par Patrick Gonord

IV - Introduction : Eléments d'un programme C

Ce chapitre présente les éléments principaux entrant dans la composition d'un programme C. Il introduit des principes généraux, utiles pour la compréhension du langage, qui seront détaillés dans la suite du cours.

IV-A - Grandeurs en C

Les grandeurs que manipule le programme C peuvent être classées en deux catégories : des valeurs ou des objets (ou variables). Les valeurs représentent une information alors que les objets sont des éléments permettant le stockage des informations.

IV-A-1 - Valeurs

Une valeur est une grandeur abstraite, non modifiable par nature, exprimant une information. Par exemple le nombre entier 5 , la valeur booléenne Vrai, la couleur Rouge,...
Quelques remarques pour préciser la notion de valeur :
a- Il n'est évidemment pas possible de modifier ce qu'est l'entier 5 pour qu'il devienne la même entité que, par exemple, l'entier 3.
On peut éventuellement changer la couleur de "quelque chose", mais on ne peut changer ce qu'est la couleur Rouge. Ce "quelque chose" possède alors (entre autres) une propriété de couleur qui (peut être) peut changer. On utilisera pour le représenter un **objet** pour stocker cette information (et celles décrivant ses autres propriétés). En conclusion, l'assignation n'est pas définie pour les valeurs.
b- Je ne peux pas faire n'importe quoi avec cet entier 5. Par exemple, écrire **5+Vrai** n'a pas de sens. Par contre, personne ne fera d'objections à ce que j'écrive **5+3** .
c- Ce qu'est l'entier 5 n'est pas modifié que je le représente en chiffres arabes 5 ou en chiffres romains V ou en base binaire 101 plutôt qu'en base décimale. Mais il faut convenir d'une représentation, d'un codage, pour se comprendre. Si je choisis de coder les nombres en chiffres romains, comment faire la différence entre l'entier codé V et la lettre de l'alphabet V ? Un même codage peut représenter deux choses différentes. Le codage ne permet pas en général de savoir le genre de valeur à laquelle on a affaire. Si on ne connaît pas le genre de la valeur codée, on ne peut pas interpréter le code et la valeur est inconnue. On pourrait envisager de résoudre cette ambiguïté en tenant compte du contexte d'emploi de la valeur ou en ajoutant dans le code une information indiquant son genre. La solution adoptée en C est différente.

Pour résoudre ces problèmes, chaque valeur en C possède un **type** qui définit :

1- L'**ensemble** auquel appartient la valeur (dans l'exemple de l'entier 5, l'ensemble (ou plutôt un sous-ensemble) des entiers) ce qui permet de savoir quelles opérations sont définies pour les valeurs de ce type et ce que ces opérations font exactement ;
2- La **représentation** (le codage) en machine utilisée pour les valeurs de ce type.

Une valeur peut être parfois composée à partir d'autres valeurs : Par exemple, une valeur représentative d'une date peut être considérée comme composée de trois autres valeurs : le jour, le mois et l'année (si cette composition convient, car elle n'est pas obligatoire). On dira dans ce cas que la valeur est un *agrégat*.
(*Note* : Le seul cas en C de valeurs agrégats sont les valeurs de type **struct**)

IV-A-2 - Objets

Un objet, ou variable, est un conteneur d'information. Il a un caractère très concret en ce sens qu'un objet est situé quelque part dans le monde physique. On utilise un objet pour représenter une donnée du programme dont la valeur peut changer : le prix du pétrole, l'âge du capitaine,...
En C, un objet est caractérisé par son **type**, son **adresse** et sa **valeur**.

IV-A-2-a - Type d'un objet

Les propriétés d'un objet sont caractérisées par un **type** qui définit :

1- Le **type de l'information** qui est stockée par l'objet et si l'objet est un *agrégat* (composé d'autres objets) ou non ;
2- Les **opérations** qui sont définies pour cet objet ;
3- La **représentation** en mémoire de l'objet et notamment la taille mémoire qu'il occupe. Pour créer un objet, il faut connaître la place mémoire dont il a besoin, donc préciser son type au moment de sa création ;
4- Le mode d'**accès** à l'information stockée par l'objet. Cet accès est fait à partir de ce qu'on appellera ici "la **valeur de l'objet**" (pour des raisons qui apparaîtront lorsqu'on parlera des règles applicables aux opérandes).

Le type d'un objet n'est pas modifiable.
(*Note* : Les objet agrégats en C sont les tableaux et les objets de type **struct**)

IV-A-2-b - Adresse d'un objet

Une valeur permet de caractériser l'emplacement d'un objet : son **adresse**.
Pour accéder à l'information qu'il contient, soit pour la lire soit pour la modifier, il faut savoir d'une manière ou d'une autre où l'objet se trouve, donc connaître son adresse. Comment est représentée cette adresse est sans importance : il suffit qu'à partir de cette valeur on puisse accéder à l'objet.

On a donc besoin d'avoir un type particulier de valeurs pour représenter les adresses d'objets. Ces valeurs seront décrites au chapitre ???
Les objets qui stockent ce type d'information sont appelés des **pointeurs**.

L'adresse d'un objet n'est pas modifiable, autrement dit on ne peut pas déplacer un objet.

IV-A-2-c - Valeur d'un objet

La **valeur d'un objet** permet d'accéder à l'information stockée par l'objet. Deux cas se présentent :

1 **L'objet est un agrégat du genre tableau** ;
dans ce cas, la valeur de l'objet est l'**adresse** où se trouve stockée l'information.
Connaissant cette adresse, on peut alors accéder à l'information. Cette adresse étant non modifiable, la valeur d'un objet du genre tableau est non modifiable.
Le type de l'objet est alors "*tableau de* ...", différent de celui de sa valeur qui est "*adresse de* ...".
2 **Pour tous les autres objets**,
la valeur de l'objet est directement **l'information** contenue dans l'objet.
Si on lit la valeur de l'objet, on lit l'information qu'il contient ; si on modifie la valeur de l'objet, on modifie l'information qu'il contient.
La valeur de l'objet a alors le même type que celui de l'information qu'il contient et ce type sera également le type de l'objet.

Note : les tableaux seront décrits en détails au chapitre ???

La valeur d'un objet est en général modifiable. Les deux exceptions sont
- les objets du genre tableau
- les objets dont la valeur a été déclarée explicitement constante par le mot réservé **const**.

IV-A-2-d - Création et destruction des objets

Deux actions fondamentales sont associées à un objet : sa création et sa destruction.
- **Créer** un objet consiste à lui réserver une place suffisante en mémoire (à une adresse qu'on ne peut pas modifier). Si l'objet a été déclaré constant par le mot clé **const**, sa valeur n'est pas modifiable après sa création et une valeur doit donc lui être attribuée au moment même de sa création (initialisation).
- **Détruire** l'objet consiste à reprendre la place mémoire qui lui a été attribuée et la rendre à nouveau disponible ; l'adresse de l'objet devient invalide et essayer d'accéder à l'objet conduit alors à une catastrophe.
Il est donc important de connaître les procédures conduisant à la création ou à la destruction des objets.

IV-A-2-e - Opérations sur les objets

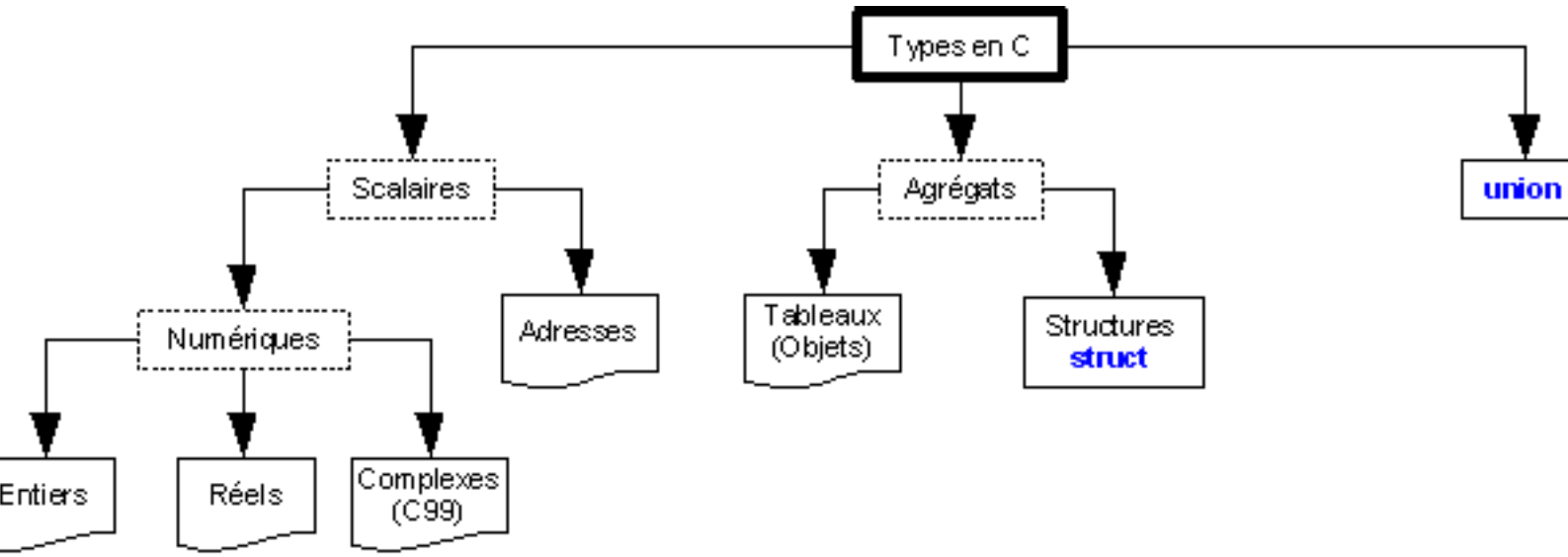
Trois opérations sont définies pour tous les objets. Elles permettent :

- d'obtenir l'objet situé à une adresse donnée :
  Si **A** est l'adresse d'un objet, l'objet situé à cette adresse est ***A**
  A est une valeur et ***A** est un objet ;
- d'obtenir l'adresse d'un objet :
  Si **Objet** est un objet, son adresse est **&Objet**
  C'est une valeur. Il s'ensuit que ***&Objet** est l'objet **Objet** ;
- d'obtenir la place mémoire occupée par un objet :
  Si **Objet** est un objet, **sizeof Objet** est la taille en bytes occupée par **Objet** en mémoire.

Note : un byte est la plus petite quantité mémoire repérable par une adresse ; sa valeur dépend des systèmes (souvent, mais pas toujours, un octet soit 8 bits).

IV-A-3 - Types en C

Le diagramme ci-dessous récapitule les différents types de données disponibles en C. Ils seront décrits en détails dans les chapitres suivants.



IV-B - Expressions

Une expression décrit un ensemble d'opérations portant sur des valeurs ou des objets. Elle est constituée d'opérateurs et d'opérandes. Dans sa forme la plus réduite, ce peut être simplement une valeur (ce qui n'a d'utilité que si cette expression constitue un appel à une fonction).

IV-B-1 - Opérateurs et opérandes

Un opérateur indique une opération à effectuer ; ses opérandes sont les grandeurs utilisées pour réaliser l'opération. En C, selon l'opérateur, il peut avoir un, deux ou trois opérandes ; on parle alors d'un opérateur unaire, binaire (rien à voir ici avec le code binaire) ou ternaire. Selon l'opérateur, les opérandes doivent être des valeurs ou des objets et le résultat de l'opération peut être une valeur ou un objet.
Ainsi, on a déja vu les opérateurs unaires *****, **&** et **sizeof** :

- L'opérateur unaire ***** a un opérande qui est une valeur et le résultat est un objet (celui situé à l'adresse spécifiée par l'opérande).
- L'opérateur unaire **&** a un opérande qui est un objet et un résultat qui est une valeur (l'adresse de l'objet désigné par l'opérande).
- Le cas de l'opérateur **sizeof** est particulier :
  Son opérande peut être un objet, un type ou une valeur. Le résultat est une valeur entière donnant la taille mémoire de l'objet, d'un objet de ce type ou d'un objet du même type que la valeur.

On peut citer comme exemples simples d'opérateurs binaires l'addition **+** et la multiplication ***** de deux entiers comme dans les expressions **2+3** et **2*3**. L'évaluation de ces expressions donnera des valeurs, respectivement les entiers 5 et 6. Ces deux opérateurs ont deux opérandes qui sont des valeurs et donne un résultat qui est une valeur.

Certains opérateurs donnent comme résultat une valeur à caractère booléen, c'est à dire portant comme information *Vrai* ou *Faux*. C'est le cas des opérateurs de comparaison. Par exemple, l'opérateur **==** compare ses opérandes pour savoir si ils sont égaux : **2==3** a une valeur qui signifiera *Faux* et **3==3** une valeur qui signifiera *Vrai*. Les valeurs à caractère booléens seront décrites en détails au chapitre ???.

IV-B-2 - Règle de base sur les opérandes

Un opérateur possède donc un contexte qui définit la nature, valeur ou objet, de chaque opérande et du résultat de l'opération. Les règles suivantes établissent comment le contexte de l'opérateur est respecté :

> 1- L'opérateur attend un objet comme opérande. Alors l'opérande **doit** être un objet.
> 2- L'opérateur attend une valeur comme opérande. Alors si l'opérande désigne un objet, ce sera interprété comme la *valeur de l'objet*.

Exemple : Considérons l'opérateur binaire **=** qui permet de modifier la valeur d'un objet et l'expression **Obj = V**.
- Contexte : **Obj** doit être un objet (non déclaré constant); **V** est une valeur. L'opérateur **=** donne comme résultat une valeur, égale à celle donnée à l'objet et du même type que celui-ci.
- On peut écrire (supposons que **Obj** est d'un type entier) **Obj = 2** pour que **Obj** prenne pour valeur 2. Cette expression a pour valeur 2.
On peut aussi écrire **Obj1 = Obj2** où **Obj2** est un objet (que nous supposerons aussi de type entier). Comme **Obj2** est placé dans un contexte de valeur, il désigne ici la *valeur* de **Obj2** qui deviendra la valeur de **Obj1**.
- A noter que comme la valeur d'un objet tableau n'est pas modifiable, on ne pourra jamais écrire **Obj = ...** si **Obj** est un objet tableau. Autrement dit, l'opérateur **=** n'est pas défini pour les objets tableaux.

IV-B-3 - Evaluation

Pour évaluer le résultat, il faut évaluer les opérandes puis faire l'opération. Pour les opérateurs binaires, se pose la question de l'ordre d'évaluation des opérandes. La réponse est (en général) que cet ordre est indéterminé et son choix est laissé à la convenance du compilateur (les trois exceptions concernent les opérateurs binaires **&&** et **||** ainsi que l'opérateur **,** (virgule) qui seront décrits ultérieurement).
Le résultat ne doit pas dépendre de l'ordre d'évaluation des opérandes sinon il est imprévisible et l'expression est mal formée.
Que se passe t-il si les deux opérandes sont de types différents, par exemple un entier et un réel comme dans **2+3.5** ? Dans ce cas, le compilateur va essayer de trouver un type commun pour lequel l'opérateur est défini et transformer les deux opérandes dans ce type commun pour effectuer l'opération (cette transformation d'une valeur est appelé transtypage ou **cast**). Naturellement, il doit exister une règle de transformation des opérandes de leur type en le type commun. Le résultat sera alors de ce type commun. Le choix du type commun obéit à des règles précises qui seront énoncées au moment opportun.

IV-B-4 - Evaluation des expressions

Un opérateur peut avoir comme opérande une expression pourvu que le résultat de celle-ci soit une valeur ou un objet selon les exigences du contexte de l'opérateur. Si l'expression complète comporte plusieurs opérateurs, comment est-elle interprétée ?
Considérons l'expression **2*3+4** .
La question se pose donc de savoir comment le compilateur va interpréter cette expression : comme le produit de 2 et 3 auquel on ajoute 4 (et le résultat sera 10) ou comme le produit de 2 et de la somme de 3 et 4 (et dans ce cas le résultat sera 14) ?
Une possibilité est de mettre des parenthèses pour lever cette ambiguïté et écrire, selon le résultat désiré, **(2*3)+4** ou **2*(3+4)**.
En l'absence de parenthèses l'ambiguïté est levée en considérant deux propriétés des opérateurs : leur **priorité** et leur **associativité**. Ainsi **2*3+4** est interprété comme **(2*3)+4** et le résultat sera 10.
Le mécanisme priorité+associativité sera décrit en détail plus loin.
Comme autre exemple, considérons l'expression légale **A = B = C** où, pour simplifier, **A**, **B** et **C** ont le même type.
Sans même connaître le mécanisme priorité+associativité, on peut être sûr qu'il faut l'interpréter comme **A = (B = C)**.
L'interprétation **(A = B) = C** est impossible : **(A = B)** donne pour résultat une valeur or on doit avoir ***objet* = ...** ce qui viole le contexte des opérandes de l'opérateur **=**.
Considérons donc **A = (B = C)**. **B =...** impose que **B** soit un objet. De même, **A = ...** impose que **A** soit un objet. **C** est une valeur (ou la valeur d'un objet).
B = C est une valeur égale (ici) à celle de **C**. Au final, **A** et **B** prennent pour valeur celle de **C** ; L'expression a pour valeur la nouvelle valeur de **A** et a le même type que **A**.

IV-C - Instructions

On peut distinguer deux groupes d'instructions : les instructions formées d'une expression et les instructions de contrôle.

IV-C-1 - Expressions

En dehors des instructions de contrôle, que nous verrons ensuite, toutes les instructions d'un programme C sont des expressions (terminées par un ;) : exécuter ces instructions consiste simplement à évaluer ces expressions.
Pour qu'une instruction ait une influence sur le programme et serve à quelque chose, il faut que cette évaluation modifie l'état du programme ; on dit alors qu'elle a des **effets de bords**. Le plus souvent, l'effet de bord consiste à modifier la valeur d'un objet, à lire des données écrites dans un fichier ou à partir du clavier ou à écrire des données dans un fichier ou sur la console.
Exemple : L'instruction **2+3;** n'a absolument aucune action, ne fait rien et est inutile.
L'instruction **A = 2+3;** (où **A** est un objet) a un effet de bord : elle modifie la valeur de l'objet **A**.

Toutes les expressions ne sont pas des instructions : elles peuvent, par exemple, servir à établir les conditions de fonctionnement d'une instruction de contrôle ou à initialiser la valeur d'un objet.

IV-C-2 - Instructions de contrôle

L'exécution d'un programme C est séquentielle : Après avoir exécuté une instruction, on exécute l'instruction suivante dans le programme. Ce processus peut être modifié en utilisant des instructions de contrôle de flux du programme. Elles permettent d'exécuter une partie du programme uniquement si une condition est remplie, de répéter l'exécution d'une partie du programme tant qu'une condition est remplie, etc.
Ces instructions de contrôle seront décrites en détails au chapitre ???.

IV-C-3 - Bloc d'instructions

La plupart des instructions de contrôle de flux du programme conditionne l'exécution d'une seule instruction. Si, et le cas est très fréquent, vous souhaitez soumettre à condition l'exécution de plus d'une instruction, vous pouvez grouper ces instructions dans un bloc, c'est à dire englober les instructions entre les signes **{** et **}**.
Là où la syntaxe demande une instruction, vous pouvez placer un bloc d'instructions.

IV-D - Fonctions

Une fonction sert à effectuer une opération ou une action.
En ce sens, on peut la considérer comme un opérateur défini par le programmeur : ses arguments jouent le rôle des opérandes d'un opérateur et à partir de la valeur de ses arguments, elle peut fournir un résultat. Comme un opérateur, une fonction peut avoir des effets de bords
Une fonction doit effectuer un calcul (ou une action) précis et bien délimité. Les fonctions qui font plusieurs choses à la fois ont peu de souplesse d'emploi et leur utilisation est restreinte à une situation spécifique. Ecrivez plutôt plusieurs fonctions, chacune dédiée à une tâche précise et que vous pourrez ainsi réutiliser.
Exemple : Vous souhaitez calculer la valeur de N! où N est un entier entré au clavier et afficher le résultat sur la console.
Ecrivez une fonction qui prend en argument la valeur N , calcule sa factorielle et renvoie la valeur obtenue. La fonction peut jouer alors un rôle équivalent à celui d'un opérateur de calcul de la factorielle, opérateur qui n'existe pas en C. Mais la fonction ne doit pas lire sur le clavier la valeur de N ni afficher la valeur obtenue sur l'écran. Sinon la fonction est inutilisable si N est une valeur calculée par le programme ou lue dans un fichier ou si vous n'avez pas besoin d'affichage de la valeur obtenue. La procédure à suivre est donc :
- Lire au clavier la valeur de N
- Utiliser la fonction avec cette valeur comme argument
- Afficher le résultat à la console

Trois étapes que vous pouvez grouper dans une autre fonction, le calcul de la factorielle restant disponible pour être utilisé ailleurs.
Les fonctions ne doivent pas être conçues simplement pour découper arbitrairement votre code en tranches plus petites mais pour le structurer, le rendre plus clair et pour éviter de dupliquer des morceaux de code identiques qui nuisent à la maintenance du programme. C'est pourquoi, vous serez amené à écrire de nombreuses fonctions de très petite taille. Il est d'ailleurs rare qu'on soit contraint d'écrire des fonctions dépassant une centaine de lignes et dans un tel cas, on doit se poser la question : Est-ce que ma fonction ne fait pas trop de choses ?
Il est également rare que les fonctions nécessitent un grand nombre d'arguments. Si c'est votre cas, posez-vous la question : Est-ce que mes données sont bien organisées, bien représentées par les types de données que j'ai choisis ?
Une fonction doit former un ensemble autonome, c'est à dire que son fonctionnement ne doit dépendre que des arguments passés à la fonction et non pas d'éléments qui lui sont extérieurs. De cette façon, une fois que sont définis son rôle et les paramètres dont elle a besoin, vous pouvez écrire son code en faisant totalement abstraction du reste du programme. La fonction écrite et testée, l'utilisateur peut totalement ignorer le code de la fonction. Seul son "mode d'emploi" lui est nécessaire : savoir précisement ce qu'elle fait et ce que représente chaque paramètre. Elle est alors devenu un outil qui peut être utilisé et réutilisé à loisir.
En C, les arguments des fonctions sont **toujours** des valeurs et le résultat est lui aussi **toujours** une valeur (jamais un objet).