantes s'appliquent au Safe-C :

- des paires {} devraient figurer sur la même colonne ou sur la même
  ligne.
- les "if" et "else" correspondants devraient commencer sur la même colonne.
- tous les "case" d'une instruction switch doivent commencer sur la même colonne.

==========

2. Types de données

2.1 entier
2.2 énumération
2.3 virgule flottante

2.4 tableau, tableau ouvert, tableau jagged
2.5 structure, structure ouverte, structure jagged
2.6 union

2.7 pointeur
2.8 incomplet
2.9 pointeur de fonction

2.10 opaque
2.11 objet
2.12 génériques
2.13 pointeur unsafe

2.1 Types Entier

Les types entiers suivants sont disponibles:

Entier signé

tiny	1 byte	(-128 à +127)
short	2 bytes	(-32_768 à +32_767)
int	4 bytes	(-2_147_483_648 à +2_147_483_647)
long	8 bytes	(-9_223_372_036_854_775_808 à +9_223_372_036_854_775_807) (18-19 chiffres)

Entier non-signé

byte	1 byte	(0 à 255)
ushort	2 bytes	(0 à 65_535)
uint	4 bytes	(0 à 4_294_967_295)

Synonymes

int1	1 byte	(-128 à +127)
int2	2 bytes	(-32_768 à +32_767)
int4	4 bytes	(-2_147_483_648 à +2_147_483_647)
int8	8 bytes	(-9_223_372_036_854_775_808 à +9_223_372_036_854_775_807) (18-19 chiffres)

uint1	1 byte	(0 à 255)
uint2	2 bytes	(0 à 65_535)
uint4	4 bytes	(0 à 4_294_967_295)

```
  Attributs 'min 'max
  -------------------
  Les attributs 'min et 'max renvoient les valeurs minimales et maximales d'un
  type entier ou d'une variable.

  N'min: valeur minimale (la plus négative)
  N'max: valeur maximale (la plus positive)

  Exemple:
    printf ("la valeur maximale d'un int est: %d\n", int'max);


2.2 Types énumération
---------------------
Une déclaration d'énumération déclare un type énumération et une liste
de littéraux de ce type.

Exemple:

  enum Color {ROUGE, BLEU, VERT};

Chaque litéral énumération déclare une valeur constante du type énumération.


Les types énumération suivants sont prédéfinis:

  bool  1 byte  (0 à 1)
  char  1 byte  (0 à 255)
  wchar 2 bytes (0 à 65535)

  enum bool  (uint1) {false, true};
  enum char  (uint1) {nul, .., 'A', 'B', .. };
  enum wchar (uint2) {Lnul, .., L'A ', L'B', .. };

  false et true sont les littéraux du type bool;
  nul et Lnul sont les premiers littéraux des types char et wchar,
  respectivement.

Le type "wchar" stocke un symbole du jeu de caractères Unicode qui
supporte toutes les langues parlées à travers le monde (voir l'annexe A
pour plus de détails).

Le premier litéral a toujours la valeur 0, le deuxième a la valeur 1,
le troisième a la valeur 2, etc

Les litéraux de type énumération ne sont pas compatibles avec les
types entiers, mais ils peuvent être convertis.

Exemple:

  char c = (char) 65;


  Attributs 'first 'last
  ----------------------
  Les attributs 'first 'last renvoient le premier et le dernier
  littéral d'un type énumération.

  N'first : premier littéral
  N'last  : dernier littéral
```

N désigne un type énumération ou une variable de ce type.

Exemple:

```
enum Color {ROUGE, VERT, BLEU};

Color color;

color'first   // ROUGE
color'last    // BLEU

Color'first + 1 // VERT
Color'last - 1  // VERT
Color'first + 2 // BLEU
Color'first + 3 // ERREUR: erreur de compilation: débordement
Color'last - 3  // ERREUR: erreur de compilation: débordement
```

Attribut 'string

L'attribut 'string convertit une valeur énumération en son litéral string.

E'string: string représentant le litéral énumération E

E représente une valeur de n'importe quel type énumération sauf char et wchar.

Exemple:

```
color = ROUGE;
printf ("color : %s\n", color'string);
```

Une erreur à la compilation ou à l'exécution se produit si E est plus grand que le dernier litéral du type énumération.

Représentation

Par défaut, un type d'énumération est mappé sur le type de base uint4.

Toutefois, la syntaxe permet la spécification d'un autre type de base qui doit être l'un des suivants : uint1, uint2, uint4 (ou synonymes: byte, ushort, uint).

Exemple:

```
enum Color (byte) {ROUGE, VERT, BLEU};
```

2.3 types à virgule flottante

Les types à virgule flottante suivants sont prédéfinis:

float 4 bytes (1.5E-45 à 3.4E +38, 7 chiffres de précision)
double 8 bytes (5.0E-324 à 1.7E 308, 15-16 chiffres de précision)

Synonymes
---

float4 4 bytes (comme float)
float8 8 bytes (comme double)

2.4 types tableau

Un tableau E[L] est défini par un type élément E et une longueur L.

```
  Exemple:
    char buffer[80];      // un tableau de 80 caractères
    int  t1[10], t2[10];  // deux tableaux de 10 int

    char[80] buffer2;     // un tableau de 80 caractères
    int[10]  T3, T4;      // deux tableaux de 10 int

    char         screen1[25][80];  // un écran de 25 lignes de 80 caractères
    char[80]     screen2[25];      // le même
    char[80][25] screen3;          // le même
```

La longueur L d'un tableau doit avoir le type int ou uint et sa valeur doit être comprise entre 0 et 2_147_483_647 (int'max).

En outre, la taille d'un tableau ne peut pas dépasser int'max bytes.

attribut 'length or 'length(N)

L'attribut 'length renvoie une valeur de type int indiquant la longueur du tableau.
La constante optionelle N spécifie la dimension du tableau (par défaut N=1).

```
  Exemple:
    buffer'length     est égal à 80
    t1'length         est égal à 10
    screen1'length    est égal à 25
    screen1'length(2) est égal à 80
```

```
  Exemple:
    const char [7] welcome = "Bonjour";  // constante tableau de longueur 7

    char[64] etudiant;                   // variable de longueur 64

    char[80]^ ligne = new char[80];      // objet heap

    typedef char[10] NOM;                // type tableau

    void print (char [20] titre)         // paramètre
    {
      ref char[3] prefix = titre [0:3];  // référence
    }
```

correspondance des longueurs de tableau

Les longueurs de tableau doivent correspondre exactement lors d'affectations, d'appels de fonction, d'aggrégats, d'expression qualifiées, d'expressions initiales d'objets, d'expression par défaut des paramètres.

```
  Exemple:

    char[5] nom1 = "ABCDE"; // OK
    char[5] nom2 = "AB";    // ERREUR: longueur du tableau (5 != 2)

    nom2 = {'A', 'B', 'C'}; // ERREUR: longueur du tableau (5 != 3)
```

```
    void print (char[4] le titre); // paramètre

    print ("ABCD");   // OK
    print ("DO");     // ERREUR: longueur du tableau (2 != 4)
```

Les tableaux ouverts

Un tableau ouvert est un tableau dont la longueur est indéterminée:

```
    typedef char[] string;  // string est un tableau ouvert de char
```

Cela peut aussi s'écrire:

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

Les types ouverts peuvent être utilisés pour déclarer des constantes, des paramètres et des références. Toutefois, ils ne peuvent pas être utilisés pour déclarer des variables ou objets heap sans préciser de longueur.

Exemple:

```
  const welcome = "Bonjour";   // constante

  string     etudiant;   // ERREUR: la longueur manque
  string(64) etudiant;   // OK

  string^ ligne = new string;        // ERREUR: la longueur manque
  string^ ligne = new string(len);   // OK
  string^ ligne = new char[len];     // OK

  void imprimer (string titre)       // paramètre
  {
    ref string prefix = titre[0:3];  // référence (3 premiers caractères)
  }
```

Une fonction ayant un paramètre tableau ouvert (comme 'imprimer' ci-dessus) peut être appelée en passant un tableau de n'importe quelle longueur. En interne, la fonction recevra un paramètre caché avec la longueur du tableau.

De même, une référence de tableau ouvert maintiendra une variable cachée avec la longueur effective du tableau (sauf si la longueur est constante).

Tableaux jagged

Un tableau jagged est un tableau où le type d'élément est ouvert ou jagged. Chaque élément du tableau peut avoir une longueur différente.

Exemple:

```
  const string table[4] = {"Ceci", "est", "un", "exemple"};
```

Les tableaux jagged sont toujours constants, mais ils peuvent aussi apparaître en tant que paramètres de mode "in" ou référence à ceux-ci.

Notez que string(2) et string[2] ne sont pas identiques : le premier est un tableau de deux caractères, le second est un tableau de deux chaînes de caractère.

Tableau ouvert et jagged

Un tableau peut être à la fois ouvert et jagged.

Exemple:

```
const string table[] = {"Ceci", "est", "un", "exemple"};
```

construction de constantes

Les constantes peuvent être construites à partir d'aggrégats d'autres constantes,
d'éléments ou tranches de tableau, ou de champs de structure.

Exemple:

```
const string(3)    str  = "abc";
const string(3)[2] str3 = {str, str};
const string(2)    str4 = str3[0][0:2];
```

Variables String

Important:

Les variables de type string ont un caractère délimiteur de fin de chaine 'nul' qui est présent seulement si le tableau n'est pas plein.

Exemple:

```
from std use strings;

void main()
{
  string(4) nom;

  strcpy (out nom, "Luc");     // va copier "Luc" avec nul final
  strcpy (out nom, "Marc");    // va copier "Marc" (sans nul)
  strcpy (out nom, "Jacques"); // provoque une erreur d'exécution
}
```

Dans l'exemple ci-dessus, strcpy reçoit la longueur des deux tableaux et peut effectuer les vérifications nécessaires.

```
// le composant de bibliothèque 'strings' contient des fonctions
// pour gérer les chaînes de caractères comportant un caractère
// délimiteur nul optionel :

strcpy (out dest, src);  : copier une chaîne
strcat (ref dest, src);  : ajouter une chaîne à une autre
len = strlen(s);         : renvoyer la longueur active de la chaîne
cmp = strcmp(s1, s2);    : comparer deux chaînes
cmp = stricmp(s1, s2);   : comparer deux chaînes sans tenir compte de la casse
pos = strchr(s, c);      : rechercher un caractère dans une chaîne
pos = strstr(s1, s2);    : rechercher une chaîne dans une autre chaîne
...
```

note: strcpy et strcat provoquent une erreur d'exécution quand la variable cible

n'est pas assez longue.

Remarque: des fonctions similaires existent pour les types wchar/wstring :
wstrcpy, wstrcat, wstrlen, wstrcmp, wstricmp, wstrchr, wstrstr.

2.5. types structure

Un type structure définit un ensemble de champs.

Exemple:

```
struct NOEUD
{
  char [20]   nom;
  int         count;
  byte [1024] free_text;
  NOEUD^      prev, next;
}
```

La taille d'une structure ne peut pas dépasser int'max bytes.

Structures ouvertes

Les structures dites 'ouvertes' ont un paramètre appelé "discriminant" dont la valeur détermine quels champs variants font partie de la structure.

Exemple:

```
enum GenreForme {POINT, CARRE, CERCLE, TRIANGLE};

struct Forme (GenreForme genre)
{
  int x, y;

  switch (genre)
  {
    case POINT:
      null;

    case CARRE:
      int cote;

    case CERCLE:
      int rayon;

    case TRIANGLE:
      int base, hauteur;
  }
}
```

Le discriminant 'genre' est un paramètre utilisé pour créer la structure ouverte 'Forme'.

Une partie 'switch' doit être présente si et seulement si la structure comporte un discriminant, la structure est alors 'ouverte'.

Le mot-clé "null" indique une variante sans champs.

Exemple:

```
  // le type struct Forme définit les champs x, y et cote.
  typedef Forme(CARRE) Carre;

  const Forme carre = {x => 10, y => 10, cote => 5};

  Forme        sh;    // ERREUR: discriminant manque
  Forme(CARRE) sq;    // OK

  void put_square (ref Carre c);  // passe l'adresse
  void put_shape  (ref Forme f);  // passe l'adresse + discriminant

  Forme^ p1 = new Forme;     // ERREUR: discriminant manque
  Forme^ p2 = new Forme(k);  // OK (k est de type GenreForme)
```

structs jagged

Une struct jagged est une structure contenant des champs d'un type ouvert ou jagged.

Exemple:

```
  struct info
  {
    int    code;
    string titre;    // champ de type ouvert
    Forme  forme;    // champ de type ouvert
  }

  msg = const info {10, "carré", Carre'{0,0,10}};
```

Mot-clé "packed"

Le mot-clé "packed" peut précéder une déclaration de structure.

Exemple:

```
  struct foo1     // taille = 8 bytes, alignement à 4
  {
    float f;
    char  c;
  }

  packed struct foo2  // taille = 5 bytes, alignement à 1
  {
    float f;
    char  c;
  }
```

L'utilisation du mot-clé "packed" a plusieurs effets importants :

1) le compilateur n'insère pas de bytes d'alignement dans, ou à la fin, de la structure.

2) les types 'packed' sont implicitement convertibles en type "byte[]" dans les appels de fonctions, afin qu'ils puissent être transmis de

et vers l'extérieur (réseau, fichiers, ..) par de nombreuses fonctions d'entrée-sortie.

3) les pointeurs et les pointeurs de fonction ne sont pas autorisés dans les champs 'packed structs' pour éviter qu'ils ne soient corrompus. Le passage d'un pointeur de, et vers, l'extérieur est probablement une erreur de toute façon.

Grâce au packing, la représentation interne est définie avec précision (sauf pour l'ordre l'ordre des bytes) et elle est portable.

construction de constantes

Les constantes peuvent être construites à partir d'agrégats d'autres constantes, d'éléments ou tranches de tableaux, ou de champs de structures.

Exemple:

```
const Forme(CARRE)[2] sh3 = {sh, sh};
```

structures jagged

Une constante structure jagged est une structure dont au moins un champ a un type ouvert.

Exemple:

```
const Forme[2] sh4 = {sh, sh};
```

2.6 types union

Les types union ont tous leurs champs mappés sur la même zone mémoire.

La taille d'un type union est calculé en prenant le plus grand de tous ses champs.

Exemple:

```
union U
{
  int  a;
  char b;
}
```

Les types union sont toujours 'packed' implicitement.
Il n'y a pas de constantes de type union.

2.7. types pointeur

Les pointeurs sont utilisés pour accéder à des objets anonymes alloués dynamiquement;
ils ne peuvent pas accéder aux variables globales ou locales.

Un pointeur est déclaré en utilisant le symbole ^ (prononcé caret).

Exemple:

```
Forme^      prev, next;    // deux pointeurs de Forme
Forme       prev^, next^;  // idem
```

```
    Forme       table[10]^;    // tableau de 10 pointeurs vers Forme
    Forme^      table[10];     // idem
    Forme^[10]  table;         // idem

    string^ nom = new string (80);
    strcpy (out nom^, "Bonjour");
    free nom;
```

Les pointeurs vers les "types ouvert" et vers les "types non-ouverts"
ne sont pas compatibles parce que les premiers ont une entête
interne supplémentaire.

Exemple:

```
    string^     p1;
    string(10)^ p2;

    p1 = p2;   // ERREUR: les pointeurs ne sont pas compatibles
```

2.8 type incomplet

Les types incomplets sont utilisés pour déclarer des types ayant des
références mutuelles.

Exemple:

```
  typedef node2;   // type incomplet

  struct node1
  {
    int    count;
    node1^ up;             // pointeur sur struct lui-même
    node2^ gauche, droite; // utilise le type incomplet
  }

  struct node2
  {
    int    count;
    node1^ gauche, droite;
  }
```

2.9. Les pointeurs de fonction

Les pointeurs de fonction permettent des appels vers une fonction
dépendant du contenu de la variable pointeur.

Exemple:

```
  void treat_node (Forme f);   // déclaration de fonction

  typedef void TRAITER (Forme f);  // type pointeur de fonction

  void traitement ()
  {
    TRAITER traiter;   // variable pointeur de fonction

    traiter = null;
    traiter = treat_node;  // mode des paramètres et types doivent correspondre
```

```
    if (traiter != null)
      traiter (f);
  }
```

2.10. types opaques

Les types opaques sont utilisés pour déclarer des structures
dont les champs sont cachés.

Les types opaques ne peuvent être déclarés que dans les unités .h
ou dans les déclarations de package.

Exemple:

```
  ------------------------------------------------

  // drawing.h

  struct DRAW_CONTEXT;   // type opaque

  void init   (out DRAW_CONTEXT d);
  void cercle (ref DRAW_CONTEXT d, int x, int y, int rayon);

  ------------------------------------------------
```

Plus tard, le type struct complet correspondant peut être déclaré:

Exemple:

```
  ------------------------------------------------

  // drawing.c

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

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

  void cercle (ref DRAW_CONTEXT d, int x, int y, int rayon)
  {
    / / ..
  }

  ------------------------------------------------
```

Dans les endroits où la déclaration complète n'est pas visible,
il n'est pas permis de faire des copies du type opaque, par exemple
en utilisant une affectation.

Exemple:

```
  use drawing;
```

```
  void main()
  {
    DRAW_CONTEXT a, b;
    init (out a);
    b = a;     // ERREUR: affectation non autorisée pour des types opaque
  }
```

Le programmeur peut cependant fournir une fonction de clonage explicite
là où la structure complète est visible.

2.11. Type 'object'

Le type 'objet' est prédéfini comme:

```
    typedef byte[] object;    // tableau ouvert de byte
```

Le type jagged 'object[]' est utilisé pour déclarer des fonctions
comme sprintf() qui comportent un nombre variable de paramètres.

Exemple:

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

Un paramètre de type 'object[]' peut être utilisé par exemple dans les
contextes suivants :

```
  arg'length      // comme préfixe de l'attribut 'length,
                  // pour interroger le nombre de paramètres.

  arg [i]         // comme préfixe d'un élément de tableau: le résultat
                  // a le type byte[] et est convertible en n'importe
                  // quel type 'packed'.

  func (arg)      // comme paramètre dans un appel de fonction
```

2.12. type générique

Les types génériques sont utilisés pour paramétrer les types d'un algorithme,
voir le chapitre sur les génériques.

2.13. Types de pointeur 'unsafe'

Les pointeurs 'unsafe' sont semblables aux pointeurs du C.

  Exemple:

```
    int i, p*;  // note: le symbole * apparaît comme suffixe
    p = &i;
    i = *p;
    i = p[0];
    p++;
```

L'utilisation des types de pointeurs 'unsafe' n'est autorisée que dans

les régions 'unsafe' (Voir 1.11)

==========

3. Déclarations

3.1. déclaration typedef

Une déclaration typedef peut être utilisée pour déclarer un nom alias d'un type, pour
ajouter un spécificateur à un type, ou pour déclarer un type pointeur de fonction.

Exemple:

```
  typedef int ENTIER;     // nom alias

  typedef string(20) NOM_ETUDIANT;

  typedef string^ PSTRING;

  typedef int SUM (int a, int b);  // type pointeur de fonction
```

3.2. déclaration d'un objet

Une déclaration d'objet déclare une constante ou une variable.

Exemple:

```
  const int MAX = 100;
  string    buffer(MAX);
  int       i=0, j=0;
```

champ d'application

Un objet peut être déclaré au niveau global (à l'intérieur des unités
de compilation ou des packages) ou au niveau local (dans les fonctions
et instructions de bloc).

Au niveau global toute expression initiale, si présente, doit être constante.

constante et variable

Si le mot-clé 'const' est spécifié, la déclaration d'objet déclare une
constante dont la valeur est calculée au moment de la compilation.

Si aucun mot-clé 'const' n'est spécifié, la déclaration de l'objet déclare une
variable; si une expression est spécifiée, elle est la valeur initiale de
l'objet.

runtime

Les objets globaux (et aussi les objets heap) ont une occurrence unique, ils
peuvent être consultés par plusieurs threads qui peuvent effectuer des mises
à jour simultanées.

Les objets locaux peuvent avoir plusieurs occurrences : une nouvelle occurence est créée
chaque fois que la fonction déclarante est appelée, chaque occurrence locale ne peut

être accessible que par un seul thread.

Le mot-clé 'volatile' doit être spécifié pour les variables globales qui peuvent être modifiés par plusieurs threads, ou susceptibles d'être modifiés par un appel du système d'exploitation. Il indique au compilateur de toujours charger une nouvelle copie de ces variables et de les enregistrer dans la mémoire au lieu de les garder dans des registres.

Les objets locaux de plus de 4 KBytes sont alloués automatiquement sur le heap et libérés au moment de quitter leur région déclarative.

initialisation

Les variables globales, si aucune expression n'est précisée dans leur déclaration, sont
toujours initialisés avec des zéros binaires.

Les variables locales, si aucune expression n'est spécifiée, ont une valeur initialement
indéfinie. Avant qu'une variable locale ne puisse être lue, la variable complète doit
d'abord recevoir une valeur. L'initialisation d'éléments ou de tranches d'un tableau,
ou des champs d'une structure, ne compte pas comme une initialisation de la variable entière.

Prendre l'adresse d'une variable en utilisant l'opérateur unaire (&) est considéré comme un
accès en écriture suivi d'un accès en lecture de la variable.

Un paramètre de mode 'out' doit recevoir une valeur avant que la fonction se termine,
quel que soit le flot d'instructions suivi.

Les objets qui ne sont jamais lus ou utilisés dans un attribut peuvent
générer un avertissement du compilateur disant qu'ils sont inutilisés
(voir 7.19. instruction unused)

3.3. déclaration de référence

Une référence déclare un alias à un objet existant.

Les références peuvent être utilisées pour renommer des objets en des noms plus courts.

Les déclarations de références ne sont autorisés que dans un corps de fonction.

Exemple:
```
ref char   ch  = buffer[i];     // référence à un élément de tableau
ref string str = buffer[0:3];   // référence à une tranche de tableau
ref int    age = etudiant.age;  // référence à un champ âge
```

La référence ne peut pas être modifiée après sa création, elle désigne toujours le même objet. L'affectation à une référence affecte une nouvelle valeur à l'objet référencé.

Les références aux objets heap "verrouillent" l'objet jusqu'à ce que
la déclaration de référence soit hors de portée. Libérer l'objet heap
avant ce moment provoque une erreur d'exécution.

Exemple:

```
  {
    ref int count = p^.count;

    count = 2;

    free p;   // erreur d'exécution !
  }
```

3.4. Déclaration de fonction

Une déclaration de fonction précise les paramètres, valeur de retour
et options d'une fonction.

Exemple:

```
  int start_treatment ();
  int treat (char count);
  void sum (int a, int b, out c);
  void operate (ref TABLE table);
  void print (int value, int width = 1);
```

Paramètres

Il y a 3 modes de passage de paramètres:

mode d'accès au paramètre	
in	lecture seule (par copie pour les types simples, sinon par référence)
ref	lecture+écriture (par référence)
out	écriture d'abord, puis lecture+écriture (par référence)

Si aucun mode n'est spécifié, le mode "in" est implicite.

Si un paramètre a une expression constante par défaut (ex: 'width' ci-dessus),
il doit avoir le mode "in". Un argument correspondant est alors optionnel
lors de l'appel de la fonction.

Type de retour

Le type de retour d'une fonction peut être soit un type simple
(donc tableau, struct, union, opaque ou générique ne sont pas
autorisés), ou void si la fonction n'a pas de paramètres.

Options

Les options suivantes peuvent être spécifiées:

1) inline

```
  inline int treat (char count);
```

indique que la fonction devrait être insérée en ligne au lieu
d'être appelée.

2) callback

```
  [callback]
  int MainWin (HWND hwnd, UINT message, WORD w, LONG l);
```

indique que la fonction sera appelée par le système d'exploitation.

3) extern

```
  [extern "KERNEL32.DLL"]
  int GetStdHandle (int nStdHandle);
```

indique que le corps de la fonction se trouve dans une DLL externe.

3.5. corps de la fonction

Le corps de la fonction définit le fonctionnement interne d'une fonction.

Exemple:

```
  void somme (int a, int b, out int c)
  {
    c = a + b;
  }
```

Le mot-clé "public" est obligatoire pour un corps de fonction si une déclaration correspondante de cette fonction apparait plus tôt dans le fichier .h ou dans le package.

Exemple:

```
  // p.h

  void somme (int a, int b, out int c);    // déclaration


  // p.c

  public void somme (int a, int b, out int c)   // corps avec le mot clé "public"
  {
    c = a + b;
  }
```

L'asymétrie entre la déclaration et le corps est intentionnelle, l'objectif pragmatique étant une utilisation minimale du mot-clé "public".

retour de fonction

Une fonction ayant un type de retour non-void doit se terminer par une instruction return ou abort.

==========

4. Noms

4.1. noms étendus

Quand un identificateur déclaré dans une unité ou un package est ambigu

il peut être précédé du nom de l'unité ou du package pour le rendre unique.
On appelle cela un nom étendu.

  Exemple:
    strings.strcpy

4.2. objets et valeurs

Les noms suivants peuvent être construits en utilisant des objets:

  Element de tableau

  Exemple:
    tableau [indice]

  Le préfixe doit désigner un objet tableau (ou un objet pointeur unsafe).
  L'indice doit avoir le type int ou uint.

  Une erreur se produit si l'indice est en dehors de la plage du tableau.

  Tranche de tableau

  Exemple:
    tableau [offset : longueur]

  Le préfixe doit désigner un objet tableau (ou un objet pointeur unsafe).

  L'offset est la limite inférieure de la tranche, longueur est la longueur
  de la tranche: les deux doivent avoir le type int ou uint.

  Exemple:
    étant donné S = "ABCDE", S[2:3] s'évalue à "CDE".

  Les tranches vides sont autorisées : S[5:0] s'évalue comme "".

  Une erreur se produit si 'offset' ou 'longueur' ont des valeurs illégales.

  valeur d'un discriminant

  Exemple:
    struct_ouverte . discriminant

  Le préfixe doit désigner un objet de type structure ouverte.

  champ d'une structure
  ---
  struct . champ

  Le préfixe doit désigner un objet de type structure ou union.

  Si le champ appartient à une variante d'un type struct ouverte,
  une vérification à l'exécution est effectuée pour s'assurer que
  la valeur du discriminant est appropriée pour accéder à ce champ,
  à moins que cette vérification ne puisse être assurée à la compilation.

  objets déréférencés

Exemple:

```
pointeur ^
```

Le préfixe doit désigner un objet ou une valeur de type pointeur.
Le résultat est l'objet heap désigné par le pointeur.

Une erreur se produit si le préfixe a la valeur null, ou si l'objet heap a déjà été libérée plus tôt.

objet postfixé

Exemple:

```
int p*, q*;
*p++ = *q++;
```

Le préfixe doit désigner un objet de type entier ou enumération, ou un type de pointeur unsafe.

L'objet sera converti en une valeur, après quoi l'objet sera incrémenté ou décrémenté. Dans le cas d'un pointeur unsafe la valeur du pointeur sera incrémentée / décrémentée par la taille du type désigné par le pointeur.

Deref champ unsafe

Exemple:

```
pointeur_unsafe ->  discriminant_ou_champ
```

L'opérateur "->" est équivalent à la succession des deux opérateurs "*" et ".".

Nom de fonction

Un nom de fonction s'évalue comme une valeur de type pointeur de fonction correspondant au début du code exécutable du corps de fonction.

Attributs

Les attributs suivants existent:

attributs 'min et 'max

I'min: valeur minimale (la plus négative)
I'max: valeur maximale (la plus positive)

I doit désigner un type entier ou une variable (une constante n'est pas autorisée).

attributs 'first 'last

E'first : premier litéral énumération
E'last : dernier litéral énumération

E doit désigner un type énumération ou une variable (une constante n'est pas autorisée).

attribut 'length 'length(N)

```
    ----------------------------
    A'length retourne le nombre d'éléments d'un tableau A.

    A doit désigner un objet tableau, ou un type tableau non-ouvert.
    La constante entière optionnelle N spécifie une dimension de tableau (>= 1)
    qui n'est valable que si A est un tableau à N dimensions.

    attribut 'byte
    --------------
    O'byte convertit un objet O de n'importe quel type packed en un tableau de
bytes
    (byte[]) ayant la même représentation interne et la même taille.

    Exemple:
      int   i = 4;
      float f;

      f'byte = i'byte;           // copier i dans f  (4 bytes)

      if (f'byte[3] & 128 != 0)  // on suppose une représentation little-endian
        printf ("négatif");

    note: l'attribut 'byte va renvoyer des résultats différents pour des
          architectures d'ordinateur big endian et little endian.


    attribut 'string
    ----------------
    Exemple:
      printf ("color : %s\n", color'string);

    E'string convertit une valeur d'énumération E en un litéral string.

    Le préfixe doit désigner une valeur de n'importe quel type énumération,
    excepté char et wchar.

    Une erreur se produit si la valeur du préfixe est plus grande que le dernier
    litéral énumération.


    attribut 'size
    --------------
    O'size : renvoie le nombre de bytes utilisés par un objet ou type O.

    Le résultat est une valeur de type uint.



4.3. Appel de fonction
----------------------
Un appel de fonction appelle une fonction et renvoie éventuellement une valeur.

Exemple:
  diviser (i, j, out ans, out r);

  retrieve_object (    id   => nr,
                   out item => my_item);

Certains paramètres peuvent utiliser une forme "par nom", c'est-à-dire
qu'on les fait précéder du nom du paramètre et du symbol "=>".
```

Tous les paramètres doivent être fournis, dans l'ordre, sauf éventuellement les derniers paramètres si une expression constante par défaut est définie pour eux dans la déclaration de la fonction.

Le nom d'un appel de fonction doit être un nom qui dénote une valeur de type pointeur de fonction; si elle prend la valeur null, une erreur se produit.

Exemple:
```
  p^.func(exp)^.f("a");
  (*p)("a");
```

l'ordre d'évaluation

Les paramètres sont évalués de gauche à droite, tels qu'ils apparaissent dans le texte source.

Exemple:

```
  int i = 0;
  f (i++, i++, i++);  // même que f (0, 1, 2);
```

conversion

Si un paramètre effectif ou formel a le type tableau de byte,
alors la conversion de/vers un type packed est possible.

Exemple:
```
   void put (byte[] b);

  put (2);            // type int
  put (2L);           // type long
  put ((byte)2);      // type byte


  void put2 (int i);

  put2 ( byte'{0,0,0,1} );
```

type 'object[]'

Si le dernier paramètre formel est de type object[], il peut accepter une liste des paramètres effectifs de type packed.

Exemple:

```
  void f (string format, object[] arg)
  {
    ...   arg'length   arg[i]'byte
  }
```

Un paramètre ayant la forme string litéral précédent un tel paramètre de type object[] de mode in est vérifié par les modèles de format suivants :

```
    % [Drapeau] [Largeur] [. Précision] Type
```

Type		sortie
%		'%%' s'écrira '%'
d	entier signé	nombre décimal (-61)
u	entier non-signé ou enum	nombre non signé (12)

```
   x  entier ou enum            nombre hexadécimal (7fa)
   e  float, double             format scientifique (3.9265e+2)
   f  float, double             virgule flottante (392.65)
   c  char ou string            tous les char, ou max 'précision' char.
   C  wchar ou wstring          tous les wchar, ou max 'précision' wchar.
   s  string                    chaîne s'arrête à nul, ou "précision" char
   S  wstring                   chaîne s'arrête à Lnul, ou "précision" wchar

   Largeur
   -----
   (Nombre)   nombre minimum de caractères à imprimer (complété
              avec des espaces ou des zéros si nécessaire).
              La valeur n'est pas tronquée, même si le résultat est trop long.
   *          La largeur n'est pas spécifiée dans la chaîne de format, mais dans
un
              paramètre supplémentaire de valeur int précédant le paramètre à
              formater.

   Précision
   ---------
   . Nombre  Un point sans nombre indique une précision de zéro.
             Pour f: c'est le nombre de chiffres à afficher après le
                     point décimal. La précision par défaut est de 6.
                     Aucun point décimal est n'imprimé pour une précision de zéro.
             Pour s/S: c'est le nombre maximum de caractères à
                       imprimer. Par défaut on imprime tous les caractères.
   .*        La précision n'est pas spécifiée dans la chaîne de format, mais comme
un
             paramètre supplémentaire de valeur int précédant le paramètre à
             formater.

   Drapeaux
   --------
   -    Pour tous          : justifier à gauche endéans le champ
                             (valeur par défaut : justifier à droite).
   +    Pour d, e, f       : préfixe '+' pour les nombres positifs ou nuls.
   0    pour d, e, f, x, u : préfixe des chiffres '0' au lieu d'espaces;
                             (non valable avec le flag '-')


Les modèles de format suivants sont supportés pour un paramètre de mode 'out'
ou 'ref':

  l'espace blanc (ASCII de 1 à 32)
  --------------------------------
  ignorer zéro ou plusieurs caractères d'espace blanc.

  Le caractère non-blanc, à l'exception du caractère pourcentage (%)
  ------------------------------------------------------------------
  doivent correspondre, ou la fonction échoue.

  Spécificateurs de format:  %[*][Largeur] Type

    *         Les données sont lues mais ne sont pas stockées (il n'y a aucun
              paramètre correspondant).
   Largeur    Nombre maximum de caractères à lire.

   Type                        Entrée
   ----                        -------
   '%'                         se lira '%%'
```

d	tout entier signé	nombre décimal optionnellement précédé de + ou - (-61)
u	tout entier non-signé ou enum	nombre non signé (12)
x	idem à d ou u	nombre hexadécimal (7fa)
f	float ou double	nombre à virgule flottante (0,5) (12.4e +3)
e	idem que f	
c	char ou string	remplir arg, ou lire max 'width' chars.
C	wchar ou wstring	remplir arg, ou lire max 'width' wchars.
s	char ou string	idem que c mais s'arrête au premier blanc.
S	wchar ou wstring	idem que C, mais s'arrête au premier blanc.

4.6. Appel 'Run'

Un appel 'run' démarre un thread. Un paramètre optionnel peut être passé à la fonction du thread.

Exemple:

```
my_thread void (int i)
{
}

void main ()
{
  int rc;

  rc = run my_thread (23);
}
```

Un appel 'run' renvoie une valeur int indiquant si le thread a démarré correctement
(zéro indique le succès, une valeur négative indique une erreur).

5. Primaires

Un primaire désigne l'une des valeurs suivantes:

littéral entier

Exemple: 123 0x10 10L

littéral virgule flottante

Exemple: 1.23 3.0d

littéral caractère

Exemple: 'a' L'a'

littérale chaîne

Exemple: "Bonjour"   L"Salut"

```
  nul / Lnul
  ----------
  Les mots-clés réservés 'nul' et 'Lnul' désignent respectivement une valeur
  égale au premier litéral énum des types 'char' et 'wchar'.

  null
  ----
  Le mot-clé réservé 'null' représente une valeur de pointeur spéciale qui ne
  pointe sur rien. Il est compatible avec tous les pointeurs, pointeurs de
fonction
  et pointeur unsafe.

  agrégat
  -------
  Un agrégat est une valeur composée de type tableau ou structure.

    agrégat tableau
    ---------------
    Deux formes d'agrégats tableau existent:

      a) agrégat tableau positionnel
      ------------------------------
      Exemple:
        tab = {'A', 'B', c1, 'D', 'I', ch1, ch2};

      Remarque: la syntaxe permet une virgule supplémentaire ',' avant la clôture
'}'.

      b) aggrégat tableau ouvert
      --------------------------
      Exemple:
         tab = {all => c};


    aggrégat structure
    ------------------
    Deux formes d'agrégat structure existent:

      a) aggrégat structure positionnel
      ---------------------------------
      Exemple:

        struct Noeud
        {
          float  longueur;
          char   lettre;
          bool   actif;
        }

        Node n = {1.0, 'A', b};   // b peut être une variable

      Remarque: la syntaxe permet une virgule supplémentaire ',' avant la
fermeture '}'.


      b) aggrégat structure nommé
      ---------------------------
      Exemple:

        Node n2 = {longueur => 1.0, lettre => 'A', actif => b};
```

Tous les champs de la structure doivent apparaître dans le même ordre
que dans la déclaration de la structure.

expression qualifiée

Une expression qualifiée montre explicitement le type d'une expression.

a) expression qualifiée entre parenthèses

Exemple:

```
t    = tiny      ' (2);
d    = double    ' (f+1);
p    = ptr       ' (null);
str3 = string(3) ' ("ABC");
s    = Shape     ' (sh);
```

b) expression qualifiée avec aggrégat

Exemple:

```
string(3)     str3;
Forme(CERCLE) sh;

str3 = string   ' {'a', 'b', 'c'};
str3 = string(3)' {'a', 'b', 'c'};
str3 = string   ' {all => '*'};   // utilise la longueur du contexte

sh   = Shape'{0,0,12};

str3[ofs:len] = string ' {all => c};   // longueur à l'exécution
pstr^         = string ' {all => c};   // longueur à l'exécution
```

allocateur 'new'

L'allocateur 'new' crée un nouvel objet heap.

Exemples avec expression initiale:

```
string^  name1 = new string'{'a', 'b', 'c'};
string^  name2 = new char[]'{'a', 'b', 'c'};
string^  name3 = new string'("Hello");
string^  name4 = new string'(name2^);
string^  name5 = new char[len] ' { all => c };
string^  name6 = new string'(s);  // longueur dépend de s'length

p = new Forme(CERCLE) ' { x      => x,
                          y      => y+1,
                          rayon  => r  };

p = new Forme ' (sh);   // taille dépend du discriminant de sh
```

Exemples sans expression initiale (l'objet est initialisé avec des zéros):

```
NOEUD^     p      = new NOEUD;      // pointeur vers un nouveau NOEUD
Forme^     sh     = new Forme(d);   // la taille est calculée à l'aide d'une
table
```

```
    string^    name7  = new string;     // ERROR (open types not allowed)
    char[100]^ name8  = new char[100];
    char[100]^ name9  = new string(100);
    string^    name10 = new string(100);
    string^    name11 = new string(len);
```

Il est à noter que les pointeurs vers tableaux ouverts ne sont pas compatibles avec les pointeurs vers tableaux.

Exemple:

```
  string^ buffer      = new string(30); // context : open array.
  string(30)^ buffer2 = new string(30); // context : non-open array
  buffer2 = buffer;                     // ERROR: non-compatible
```

6. Expressions

Les opérateurs suivants sont définis.

priorités des opérateurs

Unaire	+	-	!	~	&	*	--	++
Terme	*	/	%					
Somme	+	-						
Expression Simple	<<	>>						
Relation	==	!=	<	>	<=	>=		
Conditionel	&&	||	&	|	^		
Test Conditionel	?	:						

évaluation

Les expressions sont évaluées de gauche à droite, tels qu'elles apparaissent dans le texte source.

promotion des types entiers

Les types signés et non-signés ne peuvent être mélangés :
ils ont toujours besoin d'une conversion.

Exemple:

```
    i + u        // ERREUR: int et uint ne peuvent pas être mélangés
    i + (int)u   // OK
    i < u        // ERREUR: int et uint ne peuvent être comparés
    i + 1        // le littéral entier est supposé être int
    u + 1        // le littéral entier est supposé être uint
    1 - 2        // est équivalent à -1
    u + (-1)     // ERREUR: -1 hors de portée de uint
```

sémantique des opérateurs

opérateur ? :

effet:

```
  if (arg1) return arg2; else return arg3;
```

```
  opérateurs && et ||
  -------------------
  effet:
    && : if (!left) return false; else return right;
    || : if (left) return true; else return right;


  opérateurs & | ^
  ---------------
  1) pour entier
  effet:
    effectuer la promotion entier sur des arguments gauche et droit, donnant un
résultat.
    opérateurs logiques bit à bit: et, ou, xor.

  2) pour bool
  effet:
    opérateurs logiques: et, ou, xor.


  les opérateurs == !=
  --------------------
  effet:
    comparaison booléenne.


  opérateurs <  >  <=  >=
  -----------------------
  effet:
    comparaison booléenne.


  les opérateurs <<  >>
  ---------------------
  effet:
    shift gauche ou droit


  les opérateurs + -
  ------------------
  1) pour entiers, virgule flottante
  effet:
    addition, soustraction.

  2) pour énumération
  types:
    gauche  : énumération
    droit   : entier
    résultat: comme à gauche
  effet:
    addition, soustraction.

  3) pour pointeur unsafe
  types:
    gauche   : pointeur unsafe
    droite   : type entier, mais le type long n'est pas autorisé.
    résultat : comme à gauche
  effet:
    convertir l'opérande droite vers un int4 signé, le multiplier par la taille du
    type pointé, éventuellement tronquer le résultat en un int4, puis
```

```
ajouter/soustraire
    ce décalage de/vers l'opérande gauche.

  4) pour le pointeur unsafe ("-" seulement)
  types:
    gauche   : pointeur unsafe
    droite   : comme à gauche
    résultat : uint
  effet:
    soustraction des deux opérandes, conversion vers un int4, division par la
taille
    du type pointé, conversion vers uint4. Une erreur d'exécution se produit si
    le type pointé a une taille de zéro.

  5) pour les strings constants ("+" uniquement)
  types:
    gauche   : constante string/wstring/char/wchar
    droite   : constante string/wstring/char/wchar
    résultat : constante string/wstring
  effet:
    concaténation des chaînes constantes.
    si un argument est unicode(w), alors le résultat est de type wstring,
    sinon le résultat est de type string.

  Exemple:
    "Marie-Jeanne est une personne sympathique." + (CR + LF) +
    "Elle vient souvent nous voir." + CR + LF


  opérateurs binaires *  /
  ------------------------
  effet:
    multiplication, division.

  opérateur %
  -----------
  effet:
    reste de la division.

  opérateurs unaires + -
  ----------------------
  effet:
    + : pas d'effet.
    - : négation

  opérateur unaires ~
  -------------------
  effet:
    not (inverse tous les bits)


  opérateur unaire !
  ------------------
  effet:
    inverse la valeur bool (zéro devient un et non-zéro devient zéro).


  opérateur unaire &
  ------------------
  effet:
    prendre l'adresse de l'objet.
```

restriction:
  cet opérateur n'est autorisé que dans les régions unsafe (voir 1.11).

opérateur unaire *
---
effet:
  l'argument est évalué, une erreur se produit s'il a une valeur null;
  le résultat est l'objet accédé via le pointeur.
restriction:
  cet opérateur n'est autorisé que dans les régions unsafe (voir 1.11);

Exemple:
```
*p = 2;
```

opérateurs unaires --/++
---
effet:
  L'objet sera incrémenté / décrémenté, puis il est converti en
  une valeur.

Exemple:
```
--i;
```

conversion

La conversion est permise pour les types suivants:

type source		type cible
entier	->	entier
énumération	->	entier
virgule flottante	->	entier
entier	->	énumération
virgule flottante	->	virgule flottante
entier	->	virgule flottante
pointeur unsafe	->	pointeur unsafe

La conversion de float vers entier tronque (n'arrondit pas).

La conversion vers un type énumération peut créer une valeur pour laquelle aucun
litéral énumération n'existe.

Lors d'une conversion aucune vérification n'est effectuée à l'exécution.

D'autres conversions qui ne changent pas la représentation des bytes sont
possibles en utilisant l'attribut 'byte.

Exemple:
```
int  i;
char c;
c = (char)i;
```

==========

7. Instructions

7.1. instruction nulle

Une instruction nulle n'a aucun effet.

Exemple:

```
  if (test)
     ;           // instruction nulle
  else
    a = 1;
```

7.2. Instruction 'clear'

Exemple:

```
  clear table, a, b, c;
```

L'instruction 'clear' remplit tous les objets spécifiés avec des zéros binaires.

7.3. Instruction d'affectation

Une instruction d'affectation copie une expression dans une variable.

Exemple:

```
  a = 1;
```

affectation de tableau

Exemple:

```
    char[80] ligne1, ligne2;
    ligne1 = ligne2;      // copie de la variable complète
```

Dans le cas d'un tableau, une vérification est faite que les deux longueurs concordent.

7.4. instruction pre/postfix

L'effet de cette instruction est d'augmenter/diminuer la valeur de l'objet de 1; les débordements sont ignorés.

Exemple:

```
  --(*p)^.count;
  b[k]++;
```

7.5. instruction d'appel de fonction

Une instruction d'appel de fonction appelle la fonction spécifiée.

Exemple:

```
  unlink (filename);
  (void) unlink (filename);
  p[k](i);
  f()(i);
  f(k)^.func(i);
  (*p)('a');
```

7.6. instruction 'return'

Une instruction 'return' termine la fonction en cours et éventuellement
renvoie une valeur.

Exemple:

```
  return;
  return 1;
```

7.7. instruction de bloc

Un instruction de bloc regroupe une série d'instructions.

Exemple:

```
  {         // Instruction de bloc
    H();
    I();
  }
```

7.8. instruction if

Une instruction if permet l'exécution conditionnelle.

Exemple:

```
   if (b)
   {
     // ...
   }

   if (b)
     ;
   else
     ;
```

7.9. instruction switch

Une instruction switch permet l'exécution conditionnelle à branches multiples.

Exemple:

```
  switch (e)
  {
    case 0:
      f();
      break;

    case 1:
      g();
      break;

    default:
      break;
  }
```

7.10. instruction 'while'

L'instruction 'while' permet la répétition d'instructions.

Exemple:

```
  while (compteur < 10)
    compteur++;
```

7.11. instruction 'for'

L'instruction 'for' est une forme plus compacte de l'instruction 'while'.

Exemple:

```
  for (i=0; i<arg'length; i++)
    printf ("arg %d : %s\n", i, arg[i]);

  for ( ; p!=null; i++,j+=3,p=p^.next)
    ;
```

7.12. instruction 'break'

L'instruction 'break' est utilisée pour quitter l'instruction 'while', 'for' ou 'switch' la plus interne.

Exemple:

```
  break;
```

7.13. instruction 'continue'

L'instruction 'continue' est utilisée pour passer à la prochaine itération d'une instruction 'while' ou 'for'.

Exemple:

```
  continue;
```

7.14. instruction 'free'

L'instruction 'free' libère l'objet heap désigné.

Exemple:

```
  free p;
  free null;
  free f();
```

Libérer un pointeur null n'a aucun effet.

Une erreur d'exécution se produit si l'objet heap est toujours référencé par une référence ou un paramètre, ou s'il a déjà été libéré.

7.15. instruction 'assert'

L'instruction 'assert' permet au programmeur d'inclure des contrôles à la compilation ou à l'exécution afin de garantir l'intégrité des données. L'expression de test ne sera jamais évaluée à false à moins que quelque chose soit très erroné.

Exemple:

```
  const int TABLE_SIZE = 100;
```

```
  assert TABLE_SIZE% 2 == 0;  // TABLE_SIZE doit être divisible par deux
```

Un échec de l'assertion arrête le programme.

7.16. instruction 'abort'

Une instruction 'abort' arrête le programme.

Exemple:

```
    switch (reponse)
    {
      case 'Y':
        printf ("oui");
        break;

      case 'N':
        printf ("non");
        break;

      default:
        abort;   // ne devrait jamais se produire
    }
```

7.17. instruction 'sleep'

L'instruction 'sleep' suspend le programme pour la durée spécifiée, en secondes.

Exemple:

```
  sleep 10;   // attendre 10 secondes
  sleep 1.5;  // attendre 1.5 secondes
  sleep 0;    // abandonner la tranche de temps en cours pour ce thread
```

L'expression doit avoir un type entier ou virgule flottante.
Les durées de plus de 86400 secondes (un jour) ou plus petit que -86400 secondes ne sont pas autorisées. Une durée négative n'a pas d'effet.

7.18. instruction 'code'

Une instruction 'code' insère du code machine directement dans le programme.

Exemple:

```
   _asm {0xC2 0x08 0x00};   // ret 8
```

7.19. instruction '_unused'

Le but d'une instruction '_unused' est de supprimer l'avertissement du compilateur qu'une variable ou un paramètre est inutilisé.

Exemple:

```
   _unused par1;
```

==========

8. Packages

==========
Un package est une enveloppe permettant de grouper une série de déclarations.

Exemple:

```
  // stack.h

  package Stack
    const int MAX = 1000;
    void push (int i);
    int pop ();
  end Stack;
```

Les packages peuvent être imbriqués.

Les déclarations de packages peuvent, mais ne doivent pas, avoir un corps de package correspondant de même nom.

S'il est présent, le corps du package doit être déclaré dans la même région déclarative qui sera le fichier source .c si le package est déclaré dans le fichier .h.

Exemple:

```
  // stack.c

  package body Stack

    int table[MAX];
    int index = 0;

    public void push (int i)
    {
      table[index++] = i;
    }

    public int pop ()
    {
      return table[--index];
    }

  end Stack;
```

Les entités déclarées dans la déclaration de package sont automatiquement visibles à l'extérieur, ce qui peut créer plusieurs entités portant le même nom au sein de la même région. Toute ambiguïté doit être résolue en faisant précéder le nom de l'entité par le nom du package, i.e: PackageName. EntityName

Les entités déclarées dans le corps du package ne sont pas visibles à l'extérieur.

Exemple (un package pour un type opaque pour créer des piles multiples):

```
  // stacks.h

  package Stacks

    struct STACK;      // an opaque type
```

```
    void push (ref STACK s, int i);
    int pop (ref STACK s);

  end Stacks;


  // stacks.c

  package body Stacks

    const int MAX = 1000;

    public struct STACK
    {
      int table[MAX];
      int index;
    }

    public void push (ref STACK s, int i)
    {
      s.table[s.index++] = i;
    }

    public int pop (ref STACK s)
    {
      return s.table[--s.index];
    }

  end Stacks;


  // prog.c

  use stacks;

  void main()
  {
    STACK s;

    clear s;
    push (ref s, 12);
    printf ("%d\n", pop(ref s));
  }
```

==========

9. Génériques

Un package générique est un modèle de package dans lequel certains types (appelés types génériques) sont paramétrables.

Une partie générique formelle déclare une liste de types génériques, ainsi que des déclarations de fonctions génériques.

Exemple (un algorithme simple pour le tri):

```
  // 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;
```

Le corps d'un package générique a la même syntaxe que pour un package non-générique.

Exemple:

```
  // 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;
```

Exemple (un algorithme de gestion d'un arbre équilibré):

```
  // btree.h

  generic <KEY,ELEMENT>
    int compare (KEY a, KEY b);
  package BalancedTree

    struct BALTREE;     // opaque type

    int create (out BALTREE bt);
    int close  (ref BALTREE bt);

    int insert   (ref BALTREE bt, KEY k, ELEMENT e);
    int remove   (ref BALTREE bt, KEY k);
    int update   (ref BALTREE bt, KEY k, ELEMENT e);
    int retrieve (ref BALTREE bt, KEY k, out ELEMENT e);

  end BalancedTree;
```

Dans les exemples ci-dessus, KEY et ELEMENT sont les types génériques, Swapping, BubbleSort et BalancedTree sont les packages génériques.

Instanciation des packages génériques

Une instanciation générique crée une copie d'un package générique par le remplacement
de chaque type générique par un type réel et de chaque déclaration de fonction générique
par une déclaration de fonction réelle.

La liste d'associations génériques doit être donnée dans l'ordre dans lequel les identificateurs apparaissent dans la déclaration de package générique.

Exemple:

```
  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);    // doit être écrit Sort_int.sort si ambigu
  }
```

10. Unités de Compilation

10.1. Unités

Deux types de fichiers sources existent:

- .h : Interface
- .c : Corps

Un fichier source .h peut apparaître seul.

Un fichier source .c peut, mais ne doit pas, se présenter seul (ex: le programme principal).

Lorsque les fichiers source .h et .c apparaissent ensemble, ils doivent
se trouver dans le même répertoire.

10.2. Démarrage

Le point d'entrée dans un programme est une fonction appelée 'main' déclarée dans un fichier source .c :

- Il ne doit y avoir qu'une seule fonction de ce nom.

- Elle doit avoir le type de retour void ou int.

- Elle doit avoir, soit aucun paramètre, soit un seul paramètre de mode 'in' et
  de type string[] qui recevra une liste des paramètres de démarrage du programme.

Exemple:

```
  // hello.c

  from std use console;

  void main()
  {
    printf ("Hello World !\n");
  }
```

10.3. clauses d'import

Deux formes de clauses d'import existent:

a) import à partir de fichiers bibliothèque .lib

Exemple:

```
  from std use strings, console;
  from be.msc.webcam use cam/webcam;
```

Remarque: les fichiers de bibliothèque "std.lib" et "be.msc.webcam.lib"
doivent être présent dans l'un des répertoires listés dans le
fichier de configuration mk.cfg

b) import de fichiers sources .h

Exemple:

```
  use util, util2;   // "util.h" and "util2.h" dans le répertoire courant
  use cam/webcam;    // "webcam.h" dans le sous-répertoire "cam"
  use ../interface;  // "interface.h" dans le répertoire de niveau au-dessus.
  use /src/testing;  // ERREUR: les chemins absolus ne sont pas autorisés.
```

Note: les chemins absolus ne sont pas autorisés, sinon le projet ne peut
pas être déplacé facilement d'un répertoire à un autre.

Tous les identificateurs apparaissant dans les noms de la bibliothèque
et les noms de source doivent être en minuscules pour éviter les problèmes
de compatibilité entre systèmes de fichiers différents.

10.4. Alias

Au cas où deux unités avec le même nom sont importés (éventuellement à partir de deux
répertoires différents), une erreur se produit. Un nom d'alias peut alors être spécifié pour lever l'ambiguïté.

Exemple:

```
  from std use bintree;
  from std use new/bintree as bintree2;

  bintree2.insert ( .. );
```

10.5. Effet de l'import

Un import de l'unité B rend visible toutes les entités de B. Toutefois, il ne rend pas visible une unité C importée par B, bien que l'unité C avec ses entités soit chargée aussi, mais dans une région non visible. En bref: seuls les identifiants
d'unités directement importées sont visibles.

Si plusieurs unités de compilation dépendent les unes des autres
de façon circulaire, une erreur de compilation se produit.

Une erreur de compilation se produit si une unité s'importe elle-même ou si la même unité
est importée deux fois, même en utilisant des alias différents.

L'ordre des imports n'est pas importante (changer l'ordre des imports
ne crée jamais d'erreurs).

10.6. Compilation

Seul les fichiers source .h et .c et optionnellement un fichier de configuration (mk.cfg) doivent être fournies pour compiler un projet.

Exemple:

```
 C:\test>  mk hello
```

L'utilitaire make (mk.exe) compile "hello.c" et crée "hello.exe".

Les dépendances sont automatiquement tracées et les fichiers source correspondants sont recompilés s'ils en ont besoin. Un fichier de make manuel n'est pas nécessaire.

Tous les noms de fichiers source doivent être en minuscules.

10.7. fichier de configuration (mk.cfg)

Le fichier de configuration doit être placé dans le répertoire courant, ou
son nom doit être fourni au compilateur par une option de compilation;
il contient toutes les options du compilateur, symboles globaux de compilation et une liste des
répertoires pour rechercher les bibliothèques importées:

- Modèle de mémoire (32 ou 64 bits) (la valeur par défaut est 32)
  ceci a un effet sur la taille du pointeur, pointeur de fonction et
  pointeurs unsafe.

- Drapeau indiquant si les pointeurs sont sécurisés par
  un mécanisme de tombstone (par défaut c'est oui)

- Indicateur précisant si les indices de tableaux sont vérifiés
  (la valeur par défaut est oui)

- Indicateur précisant si les assertions sont vérifiées
  (la valeur par défaut est oui)

- Une liste des répertoires dans lesquels le compilateur recherche
  les bibliothèques importés.

Les symboles globaux suivants sont prédéfinis et ne peuvent être définis dans

mk.cfg :

```
  WINDOWS (1 pour version windows du compilateur, sinon 0)
  UNIX    (1 pour toute version unix du compilation, sinon 0)
  MEM32   (1 pour modèle de mémoire 32-bit, sinon 0)
  MEM64   (1 pour modèle de mémoire 64-bit, sinon 0)
```

Exemple:

```
  // mk.cfg

  [symbols]    ; symboles globaux de compilation (doivent être en majuscules)
  DEBUG   = 1
  T24     = 1
  WINDOWS = 1
  UNIX    = 0

  [options]
  memory_model    = 32         ; 32 or 64 (defaut = 32)
  pointer_check   = yes        ; defaut = yes
  array_check     = yes        ; defaut = yes
  assertion_check = yes        ; defaut = yes

  [library]              ; les bibliothèques sont recherchées dans les répertoires
suivants:
  dir = ./               ; par défaut si mk.cfg n'est pas fourni
  dir = c:/safe-c/lib/
```

==========

11. Problèmes de mise en oeuvre

11.1 Alignement des données

Les structures déclarées avec le mot-clé 'packed' sont les seules à être portables sur différents systèmes. Notez que tous les types simples sont 'packed' par défaut.

Avantage des types 'packed'

- les types 'packed' peuvent être convertis en byte[] en utilisant l'attribut 'byte.
- les types 'packed' peuvent être transmis à un paramètre formel de type byte[] (dans un
  appel de fonction)
- les types 'packed' peuvent être transmis à une liste de paramètres formels
  de type object[] (dans un appel de fonction)
- un paramètre formel de type 'packed' peut recevoir un paramètre réel
  de type byte[].

Les pointeurs et pointeurs de fonction ne sont pas autorisés à apparaître dans des structures 'packed' et être passés à des routines d'E/S parce que cela pourrait les corrompre.

11.2 Implémentation des pointeurs et objets heap

En interne, les pointeurs ne contiennent pas l'adresse de l'objet heap. Au lieu de cela, ils pointent sur une structure Tombstone qui garantit

un accès pointeur sécurisé.

11.3. Erreurs d'exécution

Quand une erreur se produit à l'exécution, le composant de bibliothèque "exception" va localiser l'emplacement précis l'erreur dans le code source et écrire un fichier correspondant CRASH-REPORT.TXT dans le répertoire courant de l'application.

---

12. Bibliothèques

La bibliothèque standard "std" contient les composants .h suivants :

```
aes          : cryptage AES (Advanced Encryption Standard)
bintree      : arbre binaire
bsearch      : recherche binaire (recherche d'élément dans un tableau trié)
calender     : calendrier (get_datetime)
clipboard    : presse-papiers
console      : printf et scanf pour les applications console
crc          : checksum md5, adler et crc
db           : base de donnée simple (fichiers ISAM)
des          : chiffrement DES
draw         : dessiner en mémoire
ebcdic       : conversion ebcdic
engine       : moteur 3D
exception    : gestionnaire d'exception
files        : fichiers, répertoires et disques
fixed        : type virgule fixe large
float1       : type en virgule flottante large
float2       : type en virgule flottante 2 large
ftp          : transfert fichier par protocole ftp
ftps         : serveur de protocole ftp
http         : client et serveur internet
image        : image (jpg, gif, png, tif, bmp) et opérations (copier, rotation)
inifile      : fichiers .ini
integer      : type entier large
interval     : intervalles
math         : fonctions mathématiques
net          : carte réseau (iprtrmib.h)
odbc         : base de données sql
printer      : imprimante
process      : processus
random       : nombres aléatoires
rsa          : cryptage rsa (clés asymétriques)
selfile      : sélectionneur de boîte de dialogue de fichier
service      : processus d'arrière-plan (services Windows)
sorting      : tri de tableau (bubbleSort, heapsort, quicksort)
sound        : micro, haut-parleur
strings      : support caractère et chaîne de caractères (sprintf, strcpy,
isdigit, ..)
tcpip        : couche tcp/ip, ipv4 et ipv6
text         : texte (stockage de lignes de texte)
thread       : threads, synchronisation, timers
tracing      : fichier de trace
url          : conversion d'urls
utf          : conversion ansi, utf-8, utf-16
webcam       : webcam (vfw, directx)
win          : interface utilisateur graphique
```

```
xml          : lecteur xml
zip          : zip, unzip
```

==========

Annexe A: support international

Les programmes traditionnels stockaient chaque caractère en un byte car ils étaient développés principalement pour les États-Unis ou l'Europe.

Cette situation a radicalement changé depuis que l'Unicode a normalisé tous les jeux de caractères. Actuellement, la norme Unicode définit 1.114.112 caractères à travers le monde.

A.1 Unicode pour usage interne

L'UTF-16 (type wchar) est utilisé pour stocker des caractères Unicode en interne.

A.2 Unicode pour une utilisation externe

Lorsque qu'on transfère de l'Unicode entre systèmes informatiques, un encodage dans ce qu'on appelle "chaînes de caractères UTF-8" est largement utilisé sur Internet. En UTF-8, chaque caractère Unicode est codé en 1 à 4 bytes:

Unicode Character	Byte1	Byte2	Byte3	Byte4
0 to 127	0xxxxxxx			
128 to 2047	110yyyxx	10xxxxxx		
2048 to 65535	1110yyyy	10yyyyxx	10xxxxxx	
65536 to 1114111	11110zzz	10zzyyyy	10yyyyxx	10xxxxxx

La conversion entre UTF-16 et UTF-8 est simple et disponible dans le composant standard "utf".

==========

Annexe B: conversion Unicode pour les fichiers source

Les fichiers sources peuvent être encodés soit en ANSI, UTF-8 ou UTF-16.

==========