Chapitre 6

Première approche du langage C

Au terme de cette première partie, nous allons nous soucier d'utiliser un langage répandu et utilisé par des millions de programmeurs.

6.1 Petite histoire du langage C

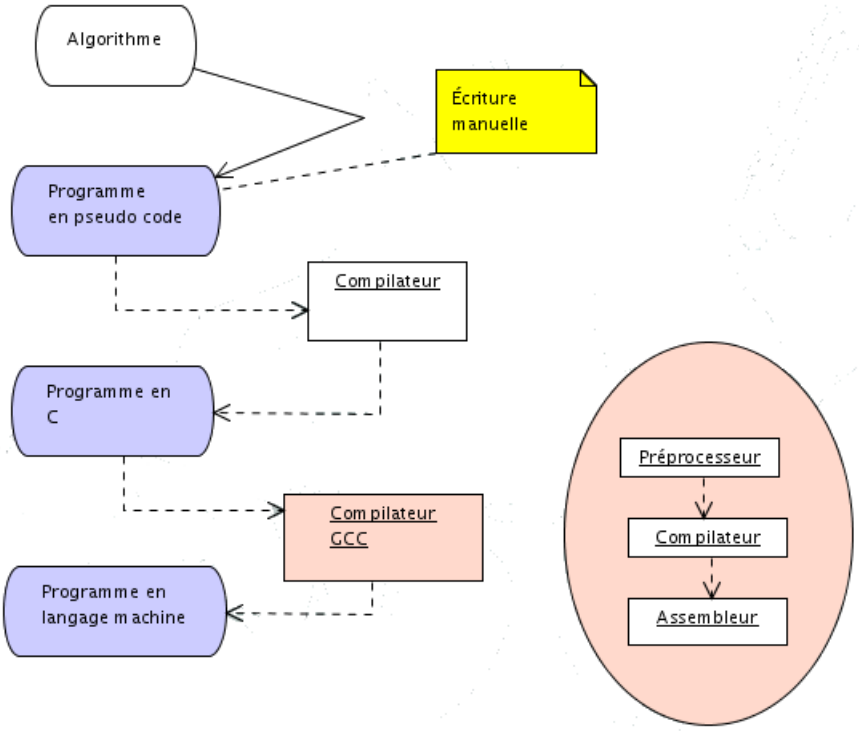
Le langage C a été inventé en 1972 par Denis RITCHIE au cours des premiers développements de ce qui allait devenir le système d'exploitation UNIX. La philosophie de ce langage était de développer des petits programmes, proches de la machine et capables de s'enchaîner facilement pour réaliser des tâches complexes. Aussi rapide et puissant que le langage machine, mais structuré (au sens d'ALGOL), ce langage a été très rapidement adopté par un grand nombre de programmeurs. Il a fait l'objet d'une normalisation par l'ANSI, puis par l'ISO dans le courant des années 90. Il est encore très employé, notamment par toute la communauté Open Source.

Le C a donné lieu à cinq nouveaux langages qui reprennent largement sa syntaxe [1] :

- le **C++** reste pratiquement compatible avec tous les programmes écrits en C (il durcit quelques règles un peu laxistes) et ajoute une dimension objet au langage.
- **Objective C**, presque inconnu dans le monde des PC, sert souvent de langage de développement sur les ordinateurs Macintosh et dans l'environnement Next. Il offre également, comme son nom l'indique, une extension objet.
- **Java** est un langage objet non compatible avec le C, mais dont la syntaxe est néanmoins fort proche. Il est conçu pour tourner sur une machine virtuelle, dont on retrouve des implémentations sur tous les systèmes d'exploitation.
- **PHP** est un langage de script libre qui reprend la syntaxe générale du C, mais en la simplifiant énormément. Il ajoute un traitement des tableaux beaucoup plus souple et des bibliothèques de gestions de modules externes (surtout des bases de données) qui en ont fait un langage idéal pour la réalisation de sites Web interactifs. La version 4, et plus encore la 5, offrent des extensions objet tout-à-fait intéressantes.
- **C#** (prononcé C sharp) constitue à la fois une réaction de Microsoft contre le C++, jugé hybride et lourd, et Java, défendu par un adversaire commercial. Comme Java, il ne doit pas supporter une compatibilité avec C et se révèle par là plus cohérent et plus fidèle à la philosophie objet. Il constitue le fer de lance de la stratégie *.net*. Notons que le projet libre Mono et d'autres projets permettent d'utiliser C# ailleurs que dans un environnement Windows.

1. Le C a d'autres descendants que je ne mentionne pas ici : les langages de script d'Unix ou AWK.

6.2 Les étapes de la compilation d'un programme



Selon le modèle créé aux premiers jours de l'informatique par John VON NEUMANN, les ordinateurs actuels disposent d'une seule zone mémoire pour y stocker les instructions et les données. Cela permet d'automatiser une série de processus de traduction : les instructions d'un niveau *n+1* sont traitées comme données par le niveau *n*. Entre le moment où l'algorithme germe dans le cerveau du programmeur et le moment où les instructions s'exécutent, on trouve une série d'expressions de l'algorithme avec un passage plus ou moins facile au niveau suivant :

1. l'algorithme est présent dans notre tête : nous savons comment nous allons réaliser la tâche que nous sommes chargés de programmer.
2. nous exprimons l'algorithme dans un langage de description d'algorithme (organigramme, pseudo-code, code LAF). Cette traduction mentale a pour effet de préciser certains détails qui n'étaient pas clairs dans l'algorithme initial (le nom des variables, certains traitements habituels comme l'incrémentation d'un compteur...).
3. l'algorithme écrit en pseudo-code est traduit dans un langage assimilable par la machine, ici le langage C. Si nous utilisons LAF, ce processus peut être entièrement automatique.
4. le programme en langage C est pris en charge par un compilateur standard. Il le traduit en langage machine directement exécutable sur le processeur. En réalité, le programme langage C est traduit en plusieurs phases :
   (a) remplacement de certaines macros en instructions plus basiques par le **préprocesseur**
   (b) traduction du code obtenu en langage d'assemblage (assembleur symbolique), c'est la **compilation** proprement dite
   (c) traduction du code d'assemblage en code machine (à l'aide d'un programme assembleur)
   (d) liaison du code obtenu avec le code des bibliothèques choisies par le programmeur (non illustré sur le schéma) : c'est l'**édition des liens**.

Dans le contexte de notre laboratoire, la compilation du code C en code machine se fait à l'aide d'une seule instruction :

```
gcc -Wall source.c -o destination -lm
```

Voici la signification des différents composants de cette ligne :

		explication
`gcc`	commande	appel du compilateur
`-Wall`	option	demande de compilation avec affichage de tous les avertissements
`source.c`	paramètre	nom du fichier source (l'extension `.c` est obligatoire)
`-o`	option	on précise le nom du programme final (faute de quoi, il s'appelera *a.out*).
destination	paramètre d'option	le nom du programme final
-lm	option	on utilisera la bibliothèque libm.a, qui contient les extensions mathématiques du langage.

Sous Windows, cette commande s'écrira un peu différemment :

```
gcc -Wall source.c -o destination.exe
```

Il faut préciser ici une extension `.exe` pour l'exécutable et il n'est pas nécessaire d'appeler la bibliothèque mathématique. Notons qu'on peut compiler avec d'autres compilateurs (par exemple, le TurboC de Borland, commande `tc`). Dans ce cas, on consultera la documentation du programme pour choisir les options.

6.3 Structure générale d'un programme

Un programme C simple se compose généralement de quatre parties :
- des déclarations d'en-têtes (avec des directives `#include`)
- des déclarations de types, de variables, de constantes et de fonction (prototypes) : elles seront détaillées dans la section suivante.
- le corps principal du programme, qui correspond à ce que nous avons appelé le traitement dans notre pseudo-code.
- des définitions de fonctions : ce dernier point sera vu dans la deuxième partie du cours.

6.3.1 directives `#include`

La compilation d'un programme C est extrêmement complexe et ne peut faire l'objet d'une description détaillée dans ce cours (voir plus tard le cours de *Langage procédural*). Le langage C fait un usage intense de bibliothèques, même pour des tâches élémentaires comme la saisie ou l'affichage d'une donnée. Un programme C commencera donc toujours par faire référence aux bibliothèques qu'il emploie. Ces références, qui varient d'un environnement à l'autre, sont contenues dans des fichiers d'en-têtes (*header*). Il est prudent de faire référence au fichier `stdio.h` (*standard input-output* : bibliothèque d'entrées/sorties standard, qui contient notamment les définitions des routines `scanf()` et `printf()`) et au fichier `math.h`, qui contient la plupart des fonctions mathématiques dont nous aurons besoin. La référence à `laf2c.h` s'impose dans le contexte de notre pseudo-code et ne doit donc pas figurer dans un programme natif[2].

2. Ce fichier contient la définition de deux macros et d'une fonction. Nous verrons que la macro `ViderTampon` est bien utile pour faciliter les lectures au clavier.

```
#include <stdio.h>
#include <math.h>
#include <laf2c.h>
```

Dans une programmation avancée, les programmeurs sont parfois amenés à définir leurs propres bibliothèques. Ils peuvent ainsi éprouver le besoin de créer eux-mêmes leur fichier d'en-tête. Dans ce cas, on le place dans le répertoire courant, ce qui nécessite une syntaxe légèrement différente.

```
#include "mabiblio.h"
```

6.3.2 déclarations

On peut déclarer des variables (en ce compris des tableaux), des constantes, et des structures (voir section suivante). Les prototypes, qui correspondent à des déclarations de fonction seront vues dans la deuxième partie du cours.

6.3.3 corps du programme

Le corps d'un programme C est une fonction `main()`, unique et obligatoire dans tout programme écrit en C. Il existe plusieurs manières de l'écrire. La première version courte risque de déclencher des messages d'avertissement avec un compilateur un peu soigneux :

```
main()
  {
    /* suite d'instructions */
  }
```

Cette version plus longue précise le type de `main()` et renvoie une valeur 0 au programme appelant (cette valeur doit s'interpréter comme « programme terminé sans erreur »).

```
int main()
  {
    /* suite d'instructions */
    return 0;
  }
```

Chaque instruction du langage C est suivie d'un point-virgule.

6.4 Déclarations diverses

6.4.1 variables

Nous avons vu cinq types de variables dans notre pseudo-code.

```
Variable Entière iNbreProduits
Variable Logique lCorrect
Variable Flottante fPrixUnitaire
Variable Caractère cReponse
Variable Chaîne sTexte
```

Ils correspondent aux types `int`, `double`[3], `char` du C. Les variables logiques sont remplacées par des entiers (1 vaut `VRAI` et 0 vaut `FAUX`). À noter, le traitement particulier des chaînes en C, qui sont en fait des tableaux de chaînes de caractères, dont on doit spécifier la taille. Le type précède toujours le nom de la variable. Chaque déclaration est suivie d'un '`;`'.

```
int iNbreProduits;
int lCorrect;
double fPrixUnitaire;
char cReponse;
char sTexte[256];
```

La taille par défaut des chaînes dans le pseudo-code, arbitrairement fixée à 256, ne gère pas la mémoire de manière parcimonieuse. Il est possible d'employer une version de la déclaration qui spécifie une taille explicite. Celle-ci est immédiate à traduire en C.

```
Variable Chaîne # 20 sNom
```

correspond à

```
char sNom[20];
```

Le langage C autorise également plusieurs déclarations sur une seule ligne :

```
int iSomme, iDifference, iProduit;
```

6.4.2 tableaux

La déclaration d'un tableau en C se fait en ajoutant directement une taille entre crochet derrière le nom d'une variable.

```
int iNotes[10];
double fPrixLivres[100],fPrixEuros[100];
```

L'accès aux éléments d'un tableau se réalise de la même façon que dans notre pseudo code. Nous n'en parlerons pas plus avant.

Pour initialiser un tableau lors de sa déclaration, il suffit d'ajouter l'énumération des valeurs des différents éléments séparés par des virgules, le tout placé entre accolades.

```
float fTFactures[5]={65.2,63.78,102.5,25.4,31.7};
```

Il est possible dans ce cas de ne pas placer de nombre entre les crochets de déclaration du tableau, le compilateur se chargeant lui-même de ce comptage.

```
char cTOptions[]={'a','c','e','r'};
```

Le C ne connaît pas les tableaux à plusieurs dimensions. Il faut alors déclarer des tableaux de tableaux, ce qui se réalise en plaçant une nouvelle paire de crochet entre l'identificateur et les crochets de la dimension du tableau constituant. Voici une déclaration d'un tableau reprenant les jours d'une année en pseudo-code et sa traduction en C.

3. Un type `float` existe également, mais ne propose qu'un maximum de 7 chiffres significatifs exacts. De même il existe d'autres types d'entiers : `short int`, `long int`, `unsigned int`. Ces différents types seront vus en détails dans le cours de langage procédural.

```
/* pseudo code */
Tableau Entiers iTAnnee taille 12:31
/* langage C */
int iTAnnee[12][31];
```

Pour accéder à une cellule d'un tableau de tableaux, on doit spécifier deux indices. `iTAnnee[0]` désigne un tableau de 31 cases représentant le premier mois. Pour accéder au troisième jour, on spécifie son indice : `iTAnnee[0][2]`.

Les chaînes de caractères étant elles-mêmes des tableaux, un tableau de chaîne de catactères sera donc en C un tableau à deux dimensions.

```
Tableau Chaines sTListeEleves taille 50
```

se traduira par

```
char sTListeEleves[50][256];
```

On pourra spécifier une taille plus réduite à la place de 256.

Il est également possible d'initialiser un tableau de tableaux :

```
int iTab [3][2]={{1,2},{3,4},{5,6}};
```

6.4.3 structures

Les structures sont assez lourdes à manipuler en C. La définition minimale d'une structure consiste à employer le mot réservé `struct` et à définir les champs entre une paire d'accolades. Dans ce cas, on utilise l'expression complexe formée `struct` et le nom de la structure pour définir les variables structurées.

```
struct ePersonne
{ char sNom[20];
  int iAnneeNaissance;
};
struct ePersonne uClient;
```

Je préfère l'emploi de `typedef` pour définir un type, qui a pour mérite d'alléger la définition des variables [4].

```
typedef struct
{ char sNom[20];
  int iAnneeNaissance;
} ePersonne;
ePersonne uClient;
```

Le mot `struct` peut être suivi d'un nom de structure ou rester anonyme (`typedef struct __ePersonne`).

Ici encore, on peut proposer une initialisation :

```
ePersonne uDupont = {"Dupont",1955};
```

Nous ne parlerons plus des structures dans la suite, puisqu'elles s'emploient en C de la même manière que dans notre pseudo-code.

4. Cette syntaxe, encore allégée, a été généralisée par C++.

6.4.4 constantes

Le langage C ne comporte pas vraiment de constantes au sens strict du terme. Une définition de variable peut être précédée d'un modificateur `const` qui a pour effet d'interdire toute modification directe de cette variable. Cela ne garantit toutefois pas l'interdiction d'une modification indirecte, par exemple au moyen d'un pointeur. En outre, `const` ne peut pas s'employer avec des chaînes de caractères. C'est pourquoi, j'ai choisi de traduire nos constantes par des directives `#define` qui sont en réalité des sortes de remplacements, style traitement de texte, par le préprocesseur. Ce type de pseudo-constante admet tous les types (en fait n'importe quoi après le nom de la constante), mais a une portée étendue à tout le fichier, ce qui nous interdit de définir des constantes locales [5].

Les déclarations suivantes du pseudo code

```
Constante NOM = "École␣de␣Commerce␣et␣d'Informatique"
Constante NBREETUDIANTS = 100
Constante TVA = 21.5
Constante OUI = 'y'
```

se traduiront par

```
#define NOM "Ecole␣de␣Commerce␣et␣d'Informatique"
#define NBREETUDIANTS 100
#define TVA 21.5
#define OUI 'y'
```

La pseudo-constante s'identifie en fait à tout ce qui suit l'espace après le nom de la constante, jusqu'à la fin de la ligne. On notera l'absence de point-virgule à la fin de la ligne. Si on en place un, il sera compris dans la constante et provoquera souvent une erreur lors son utilisation.

6.5 Commentaires

Le langage C utilise la même convention que le pseudo-code pour ajouter des notes à l'attention des lecteurs humains. Ils se délimitent à l'aide de `/*` et de `*/`.

```
/* tout ceci sera superbement ignoré par le compilateur */
```

Le langage C++ a introduit une autre convention pour les commentaires, et celle-ci est souvent acceptée par les compilateurs C. Il s'agit d'une double barre (`//`) qui permet d'ignorer la fin de la ligne [6]. Cette manière de marquer une zone de commentaire est utile dans les déclarations de variables.

```
int iNbreMystere; // le nombre à deviner par le joueur
int iProposition; // le nombre proposé par le joueur
```

C'est aussi un moyen efficace pour supprimer temporairement une instruction. Dans cet exemple, l'impression de la valeur de `iSomme` n'aura pas lieu.

```
iSomme=iSomme+iNombre;
// printf("La somme vaut %d\n",iSomme);
iEtape++;
```

5. Sur la notion de portée, voir la suite de ce cours.

6. Leur emploi est cependant impossible dans une directive `#define` puisque celle-ci utilise la fin de la ligne pour créer la constante.

6.6 Affectations

L'affectation d'une valeur ou de l'évaluation d'une expression à une variable se fait au moyen de l'opérateur '='. Il renvoie comme valeur la valeur de l'expression, qui peut être utilisée pour initialiser une autre variable.

```
iMinimum = 5;
fTaux = 21.5;
cOption = 'a';
iSomme = iI = 0;
```

Malheureusement les affectations dans les chaînes de caractères ne sont pas possibles par le même moyen. Il faut faire usage de la fonction `strcpy()`. Celle-ci ne vérifie pas que la chaîne de destination contient suffisamment de place pour recevoir le texte, ce qui peut amener des plantages de programmes. Ainsi pour traduire

```
sTest <- "Bonjour"
```

nous devrons écrire

```
strcpy(sTest, "Bonjour");
```

Le langage C possède d'autres opérateurs d'affectation . Par exemple, += et −= permettent d'ajouter ou des soustraire une valeur d'une variable données. ++ et −− incrémentent et décrémentent une variable [7]

```
/* Lister dix nombres pairs */
iNombre=0;
for(i=1;i<=10;i++)
{
  iNombre+=2;
  printf("%d\t",iNombre);
}
```

6.7 Les entrées-sorties

Le langage C n'est pas réputé pour posséder des instructions d'entrée/sortie simples, ni très sûres. Je me réfère à trois problèmes fréquents que nous verrons plus bas. Nous nous contenteront de montrer les cas standards, sans entrer dans les détails du format (qui constituent à eux seuls un chapitre entier du cours de langage procédural).

6.7.1 Sorties

Les sorties utilisent principalement la fonction `printf()`. Voici quatre exemples de sorties, respectivement avec des types entiers, flottants, caractères et chaînes :

```
printf("%d",iNbreEtudiants);
printf("%f",fPrixTVAC);
printf("%c",cOption);
printf("%s",sPrenom);
```

7. Je ne fais pas état des deux formes de ces opérateurs, par exemple dans les expressions `iTab[++i]` et `iTab[i++]`. Ces points seront revus lors de l'apprentissage du C et constituent des notations ingénieuses pas toujours compatibles avec une programmation limpide. Nos instructions `Incrémenter` et `Décrémenter` s'accommode des deux syntaxes de ces opérateurs.

Les formats, ici limités au caractère `%` suivi d'un symbole correspondant au type, peuvent contenir des directives très riches. Le langage C permet ainsi de réaliser des affichages complexes [8] :

```
printf("Les␣%d␣étudiants␣ont␣payé␣%.2f␣Euros
à␣leur␣ami%s.\n", iNbreEtudiants,fPrixTVAC,sPrenom);
```

affichera par exemple :

```
Les 6 étudiants ont payé 4.78 Euros à leur ami Michel.
```

L'affichage d'une chaîne constante peut se faire soit en plaçant la chaîne dans le format, soit avec un format et une chaîne constante comme paramètre (solution retenue par *laf2c*) :

```
// Afficher "Bonjour le monde!"
// Première version
printf("Bonjour␣le␣monde:");
// Deuxième version
printf("%s","Bonjour␣le␣monde!");
```

6.7.2 Entrées

Les entrées se réalisent souvent avec la fonction `scanf()`, qui n'attend généralement que deux arguments. Le premier est un format semblable à celui des sorties, mais plus simple, le second l'adresse d'une variable. Le fait qu'il s'agisse d'une adresse implique l'usage de l'opérateur `&`.

```
scanf("%d",&iNbreEtudiants);
scanf("%lf",&fPrixTVAC);
scanf("%c",&cOption);
scanf("%s",sPrenom);
```

Remarques :

1. Le format de lecture `%lf` d'un nombre *double* est différent du format d'affichage `%f`. Le C adopte une politique curieuse pour les formats flottants. Lors de l'affichage, un *float* est automatiquement converti en *double*. On n'utilise que le seul format `%f`. Pour la lecture, on utilise respectivement `%l` pour les variables *float* et `%lf` pour les variables *double*. Toute confusion à ce niveau entraîne des comportements erronés du programme.
2. L'absence de l'opérateur & devant le nom de la variable tableau de caractères se justifie par le fait qu'en C, une telle variable est déjà une adresse, ou plus spécifiquement, un pointeur constant.

6.7.3 Problèmes classique avec les entrées

1. Les débutants, et les autres, oublient fréquemment de placer le signe `&` devant le nom de la variable. Cela ne produit généralement pas d'erreur d'exécution, mais les données ne sont pas placées dans la variable, ce qui est généralement difficile à débusquer.

8. Le symbole ▽ marque ici une coupure de ligne due à des raisons typographiques.

2. Une lecture de nombres pose généralement des problèmes lorsqu'un utilisateur ajoute des caractères alphabétiques dans sa réponse. Ces caractères provoquent des comportements erratiques des programmes. Il n'est pas facile de contrôler cela par programme. La solution ultime consiste à tout lire en chaînes de caractères et à opérer soi-même les conversions.
3. Le retour de chariot tapé par l'utilisateur reste dans la tampon d'entrée. Cela peut avoir des conséquences dramatiques lors d'une lecture de caractère après une lecture de nombre. Le caractère sera toujours un retour de chariot. C'est pourquoi *laf2c* place toujours la macro `ViderTampon` à la suite de chaque lecture avec `scanf()`. L'effet de cette macro est de consommer tous les caractères tapés par l'utilisateur, jusque et y compris le retour de chariot.

```
#define ViderTampon while(getchar()!='\n');
```

4. La lecture d'une chaîne de caractères présente un défaut de sécurité bien gênant. Bien des pirates essaient de détecter des programmes utilisant `scanf` avec une adresse de chaîne. Avec un peu de savoir-faire, un utilisateur mal intentionné peut saturer la zone tampon, provoquer un débordement de pile et prendre le contrôle de la machine. La commande

```
Lire sTest
```

est traduite par *laf2c* [9] au moyen de

```
fgets(sTest,sizeof sTest,stdin);SupprimerCR(sTest)
```

6.8 Alternatives

6.8.1 Cas général

Le langage C utilise le mot réservé `if` pour traduire l'examen d'une condition introduite par `Si`. Il traduit `Sinon` par `else`. Par contre, il n'a aucun mot pour traduire `Alors` et `FinSi`. Pour le premier des deux mots, cela ne pose pas de problème parce C exige que la condition soit mise entre parenthèses, le début du premier traitement est donc clairement indiqué. Il serait par contre impossible de distinguer la dernière instruction du traitement alternatif et l'instruction qui suit le passage conditionnel.

```
Si fNombre2 Egal A 0 Alors
   Message "Une␣division␣par␣0␣est␣impossible."
Sinon
   fQuotient <- fNombre1 / fNombre2
   Message "Le␣quotient␣vaut␣:␣"
   Afficher Flottant fQuotient
Fin de Si
```

9. La traduction exacte dépend en fait du contexte. J'ai essayé de proposer une solution raisonnable. Les chaînes passées en paramètres de routine sont supposées avoir une taille de 256 caractères. Il est possible d'opérer une lecture en précisant la taille du tampon (`Lire #15 sTest`). Il est évident que tous ces problèmes se retrouvent en C. Il est impossible de connaître la taille d'un tableau passé en argument d'une fonction. La meilleure solution consisterait à ne jamais effectuer de lecture dans une telle variable. Il reste que l'utilisation d'une variable intermédiaire ne règle pas le problème du débordement lors de la recopie dans la chaîne passée en paramètre, à moins de passer la taille de la chaîne comme autre paramètre afin de pouvoir contrôler finement cette recopie. S'il existait une solution simple, elle serait connue depuis longtemps.

Comment dès lors marquer clairement les limites des traitements alternatifs ?

```
/* Programme simplifié */
if(fNombre2==0)
   printf("Une division par 0 est impossible.");
else
   printf("La division par %f est possible",fNombre2);
/* suite du programme */
```

La solution imaginée par les concepteurs du C est astucieuse :

- une instruction conditionnelle ne comporte que deux instructions internes : l'une exécutée si la condition est vraie, l'autre si elle ne l'est pas. La version simplifié de notre division illustre cette pratique.
- si l'un des traitements alternatifs (ou les deux) doivent s'exprimer au moyen de plusieurs instructions, celles-ci sont transformées en une grosse instruction, nommé bloc. Le bloc se marque à l'aide d'une paire d'accolades. Nous devrons donc entourer d'accolades les instructions à effectuer lorsque le quotient n'est pas nul. Il est possible, voire même prudent, de créer un bloc, même lorsqu'il n'y a qu'une seule instruction. Nous sommes à présent en mesure de traduire notre calcul de quotient :

```
if(fNombre2==0)
  {
    printf("Une division par 0 est impossible.");
  }
else
  {
    fQuotient = fNombre1 / fNombre2;
    printf("Le quotient vaut :");
    printf("%f",fQuotient);
  }
```

Pour exprimer la condition, la langage C a recours à six comparateurs principaux.

Opérateur	Exemple	Signification	Opérateur	Exemple	Signification
==	`fDelta == 0`	Egal	!=	`fDelta != 0`	Différent
<	`fDelta < 0`	Inférieur	>	`fDelta > 0`	Supérieur
<=	`fDelta <= 0`	Inférieur ou égal	>=	`fDelta >= 0`	Supérieur ou égal

Il convient de ne pas confondre le comparateur `'=='` avec le signe d'affectation d'une valeur à une variable '='. Une telle confusion est dramatique. Le programme suivant, censé afficher un message si `iNbre` vaut 20, affichera toujours le message et écrasera la valeur lue au clavier :

```
scanf("%i",&iNbre);
if(iNbre=20) // Erreur : = au lieu de ==
   printf("Vous avez demandé 20");
```

En effet, l'expression entre parenthèse affecte 20 à la variable et renvoie la valeur 20. Cette valeur, non nulle, est considérée comme vrai et le test réussi. La suite du programme ignorera totalement la valeur entrée par l'utilisateur. Voici ce qu'il fallait écrire :

```
scanf("%i",&iNbre);
if(iNbre==20)
```

```
   printf("Vous avez demandé 20");
```

La nature spéciale des chaînes de caractères fait qu'elles ne permettent pas d'effectuer des comparaisons à l'aide des opérateurs standard. Voici à quoi correspondent quatre comparaisons de deux chaînes en C :

`sTest == "Bonjour"`	`strcmp(sTest,"Bonjour")==0`
`sTest <> "Bonjour"`	`strcmp(sTest,"Bonjour")!=0`
`sTest > "Bonjour"`	`strcmp(sTest,"Bonjour")>0`
`sTest < "Bonjour"`	`strcmp(sTest,"Bonjour")<0)`

6.8.2 Structures à choix multiples

Rappelons un exemple de notre pseudo code :

```
Examen si iX
   vaut 1,2,3
      Message "Le nombre est compris entre 1 et 3"
   fin de cas
   vaut 4
      Message "Le nombre vaut quatre"
   fin de cas
   Défaut
      Message "Le nombre n'est pas compris entre 1 et 4"
   fin de cas
Fin de examen
```

C utilise les mots réservés `switch`, `case`, `break` et l'étiquette prédéfinie `default` pour représenter cette structure. L'exemple suivant affichera un message décrivant une propriété du nombre mémorisé dans `iX`.

```
switch(iX)
   {
     case 1 :
     case 2 :
     case 3 :
       printf("Le nombre est compris entre 1 et 3");
       break;
      case 4 :
       printf("Le nombre vaut 4");
       break;
     default :
       printf("Le nombre n'est pas compris entre 1 et 4");
       break;
   }
```

La présence de `break` est indispensable, vu qu'en son absence, le programme exécute toutes les instructions qui suivent l'étiquette. L'absence de `break` aux étiquettes 1 et 2 fait que le programme exécute dans ce cas le traitement situé à l'étiquette 3. La plus grande prudence s'impose dans l'écriture d'une structure `switch`. En effet, elle admet différentes variantes (non expliquées ici) qui sont tout sauf compatibles avec les règles de la programmation structurée. On s'en tiendra donc strictement à la présentation mentionnée ici.

6.9 Boucles

Peu de langages gèrent les boucles d'une manière parfaitement claire. Le langage Eiffel constitue à cet égard une remarquable exception. Notre pseudo code s'en inspire d'ailleurs. Le langage C connaît trois types de boucles structurées, respectivement `while`, `do...while` et `for`. Reprenons les cinq éléments constitutifs d'une boucle, selon notre approche, et voyons ce que chacun d'eux devient en C :

- l'**initialisation** : j'ai lourdement insisté sur le fait que ces instructions relevaient surtout d'une saine pédagogie. Les instructions de l'initialisation sont tout simplement celles qui précèdent le premier mot de la boucle (`while`, `do` ou `for`).
- la **condition** s'exprime soit après le mot `while` et figure alors entre parenthèses, ou constitue le deuxième argument de `for`.
- la **position de l'évaluation de la condition** se marque par la position de la condition elle-même : dans l'instruction `while(`*`condition`*`)` l'évaluation se fait au début de la boucle, dans l'instruction `do...while(`*`condition`*`)`, l'évaluation se fait après l'exécution des instructions situées entre les deux mots clés.
- le **corps de la boucle** est généralement composé d'une instruction multiple, c'est-à-dire une série d'instructions entourées d'accolades.
- l'**instruction d'avancement** n'a pas de statut particulier, pas plus que dans notre pseudo-code. Il faudra revenir néanmoins sur le caractère particulier de l'instruction `for`.

6.9.1 Les boucles à test initial

Le langage C utilise le mot réservé `while` pour marquer le début d'une boucle. Comme pour l'alternative, la condition est mise entre parenthèse. L'instruction à répéter vient juste après et comme précédemment, pour répéter plusieurs instructions, on utilise des accolades pour les transformer en une instruction multiple.

```
instruction0;
while (Condition)
{    instruction1 ;
     instruction2 ;
                ...
     instructionN ;
}
```

Voici un exemple simple :

```
/* Somme des dix premiers nombres entiers */
#include <stdio.h>
int iSomme, iCompteur;
main()
{  iSomme = 0;
   iCompteur = 1;
   while (iCompteur <11)
   {
      iSomme = iSomme + iCompteur;
      iCompteur++;
   }
   printf("La somme des dix premiers nombres vaut %d.",
          iSomme);
}
```

6.9.2 Les boucles à test final

Le langage C utilise les mots réservés `do` et `while` pour marquer le début et la fin de la boucle. Comme pour l'alternative, la condition est mise entre parenthèse. L'instruction à répéter est placée entre les deux mots réservés et comme précédemment, s'il faut répéter plusieurs instructions, on utilise des accolades pour les transformer en une instruction multiple.

```
instruction0;
do
  {     instruction1 ;
        instruction2 ;
                ...
        instructionN ;
  }
while (Condition) ;
```

Voici par exemple, un petit programme destiné à l'apprentissage du calcul très élémentaire.

```
#include <stdio.h>
int iResultat, int iReponse;
/* Début du programme */
int main()
{ iResultat = 56;
  do
    {
      printf("%s","Combien␣font␣7␣x␣8␣?␣");
      scanf("%d",&iReponse);
    }
  while(iReponse!=iResultat);
  return 0;
}
```

6.9.3 Les boucles `for`

Les boucles commençant par `for` constituent une variante des boucles à test initial. La paire de parenthèses qui suit le mot clé contient trois arguments, séparés par des points virgules :

- les instructions d'initialisation, séparées par des virgules, qui ne sont exécutées qu'une seule fois ;
- la condition de continuation, qui est évaluée avant d'entrer dans la boucle ;
- une ou plusieurs instructions qui font progresser la boucle (si elles sont plusieurs, elles sont séparées par des virgules). Ces instructions sont toujours exécutées après la dernière instruction du corps de la boucle. Elles correspondent à notre moteur.

A titre d'exemple, voici un programme qui additionne les dix premiers nombres dans un accumulateur.

```
#include <stdio.h>

int iSomme, iCompteur;

main()
```

```
{
  for(iSomme=0,iCompteur=1;iCompteur<=10;iCompteur++)
    iSomme+=iCompteur;
  printf("Somme␣des␣dix␣premiers␣nombres␣:␣%d\n␣",iSomme);

}
```

L'instruction `for` est fréquemment utilisée pour traduire notre boucle avec compteur, qui n'a pas d'équivalent exact en C :

```
Init
Compter avec iNombre de 1 A 10
    Afficher entier iNombre ~:
fin de compter
```

Soit en langage C

```
for(iNombre=1;iNombre<=10;iNombre++)
    printf("%d\t␣",iNombre);
```

Vocabulaire

ACCOLADES, AFFECTATION, ARGUMENT, BLOC D'INSTRUCTIONS, FONCTION, FORMAT