
Programmer en Safe-C : un tutorial

Table des matières

---

1. Introduction

Le Safe-C est un descendant du C. Il vous permet d'écrire des programmes clairs et bien structurés. Les aspects dangereux du C ont été supprimés, ce qui facilite beaucoup la recherche d'erreur.

Ce tutorial est un résumé pour apprendre très rapidement le Safe-C.
Accrochez-vous !

2. Un programme simple

```
// hello.c : mon premier programme

from std use console;

void main()
{
  printf ("hello, world");
}
```

Un programme Safe-C consiste en un ou plusieurs fonctions. main() est une telle fonction, et en fait tous les programmes ont une fonction main() parce que c'est là que démarre le programme. En général la fonction main() va appeler d'autres fonctions.

Ici la fonction main ne contient qu'une seule instruction, printf() qui sert à afficher un texte sur la console. Notez que printf a été importé de la librairie standard "std" grâce à la commande "from std use console;". En effet, si vous allez voir dans le composant 'console', vous trouverez la définition de cette

fonction printf.

Pour appeler une fonction, on donne simplement son nom (par exemple printf), on donne d'éventuels arguments entre parenthèses ( ). S'il n'y a pas d'arguments, on ajoute simplement une paire de parenthèses vides ().

Les accolades { } entourent les instructions que la fonction exécute.
Chaque instruction se termine par un point-virgule (;)

Ce programme va donc afficher :

```
hello, world
```

3. Un programme Safe-C plus grand : variables et types

Voici un programme plus grand qui additionne 3 entiers et imprime leur somme :

```
// deux.c : mon premier programme

from std use console;

void main()
{
   int a, b, c, sum;

   a = 1;
   b = 2;
   c = 3;

   sum = a + b + c;

   printf ("sum is %d", sum);
}
```

Le Safe-C vous propose les types de variables suivants :

```
int    : les entiers
char   : contient 1 caractère (par exemple la lettre 'A')
float  : contient un nombre à virgule
double : idem que float mais en double précision
bool   : un booléen : contient un état vrai (true) ou faux (false)
```

Il y a aussi des tableaux, des structures et des pointeurs que nous verrons plus tard.

Toutes les variables d'un programme Safe-C doivent être declarées. Les déclarations doivent précéder les instructions qui les utilisent.
La déclaration :

```
int a, b, c, sum;
```

déclare les variables entières a, b, c, et sum.

Les noms de variables sont constituées de lettres A-Z, a-z, ou chiffres 0-9, ou du caractère souligné _ pour séparer des mots.

Exemple:

```
int    mon_total;
double f;
bool   b;
```

4. Constantes

Quand une valeur apparait plusieurs fois dans un programme, on peut le déclarer comme constante, ce qui vous facilite la vie quand vous souhaitez la modifier par la suite.

Exemple:

```
  const int    MAX_JOUEURS = 32;
  const double PI  = 3.1415927;
  const bool   DEBUG  = true;
  const char   NEWLINE = '\n';
  const string HELLO  = "Hello\n";
```

La séquence '\n' est une notation qui indique "aller à la ligne". Elle ne désigne en fait qu'un caractère et pas deux comme on pourrait le penser.

Il y a d'autres notations comme : '\t' pour tabulation, et '\\' pour le \ lui-même.

La constante nul (ou '\0') indique un caractère de valeur zéro.

5. Entrées/Sorties Simples

printf est en fait l'une des fonctions les plus riches pour formatter un texte. Elle permet de spécifier un nombre variable d'arguments dont le premier indique toujours le format.

Exemple :

```
  printf ("a = %d, b = %d, c = %d\n", a, b, c);
```

```
Premier argument   : "a = %d, b = %d, c = %d\n"
Deuxième argument  : a
Troisième argument : b
Quatrième argument : c
```

Le compilateur va vérifier que les arguments correspondent. Ici le format indique des %d, donc des entiers décimaux.

Cet exemple va imprimer :

```
  a = 1, b = 2, c = 3
```

On utilise %d pour des entiers, %c pour des char, %u pour des bool, %f pour des float ou double, %s pour des strings (tableaux de char).

6. If, operateurs relationnels, instructions composées

L'instruction de test simple est le "if" :

```
   char c;

   c = '?';

   if (c == '?')
     printf ("c contient un point d'interrogation !\n");
```

La condition à tester est inclue entre parenthèses. Elle est suivie d'une instruction à exécuter si la condition est vraie.

La séquence "==" indique un test d'égalité. En voici d'autres :

```
==      égal à
!=      pas égal à
>       plus grand que
<       plus petit que
>=      plus grand ou égal à
<=      plus petit ou egal à
```

Ces séquences peuvent être combinées avec les opérateurs :

```
&&      et (les deux doivent être vrai)
||      ou (au moins l'un des deux doit être vrai)
^       excl (l'un des deux doit être vrai mais pas les deux)
!       pas (doit être faux)
```

Par exemple nous pouvons tester si un caractère est blanc ou tab ou nouvelle ligne avec :

```
if (c == ' ' || c == '\t' || c == '\n')
  printf ("séparateur!");
```

S'il y a plus d'une instruction à exécuter, il est nécessaire de les entourer d'accolades.

Par exemple imaginez que vous avez deux variables a et b et que vous voulez les trier :

```
if (a > b)   // si a plus grand que b
{
  t = a;     // sauver a
  a = b;     // mettre b dans a
  b = t;     // restaurer dans b
}
```

A la fin de cette instruction if, a contiendra le plus petit des deux, et b le plus grand.

Notez qu'il n'y a pas de point-virgule après l'accolade }.

7. Instruction "While", assignation.

L'instruction de répétition de base est le "while".

Exemple:

```
void main ()
{
  int i;

  i = 0;

  while (i < 4)
  {
    printf ("i = %d\n", i);
    i++;
  }
}
```

Le "while" fonctionne comme suit :

(a) evaluer la condition booléenne (i < 4)
(b) si la condition est vraie, exécuter les instructions entre {}, ensuite revenir au (a).

Notez que comme la condition est évaluée avant d'exécuter les instructions, il se peut que les instructions ne soient jamais exécutées.

8. Arithmétique

Les opérateurs arithmétiques de base sont :

+	addition
-	soustraction
*	multiplication
/	division (avec troncature si les arguments ont le type int)
%	reste de la division entière

Exemple :

```
x = a % b;
```

Calcule le reste de la division de a par b et place le résultat dans x.

Le Safe-C permet d'effectuer des conversions (appelés aussi casts) pour convertir par exemple un type vers un autre, par exemple des char vers int.

Exemple:

```
int  i;
char c;

c = 'A';

i = (int)c;   // i va contenir le code ascii de c

c = (char)i;  // c va contenir le caractère correspond à la valeur i
```

Par exemple, pour convertir c en majuscule, on écrira :

```
if (c >= 'a' && c <= 'z')
  c = (char)((int)c - (int)'a' + (int)'A');
```

9. Clause "Else"

Dans une instruction "if" on peut ajouter une clause "else" pour exécuter une série d'instructions si la condition est fausse.

Par exemple si on veut calculer dans x la plus grande des valeurs de a et b, on écrira :

```
if (a > b)
  x = a;
else
  x = b;
```

Observez qu'il y a un point virgule après x=a.

Le "if" ci-dessus étant très fréquent, il peut être écrit sous une forme équivalente mais plus courte à écrire :

```
  x = (a > b) ? a : b;
```

ce qui se lit : si a > b, placer dans x le a, sinon le b.

Les "if" et "else" peuvent se combiner en plusieurs branches :

```
       if (...)
       {...}
       else if (...)
       {...}
       else if (...)
       {...}
       else
       {...}
```

Ces conditions sont testées dans l'ordre et un seul des blocs d'instructions sera exécuté : le premier pour lequel la condition est vraie. Si aucune n'est vraie le bloc final sera exécuté.

Par exemple, pour compter le nombre de lettres et chiffres d'un string, on écrira :

```
  void main()
  {
    int  nb_lettres, nb_chiffre, nb_autre, i;
    char c;
    const string phrase = "Bonjour, mon âge est 20 ans !";

    nb_lettres = 0;
    nb_chiffres = 0;
    nb_autre = 0;

    for (i=0; i<phrase'length; i++)
    {
      c = phrase[i];

      if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
        nb_lettres++;
      else if (c >= '0' && c <= '9')
        nb_chiffres++;
      else
        nb_autre++;
    }
  }
```

10. Opérateur d'incrementation et décrementation

i++ est équivalent à i = i + 1; mais plus clair : cela ajoute 1 à la variable. De la même façon il existe i--; pour diminuer de 1.

L'opérateur "++" peut aussi s'appliquer à des char, par contre il n'est pas autorisé pour les nombres flottants.

A noter qu'il existe ++i et i++ Quelle est la différence ? Et bien i++ calcule d'abord la value de i et ajoute ensuite 1 à la variable, tandis que ++i ajoute d'abord 1 à la variable et en calcule ensuite la valeur. Subtil ? voici un exemple :

```
  int i;

  i = 1;
```

```
  x = i++;
  // ici x vaut 1 (valeur avant le ++), et i vaut 2

  i = 1;
  x = ++i;
  // ici x vaut 2 (valeur après le ++), et i vaut 2.

  i = 1;
  printf ("%d, %d, %d", i++, i++, i++);   // va afficher 1, 2, 3
```

11. Tableaux

En Safe-C on peut créer des tableaux. Par exemple pour créer un tableau de dix int :

```
       int x[10];
```

Ici on crée un tableau dont les cases sont numérotées de 0 à 9 (les tableaux commencent toujours à zéro). On a donc créé les entiers suivants :
x[0], x[1], x[2], ..., x[9]

Pour accéder à une case du tableau, on écrit :

```
      x[0] = 5;    // mettre 5 dans la case 0
```

Ou plus intéressant :

```
      x[i] = 5;    // mettre 5 dans la case i du tableau
```

On aura une erreur si i n'est pas compris entre 0 et 9.

On peut prendre une tranche d'un tableau. Exemple si je veux recopier les 2 premiers int dans les 2 derniers :

```
     x[8:2] = x[0:2];   // recopie x[0] et x[1]  dans  x[8] et x[9]
```

La valeur avant le : indique le début de la tranche, la valeur après le : indique le nombre d'éléments.

Pour effacer le tableau (mettre tous les éléments à zéro), on écrira :

```
       clear x;
```

Si on veut mettre tous les éléments à 3 par exemple, on écrira :

```
       x = {all => 3};
```

Si on veut mettre les 5 premiers à 1 et les 5 suivants à 2 on écrira :

```
       x[0:5] = {all => 1};
       x[5:5] = {all => 2};
```

Pour interroger la longueur d'un tableau, on utilisera l'attribut 'length.

Exemple:

```
      longueur = x'length;   // longueur vaudra 10
```

On peut créer aussi des tableaux à plusieurs dimensions, par exemple:

```
int y[10][10];
```

Qui crée les 100 int suivants :
y[0][0], y[0][1], ... y[0][9], y[1][0], ..., y[9][9]

12. tableaux de char, strings

On stocke souvent du texte dans des tableaux de char. Pour délimiter le texte dans la variable tableau on utilisera un caractère 'nul' qui en safe-c n'est pas présent lorsque le tableau est entièrement rempli.

```
char s[5];

s = "Hello";  // ici le tableau est rempli

s = "Ici";    // erreur !!
```

Pour recopier un string dans un tableau de char, on utilisera la fonction strcpy() du composant de librairie 'strings'. Cette fonction va recopier un tableau dans un autre, et remplir les cases non-utilisées du tableau avec un caractère nul.

Exemple:

```
strcpy (out s, "Ici");    // s[0] = 'I';  s[1] = 'c';  s[2] = 'i';
                          // s[3] = nul;  s[4] = nul;
```

Lors de l'impression d'un string, on utilisera le code %s :

```
printf ("s = %s\n", s);
```

printf() sait qu'il doit s'arrêter, soit à la fin du tableau, soit lorsqu'il rencontre un caractère nul.

Un string constant se déclare comme ceci :

```
const string HELLO = "Hello";
```

Pour construire un string, utilisez la fonction sprintf(). Elle fonctionne comme printf() mais au lieu d'écrire le résultat sur la console elle stocke le résultat dans une variable donnée comme premier paramètre.

Exemple:

```
char buffer[1024];
sprintf (out buffer, "a = %d", a);
```

Pour comparer deux strings par ordre lexicographique, on utilise la fonction strcmp() qui est disponible dans le composant 'strings'. strcmp() renvoie +1 si le premier string est plus grand, 0 s'ils sont égaux, ou -1 si le premier est plus petit.

Exemple:

```
if (strcmp (s1, s2) > 0)
  ...
```

13. Instruction "For"

L'instruction "for" est un "while" condensé. Au lieu d'écrire :

```
initialization;
while (condition)
{
  instruction;
  increment;
}
```

On écrira :

```
for (initialization; condition; increment)
{
  instruction;
}
```

Ce qui permet de gagner 2 lignes.

A noter que les 3 parties du "for" sont optionnelles, ce qui dans sa forme la plus simple se résume à :

```
for (;;)
{
  instruction;
}
```

pour faire une boucle qui tourne indéfiniment. A noter que s'il n'y a pas de condition, elle est vraie par défaut.

14. Fonctions

Dans un programme plus important, vous découperez les blocs logiques en fonctions :

```
void func1 ( ... )
{
  ...
}

void func2 ( ... )
{
  ...
}

void main ()
{
  ...
}
```

La fonction main pourra appeler une fonction func1 ou func2 comme vous l'avez vu avec printf().

Il faut déclarer une fonction avant de pouvoir l'appeler, c'est pour ça qu'elle apparaisse dans l'ordre inverse des appels. Il est cependant possible de pré-déclarer une fonction pour pouvoir l'appeler avant de la déclarer.

Exemple:

```
void func2 ( ... );  // pré-déclaration (remarquez le point-virgule et l'absence
                     // d'accolades

void func1 ( ... )
{
  ...
  func2 ( ... );     // ici on peut appeler func2
}

void func2 ( ... )
{
  ...
}

void main ()
{
  ...
}
```

Si votre programme devient très grand, vous aurez besoin de le découper en plusieurs fichiers .c Il faut alors créer aussi des fichiers .h pour spécifier au compilateur les interfaces entre les fichiers.

Exemple:

```
// util.h

void func ( ... );  // pré-déclaration (remarquez le point-virgule et l'absence
                    // d'accolades)
```

```
// util.c

public void func ( ... )     // remarquez ici le mot-clé "public" pour indiquer
                             // que cette fonction pourra être appelée
{                            // à partir d'un autre fichier que util.c
  ...
}
```

```
// hello.c

use util;     // ici on importe util.h avec une clause "use"
              // pour pouvoir appeler func()

void main ()
{
  func( ... );
}
```

Les fichiers .h peuvent contenir uniquement des pré-déclarations de fonctions (remarquez le point-virgule et l'absence d'accolades), ainsi que des déclarations de variables globales et de types.

Une fonction peut comporter des paramètres qui permettent de passer une valeur

à la fonction.

Exemple:

```
void func1 (int i, char c, bool b, float f, string s)
{
  ..
}

void main ()
{
  int i;
  func1 (i+2, 'A', true, 1.345, "Hello");
}
```

Il existe aussi des paramètres "out" qui permettant à la fonction de renvoyer un résultat :

Exemple:

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

void main ()
{
  int i, j, k;
  i = 1;
  j = 2;
  additionner (i, j, out k);
  printf ("resultat = %d\n", k);
}
```

Finalement il existe des paramètres "ref" (par référence) qui permettant de passer une variable à modifier :

Exemple:

```
void ajouter10 (ref int a)
{
  a += 10;
}

void main ()
{
  int i;
  i = 1;
  ajouter10 (ref i);
  printf ("i = %d\n", i);
}
```

Jusqu'ici les fonctions commençaient toujours par le mot-clé "void", mais elles peuvent aussi renvoyer une valeur simple (valeur simple signifie pas de tableau ou de struct).

Exemple:

```
int additionner (int a, int b)
{
  return a + b;    // envoie le résultat grâce au mot-clé "return"
```

```
}

void main ()
{
  int i, j, k;
  i = 1;
  j = 2;

  k = additionner (i, j);

  printf ("resultat = %d\n", k);
}
```

15. Variables bloc, locales et globales

On peut déclarer des variables à différents endroits.

Variables locales :

```
   void main ()
   {
     int i;
     ...
   }
```

Variables dans un bloc accolade:

```
   void main ()
   {
     ...

     if (...)
     {
       int i;
       ...
     }
   }
```

Une telle variable n'existe que tant que le programme exécute le bloc accolade, elle perd sa valeur dès qu'on sort du bloc.

Variable globale :

```
   int i;

   void main ()
   {
     ...
   }
```

Une variable globale 'existe' pendant toute la durée du programme.

A remarquer qu'il n'est pas autorisé de déclarer une variable globale après une fonction :

```
   void func ()
   {
     ...
   }
```

```
   int i;   // erreur !
```

Il faut déclarer toutes les variables globales en haut du fichier.c, juste après les clauses "use".

On peut cependant les déclarer tout de même après une fonction en utilisant un package :

```
   void func ()
   {
     ...
   }

   package Compteur
     int i;
   end Compteur;
```

Variable externe:

```
  // util.h
  int g_counter;
```

Une variable externe 'existe' pendant toute la durée du programme. Elle est disponible pour toutes les parties du programmes qui importent util.h via une clause "use".

16. Les pointeurs

Vous avez vu au chapitre précédent comment on déclare une variable.

Celles-ci ont toutefois certaines limitations :
- le nombre de variables est figé lors de l'écriture du programme;
- la taille des tableaux est constant.

Les pointeurs permettent de passer outre ces limitations, ils introduisent cependant une complexité accrue, donc ne les utilisez que si c'est nécessaire.

Voici une utilisation d'une variable i :

```
  void func ()
  {
    int i = 2;
    printf ("i = %d\n", i);
  }
```

Maintenant voici le programme équivalent avec un pointeur :

```
  void func ()
  {
    int^ pi;

    pi = new int ' (2);

    printf ("i = %d\n", pi^);

    free pi;
  }
```

Remarquez qu'on a déclaré une variable pi (pointeur vers un int), avec la notation ^.

L'allocateur 'new' va réserver dynamiquement un 'int' en mémoire, le remplir avec la valeur 2, et stocker l'adresse mémoire du 'int' dans la variable 'pi' (pointeur vers i).

Pour utiliser notre variable (dans le printf), nous devrons passer par le pointeur et ajouter le symbole chapeau ^ pour dire que nous voulons accéder à "la variable pointée par pi".

Et enfin le plus important, quand vous avez fini d'utiliser la variable, vous devez libérer son espace avec l'instruction free. Si vous l'oubliez vous finirez par consommer toute la mémoire du programme ce qui va provoquer une erreur et un arrêt.

Alors évidemment tout ceci est beaucoup plus lourd. A quoi cela sert-il ?

Et bien cela devient utile par exemple lorsqu'on veut déclarer un tableau d'une longueur qu'on ne connait pas à l'avance.

```
  void func (int len)
  {
    int^ pi;

    pi = new int [len];

    .. // ici j'accéde aux éléments du tableau via :  pi^[i]

    free pi;
  }
```

Autre utilité des pointeurs : si on veut garder le tableau même quand la fonction se termine. Cela ne pose aucun problème, il suffit de stocker le pointeur dans une variable. Tant que vous ne faites pas de free, le tableau reste à votre disposition.

Voici un autre exemple:

```
  void func ()
  {
    int^ pi;

    pi = new int;

    pi^ = 1;

    pi = new int;

    free pi;
  }
```

Ici on réserve un 'int' via 'new', on stocke la valeur 1 dedans, et ensuite on réserve un autre 'int' via 'new' dans pi. Cela signifie qu'on a perdu le pointeur vers le premier 'int' et qu'on ne pourra plus y accéder ni le libérer avec 'free' ! C'est un exemple de ce qu'il ne faut pas faire.

Deux pointeurs peuvent par contre pointer sur la même variable, cela ne pose aucun problème.

```
  void func ()
  {
    int^ p1, p2;
```

```
    p1 = new int;
    p2 = p1;     // maintenant p1^ et p2^ designent le même 'int'

    free p1;
    free p2;     // erreur !! la variable a déjà été libérée !
                 // il ne faut qu'un seul free.
  }
```

Quand on veut qu'un pointeur ne pointe vers rien du tout, on lui assigne la valeur null (à ne pas confondre avec le caractère nul) :

```
   p1 = null;
```

Les pointeurs permettent surtout de construire des listes en mémoire.
Il suffit qu'un pointeur pointe sur un noeud, et que chaque noeud pointe sur le noeud suivant, le dernier pointant sur null, pour en constuire une.

Nous renvoyons pour cela à des manuels de C qui vous expliqueront cela en détail.

17. instruction "switch", "break", "continue"

L'instruction "switch" permet de remplacer certains "if" à plusieurs branches.

Quand votre "if" sert à comparer une valeur à une série de constantes comme ceci :

```
       if (c == 'a') ...
       else if (c == 'b') ...
       else if (c == 'c') ...
       else ...
```

alors vous pouvez remplacer ce code par :

```
       switch (c)
       {
         case 'a':
           ...
           break;

         case 'b':
           ...
           break;

         case 'c':
           ...
           break;

         default:
           ...
           break;
       }
```

Le branche 'default' est exécutée quand c n'est égal à aucune des valeurs.

L'instruction "break" s'utilise aussi dans les instructions "for" et "while" et sert à quitter immédiatement la boucle.

L'instruction "continue" permet d'exécuter immédiatement le tour de boucle suivant sans exécuter le reste des instructions.

18. Structures

Les structures sont utilisées pour grouper des variables en une collection.

Exemple:

```
struct ETUDIANT
{
  string(120) nom;
  string(80)  prenom;
  int         age;
  char[6]     matricule;
  bool        boursier;
  string(200) addresse;
}
```

Une fois cette structure déclarée, on peut déclarer une variable 'e' de type ETUDIANT comme ceci :

```
ETUDIANT e;
```

Pour accéder aux différents champs de la structure, on utilise un point :

```
e.age = 20;
e.matricule = "E001234";
```

Les structure se combinent avec les tableaux aussi, par exemple si on veut déclarer une variable de 100 étudiants :

```
ETUDIANT classe[100];

clear classe;    // la commande 'clear' permet d'effacer toute une variable

classe[5].age = 3;
```

Imaginez qu'on veuille écrire un programme qui calcule le nombre d'étudiants de moins de 20 ans :

```
int i, count;

count = 0;

for (i=0; i<classe'length; i++)
{
  if (classe[i].age < 20)
    count++;
}

printf ("nbre d'étudiants de moins de 20 ans : %d\n", count);
```

Si vous ne savez pas combien d'étudiants vous aurez lors de l'écriture du programme, vous pouvez réserver un tableau avec un pointeur :

```
void reserver (int nombre)
{
  ETUDIANT[]^  classe;

  classe = new ETUDIANT [nombre];

  classe^[0].age = 2;

  ...
```

Remarquez l'instruction ci-dessus qui combine "^", "[]" et ".", les trois

mécanismes pour accéder à un pointeur, un tableau et une structure respectivement.

19. Initialization des variables

Les variables globales sont toujours initialisées par défaut avec des zéros. Vous devez par contre initialiser les variables locales vous-même, sinon le compilateur va vous donner une erreur. Cela se fait généralement par une assignation :

```
int i = 2;  // assignation
```

ou par une instruction clear qui met la variable à zéro :

```
int[100]  table;

clear table;
```

ou par un aggregat :

```
struct EMPLOYE
{
  char[6]     matricule;
  int         age;
}

EMPLOYE e;

e = {matricule => "123456",       // aggregat pour type struct
     age       => 40};
```

20. Directives #if #endif

Pour enlever temporairement un bloc de code d'un programme, vous pouvez l'entourer des directives #if 0 et #endif

Exemple:

```
    a = 1;
    printf ("a=%d\n", a);

#if 0           // morceau de code enlevé
    a = 2;
    printf ("a=%d\n", a);
#endif

    a = 3;
    printf ("a=%d\n", a);
```

Pour le remettre, il vous suffit de changer le #if 0 en #if 1

Vous pouvez aussi définir une valeur macro booléenne pour enlever plusieurs morceaux en une fois, cependant cela ne fonctionne qu'à l'intérieur d'un même .c :

Exemple:

```
#define debug  0    // 0 pour enlever, 1 pour remettre
```

```
    a = 1;
    printf ("a=%d\n", a);

  #if debug
    a = 2;
    printf ("a=%d\n", a);
  #endif

    a = 3;
    printf ("a=%d\n", a);

  #if debug
    a = 4;
    printf ("a=%d\n", a);
  #endif
```

21. Operations sur les Bits

Le Safe-C possède plusieurs opérateurs opérant au niveau "bit" :

|	inclusive OR
^	exclusive OR
~	(tilde) 1's complement
!	logical NOT
<<	left shift
>>	right shift

avec la même signification qu'en C.

22. Assignation

L'instruction d'assignation peut être combinée avec l'addition, par exemple:

```
x += 10;   // ajoute 10 à x
```

ou:

```
x -= 10;   // enlève 10 de x
```

De la même façon il existe : *= /= %= &= |= ^= <<= >>=

23. Nombres à virgule flottante

Exemple:

```
from std use math;

void main()
{
  double sum;
  double tab[10];
  int    i;

  for (i=0; i<tab'length; i++)
    tab[i] = sin ((double)i * PI / 180.0);

  sum = 0.0;
  for (i=0; i<tab'length; i++)
    sum += tab[i];

  avg = sum / (double)tab'length;
```

```
}
```