CHAPITRE

4

Les bases du langage C

4.1 Nature & composantes

4.1.1 Introduction

L'ordinateur accepte une série d'instructions couchées dans un langage qui lui est propre : l'assembleur. La somme de ces instructions forme un *algorithme*. L'assembleur est donc un langage *interprété* par le processeur.

L'assembleur, langage-machine, est codé à même l'unité centrale de logique et d'arithmétique (ALU ou UCLA). Il permet donc d'effectuer des *tests de logique*, des comparaisons, des opérations d'addition, de soustraction, etc.

> Un algorithme transcrit en assembleur est un logiciel, exécutable par l'ordinateur.

L'assembleur est un langage basique, simple (une quarantaine d'instructions) $\Rightarrow$ par conséquent il faut beaucoup d'instructions en assembleur pour coder un algorithme, même l'algorithme d'une tâche simple.

Un langage syntaxique de haut niveau, tel que le C, permet de transcrire un algorithme plus facilement. Toutefois il devient nécessaire de le convertir (traduire) en langage-machine.

convertir $\Longrightarrow$ compilation : le C est un langage compilé, non interprétable directement par l'ordinateur.

Le C, son organisation, sa structure

En principe la réalisation d'un algorithme ne nécessite qu'une liste détaillée des opérations que nous désirons effectuer. On distinguera trois types de langages selon leur niveau de complexité.

a) *langage instructionnel*, il regroupe un ensemble d'instructions pour une application bien précise. Son utilité se limite à la tâche visée. Par exemple l'assembleur est un langage de ce type. Exemple d'utilisation : les opérations de mise en route de l'ordinateur (BIOS) sont codées en assembleur.

b) *langage fonctionnel*, qui permet d'effectuer des opérations répétitives sur des données variées, en regroupant puis identifiant les instructions pertinentes à effectuer. Le C est un langage de ce type, tout comme le Fortran ou le Pascal.

c) *langage orienté objet*, type le plus évolué, permet de regrouper des familles de fonctions et des paramètres pour une modularité parfaite. Le C++, ainsi que Java et ses dérivées, en sont des exemples.

Premier pas vers la programmation

Pour exécuter un programme, il faut au minimum :

$\rightarrow$ une ou plusieurs *instructions* à exécuter. Les opérations sont effectuées les une après les autres. Il faut donc pouvoir les identifier, les reconnaître.

En C, le point-virgule (;) indique la fin d'une instruction. Il sert de séparateur entre deux instructions.

$\rightarrow$ un regroupement en *blocs* de ces instructions

On utilise les accolades $\{$ et $\}$ pour indiquer le début et la fin d'un bloc d'instructions. Donc les accolades : toujours en paires.

En C, l'*expression* est une suite de composantes élémentaires, de syntaxe correcte, reconnues par le langage. (Voir liste de mots-clefs.)

Par exemple $a = 1+2$ est une expression où apparait une opération d'addition (+) et une opération d'affectation (stockage) du résultat (=) à la variable a. En ajoutant un ' ;'

$$a = 1 + 2;$$

nous formons une *instruction* du langage. *Deux ou plusieurs* expressions peuvent être mise bout à bout pour former une instruction : il suffit de les séparer par une virgule. Par exemple

$$\{a = 1 + 2, b = \sin(a);\}$$

forme un *bloc* d'une instruction : la virgule sert de séparateur entre deux expressions. Lecture : de gauche à droite !

4.1.2 Hiérarchie du langage C :

Les différents constituants du langage peuvent être mis en ordre croissant de complexité :

$$\boxed{\text{expressions} \rightarrow \text{instructions} \rightarrow \text{bloc} \Rightarrow \text{fonction} \Rightarrow \text{Algorithme}}$$

Propriétés générales des fonctions :
- une fonction peut être réutilisée séquentiellement
- Elle peut accepter un ou plusieurs *arguments*
- toute fonction a un identifiant (un nom, par exemple : *main, sin, ...*)
- dans un programme complet : une seule fonction appelée *main*

Pour pouvoir effectuer toutes les opérations, calculs, etc sans faute, il faut préciser la nature de toute quantité ou variable apparaissant dans une instruction, voire à l'intérieur d'une fonction. On parlera de type de variables et fonctions. Ainsi il est nécessaire de donner un type aux fonctions elles-mêmes. Les différents types sont discutés plus loin.

Signature d'une fonction : syntaxe

< type > nom (< type 1 > variable 1, <type 2 > variable 2 ...)
{
instructions
}

Composantes syntaxique du langage

Pour rédiger une fonction (disons `main`), il nous faut connaître les composantes syntaxique du langage.

Il existe deux saveurs de ces composantes :

(a) les *identifiants*, c'est-à-dire
- essentiellement des noms (de variables, fonctions)
- tous les types personnalisés définis par l'utilisateur

– composés de lettres (a-Z), chiffres (0-9) et le blanc souligné (_)

Restrictions
– le 1ier symbole d'un identifiant ne peut être un chiffre
– ne peut être un mot–clef (voir plus bas)
– longueur max : dépend de l'ordinateur (normalement > 31 symboles).
– majuscules/minuscules différenciées

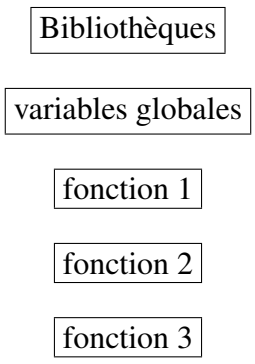
(b) les *mots-clefs* du langage (voir liste)

4.1.3 Anatomie d'un programme C

Les composantes d'un programme C

1. bloc d'instructions principal : `main`
   – Unité de base du programme
   – Point de départ à l'exécution
   – 1 ou plusieurs bloc(s) d'instructions
2. bloc d'instructions dérivés ou secondaires : les *fonctions*
   $\Rightarrow$ un bloc d'instructions s'appelle aussi *fonction* lorsqu'il est précédé d'un identifiant (nom).
3. déclaration de variables (1) locales (2) globales
   $\Rightarrow$ le C est un langage dit *typé* parce que chaque variable ou paramètre de l'algorithme doit avoir été déclaré et affecté d'un type spécifique avant d'être initialisé (voir §4.2).
4. directives au compilateur (logiciel de compilation).
5. bibliothèques (locales ou extérieures).
   $\Rightarrow$ il s'agit de données supplémentaires, standardisées, que nous pouvons inclure implicitement dans notre algorithme.

Organigramme d'un programme C



⋮

fonction principale ou *main*

4.2 Les variables en C

4.2.1 Types et définitions

– Rappel : *variable*
⟶ au sens mathématique, une inconnue.
Par exemple,

$$2x + 2 = 1 \Rightarrow x = -1/2$$

comme résultat unique.
⟶ au sens informatique : il s'agit de la représentation d'une adresse mémoire.
– Une affectation à une variable = rangement d'une valeur à cette adresse

Particularité du C : il faut déclarer toute variable **avant** d'y faire référence.

4.2.2 Déclaration de variables : syntaxe

La déclaration : voir liste des mots-clefs.
Notation : $<>$ pour 'obligatoire' ;[ ] pour facultatif

[Classe d'allocation] $<$ type $>$ $<$ $identifiant$ $>$ [= valeur]

Types de variables : gestion plus efficace de l'espace mémoire, différents types occupant chacun un espace mémoire qui lui est propre. Ceci tient de ce que les nombres et symboles (lettres, chiffres, etc) sont codés dans une base *binaire*. L'espace mémoire d'une variable d'un type spécifique peut donc se mesurer en bits d'information, ou, plus couramment, en *octets* (1 octet = 8 bits).
Tour à tour :

1. Classe d'allocation : elle spécifie la *portée* d'une variable dans le programme.
   a auto : espace mémoire fragile, effacé après complétion d'une suite d'instructions. Variable **non** initialisée.
   b static : préservée en mémoire, durée de vie = celle du programme. Initialisation à = 0.
   c extern : variables globales, hors fonctions. Similaire à static.

2. type : 8 mots-clefs
   `char, int, short, long, float, double, signed, unsigned`

   les plus communs sont

   a char (symbole, lettre)
   tous les caractères d'un clavier
   ASCII : codé 7 bits soit 2^7 = 128 symboles
   ISO : 8 bits ou 2^8 = 256
   ISO/IEC : sur 32 bits ou $2^{32} = 4294967296 \sim 4 \times 10^9$ (la table inclue symboles anciens ou hieroglyphiques, etc ..)

   b int (entier), nombres de l'ensemble N.
   long : 32 bits ou 64 bits ; short : 16 ou 32 bits (dépend des ordi)
   Par exemple, sur 4 octets (=32 bits) : $[-2^{31}, 2^{31}[$ (entiers signés)
   alors que pour les entiers non-signés (unsigned) : $[0, 2^{32}]$

   c float (flottant) : les réels (précision finie).
   double $\longrightarrow$ comme long pour les entiers.
   Par défaut : float = précision simple (32 bits) et double = 64 bits

   L'utilitaire SIZEOF détermine ces grandeurs en octets sur l'ordinateur.
   Par exemple, pour obtenir le nombre d'octets représentant un flottant double précision, SIZEOF `(long double) = 12` sur ordinateur Apple G3.

4.2.3 Déclarations de variables et leur portée

Portée	variable existe dans ..
programme	tous les fichiers et fonctions constituant le programme dans son entier
fichier source	de l'endroit où elle est déclarée jusqu'à la fin du fichier. Donc : mettre les déclarations au haut du fichier pour éviter erreurs possibles.
fonction	du début à la fin de la fonction, ie en déclaration, comme argument.
bloc d'instructions	de la déclaration et jusqu'à la fin du bloc

Classes d'allocation

Classe	Variable déclarée dans ..		
	fichier	argument de fonc	bloc d'instruction
auto register	impossible	portée : fonc / dyn	portée : bloc /dyn
static	portée : fichier/statique	impossible	portée : bloc/statique
extern	portée : prog/statique	impossible	portée : bloc/statique
Défaut :	portée : prog / statique (≡ extern)	bloc/dyn	bloc/dyn (≡ auto)

4.2.4 Entier ou caractère .. ?

En C : un caractère (lettre, chiffre) est traité comme un entier (correspondance au tableau ASCII).

Par exemple, considérez le bloc d'instructions suivant,

```
{ char c = 'A' ;
c = c + 1 ;
printf( ``%c'', c ) ;}
```

Ceci donne comme résultat l'impression de la lettre b à l'écran. On aurait pû aussi bien écrire

```
{ char c = 65 ;
c = c + 1 ;
printf( ``%c'', c ) ;}
```

pour obtenir le même résultat. Le 'A' majuscule correspond en effet à l'entier 65 (voir votre Table ASCII distribuée en TD).

4.2.5 Les constantes : propriétés, notation

Les constantes du langage C sont typées, comme les variables, mais leur déclaration est implicite. Les contantes comprennent :

1. *Entier* : base décimale ou hexadécimale (base 16)
   on peut préciser la *taille* de l'entier en utilisant la notation adéquate
   1234 : entier normal en base décimale (précision simple)
   1234U : unsigned integer (entier non-signé, donc *positif*)
   1234L : long int, donc entier long, permet de traiter des nombres grands
2. *Réel* : point décimal et exposant
   Il respecte la convention usuelle des nombres ; par exemple : 12.34 = douze virgule trente-quatre, mais
   12.3e-4 veut dire 0.00123
   12.34F (F pour float)
   12.34L (pour long double, plus de chiffres après le point)
   Les entiers ou réels *long* ou *double* occuperont un espace mémoire deux fois plus grand que leur équivalent standard, car ils sont codé avec deux fois le nombre d'octets.
3. *Character* (= 'Caractères') : les lettres, mais aussi tout symbole du clavier
   Par exemple, les lettres sont notées 'a', 'b', ... et les symboles tels que '#', '%', ..
   Les exceptions :
   – Le symbole \ et l'apostrophe ont tous deux besoin d'une notation spéciale, c'est-à-dire ⇒ \\, \' (même chose pour les guillemets, ¨).
   – Le point d'interrogation (?) est aussi évoqué en le précédant d'un slash → \ ?
   Les constantes spéciales : Ces symboles, non-imprimables, se retrouvent par leur code hexadécimal. En principe, tous les symboles et constantes spéciales peuvent s'exprimer en base hexadécimale. Par exemple, le symbole étoile (*) devient dans cette base `\x2a`
   Quelques symboles spéciaux importants, qui n'ont pas d'équivalent graphique mais peuvent être évoqués par construction d'un symbole composé :
   (a) `\n` : nouvelle ligne
   (b) `\t` : tabulation horizontale
   (c) `\a` : alerte (sond de cloche)
   (d) `\v` : tabulation verticale
   (e) `\r` : retour (du chariot)
4. Chaînes de caractères : Une chaîne est une expression de plusieurs caractères, entourés de guillemets.
   Par exemple, `'' Ceci est une chaîne .. ''`

Les guillemets : bornent une chaine de caractères. Une chaîne de caractères se termine toujours par la constante spéciale '\0', laquelle est ajoutée à la chaîne *automatiquement* lors de sa création ou de sa lecture (saisie). Ce symbole sert donc de point de repère pour terminer une chaîne.

Dans une chaîne, les guillemets s'expriment comme ceci ..

```
'' Ceci est une \''chaîne\'' avec
 guillemets à l'intérieur .. ''
```

4.2.6 Conversion de types, le cast

Type de variable = affectation d'espace mémoire spécifique. Donc pour effectuer un calcul de manière auto-cohérente et sans perte d'information, il faudra s'assurer que les variables utilisées ont

- le type adéquat lors de leur déclaration
- des types identiques ou équivalent, qui permettront de faire des opérations entre elles.

Pour s'assurer que les variables sont de même type, par exemple pour effectuer une opération mathématique, on peut *convertir* le type d'une variable au moyen d'un utilitaire du langage. le *cast* . Considérez l'exemple suivant.

Soit

```
float x = 17 ;
```

un flottant auquel est affecté un *entier*, 17 : par conversion de type nous avons alors x = 17.0000, un réel (point décimal). Cette conversion est faite automatiquement par le signe d'égalité (=) ! L'affectation (=) effectue des cast implicite pour les types simples du langage. Par contre la division

```
{ x = 17 / 2 ; printf( ''%d'', x ) ; }
```

donnera 8.0000 comme résultat. La division entière tronque le résultat 8.500. On peut corriger en applicant un cast au chiffre 17 pour le convertir en flottant :

```
{ x = ( (float)17 ) / 2 ; }
```

donne alors $x = 8.5$, le bon résultat. Pourquoi ? Tout entier admet une représentation binaire *exacte*, alors que les réels non. Pensez à ceci : le réel 1. est, au sens mathématique, suivi d'un nombre *infini* de zéros – alors que le flottant x ici prend 4 chiffres après le point seulement. Ainsi la division d'un réel par un entier fait intervenir deux nombres de précision différente – le résultat de la division est du type ayant la plus mauvaise précision : ici, réel. Il n'y a donc plus de conversion par le signe d'égalité comme avant. À revoir en TD.

4.3 Les opérateurs du langage

Opération ⇒ relation entre variables et constantes ...

Syntaxe : variable ou expression *opérateur* variable ou expression

4.3.1 Les différents opérateurs

1. L'*affectation* : c'est le signe '=' ; donc la syntaxe inclue une conversion de type implicite, puisqu'il y a rangement de la solution à l'adresse de la variable, c'est-á-dire l'*espace mémoire* occupé par cette variable. L'espace mémoire occupé par une variable dépend de son type (exemple : long int et short int, entier long et entier court, n'occupe pas le même espace puisqu'ils sont codés avec un nombre d'octets (1 octet = 8 bits d'information) dans la mémoire de l'ordinateur.
2. Les *opérateurs arithmétiques* (+, -, *, /) ainsi que le résidu (%).
   Par exemple, x = 17%2 ⇒ x = 1.
   Ce dernier opérateur (%) ne s'applique qu'à des types entiers.
   Remarque : il n'y a pas d'opérateur pour élever un nombre à une puissance donnée (par exemple, x^2). Il faudra faire appel à une fonction pour élever un nombre à une puissance donnée.
3. Les *opérateurs relationnels* ($>, <, >=, <=, ==, !=$).
   Exemple : les opérations *Si (a = b) instruction # 1 ; Sinon : instruction # 2* se transcrivent en C par
   { int a = 0, b = 1 ; if (a == b) printf("\n a = b ") ;
   else printf("\n a .ne. b ") ; } La phrase 'si a = b ..' sous-entend une *comparaison* entre la valeur numérique de la variable a et celle de la variable b, *pas* une affectation vers a. Si d'aventures vous oubliez de doubler le symbole d'égalité, alors le C fera une affectation et non le test de logique.
4. *Opérateurs de logique booléenne* : && (et), || (ou), ! (négation)
   Test logique :
   'Si ... ' : Si((i>0) && (i<9) && (i !=5)) alors ..
5. *Affectation composée* : aux opérateurs arithmétiques, ajouter le '=' :
   Exemple :
   x += 2 est la même chose que x = x + 2
   x /= 3 est la même chose que x = x / 3
6. *Incrémentation / décrémentation* (++ / - -)
   Exemple :

`a++;`

est une instruction équivalente à

`a = a + 1;`

De même on peut écrire

`b = a++;`

qui veut dire

b = a ;

a = a + 1 ;

c'est-à-dire l'affectation ne se fait qu'avant l'incrémentation de a. Sinon : inverser l'ordre,

`b = ++a;`

7. *Opérateur virgule* : permet de combiner des expressions.
   exemple : b = ((a = 3), (a+2)) ;
   donne
   b = 5 ;
8. *Opérateur conditionnel ternaire :* si (condition) alors *expression 1* : sinon *expression 2*
   Considérez l'exemple suivant :
   `{ y = ( x >= 0 ) ? x : -x; }`
   qui affecte la valeur absolue de x à y. Lorsque la condition est satisfaite, l'expression qui précède les deux points est retournée comme résultat, sinon celle qui suit les deux points.

Jusqu'à présent nous n'avons appris qu'à composer des pseudo-code. Nous avons vu l'importance de

1. identifier variables et paramètres
2. résoudre toutes les difficultés de l'algorithme de résolution
3. structurer son programme .. c'est-à-dire respecter les directives de l'algorithme, réfléchir 'comme un ordinateur'.

Maintenant : transcrire et exécuter un programme.
Exemple du TD no. 1.

4.3.2 Pseudo-code du TD 1

```
/* STUSM TD no 1 : solution \'a l'exo 2 */
/* Une biblioth\'eque syst\'eme */
inclure biblio entr\'ees/sorties pour interfa\c{c}age
```

```
main () {
declarations de variables et constantes :
entier H initialiser a 1610 (parametre du probleme) ;
entiers date, x et y (x,y sont les inconnus);
date : saisie de donnee (formattee);
date : doit etre < a 1664 (disons) et > a H=1610
(sinon : refaire la saisie de date ; )

ensuite : isoler pour x,y etant donne date, H;

finalement   envoyer les resultats a l'ecran ;
terminer programme  ;
}
```

4.3.3 Opérateurs du langage : TD 1

Nous connaissons déjà les mots-clefs du langage. Il faut les mettre en œuvre.

1. L'*affectation* : correspond à mettre en mémoire.
   On en connait des exemples, comme les déclarations ou des expressions arithmétiques
   x = date - H ; y = 2 * x ;
2. Les opérateurs booléens (logique) ; dans notre exemple
   " plus petit que H ou plus grand que 1664" devient $\longrightarrow$ date $<$ H $||$ date $>$1664
3. Un autre opérateur utile est la virgule : elle permet des expressions plus compactes

   ```
   /* op\'erateur virgule pour rabouter deux expressions */
    y = ( x = date - H , 2*x ) ;
   ```

4. Autres exemples suivront ..

4.3.4 Priorité des opérateurs

Les opérations entre variables et constantes sont hierarchiques : elles ne s'effectuent pas dans un ordre arbitraire.

Règle d'or : les parenthèses = plus haute autorité. On peut toujours remplacer les parenthèses par un résultat, une variable : elles sont donc à évaluer en premier.

Parenthèse $\rightarrow$ arithmétique $\rightarrow$ relationnels $\rightarrow$ booléen

La table qui précède résume l'ordre de priorité des opérateurs. Parmi les opérations arithmétiques, il existe aussi un ordre de priorité :

> Opérations arithmétiques : multiplication, division, résidu → addition, soustraction.

Quelques exemples : Soit x, y des réels, et n un entier ; considérez l'expression

$$2 * x + n < y$$

L'ordre de lecture est : 2*x, suivi de + n, résultat comparé à y (plus petit ?) : le résultat final = 0 ou 1. L'expression

$$2 * x + (n < y)$$

donnerait priorité au résultat de l'opération de relation, puisqu'il est entre parenthèses. Quel serait le *type* du résultat final ?

Exercice. Faites la lecture des expression suivantes :

$$n\%2 + x > y||n\%4 + x < y$$

$$n\%2 + x > (y||n\%4) + x < y$$

4.3.5 Exemple de programme C

```
/* STUSM TD no 1 : solution de l'exercice  2 */
/* Une biblioth\`eque syst\`eme */

#include <stdio.h>
main () {

/* declarations de variables et constantes */
int H = 1610, date, x, y ;

/* saisie de donnee (formattee) */
printf( "%s", " Date de la conversation ? " ) ;
scanf( "%d", &date ) ;

/* Contrainte sur les chiffres .. */
if( date < H || date > 1664 ) {
```

```
printf( "%s\n", "Mauvaise date .. " ), exit(0) ; }

/* resolution : algo .. */
x = date - H ; y = 2 * x ;

/* operateur virgule pour rabouter deux expressions */
y = ( x = date - H , 2*x ) ;

/* mise en page et sauvegarde, affichage .. */
printf( "%s%d%s%d\n", "Les ages sont = ", x, " et ", y ) ;
exit(0) ;
}
```

Compilation et exécution de ce programme

Nous n'avons ici que transcrit l'algo du pseudo-code (cf. TD1) en C. Il reste à le transcrire en langage binaire pour l'ordinateur ⇒ obtenir l'exécutable, le *logiciel*.

Le compilateur se charge de traduire en binaire votre code C :
- par defaut, il recherche des fichiers d'extension claire tel TD1.c
- le nombre de possibilités de compilation varie d'un ordinateur à l'autre : il n'y a pas de règles fixes ..
- sur une plate-forme Unix ou Linux on retrouve automatiquement un compilateur C. En effet, ces systèmes d'exploitations ont été rédigés en C.

Le système d'exploitation Linux (ici Darwin Linux) comprend un compilateur développé par le groupe GNU :

cmb@fudi++(/STUSM/Codes)(76)> g++ TD1.c

génère un code A.OUT qui est exécutable. La commande système
cmb@fudi++(/STUSM/Codes)(76)> g++ -o TD1.x TD1.c

génère un logiciel TD1.X exécutable.

4.4 Instructions de branchement

4.4.1 Branchement conditionnel

Nous avons déjà rencontré une première instruction de branchement : le "si ... alors ... sinon " du pseudo-code devient " if ... else .. " . il y en a d'autres. Leur

syntaxe est comme suit :

1. le IF :
   if (expression logique *)* { *instructions* } *else* { *instructions 2* }
   Plusieurs IF peuvent s'imbriquer l'un dans l'autre
   if (cd1) { instr 1 } else if (cd2) { instr 2 } else if (cd3) { insrt 3 }
   ...
2. le SWITCH : comme un IF, mais au cas par cas. Attention - dans ce qui suit, les quantités res1, res2 .. sont des **constantes** du langage (des chiffres, par exemple).
   switch (expression) {
   case res1 : instr1 ; break ;
   case res2 : instr2 ; break ;
   ⋮
   [defaults : instr ;]
   }
   Exemple : un bloc d'instructions avec switch

```
{ int n = 3 ; switch( 2*n+1 )
  { case 1 : printf("petit\n"); break ;
    case 3 : printf("grand\n");  break ;
    default : printf("toto"); }
}
```

3. Boucle FOR : permet de répéter un calcul tant qu'une certaine condition est satisfaite.
   for (expression 1 ; condition ; expression 2) { *instructions* }
   Dans cette syntaxe on reconnait
   1) l'initialisation de variables (compteurs) ;
   2) une condition de relation ;
   3) une mise à jour des compteurs et variables.
   Exemple :

```
 for (i=1, fact=1 ; i< =n ; fact =fact*i, i=i+1)
{...}
```

   Remarque : le bloc d'instructions est exécuté avant la mise à jour des compteurs (ici : l'entier i).
4. boucle WHILE : comme le FOR mais accepte une seule expression en argument :
   WHILE (expression) { ... }

Le bloc d'instruction qui suit le mot *while* est exécuté tant et aussi longtemps que l'expression entre parenthèses est évaluée à *vrai*, c'est-à-dire une expression mathématique retournant un résultat non nul. Ici au moins une variable qui fait partie de l'expression du while doit être modifiée par le bloc d'instructions, sinon : la condition est toujours satisfaite, et la boucle est infinie.

5. boucle DO { instructions } WHILE (condition) ;
   Comme le while, mais inverse l'ordre d'exécution des instructions et l'évaluation de l'expression entre parenthèses .. **Attention** : le point-virgule qui suit le mot 'while' est essentiel.

Exemple d'utlisation de la boucle do ... while : dans notre programme on aurait voulu reprendre la saisie de données si les dates sont fausses .. on pourrait faire cela en écrivant aux endroits appropriés du code

```
/* saisie de donnees (formattees) */
do {
  printf( "%s", " Date de la conversation ? " ) ;
  scanf( "%d", &date ) ;
}
while( date < H || date > 1664 ) ;
```

Plutôt que ce que nous avions auparavant :

```
/* Contrainte sur les chiffres .. */
  if( date < H || date > 1664 ) {
    printf( "%s\n", "Mauvaise date .. " ), exit(0) ; }
```

4.4.2 Branchement non-conditionnel

Trois mots-clefs du C permettent d'interrompre un bloc d'instructions pour rediriger le programme.

1) break : interrompt le bloc d'instructions, repart à la suite du bloc courant (voir surtout le switch).

2) continue : exactement cela - instruit le programme de poursuivre l'exécution au point où apparait le mot 'continue'.

3) goto : permet de renvoyer n'importe où dans le programme, ce qui requiert un libellé ; celui-ci peut-être un chiffre ou un identifiant, dans tous les cas suivi d'un double point (:) . Par exemple on pourrait utiliser le mot-clef goto pour récrire une boucle FOR :

```
     ⋮
10 : i++;
   if( i < 5 ) goto 10;
     ⋮
```

4.5 Les affectations composées

En général le C permet d'exprimer de manière compacte certaines expressions répétitives. Ainsi on peut récrire

$$\text{pour : } a = a + 1 \longrightarrow a+ = 1$$
$$\text{pour : } a = a - 1 \longrightarrow a- = 1$$

et ainsi de suite : $* =$, $/ =$ ainsi que $\% =$ sont aussi possibles. Ce sont des affectations *composées* , car il s'agit d'affecter une variable d'une valeur nouvelle en combinant l'opérateur d'affectation '=' avec une opération arithmétique.

Les boucles FOR, WHILE, sont mieux écrites à l'aide des expressions d'incrémentation et décrémentation : ++ et - -, respectivement.

Exemples :

b = a++ est équivalent aux instructions b = a ; a = a + 1 ;

b = ++a est la même chose que a = a + 1 ; b = a ;

On peut ainsi reprendre la syntaxe d'un FOR pour y insérer des '++' ou '- -' au besoin.

Par exemple : TD 1 no 3, nous voulions élever un nombre à une puissance entière n.

```
/* Version simple, a**n boucle */
int main( ) {
int a, n = 5, y, m = 1 ;
if( n == 0 ) return 1 ;
y  = a ;
for( m = 2 ; m <= n ; m++ )
    { y *= a ;}
    exit(0) ;
}
```

Reconnaissez-vous votre algorithme ? Qu'y a-t-il de différent ici ?

4.6 Les fonctions en C

4.6.1 Définition intuitive :

Une fonction est comme en mathématiques une opérations sur des variables nous procurant un résultat :

$$y = f(x).$$

Entre autres, la notation (en C) et leur utilisation sont très similaires :
- le nom de la fonction (f) respecte les contraintes sur les identifiants du C.
- une fonction accepte un ou plusieurs arguments (tel x ici).
- elle renvoie une valeur (nécessaire si à la droite de l'affectation =)
- le mot-clef **return** permet de terminer une fonction et renvoyer des valeurs
- se compose d'un bloc d'instructions, donc de variables locales (par défaut).

L'utilisation d'une fonction se fait par l'appel de la fonction incluant tous ses arguments.

La déclaration de fonctions : respecter la syntaxe

[type] *identifiant* (<type1> arg1, <type2> arg2, ...) { bloc d'instructions définissant toutes les opérations de la fonction ; return [expression] ; }

Quelques remarques :
- si le type de la fonction n'est pas spécifié, alors il est égal à int (entière) par défaut.
- si le type choisi est void (vide, nul) alors la fonction ne peut pas retourner le résultat des instructions.
- le mot-clef return est essentiel pour terminer une fonction ; exception : la fonction principale *main*.

En somme, le mot-clef **return** permet de terminer le bloc d'instructions qu'est la fonction : il remplit le même rôle que le mot-clef **break** rencontré plus tôt (se référer au *switch* notamment).

4.6.2 Déclaration de fonctions : le patron

Dans un programme : les fonctions sont indépendantes l'une de l'autre.

⇒ pour le C : l'identifiant (nom) d'une fonction est un nouveau mot, il doit être défini (déclaré).

⇒ une déclaration doit donc précéder l'appel (utilisation) à la fonction. Problème : si la fonction est longue, ou si nous avons plusieurs fonctions,
- perte de clarté
- la fonction *main* encapsule l'algorithme, pas les autres

On peut simplifier en introduisant en début de programme un **patron** pour chaque fonction déclarée.

Un patron de fonction s'écrit comme ceci :

<type > *identifiant* (<type1>, <type2>, ..) ;

Le programme prend alors l'allure suivante

⋮

patron de fonction 1 ;
patron de fonction 2 ;

⋮

fonction *main* () { }
déclaration de fonction 1
déclaration de fonction 2

Exemple : récrire un bloc d'instructions en fonction (cf. fichier fonction.c).

Bloc d'instructions $\longrightarrow y = x^n$. Note : x, n déjà saisis.

⋮

```
/* Declare, initialise i, y ; calcule y = x**n  */
{ int i ; float y ;
  y = 1. ;
  for( i=0; i < n ; i = i + 1 ) { y = y*x ; }

  /* Ecrire resultat a l'ecran */
  printf( "%s %f\n", "Resultat y = ", y ) ;
}
```

⋮

On peut récrire le bloc d'instructions facilement pour en faire une fonction. La déclaration de cette fonction pourrait être par exemple

```
int puiss ( float x, int n ){

/* Declare, initialise i, y ; calcule y = x**n  */
int i ; float y ;
y = 1. ;
for( i=0; i < n ; i =i+1 ) { y = y*x ; }

/* Ecrire resultat a l'ecran */
printf( "%s %f\n", "Resultat y = ", y ) ;
 return ;  }
```

Dans le programme principal l'appel de la fonction **puiss** se fait simplement par (voir fichier fonction1.c)

$$puiss(x, n);$$

Pour retourner le résultat, le stocker dans une variable qui appartient au programme principal *main* : il suffit de passer le résultat en argument de **return**.

Exemple :

$$\vdots$$

```
for( i=0; i < n ; i =i+1 ) { y = y*x ; }

/* Retourner le r\'esultat au main */
return y ;  }
```

L'appel de la fonction devient alors

$$y = puiss(x, n);$$

4.7 Structure d'un programme

On sait déjà où placer bibliothèques et fonctions dans un programme pour pouvoir compiler et exécuter. Il reste à voir plus formellement les éléments structurateurs d'un programme

1. Séparateurs

   Ce sont par exemples des espaces vides, des tabulations, ou des caractères de contrôle tel \n.

   Ils sont ignorés par le compilateur - sauf dans une **chaîne de caractères**.

2. Commentaires

   Nous les avons rencontrés : ce sont des messages encadrés des symboles /* ... */

   Ils sont ignorés par le compilateur

3. Directives au pré-processeur

   – Effectivement des commandes passées au compilateur C.

   – Toujours précédées du symbole # dans le fichier source.

   – Jamais ignorées du compilateur : servent à redéfinir des paramètres, voire les parties exécutables d'un programme.

Quelques exemples de directives suivent

4.7.1 Liste de directives au compilateur C

Pourquoi des directives au compilateur ? Elles permettent de simplifier l'édition du fichier contenant le code source (en langage C). Elles permettent une plus grande portabilité du programme, par exemple pour le faire tourner sur des ordinateurs différents.

1. #include

   Cette directives permet d'inclure des fichiers dans notre code sans avoir à les copier. La syntaxe en est
   - #include <nom_fichier> pour des **bibliothèques** du système
   - #include "/chemin_complet/nom_fichier" pour un fichier situé dans un répertoire inhabituel, par exemple ses propres répertoires. Exemples :

     `#include <stdlib.h >` inclue la bibliothèque standard du C
     `#include "fichier.h"` inclue fichier.h recherché dans le répertoire actuel.

2. #define

   Pour définir symboles ou macros. Par exemple

   ```
   #define COULEUR BLEU
   ```

   indique au compilateur de remplacer dans le programme le symbole COULEUR par BLEU. On peut ainsi remplacer un symbole par la valeur désirée sans avoir à parcourir tout le programme : il suffira de mettre une directive #define appropriée au haut du fichier pour affecter l'ensemble du fichier.

   On peut aussi définir des **macros-fonctions**. Par exemple

   ```
   #define MAX(a,b)  ( a > b ? a : b )
   ```

   remplace toutes les occurences de MAX(x,y) par l'expression C de comparaison ternaire (x > y ? x : y) ; même chose pour MAX(i,j), et ainsi de suite.

3. Compilation conditionnelle

   Il peut arriver que nous désirions inclure certaines parties d'un code C pour un calcul particulier, sans pour autant en modifier l'algorithme. Par exemple, pour valider un programme. La compilation conditionnelle permet d'effectuer une compilation sur mesure à l'aide de l'option de compilation $-D$. (D pour 'définir.) L'instruction IF du C est aussi présente à la compilation mais de notation différente :

   #if condition

   instruction C arbitraire, voire bloc d'instructions

   #endif

Remarque : la structure IF .. ELSE IF .. du C devient ici #IF ... #ELIF ... #ENDIF . Les conditions imposées au #IF utilisent les mêmes opérateurs que le C.

Exemple :

```
#if PROCESSEUR == PC
static int long_int = 32 ;
#elif PROCESSEUR == ALPHA
static int long_int = 64 ;
#endif
```

Ces directives permettraient de compiler le programme en tenant compte du type d'ordinateur utilisé et des définitions d'entiers sur cet ordinateur. Un symbole tel PROCESSEUR est déclaré lors de la compilation par une commande telle que

```
gcc -DPROCESSEUR=PC  fichier_source.c
```

ici *gcc* invoque le compilateur C. (Note - cet exemple présume que les variables PC et ALPHA ont déjà été définies, à l'aide de #define.)

Ainsi dans l'exemple ci-haut le compilateur inclurait l'instruction

static int long_int = 32 ;

pour la génèse d'un exécutable a.out.

On peut également vérifier l'existence de symboles sans les initialiser. Pour cela on utilise la directive #ifdef ainsi :

#ifdef SYMBOLE *instruction C arbitraire, voire bloc d'instructions*

#endif

Un SYMBOLE est alors défini au moment de la compilation par l'option -D :

```
gcc -DSYMBOLE fichier$\_$source.c
```

Remarque : On peut définir plusieurs symboles en invoquant plusieurs fois l'option -D du compilateur. Il existe également une directive #ifndef (if not define : si non defini) qui vérifie si un symbole n'existe pas. Elle accepte la même syntaxe que #ifdef.