Introduction au langage C

Table des matières

Chapitres du cours

I

A. Introduction au langage C

1. Langage C et norme ANSI

Définition

Le langage C est né en 1972, date à laquelle Denis Ritchie l'a conçu dans un but précis : écrire un système d'exploitation (UNIX). Il s'est inspiré du langage B créé précédemment par K.Thompson qu'il a haussé au niveau d'un langage évolué, notamment en l'enrichissant de structures de contrôle (boucles) et de constructeur de types, tout en lui conservant ses aptitudes de programmation proche de la machine.

Il a toutefois fallu attendre la parution, en 1978, de l'ouvrage "The C programming language" De Kernighan et Ritchie pour voir apparaître une première définition du langage C. Depuis cette date, le langage C a continué d'évoluer, à travers les nombreuses versions de compilateurs qui ont vu le jour.

A partir de 1982, l'ANSI (American National Standards Institute) a formé un comité (connu sous le sigle X3J11) chargé de définir un standard. Ses travaux ont abouti à une proposition de normalisation, celle-ci définit les règles syntaxiques du langage C. En outre, elle fournit également les spécifications d'un ensemble de routines (des fonctions et des macros) formant ce que l'on nomme la "bibliothèque standard". Ce point est fondamental car la moindre opération d'entrée-sortie (écriture d'information sur l'écran et lecture d'information depuis le clavier par exemple) en C fait appel à au moins une routine de cette bibliothèque.

En effet, contrairement à d'autres langages le C ne possède pas de telles fonctions de façon native. L'usage veut que l'on parle de «C norme ANSI ». Le langage C est un langage compilé, ce qui signifie qu'un programme écrit en C doit subir une suite

de traitements pour que la machine puisse l'exécuter. Nous les détaillerons dans ce chapitre.

2. Environnements de programmation en C

Conseil

Pour pouvoir suivre ce cours avec succès, vous avez besoin d'un compilateur ANSI-C

Exemple

- Turbo C
- Borland C
- Microsoft C
- intel C
- gcc cc
- Zortech C
- Symantec

Remarque

Les système d'exploitation de type UNIX disposent de compilateurs C immédiatement disponible.

Il s'agit de gcc et cc.

3. Le fichier source

Pour créer un fichier source, il faut utiliser un programme appelé éditeur de texte, puis taper votre programme en respectant la syntaxe du langage C et enregistrer le fichier ainsi créé en indiquant son nom. Le fichier source d'un programme écrit en langage C est un simple fichier texte dont l'extension est par convention « .c » (exemple toto.c).

Attention

L'extension doit être en minuscule.

Définition

Ce fichier source doit être un fichier texte non formaté, c'est-à-dire un fichier texte dans sa plus simple expression, sans mise en forme particulière ou caractères spéciaux, il contient uniquement les caractères ASCII de base (ASCII = American Standard Code for Information Interchange. C'est une norme de codage pour les caractères.

Elle définit 128 caractères numérotés de 0 à 127). L'ensemble du texte doit respecter la syntaxe du langage C.

4. Premier programme en C

Fondamental

Un programme exécutable écrit en langage C, comporte obligatoirement une fonction principale appelée main() renfermant les instructions qui doivent être exécutées. Celles-ci sont comprises entre les accolades ouvrante et fermante qui suivent le nom de la fonction.

Exemple

Voici un exemple de programme en C :

```
#include<stdio.h>
void main()
{
   printf("Ceci est votre premier programme\n");
}
```

Définition

Il s'agit du texte contenu dans un fichier nommé «toto.c» et vous le voyez tel qu'il apparaît dans l'éditeur de texte. Ce texte respecte la syntaxe du C.

On vous demande de l'admettre pour l'instant. Les éléments d'explications vous seront fournis dans les chapitres suivants.

Le type retourné par la fonction main() est void, c'est-à-dire « rien ». La norme actuelle du C (C99) impose que le type retourné par la fonction soit explicitement annoncé, on ne peut donc pas écrire « main() » sans spécifier le type de retour de cette fonction .

Cette notion de type est importante et sera développé dans les chapitres suivants. Le langage C est un langage dit fortement typé. La fonction printf() produit une émission de caractères en séquence vers la sortie standard nommée stdout (par défaut il s'agit de l'écran).

Il faut inclure un fichier nommé « stdio.h » qui définit l'usage de cette fonction printf. Entre une paire de " (double quote) nous avons une chaîne de caractères constante. On parle de littéral constant de type chaîne de caractères. De fait, "Ceci est votre premier programme \n" est donc une chaîne de caractères qui va s'afficher telle quelle.

Il s'agit du paramètre réel (au sens effectivement transmis) de la fonction printf. La séquence des deux caractères \ et n sera interprétée par la fonction printf comme l'affichage d'un saut de ligne. Elle correspond au caractère « saut de ligne » qui existe dans la table des codes ASCII.

5. Précompilation et Compilation

Définition

Parler de la compilateur C constitue un abus de langage. Il s'agit en fait d'un outil qui enchaîne trois étapes :

- La précompilation
- La compilation
- L'édition des liens

Par exemple sous UNIX la compilation est lancé par la commande « cc toto.c ». Où cc est le nom du compilateur.

Le programme source toto.c ne peut pas être exécuté de manière immédiate par l'ordinateur tel qu'il se présente à nos yeux. Il faut le traduire en langage machine (ou langage binaire), c'est à dire des instructions élémentaires que le microprocesseur peut exécuter. Pour rendre le texte exploitable par la machine, on utilise un programme destiné à le traduire : le compilateur C.

La pré-compilation modifie le texte du fichier source en interprétant les directives destinées du préprocesseur. Le préprocesseur est un programme qui va traiter des directives qui lui sont destinés. Les directives de pré-compilation commencent par le symbole #. Elles sont facilement repérables. Le C ne possède pas la possibilité

d'écrire un texte sur l'écran : on doit donc inclure une bibliothèque qui intègre ces fonctions manquantes. La directive #include <stdio.h> réalise ainsi l'inclusion des définitions précises des fonctions standards d'entrées/sorties. Ces définitions se trouvent dans le fichier stdio.h. «stdio.h» signifie standard input/output et l'extension «.h» indique un fichier «header», simplement c'est le fichier d'entête qui donne la définition de certaines fonctions.

A ce stade, la précompilation est terminé nous obtenons un fichier dans lequel des directives ont été appliquées. Le compilateur proprement dit est un programme qui va vérifier la syntaxe et la concordance des types du fichier généré par le pré-compilateur. Si le programme source ne comporte pas d'erreur de syntaxe il va générer un fichier composé d'instructions machines.

Le résultat de la compilation est un fichier appelé module objet, vous avez obtenu un nouveau fichier toto.o à partir du fichier source toto.c. Bien que formé d'instructions machine, il n'est pas exécutable. En effet, la précompilation a permis de donner la définition de la fonction printf. La compilation a préparé son usage mais n'a pas ajouté les séquences d'instructions machine qui la réalise. L'ensemble des instructions de toutes les fonctions définies dans stdio.h (la bibliothèque proprement dite) se trouve ailleurs dans un autre module objet déjà compilé. Il reste à les lier avec notre fichier objet pour pouvoir les utiliser. Le programme compilé est prêt à utiliser la fonction printf mais il ne la possède pas.

6. Édition de liens

Définition

Le rôle de l'éditeur de lien est de réunir le module objet toto.o précédemment construit et le module correspondant à la bibliothèque standard (<stdio.h>) pour générer le programme exécutable nommé par exemple « toto.exe ». L'éditeur de lien va remplacer l'appel préparé de la fonction printf par son appel réel.

Remarque

Dans la plupart des cas, les trois programmes éditeur de texte, compilateur (avec le pré-compilateur) et l'éditeur de liens sont intégrés harmonieusement au sein d'un environnement dédié au développement en langage C. Les étapes compilation puis édition de liens sont enchaînées automatiquement sans que le programmeur ait besoin de le faire explicitement.

A vous de vous familiariser avec l'environnement de développement choisi. Vous n'avez pas à vous préoccuper de savoir où se trouve le module objet associé aux fonctions définies dans « stdio.h ». A l'installation du compilateur ces fichiers sont rangés quelque part. Le compilateur sait les retrouver, c'est implicite.

Les étapes de précompilation, de compilation et d'édition des liens sont généralement automatiquement enchaînée. Les fichiers intermédiaires issus de la précompilation et de la compilation sont automatiquement détruits. Il est cependant possible d'utiliser des options pour les effectuer séparément afin d'obtenir ces fichiers intermédiaires.

7. Fonctionnement

Il reste à lancer l'exécution de notre premier code C exécutable. A l'écran la fonction printf affiche :

Ceci est votre premier programme

puis saute une ligne à cause du caractère \n qui demande explicitement à la fonction printf de sauter une ligne. Par défaut printf ne saute aucune ligne.

8. Premiers éléments en C

La manière d'écrire un code en langage C a son importance. Le langage C est par exemple sensible à la casse (en anglais case sensitive), cela signifie qu'un nom contenant des majuscules est différent du même nom écrit en minuscules. Ainsi, les spécifications du langage C précisent que la fonction principale doit être appelée main() et non Main() ou MAIN(). De la même façon, on remarquera que la fonction printf() est écrite en minuscules.

L'instruction printf() se termine par un « ; », car en langage C, toute instruction se termine par un point virgule. Le compilateur détecte ainsi la fin d'une instruction. Le « ; » est ce que l'on appelle un délimiteur.

Le langage C est un langage à typage fort, toute information doit obligatoirement avoir un type clairement défini. Dans l'exemple nous avons écrit

```
void main ()
```

Nous avons dit que le main est la fonction principale. C'est une fonction, elle renvoie donc une valeur et c'est le cas. La fonction dont le nom est main renvoie une information de type void. void est un des type simple de C. Cette notion de type simple sera développée plus loin. A ce stade, nous dirons que void signifie « rien » ce qui est logique car la fonction main ne renvoie pas d'information.

Dans l'exemple nous avons utilisé une règle implicite que nous rendons explicite : en C nous ne pouvons utiliser une chose que si cette chose est définie avant. Nous utilisons la fonction printf pour réaliser un affichage. Comme nous l'avons expliqué cette fonction n'existe pas directement dans le langage C. Le compilateur ne la connait donc pas. Sauf que nous avons pris soin d'inclure sa définition grâce à la directive #include <stdio.h> avant de l'utiliser. Les inclusions de bibliothèques s'effectuent généralement au début du programme pour plus de clarté.

9. Erreur et avertissement à la compilation

Quand le programmeur déclenche la compilation, le compilateur affiche éventuellement des messages. Deux types de messages existent :

- Error
- Warning

Définition

Error est suivi d'un message en anglais concernant la nature de l'erreur détectée. Ce qui signifie que la compilation a échoué en un point du programme source. L'erreur se situe donc avant ce point. Vous n'avez pas respecté la syntaxe du langage C, à vous de corriger votre code source avant de relancer une compilation. Le message est censé vous aider à trouver la nature de cette erreur.

Définition

Warning est suivi d'un message en anglais. C'est un avertissement. Le compilateur a réalisé le travail mais vous signale qu'il a détecté un problème potentiel quand vous exécuterez votre programme. A vous de trouver la cause de cette avertissement.

Un code C correctement écrit ne génère évidemment aucun message « error » et s'il est vraiment bien écrit aucun message « warning ». La maîtrise de la syntaxe, la rigueur et l'expérience permettent d'obtenir des codes sources corrects.

Une compilation réussie sans erreurs ni warnings ne garantit en aucun cas une bonne exécution du programme. Le compilateur ne vérifie pas les erreurs de logique ou d'algorithmique. Il ne vérifie que la syntaxe et la concordance des types.

En résumé, ce n'est pas parce qu'un programme est compilé correctement qu'il va fonctionner.

10. Ajout de commentaires

Lorsqu'un programme devient long et compliqué, il peut être intéressant (il est même conseillé) d'ajouter des lignes de commentaires dans le programme, c'est-à-dire des portions du fichier source qui ont pour but d'expliquer le fonctionnement du programme sans que le compilateur ne les prenne en compte (car il génèrerait une erreur). Pour ce faire, il faut utiliser des balises qui vont permettre de délimiter les explications afin que le compilateur ignore le texte qui les suit.

Syntaxe

Deux types de balises possibles:

«//»: mets en commentaire tout ce qui suit sur la ligne

«/*»: met en commentaire tout ce qui suit dans le fichier jusqu'à la balise fermante «*/»

Exemple

```
#include < stdio.h >
//ceci est mon programme main
void main() // le main
{
/*
cette fonction affiche en sortie standard la phrase:
«Ceci est votre premier programme»
*/
printf("Ceci est votre premier programme");
}
```

Remarque

Les commentaires ne peuvent être imbriqués les uns dans les autres.

Les commentaires sont supprimés par la précompilation.

B. Algorithmes et langages

En informatique, la programmation impérative est un paradigme de programmation qui décrit les opérations en termes de séquences d'instructions exécutées par un ordinateur pour modifier l'état du programme.

Le langage C est un langage impératif. Il est donc essentiel d'être capable de définir clairement l'état d'un programme et la séquence d'instructions qui modifie cet état. Ce qui nous amène naturellement à la notion d'algorithme.

1. Éléments de base

a) Notion d'algorithme

Étant donné un traitement à effectuer sur des données, un algorithme est l'énoncé d'une séquence d'actions primitives réalisant ce traitement sur les données.

Il s'agit de l'ensemble des règles opératoires dont l'application permet de résoudre un problème au moyen d'un nombre fini d'instructions.

L'algorithmique est ainsi une discipline de l'informatique qui consiste à bien analyser le problème avant de commencer à programmer, et à proposer un algorithme adapté à ce problème.

Nous présentons ci-dessous deux exemples très simples d'algorithmes. Le premier énonce les règles à mettre en œuvre pour « sortir une voiture du garage », le second, celles qui permettent de « résoudre une équation du second degré ».

Exemple : Exemple n°1 : Sortir une voiture du garage

1. Ouvrir la porte du garage
2. Prendre la clef
3. Ouvrir la porte avant gauche
4. Entrer dans la voiture
5. Mettre au point mort
6. Mettre le contact

...

Exemple : Exemple n°2 : Équation du premier degré A X + B = 0

```
1: Lire les coefficients A et B
2: Si A est non nul
  alors
    affecter à X la valeur - B / A
   afficher à l'écran la valeur de X
 Sinon
   Si B est nul
    alors
      Afficher à l'écran "tout réel est solution"
    Sinon
      Afficher à l'écran "pas de solution"
```

Remarque

Un algorithme doit être indépendant du langage informatique. Une fois l'algorithme mis au point, vérifié et stabilisé, il est transcrit dans le langage informatique choisi pour l'implémenter.

Vous remarquerez que certaines instructions sont numérotées. Cela correspond ici à la fois à une étiquette qui repère l'instruction et à l'ordre dans lequel elles doivent être exécutées. Cependant, étiquette et ordre sont deux notions différentes.

Dans l'exemple 2, certaines instructions comme « affectation à X la valeur de -B/A » ne sont pas étiquetées. Leurs emplacements dans le texte qui décrit l'algorithme sont suffisants pour indiquer sous quelles conditions elles sont exécutées.

b) Caractéristiques d'un algorithme

Nous proposons quelques règles qui permettent de caractériser plus précisément un algorithme

- Tout objet manipulé dans l'algorithme doit être clairement défini avant son usage.
- Chaque objet ne doit souffrir d'aucune ambiguïté.
- Toute combinaison d'opérations élémentaires doit utiliser des opérations supposées connues.
- Pour chaque jeu de données d'entrée, l'algorithme doit fournir un résultat grâce à un nombre fini d'opérations.

2. Méthodologie

La méthode que nous proposons pour développer un algorithme comporte trois étapes :

Méthode

1.Définir clairement le problème

2.Rechercher une méthode de résolution (formules...)

3.Réaliser l'algorithme en étant explicite.

a) Définition du problème

- Spécification d'un ensemble de données à partir de l'énoncé du problème, d'hypothèses, ou de source d'informations externes, ...
- Spécification d'un ensemble de buts à atteindre à partir des résultats attendus, des suites d'opérations à effectuer, ...
- Spécification des contraintes à partir de l'énoncé, du contexte, ...

b) Recherche d'une méthode de résolution

Dans un premier temps il faut :

- Clarifier l'énoncé
- Simplifier le problème
- Ne pas chercher à le traiter directement dans sa globalité
- S'assurer que le problème puisse être traité car il existe des problèmes indécidables.

Ensuite il faut rechercher une stratégie de construction de l'algorithme :

- Décomposer le problème en sous problèmes partiels plus simples, dont la résolution implique la solution du problème global
- Effectuer des raffinements successifs
- Le niveau de raffinement le plus élémentaire est celui des instructions.

Il n'existe pas de stratégie automatique pour mettre au point une méthode de résolution.

i - Réalisation d'un algorithme

Définition

Un algorithme est un ensemble de règles ou d'instructions destinées à résoudre le problème posé.

Il doit être conçu indépendamment du langage de programmation et du système

informatique.

Attention

- L'algorithme doit être exécuté après un nombre fini d'opérations.
- L'algorithme doit être spécifié clairement, sans la moindre ambiguïté.
- Le type des données doit être précisé (entier, réel, etc.).
- L'algorithme doit fournir au moins un résultat. (un affichage est un résultat)
- L'algorithme doit être effectif : toutes les opérations doivent pouvoir être simulées par un homme en un temps fini.

3. Notion d'identificateur

La construction d'un nom pour désigner un objet de manière unique suit des règles précises en programmation. Le nom choisi est ce que l'on appelle un identificateur.

Nous allons, dans ce qui suit, donner les règles qui concernent le langage C, règles que nous allons utiliser pour l'algorithmique.

Un identificateur est un mot constitué d'au moins un symbole parmi les symboles qui correspondent à des lettres minuscules (de 'a' à 'z'), des lettres majuscules (de 'A' à 'Z') ou le symbole '_' (lire underscore).

S'il y a plusieurs symboles alors ils sont juxtaposés, le caractère espace est interdit. S'il y a plusieurs symboles alors les symboles qui suivent le premier peuvent également être choisis parmi les chiffres de '1' à '9'.

Exemple : Exemple d'identificateurs corrects :

- x
- vitesse_objet
- Pi
- true54

Exemple : Exemple d'identificateurs incorrects :

- 34x : car il ne commence pas par 'a' ..'z' ou 'A'..'Z'
- vitesse objet : car il y a un espace entre vitesse et objet

Nous introduisons ci-après les éléments nécessaires à la réalisation d'algorithme.

D'autres éléments seront introduit en fonction des besoins par la suite.

4. Type, Donnée, constante, et variable

a) Type

Définition

Dans la vie courante nous utilisons des entiers, des réels, ... pour effectuer des calculs.

Ils correspondent à ce que nous appelons des types en informatique.

b) Donnée

Définition

Une donnée est une valeur manipulée par l'utilisateur pendant l'exécution du programme.

Elle peut être lue interactivement, calculée ou issue d'un fichier de données.

Exemple

la température, l'âge du patient, etc.

Attention : Nécessités

- Définir le type des données (entiers, réels, etc).
- Structurer et organiser les données.

c) Constante

Définition

Une constante est une valeur affectée lors de la programmation.

Elle ne peut changer lors de l'exécution du programme.

Il est préférable de désigner une constante par un nom (identificateur) si elle doit être utilisée souvent.

Exemple

Pi, la constante de gravitation, etc.

d) Variable

Une variable est désignée par un identificateur (il s'agit de son nom).

Une variable correspond à un réceptacle d'une certaine taille qui permet de stocker un type d'information. Le réceptacle contient une valeur (la valeur de la variable) qui est susceptible de changer au cours de l'exécution de l'algorithme.

Le couple (identificateur, réceptacle) définit une variable.

Attention : Nécessités

- Il faut définir le type des variables (entiers, réels ...) car la taille du réceptacle dépend du type et parce que la nature des opérations (+,-, ...) que l'on peut effectuer dépend du type des variables mises en jeux.
- Il faut choisir des identificateurs ayant un sens à la lecture, cela facilite le travail.
- Dans les langages compilés comme le C, il faut déclarer les variables avant de les utiliser. C'est-à-dire donner un type et un identificateur.

Exemple : Exemples de variables

- L'inconnue dans une équation dont les paramètres changent
- Le dosage des médicaments choisis pour un malade donné
- La vitesse d'un objet en chute libre...

e) Opérateur et notion de compatibilité de type

Pour effectuer des calculs, nous utilisons des opérateurs comme l'addition. Nous utilisons usuellement le signe '+' pour désigner l'opération d'addition. Le signe '+' est un opérateur binaire car pour pouvoir l'appliquer nous avons besoin de deux opérandes. Quand nous utilisons l'addition dans la vie courante, nous vérifions implicitement que les deux opérandes peuvent être additionnées.

En informatique nous appelons cela la vérification de la compatibilité des types des opérandes.

En résumé, additionner un entier et un réel est possible, par contre additionner un entier et un caractère n'a pas de sens. Il en est de même en informatique, les opérateurs ne peuvent être utilisés que sur des données dont les types sont compatibles par rapport à l'opération que l'on souhait effectuer.

5. Éléments d'algorithmique

a) Affectation

Définition

L'instruction d'affectation est l'opération qui consiste à attribuer une valeur à une variable pendant l'exécution du programme.

Cette opération consiste à changer le contenu du réceptacle. L'ancien contenu est perdu. Nous avons écrit dans l'exemple 2 « affecter à X la valeur -B/A ».

En algorithmique on utilise un opérateur d'affectation. Nous le notons = dans le cadre de ce cours.

On note : Variable = constante ou résultat de l'évaluation d'une expression.

Exemple

X = 1

(la variable dont l'identificateur est x prend la valeur constante 1, le réceptacle contient désormais la valeur 1)

X = 2 x 3+5

(la variable x prend la valeur du résultat de l'évaluation de l'expression 2 x 3+5)

X = -B/A (la variable x prend la valeur du résultat de l'évaluation de l'expression -B/A, c'est-à-dire que l'on prend la valeur de la variable B que l'on divise par la valeur de la variable A, puis on calcule l'opposé)

Le signe = est utilisé pour l'affectation. Or, nous allons avoir besoin de tester l'égalité de deux choses. Si l'on utilise encore le signe = il y a risque de confusion.

A un opérateur correspond un ou plusieurs symboles qui l'identifie de manière unique. Nous décidons de doubler le signe = pour définir l'opérateur de test d'égalité (soit ==).

Lors de la transcription de l'algorithme vers le langage de programmation ciblé, il faut trouver l'équivalent de l'opérateur d'affectation '='.

L'opérateur d'affectation existe toujours dans un langage impératif. Désormais A = B signifiera que « A prend la valeur de B ».

En revanche, A == B signifiera « est-ce que A est égal à B » ? Et la réponse sera alors soit vrai soit faux.

b) Branchement conditionnel ou structure de sélection simple

Il existe deux formes de branchements conditionnels possibles :

Syntaxe : Première forme

SI <Condition> ALORS

    ensembles d'instructions si <Condition> est vraie

première instruction qui suit le si ... alors

Explication

Si la <Condition> est vraie alors l'ensemble d'instructions situées après le “alors” sont exécutées (l'une après l'autre).

Puis l'algorithme se poursuit et on exécute l'instruction qui suit le si ... alors.

Si la <condition> est fausse alors aucune des instructions qui doivent s'exécuter si la <condition> est vraie n'est effectuée et l'algorithme se poursuit en exécutant l'instruction qui suit le si <condition> alors. Dans ce cas, on dit qu'il y a branchement.

Syntaxe : Deuxième forme

SI <Condition> ALORS

ensembles d'instructions si vraie

SINON

ensembles d'instructions si fausse

première instruction qui suit le si ... alors ... sinon ...

Explication

Si la <Condition> est vraie alors l'ensemble des instructions situées après le « alors » est exécuté puis on se branche au-delà du « sinon » et de la suite d'instructions qui lui sont associées, puis l'algorithme se poursuit et on exécute les instructions en italique.

Si la <Condition> est fausse alors on « saute » le « alors » et l'ensemble des instructions qui sont associées. Là encore, il y a branchement. Puis on exécute les instructions qui suivent le "sinon", puis l'algorithme se poursuit en exécutant l'instruction en italique

Exemple

- Forme simple :

"si survient un virage,

alors je tourne le volant"

- Forme alternative :

"si le feu est au vert,

alors je peux passer sinon je dois m'arrêter !"

"si deux droites sont sécantes,

alors elles ont un unique point commun

sinon elles sont parallèles."

Valeur absolue d'un nombre :

si X >= 0 alors val_abs_x = X

sinon val_abs_x = - X

Maximum de deux nombres :

si A >= B alors MAX = A

sinon MAX=B

De même lors de la transcription il faudra trouver les équivalents dans le langage cible.

c) Séquence d'instructions

Vous avez naturellement lu les exemples d'algorithmes de haut en bas, ligne par ligne et pour chaque ligne de la gauche vers la droite.

Un algorithme est constitué de séquences d'instructions rangés dans des blocs (voir plus bas). Les instructions d'une séquence sont exécutées de haut en bas, ligne par ligne dans l'ordre de leurs déclarations.

d) Exécution à la main

Un humain doit pouvoir dérouler à la main un algorithme. Considérez l'exemple 2 présenté au début de ce cours :

Exemple : Équation du premier degré A X + B = 0

```
1: Lire les coefficients A et B
2: Si A est non nul
    alors
        affecter à X la valeur - B / A
        afficher à l'écran la valeur de X
  Sinon
    Si B est nul
        alors
          Afficher à l'écran "tout réel est solution"
        Sinon
         Afficher à l'écran "pas de solution"
```

Prenez une feuille de papier. Dessinez trois rectangles pour représenter les 3 réceptacles des valeurs des variables A, B et X.

Marquez A devant le premier rectangle, B devant le second et X devant le dernier (c'est-à-dire leur identificateurs).

Exécutez l'algorithme instruction par instruction. Quand une instruction modifie la valeur d'une variable, effacez l'ancienne valeur contenue dans le rectangle associé puis écrivez à la place la nouvelle valeur.

Quand l'algorithme est terminé gommez le contenu des rectangles. Recommencez pour tester différents cas possibles.

L'état du programme correspond aux valeurs que prennent les variables entre deux instructions. Avant une instruction l'état est stable. Après, il est stable aussi. C'est l'exécution d'une instruction qui change l'état du programme.

Les simulations vous montreront différentes exécutions de programme écrit en C. Il est important de savoir exécuter un bout de code à la main, car cela aide à trouver des erreurs. Cependant, toutes les erreurs ne peuvent être trouvées ainsi.

e) Notion de bloc

Dans l'exemple numéro 2, équation du premier degré, la séquence d'instructions :

```
affecter à X la valeur de -B/A
afficher à l'écran la valeur de X
```

s'effectue si la condition « A est non nul » est vraie. Dans ce cas, on effectue d'abord la première puis la seconde instruction. Ces deux instructions font partie de ce que nous appelons un bloc.

Le bloc commence juste après le premier « alors » et se termine juste avant le premier « sinon ». Ce bloc s'effectue uniquement si la condition est vraie.

Cette notion de bloc est importante. Elle ne se réduit pas uniquement à un ensemble d'instructions devant s'exécuter en séquence. Nous le verrons plus tard.

A ce stade, l'important est de savoir qu'un bloc contient certains éléments d'algorithme (et par extension de programme), qu'il peut ou non être activé à l'exécution, qu'il a un début et une fin.

Il peut s'avérer utile de délimiter explicitement le début et la fin d'un bloc. Dans ce cas il suffit de réserver à cet usage par exemple 'début' et 'fin' comme mots que

l'on utilisera pour délimiter explicitement le début et la fin d'un bloc.

f) Indentation

Considérons l'exemple n°2. La présentation de cet algorithme n'est pas anodine. Il ne possède que deux instructions principales :

1: lire les coefficients A et B.

2: si A est non nul alors ....

Vous remarquerez que le 'l' de la première ligne et le 's' de la seconde sont alignés dans une même colonne verticale.

Les autres instructions, organisées en bloc, ne s'effectuent que si certaines conditions sont remplies.

Pour faire apparaître visuellement ces blocs et pour mettre en évidence ce qui déclenche leur activation nous avons procédé à des décalages.

Vous remarquerez que le début des instructions d'un même bloc sont alignés dans une même colonne. Ce procédé de présentation s'appelle l'indentation. Il n'y a pas de règles immuables pour indenter un algorithme.

Vous remarquerez cependant que cela peut grandement aider à mettre en évidence l'organisation des blocs, même s'ils sont imbriqués.

Attention

Il ne faut pas confondre bloc et indentation d'instructions. Ce sont deux notions différentes. Un bloc possède un début et une fin et d'autres éléments que des instructions.

Nous distinguons cependant deux catégories de langage informatique : les langages à indentation forte et les langages à indentation faible.

Dans les langages à indentation forte (ex:. Fortran, Python) l'indentation est obligatoire pour marquer les début et fin de chaque bloc, dans ce cas on n'a pas besoin de délimiteurs comme 'début' ou 'fin'.

Tout ce qui concerne un bloc est aligné dans une même colonne. Dans les langages à indentation faible (ex: C , Pascal) il faut signaler explicitement le début et la fin de chaque bloc à l'aide de délimiteurs spéciaux.

Pour le C nous avons déjà rencontré ces délimiteurs, il s'agit de '{' pour le début d'un bloc et du '}' pour la fin d'un bloc. Cependant pour des raisons évidentes de lisibilité il vaut mieux indenter son code C, mais vous pouvez le faire à votre convenance rien n'est imposé.

Vous remarquerez que dans l'algorithmique que nous utilisons les blocs sont mis en évidence par indentation.

g) Branchement sans condition

Il faut d'abord repérer l'instruction de l'algorithme à laquelle on veut se brancher à l'aide d'une étiquette.

Ensuite, quand on désire exécuter à nouveau cette instruction (et celles qui la suivent) il suffit d'utiliser l'instruction « allez à » en précisant l'étiquette.

En C, cela correspond à l'instruction « goto ».

Remarque

Les instructions de type :

- affectation (avec usage de l'opérateur d'affectation)
- branchement conditionnel (structure de sélection simple)

- branchement sans condition

constituent trois instructions de base des langages impératifs.

Attention

Nous avons introduit le branchement sans condition dans un but pédagogique.Les structures itératives (les boucles) que nous verrons plus loin les utilisent sans que le programmeur ait besoin de les écrire et de les manipuler explicitement.

Un programme correctement écrit n'a pas besoin de « goto ».

L'usage des « goto » est à proscrire, car il diminue la lisibilité des codes et c'est une source d'erreur.

6. Exemples

Objectifs

Nous traitons sur trois exemples les éléments d'algorithmiques introduits précédemment.

L'objectif du premier exemple est de mettre en œuvre la démarche méthodologique en montrant les inconvénients de différentes solutions.

Les deux autres ont pour but de montrer des exemples correctement traités.

Nous supposons que nous disposons des fonctions « lire », qui lit une donnée tapée au clavier par l'utilisateur et qui range sa valeur dans une variable, ainsi que de la fonction « afficher », qui écrit quelque chose à l'écran.

a) Calcul de la surface d'un cercle

On se propose d'écrire un algorithme permettant de calculer la surface d'un cercle, de rayon 12 et d'afficher le résultat de ce calcul. Après avoir analysé le problème, nous proposons plusieurs solutions dont nous examinons les inconvénients.

i - Analyse du problème

il faut disposer de la valeur du rayon R

calculer la surface : pi x R^2

afficher le résultat.

ii - Premier algorithme

afficher (3.14159 x carré(12))

Attention : Inconvénients

manque de souplesse. Cette solution est très mauvaise. En effet, pour calculer la surface d'un cercle ayant un rayon autre que 12, il faudra modifier le texte de l'algorithme pour remplacer la valeur 12 par la nouvelle valeur du rayon. Cette solution est beaucoup trop rigide. Le résultat est figé. Il est préférable de demander à l'utilisateur la valeur du rayon, puis d'effectuer le calcul. En outre, l'algorithme n'est pas explicite, carré(12) semble réaliser le calcul RxR si l'on se réfère à l'analyse, mais ce n'est pas explicitement dit.

iii - Deuxième solution

lire (Y)
afficher (3.14159 x carré(Y))

Attention : Inconvénients

En fait, Y n'est pas clairement définie. Le nom Y désigne une variable mais l'algorithme ne l'énonce pas. De plus, le nom de la variable est mal choisi, car peu explicite. L'opérateur binaire x correspond à la multiplication de deux opérandes (ex: a x b) ce n'est pas explicite non plus (même si cela semble évident pour vous). Il risque d'être confondu avec la lettre 'x'.

iv - Troisième solution

pi=3.14159
demander (entrez le rayon du cercle :)
lire (rayon)
afficher (La surface est pi * carré(rayon) pour un cercle de valeur rayon)

Attention : Inconvénients

l'algorithme commence à être plus clair mais il y a encore beaucoup d'éléments implicites.

La variable dont le nom est rayon n'est pas clairement définie, pi semble être une constante mais l'algorithme ne l'explicite nullement. Nous avons remplacé le signe 'x' de la multiplication par le signe '*', ce qui lève la confusion possible.

b) Calcul du salaire net

On désire calculer un salaire net à partir du salaire horaire brut, du nombre d'heures effectuées et du pourcentage de charges à retenir sur le salaire brut. Nous utiliserons désormais le signe '*' pour désigner l'opérateur de multiplication.

i - Données

SH : le salaire horaire, un réel, fourni par l'utilisateur
NH : le nombre d'heures, un réel, fourni par l'utilisateur
PR : le % de retenues non plafonnées, un réel, fourni par l'utilisateur

ii - Variables

SH, NH, PR de type réel pour les données
R de type réel pour le calcul des retenues
SN de type réel pour le salaire net
SB de type réel pour le salaire de base

iii - Instructions

```
afficher (entrez le salaire horaire)
lire (SH)
afficher ( entrez le nombre d'heure)
lire (NH)
afficher (entrez le % de retenue, ex 0.1 pour 10%)
lire (PR)
SB = SH * NH
R = SB * PR
SN = SB - R
afficher (le salaire net est : SN)
```

c) Calcul du PGCD

Le PGCD de deux entiers est leur plus grand diviseur commun.

Le principe adopté est l'algorithme d'Euclide que l'on peut formellement décrire ainsi :

La division entière se définit par A= (B * Q) + R avec A, B, Q, R entiers naturels. Nous définissons par le signe % l'opérateur binaire tel que A % B calcule la valeur de R. C'est le reste de la division entière. L'opérateur % défini le modulo mathématique.

si R= 0 alors le PGCD de A et B vaut B

si R≠ 0 alors le PGCD de A et B est le même que le PGCD de B et R

i - Données

A et B deux valeurs entières fournies par l'utilisateur

ii - Variables

A et B entiers

R entier pour le reste

le_pgcd entier pour le résultat

iii - Instructions

```
afficher (calcul du PGCD de deux entiers)
afficher (entrer le premier entier)
lire (A)
afficher (entrer le deuxième entier)
lire (B)
truc : R = A % B
    si R≠ 0
        alors
            A = B
            B = R
            aller à truc:
    sinon le_pgcd = B
afficher ( le PGCD de A et B est, le_pgcd)
```

Remarque

Remarquez que « truc: » est une étiquette. Nous avons un branchement sans condition.

Il existe une solution avec une boucle (voir structures itératives) qui est préférable. Il peut exister plusieurs algorithmes pour traiter un même problème.

Comme vous pouvez le constater, ces deux algorithmes sont plus précis. Ils ne sont cependant pas sans ambiguïtés.

Nous avons présenté un cadre pour concevoir des algorithmes simples. Nous avons introduit des concepts importants.

Cependant nous pouvons nous permettre de prendre certaines libertés dans la rédaction d'un algorithme quand celui-ci est très simple et qu'il y a peu d'ambiguïtés possibles.

Par exemple dans des cas très simples on peut omettre de définir explicitement les variables, ou encore ne pas utiliser de délimiteurs de début et fin de bloc. Par contre si l'algorithme est complexe nous vous invitons à utiliser ce qui vous a été présenté.

7. Langage

Dans le cadre de ce cours, nous utiliserons le langage C, très utilisé dans le monde de l'informatique. Pour construire un programme en C, il faut écrire un code source en respectant la grammaire du langage C

a) Grammaires et langages

Dans les langages des humains, la grammaire (syntaxe) permet de construire des phrases, la sémantique donne un sens à ces phrases.

C'est la même chose pour les langages informatiques. La syntaxe des langages de programmation est très rigide et dispose d'un vocabulaire limité.

Le programmeur doit connaître les éléments de syntaxe qui lui sont nécessaires pour programmer. Une grammaire peut se décrire à l'aide du quadruplet :

(VT, VN, Symbole distingué, règles)

- VT : vocabulaire terminal
- VN : vocabulaire non terminal
- Symbole distingué : un élément du vocabulaire non terminal qui sert de point de départ pour l'analyse syntaxique.
- Règles : ensemble des règles à appliquer pour déterminer si ce que l'on soumet à l'analyse est correct.

Complément

La syntaxe du langage C peut se décrire à l'aide d'un ensemble de diagrammes de CONWAY. Il s'agit d'une représentation graphique de la grammaire.

Ils permettent de vérifier si la syntaxe que l'on utilise est correcte pour le compilateur C. Un diagramme porte un nom.

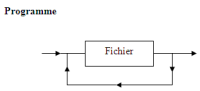
Il comporte des flèches qui relient des boites carrées et des boites arrondies (ovales ou cercles) dans lesquelles sont inscrits des noms ou des symboles. Les boites arrondies (ovales ou cercles) correspondent à un mot du vocabulaire terminal (ou symbole terminal).

Si la boite arrondie est grisée alors le nom inscrit correspond à des symboles terminaux évidents que l'on ne détaille pas.

Des mots du langage C comme « main » ou « void » sont des mots du vocabulaire terminal. Ces mots sont des mots réservés du langage C qui ne peuvent pas être utilisés par le programmeur pour servir d'identificateur par exemple.

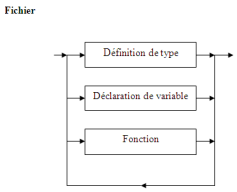
Les boites carrées correspondent au VN, vocabulaire non terminal. Le mot que contient une telle boite correspond au nom d'un autre diagramme de CONWAY qui existe forcément.

Le symbole distingué en C correspond au diagramme de programme.



programme

Pour vérifier si la syntaxe d'un programme C est correcte il faut partir de là. Et soumettre à l'analyse le texte du programme écrit. Un programme est donc un fichier. Il faut chercher le diagramme de fichier :



Fichier

et continuer de suivre les diagrammes jusqu'à obtenir une analyse complète sans bloquer dans un seul diagramme.

Par exemple, on peut se demander si notre premier programme C (vu au chapitre 1 et que nous rappelons ci-dessous) est un fichier.

```
void main ()
{
printf ("Ceci est votre premier programme\n");
}
```

Sachant qu'il ne comporte ni définition de type ni déclaration de variable, nous devons déterminer s'il comporte une fonction.

Il faudrait donc soumettre le texte au diagramme de CONWAY de "fonction" pour savoir si la fonction 'main' est syntaxiquement correcte.

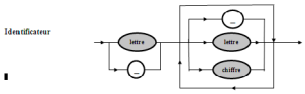
La réponse est oui.

C'est donc un fichier. Notre premier programme en C est donc un fichier, c'est donc bien un programme.

L'analyse syntaxique est un succès quand on revient au symbole distingué et que l'on sort du diagramme de CONWAY associé sans avoir rencontré d'erreur.

Ce mécanisme, dit de dérivation, correspond à l'application graphique des règles de grammaire.

Autre exemple : le diagramme de CONWAY d'identificateur est :



Identificateur

Ce qui correspond exactement à ce que nous avons défini précédemment sous forme de phrases.

Lettre et chiffre correspondent à des symboles terminaux évidents que l'on ne détaillera pas.

Le compilateur C vérifie si la syntaxe de votre programme est correcte. Si la compilation échoue, elle ne l'est pas. Les diagrammes de CONWAY du langage C sont fournis en document annexe.

Ils pourront vous aider à trouver vos erreurs.

Remarque

Différents éléments de base de la syntaxe du C vous seront donnés dans les autres chapitres ainsi que leurs sens (sémantique).

Nous ne serons pas exhaustifs dans ce cours introductif, nous ne pourrons pas explorer toutes les possibilités du C que l'on peut retrouver dans les diagrammes.

N. B : Les directives de pré-compilation ne sont pas analysables avec les diagrammes fournis. Ils ne concernent que le compilateur.

8. Exercices de Révision (QCM)

Exercice 1

1. Un algorithme est clair.

- ○ Vrai
- ○ Faux

Exercice 2

2. Un algorithme est général.

- ○ Vrai
- ○ Faux

Exercice 3

3. Un algorithme est directement interprétable par l'ordinateur.

- ○ Vrai
- ○ Faux

Exercice 4

4. Un algorithme est accessible à tous les acteurs du domaine.

- ○ Vrai
- ○ Faux

Exercice 5

5. Un algorithme est dépendant du langage utilisé.

- ○ Vrai
- ○ Faux

Exercice 6

6. Un algorithme est unique.

- ○ Vrai
- ○ Faux

Exercice 7

7. Un algorithme est facile à comprendre.

- ○ Vrai
- ○ Faux

Exercice 8

8. Peut-on redéfinir la valeur d'une constante au cours d'un programme ?

○	Oui
○	Non

Exercice 9

9. Peut-on redéfinir la valeur d'une variable au cours d'un programme ?

○	Oui
○	Non

C. Structure d'un programme

1. Le langage C

a) "C" est un langage typé et structuré

Définition

C est un langage typé, ce qui signifie que toutes les variables doivent avoir un type. Le type doit être explicitement défini, afin d'enlever toute ambiguïté.

C est un langage structuré car l'organisation d'un programme se fait en "blocs d'instructions" emboîtés. Pour cela, on utilise des identificateurs afin de spécifier les blocs (les fonctions) et des indentations pour mieux visualiser l'architecture du programme.

b) Structure globale d'un programme C

i - Fichier source

Définition

Un fichier source écrit en langage C est composé de :

- définition de types.
- déclaration de variables.
- déclaration de fonctions.

Ceci se réalise dans un ordre quelconque du moment que la règle : « ce qui est utilisé doit être explicitement déclaré avant » est respectée.

Pour pouvoir s'exécuter, un programme C doit contenir la déclaration d'une fonction spéciale appelée main qui sera le point d'entrée de l'exécution. Il s'agit de la fonction invoquée au démarrage de l'exécution. Toutes les autres fonctions sont en fait des blocs d'instructions auxquels le programmeur donne un identificateur (nom). Elles peuvent être ensuite utilisées (invoquées) dans la fonction principale main.

Une déclaration de fonction se décompose en deux parties :

- L'en-tête de la fonction : Il est composé du type de valeur qu'elle retourne, de son nom (un identificateur) et d'une description des paramètres qu'elle recevra quand elle sera appelée (ce sont les paramètres formels).
- Les paramètres formels sont définis entre une parenthèse ouvrante «) » et une parenthèse fermante «) ».

Exemple : Exemple de programme

Nous présentons un programme qui donne la moyenne de N nombres entrés au clavier. L'utilisateur doit préciser le nombre N de données qu'il va taper.

A ce stade du cours, il n'est pas nécessaire de comprendre le contenu de ce programme. Il suffit simplement d'y distinguer l'architecture globale décrite précédemment (déclarations de variables, blocs, indentations).

```
#include <stdio.h>
void moyenne()
//en-tête de la fonction moyenne, elle n'attend pas de
paramètres
// début du bloc de la fonction
{
// déclaration de variables
float donnee, somme, moyenne; int i, n;
// La séquence d'instructions de la fonction moyenne
// Inutile de comprendre le contenu. L'essentiel est
d'identifier la structure
printf("entrer le nombre de donnees");
scanf("%d",&n);
if (n > 0)
{
somme = 0;
for (i=1; i < N; i++)
{ scanf("%d",&donnee);
somme = somme + donnee;
}
moyenne = somme / n;
printf("moyenne = %d",moyenne);
}
else printf("pas de donnees");
}
// fin du bloc de la fonction
void main ()
// en-tête de la fonction main, elle n'attend pas de
paramètres
{
moyenne () ;
}
```

Deux fonctions sont définies, moyenne et main. La fonction principale main appelle la fonction moyenne définie avant. La règle est bien respectée.

Nous allons dans un premier temps travailler uniquement avec la fonction principale main. L'en-tête de cette fonction est :

void main()

La fonction main ne renvoie rien et n'attend aucun paramètre.

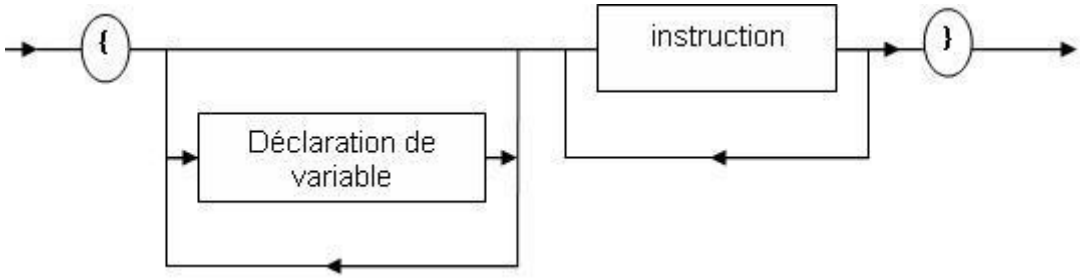
Nous allons maintenant voir comment on construit un bloc d'instructions.

ii - Bloc d'instructions

Définition

Un bloc commence par le délimiteur « {« . Viennent ensuite éventuellement des déclarations de variables. Puis nous avons une séquence d'instructions. Le délimiteur « ; » sépare les instructions. Enfin, le délimiteur « } » marque la fin du bloc.

Cette description correspond au diagramme de Conway suivant :



bloc

Remarque

Nous n'avons pas encore déclaré de variables. Par contre nous connaissons un type d'instruction : l'appel de fonction puisque nous avons déjà utilisé la fonction "printf" dans notre premier programme. L'exemple de la fonction moyenne qui suit vous fait découvrir d'autres instructions que nous allons explorer par la suite.

2. Directives au préprocesseur

Une action de pré-compilation est effectuée avant la compilation proprement dite. Le pré-processeur est le programme chargé de la pré-compilation. Durant la pré-compilation, il va supprimer, ajouter ou remplacer certaines chaînes de texte dans le fichier source selon les directives qu'il va interpréter. Son résultat est un fichier source modifié qui sera ensuite compilé.

Une directive au pré-processeur commence toujours par le caractère « # ». Elle se termine par défaut au passage à la ligne dans le code source, sauf si la ligne se termine par le caractère « \ » (lire backslash) auquel cas elle continue sur la ligne suivante.

Il existe de nombreuses directives. Dans ce cours introductif nous n'en verrons que deux : « define » et « include ». Ces directives peuvent être placées n'importe où dans le code source. Elles deviennent actives à partir de l'emplacement où elles sont définies. Nous préconisons de les placer au début du code source.

a) #define

Deux type d'utilisations sont possibles pour le #définе :

Syntaxe : Première utilisation : substitution de symboles

#define chaîne_1 chaîne_2

- La chaîne_1 est remplacée par la chaîne_2 à chaque fois qu'elle est rencontrée dans le code source (avant sa compilation).
- La chaîne_2 commence au premier caractère non blanc après chaîne_1, et se termine sur le premier caractère non blanc rencontré à droite.
- Généralement et pour plus de clarté la la chaîne_1 est écrite en majuscules mais ce n'est pas obligatoire.

Fondamental

Deux type d'utilisations sont possibles pour le #définе :

i - Première utilisation : substitution de symboles

Syntaxe

#define chaîne_1 chaîne_2

- La chaîne_1 est remplacée par la chaîne_2 à chaque fois qu'elle est rencontrée dans le code source (avant sa compilation).
- La chaîne_2 commence au premier caractère non blanc après chaîne_1, et se termine sur le premier caractère non blanc rencontré à droite.
- Généralement et pour plus de clarté la la chaîne_1 est écrite en majuscules mais ce n'est pas obligatoire.

Exemple

Le #define était la seule façon de définir des constantes dans les versions initiales des compilateurs C. Ainsi, par exemple avec les déclarations suivantes :

#define NMAX 100

#define PI 3.1415926

à chaque fois que la chaîne NMAX sera rencontrée, lors de la pré-compilation, elle sera remplacée par la chaîne 100. De même, la chaîne PI sera remplacée par 3.14159265.

Exemple

La substitution de symboles permet aussi de modifier l'aspect d'un programme source. Ce qui peut aider le programmeur à clarifier son code. Les substitutions suivantes :

#define begin {

#define end }

Sur le programme :

```
void main ()
begin
printf ("\nBonjour");
end
```

vont donner :

```
void main ()
{
printf ("\nBonjour");
}
```

1. Deuxième utilisation : Macro_instruction

Définition

Une définition de macro_instruction permet une substitution de texte paramétrée par des arguments.

#define chaîne1(a1,a2,....) chaîne2

- Il n'y a pas d'espace entre la chaîne1 et la '('
- Lors de la pré-compilation, dans le texte du programme, le symbole (x,y,...) est remplacé par la chaîne2 dans laquelle toutes les occurrences des arguments formels a1, a2, sont remplacées par les arguments effectifs x, y, ...

Exemple

#define carre(Y) (Y)*(Y)

#define mauvais_carre(Y) Y*Y
si l'on écrit dans le code source :
carre(3) on obtiendra (3) * (3) après substitution
carre(k) on obtiendra (k) * (k) après substitution
carre(k+1) on obtiendra (k+1) * (k+1) après substitution
après la précompilation.
Par contre si l'on écrit dans le code source :
mauvais_carre(3) on obtiendra 3 * 3 après substitution
mauvais_carre(k) on obtiendra k * k après substitution
mauvais_carre(k+1) on obtiendra k+1 * k+1 après substitution
après la précompilation.
L'expression k+1 * k+1 vaut en fait 2 * k+1, ce qui ne correspond pas à ce qui est souhaité. Si l'on omet les parenthèses, des erreurs difficiles à détecter peuvent être générées.

b) #include

Fondamental

Généralement les fichiers à inclure ont un nom qui se termine par .h (header) mais ce n'est pas obligatoire. Exemple #include <stdio.h>.

#include <stdio.h> va chercher le fichier stdio.h dans un répertoire spécial contenant les fichiers header des bibliothèques standards et incorpore le code trouvé dans le fichier source.

#include "/user/monHeader.h" va chercher le fichier monHeader.h dans le répertoire /user/ et l'incorpore dans le fichier source. Comme nous l'avons expliqué, cela permet d'inclure dans le fichier les définitions de fonctions de bibliothèque.

Remarque

Dans les environnements de développement en C, il est possible d'obtenir le fichier résultat de la pré-compilation. Par exemple, sous unix la commande : « cc -E toto.c > toto.i » va écrire le fichier texte toto.i, résultat de la pré-compilation du fichier source toto.c. Ce qui permet de regarder le fichier intermédiaire généré par le pré-compilateur. Précisons que ce fichier intermédiaire est automatiquement détruit (option par défaut).

3. Octet et adresse

Les ordinateurs travaillent en binaire sur des digits (chiffres binaires) qui valent soit 0 soit 1. Les digits (bits en anglais) sont regroupés par 8, ils constituent alors un octet (byte en anglais). La mémoire centrale d'un ordinateur peut être simplement vue comme un long ruban d'octets.

On accède à chaque octet en connaissant son numéro d'ordre au sein du ruban d'octet : c'est son adresse mémoire.

Connaissant une adresse, on peut lire son contenu ou écrire une nouvelle valeur pour l'octet. Les micro-processeurs les plus courants actuellement sont dits à architecture 32 bits. Les adresses mémoires sont gérées sur 4 octets (4 x 8 = 32 bits).

Une adresse mémoire, c'est-à-dire le numéro d'ordre d'un octet dans le ruban, prend ainsi sa valeur dans l'intervalle [0, 4 294 967 295]. En informatique, on commence à compter à partir de 0. Il y a donc potentiellement 2^32 = 4 294 967 296 octets adressables par le micro-processeur.

En résumé, une architecture 32 bits peut gérer au plus 4 gigaoctets (2^30 octets = 1 Gigaoctet) de mémoire centrale. Les ordinateurs courants sont vendus avec entre 1Go et et 4Go, et chaque octet possède une adresse entre 0 et (2^32)-1. La valeur de cette adresse peut se représenter et se manipuler dans un paquet de 4 octets.

4. Données et adresses

Les données (constantes et valeurs de variables) que manipule un programme sont stockées dans des octets et par extension par groupe d'octets contigus, cela correspond au couple (identificateur, réceptacle).

Par exemple, un entier en C est une donnée stockée sur 2 ou 4 octets. Supposons qu'un entier est codé sur 2 octets. Quand on travaille sur un entier, on sait que l'on travaille sur un groupe de 2 octets rangés dans la mémoire centrale de la machine et que ces 2 octets sont contigus.

Chaque octet possède une adresse mémoire. Puisque la taille est connue (c'est 2), pour manipuler l'entier il suffit de connaître l'adresse de son premier octet. Le suivant sera automatiquement utilisé par le microprocesseur pour le effectuer le travail nécessaire.

Toute donnée doit être déclarée avant usage. Le compilateur se charge de vérifier la compatibilité des types des données impliquées dans une expression. Par exemple quand on écrit x+3, le compilateur vérifie que le type de l'identificateur x est compatible avec la valeur 3 (entière) par rapport à l'opération `+'.

Nous ne détaillerons pas la façon dont est codé un entier sur 2 octets. Ce n'est pas notre propos ici. Par contre, il est important de savoir, quand on programme en C, qu'une donnée est stockée sur plusieurs octets (sa taille), contigus au sein du ruban d'octets, et que, connaissant sa taille, il suffit de connaître l'adresse du premier pour avoir toute l'information.

Cette notion d'adresse d'une donnée est très importante car le C utilise abondamment les pointeurs qui sont des informations de type adresse d'un octet.

Remarque

Les architectures 64 bits commencent à se démocratiser. Potentiellement, elles peuvent gérer 2^64 octets de mémoire centrale. Soit 18446744073709551616 octets. Elles ont été conçues pour « casser » la barrière de 4Go.

Sur ces machines on peut avoir plus de 4Go de mémoire centrale. Dans ce cas une adresse sera manipulée sous la forme d'une valeur que l'on peut coder sur 8 octets.

Conceptuellement, cela ne change rien, il suffit uniquement de savoir si l'architecture est 32 ou 64.

5. Les types simples

Les données (variables ou constantes) manipulées en langage C sont typées. Il faut préciser le type de la donnée, ce qui permet de connaître son occupation mémoire (le nombre d'octets), sa représentation en mémoire centrale et la nature des opérations que l'on peut effectuer avec.

Suivent ci-après les types simples qui sont pré-définis au sein du langage C, il s'agit :

- Des nombres : entiers (int) ou réels, c'est-à-dire à virgule (float).
- Des caractères (char)
- Des pointeurs qui stockent l'adresse d'une donnée, ils « pointent » vers une

autre donnée.

a) Les entiers

Les entiers sont signés par défaut, cela signifie qu'ils comportent un signe. Le type entier de base en C est dénommé int. Il correspond généralement à un réceptacle de 4 octets. Dans ce cas les valeurs possibles des littéraux vont de – 2 147 483 648 à + 2 147 483 647. A partir de ce type, les types dérivés suivants existent :

- unsigned int
- short int
- unsigned short int
- long int
- unsigned long int

Le qualificateur « unsigned » combiné avec les qualificateurs « short » et « long » permet d'obtenir des variations. « unsigned » signifie non signé. Si un int est sur 4 octets alors un unsigned int l'est aussi. Dans ce cas ses littéraux varieront de 0 à + 4 294 967 295. Il faut faire attention, la taille d'un int peut dépendre du microprocesseur. Dans la génération actuelle de microprocesseur, sa taille est de 4 octets.

b) Les réels

Les nombres réels sont des nombres à virgule flottante, c'est-à-dire des nombres dans lesquels la position de la virgule n'est pas fixe. Une partie des bits d'un réel sert à coder l'exposant, le reste des bits permettent de coder le nombre sans virgule, la mantisse. Nous ne rentrerons pas dans le détail du codage d'un nombre réel au sein de la machine. En C on les appelle les flottants.

Il existe deux types de réels :

- float : littéraux de 3.4 10-38 à 3.4 10+38
- double : littéraux de -1.7 10-308 à 1.7 10+308

Dans le code source nous écrirons par exemple : 3.4e-38 sur une seule ligne pour un littéral réel.

Le float correspond à une simple précision, le double à une double précision. Un float correspond à un réceptacle de 4 octets. Le réceptacle est de 8 octets pour un double.

c) Les caractères

Le type char (provenant de l'anglais character) permet de stocker la valeur ASCII d'un caractère, c'est-à-dire un nombre entier codé sur un unique octet.

Puisque par défaut les nombres sont signés. Une donnée de type char est donc signée. Cela ne signifie bien sûr pas que la lettre possède un signe mais tout simplement, que dans la mémoire, la valeur codant le caractère peut être négative.

Si l'on désire, par exemple, stocker la lettre B (son code ASCII est 66), on pourra définir cette donnée soit par le nombre 66, soit en notant 'B' où les apostrophes simples permettent d'encadrer un littéral de type caractère.

d) Le type rien (void)

Le type void est le type de résultat d'une fonction qui ne produit aucun résultat. Nous l'avons déjà rencontré avec la fonction principale main que nous avons utilisée et qui ne retournait aucune valeur.

Il est impossible de déclarer directement une variable ou une constante de type void.

e) Le type "pointeur vers"

Comme nous l'avons précisé, toute donnée du programme possède un type. Une déclaration génère un réceptacle de un ou plusieurs octets. Si l'on connaît la taille du réceptacle et l'adresse en mémoire de son premier octet, nous pouvons accéder à son contenu, c'est-à-dire à la valeur qui y est contenue.

En C, on appelle pointeur une information qui correspond à l'adresse mémoire d'un octet. Cependant, si l'on n'en connaît pas la taille cela ne suffit pas. Par conséquent, on doit définir le type pointeur relativement à ce qui est pointé.

Concrètement, on ajoute après l'identificateur de type simple le signe '*' pour déclarer un type pointeur vers.

Ce qui implique que, par exemple, une valeur de type « char * » sera codée sur 4 octets (c'est une adresse). Cette valeur correspond à l'emplacement d'un octet dans la mémoire centrale qui correspond à un caractère.

Le langage C utilise abondamment les pointeurs. C'est une des difficultés à maîtriser.

Sur une architecture 32 bits, les littéraux de type adresse (valeur des pointeurs) sont compris dans l'intervalle [0, 4294967295].

Le type « void * » existe. Il ne signifie pas « pointeur vers rien » mais pointeur vers un octet quelconque de la mémoire centrale. Ce n'est pas la même chose que « char * ».

f) La convention logique du C

Il n'existe pas de type booléen en C. Il existe par contre une convention pour distinguer ce qui est faux de ce qui est vrai.

Considérons un entier (int) sur 4 octets. Si cet entier vaut 0 alors tous les digits de ses 4 octets sont à 0. Dans ce cas on convient que ce dernier vaut faux. Toute autre valeur correspond à vrai car au moins un digit vaut 1.

Vous devez l'admettre. De même pour un float, 0.0 correspond à faux et toute autre valeur à vrai.

En résumé, 0 correspond à faux, une autre valeur correspond à vrai.

g) Tableau récapitulatif des types

Type de donnée	Signification	Taille (en octets)	Plage des littéraux
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32 768 à 32 767
unsigned short int	Entier court non signé	2	0 à 65 535
int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	Flottant (réel)	4	3.4×10^{-38} à 3.4×10^{-38}
double	Flottant double	8	1.7×10^{-308} à 1.7×10^{308}

Tableau 1 tableau_types

6. Les déclarations

a) Variables

Déclarer une variable, c'est créer le couple (identificateur de variable, réceptacle) qui définit l'ensemble des valeurs qu'elle peut prendre (les littéraux). Toutes les variables utilisées dans un programme doivent être déclarées avant usage et correctement initialisées.

Syntaxe : Syntaxe de la déclaration

type_de_variable id [= valeur] [, id [= valeur]];

Une telle déclaration se fait en faisant suivre le nom du type par la liste des identificateurs des variables (qui correspondent à ce type).

Exemple : Exemple

int I; /* déclaration de la variable I de type int */

int i,j; /* déclaration de deux variables i et j de type int, le délimiteur ',' permet de déclarer une liste de variable de même type */

short int k; /* déclaration de la variable k de type short int */

float f; /* déclaration de la variable f de type float */

double d1,d2; /* déclaration de deux variables d1 et d2 de type double, vous remarquerez le signe ',' qui sépare les deux identificateur d1 et d2 */

char ch; /* déclaration de la variable i de type caractère */

int * p_sur_int; /* déclaration de la variable p_sur_int de type pointeur sur entier */ char * vers_char; /* déclaration de la variable vers_char de type pointeur sur char */

Il est possible de donner une valeur initiale aux variables ainsi déclarées. On dit alors qu'elle est initialisée à la déclaration. Ce n'est pas obligatoire.

Exemple

int i = 54; /* déclaration de la variable i de type int et affectation de la valeur 54*/

int i = 34, j = 12;

char ch='o';

float y = 1.5, z = 2.0;

Vous remarquerez que le signe « ; » indique la fin de la déclaration de variables d'un même type.

De même, vous remarquerez que le caractère « , » est utilisé pour séparer une liste d'identificateurs de variables d'un même type lors de la déclaration.

On peut déclarer des variables dans le programme auquel cas elles sont globale ou au début d'un bloc auquel cas elles sont locales au bloc.

b) Constantes

Méthode

L'utilisation de constantes en programmation est vivement conseillée. Elles permettent :

- Une notation plus simple et plus lisible.
  Exemple : PI à la place de 3.141592653

- La possibilité de modifier simplement la valeur spécifiée dans la déclaration au lieu d'en rechercher les occurrences, puis de modifier chacune dans le programme complet.

Pour cela, nous utiliserons la directive « #define » donnée au pré-compilateur. Cependant, pour information, dans les normes récentes du C on peut déclarer une constante comme dans les exemples qui suivent :

const int a=10 ;

const float b=7.77 ;

ce qui définit les constantes a et b.

7. La fonction "sizeof" du compilateur

Nous introduisons la fonction « sizeof » pour illustrer les notions de taille de réceptacle pour les données. « sizeof » attend comme argument un type et il renvoie le nombre d'octets nécessaire pour le coder dans la mémoire, c'est-à-dire la taille d'un réceptacle de ce type.

On peut remarquer ainsi que la taille d'une donnée de type « pointeur vers » est identique quel que soit le type simple pointé.

Le programme C suivant affiche la taille de chaque type simple. Copiez-le et compilez.

```
#include <stdio.h> void main()
{
printf("\n %d octets pour variable de type char
",sizeof(char));
printf("\n %d octets pour variable de type unsigned char
",sizeof(unsigned char));
printf("\n %d octets pour variable de type short_int
",sizeof(short int));
printf("\n %d octets pour variable de type int
",sizeof(int));
printf("\n %d octets pour variable de type long int
",sizeof(long int));
printf("\n %d octets pour variable de type float
",sizeof(float));
printf("\n %d octets pour variable de type double
",sizeof(double));
printf("\n %d octets pour adresse vers variable de type
char ",sizeof(char * ));
printf("\n %d octets pour adresse vers variable de type
unsigned char ",sizeof(unsigned char * ));
printf("\n %d octets pour adresse vers variable de type
short_int ",sizeof(short int * ));
printf("\n %d octets pour adresse vers variable de type int
",sizeof(int *));
printf("\n %d octets pour adresse vers variable de type
long int ",sizeof(long int *));
printf("\n %d octets pour adresse vers variable de type
float ",sizeof(float *));
printf("\n %d octets pour adresse vers variable de type
double ",sizeof(double *));
printf("\n");
}
```

Un exemple d'affichage obtenu après exécution du code

```
1 octets pour variable de type char
```

```
1 octets pour variable de type unsigned char
2 octets pour variable de type short_int
4 octets pour variable de type int
4 octets pour variable de type long int
4 octets pour variable de type float
8 octets pour variable de type double
4 octets pour adresse vers variable de type char
4 octets pour adresse vers variable de type unsigned char
4 octets pour adresse vers variable de type short_int
4 octets pour adresse vers variable de type int
4 octets pour adresse vers variable de type long int
4 octets pour adresse vers variable de type float
4 octets pour adresse vers variable de type double
```

A l'exécution vous obtiendrez les tailles des types simples correspondant à votre ordinateur. Vous remarquerez qu'un pointeur est codé sur le même nombre d'octets quelque soit le type de données vers lequel il pointe.

Vous remarquerez que si vous demandez la taille d'une donnée de type « void * », vous obtenez 4.

C'est la taille d'une donnée de type adresse mémoire (pointeur vers un octet).

8. Les constituants élémentaires du langage C

a) L'alphabet

Définition

Un code source écrit en langage C utilise la plupart des signes de la table des codes ASCII non étendue, c'est-à-dire les caractères correspondant aux codes compris dans l'intervalle [0,127].

b) Les séparateurs

Définition

Les séparateurs servent à organiser la présentation du code source. Ils permettent l'indentation (espaces, plusieurs espaces, tabulations, saut de ligne). Ils font parti de l'alphabet et sont utiles pour délimiter certaines parties du code source. Par exemple, on écrit :

int x;

pour déclarer la variable de type entier dont l'identificateur est x.

Au moins un espace est nécessaire entre « int » et « x ».

c) Les commentaires

Définition

Les commentaires ont déjà été introduits : deux types de balises sont utilisés pour les indiquer. //suivi d'un commentaire sur une unique ligne.

/* est la balise de début de commentaire, on peut mettre ce que l'on veut sur autant de ligne que l'on veut jusqu'à la balise qui le termine */

d) Les identificateurs

Définition

Les identificateurs permettent de donner un nom aux différents éléments du programme que l'on va manipuler. Par exemple des constantes, des variables ou

des fonctions.

La façon de construire un identificateur syntaxiquement correct a été présentée, sous forme de texte et grâce à un diagramme de Conway.

e) Les mots réservés du langage

Définition

Les mots réservés du langage sont nécessaires à la sémantique. Ce sont des :

- spécificateurs de type : int, char, ..., struct, ...
- spécificateurs de classe d'allocation : auto, exter, typedef, ...
- opérateurs symboliques : if, else, while, switch, ...
- étiquette : default, ...

Vous ne pouvez pas utiliser ces mots comme identificateurs de variables, de constantes ou de fonctions. En voici la liste :

```
auto
break
case
char
continue
const
default
do
double
else
extern
float
for
goto
if
int
long
register
return
short
sizeof
static
struct
switch
typedef
union
unsigned
while
```

f) Les opérateurs

Définition

Ils sont représentés par 1 ou 2 caractères juxtaposés. Il existe 3 classes d'opérateurs :

- Les opérateurs unaires : ils sont associés à un identificateur soit de façon préfixe (juste devant) soit de façon postfixe (juste derrière), cela dépend de leur fonction. Exemple -, *, &, +, ++, --, ^, !
- Les opérateurs binaires : ils mettent en relation deux expressions, Exemple ==, +, *, /, "
- Un opérateur ternaire : il n'y en a qu'un le « ? » : exemple (2<3) ?1 :0 ou encore (4<3) ?1 :0 Ils seront détaillés dans le chapitre suivant.

Ils seront détaillés dans le chapitre suivant.

g) Les délimiteurs

Définition

Comme leur nom l'indique ils servent à délimiter des éléments du programme. Il en existe 5 :

- ; qui termine une déclaration de variable ou une instruction.
- , qui sépare deux éléments dans une liste.
- (et) qui encadrent une liste de paramètres.
- [et] qui encadrent une dimension ou l'indice d'un tableau.
- { et } qui encadrent un bloc ou une liste de valeurs d'initialisation.

9. Exercices de Révision (QCM)

Exercice 1

1. Quelles sont les en-têtes de déclaration de la fonction « main » que vous avez déjà rencontré.

- ○ void Main (id1, id2, ..., idn)
- ○ void main ()
- ○ void main (id1, id2, ..., idn)

Exercice 2

2. Le type "char" correspond à :

- ○ Un entier codé sur un unique octet
- ○ Un réel.
- ○ Ni l'un ni l'autre.

Exercice 3

3. Le type "unsigned int" correspond à :

- ○ Un entier.
- ○ Un réel.
- ○ Ni l'un ni l'autre.

Exercice 4

4. Le type "Excel" correspond à :

- ○ Un entier.
- ○ Un réel.
- ○ Ni l'un ni l'autre.

Exercice 5

5. Le type "Word" correspond à :

- ○ Un entier.
- ○ Un réel.
- ○ Ni l'un ni l'autre.

Exercice 6

6. Le type "extended" correspond à :

- ○ Un entier.
- ○ Un réel.
- ○ Ni l'un ni l'autre.

Exercice 7

7. Le type "int" correspond à :

- ○ Un entier.
- ○ Un réel.
- ○ Ni l'un ni l'autre.

Exercice 8

8. Le type "Sentence" correspond à :

- ○ Un entier.
- ○ Un réel.
- ○ Ni l'un ni l'autre.

Exercice 9

9. Le type "double" correspond à :

- ○ Un entier.
- ○ Un réel.
- ○ Ni l'un ni l'autre.

Exercice 10

10. Le type "dual" correspond à :

- ○ Un entier.
- ○ Un réel.
- ○ Ni l'un ni l'autre.

Exercice 11

11. On est obligé d'affecter une valeur à une variable lors de sa déclaration.

○	Vrai
○	Faux

Exercice 12

12. Où peut-on déclarer une variable ?

☐	Dans la fonction principale main
☐	Avant la fonction principale main
☐	Avant une directive au précompilateur "#include"

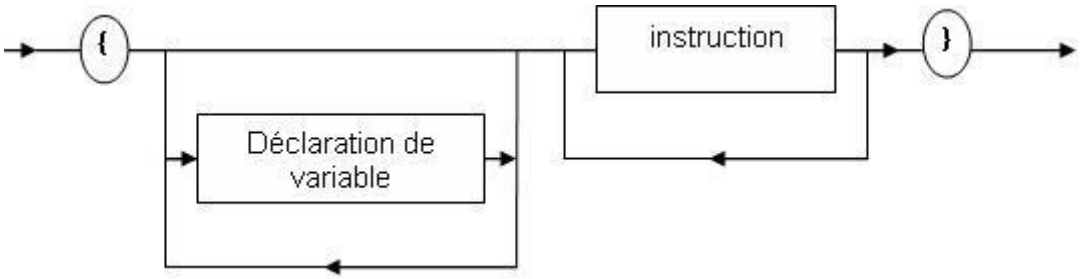
D. Premiers pas en C

1. Instruction et expression

Une instruction correspond à :

- une instruction simple qui se termine par le délimiteur point-virgule ';'
- un bloc qui commence par le délimiteur '{' et se termine par le délimiteur '}' Le bloc correspond à ce que nous appelons une instruction composée.

Nous rappelons le diagramme de Conway de bloc :



bloc

Les blocs d'instructions permettent de regrouper, dans un même bloc, une séquence d'instructions simples qui seront exécutées l'une après l'autre. Un bloc est vu comme une unique instruction, bien qu'il puisse comporter des déclarations de variables et une séquence d'instructions. Il est possible d'imbriquer un bloc à l'intérieur d'un autre bloc puisqu'un bloc est une instruction.

Nous rappelons qu'il faut utiliser l'indentation pour mettre visuellement en évidence l'organisation des instructions. Nous verrons par la suite dans quels cas il sera utile de faire des blocs d'instructions.

Nous allons voir comment construire des instructions simples.

a) Instruction d'affectation simple

Définition

L'instruction d'affectation simple utilise l'opérateur d'affectation simple qui est le '=' en C. C'est ce que l'on appelle un opérateur binaire car il comporte à sa gauche un premier opérande et à sa droite un deuxième opérande.

Nous introduisons l'affectation simple à l'aide du diagramme de Conway suivant :

instruction

Fondamental

A ce niveau de connaissances, nous dirons juste que :

- l'expression est évaluée (on fait un calcul)
- puis la valeur calculée est affectée (rangement) dans la variable (identificateur)

Attention

Les deux opérandes doivent avoir des types compatibles (la valeur calculée lors de l'évaluation de l'expression à droite du signe '=' doit être compatible avec le type de la variable affectée).

Exemple : Exemple de programme avec affectation simple :

```
#include <stdio.h>
int main()
{
int x , y , resultat;
// declarations de 3 variables de types entiers dont les
identificateurs sont
// respectivement x, y et resultat.
x = 3; // ici expression est le litteral constant 3
y = 7; // ici expression est le litteral constant 7
resultat = x + y; // ici expression est x + y
// vous remarquerez que toutes les instructions simples se
terminent bien par ';'.
// vous remarquerez aussi que x, y et resultat sont des
identificateurs de variables
// qui se situent a gauche de l'operateur binaire
d'affectation simple '='
}
```

b) Expressions et évaluation des expressions

Définition

Une expression représente une donnée qui possède une valeur. On parle d'évaluation d'une expression.

Une expression peut être simplement un identificateur de constante, un identificateur de variable, ou un littéral constant. Elle peut être également un ensemble d'identificateurs de constantes, de variables, ou des littéraux constants reliés par des **opérateurs**.

Ces **opérateurs** peuvent être des **opérateurs binaires** qui nécessitent deux opérandes (comme le +) ou des **opérateurs unaires** qui travaillent sur un unique opérande.

Exemple : Exemple d'expressions

```
20.6 // un littéral constant de type float
3 + 5 // deux littéraux constants de type int reliés par
l'opérateur d'addition +
'b' // un littéral constant de type char truc + machin
// deux identificateurs (constantes ou variables) reliés
par l'operateur '+'
5 * (2 + 8 / 3) // une expression plus complexe.
```

Dans une expression, nous pouvons ainsi avoir plusieurs opérateurs. Ces

opérateurs n'ont pas le même ordre de priorité. Par exemple l'expression 5*3+2, correspond au calcul de 5*3 (on obtient un résultat intermédiaire qui est 15) puis on ajoute 2.

L'expression s'évalue donc à 17. Il est important de connaître les priorités entre les opérateurs (quel opérateur est appliqué avant quel autre).

Conseil

Nous allons présenter les opérateurs et leurs priorités. Cependant nous vous conseillons d'ores et déjà d'utiliser des parenthèses pour mettre en évidence l'ordre de calcul que vous souhaitez. En effet, (5*3)+2 ou 5*(3+2) sont des expressions plus claires à comprendre que 5*3+2. L'utilisation de parenthèses affirme l'ordre des calculs souhaités.

Nous retrouvons généralement les expressions à droite d'un opérateur d'affectation. Ce n'est cependant pas toujours le cas, comme nous le verrons les chapitres suivants.

2. Opérateurs

a) Arithmétiques

La construction d'expressions nécessite l'utilisation d'opérateurs arithmétiques synthétisés dans la table suivante :

signe	signification
+	addition
-	soustraction
*	multiplication
/	division
%	modulo
-	nombre négatif

Tableau 2 Opérateurs arithmétiques

Exemple : Exemple de programme

```
#include <stdio.h>
int main()
{
int x , y , resultat;
x = 17;
y = 3;
resultat = x + y;
      // resultat vaut 20
resultat = x - y;
      // resultat vaut 14
resultat = x * y;
      // resultat vaut 51
resultat = x / y;
      // resultat vaut 5 ATTENTION !
resultat = x % y;
      // resultat vaut 2
resultat = -resultat;
      // resultat = -2
}
```

Le résultat de la division est 5. En effet, x et y sont déclarés comme entiers. Le compilateur va donc choisir comme opération la division euclidienne. Le résultat de la division euclidienne 17 / 3 est 5 et son reste 2. On a bien 17 = 3 * 5 + 2.

L'opérateur % donne le reste de la division euclidienne, c'est-à-dire 2.

Attention

Il existe plusieurs types d'arithmétiques, on distingue par exemple l'arithmétique entière et l'arithmétique flottante. Nous rappelons que les entiers et les flottants ne sont pas codés de la même façon en mémoire.

Les opérations +, -, * et / de la table ci-dessus correspondent à des instructions différentes au sein du microprocesseur. A la compilation, il faut que le compilateur choisisse quel type d'instruction machine utilisé.

b) Relationnels

L'ensemble des opérateurs relationnels du C est présenté dans le tableau ci-dessous. Rappelons que le langage C utilise une convention logique (0 pour faux, toute autre valeur que 0 pour vrai), et non pas des valeurs booléennes.

Signe	signification
==	égalité confusion possible avec l'affectation =
!=	différence
<=	infériorité ou égalité
>=	supériorité ou égalité
>	supériorité
<	infériorité

Opérateurs relationnels

Exemple

3 != 4 s'évaluera à vrai,

3 == 4 s'évaluera à faux,

7 == 7 s'évaluera à vrai,

3 <= 8 s'évaluera à vrai, etc.

Attention

Ne pas confondre l'opérateur d'affectation simple "=" avec l'opérateur de test d'égalité "==", que nous venons d'introduire ci-dessus.

c) Logiques

Les opérateurs logiques sont au nombre de trois, ils sont utilisés au sein d'expressions qui s'évaluent à vraie ou faux (selon la convention logique du langage C).

Signe	signification	Binaire ou unaire
&&	et	binaire
||	ou	binaire
!	non	unaire

Tableau 3 Opérateurs logiques

Exemple

```
int x = 3, y = 4, resultat;
resultat = (x < 3) && (y == 4); // ici resultat contient 0,
qui signifie faux selon la convention logique du C
resultat = (x < 3) || (y == 4); // ici resultat contient
autre chose que 0, qui signifie vrai selon la convention
logique du C resultat = !resultat; // ici resultat contient
0, qui signifie vrai selon la convention logique du C
```

Dans l'exemple ci-dessus, nous calculons des expressions avec des opérateurs logiques.

Le résultat de chaque expression est affecté à la variable résultat. Les commentaires indiquent ce que contient cette variable en utilisant la convention logique du langage C.

Le && correspond au et logique, le « || » correspond au ou logique et que le « ! » correspond à l'inversion logique.

d) Bit à bit

Le langage C est semblable au fonctionnement d'un microprocesseur. Il offre la possibilité de manipuler les données au niveau de chaque bit (digit) qui compose un octet (byte). Les opérateurs bit à bit sont utilisés à cette fin.

La table ci-dessous est donnée pour compléter la présentation des opérateurs. Nous ne détaillerons pas le fonctionnement de ces opérateurs dans le cadre de ce cours de programmation C.

Signe	Utilisation et indication succincte du résultat
&	et (1 & 1) -> 1
|	ou (1 | 0) -> 1
^	ou exclusif (1 ^ 1) -> 0
<<	décalage à gauche
>>	décalage à droite

Tableau 4 Opérateurs bit à bit

e) Pré et post incrémentation / décrémentation

Les opérateurs de pré et post incrémentation/décrémentation sont des opérateurs unaires qui s'appliquent à l'identificateur de variable auquel ils sont accolés.

L'incrémentation ++ consiste à augmenter la valeur de la variable de 1, et la décrémentation -- à la diminuer de 1.

Exemple

```
int x = 5;
      // déclaration d'un entier x et initialisation avec
la valeur 5
x++;
      // opérateur unaire de post incrémentation
      // x vaut 6 ici
++x;
      // opérateur unaire de pre-incrémentation
      // x vaut 7 ici
```

On pourrait croire que l'expression x++ est identique à ++x, soit l'écriture x = x +1.

Cela n'est pas tout à fait exact.

Exemple

```
int x = 7, y = 3;
y = y + x++;
      // x vaut 8 ici et y vaut 10 c-a-d 7+3
x = ++y + 50;
      // x vaut 61, c'est-à-dire 11+50 et y vaut 11
```

Explication : si l'on utilise la post-incrémentation dans une expression, alors la valeur de la variable est d'abord utilisée pour effectuer le calcul de l'expression puis elle est incrémentée d'une unité ensuite.

Si l'on utilise la pré-incrémentation dans une expression alors la valeur de la variable est d'abord incrémentée d'une unité avant d'être utilisée ensuite dans le calcul de l'expression.

Exemple

L'exemple ci-dessus est équivalent à :

```
int x = 7, y = 3;
y = y + x;
x = x + 1;
      // x vaut 8 ici et y vaut 10 c-a-d 7+3
y = y + 1
x = y + 50;
      // x vaut 61, c-a-d 11+50 et y vaut 11
```

Cela fonctionne de manière identique pour les pré et post décrémentations

f) Affectation composée

Le langage C a été conçu pour écrire des instructions les plus compactes possibles.Un des objectifs était d'écrire un code source le plus proche des instructions du microprocesseur. De nos jours, les compilateurs sont écrits pour optimiser le fonctionnement du code écrit.

Il n'est plus utile pour des programmes d'utiliser des notations certes courtes mais qui peuvent s'avérer difficiles à comprendre par la suite.

Ci suit table qui résume les combinaisons possibles entre l'opérateur d'affectation = et les opérateurs binaires qui effectuent des calculs.

signe	utilisation	équivalent
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
<<=	x <<= y	x = x << y
>>=	x >>= y	x = x >> y
&=	x &= y	x = x & y
^=	x ^= y	x = x ^ y
|=	x |= y	x = x | y

Opérateurs d'affectation

Cette table est donnée pour information, nous vous conseillons de ne pas d'utiliser ces opérateurs d'affectations composées tant que vous n'avez pas acquis une certaine maîtrise du langage C.

Utilisez plutôt les notations équivalentes données dans le tableau.

g) "adresse de" et "contenu de"

Les deux signes '&' et '*' ont une signification particulière quand ils sont placés devant un identificateur de variable. Ils sont liés à l'usage des variables de type pointeurs qui ne serviraient à rien s'il n'y avait pas la possibilité d'accéder à l'adresse d'une variable car on ne pourrait pas initialiser la valeur d'une variable de type pointeur.

Il s'agit de deux opérateurs unaires. Le '&' permet de récupérer la valeur de l'adresse du premier octet d'une variable, le '*' permet d'utiliser la valeur d'une adresse (premier octet d'un paquet d'octets) pour accéder au paquet d'octets dans son ensemble.

Exemple : Exemple : utilisation de pointeurs

```
int i =17;
      // Declare une variable entiere (supposons codee sur
4 octets)
int *pt_i ;
      // Declare une variable de type pointeur sur un
entier, le contenu
      // sera une valeur qui correspondra à l'adresse
memoire d'un octet
      // a ce stade, i est initialisee avec la valeur 0,
mais pt_i n'est pas initialisee.
pt_i = &i;
      // Initialise le pointeur avec l'adresse du premier
octet de la variable i
      // a ce stade pt_i contient l'adresse d'un unique
octet MAIS le compilateur sait que
      // pt_i est une variable de type pointeur sur entier.
*pt_i = *pt_i+1;
      /* Effectue un calcul sur la variable pointée par
pt_i, c'est-a-dire sur i lui-même, puisque pt_i contient
          l'adresse de i. */
      /* A ce stade, i ne vaut plus 0, mais 1. */
```

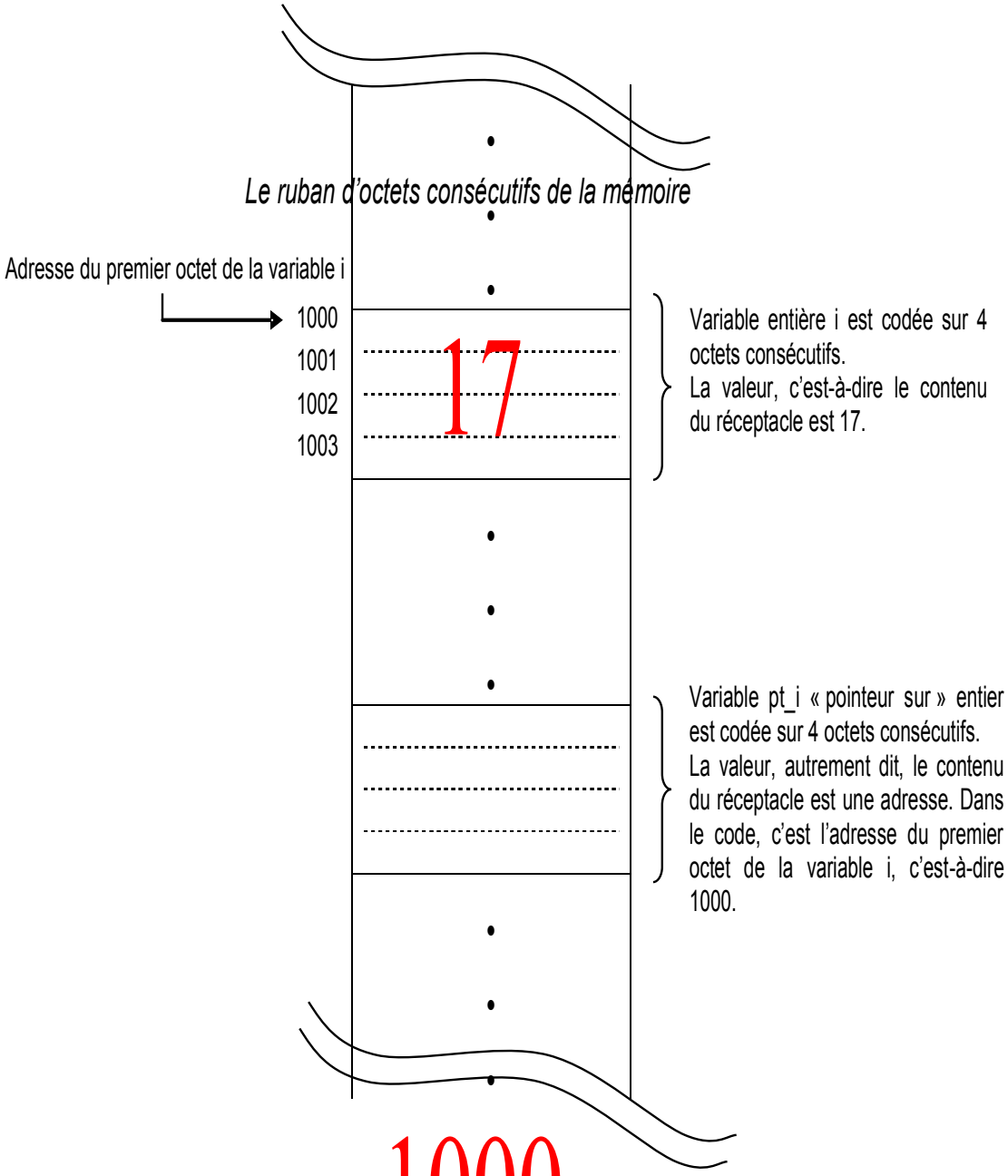
Détail de l'instruction *pi = *pi+1.

La figure qui suit montre une carte possible de la mémoire centrale utilisée par le programme. C'est un extrait du ruban d'octet qui constitue la mémoire centrale. Nous allons ici considérer qu'un entier est codé sur 4 octets consécutifs. A droite de l'opérateur d'affectation = nous avons une expression *pt_i +1.

La variable pt_i contient une valeur de type adresse mémoire (elle a été déclarée comme un pointeur vers un entier). L'opérateur unaire * signifie que l'on va utiliser la valeur de l'adresse mémoire (du premier octet de l'entier) pour récupérer la

valeur d'un entier qui est codé dans les 4 octets consécutifs.

Puisque pt_i contient la valeur de l'adresse du premier octet qui correspond à l'emplacement mémoire où se trouve rangée la valeur de l'entier i et que le compilateur sait que c'est un pointeur vers un entier, nous allons récupérer exactement les 4 octets consécutifs qui contiennent la valeur de l'entier i.



Une illustration de la « carte » de la mémoire utilisée par le programme. Nous avons représenté les réceptacles associés aux variables du programme au sein du « ruban » d'octet.

Graphique 1 Une illustration de la « carte » de la mémoire utilisée par le programme

A ce stade *pt_i vaut donc 0 dans l'expression *pi+1. Nous ajoutons ensuite 1. La valeur de l'expression est donc 1. A gauche de l'opérateur d'affectation nous

avons *pt_i. Cette fois-ci, puisque nous sommes à gauche de l'opérateur d'affectation, l'opérateur unaire (*) signifie que l'on va utiliser la valeur de l'adresse mémoire (premier octet de l'entier) pour ranger la valeur d'un entier (c'est donc 1) dans les 4 octets consécutifs.

En résumé le contenu de la variable i est 1 désormais. En d'autres termes *pt_i = *pt_i +1 est équivalent à i = i+1, sauf que nous avons utilisé une indirection via une adresse pour réaliser l'opération.

Attention

Il est très important de s'assurer que les pointeurs que l'on manipule sont initialisés correctement et ne contiennent pas n'importe quoi.

En effet, accéder à une zone mémoire via un pointeur non initialisé revient à lire ou à écrire dans la mémoire à un endroit inconnu et non maîtrisé (selon la valeur initiale du pointeur lors de sa création).

Complément

En général, on initialise les pointeurs dès leur création, ou, s'ils doivent être utilisés ultérieurement, on les initialise avec le pointeur nul (NULL). Cela permettra de faire ultérieurement des tests sur la validité du pointeur ou au moins de détecter les erreurs.

Le pointeur nul se note NULL. C'est une valeur particulière qui est définie dans le fichier d'en-tête stdlib.h. En C, elle représente la valeur d'une adresse invalide.

Attention

Il ne faut pas confondre l'utilisation du signe (*) lors de la déclaration d'une variable de type "pointeur vers " et l'utilisation du signe (*) au sein d'une instruction.

Dans le premier cas, cela permet d'avertir le compilateur : la valeur (une adresse mémoire) de la variable de type "pointeur vers" sera utilisée pour manipuler autant d'octets que nécessaires (la taille du réceptacle) pour coder le type de la variable qui est pointée.

Dans le second, cela permet de manipuler, effectivement, tout le paquet d'octets en utilisant l'adresse du premier. Déclarer le type de ce qui est pointé permet de vérifier que les opérations effectuées sont compatibles avec ce type. Il est à présent facile de comprendre pourquoi il faut spécifier le type de donnée qui est pointée.

Exemple

Considérons les déclarations suivantes :

```
int *pt_1 ;
      // variable de type pointeur vers un entier
char *pt_a;
      // variable de type pointeur vers un caractère
float *pt_z;
      // variable de type pointeur vers un float
```

Il faut interpréter correctement ces déclarations. Les identificateurs pt_1, pt_a et pt_z sont des identificateurs de variables.

Ces variables contiennent des valeurs qui sont des adresses d'octets, ce sont des pointeurs.

En précisant le type de la valeur pointée, le compilateur peut déterminer le nombre d'octets consécutifs à utiliser pour les opérations de récupération d'information ou de recopie d'information. C'est-à-dire la taille de la donnée qui est

automatiquement pris en compte.

Remarque

Lors d'une déclaration, il faut répéter l'étoile dans la déclaration de plusieurs pointeurs : int *a, *b, *c,*d; .... ce qui signifie que syntaxiquement a, b, c et d sont des pointeurs d'entiers. Donc *a, *b, *c et *d sont des entiers quand on utilise cette syntaxe DANS une instruction.

Si l'on écrit : int *a, b, c, d;.... Selon cette déclaration, seul a est un pointeur d'entier, les autres sont des entiers.

Attention

Considérons le code suivant :

```
int i = 58;
      // Déclare une variable entière (supposons codée sur
4 octets)
int *pt_1 ;
      // Déclare une variable de type pointeur sur un
entier, le contenu
      // sera une valeur qui correspondra à l'adresse
mémoire d'un octet
      // A ce stade, i est initialisée avec la valeur 58,
mais pt_1 n'est pas initialisée.
pt_1 = &i;
      // Initialise le pointeur avec l'adresse du premier
octet de la variable i
      // A ce stade pt_i contient l'adresse d'un unique
octet MAIS le compilateur sait que c'est
      // Une variable de type pointeur sur entier.
pt_1 = pt_1+1;
      // augmente de 1 valeur de la variable pointeur
(c'est ce que l'on croit ...)
*pt_1 = *pt_1 - 58;
```

Ces instructions ne provoqueront pas d'erreur syntaxique à la compilation. Cependant, nous commettons une erreur. La valeur de pt_1 est initialisée avec la valeur de l'adresse du premier octet de l'entier i codé sur 4 octets consécutifs.

Augmenter la valeur de la variable pt_1 va faire que l'on ne pointera plus sur le premier octet. En toute logique on pourrait se dire que l'on pointe sur le deuxième octet (à cause du +1), puisqu'ils sont consécutifs.

Ce raisonnement est faux. En fait, la nouvelle valeur de pt_1 est égale à l'ancienne plus 4 octets de décalage.

A la compilation le compilateur possède l'information sur la taille dévolue au stockage d'un entier (nous avons supposé ici que c'était 4). Puisque la variable pt_1 est déclarée comme un pointeur vers un entier, le compilateur va supposer que l'on veut passer à l'entier suivant. Il va ainsi augmenter la valeur de pt_1 de la quantité 4.

Complément

En résumé l'instruction :

pt_1 = pt_1 + 1;

Compilé par le compilateur C, va générer l'instruction suivante à destination du micro processeur: pt_1 = pt_1 + 1*sizeof(int)

Cette génération est totalement transparente pour le programmeur qui doit savoir intérpréter cette instruction. Nous avons dit qu'il existait une arithmétique entière

et une arithmétique flottante, il existe également une arithmétique particulière sur les pointeurs.

Elle tient compte de la taille de la donnée qui est pointée, c'est-à-dire du type de donnée utilisée pour déclarer la variable "pointeur vers".

3. Problème de conversion implicite et conversion explicite

Nous allons aborder le problème important en C des conversions réalisées par le compilateur au sein des expressions. Elles sont dites implicites

Exemple

int x = 3, y = 2; // déclaration de deux variables entières x et y et initialisation float z; // déclaration d'une variable flottante z z = x/y;

```
int x = 3, y = 2;
// déclaration de deux variables entières x et y et
initialisation
float z; //
déclaration d'une variable flottante z
z = x/y;
```

Le contenu de z va être 1.0 et non pas 1.5. Ce contenu est logique si l'on connaît la manière dont le langage C réalise la compilation et mène les évaluations.

En premier lieu, rappelons qu'une variable entière n'est pas codée de la même manière qu'une variable flottante. Au niveau des instructions du micro processeur il y a des opérations en arithmétique entières et des opérations en arithmétique flottante.

Les opérations comme +, - , * et / par exemple ne sont pas réalisées de la même façon selon que les opérandes sont entières ou flottantes. Quand le compilateur rencontre l'expression x/y il choisit d'effectuer une opération soit en arithmétique entière soit en arithmétique flottante. Observons ce qui déclenche le choix.

Méthode

Les règles sont simples :

Si les opérandes sont de même type, l'instruction sera traitée à la compilation en fonction de ce type : arithmétique entière ou (exclusif) arithmétique flottante. Si les opérandes sont de types différents (entier et flottant) le compilateur choisira de traiter l'instruction en arithmétique flottante

Dans l'exemple ci-dessus, x et y sont entiers. C'est la division entière qui sera choisie, c'est-à-dire le résultat de la division Euclidienne en mathématique.

L'expression x/y s'évaluera à la valeur entière 1. Ce résultat sera ensuite affecté à la variable flottante z.

Les deux formats entier et flottant ne sont pas les mêmes. Le compilateur va donc automatiquement générer une instruction de conversion de type et il va mettre en place une conversion du résultat entier 1 en un flottant 1.0 puis ensuite ranger cette valeur dans la variable z.

En résumé, le compilateur choisit le type d'instructions arithmétiques en fonction des opérandes et en plus il effectue des opérations de conversion de format qui sont implicites (entière ou flottante).

Exemple : Autre exemple

```
int x, y = 2;
// declaration de deux variables entieres x et y et
initialisation
float z = 3.0;
// declaration d'une variable flottante z
x = z/y;
```

A l'exécution la variable x contiendra la valeur 1.

Voici comment le compilateur travaille :

- Il constate que z est un flottant (le type de la variable est connu)
- Il constate que y est un entier (le type de la variable est connu)
- Il choisir d'effectuer une instruction de division flottante
- Le type de y n'est pas compatible avec une instruction de division flottante
- Le compilateur génère automatiquement une instruction de conversion pour transformer y qui contient une valeur entière en une valeur flottante qu'il pourra utiliser pour l'instruction de division flottante.

Attention

La variable y est toujours un entier, sa valeur sera convertie au sein même du microprocesseur en une valeur flottante, aucune variable supplémentaire n'est créée, la valeur intermédiaire créée n'existe qu'au sein du microprocesseur que le temps nécessaire pour effectuer le calcul.

Ceci est donc dynamique et temporaire.

- Le résultat de l'évaluation de l'expression z/y sera un flottant.
- L'instruction d'affectation simple veut ranger cette valeur dans une variable de type entier (le type de la variable est connu).
- Il constate que les deux formats de données ne sont pas compatibles.
- Le compilateur génère automatiquement une nouvelle instruction de conversion pour transformer le résultat de l'expression qui est une valeur flottante en une valeur entière.
- Finalement, il met en place l'instruction pour ranger cette dernière valeur au sein de la variable entière x.

Complément

A l'exécution, voici ce qui ce passe : La valeur de z est 3.0, la valeur de y est 2, elle est dynamiquement convertie en 2.0, puis le calcul 3.0/2.0 est effectué, on obtient 1.5 (c'est un flottant), puis la valeur 1.5 (flottant) est convertie en la valeur entière 1 (troncature), la valeur 1 obtenue est rangée dans la variable x.

Le contenu de la variable x est désormais 1 (entier). Comme vous pouvez le constater, cette simple instruction "x = z/y;" induit des comportements par défaut du compilateur.

Il faut les connaître car ils peuvent être une source importante d'erreur quand on débute en langage C.

Attention

La conversion d'entier en flottant s'effectue sans perte d'information. La conversion de flottant en entier s'effectue avec des troncatures, donc avec perte d'informations.

En résumé : si les opérandes sont de type différents c'est le type le plus "large" qui

est choisi.

Les conversions de type peuvent être maîtrisées par le programmeur grâce à des demandes explicites de conversion de type. C'est ce que l'on appelle en C les opérations de « cast ».

Exemple

```
int x = 3, y = 2;
// déclaration de deux variables entières x et y et
initialisation
float z, t ;
// déclaration d'une variable flottante z
t = (float )x;
// demande explicite de conversion de x, entier en une
valeur flottante
// que l'on range dans la variable t qui est un flottant, t
contient 3.0
z = t/y;
// t est un flottant, donc arithmétique flottante donc la
valeur de y est dynamiquement
// Convertie en valeur flottante pour mener à bien le
calcul.
// L'expression retourne 1.5 et donc la variable z contient
1.5 (flottant)
z = (float)x/y ; // cette instruction fera le même travail
z = x/(float ) y; // et celle-ci aussi
```

Complément : En résumé

Pour demander explicitement la conversion d'une valeur d'une expression dans un type voulu, il suffit d'utiliser la syntaxe : (<type>) devant l'expression, sachant que <type> doit être un type prédéfini.

A ce niveau de connaissance, nous pouvons utiliser par exemple (int), (char), (float) comme opérateurs de « cast ». L'opérateur « cast » est opérateur unaire.

4. Priorité des opérateurs

Nous donnons dans le tableau ci-dessous les priorités relatives entre les opérateurs. Les priorités sont décroissantes du haut du tableau vers le bas. La colonne de gauche contient les opérateurs et la colonne de droite indique le sens de l'évaluation selon ces opérateurs. Le tableau comprend tous les opérateurs du C.

Vous remarquerez que certains n'ont pas encore été rencontrés. Ils le seront dans les chapitres suivants.

Opérateurs	évaluation
( ) [ ] -> .	gauche-droite
! ~ ++ -- (<type>) * & sizeof	droite-gauche
* / %	gauche-droite
+ -	gauche-droite
<< >>	gauche-droite
< > <= >=	gauche-droite
== !=	gauche-droite
&	gauche-droite
^	gauche-droite
|	gauche-droite
&&	gauche-droite
||	gauche-droite
? :	droite-gauche
= += -= *= /= %= &= ^= |= <<= >>=	droite-gauche

Priorité des opérations

Conseil

Comme vous pouvez le remarquer dans le tableau, les parenthèses sont au plus haut niveau de priorité. Une expression entourée de parenthèses sera évaluée en premier.

Il ne faut pas hésiter à les utiliser, elles clarifient l'écriture des expressions et cela vous évitera d'avoir à connaitre parfaitement le tableau ci-dessus.

Vous remarquez aussi que les opérations de changement de type (c'est-à-dire le cast) sont notées "(<type>)" dans le tableau ci-dessus et qu'elles ont un haut degré de priorité.

5. Instructions d'entrées / sorties

Les fonctions d'entrée/sortie permettent l'échange d'informations entre les variables du programme et les périphériques (clavier, écran, disque, etc.). Nous les considérons comme des entrées/sorties standards qui concernent le clavier et l'écran.

Nous allons présenter 4 éléments de la bibliothèque standard <stdio.h> : "getchar()" et "putchar()", "printf" et "scanf".

Ces fonctions sont définies dans le fichier <stdio.h> que l'on inclut au début d'un programme.

N.B : Les caractères constants (littéraux constants) sont repérés entre une paire de signe ' dans un programme C. Il s'agit d'un unique caractère par exemple 'b'. Une chaîne de caractères constante est repérée entre une paire de caractères ".

Par exemple "\nbonjour".

a) Caractères spéciaux

Définition

Nous allons utiliser certains caractères de la table des codes ASCII qui permettent de réaliser des affichages particuliers. Pour pouvoir les manipuler, le langage C utilise ce que l'on appelle une « séquence d'échappement ». Cela veut simplement dire qu'ils sont précédés d'une contre barre \.

Cette dernière permet d'avertir le compilateur que le caractère qui suit la \ doit être considéré de façon spéciale.

Nous en avons déjà rencontré un : le '\n' qui effectue un saut de ligne puis un positionnement au début de la ligne suivante.

Voici la liste des séquences d'échappements

- \n : nouvelle ligne (NL=New Line)
- \r : retour chariot (CR= carriage return)
- \t : tabulation horizontale (HT= horizontale tabulation)
- \f : saut de page (FF= form feed)
- \v : tabulation verticale (VT= vertical tabulation)
- \a : signal d'alerte (un bip)
- \b : retour arrière (BS = BackSpace)
- \\ : caractère d'échappement (affiche l'antislash \)
- \YY affiche le caractère dont le code ASCII est représenté par sa valeur hexadécimale. YY étant la valeur comprise entre 00 et FF.
- \YYY affiche le caractère de code octal YYY

Les séquences sont utilisables avec la fonction printf() mais aussi dans d'autres circonstances.

Attention

Puisque le ' est utilisé pour marquer le début et la fin d'un littéral constant de type caractère, il faut utiliser le '\'' si on veut afficher le caractère ' seul. Ce n'est donc pas un caractère spécial. Cependant la séquence d'échappement est nécessaire si l'on veut manipuler le caractère '.

De la même façon, si on veut afficher " ou \ au sein d'une chaîne de caractères constante il faudra écrire \" et \\.

b) Fonction d'écriture putchar()

Définition

Le mécanisme d'entrée sortie de cette fonction est très simple et fiable à condition d'en comprendre le fonctionnement.

La fonction "putchar()" manipule uniquement des caractères, c'est à dire des octets dont le contenu correspond au code ASCII du caractère. Elle écrit le caractère passé en argument sur la sortie standard (l'écran).

Le paramètre à placer entre les () doit être de type caractère.

Exemple

```
#include <stdio.h>
int main()
{
char c;
c = '\n';
putchar(c);
putchar('\'');
putchar('\t');
putchar('\0');
       // c'est un caractere qui ne s'affiche pas a l'ecran
(invisible)
      // Il est utilise pour marquer la fin d'une chaine de
caractere
putchar('A');
      // c'est le caractere A dont le code ASCII decimal
est 65
c = 65;
      // puisque 65 est le code ASCII du caractere et que
la variable c
      // peut etre manipulee comme un entier putchar(c);
      // on affiche a nouveau le caractere A putchar('\f');
putchar('a');
c = '\n';
putchar(c);
}
```

Affichera :

' AA

a

c) Fonction de lecture getchar()

Définition

C'est une fonction sans paramètre qui « capte » la valeur du caractère lu à la

position courante de l'entrée standard (le clavier) et le retourne. Quand un programme exécute cette fonction, le curseur dans la fenêtre d'exécution est « en attente », l'utilisateur doit alors taper au moins un caractère du clavier.

Ce caractère apparaît aussi à l'écran (en écho) puis il doit terminer sa saisie par la touche « return ».

Exemple

```
#include <stdio.h>
int main()
{
char c;
c = getchar();
       // le programme suspend son fonctionnement jusqu'a
ce que l'utilisateur tape
      // au moins un caractère puis la touche <return>
putchar(c);
}
```

L'utilisateur tape par exemple la touche <a> puis la touche <return>. Il a tapé deux caractères car le <return> est un caractère en fait c'est le caractère '\n' (NL).

On obtient à l'affichage :

a

a

Le premier 'a' affiché correspond à ce que tape l'utilisateur (c'est l'écho d'affichage), le second 'a' correspond à l'affichage du caractère saisi réalisé avec "putchar".

Fondamental

Lorsque l'utilisateur tape quelque chose au clavier, les caractères tapés sont stockés dans une zone appelée zone tampon ou buffer. Le programme consomme les caractères de cette zone avec getchar().

L'appui sur la touche <return> provoque l'envoie du tampon vers le programme. Il ne vous a pas échappé que l'utilisateur a tapé 'a' puis 'return', le caractère qui correspond à la touche <return> est l'interligne, c'est-à-dire le caractère '\n', que nous noterons aussi <NL> dans le texte explicatif pour visualiser les effets de son affichage.

Il est toujours dans le tampon et peut être consommé.

Exemple : Premier exemple

```
#include <stdio.h>
int main()
{
char c;
c = getchar();
      // le programme suspend son fonctionnement, il attend
une donnée du clavier
    // L'utilisateur tape le caractère 'a'
    // puis <return>, le tampon contient deux caractères.
Le programme reprend son
    // Fonctionnement et ce premier getchar consomme le
premier caractere 'a'.
    // Dans la variable c il y a le caractere 'a'.
    // Il reste toujours le caractere <return> dans le
tampon.
```

```
c = getchar();
    // Ce second getchar consomme le caractere <return>.
   // Dans la variable c il y a le caractere <return>. Le
tampon est vide.
putchar(c);
   // On affiche donc un <return>, c-a-d '\n', c-a-d <NL>.
}
```

Pour les besoins de l'explication, nous écrirons explicitement <NL> quand le caractère est affiché.

A l'exécution si vous tapez le caractère 'a' suivit de <return>, vous aurez à l'écran :

a <NL>

<NL>

Le programme ne se bloque pas en attente d'un caractère de la part de l'utilisateur au deuxième "getchar()" car le tampon n'est pas vide.

On continue de consommer les caractères du tampon.

Exemple : Deuxième exemple

```
int main()
{
char c;
c = getchar();
    // Le programme suspend son fonctionnement, il attend
une donnée du clavier
    // si vous tapez exactement : alpha<return>, vous
consommez 'a'
putchar(c);
    // vous affichez 'a'
c = getchar();
    // Le tampon n'est pas vide, le programme ne bloque
pas, vous consommez 'l'
putchar(c);
    // vous affichez 'l' }
```

Le programme ci-dessus ne demande pas une nouvelle saisie au second getchar(). Il consomme les caractères disponibles dans le tampon. Et cela peut continuer tant que le tampon n'est pas vide.

Complément

Les deux fonctions "getchar" et "putchar" sont limitatives. On peut toujours convertir le caractère ou la suite de caractères en un type donné ou inversement (entier par exemple), mais pour faciliter l'échange, on utilise les entrées/sortie formatées suivantes qui ont été conçues pour cela.

d) Fonction d'écriture : printf ()

Syntaxe : Usage

printf (param_1, param_2, param_3,..., param_n)

où param_1 à param_n est une suite d'au moins un paramètre transmis à la fonction, séparés par des ','.

param_1 : est obligatoire c'est une chaîne de caractères qui comporte des caractères à afficher tels quels et éventuellement des consignes de formatage pour les autres paramètres (s'il y en a). Il doit y avoir autant de consignes que de paramètres restant à afficher.

Le résultat est tel que l'on affiche les arguments param_2, param_3,..., param_n

aux formats spécifiés dans param_1.

Une consigne de formatage commence toujours par le caractère '%', suivi d'autres caractères juxtaposés qui définissent le format d'affichage. C'est donc une séquence d'échappement.

Les paramètres param_2, param_3,..., param_n sont des expressions qui retournent une valeur. Cela peut être simplement des identificateurs de constantes ou de variables, ou en encore des littéraux.

Exemple

```
printf("\nbonjour chez vous");
```

affichera : bonjour chez vous

Le '\n' n'est pas un formatage, c'est juste le caractère interligne. On saute donc une ligne avant d'afficher la suite de caractères "bonjour chez vous".

Dans ce cas param_1 est le seul paramètre. Il s'agit d'une simple chaîne de caractères sans consigne de formatage.

Les consignes décrivent comment écrire les paramètres param_2, param_3,..., param_n à l'écran.

Exemple

```
#include <stdio.h>
int main()
{
char car;
car = getchar();
// on suppose que l'utilisateur tape le caractère A puis
<return>
printf("le caractere est %c et son code est %d\n",car,car);
}
```

A l'écran nous aurons :

< NL>

Le caractère est A et son code est 65 < NL >

Remarque

Vous avez remarqué que la valeur de la variable car était affichée avec la consigne %c puis la valeur de cette variable était affichée avec la consigne %d. On affiche car sous sa forme caractère puis sous la forme de la valeur de son code ASCII exprimé en décimal.

Les formats que nous utiliserons couramment seront %c (pour char), %d (pour int), %f (pour float), %s (pour char *, c.-à-d.chaîne de caractères) et %p (pour la valeur d'un pointeur, c-à-d une adresse, c.-à-d. un <type> *).

Méthode

Nous donnons, à titre d'information, la façon de construire la séquence de formatage, mais nous ne détaillerons pas. A l'utilisateur de faire des essais ou de se reporter à un ouvrage de référence. Après le signe % du début de la séquence d'échappement, on trouve :

- Eventuellement un ou plusieurs indicateurs modifiant la signification de la conversion (-, +, #).
- Eventuellement un nombre pour indiquer la taille minimale du champ d'affichage concerné.
- Eventuellement un '.' suivi d'un nombre pour la précision (pour les nombres

à virgules).

- Eventuellement le caractère 'l' si le nombre entier concerné (formats ld, lo, lu, lx ou lX) est un entier long ou lf si c'est un double ;

Enfin, et obligatoirement le caractère indiquant le format de conversion parmi :

instruction C	résultat
printf("|%d|\n",12345);	|12345|
printf("|%+d|\n",12345);	|+12345|
printf("|%8d|\n",12345);	| 12345|
printf("|%8.6d|\n",12345);	| 012345|
printf("|%x|\n",255);	|ff|
printf("|%X|\n",255);	|FF|
printf("|%#x|\n",255);	|0xff|
printf("|%f|\n",1.23456789012345);	|1.234568|
printf("|%10.4f|\n",1.23456789);	| 1.2346|

Tableau 5 Quelques exemples

Exemple

```
#include <stdio.h>
int main()
{
float pourcent = 3.85;
printf("\nle pourcentage est\n$%+10.3f$\n",pourcent);
}
```

Affichera :

le pourcentage est

$ +3.850$

Entre les deux caractères $ il y a 10 caractères, le signe + a été ajouté, il y a 3 chiffres après la virgule.

C'est le format demandé pour la valeur de la variable pourcent de type float.

caractère	type argument	Conversion à l'affichage
d	int	entier en notation décimale signée
o	int	entier en notation octale non signée
x, p ou X	int	entier en notation hexadécimale non signée ; les lettres a, b, c, d, e, f sont utilisées pour x et p ; les lettres A, B, C, D, E, F pour X
u	int	entier en notation décimale non signée
f	float ou double	réel en notation décimale signée de la forme [-]mmm.dddddd où le nombre de d est déterminé par l'argument de précision. Par défaut c'est 6.
e ou E	float ou double	réel en notation décimale de la forme [-]m.ddddddexx pour e et [-]m.ddddddExx pour E. Le nombre de d est déterminé par l'argument de précision. Par défaut c'est 6.
g ou G	float ou double	affichage dans le style e ou E quand l'exposant est inférieur à -4 ou supérieur à la précision, sinon affichage dans le style de f
c	char ou int	affichage d'un seul caractère pour char et affichage d'un seul caractère après la conversion de l'int en unsigned char
s	char *	affichage d'une chaîne de caractères jusqu'à trouver le caractère marqueur de fin de chaîne '\0' ou jusqu'à ce que le nombre de caractères affichés corresponde à la précision
p	void *	affichage de l'argument en notation pointeur (adresse en hexadécimal)
%		permet l'affichage du caractère '%' lui-même

Tableau des caractères spéciaux

e) Fonction de lecture : scanf ()

Définition

L'objectif de cette fonction est de récupérer des données lues au clavier dans des variables avec conversion automatique à l'aide d'un format de conversion.

Syntaxe : Usage

scanf(param_1, param_3,..., param_n)

param_1 : est obligatoire c'est une chaîne de caractères qui comporte uniquement

des consignes de formatage pour les autres paramètres (il y doit y en avoir au moins un).

Il faut autant de consignes que de paramètres.

Exemple

```
#include <stdio.h>
void main()
{
char car;
int a, b;
printf("\nentrez un caractere puis return ");
scanf("%c",&car); printf("\nle caractere tape est : %c
",car);
printf("\n\nentrez maintenant :");
printf("un entier, un espace, un caractere, un espace, un
entier");
printf("\npuis return");
scanf("%d %c %d",&a,&car,&b);
printf("\nvous avez lu %d %c %d",a,car,b);
printf("\n");
}
```

Attention

Vous remarquez dans l'exemple que param_2, param_3,..., param_n sont des pointeurs, c'est à dire des adresses mémoires. En effet, une valeur lue au clavier doit être rangée dans une variable (réceptacle), il faut savoir où se trouve ce réceptacle dans la mémoire centrale.

Nous avons donc besoin d'avoir son adresse. Nous utilisons pour cela l'opérateur "&" devant l'identificateur de la variable pour pouvoir obtenir son adresse mémoire.

Scanf() range aux adresses qui correspondent aux arguments param_2, param_3,..., param_n des informations issues de chaînes de caractères tapées au clavier qui sont AUTOMATIQUEMENT converties aux formats spécifiés dans la partie param_1.

Scanf( ) ne peut pas afficher quelque chose, c'est le rôle de printf. La primitive scanf lit les caractères sur l'E/S standard (le clavier), puis les interprète et les convertit selon les formats spécifiés, et enfin stocke les valeurs dans les emplacements mémoires donnés par les arguments.

Chaque format de conversion est introduit par le caractère '%' qui est une séquence d'échappement. Les formats sont du même type que ceux du printf.

C'est à dire %c %d %f ou %s. Cependant, comme il ne s'agit pas d'un affichage, on ne précise donc pas autre chose que %c, %d, %f ou %s.

Méthode

Il faut impérativement mettre & devant la variable car on a besoin de l'adresse de la variable, sauf bien entendu si la variable elle-même est une adresse (une variable de type "pointeur vers", correctement initialisée).

Bien qu'il soit possible d'effectuer plusieurs lectures comme dans l'exemple avec scanf("%d %c %d",&a,&car,&b) nous préconisons plutôt d'effectuer des lectures séparées car cela évite des erreurs de saisie. La fonction "scanf" fait automatiquement un retour à la ligne après son exécution (après la saisie de l'utilisateur).

6. Choix Simple, structures alternatives

Syntaxe

La syntaxe est la suivante :

if (condition) instruction_si_vrai else instruction_si_faux

Remarque

La condition est obligatoirement entre parenthèses. Le compilateur peut ainsi facilement détecter le début de la condition logique et la fin de la condition logique.

Le mot clef "then" utilisé en algorithmique ou dans d'autres langages est inutile car l'instruction_si_vrai est immédiatement après la parenthèse fermante. La partie "else" est facultative.

Nous rappelons qu'une instruction peut être un bloc qui est assimilé à une unique instruction à activer.

La valeur de la condition utilise la convention logique du langage C : 0 est faux, autre chose que 0 est vrai.

On peut imbriquer plusieurs "if". Dans ce cas il est conseillé d'utiliser des indentations (marges décalées) pour améliorer la lisibilité du code.

Exemple : Exemple 1 : illustration de l'usage de la convention logique du C

```
void main()
{
float X;
printf("\nTapez un nombre flottant : ");
scanf("%f", &X);
if (X)
   printf("\nx = %f c'est vrai pour le C",X);
else
    printf("\nx= %f c'est faux pour le C",X);
}
```

Si l'on tape 0.0 ou encore 0, cela affichera "x= 0.0 c'est faux pour le C", dans tous les autres cas cela affichera la valeur saisie suivie de "c'est vrai pour le C". Vous pouvez essayer avec une variable de type int.

Exemple : Exemple 2 : Equation du premier degré avec C=0

Ecrire un programme qui résout une équation du premier degré Ax+b=0. Elle lit les valeurs de A et B entrées par l'utilisateur.

```
/*Programme Premier_Degre*/
#include < stdio.h >
Int main()
{
float a,b ;
printf("\nentrez le coefficient a : ");
scanf("%f",&a);
printf("\nentrez le coefficient b : ");
scanf("%f",&b);
printf("\ncoefficients a = %f et b = %f : ",a,b);
if (a==0)
   if (b==0)
          printf("indetermine !! \n") ;
   else
           printf("impossible !! \n") ;
```

```
else
    printf("la solution est : %f", -b/a);
printf("\n");
return 0;
}
```

Exemple : Exemple 3 : Maximum de deux nombres

Ecrire un programme qui calcule le maximum de deux nombres entrés au clavier

```
#include < stdio.h >
void main
{
float X,Y ;
float MAX;
printf("\nTapez un nombre : ");
scanf("%f", &X);
printf("\nTapez autre nombre : ");
scanf("%f",&Y);
if (X > Y ) /*évaluation de la condition*/
   MAX = X ;
else
    MAX = Y ;
printf("\nLe plus grand nombre est %f ",MAX);
printf("\n");
}
```

Attention

voici un code intéressant pour comprendre une erreur classique en C.

```
void main()
{
int x,y;
printf("\nTapez un nombre : ");
scanf("%d", &x);
printf("\nTapez autre nombre : ");
scanf("%d",&y);
if (x=y)
{
      x = 1;
      y = 1;
}
else
{
    x = 2;
    y = 2;
}
printf("\nx = %d et y = %d ",x,y);
}
```

Ce programme affichera toujours x=1 et y=1 sauf si y vaut 0. Dans ce cas, quelque soit la valeur de x, on affichera x=2 et y=2.

En effet, nous avons écrit if (x=y) au lieu de if (x==y). L'opérateur d'affectation n'est pas le même que l'opérateur de test. De plus, on a le droit d'utiliser l'opérateur d'affectation dans une expression qui correspond à une condition logique (nous vous le déconseillons fortement pour débuter).

Donc x=y, signifie bien x prend la valeur de y. Mis à part le cas où y vaut 0, l'expression est vrai au sens de la convention logique du langage C (voir exemple

1). Le bloc avec les instructions x=1; y=1; s'effectue et on affiche finalement "x = 1 et y = 1 ".

Si y vaut 0, l'expression est fausse au sens de la convention logique du C. Le bloc x=2; y=2; s'effectue

On affiche finalement "x = 2 et y = 2". C'est une erreur très classique que l'on risque de réaliser régulièrement si l'on tape le code trop vite sans le relire.

7. Instruction "break"

L'instruction break ne peut s'utiliser qu'à l'intérieur d'une boucle (que nous verrons au chapitre suivant) ou d'un « switch ».

Dans le premier cas, elle correspond à un branchement sans condition sur la première instruction qui suit la boucle dans laquelle elle est utilisée.

Dans le second, elle permet de se brancher à la première instruction qui suit le '}' fermant le switch.

Dans les deux cas, le programmeur ne gère lui-même ni étiquette, ni branchement. Le branchement est automatiquement géré par le compilateur.

8. Sélection Multiple : l'instruction switch

L'instruction « switch » permet d'effectuer une suite de tests d'égalité consécutifs pour une valeur donnée et de déclencher des instructions selon la valeur.

Exemple

Nous allons illustrer son fonctionnement par un exemple.

```
int main()
{
int c;
printf("\n entrer un nombre ");
scanf("%d",&c);
switch (c)
  {
      case 1: printf("\n un");
      case 2 : printf("\n deux");
      case 3 : printf("\n trois");
      default : printf("\n autre chose");
   }
printf("\n");
}
```

Si l'utilisateur tape le chiffre 1, on affiche : un deux trois autre chose S'il tape le chiffre 2 on affiche : deux trois autre chose S'il tape le chiffre 3 on affiche : trois autre chose S'il tape un entier autre que 1, 2 ou 3, on affiche : autre chose

La valeur de c étant connue, le switch va tester d'abord si elle vaut 1.

Si c'est le cas il va en séquence exécuter TOUTES les instructions à partir de celle qu'il doit effectuer si le test est vrai.

Si c ne vaut pas 1, il teste s'il vaut 2, s'il vaut 2 il va en séquence exécuter TOUTES les instructions à partir de celle qu'il doit effectuer si le test est vrai, etc.

C'est ainsi que fonctionne un « switch ». Si l'on désire exécuter juste l'instruction associée à un « case », il faut ajouter l'instruction break après.

Le code modifié est :

```
int main()
```

```
{
int c;
printf("\n entrer un nombre ");
scanf("%d",&c);
switch (c)
   {
      case 1: printf("\n un"); break;
      case 2 : printf("\n deux"); break;
      case 3 : printf("\n trois"); break;
      default : printf("\n autre chose");
   }
// si une des instructions 'break' ci-dessus est effectuee,
on se branche directement ici
// apres le delimiteur '}' qui ferme le bloc de
l'instruction 'switch'.
printf("\n");
}
```

Dans ce dernier exemple, si l'utilisateur tape '1', le programme affichera uniquement 'un'. S'il tape '2', le programme affichera uniquement 'deux".

S'il tape '3', il affichera uniquement 'trois'. Si l'utilisateur tape autre chose, le programme affichera 'autre chose'.

L'instruction break effectue un branchement automatique au delà du délimiteur '}' qui ferme le bloc de l'instruction "switch".

Attention

Pour comparer on ne peut utiliser, dans le « case », que des expressions constantes. C'est à dire des expressions dont les valeurs sont déterminées lors de la compilation ou de l'édition de lien.

Conseil

L'instruction « Switch » est très utile pour la lisibilité du programme, car elle permet d'éviter, dans le cas de test d'égalités pour la valeur d'une expression, des if... else... imbriqués.

Exemple : Autre exemple (une petite calculatrice simple)

```
#include < stdio.h >
void main( void ) {
float x,y, res;
char op; int impos=0;
/* Lecture de l'opération et des deux nombres */
printf("Opération(+ - * /): ");
scanf("%c", &op);
printf("Premier opérande: ");
scanf("%f", &x);
printf("Deuxième opérande: ");
scanf("%f", &y);
/* Suivant le caractère op, une addition, une soustraction,
une multiplication ou une division est réalisée */
switch (op)
{
      case '*': res = x * y; break;
      case '/': if(y!=0) res = x / y;
                  else {
                  printf("Division par zéro
impossible !\n");
```

```
                    impos=1;
                    }break;
case '+': res = x + y; break;
case '-': res = x - y; break;
default: impos = 1;
printf("Caractère non reconnu !\n");
}
if( impos ==0)
      printf("Résultat : %f\n", res);
return 0;
}
```

9. Fonctions mathématiques

Complément

Nous pouvons également utiliser comme instructions des fonctions mathématiques.

Ces fonctions sont définies dans le fichier d'en-tête math.h. Voici les plus courantes :

- acos (x) retourne la valeur de arc cosinus(x) dans l'intervalle [0,pi], x doit être compris entre -1 et +1.
- asin (x) retourne la valeur de arc sinus(x) dans l'intervalle [-pi/2,pi/2], x doit être compris entre -1 et +1.
- atan (x) retourne la valeur de arc tangente(x) dans l'intervalle [-pi/2,pi/2].
- atan2 (x ,y) retourne la valeur de arc tangente(y/x) dans l'intervalle [-pi,pi].
- ceil (x) retourne le plus petit entier supérieur ou égal à x.
- cos (x) retourne le cosinus de x.
- cosh (x) retourne le cosinus hyperbolique de x.
- exp (x) retourne exponentiel de x.
- fabs (x) retourne la valeur absolue de x (x est un décimal).
- floor (x) retourne le plus grand entier inférieur ou égal à x.
- fmod (x,y) retourne le reste réel r de x/y, tel que x=iy+r, ou i est un entier, r a le même signe que x, et r est en valeur absolue inférieur à y. Si y=0, il y a overflow et la fonction retourne 0.
- log (x) retourne le logarithme néperien de x.
- log10 (x) retourne le logarithme décimal de x.
- pow (x,y) retourne la valeur de x à la puissance y.
- sin (x) retourne la valeur de sinus de x.
- sinh (x) retourne la valeur de sinus hyperbolique de x.
- sqrt (x) retourne la racine carrée de x.
- tan (x) retourne la valeur de tangente (x).
- tanh (x) retourne la tangente hyperbolique de x.

Elles retournent toutes des valeurs de type double et leurs arguments doivent être

10. Exercices de révision (QCM)

Exercice 1

1. Dans une structure if...

- ○ Les parenthèses encadrant la condition logique sont obligatoires.
- ○ Le mot clé « else » est obligatoire.
- ○ La condition, énoncée juste après if, est suivie d'un point virgule.

Exercice 2

2. En programmation en langage C, quel signe utilise-t-on pour l'affectation ?

- ○ =
- ○ :=
- ○ ==

Exercice 3

3. En programmation en langage C, quel signe utilise-t-on pour le test d'égalité ?

- ○ =
- ○ :=
- ○ ==

Exercice 4

4. Quelle est la position du curseur après l'exécution de la commande "scanf" ?

- ○ A la fin de la ligne.
- ○ Au début de la ligne suivante.
- ○ A la fin du texte affiché ou entré.
- ○ Au début de la ligne courante.

Exercice 5

5. L'instruction «switch» sert à éviter des instructions :

- ○ while... imbriquées.
- ○ do... while imbriquées.
- ○ if... else... imbriquées.
- ○ for... imbriquées.

Exercice 6

6. Indiquer le résultat de l'expression logique suivante :
(A==A) && (B==B)

- ○ Vrai
- ○ Faux

Exercice 7

7. Indiquer le résultat de l'expression logique suivante :
(A!=A) && (B==B)

- ○ Vrai
- ○ Faux

Exercice 8

8. Indiquer le résultat de l'expression logique suivante :
(A!=A) && (B!=B)

- ○ Vrai
- ○ Faux

Exercice 9

9. Indiquer le résultat de l'expression logique suivante :
(A==A) || (B==B)

- ○ Vrai
- ○ Faux

Exercice 10

10. Indiquer le résultat de l'expression logique suivante :
(A==A) || (B!=B)

- ○ Vrai
- ○ Faux

Exercice 11

11. Indiquer le résultat de l'expression logique suivante :
(A!=A) || (B!=B)

- ○ Vrai
- ○ Faux

E. Les boucles

Considérons la liste des cinq problèmes suivants :

Problèmes à traiter

1. Intérêts d'un livret bancaire : Une somme d'argent (S) est déposée sur un livret bancaire. Ce livret procure un intérêt annuel (intérêt exprimé en %, exemple intérêt=2 pour 2%). Nous cherchons à déterminer quel sera le solde du livret après un certain nombre d'années (nb_annee).
2. Le calcul de n! la factorielle d'un nombre entier n positif : nous savons que par définition factorielle de zéro vaut 1 (0!=1) et que n! = n*(n-1)*(n-

2)*(n-3)*...*(2)*(1). Pour un nombre n nous pouvons aussi écrire n! = 1*2*3*4*5*6*7*...*(n-1)*n.

3. Le nombre de chiffres : soit n1 un nombre entier positif ou nul. Nous voulons déterminer le nombre de chiffres qui le compose.
4. Nous désirons effectuer la saisie d'un nombre n compris dans l'intervalle [a, b]. Si le nombre n saisi n'est pas dans cet intervalle un message d'erreur sera affiché et la saisie recommencera.
5. Les vases communicants : soit n1 et n2 deux nombres entiers positifs. Nous voulons faire en sorte que n1 soit égal à n2 en appliquant une méthode de transvasement d'une unité à chaque fois. Si n1 est plus grand que n2, nous retranchons 1 à n1 et ajoutons 1 à n2, et si n2 est plus grand que n1, nous retranchons 1 à n2 et ajoutons 1 à n1. L'objectif est de compter le nombre total de transvasements par cette méthode. Mais est-ce toujours possible d'avoir n1 égal à n2 ?

Pour traiter ces problèmes nous avons besoin d'effectuer plusieurs fois de suite des instructions sur des données qui vont évoluer. Pour cela nous utilisons des structures algorithmiques particulières : les boucles.

Après avoir introduit les concepts liés aux boucles et les syntaxes nécessaires au traitement de ces boucles, nous fournirons les solutions à ces 5 problèmes en fin de chapitre.

1. Définition

Définition

Une boucle effectue plusieurs fois une suite d'instructions (ou un bloc d'instructions). Nous disons qu'une boucle itère un traitement. Une itération correspond à la réalisation du traitement. Nous pouvons formellement numéroter chaque itération. Par exemple, à la troisième itération de la boucle, le traitement (le bloc d'instructions placées dans la boucle) est effectué pour la troisième fois. Quand toutes les itérations sont terminées, le traitement qui suit la boucle est effectué.

Attention

En algorithmique nous distinguons deux type de boucles :

- Les boucles à bornes définies (ou dites déterministes)
- Les boucles à bornes indéfinies (ou dites indéterministes)

Remarque

Dans les deux cas, trois questions de fond se posent systématiquement :

- quelles sont les valeurs des variables avant d'effectuer la boucle?
- quelles variables et quelles valeurs permettent d'arrêter la boucle?
- comment faire évoluer la valeur de ces variables ?

Les variables qui sont utilisées pour déterminer l'arrêt de la boucle sont appelées variables de contrôle de boucle.

2. Boucles à bornes définies

Fondamental

Nous considérons formellement :

- une variable de contrôle de boucle (notée i pour l'explication)
- une borne inférieure (notée valeur_inf pour l'explication)
- une borne supérieure (notée valeur_sup pour l'explication)

- une instruction d'initialisation de la variable de contrôle de boucle (i)
- une condition logique entre la variable de contrôle de boucle i et la borne inférieure (valeur_inf) ou entre la variable i et la borne supérieure (valeur_sup)
- une instruction faisant évoluer la valeur de la variable de contrôle de boucle (i)

Exemple

Nous supposons que la variable i a été préalablement correctement déclarée.

Nous souhaitons écrire un programme qui affiche les trois premiers chiffres : 0, 1, 2.

Pour cela, dans le langage algorithmique, nous écrivons :

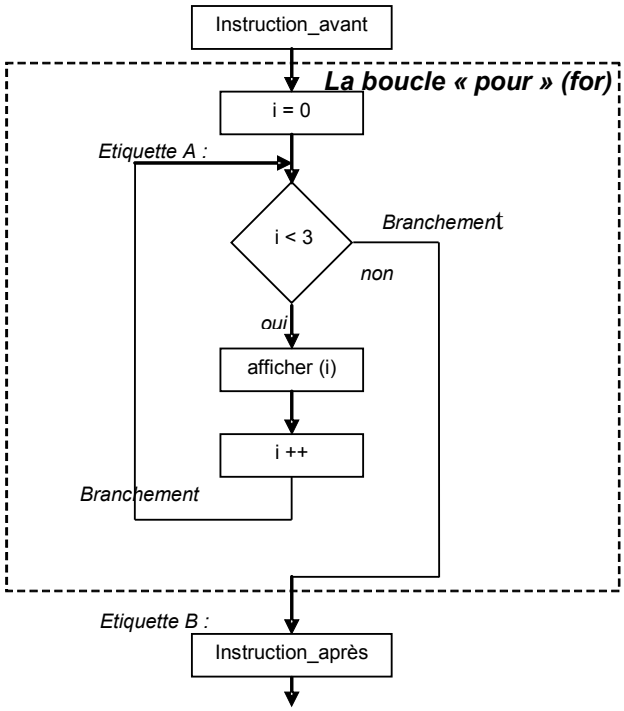
```
instruction_avant
pour i variant de 0 à 2 avec un pas de 1
      afficher(i)
instruction_après
```

Cette boucle utilise implicitement les notions d'étiquette, de branchement conditionnel et de branchement inconditionnel vers une étiquette.

De façon équivalente, on pourrait écrire :

```
                                    instruction_avant
                                   i = 0
Étiquette 1 :          si (i<3)
                                                  alors

afficher(i)

i = i + 1

aller à Etiquette 1
                                      sinon

aller à Etiquette 2
Étiquette 2 :
                                    instruction_après
```

Pour décrire graphiquement le fonctionnement de cette boucle, nous introduisons l'organigramme suivant qui permet de suivre le déroulement des instructions :



Graphique 2 Schéma de la boucle for

Dans un organigramme, une instruction est placée dans un rectangle. Un losange correspond à un test. Le résultat du test vaut soit vrai soit faux. Les flèches indiquent très clairement l'enchaînement des instructions. La lecture de l'organigramme se fait du haut vers le bas. La boucle "pour" correspond au rectangle dessiné en pointillé.

« instruction_avant » est l'instruction exécutée avant la boucle.

« instruction_après » est l'instruction exécutée après la boucle.

La variable de contrôle de boucle a pour identificateur i, la valeur 0 correspond à la borne inférieure (valeur_inf), la valeur 3 correspond à la borne supérieure (valeur_sup), l'instruction d'initialisation est i =0, la condition logique est i<3 et enfin l'instruction faisant évoluer la valeur de la variable de contrôle de boucle i est i = i+1 (car nous avons précisé que le pas était de 1). Le bloc d'instructions exécutées à chaque itération est ici réduit à une seule instruction "afficher" qui affiche la valeur de la variable de contrôle de boucle i.

Syntaxe

Ceci est conforme à la syntaxe :

```
instruction_avant
pour i variant de 0 à 2 avec un pas de 1
          debut
                    afficher(i)
          fin
instruction_après
```

Méthode

Le fonctionnement de la boucle "pour" se comprend très facilement si l'on s'appuie

sur l'organigramme : L'instruction « instruction_avant » est exécutée. Puis la boucle « pour » commence. La valeur 0 est affectée à la variable de contrôle de boucle i.

- La condition logique i<3 (i vaut 0) est évaluée, le résultat du test vaut "vrai". Le bloc d'instructions s'effectue et i=0 s'affiche à l'écran. La première itération est terminée. L'instruction i=i+1 faisant évoluer la valeur de la variable de contrôle de boucle est effectuée, i vaut 1. Un branchement est automatiquement effectué pour que la condition logique soit à nouveau évaluée.
- La condition logique i<3 est évaluée (i vaut 1), le résultat du test vaut "vrai". Le bloc d'instructions s'effectue et i=1 s'affiche à l'écran. La seconde itération est terminée. L'instruction i=i+1 fait évoluer la valeur de la variable de contrôle de boucle, i vaut 2. Un branchement est automatiquement effectué pour que le test soit à nouveau évalué.
- La condition logique i<3 est évaluée (i vaut 2), le résultat du test vaut "vrai". Le bloc d'instructions s'effectue et i= 2 s'affiche à l'écran. La troisième itération est terminée. L'instruction i=i+1 fait évoluer la valeur de la variable de contrôle de boucle, i vaut 3. Un branchement est automatiquement effectué pour que le test soit à nouveau évalué. • La condition logique i<3 est évaluée (i vaut 3), le résultat du test vaut "faux". Un branchement est automatiquement effectué pour exécuter la suite du programme. La boucle « pour » est terminée. « instruction_apres » est exécutée.

Complément

La boucle effectue simplement l'affichage de la variable de contrôle de boucle i. Cette variable prend successivement les valeurs 0, 1, 2, 3. Mais l'extrait de code affiche uniquement les valeurs : 0, 1 et 2.

Vous remarquerez que pour terminer la boucle, la condition logique i<3 doit être fausse. Ce qui signifie que i vaut 3 après la boucle car sa valeur n'a pas changé après la dernière instruction i=i+1.

Vous remarquerez également que l'instruction faisant évoluer la valeur de la variable de contrôle de boucle est effectuée APRES les instructions du bloc et AVANT le branchement. Elle est automatiquement générée à partir de l'écriture "avec un pas de 1" dans la syntaxe de la boucle.

Nous pourrons prendre quelques libertés avec la syntaxe du "pour". Par exemple, si les pas d'incrémentation sont toujours de 1, on pourra simplement écrire :

```
pour i variant de 0 à 2
    debut
        afficher(i)
    fin
```

On dit que ce sont des boucles à bornes définies car on connaît exactement le nombre d'itérations qu'effectuera la boucle. A ce stade, les notations et les bases sont établies et nous allons les utiliser pour présenter les boucles à bornes non définies.

3. Boucles à bornes non définies

Lorsque le nombre d'itérations qu'effectuera la boucle n'est pas connu d'avance ou que le pas est variable, il existe deux autres types de boucles que l'on présente ainsi en langage algorithmique :

Syntaxe : Syntaxe de la boucle 1 :

Syntaxe de la boucle 1 :

```
tant que <condition logique est vraie> faire
<bloc d'instructions>
```

Syntaxe : Syntaxe de la boucle 2 :

```
faire
<bloc d'instructions>
Tant que <condition logique est vraie>
```

Dans les deux cas la <condition logique> permet de tester des valeurs de variables.

Dans le cas de la boucle « tant que ... faire », si la <condition logique> est initialement fausse, le bloc d'instructions ne sera pas effectué. En effet, le test est effectué avant de déclencher l'exécution du bloc d'instructions.

Dans le cas de la boucle « faire ... tant que », le bloc d'instructions est effectué avant le test. Le bloc sera donc exécuté au moins une fois.

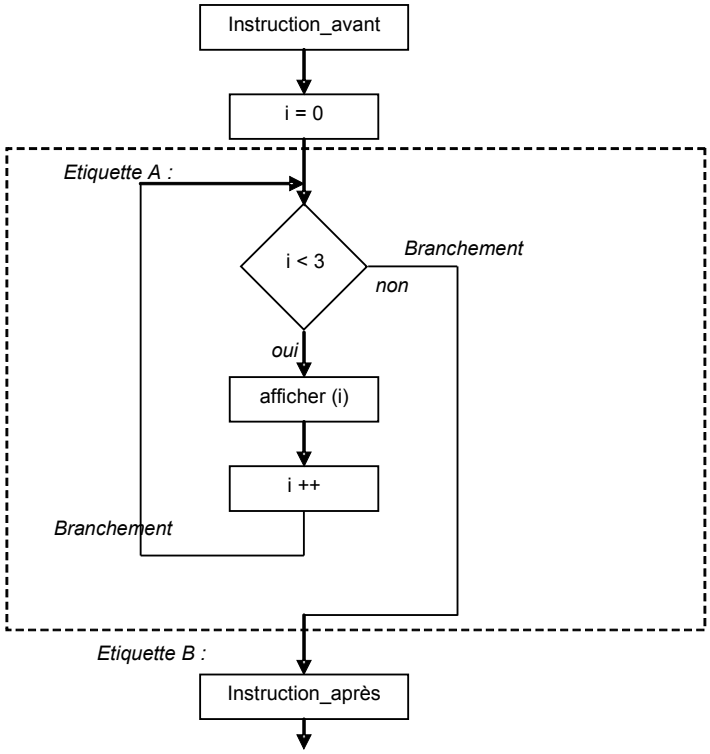
Exemple

Traitons de nouveau l'affichage des valeurs 0, 1 et 2 avec ces deux nouvelles boucles.

Solution boucle « tant que ... faire »:

```
i=0 ;
tant_que (i<3)
debut
afficher(i)
i = i+1
fin
instruction_après
```

Présentation de l'organigramme associé :



Graphique 3 Schéma de la boucle while

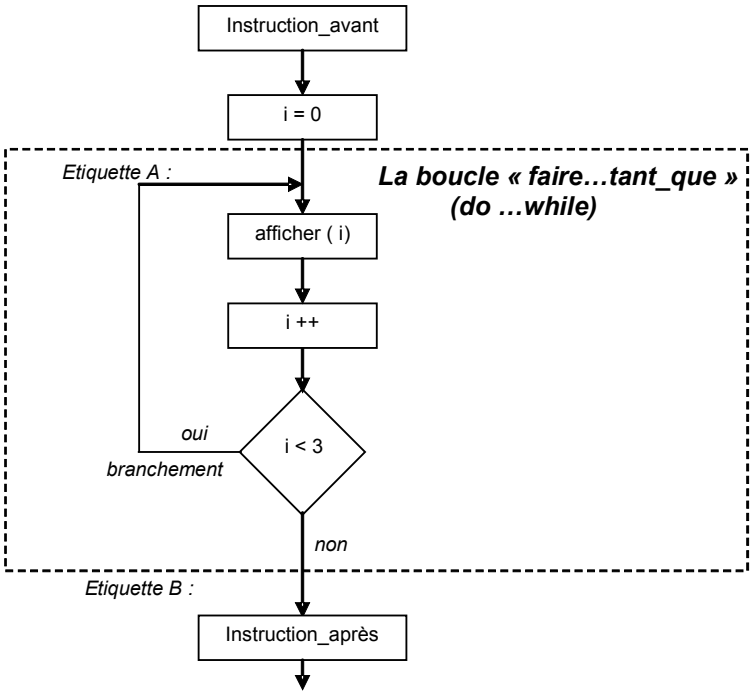
Pour exécuter cette boucle à la main (instruction par instruction), il suffit de suivre l'organigramme, de dessiner un rectangle (formellement un réceptacle), de le nommer i (identificateur) et d'y noter/effacer les valeurs successives de la variable i (identificateur, réceptacle).

Nous affichons 0, 1 et 2. Après la boucle, i vaut 3 comme dans le cas de la boucle « pour ». La boucle correspond au rectangle en pointillé. Les rectangles de « pour » et « tant_que faire » ne recouvrent pas les mêmes éléments.

Dans le cas de « tant que ... faire » nous remarquons que l'instruction faisant évoluer la valeur de la variable de contrôle de boucle fait désormais partie intégrante du bloc d'instructions exécutées à chaque itération de la boucle « tant_que ».

Le programmeur doit écrire des instructions pour faire évoluer les valeurs des variables sur lesquelles porte le test de sortie de boucle.

Solution du même problème avec une boucle "faire ... tant que" avec syntaxe algorithmique :



Graphique 4 boucle do

```
instruction_avant i=0 ; faire début afficher(i) i = i + 1
fin tant_que (i<3) instruction_après
```

Présentation de l'organigramme associé :

Là encore une exécution à la main permet d'en appréhender le fonctionnement. Nous affichons 0, 1 et 2. Après la boucle i vaut 3 comme dans le cas de la boucle "pour".

4. Instructions « for »

Définition

Les boucles en C utilisent des conditions d'arrêt.

Ce sont des expressions qui s'évaluent à vrai ou faux en utilisant la convention logique du C que nous rappelons : 0 signifie faux, autre chose que 0 signifie vrai. Soit l'extrait d'un code écrit en C qui utilise la boucle à bornes définies "for" pour afficher les trois premiers chiffres :

```
instruction_avant for (i=0;i<3;i++) // (rappel : i++ est la
post_incrémentation) { printf("\n i=%4d",i); }
instruction_après
```

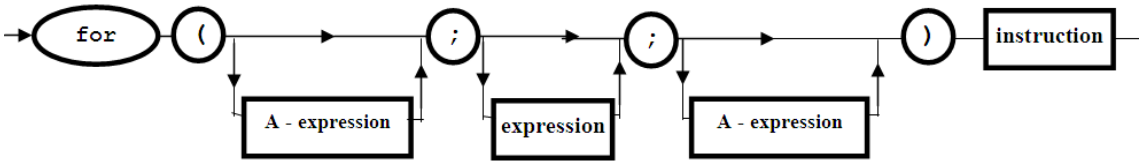
```
for (i=0;i<3;i++) // (rappel : i++ est la
post_incrémentation) printf("\n i=%4d",i);
instruction_après
```

```
instruction_avant for (i=2;i>=0;i--)
{
printf("\n i=%4d",i);
}
instruction_après
```

Le bloc d'instructions commence par "{" et se termine par "}", il s'agit d'une instruction composée qui ne comporte ici qu'une seule instruction « printf ». Nous

tenons à préciser que l'utilisation des accolades est facultative quand la boucle ne comporte qu'une seule instruction. On peut donc écrire :

La syntaxe for (i=0;i<3;i++) correspond au diagramme de Conway suivant :



Graphique 5 Diagramme de Conway de la boucle for

Entre la parenthèse ouvrante "(" et fermante ")", nous retrouvons dans l'ordre :

- une instruction d'initialisation de la variable de contrôle de boucle i
- une condition logique qui teste la valeur de la variable de contrôle de boucle i
- une instruction qui fait évoluer la valeur de la variable de contrôle de boucle i

Ces éléments sont séparés par le délimiteur « ; ».

De fait, si l'on désire afficher les valeurs 2, 1, 0, au cours d'un programme, il suffit d'écrire :

Pour sortir de la boucle, le test i>=0 doit être évalué à faux, donc lorsque i vaut -1.

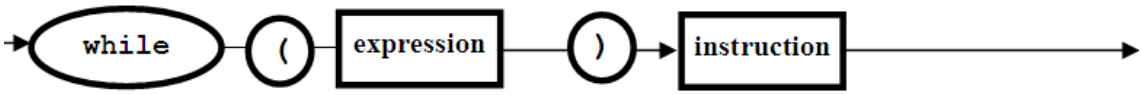
Le compilateur identifie la syntaxe de la boucle « for » et s'occupe de la gestion des étiquettes et des branchements. Ceci est entièrement transparent pour le programmeur.

5. Instruction « while»

Si l'on traite le même problème avec l'équivalent d'une boucle « tant_que faire » en C, on obtient :

```
instruction_avant
i=0 ;
while (i<3)
{
printf("\n i=%4d",i);
i++;
}
instruction_après
```

La syntaxe « while (i<3) » correspond au diagramme de Conway suivant :



Graphique 6 Digramme de Conway de la boucle while

La paire de parenthèses autour de la condition logique (expression) est obligatoire en C.

Examinons maintenant le code :

```
instruction_avant
i=0 ;
while (i<3)
{
i++;
printf("\n i=%4d",i);
}
instruction_après
```

L'exécution de ce code affichera 1, 2, 3 et non pas 0, 1 et 2.

Et i vaut 3 en sortie de boucle.

Dans le cas d'une boucle tant_que (while), il est de la responsabilité du programmeur de décider des emplacements des instructions qui font évoluer les valeurs des variables impliquées dans la condition d'arrêt (le test d'arrêt) au sein du bloc d'instructions effectuées à chaque itération.

Examinons le code suivant :

```
instruction_avant i=-1 ;
while (i<2)
{
i++;
printf("\n i=%4d",i);
}
instruction_après
```

L'exécution de ce code affichera 0, 1, 2 mais i vaut 2 en sortie de boucle.

Les valeurs initiales des variables utilisées dans la boucle a un impact sur son fonctionnement.

6. Instruction « do... while »

Si l'on traite le même problème avec l'équivalent en langage C de la boucle "faire ... tant que", nous obtenons :

```
instruction_avant i=0 ;
do
{ printf("\n i=%4d",i); i++; }
while (i<3);
instruction_après
```

La syntaxe du do ...while (i<3) correspond au diagramme de Conway suivant :

Graphique 7 Le diagramme de Conway de la boucle do

La paire de parenthèses autour de la condition logique (expression) est obligatoire en C.

L'initialisation des variables avant la boucle et l'emplacement des instructions ont, là encore, un impact sur le déroulement de la boucle.

7. Boucles imbriquées

Les boucles sont des instructions. Une boucle est une instruction qui exécute une ou plusieurs instructions (dans ce dernier cas on a un bloc d'instruction). Nous pouvons donc imbriquer plusieurs boucles.

Exemple

```
int i,j ; for (i=1;i<10;i++) { for (j=1;j<10;j++)
{ printf("%3d",i*j); } printf("\n"); }
```

Affichera :

1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81

La boucle sur « j » est appelée boucle interne. Elle affiche le produit « i*j » sur une ligne.

La boucle sur « i » est appelée boucle externe. Elle affiche donc une ligne (boucle sur j) puis effectue un saut de ligne.

On peut donc imbriquer autant de boucles (« for », « while », « do ») que l'on veut.

8. Choix de la boucle

Exemple

Nous disposons de trois types de boucle "pour", "tant que ... faire" et "faire ... tant que".

Il n'y a pas de règles pour choisir le type de boucle à utiliser.

En effet, l'exemple de l'affichage des trois premiers chiffres (0, 1 et 2) le montre : on peut trouver des formulations équivalentes avec, éventuellement, des tests supplémentaires.

Par exemple, on peut fabriquer une boucle "while" avec une boucle "do" :

```
if (condition) do
bloc_instructions
while (condition)
```

Puisqu'un test supplémentaire est utilisé avant de lancer la boucle "do", si ce dernier est faux alors la boucle ne sera pas effectuée. Ce qui revient à avoir une boucle "while".

Exemple

Autre exemple, si l'on veut faire une boucle "do" avec une boucle "while" :

```
bloc_instructions
while (condition)
bloc_instructions
```

En ajoutant une copie du bloc d'instructions avant la boucle "while", on a la garantie qu'il est exécuté au moins une fois, ce qui revient à avoir une boucle "do". Cependant, il est plus judicieux de faire le bon choix dès le départ, et de choisir des formes de boucles qui facilitent la lisibilité du programme.

Complément

Revenons sur les cinq exemples introduits au début du chapitre.

1. Intérêts d'un livret bancaire : les données sont la somme d'argent « S », le taux d'intérêt « interet », et le nombre d'année « nb_annee ». Le nombre d'itérations est fixé par le nombre d'années, il est connu, il faut utiliser une boucle « for ».
2. Le calcul de la factorielle de n : la donnée est le nombre « n » qui est connu il faut utiliser une boucle « for ».
3. Nous désirons effectuer la saisie d'un nombre « n » compris dans l'intervalle [a,b]. La donnée est le nombre « n ». On ne sait pas combien de saisies seront nécessaires pour avoir le nombre dans l'intervalle souhaité, il faut utiliser une boucle à bornes non définies. Il faut effectuer au moins une saisie, il vaut mieux utiliser une boucle "do".
4. Le nombre de chiffre : la donnée est « n », le nombre. Nous ne connaissons pas le nombre de chiffres qui le compose puisque c'est ce que l'on recherche, il faut utiliser une boucle à bornes non définies. Si initialement « n » ne comporte qu'un unique chiffre aucune itération n'est nécessaire, il vaut mieux utiliser une boucle « while ».

5. Les vases communicants : les données sont « n1 » et « n2 ». Il faut arrêter quand « n1 » et « n2 » sont égaux. Tel que le problème est formulé, il faut utiliser une boucle à bornes non définies car on ne sait pas combien de fois nous allons transvaser. Si initialement « n1 » est égal à « n2 » aucune itération n'est nécessaire, il vaut mieux utiliser une boucle while.

La section suivante vous offre quelques conseils pour choisir judicieusement le type de boucle à appliquer.

9. Conseils

Attention

Il faut prendre garde aux points suivants :

- Faire attention aux conditions initiales, aux conditions d'arrêt et à l'évolution des variables impliquées dans le test de sortie.
- Pour mettre au point la boucle, il faut savoir exécuter la boucle à la main (simuler l'exécution de l'algorithme ou du programme, instruction par instruction), en faisant évoluer les valeurs des variables.
- Les instructions contenues dans une boucle « while » ou « do...while » doivent permettre de modifier les variables impliquées dans le test d'arrêt de telle sorte que la boucle se termine. Pour la boucle « for », il ne faut pas que l'exécution des instructions de la boucle change la valeur de l'incrément. En effet, un tel changement risque de rendre la boucle non déterministe.

Pour le point 1, nous avons mis en évidence l'influence de ces éléments dans la présentation des boucles à bornes non définies en C en présentant plusieurs exemples avec la boucle while.

Pour le point 2, tant que l'on ne maîtrise pas le fonctionnement des boucles, il est utile d'effectuer des exemples à la main comme nous l'avons fait. éventuellement, vous pouvez dessiner un organigramme pour vous aider à dérouler la boucle que vous avez construite.

Le point 3 correspond à une erreur classique dont nous donnons un exemple simple ci-après.

Exemple

Considérons l'exemple suivant :

```
i=0 ;
while (i<3)
{ printf("\n i=%4d",i); }
```

Le programme reste bloqué dans une boucle infinie. En effet, i vaut 0 avant d'entrer dans la boucle. La valeur de i n'est jamais modifiée dans le corps de la boucle. La valeur reste à 0 et comme la condition logique évalue si i<3, elle est toujours évaluée à vrai.

La boucle est infinie.

Un exemple plus compliqué vous est fourni dans la proposition de modification de la solution du problème 5 (transvasement). Pour bien le comprendre, il faut suivre précisément les directives.

10. Solutions des problèmes en langage C

Nous vous donnons les solutions des cinq problèmes posés. Etudiez-les attentivement selon deux aspects :

- **algorithmique** : comment ces solutions répondent au problème posé
- **syntaxique** : comment la solution a été formulée en langage C

a) Intérêts d'un livret bancaire

Exemple

```
#include <stdio.h>
int main()
{
        float S, interet;
        int i, nb_annee;
        printf("\nentrer la somme ");
        scanf("%f",&S);
        printf("enter le taux, exemple : 3.5 pour 3.5%% ");
                        // pour pouvoir écrire « % » à
l'écran, il faut utiliser
                        // « %% ». si on écrit juste « % »
l'unique signe « % »
                        // n'est pas affiché, car le signe
« % » unique est
                        // interprété comme une commande de
formatage.
        scanf("%f",&interet);
        printf("entrer le nombre d'annees : ");
        scanf("%d",&nb_annee);
        for (i=1;i<=nb_annee;i++)
        {
                S = S + (S*interet)/100.0;
                printf("\nau bout de %d annees vous avez
%5.2f ",i, S);
        }
        printf("\n\nau bout de %d annees vous avez %5.2f
",nb_annee, S);
        printf("\n\n");
}
```

Exemple de sortie écran

entrer la somme 100
enter le taux, exemple : 3.5 pour 3.5% 2.5
entrer le nombre d'annees : 3
au bout de 1 annees vous avez 102.50
au bout de 2 annees vous avez 105.06
au bout de 3 annees vous avez 107.69
au bout de 3 annees vous avez 107.69

b) Le calcul de la factorielle de n

Exemple

```
#include <stdio.h>
int main()
{
        int i, n, resultat;
        printf("\nentrer un nombre entier positif ");
```

```
        scanf("%d",&n);
        resultat = 1; // car 0!=1
        for (i=2;i<=n;i++)
        {
                resultat = resultat*i; //resultat = 1 * 2 *
3 * ...
        }
        printf("\n\nla factorielle de %d est %d
",n,resultat);
        printf("\n\n");
}
```

Exemple de sortie écran

entrer un nombre entier positif 5
la factorielle de 5 est 120

c) Nous désirons effectuer la saisie d'un nombre n compris dans l'intervalle [a,b]

Exemple

```
#include <stdio.h>
#define a 2
#define b 36
int main()
{
        int n;
        printf("\nL'intervalle est [%d, %d]",a,b);
        printf("\nce sont des constantes definies avec
#define\n");
        do
        {
                printf("\nentrer un nombre entier positif
ou nul ");
                scanf("%d",&n);
                if ((n<a) || (n>b))
                        printf("\npas dans l'intervalle
[%d, %d]\n",a,b);
        }
        while ((n<a) || (n>b));
        printf("\n\nLe nombre saisi est %d ",n);
        printf("\n\n");
}
```

Exemple de sortie écran

L'intervalle est [2, 36]
ce sont des constantes definie avec #define
entrer un nombre entier positif ou nul 0
pas dans l'intervalle [2, 36]
entrer un nombre entier positif ou nul 10
Le nombre saisi est 10

d) Le nombre de chiffres

Exemple

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

```
int main()
{
        int nb_chiffre, n, inter;
        printf("\nentrer un nombre entier positif ou nul
");
        scanf("%d",&n);
        nb_chiffre = 1; // par défaut on a un chiffre même
pour 0
        inter = n;
        while ((inter / 10) != 0) // si c'est vrai il n'y a
qu'un chiffre et on arrête
        { // Puisque “inter” et 10 sont entiers le signe
“/” correspond ici à la division entière
                inter = inter/10;
                nb_chiffre++;
        }
        printf("\n\nil y a %d chiffre(s) dans %d
",nb_chiffre,n);
        printf("\n\n");
}
```

Exemple de sortie écran :

entrer un nombre entier positif ou nul 65879
il y a 5 chiffre(s) dans 65879

e) Les vases communicants

Exemple

```
#include <stdio.h>
int main()
{
        int i, n1, n2;
        int transvasement;
        printf("\nentrer un nombre entier positif ");
        scanf("%d",&n1);
        printf("\nentrer un nombre entier positif ");
        scanf("%d",&n2);
        transvasement = 0;
        while ((n1 != n2)&&(n1+1!=n2)&&(n1!=n2+1)) //
réfléchissez à cette condition d'arrêt.
        {
                if (n1>n2)
                {
                        n1--;
                        n2++;
                }
                else
                {
                        n1++;
                        n2--;
                }
                transvasement++;
        }
        printf("\n\nil y a eu %d transvasements et n1 = %d
et n2 = %d ", transvasement,n1,n2);
        printf("\n\n");
}
```

Exemples de sortie écran :

Exemple 1:
entrer un nombre entier positif 8
entrer un nombre entier positif 2
il y a eu 3 transvasements et n1 = 5 et n2 = 5
Exemple 2 :
entrer un nombre entier positif 8
entrer un nombre entier positif 3
il y a eu 2 transvasements et n1 = 6 et n2 = 5

Attention : Question

si l'on change le test ((n1 != n2) && (n1+1!=n2) && (n1!=n2+1)) par (n1 != n2) que se passe-t-il avec n1=7 et n2=3, puis avec n1 = 7 et n2 = 4 ?
Exécutez à la main le code pour répondre, vous verrez (voir conseils point 3).

11. Autres Exemples de boucles

Exemple

Nous avons vu des boucles qui travaillaient avec des entiers. Voici un exemple qui affiche la portion de la table des codes ASCII des caractères contenus entre les valeurs 65 pour le 'A' et 90 pour le 'Z'.
Nous utiliserons ici la conversion implicite d'un caractère en un entier en se basant sur son code ASCII.

```
#include <stdio.h>
#define a 65
#define b 91
int main()
{
        char c;
        for (c=a;c<b;c++)
        {
                printf("\nLe caractere %c ",c);
                printf("valeur decimale %3d ",c);
                printf("valeur hexadecimale %2X",c);
        }
        printf("\n\n");
}
```

Ce qui finalement revient à avoir une boucle sur des entiers.

Exemple

Voici une autre boucle qui affiche les lettres minuscules :

```
#include <stdio.h>
int main()
{
        char c;
        for (c='a';c<='z';c++)
        {
                printf("\nLe caractere %c ",c);
                printf("valeur decimale %3d ",c);
                printf("valeur hexadecimale %2X",c);
        }
        printf("\n\n");
```

```
}
```

Exemple

Enfin, une boucle qui utilise les fonctions sin et cos de la bibliothèque mathématique pour afficher les sinus est cosinus des angles entre 0 et PI avec un pas de PI/12 :

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159265358979323846
int main()
{
        float angle, deg_angle, val_sinus, val_cosinus,
incre_angle;
        angle = 0.0; // l'angle initial est 0
        incre_angle = PI/12; // Valeur de l'incrément de
l'angle
        do
        {
                val_sinus = sin(angle);
                val_cosinus = cos(angle);
                deg_angle = (angle*180)/PI; //conversion de
l'angle en degree pour affichage
                printf("\nangle en degre %5.1f ",
deg_angle);
                printf("sinus %5.6f ", val_sinus);
                printf("cosinus %5.6f ", val_cosinus);
                angle = angle + incre_angle; //
incrémentation de la valeur de l'angle
        } while (angle<PI); //quand « angle >= PI » la
boucle s'arrête.
        printf("\n\n");
}
```

Remarque

Nous supposons que ce fichier source s'appelle « table_sinus.c ». Sous un système Unix ou Linux la commande de compilation devra être :

gcc table_sinus.c - o table_sinus –lm

L'option « -lm » est nécessaire pour la librairie mathématique (<math>).

12. Instruction continue

Définition

C'est une instruction de branchement automatique que l'on utilise au sein d'une boucle.

Si on exécute l'instruction "continue" au sein d'une boucle "do" ou "while" alors la prochaine itération recommence au niveau de l'évaluation de l'expression de la condition d'arrêt.

Exemple : Exemple avec "do"

```
i = 2;
do
{
        i++;
```

```
        if (i==5)
        continue;
        printf("\n i = %d",i);
} while (i<7);
```

Cet exemple affichera :

i = 3

i = 4

i = 6

i = 7

En effet, quand "i" vaut 5, le code exécute l'instruction "continue" qui "saute" le "printf" et va se brancher au niveau du test "while (i<7)" pour évaluer la condition d'arrêt.

Exemple : Exemple avec "while"

```
#include <stdio.h>
int main(){
        int i,j,x;
        x = 1;
        i = 5;
        j = 0;
// Comme nous utilisons la post décrémentation « i - - »,
nous effectuons d'abord la comparaison i<0 puis
// la décrémentation i=i - 1.
        while ( (i--) > 0 )
        {
                x += 1;
                if ( x % 2 )
                continue;
                j += x * x;
        }
        printf("\nen sortie j = %d\n",j);
}
```

Cet exemple affichera en sortie

j = 56

La boucle s'effectue pour « i = 5,4,3,2,1 » et « x » vaut respectivement 2, 3, 4, 5, 6 avant le « if (x % 2) ».

Si « x %2 vaut 0 » alors « x » est pair et on fait l'instruction « j += x * x; ».

Si la valeur de l'expression « x %2 » est différente de 0 (x est impair) alors le code exécute le « continue ».

L'instruction « j += x * x; » est « sautée » et on calcule « (i--) > 0 ».

Ce code C effectue la somme « 2*2+4*4+6*6 = 4+16+36=56 ».

Rappel

Nous vous rappelons qu'une boucle for s'écrit comme un triplet for (partie1; partie2 ; partie3).

Si on exécute l'instruction "continue" au sein d'une boucle "for" alors toutes les instructions de la boucle qui suivent le "continue" sont "sautées", puis "partie3" du triplet est exécutée, puis "partie2", ce qui permet de déterminer si la boucle doit recommencer.

Exemple : Autre exemple

```
for (i=0; i<10;i++)
{
        if ((i%3)==0) continue;
        printf("\ni = %d ",i);
}
printf("\nsortie : ");
printf("\n\ti = %d ",i);
```

Ce morceau de code C affiche :

i = 1

i = 2

i = 4

i = 5

i = 7

i = 8

Sortie:

i = 10

La boucle for s'effectue pour « i = 0,1,2,3,4,5,6,7,8,9 ».

Si « i » est divisible par 3, c'est-à-dire que « (i%3)==0 », alors on effectue l'instruction « continue », ce qui va directement effectuer l'instruction « i++ » qui correspond à « partie3 », puis le test « i<10 » qui correspond à « partie2 » est effectué.

Cette boucle n'affiche pas les multiples de 3.

13. L'instruction break

Définition

C'est une instruction de branchement automatique en fin de boucle "do", "while", "for", ou en fin de "switch" (nous avons déjà vu le fonctionnement dans le cas du "switch").

L'instruction break termine l'exécution de l'instruction englobante "do", "while", "for",, ou "switch" dans laquelle elle apparaît. L'instruction qui est alors exécutée est celle qui suit immédiatement la fin du "do", "while", "for", ou "switch"

Exemple

```
int i;
int j;
for(i=0;i<100;i++)
{
        printf("\ni = %d ",i);
        printf( "\nPour quitter taper -1 : " );
        scanf("%d", &j);
        if (c == -1)
        break;
}
```

Cette boucle va potentiellement afficher tous les nombres de 0 à 99. Nous demandons à l'utilisateur une lettre.

S'il tape 'Q' nous sortons de la boucle même si elle n'est pas terminée (même si i=100 n'est pas atteint).

Attention

Les bonnes pratiques de la programmation structurée bannissent l'utilisation de « break » au sein d'une boucle « for ».

Cet exemple est donné à titre d'illustration uniquement. Dans ce cas il vaut mieux utiliser une autre boucle en testant deux conditions de sortie « (i<100) » et « (j=-1) ».

Exemple

```
#include <stdio.h>
#include <math.h>
#define MAX 100
for (i=MAX/2; i<MAX;i++)
{
        if ((((i%2)==0) || ((i%3)==0))||((i%5)==0))
        continue;
        else
        if (((i%7)==0) && ((i%11)==0)) break;
        printf("\ni = %d ",i);
}
printf("\nsortie : ");
printf("\n\ti = %d ",i);
```

Ce morceau de code affichera :

i = 53

i = 59

i = 61

i = 67

i = 71

i = 73

La boucle se déroule normalement de MAX/2 à MAX-1 inclus. Elle affiche la valeur de la variable de contrôle de boucle i.

Si i est multiple de 2, 3 ou 5 on ne l'affiche pas car le "continue" passe automatiquement à l'itération suivante en effectuant avant l'instruction 'i++'.

Si i est multiple de 7 et 11 (donc i=77, compte tenu de la boucle), alors l'instruction 'break' fait sortir de la boucle.

14. Compléments sur la boucle "for"

Rappel

Nous vous rappelons qu'une boucle for s'écrit comme un triplet for (partie1; partie2 ; partie3) ces trois parties sont optionnelles, ainsi la boucle :

```
for(;;) {
        printf( "\ntaper Q pour quitter la boucle : " );
        scanf("%c", &c);
        if (c == 'Q')
                break;
}
```

est parfaitement compilée. Elle correspond à une boucle infinie dont on ne peut sortir qu'en tapant la lettre 'Q'.

En outre, chaque partie peut comporter plusieurs éléments syntaxiques séparés par le délimiteur ',',

Exemple

```
int i,j,compte=0 ;
for (j=5,i=0; i<10, j>0;i++,--j)
{
        printf("\ncompte = %d i = %d j = %d",compte, i,j);
        compte ++;
}
printf("\nsortie : ");
printf("\n\tcompte = %d i = %d j = %d",compte, i,j);
printf("\n");
}
```

affichera :

compte = 0 i = 0 j = 5

compte = 1 i = 1 j = 4

compte = 2 i = 2 j = 3

compte = 3 i = 3 j = 2

compte = 4 i = 4 j = 1

sortie :

compte = 5 i = 5 j = 0

La boucle ne s'effectue que 5 fois. En effet, la valeur j=0 sera atteinte avant la valeur i=10.

Complément

On peut donc initialiser plusieurs variables dans "partie1", élaborer plusieurs tests séparés dans "partie2" et faire évoluer plusieurs variables de contrôle de boucle dans "partie3".

Conseil : Commentaires et conseils pour "continue", "break" et variantes de "for"

Les instructions "continue", "break" et les variantes de "for" sont à déconseiller.

Nous avons tenu à vous les présenter pour que vous les connaissiez si vous les rencontrez.

Cependant, la grande majorité des applications ne nécessitent pas leur usage, on peut toujours trouver une autre solution.

Il n'y a pas d'algorithme unique pour traiter un problème. Si vous voulez les utiliser apprenez bien leur fonctionnement en complétant cette présentation par la lecture d'ouvrages plus détaillés.

Ce cours introductif donne les bases et ouvre des perspectives, libre à vous de les explorer pour aller plus loin.

F. Tableaux, chaînes, et pointeurs

1. Définition

Définition

Les tableaux permettent de gérer un ensemble de variables de même type.

Un tableau est une zone mémoire constituée de cases contiguës où sont rangées

des données de même type. Les cases ont donc une taille identique. Un tableau possède un nombre fixé de cases qui doit être connu quand il est créé. La zone mémoire possède donc un début et une fin. Pour accéder à une case, nous utilisons un indice qui repère le numéro de la case à laquelle on fait référence.

L'indice d'un tableau est un index, il y a ainsi un lien étroit entre tableau et pointeur comme nous allons le voir dans ce chapitre.

Comme toute variable, une case d'un tableau doit être initialisée avant d'être utilisée.

2. Les tableaux unidimensionnels

a) Déclaration

Syntaxe

La syntaxe de déclaration d'une variable de type tableau est la suivante :

type identificateur [N];

où :

- « type » est le type de variable contenu dans de chaque case du tableau
- « identificateur » est le nom donné au tableau
- « N » est une valeur entière qui donne le nombre total de cases du tableau

La valeur de N doit être connue à la compilation, il n'est pas possible d'utiliser des tableaux de taille variable.

Par contre, nous pouvons décider de n'utiliser qu'une partie des cases d'un tableau.

Exemple : Exemple de déclaration

```
int tableau[3];
```

est une déclaration qui réserve une zone mémoire qui peut être représentée comme suit :

numéro de la case	case 1	case 2	case 3
Contenu de la case	?	?	?
indice pour accéder à la case	0	1	2

Tableau (indice, contenu)

"tableau" est l'identificateur de la variable tableau qui correspond à une zone mémoire où nous pouvons stocker 3 entiers. Elle comporte 3 cases. Chaque case peut contenir une valeur entière de type int.

Il ne faut pas confondre le numéro d'un élément du tableau et l'indice de la case qu'il occupe. Vous remarquerez qu'en langage C les indices de cases commencent à 0. Ainsi, la déclaration int t[3] crée 3 cases dont les indices sont 0, 1, 2. Nous avons bien 3 indices pour repérer les 3 cases.

Le compilateur alloue la zone mémoire, mais n'initialise pas le contenu des cases. C'est pourquoi nous avons indiqué « ? » pour le contenu de chaque case.

Syntaxe

Exemple d'accès à une case du tableau "tableau" :

tableau [0] La valeur 0 correspond à l'indice de la première case du tableau. Nous rencontrons les délimiteurs '[' et ']' qui encadrent une dimension (pour la

déclaration) ou l'indice (pour accéder à une case) d'un tableau comme présenté au chapitre 3.

C'est la technique la plus simple pour accéder à une case du tableau. De façon générique l'accès à une case du tableau respecte la syntaxe suivante :

identificateur_du_tableau [expression entière]

En effet, la valeur d'un indice d'une case peut être le résultat d'un calcul qui renvoie une valeur entière.

Un tableau à une dimension est également appelé vecteur.

Attention

Le nombre de cases d'un tableau doit impérativement être connu à la compilation. En effet, il faut pouvoir réserver la mémoire nécessaire à la zone de données contiguës.

A l'issue de la déclaration, le tableau est créé mais le contenu de ses cases n'est pas initialisé. Par exemple « int truc[3] », déclare un tableau de 3 entiers, mais les entiers truc[0], truc[1] et truc[2] ne sont pas initialisés.

b) Initialisation à la déclaration

Exemple

```
int truc [ 3 ] = {999, 777, 222} ;
```

A droite de l'opérateur d'affectation nous avons une liste de valeurs d'initialisation. Nous retrouvons les délimiteurs '{'et '}' qui servent ici à encadrer les valeurs de la liste. Les valeurs de la liste sont séparées par le délimiteur ','. Les valeurs sont rangées dans l'ordre gauche vers droite dans les cases du tableau 'truc' à partir de la case d'indice 0 (zéro). Suite à cette déclaration, truc[0] contient 999, truc [1] contient 777 et truc [2] contient 222. Toutes les cases du tableau 'truc' sont initialisées.

Si on déclare :

```
int machin [ 10 ] = {9, 7, 2} ;
```

Nous constatons qu'il y a moins de valeurs dans la liste d'initialisation que de cases dans le tableau. Dans ce cas, seules les premières cases seront initialisées. A partir de la case d'indice 3, les valeurs ne sont pas initialisées, par exemple nous ne connaissons pas la valeur de machin[3].

c) Initialisation par affectation

Syntaxe

Nous pouvons simplement écrire :

```
truc [0] = -1;
truc [1] = 10;
truc [2] = truc[1]*truc[1]; // nous avons en ce point
truc[2] = 100
```

Cependant, pour réaliser l'initialisation complète des cases d'un tableau, il est plus judicieux d'utiliser une boucle à bornes définies puisque l'on connaît le nombre maximum de cases.

Exemple

initialisation des cases de "machin" avec la valeur qui correspond à son indice de case :

```
for (i=0;i<10;i++)
{
    machin[i] = i;
}
```

Il faut évidemment que la variable i ait été préalablement déclarée.

La partie droite de l'opérateur d'affectation correspond à une expression qui retourne une valeur qui doit être compatible avec le type d'une case du tableau.

Exemple

Autre exemple :

```
for (i=0;i<10;i++)
{
machin[i] = 10-i;
}
```

Remplira les cases 0,1,2,3,4,5,6,7,8,9 du tableau avec dans l'ordre les valeurs 9,8,7,6,5,4,3,2,1.

d) Initialisation par lecture

Exemple

```
#include <stdio.h>
int main()
{
 int i ;
 float t_r [10] ;
 printf("\nentrer 10 valeurs flottantes") ;
  for (i = 0 ; i < 10 ; i ++)
   {
         scanf(" %f ", &t_r[ i ]);
    }
}
```

« t_r[i] » est une variable de type flottante et &t_r[i] est bien son adresse mémoire. C'est ce dont a besoin le « scanf »pour ranger l'information lue au clavier.

3. Techniques algorithmiques liées aux tableaux

Conseil

L'usage de constantes pour coder le nombre de cases maximales du tableau est conseillé.

Exemple

Considérons le code :

```
#include <stdio.h>
int main()
{
    int machin[10];
   for (i=0;i<10;i++)
     {
           machin[i] = 10-i;
      }
}
```

Remarque

Si nous souhaitions travailler avec un tableau de taille 50, nous aurions besoin de modifier 3 lignes.

Exemple

Considérons maintenant le code :

```
include <stdio.h>
#define MAX 10
int main()
{
    int machin[MAX];
    for (i=0;i<MAX;i++)
     {
         machin[i] = MAX-i;
     }
}
```

Avec ce code, si l'on veut travailler avec un tableau de 50 cases, il suffit de modifier une ligne (le « #define »). Le précompilateur fera le travail de modification à notre place. C'est une technique de programmation qui évite bien des erreurs.

Rappel

Nous rappelons que la taille d'un tableau doit être connue à la compilation et que la taille d'un tableau ne peut être modifiée au cours de l'exécution d'un programme.

Exemple

Soit le problème suivant posé : le nombre d'éléments que l'on veut ranger et utiliser dans un tableau est inconnu, mais on sait qu'il n'y en aura pas plus que 1000.

Dans ce cas, il suffit d'utiliser la technique que nous illustrons dans le code suivant :

```
include <stdio.h>
#define MAX 1000
int main()
{
     int machin[MAX];
     int nb_machin; // le nombre d'entiers rangés dans le
tableau machin
                                    // le reste du code ...
}
```

Nous déclarons le tableau de taille maximale.

Puis on ajoute la déclaration de la variable « nb_machin » qui indique le nombre d'éléments que l'on a effectivement rangés dans le tableau « machin ».

Vous remarquerez le choix effectué pour construire l'identificateur de la variable qui gère le nombre d'entiers rangés dans le tableau.

Bien évidemment, c'est au programmeur d'incrémenter « nb_machin» quand on ajoute un élément et de le décrémenter quand on en retire un.

4. Les tableaux à deux dimensions

a) Problème de la linéarisation de la structure de donnée

Voici un tableau à deux dimensions tel qu'il est présenté en mathématique par exemple :

7	8	9
15	16	17

Exemple

Nous remarquons immédiatement qu'il s'agit d'une donnée qui possède deux dimensions. Il faut un indice de ligne et un indice de colonne pour accéder au contenu d'une case.

Or, la mémoire d'une machine est vue comme un très grand ruban d'octets, c'est une organisation linéaire avec une seule dimension puisque l'on accède à un octet à l'aide de son adresse mémoire. Ce ruban peut être vu comme un tableau à une dimension.

Il faut donc trouver une technique pour linéariser un tableau en deux dimensions afin qu'il puisse être rangé et manipulé dans un tableau à une dimension.

Nous avons deux possibilités :

- linéarisation colonne/colonne
- linéarisation ligne/ligne

Dans le premier cas le tableau est découpé en colonnes, et les colonnes consécutives sont rangées les unes derrière les autres. Dans le second cas le tableau est découpé en lignes, et les lignes sont rangées les unes derrière les autres. Le langage C utilise la deuxième technique.

Le découpage et l'organisation en lignes donne :

7	8	9	15	16	17

Le découpage et l'organisation en lignes

Les valeurs 7, 8, 9, 15, 16, 17 peuvent se ranger les unes à la suite des autres dans la mémoire : nous avons linéarisé le rangement du tableau.

Un tableau à deux dimension est aussi appelé matrice.

b) Déclaration et initialisation à la déclaration

Exemple : Exemple la déclaration

```
#define MAX_L 2
#define MAX_C 3
int t[MAX_L] [MAX_C];
```

Permet de définir la variable "t" comme un tableau à deux dimensions de 2 lignes et 3 colonnes.

Pour accéder à une case il suffit d'utiliser la syntaxe :

```
t[1][2]
```

Ce qui permet d'accéder à la case qui contient un entier qui se trouve en deuxième ligne, troisième colonne du tableau. (les indices commencent à zéro).

Bien que le tableau soit "linéarisé", pour accéder aux cases du tableau « t », l'opération d'indexation s'écrit tout simplement « t[i][j] ». Le compilateur se charge de retrouver la bonne case, cette syntaxe est transparente pour le programmeur.

Exemple : Exemple d'initialisation à la déclaration

```
#define MAX_L 2
#define MAX_C 3
double td[MAX_L] [MAX_C] = {{37.2,37.5},{38.4, 40.5,
43.2}};
```

L'initialisation s'effectue ligne par ligne avec les listes de valeurs d'initialisation. Comme pour les tableaux à une dimension, si une liste n'est pas complète, seules les premières cases sont remplies dans l'ordre de la liste.

Ainsi, sur l'exemple ci-dessus la case td[0][2] n'est pas initialisée car il n'y a pas de troisième valeur dans la liste d'initialisation de la première ligne.

c) Initialisation par affectation

Exemple

En considérant les déclarations :

```
#include <math.h>
#define MAX_L 2
#define MAX_C 3
#define PI 3.14159265
double td[MAX_L] [MAX_C];
int i,j;
```

Nous pouvons écrire :

```
for (i=0;i<MAX_L;i++)
{
     for (j=0;j<MAX_C;j++)
      {
           td[i][j] = sin(PI/4);
       }
}
```

Nous avons initialisé toutes les cases du tableau "td" par calcul. Le type d'une case du tableau doit être compatible avec le type de valeur retournée par l'expression à droite de l'opérateur d'affectation.

d) Initialisation par lecture

Exemple

Considérons :

```
#define MAX_L 2
#define MAX_C 3
int t[MAX_L] [MAX_C];
int i,j;
```

Puisque « t[i][j] »correspond à la variable de type entier rangée aux indices (i,j) du tableau, l'expression « &t[i][j] » retourne son adresse, ce dont à besoin la fonction « scanf » pour lire une valeur au clavier et la ranger au bon format dans la case.

Exemple

Par exemple, le bout de code :

```
for (i=0;i<MAX_L;i++)
{
      for (j=0;j<MAX_C;j++)
```

```
        {
                scanf("%d",&t[i][j]);
        }
}
```

va effectuer la lecture et affecter la valeur de toutes les cases du tableau « t ».

Par souci de lisibilité, nous vous conseillons d'écrire « &(t[i][j]) ». Cette écriture met plus en évidence que l'opérateur unaire « & » (adresse de) s'applique à la case « t[i][j] ».

5. Débordement par excès et par défaut

Définition

Le débordement par excès survient quand on cherche à accéder à une case en dehors du tableau en utilisant un indice plus grand que l'indice maximal possible.

Exemple

```
include <stdio.h>
#define MAX 1000
int main()
{
    int machin[MAX];
    machin[1500] = 10;
}
```

La case d'indice 1500 n'existe pas puisque l'indice maximal pour le tableau qui a été déclaré est 999.

Attention

Le débordement par défaut survient quand on cherche à accéder à une case en dehors du tableau en utilisant un indice plus petit que l'indice minimal possible.

Exemple

```
machin[-2] = 17;
```

La case d'indice -2 n'existe pas puisque l'indice minimal est 0.

Remarque

Cela peut sembler très surprenant que le compilateur accepte ces instructions, mais c'est le cas. Plus surprenant encore, le programme compilé peut aussi bien fonctionner que ne pas fonctionner.

Les débordements constituent évidemment des erreurs graves de programmation.

Car il est très dangereux d'accéder ainsi à une donnée qui n'existe pas et de la modifier.

En effet, on accède ainsi à des octets dans la mémoire et on les modifie. Cela peut faire "planter" le code, comme cela peut ne pas le faire "planter".

Et puisque le C ne vérifie pas ce genre de chose, ce sont des erreurs très difficiles à détecter quand on les commet.

6. Les chaînes de caractères

a) Tableau de caractères

Syntaxe

Nous pouvons définir des tableaux de caractères, par exemple :

```
char t[10];
```

Dans ce cas, chaque case peut contenir un caractère. Nous pouvons le déclarer et l'initialiser ainsi :

```
char t[10] = {'a','l','p','h','a'};
```

Dans ce cas seules les 5 premières cases seront initialisées. De même, nous pouvons écrire ensuite :

```
t[6]='b';
```

A ce stade, les cases d'indices 0,1,2,3,4 et 6 sont initialisées, pas la case d'indice 5.

Remarque

Ce tableau de caractères ne constitue pas une chaîne de caractères, d'une part parce que le caractère de la case 5 n'est pas initialisé, et d'autre part parce que l'on ne sait pas quelle est la case qui contient le dernier caractère.

En résumé, un simple tableau de caractères n'est pas une chaîne de caractères.

b) Convention de codage des chaînes de caractères en langage C

Définition

En langage C, une chaîne de caractères correspond à un tableau de caractères et à l'utilisation d'une convention pour marquer la fin de la chaîne de caractères. C'est-à-dire la fin de la partie du tableau qui contient effectivement les caractères qui constituent la chaîne.

Pour cela, le langage C utilise un caractère particulier de la table des codes ASCII, c'est le '\0' (back slash zéro). Il sert de marqueur pour indiquer la fin d'une chaîne de caractères.

Un des caractères du tableau doit être le marqueur de fin de chaîne '\0'. Dans ce cas tous les caractères qui précèdent font partie de la chaîne de caractères et on ne se préoccupe pas de ceux qui suivent le marqueur.

Rappel : les littéraux constants

Nous rappelons que la syntaxe 'a' désigne un unique caractère tandis que la syntaxe "a" désigne une chaîne de caractères constante.

Dans le premier cas nous avons un unique octet, dans le second la chaîne de caractère occupe deux octets dans une zone mémoire qui est un tableau de caractères. En effet, le premier octet contient le codage du caractère 'a', le second le codage du '\0' qui marque la fin du littéral constant de type chaîne de caractères "a".

c) Initialisation d'une chaîne à la déclaration

Exemple

```
char s1[ 10 ] = {'b', 'o', 'n', ' ', '!', '\0'} ;
```

On obtient les 6 premières cases du tableau qui contiennent :

s1	b	o	n		!	\0
indice de case	0	1	2	3	4	5

Exemple

C'est une chaîne de caractères car nous utilisons la convention.

L'instruction :

```
printf("%s",s1);
```

Affichera, en séquence les uns derrières les autres tous les caractères du tableau s1 jusqu'à trouver le caractère '\0'.

C'est-à-dire que l'on obtiendra : « bon ! » à l'écran. Sur l'écran.

Ce qui est équivalent à :

```
i = 0;
while (s1[i]!='\0') putchar(s1[i++]);
```

Remarque

Nous pouvons remarquer que la liste d'initialisation fait explicitement apparaître le caractère de marquage '\0'.

Cette initialisation par liste peut être fastidieuse, il existe une autre façon de faire dans le cas des chaînes de caractères.

Exemple

```
char s2[10] = "bonjour";
```

En effet comme pour le "a", le caractère '\0' est implicite après le caractère 'r' dans le littéral constant de type chaîne de caractères "bonjour".

Pour s2, qui comporte 10 cases, seules les 8 premières cases seront utilisées.

Introduisons maintenant les syntaxes équivalentes suivantes :

```
char *s3 = "bonjour";
char s4[] = {'b','o','n','j','o','u','r','\0'};
```

A la compilation, le compilateur réserve automatiquement autant de cases que nécessaires pour ranger les caractères de l'initialisation à la déclaration.

Ces syntaxes impliquent que s3 et s4 sont des tableaux d'exactement 8 caractères, 7 sont nécessaires pour les lettres de "bonjour" et un huitième pour coder le '\0' qui marque la fin des chaînes.

Cette valeur est calculée par le compilateur à partir des nombres de caractères utilisés pour l'initialisation à la déclaration.

d) Initialisation d'une chaîne par saisie

Exemple

Considérons l'exemple :

```
char chai[ 6 ] = {'a', '\0'} ; scanf("%s",chai);
```

Initialement la chaîne « chai » contient le littéral constant de type chaîne de caractères « a ». Puis l'usage de « scanf » avec le format %s effectue la lecture au clavier d'une chaîne de caractères qui va écraser cette ancienne valeur.

Vous remarquerez que nous n'utilisons pas l'opérateur & (adresse de) devant l'identificateur « chai » du tableau de caractères, ce n'est pas une erreur (se référer aux explications dans la section "lien entre tableau, indice et pointeur")

Si l'on saisit plus de caractères que ne peut en contenir le tableau, seuls les

premières cases seront remplies, la chaîne saisie est alors tronquée et les caractères en surnombre sont perdus.

e) Initialisation d'une chaîne par calcul

Remarque

Le tableau peut évidemment se remplir case par case mais il ne faut pas oublier d'ajouter le '\0' à la fin.

Exemple

```
#define MAX 5 int i; char truc[MAX]; for (i=0;i<MAX-1;i++)
truc[i]='A'+i; truc[MAX-1] = '\0';
```

Les cases d'indices 0,1,2,3 sont remplies avec successivement les caractères 'A', 'B', 'C', 'D' (dans l'ordre des codes de la table ASCII). Ensuite la case d'indice 4 est remplie avec '\0'.

Mais il est préférable d'utiliser les fonctions prédéfinies dans la bibliothèque <strings.h>.

Exemple

```
#include <string.h>
#define MAX_LEN 8
char toto[MAX_LEN];
strcpy(toto,"bonjour");
```

Où la fonction strcpy copie le mot « bonjour » dans la chaîne de caractères toto.

Remarque

Le caractère '\0' prend une place dans le tableau de caractères. Si vous avez besoin de manipuler des chaînes de 8 caractères, il faudra déclarer des tableaux de caractères de taille 9.

f) Bibliothèque <string.h>

Définition

Cette bibliothèque fournit un ensemble de fonctions qui permettent de manipuler des chaînes de caractères. En voici la liste :

strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strtok.

Vous remarquerez que leurs noms commencent toujours par "str" pour string (chaîne en anglais).

Nous détaillons le fonctionnement de :

- char * strcpy (chaîne1, chaîne2) : copie le contenu de « chaîne2 » dans « chaîne1 ». L'ancienne valeur de chaîne1 est perdue. Retourne un « pointeur sur » chaîne1.
- int strlen (chaîne) : renvoie le nombre de caractères de la chaîne. Cette fonction compte les caractères jusqu'à trouver le '\0'. La valeur renvoyée est le nombre de caractères avant le '\0'
- char * strcat (chaîne1, chaîne2) : copie le contenu de chaîne2 à la fin de chaîne1. Retourne un « pointeur sur » chaîne1.
- int strcmp (chaîne1 , chaîne2) : compare les deux chaînes, retourne 0 en cas d'égalité, une valeur négative si « chaîne1 » est "avant" « chaîne2 »

dans l'ordre lexicographique, une valeur strictement positive sinon.

- char * strncat (chaine1 , chaine2 , n) : concatène au plus les « n » premiers caractères de « chaine2 » à « chaine1 ». Retourne un pointeur sur « chaine1 ». Si n est plus grand que la longueur de « chaine2 », la longueur de « chaine2 » est utilisée à la place de n. Retourne un « pointeur sur » chaîne1.
- char * strncpy (chaine1 , chaine2 , n) : copie « n » caractères de « chaine2 » sur « chaine1 ». Retourne « un pointeur sur chaine1 ». Si « n » est inférieur à longueur de « chaine2 » « \0 » n'est pas ajouté en fin de « chaine1 ». Si « n » est supérieur à longueur de « chaine2 », « chaine1 » est complétée par des « \0 ». Retourne un « pointeur sur » chaîne1.

Pour les opérations de copies, si la chaîne de caractères cible « chaîne1 » ne contient pas assez de place pour contenir « chaîne2 » alors seuls les caractères que l'on peut copier le sont. On dit que « chaîne2 » est tronquée.

Nous ne les décrirons pas toutes dans le cadre de ce cours introductif. Connaissant leur nom, il vous sera facile de retrouver ce qu'elles effectuent comme traitement en consultant les documentations adhoc.

Attention

Si vous ne leur fournissez pas de tableaux de caractères qui comportent quelque part le caractère de marquage '\0', vous pouvez avoir des problèmes en les utilisant.

Attention

« chaine1 » et « chaine2 » sont des chaînes de caractères c'est-à-dire qu'il faut transmettre aux fonctions des « pointeurs sur char »

En C, on ne peut pas comparer deux chaînes de caractères avec les opérateurs >, >=, <, <=, ==. Il faut utiliser la fonction « strcmp ».

7. Lien entre tableau, indice et pointeur

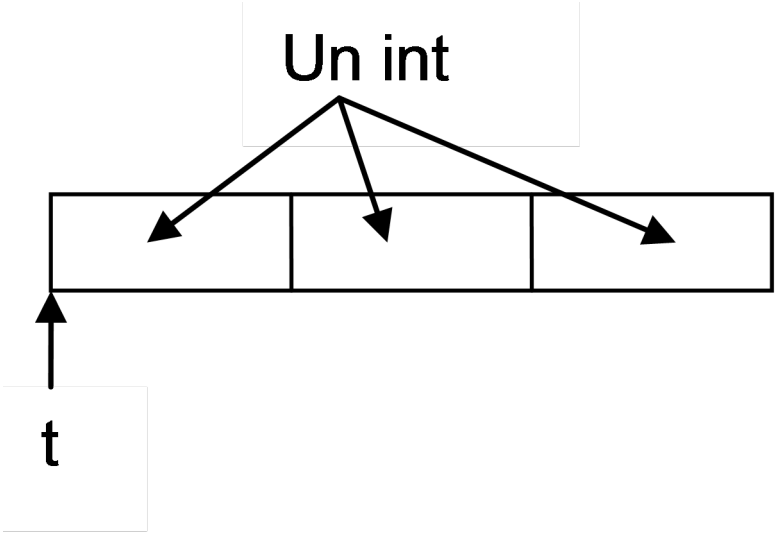
Soit la déclaration de tableau suivante :

```
int t[3];
```

Nous avons déclaré et réservé une zone mémoire contiguë.

L'identificateur « t » est en fait l'identificateur d'une CONSTANTE qui est de type 'pointeur vers int'.

La valeur de cette constante est l'adresse du premier octet du premier élément du tableau. Comme le montre le schéma ci-dessous :



Caractéristiques d'un tableau

En résumé, « t » est l'identificateur d'un pointeur constant vers un entier et dont la valeur est l'adresse du premier octet de la zone réservée.

Puisque « t » est une constante, on ne peut pas modifier sa valeur.

Exemple :

```
int t[3];
int * point_sur_int;
int i = 5;
point_sur_int = &i;
t = point_sur_int;
/* t est l'identificateur d'un pointeur constant : cette
instruction est INTERDITE !*/
t = t + 1;
/* t est l'identificateur d'un pointeur constant : cette
instruction est INTERDITE */
```

L'opérateur d'affectation n'est pas directement utilisable entre les identificateurs de tableaux. Mais on peut manipuler les tableaux case par case.

Par exemple, si on veut faire la somme de 2 vecteurs « v1 » et « v2 » dans « v3 », on ne peut pas écrire « v3=v1+v2 ». Il faut ajouter les cases de « v1 » et « v2 » une à une à l'aide d'une boucle selon une instruction du type « v3[i] = v1[i] + v2[i] ».

a) Cas des tableaux unidimensionnels

Exemple

Puisque dans la déclaration :

```
int t[10];
```

« t » est l'identificateur d'un pointeur constant, nous pouvons introduire une autre manière d'accéder à une case du tableau en utilisant l'arithmétique sur les pointeurs.

En effet, les instructions suivantes sont strictement identiques :

```
t[i] = 0;
*(t+i) = 0;
```

De la même façon : &t[i] et (t+i) fournissent la même valeur. C'est l'adresse du

premier octet de la case d'indice i du tableau t.

En fait, quand nous écrivons :

```
t[i] = 0;
```

Le compilateur génère des instructions machine que l'on peut écrire comme :

```
*(t+ i*sizeof(int))= 0;
```

ce qui permet de remplir la case i à partir de son premier octet avec la valeur 0.

Les 4 octets consécutifs que l'on trouve à partir de l'adresse t+ i*sizeof(int)) sont donc initialisés.

C'est donc un décalage de i entiers à partir de l'adresse du début du tableau. Ceci est transparent pour le programmeur.

```
float t[3] = {1.0, 2.0 ,3.0};
/* t est l'identificateur d'un pointeur constant, dont la
valeur est l'adresse du premier octet du tableau */
float * pt_sur_flottant;
/* « pt_sur_flottant » est l'identificateur d'une variable
qui contient une adresse vers un flottant */
/* les deux lignes suivantes sont equivalentes */
pointeur_sur_flottant = t; // on peut affecter la valeur
d'une constante à une variable
pointeur_sur_flottant = &t[0]; // c'est la même chose que
l'instruction ci-dessus
/* car &*(t+0) équivanut à (t+0) donc t */
/* et les deux lignes suivantes sont aussi équivalentes */
*pointeur_sur_flottant = -273.15;
/* car c'est *t donc *(t+0) donc t[0] */
t[0] = -273.15; // c'est la façon la plus simple de
procéder.
```

Nous avons donc 2 façons de réaliser l'accès à des cases d'un tableau "bidule" :

```
bidule[i] et *(bidule +i)
```

Mais la notation bidule[i] est beaucoup plus facile à comprendre, à lire et à écrire.

C'est donc celle qu'il faut privilégier.

b) Cas des tableaux bi-dimensionnels

Exemple

Considérons l'exemple :

```
#define MAX_L 2
#define MAX_C 3
int t[MAX_L] [MAX_C];
int i,j;
```

Cela correspond au dessin linéarisé suivant :

t						
indices	0	1	2	3	4	5
	t[0][0]	t[0][1]	t[0][2]	t[1][0]	t[1][1]	t[1][2]

Linéarisation d'un tableau à deux dimension

En effet, le tableau est linéarisé selon le schéma ligne/ligne.

Dans ce cas, t est l'identificateur d'un pointeur constant aussi. Mais ici c'est l'adresse d'une adresse.

C'est l'adresse de l'adresse de la première ligne. Et *t est donc l'adresse de la première ligne c'est en fait *(t+0).

Donc :

(t+i) est l'adresse de l'adresse de la ligne d'indice i

*(t+i) est l'adresse de la ligne i

*(t+i) + j est l'adresse du premier octet de la case j de la ligne i car on a un décalage de j entiers dans la ligne i

((t+i)+j) est donc la valeur de l'entier qui se trouve là,

et c'est la même chose que t[i][j]

donc, si vous avez bien compris le cours sur les tableaux :

t[1][2] est la même chose que :

*(t[1]+2)

((t + 1) + 2)

*((int *)t+5)

Qui sont des façons d'accéder à la case qui se trouve en ligne 1 colonne 2.

Explications particulière pour la notation « *((int *) t+5) » : il y a 3 colonnes et ((1*3)+2)=5 est bien le nombre d'entiers depuis le début du tableau qui correspond à la case d'indices (1,2) avec la linéarisation.

Mais il faut un « cast » car « t » est vu comme « un pointeur de pointeur vers entier» : il faut que le compilateur le "voit" comme un « pointeur vers entier » pour ne pas avoir de « warning » à la compilation.

Si on écrit directement l'expression (t+5) pour calculer l'adresse d'un entier (pointeur vers entier), le compilateur va l'interpréter comme l'accès à la ligne d'indice 5, ce qui peut être gênant.

Attention

Quand on écrit dans le programme :

```
&(t[i][j])
```

la machine calcule en fait l'expression mathématique suivante pour avoir l'adresse :

```
t + ( i x MAX_C + j) x Taille
```

avec Taille = sizeof(int)

C'est-à-dire la taille en nombre d'octets d'une case du tableau.

Et forcément quand on écrit :

```
t[i][j] = 0;
```

Le compilateur effectue quelque chose comme :

```
*(t + (i x MAX_C + j) x Taille) = 0
```

Un tableau à deux dimensions peut donc être vu comme un tableau de tableau à une dimension.

Considérons la déclaration suivante :

```
float machin[2][4] = {{10.0,9.0,8.0,7.0},
{51.0,52.0,53.0,54.0}};
```

Le compilateur le « voit » comme dans la figure suivante :

pointeur de ligne :

```
machin+0 -> 10.0 9.0 8.0 7.0
machin+1 -> 51.0 52.0 53.0 54.0
```

La valeur « machin[0] » est une valeur qui est un pointeur vers le premier octer de la ligne 0, ce qui correspond à *(machin +0).

Ce qu'il faut retenir :

- le compilateur à absolument besoin de connaître le nombre de colonnes car il a besoin de la valeur MAX_C pour effectuer ces calculs d'adresses
- l'usage des pointeurs pour accéder aux tableaux à deux dimensions est compliqué.

Conseil : Usage des tableaux multidimensionnels

N'utilisez pas les pointeurs avec des tableaux à 2 dimensions

Utilisez les notations

- tab[i][j] pour les opérations manipulant la valeur de la case (i,j)
- &(tab[i][j]) pour les opérations qui ont besoin de l'adresse de la case comme le « scanf » par exemple.

En effet :

scanf("%d",&t[i][j]); sera toujours plus clair que

scanf(%d",&*((int *)*(t+i)+j));

Les tableaux à « n » dimension (n>2) correspond à un tableau de tableaux de dimension n-1.

Nous ne présenterons pas en détail les tableaux à plus de 2 dimensions, sachez seulement que :

```
#define MAX_L 2
#define MAX_C 3
#define MAX_P 7
int t_3[MAX_L] [MAX_C] [MAX_P];
int i,j,k;
```

Déclare un tableau à 3 dimensions, et que :

```
t_3[i][j][k]= 7;
```

pour mettre la valeur 7 dans la case d'indices (i,j,k) ainsi que :

```
scanf("%d",&(t_3[i][j][k]));
```

pour saisir la case d'indices (i,j,k), fonctionnent parfaitement.

Et « t_3 » est l'identificateur d'une constante qui est vue comme une adresse d'adresse d'adresse d'entier par le compilateur ...

Exemple : Pour terminer

Comme nous l'avons vu il y a un lien étroit entre pointeurs et indices de tableau.

Considérons l'exemple suivant :

```
#include <stdio.h>
#define MAX 30
int main()
{
    float t_reels[MAX];
     int i,j;
     float * p1, *p2;
     p1 = &t_reels[3];
     p2 = &t_reels[7];
```

```
     printf("\n7-3 = %d",7-3);
     printf("\n(p2-p1) = %d ",(p2-p1));
     printf("\n(p2-p1) avec cast void = %d\n",(void *)p2-
(void *)p1);
     printf("\n\n");
}
```

Nous aurons à l'exécution l'affichage :

```
7-3 = 4
(p2-p1) = 4 (p2-p1) avec cast void = 16
```

En effet, "p1" et "p2" sont deux variables de type "pointeur vers flottant".

Elles sont initialisées avec deux adresses de flottant qui correspondent aux cases 3 et 7.

L'expression "(p2-p1)" effectue la soustraction de deux pointeurs.

Le compilateur sait que ce sont deux pointeurs vers des flottants. Il calcule donc le nombre de flottant qui séparent les deux adresses en utilisant l'information sur la taille d'un flottant.

Si l'on transforme les "pointeurs sur float" en "pointeur sur void" avec des cast, la différence donne 16 car dans ce cas on soustrait deux adresses d'octets. Sur cette machine, nous en déduisons qu'un flottant est codé sur 4 octets.

8. Exercices de Révision (QCM)

Exercice 1

1. On considère deux tableaux T1 et T2. Peut-on copier le contenu de T2 dans T1 sans perdre d'information ?

○	Directement si T1 et T2 sont de même taille ? On utilise l'instruction T1 = T2 ;
○	Directement si la taille de T1 est supérieure à la taille de T2 ? On utilise l'affectation T1 = T2 ;
○	Directement si la taille de T2 est supérieure à la taille de T1 ? On utilise l'affectation T1 = T2 ;
○	Elément par élément à l'aide d'une boucle dès que la taille de T1 est >= à la taille de T2 ?

Exercice 2

2. Quelle(s) déclaration(s) correspond(ent) à une matrice de N lignes et M colonnes ?

○	float Identificateur[N][M] ;
○	float Identificateur [M-1][N-1] ;
○	float Identificateur1 [M-1] Identificateur2 [N-1] ;
○	float Identificateur1 [N-1] Identificateur2 [M-1] ;

Exercice 3

3. En langage C, dans une chaîne de caractères, le premier caractère a pour indice :

○	0
○	1

Exercice 4

4. Pour accéder à la troisième case du vecteur Carte, on utilise l'instruction :

○	Carte [3] ;
○	Carte [2] ;
○	Carte {2} ;
○	Carte {3} ;
○	Carte (2) ;

Exercice 5

5. Pour accéder à la case située à la 2éme ligne et la 3éme colonne de la matrice T, quelle est la bonne syntaxe?

○	T [2,1] ;
○	T [1,2] ;
○	T [2] [1] ;
○	T [1] [2] ;
○	T (2, 1) ;

Exercice 6

6. on définit les variables de la manière suivante :
int in;
int tabint[10];
char car;
*int *ptint;*
*char *ptchar ;*
Cocher ce qui est juste :

☐	ptint = ∈ *ptint = 12;
☐	ptint=&tabint; *ptint=4;
☐	ptchar = &car; *ptchar = 'a';
☐	tabint[in] est équivalent a *(tabint + in)
☐	ptint=tabint; *ptint=4;

Exercice 7

7. Peut-on changer la taille d'un tableau en cours d'exécution du programme ?

○	Oui, en affectant une nouvelle valeur à la variable définissant la taille du tableau.
○	Non, c'est impossible.
○	Oui, en redéfinissant la valeur de la constante indiquant la taille du tableau.

Exercice 8

8. Soit P un pointeur qui 'pointe' sur un tableau A:
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
*int *P;*
P = A;
*Quelle valeur correspond à : *(P+*(P+8)-A[7])*

○	14
○	33
○	23

Exercice 9

*9. On considère la déclaration suivante :char *ptchar, ptchar peut contenir des valeurs qui sont :*

○	Des valeurs de variables de type caractère (char).
○	Des adresses de variables de type caractère (char).

Exercice 10

10. On définit les constantes et les variables suivantes :
#define a 5
#define b 7
#define g 5.6
float c,d ;
int e,f ;
Lesquelles de ces déclarations sont justes ?

☐	int identificateur1 [10] [b] ;
☐	float identificateur2 [10][b] ;
☐	int identificateur3[a][b] ;
☐	int identificateur4[g][b] ;
☐	char identificateur5[1][c] ;
☐	float identificateur6 [e][f] ;
☐	int identificateur7 []={ fievre, delire, nausee } ;

Exercice 11

11. Comment fait-on pour déclarer un tableau dont les éléments sont de types différents ?

○	Type_case répertoire [M] ; et on précise le type des variables à chaque entrée.
○	C'est impossible.

Exercice 12

12. Que fait la fonction strncpy(s2, s3, 5)?

○	Copie les 5 premiers octets de s3 dans la chaîne pointée par s2.
○	Copie les 5 premiers octets de s2 dans la chaîne pointée par s3.
○	Copie les 5 derniers octets de s3 dans la chaîne pointée par s2.

G. Les fonctions

Nous avons déjà utilisé des fonctions de bibliothèques standard comme « printf » (déclaration dans « stdio.h ») ou encore « sin » (déclaration dans « math.h »).

Dans ce chapitre nous allons apprendre à déclarer et invoquer une fonction que nous avons conçue.

1. Qu'est ce qu'une fonction ?

Définition

Une fonction est un bloc d'instructions auquel on donne un nom pour pouvoir l'utiliser.

Eventuellement, ce bloc d'instructions peut avoir besoin de valeurs pour pouvoir travailler sur des données qu'on lui transmet. Dans ce cas, on dit que la fonction a besoin de paramètres.

2. Comment fonctionne l'invocation d'une fonction de bibliothèque ?

Fondamental

Nous supposons que la constante PI et que l'inclusion des bibliothèques <stdio.h> et <math.h> ont été effectuées correctement. De même x est une variable qui correspond à un float.

Considérons maintenant la portion de code suivante :

```
instruction_avant;
x = sin(PI/4);
printf("\nLa valeur du sinus est %f ",x);
instruction_après;
```

Nous avons quatre instructions qui sont exécutées l'une à la suite de l'autre.

Nous ne détaillons pas « instruction_avant » et « instruction_après », qui sont juste là pour les besoins de l'explication.

« instruction_avant » est effectuée.

Pour la seconde instruction « x = sin(PI/4); », à droite de l'opérateur d'affectation,

nous avons l'appel de la fonction sinus qui a pour paramètre le résultat de l'expression PI/4. On dit que la fonction "sin" est invoquée (ou appelée, ou utilisée) dans cette instruction. Le paramètre réel (au sens effectivement transmis) de la fonction est une valeur flottante. C'est la valeur qui est transmise par copie à la fonction pour qu'elle puisse s'exécuter. L'expression est donc calculée puis son résultat est transmis par copie à la fonction. Quand la fonction a terminé son travail elle retourne une valeur qui est un flottant. Cette valeur est ensuite affectée à la variable « x ».

Pour la troisième instruction, nous avons juste l'appel de la fonction « printf » avec deux paramètres réels. Le premier paramètre est la chaîne de caractères constante "\nLa valeur du sinus est %f" et le second paramètre est l'identificateur de la variable « x ».

La fonction reçoit donc deux valeurs. La chaîne de caractères constante et la valeur de « x ». Les deux valeurs sont transmises par copie à la fonction. La fonction « printf » utilise ces deux valeurs. Elle identifie les séquences d'échappements au sein de la chaîne de caractères et effectue les traitements appropriés.

Elle effectue l'affichage à l'écran de :

La valeur du sinus est 0.707106781187

Cette fonction « printf » ne renvoie pas de valeur.

Les fonctions en « C » retournent toujours une valeur. Ce qui semble contradictoire avec la description ci-dessus. Une fonction qui ne retourne aucune valeur retourne donc la valeur « rien », c'est-à-dire une donnée de type « void », même si aucune valeur n'est retournée.

Les fonctions en C sont typées car elles retournent toujours une valeur (void dans le cas où aucune valeur n'est à retourner).

Les fonctions permettent :

- de décomposer une longue séquence de traitements en un ensemble de petits traitements
- de fournir des éléments de base de haut niveau réutilisable à volonté
- d'améliorer la lisibilité du code
- de faciliter la maintenance d'un code
- d'éviter la répétition de morceaux de codes identiques.

Fondamental

Revisitons plus en détail la séquence d'instructions :

```
instruction_avant;
x = sin(PI/4);
printf("\nla valeur du sinus est %f ",x);
instruction_après;
```

1. La première instruction instruction_avant est exécutée.
2. La seconde x = sin(PI/4) est exécutée. Il y a recopie d'informations, puis branchement vers la portion de code qui effectue le calcul numérique du sinus. La fonction « sin » travaille sur la donnée transmise (exécution d'instructions). Puis nous avons à nouveau un branchement pour revenir à la suite du déroulement du code. Il y a récupération de la valeur retournée par la fonction. Enfin il y a affectation de la valeur retournée (0.707) à la variable « x ».
3. A ce stade, nous sommes revenus dans le déroulement du code et prêts à exécuter l'instruction printf("\nla valeur du sinus est %f ",x). Il y a recopie d'informations, puis branchement vers la portion de code de « printf », la fonction « printf » travaille sur les données transmises (exécution d'instructions), puis de nouveau branchement pour revenir à la suite du

déroulement du code.
4. A ce stade, nous sommes prêts à exécuter « instruction_après ».
5. Enfin, cette instruction est exécutée.

Complément : En résumé

Vous avez utilisé de façon totalement transparente pour vous des recopies d'informations et des branchements. C'est le compilateur qui fait le travail pour vous.

Pour créer vos propres fonctions, il faut utiliser une syntaxe qui permet au compilateur de mettre en place ces mécanismes de recopie d'informations et de branchements. La déclaration d'une fonction a pour objet de fournir au compilateur toutes les informations nécessaires pour générer et gérer les recopies/transmissions d'informations, l'exécution des instructions de la fonction et les branchements.

Ensuite, le compilateur mettra en place les mécanismes décrits pour que votre fonction soit utilisable, à chaque fois que vous utiliserez votre fonction. Nous distinguons clairement :

- la déclaration d'une fonction (on dit aussi définition)
- l'usage (invocation, appel) d'une fonction connue (qui donc a été préalablement déclarée)

3. Déclaration d'une fonction et compilation

Syntaxe

La déclaration d'une fonction se réalise de la manière suivante :

```
type identificateur_de_la_fonction(liste de paramètres formels) bloc
```

Exemple

Ci-dessous, un exemple de déclaration fonction et des explications sur ce qui se passe lors de la compilation :

```
int plus_un(int a)
                          /* identificateur de la fonction et paramètre */
{
                         /* début du bloc de function */
  int inter;
  inter = a+1;
  return inter;
}
```

1. La première ligne « int plus_un(int a) » définit l'en-tête de la fonction.
2. Ensuite nous trouvons un bloc qui commence par '{' et se termine par '}'.
3. Examinons l'en-tête de la gauche vers la droite :
   - int est le type de valeur retournée par la fonction
   - plus_un est l'identificateur de la fonction
   - le délimiteur '(' commence la liste des paramètres formels
   - int est le type du paramètre formel
   - a est l'identificateur du paramètre formel ☐ le délimiteur ')' ferme la liste de paramètres formels

Fondamental

La liste de paramètres formels est obligatoire même si elle est vide. En résumé, les

« (« et «) » sont obligatoires. La notion de paramètre formel est fondamentale.

Quand on déclare une fonction, on ne sait pas encore quelle valeur exacte nous lui transmettrons quand nous l'utiliserons. Cependant, nous avons besoin d'indiquer au compilateur que l'on va transmettre une valeur à la fonction et que cette valeur devra être recopiée pour que la fonction puisse l'utiliser. C'est pourquoi on décrit formellement le type de la donnée et qu'on lui donne un identificateur pour nom.

Nous avons là encore un couple (identificateur, réceptacle). La valeur de la donnée sera recopiée dans le réceptacle au moment de l'appel (invocation, utilisation) de la fonction et nous pouvons l'utiliser car il a un identificateur, c'est-à-dire un nom.

A la suite de la déclaration, nous trouvons le bloc de la fonction :

Attention

Attention, ce n'est pas exactement la même chose qu'une variable. Il s'agit d'un paramètre formel.

A ce stade, un paramètre formel peut être vu comme une variable dotée en plus d'un mécanisme de recopie automatique d'informations (Attention, ce n'est pas la seule différence).

Fondamental

A la suite de la déclaration, nous trouvons le bloc de la fonction :

```
{
 int inter;
 inter = a+1;
 return inter;
}
```

Nous déclarons une variable « inter », c'est la variable locale de la fonction « plus_un ». Quand on évoque la notion de variable locale il faut toujours préciser à quelle fonction elle est locale.

Puis, suit l'instruction « inter = a+1 ». Qui met en jeu la variable locale « inter » et le paramètre formel « a ». Puisque nous sommes dans le bloc de la fonction, ces deux identificateurs sont parfaitement définis et le compilateur accepte de générer l'instruction machine correspondante. Il va prendre la valeur du paramètre formel « a », ajouter 1, et ranger le résultat dans la variable locale "inter".

Enfin, nous avons l'instruction “return inter”. Le mot réservé “return” est utilisé au sein de la déclaration du bloc de la fonction pour mettre en place le mécanisme qui permet à la fonction de retourner de l'information. La fonction est de type « int» et cette fonction retourne la valeur contenue dans la variable locale "inter" qui est elle-même un entier. Cette opération est donc licite.

Le “return” permet au compilateur de mettre en place non seulement le mécanisme de transmission du résultat de la fonction, mais aussi le mécanisme de branchement pour revenir à l'instruction qui suit l'invocation de la fonction. L'instruction « return » ne peut être utilisée que dans une fonction. Elle retourne la valeur d'une expression.

Si la fonction est de type « void » il ne faut pas utiliser « return ».

A ce stade, quand le compilateur a terminé de compiler cette déclaration de fonction, toutes les informations et mécanismes nécessaires à son usage sont définis et prêts à être utilisés. De plus, les instructions de la fonction ont été compilées et rangées. Le compilateur sait exactement où les instructions sont rangées afin de réaliser les branchements. C'est lui qui gère cela, pas vous.

4. Invocation d'une fonction

Exemple

Considérons le code complet suivant :

```
#include <stdio.h>
int plus_un(int a)
{
 int inter;
      /* point point_fonc */
 inter = a+1;
  return inter;
}
void main()
{
  int x,y;
      /* point 1 */
 x = plus_un(7);
     /* point 2 */
 y = plus_un(x);
   /* point 3 */
}
```

Considérons la compilation de ce code du point de vue de l'usage de la fonction « plus_un ».

Dans un premier temps, il y a inclusion des définitions des fonctions d'entrée/sortie de la bibliothèque « stdio.h ».

Puis compilation de la fonction « plus_un » comme expliqué avant. Le compilateur rencontre alors l'en-tête de la fonction

```
 void main ()
```

Cette fonction ne retourne rien et n'a besoin d'aucun paramètre pour fonctionner. Le compilateur détecte que c'est la fonction principale. C'est elle qui sera exécutée. Tout ce qui est avant n'est que déclaration.

Nous avons ensuite le bloc main, il y a déclaration de deux variables locales pour la fonction main : « x » et « y » de type entier. Le compilateur rencontre ensuite l'instruction :

```
  x = plus_un(7);
```

Il vérifie s'il connaît l'identificateur « plus_un », c'est le cas, c'est un identificateur de fonction qu'il a compilé. Vous remarquerez que la valeur 7 est le paramètre réel de la fonction. Il vérifie alors si le nombre de paramètres réels est identique au nombre de paramètres formels attendus par la définition de la fonction. C'est le cas, la fonction est déclarée avec un paramètre formel et on veut l'utiliser avec un paramètre réel. Le compilateur vérifie alors si le type du paramètre réel est compatible avec le type du paramètre formel déclaré. Là encore c'est le cas, la fonction attend un entier, nous lui transmettons une valeur entière.

A ce moment, la vérification de l'invocation de la fonction est terminée.

Le compilateur met en place le mécanisme de recopie de la valeur 7 dans le réceptacle du paramètre formel "a". Il met ensuite en place le mécanisme de branchement vers le code compilé de la fonction pour exécuter les instructions, puis il réalise le mécanisme de branchement de retour de la fonction et enfin exécute le mécanisme de récupération de la valeur retournée (return inter). La valeur retournée sera recopiée dans la variable « x ».

A ce stade, il reste à compiler l'instruction :

```
y = plus_un(x);
```

La seule chose qui diffère concerne la recopie d'information. Dans ce cas on va recopier la valeur de la variable « x » au sein du réceptacle du paramètre formel « a ». Puis la valeur retournée sera recopiée dans la variable « y ».

La compilation échoue si :

- le nombre de paramètres réels n'est pas le même que le nombre de paramètres formels déclarés
- les types des paramètres réels utilisés pour l'appel de la fonction ne sont pas compatibles avec les types des paramètres formels déclarés. C'est la vérification de la concordance de types.

Considérons maintenant l'exécution du code :

a) Au point 1

Les variables locales x et y existent, mais ne sont pas initialisées.

La fonction est appelée : il y a création temporaire des réceptacles pour le paramètre formel "a" et pour la variable locale « inter », puis recopie de 7 dans « a », nous venons d'être "branché" au point « point_fonc », l'instruction « inter = a+1; » est exécutée, « a » vaut toujours 7 mais « inter » vaut 8, puis l'instruction « return inter » est effectuée.

Le mécanisme de récupération de la valeur 8 est activé, un branchement de retour vers l'instruction suivante de la fonction principale « main » est effectué et les réceptacles du paramètre formel « a » et de la variable locale « inter » sont récupérés. Ils n'ont existé que pour la durée d'appel et de travail de la fonction. Enfin la valeur 8 retournée par la fonction est affectée à la variable « x ».

b) Au point 2

« x » vaut 8, la valeur de « y » est inconnue.

La fonction est appelée à nouveau :

Le paramètre réel est la valeur de « x », c'est donc la valeur 8 qui est recopiée dans "a", puis cela fonctionne exactement de la même façon que dans le premier cas. La valeur retournée par la fonction est donc 9, qui est affecté à la variable « y ».

c) Au point 3

x vaut 8, et y vaut 9

d) Explications

Attention

- Les paramètres formels permettent de décrire ce que l'on recevra quand la fonction sera effectivement invoquée
- Les paramètres réels sont ceux qui sont effectivement transmis quand on invoque la fonction
- L'unique mécanisme de transmission d'information entre paramètres réels et paramètres formels est la recopie des valeurs des paramètres réels dans les réceptacles des paramètres formels.
- Les paramètres formels et les variables locales d'une fonction ont une durée de vie limitée, ils n'existent que pendant le temps que la fonction travaille. Au point 1 ni "a" ni "inter" n'existent, pas plus au point 2, pas plus au point 3.

Ils n'existent qu'au point « point_fonc » !

Méthode

Au moment de l'invocation d'une fonction il y a donc :

1. création du contexte de travail de la fonction, c'est-à-dire création dynamique de réceptacles pour les paramètres formels et les variables locales de la fonction qui est invoquée
2. recopie des valeurs de paramètres réels dans les réceptacles des paramètres formels
3. exécution des instructions de la fonction
4. transmission d'information (si la fonction renvoie une valeur autre que de type void)
5. destruction du contexte de travail de la fonction
6. poursuite du déroulement du code qui suit l'invocation de la fonction

Remarque

Vous avez remarqué que nous n'avons volontairement plus parlé de branchements. C'est désormais implicite, le compilateur fait le travail de façon transparente.

Vous remarquerez que l'identificateur de la fonction permet à sa simple lecture de comprendre ce que fait la fonction. Il est de bonne pratique de donner à la fonction que l'on déclare un identificateur significatif de la nature du traitement qu'elle effectue.

Rappel

Nous rappelons qu'une fonction doit être déclarée avant d'être utilisée.

Fondamental

Si la fonction retourne une valeur d'un type différent de « void » alors l'appel peut être inséré dans une expression.

Exemple

Déclaration de la fonction impair

```
int impair(int x)
{
  return (x%2);
          // (modulo 2) vaut 1 (vrai en C) si impair, 0
(faux) sinon
}
```

Usage dans une expression :

```
scanf("%d", &truc);
if (impair(truc) && (truc > 10)
    printf("\n %d est impair et plus grand que 10", truc);
```

Exemple

Définition de la fonction qui calcule la norme Euclidienne d'un vecteur de deux coordonnées :

```
float norm_2(float x, float y)
{
  return sqrt(x*x+y*y);
}
```

usage de la fonction normalisation du vecteur de coordonnées « a » « b » flottantes :

```
a = a/norm_2(a,b);
b = b/norm_2(a,b);
```

Exemple : Autre exemples d'appels de fonctions prédéfinies

```
x = sin (y) ;
    // fonction retournant un double
x = pow (8,3) ;
    // 8*8*8 fonction avec 2 paramètres
print(" i = " , i);
    // fonction utilisée pour afficher
```

5. Paramètres formels, paramètres réels et variables locales

Fondamental

L'unique mécanisme qui permet de transmettre de l'information à une fonction est la copie de cette information dans le réceptacle d'un paramètre formel.

Méthode

A la déclaration, on définit une liste de paramètres formels entre "(" et ")" séparés par des ",", si la liste est vide implicitement le paramètre est void.

Exemple

```
int truc()
```

est identique à

```
int truc(void)
```

Conseil

Nous vous conseillons dans ce dernier cas d'utiliser int truc(void), car cette écriture est plus explicite.

A l'invocation, liste d'arguments entre "(" et ")" correspond aux paramètres réels, cela peut être des constantes, des variables ou des expressions, à condition que la concordance des types soit respectée. Chaque valeur est ensuite recopiée dans le paramètre formel correspondant.

En effet, la correspondance entre paramètres formels et paramètres réels se fait en suivant l'ordre dans la liste.

Remarque

La déclaration d'une variable locale d'une fonction est valable jusqu'à la fin du bloc de la fonction.

Puisque la fonction travaille sur des copies d'informations, toute modification apportée à un paramètre formel dans le bloc de la fonction n'a strictement AUCUNE influence sur la valeur du paramètre réel qui lui correspond.

Exemple : Exemple

```
int plus_un(int x) // ici l'identificateur x est celui du
paramètre formel
{
      x ++;
      return x ;
}
void main( )
{
      int x, y ;      // x est ici une variable locale de
la fonction main,
       x = 1 ;
      y = 0;
      y = plus(x) ;     // x est ici la variable locale de
la fonction main,
                                    // la valeur de x
variable locale vaut 1, elle est utilisée comme
                                    // paramètre réel , sa
valeur est donc recopiée
                                    // dans le réceptacle de
x paramètre formel.
                                    // ATTENTION : le nom de
l'identificateur est le même mais ce sont deux choses
                                    // différentes, il y a
un "x" défini dans la fonction "main"
                                    // et c'est une variable
locale
                                    // et un "x" défini dans
la fonction plus_un et c'est un paramètre formel.
                                    // La valeur de l'un est
recopiée dans l'autre, c'est tout !
      printf("\n%d",x);
                                     // ici nous affichons
la valeur de « x » qui vaut toujours 1, variable locale
                                     // de la fonction main
qui n'a pas été modifiée
                                     // dans la fonction
plus_un puisque la fonction a travaille
                                     // sur une COPIE de la
valeur !
}
```

Exemple : Autre exemple plus compliquer

```
void plus(int x, int y)
{
      y = x ++ ;
}
void main( )
{
      int x, y ;
      x = 1 ;
      y = 0;
      plus(x,y);
      printf("\n%d %d",x,y);
}
```

Ce code affiche 1 et 0 et ne modifie en rien les variables locales x et y de la

fonction principale main.

Les paramètres formels contiennent des copies des valeurs des paramètres effectifs. On peut modifier les valeurs des paramètres formels au sein de la fonction cela n'aura aucun impact sur les valeurs transmises.

Si on VEUT que les modifications effectuées sur les paramètres formels soient effectives c'est IMPOSSIBLE.

Attention : Que faire si on désire modifier quand même ce qui est transmis ?

Dans ce cas, puisque l'unique mécanisme de transmission d'information est la copie, il suffit de transmettre la copie de l'adresse de ce que l'on VEUT modifier.

Ensuite dans le bloc de la fonction il suffit d'utiliser l'opérateur d'accès au contenu '*' pour modifier ce qui est pointé. On ne touche pas à la valeur du pointeur qui est une copie d'information mais on l'utilise pour modifier ce qui est pointé. Nous en avons le droit car nous utilisons le mécanisme d'indirection.

Exemple

Par exemple, la séquence d'instructions pour permuter le contenu de deux variables est :

```
void main( )
{
      int x, y, inter;
       x = 3 ;
       y = 0 ;
      inter = x;             // sauvegarde de la valeur de
x
      x = y ;                   // x prend la valeur de y
mais l'ancienne valeur de x a été sauvegardée
      y = inter;             // a ce stade les contenus de
x et y ont été échanges.
}
```

Exemple

Si l'on désire construire une fonction pour faire cela et que l'on écrit :

```
void permute(int x, int y)
{
      int inter;
      inter = x;
     x = y ;
      y = inter;
}
void main( )
{
      int x, y, a, b ;
      x = 12 ;
      y = 24 ;
      a = 2 ;
      b = 4 ;
      permute(x,y) ;
      permute(a,b) ;
}
```

Remarque

Toutes les variables locales de main conservent leur valeur initiale, car nous avons

transmis des copies.

Par contre si on écrit :

```
void permute(int * x, int * y)
            // x est un paramètre formel de type pointeur
vers entier
            // y est un paramètre formel de type pointeur
vers entier
{
       int inter;
       inter = *x; // on accede a ce qui est pointe par x
                                  // la variable locale inter
contient l'entier trouve.
                                  // a l'adresse qui est dans
x
       *x = *y ;     // de meme pour y
       *y = inter; // y est un pointeur vers entier, nous
modifions ce
                                  // qui pointe en y rangeant
la valeur de inter
}
void main( )
{
      int x, y, a, b ; // x,y,a,b sont ici des variables
locales de la
                                        // fonction main,
ce sont des entiers.
      x = 12 ;
      y = 24 ;
      a = 2 ;
      b = 4 ;
      permute(&x,&y) ;// recopie de l'adresse de x et de
l'adresse de y
                                         // tous deux entiers,
ces valeurs d'adresses sont les
                                         // deux valeurs des
paramètres réels recopies dans
                                         // les deux
réceptacles des paramètres formels x et y
                                        // de la fonction
permute
                                         // qui eux sont des
réceptacles qui contiennent désormais
                                        // des valeurs de
pointeur vers entiers !
      permute(&a,&b) ;
}
```

Les contenus des variables locales « x » et « y » de « main » et « a » et « b » ont été échangés. Nous avons transmis des copies des adresses de « x » et « y » et la fonction travaille avec l'opérateur « * » qui accède à ce qui est pointé.

De même pour « a » et « b ».

Complément : Conclusion

- On peut modifier ce qui est transmis dans une fonction cela n'a aucun impact, car c'est une copie d'information
- Mais si on transmet une adresse, on peut l'utiliser pour accéder à l'emplacement qui correspond.

Les mêmes règles s'appliquent dans le cas de paramètres de type « pointeur vers

». Nombre identique de paramètres réels/formels, concordance de type. Par exemple, si la fonction attend un « pointeur vers flottant » il faudra lui transmettre « un pointeur vers flottant ».

Vous remarquerez que la fonction "echange" attend deux pointeurs vers entier et qu'on l'utilise en lui fournissant deux « pointeurs vers entier », c'est cohérent, il y a concordance de type.

Attention

c'est le programmeur qui décide, s'il transmet ou non des pointeurs quand il conçoit sa fonction.

Si l'on VEUT modifier quelque chose transmis à une fonction, alors il FAUT impérativement passer les paramètres par adresse (pointeur).

Les modifications du contenu en utilisant leurs adresses transmises en paramètre sont effectives, quand la fonction est terminée ces modifications sont pérennes. Attention à ce que vous faites :

Exemple

```
int plus(int * x) // transmission d'un pointeur
{
      * x = * x + 1 ; // utilisation du pointeur pour
modifier
      return ( * x) ; // ce qui est pointé par (mécanisme
d'indirection)
}
void main( )
{
      int x, y;
      x = 7;
      y = plus(&x) ;
}
```

A la fin du programme x vaut 8 et y aussi.

6. Transmission d'un tableau en paramètres

a) Cas des tableaux à une dimension

Fondamental

Puisque l'identificateur du tableau déclaré est un pointeur constant, il suffit de dire que l'on utilise un pointeur pour transmettre un paramètre de type `tableau' dans une fonction.

Syntaxe

Nous supposons que les tableaux « a », « b » et « c » ont été déclarés comme :

```
int a[MAX], b[MAX], c[MAX];
```

Et que MAX soit déclaré comme :

```
#define MAX 10
```

au début du programme par exemple.

Ci-dessous des exemples de déclarations de l'en-tête d'une fonction qui attend des tableaux en paramètres formels :

```
void somme_vect(int v[MAX], v1[MAX], v2[MAX])
void somme_vect(int v[], v1[], v2[])
void somme_vect(int *v, *v1, *v2)
```

Toutes ces notations sont équivalentes pour l'en-tête.

Pour les tableaux à une seule dimension nous n'avons pas besoin de spécifier la valeur MAX.

Ensuite le bloc de la fonction est :

```
{
   int i;
    for (i=0;i<MAX;i++) // ici MAX est connue grâce à la
pré-compilation
                                                  // cela n'a
rien a voir avec le v[MAX] d'un
                                                  // des en-têtes
proposés
         {
                v[i] = v1[i]+v2[i];
         }
}
```

Rappel

Nous n'avons pas besoin de connaître la taille d'un tableau à une dimension (un vecteur). Nous avons juste besoin de savoir où il se trouve.

En effet, nous vous rappelons que :

```
v[i] = v1[i]+v2[i]
*(v +i) = *(v1 + i)+ *(v2 +i)
```

C'est strictement la même chose (voir cours précédent sur les tableaux).

Exemple

Ci-dessous , des exemples d'invocations de la fonction avec les tableaux en paramètres (en supposant que les tableaux « b » et « c » ont été correctement initialisés)

```
somme_vect( &a[0], &b[0], &c[0]);
somme_vect(a, b, c);
```

Remarque

Nous remarquons au niveau des deux appels de la fonction "somme_vect" que par exemple "&a[0]" dans le premier et que "a" au niveau du deuxième appel correspondent à une valeur qui est un pointeur. Ce qui correspond à l'en-tête de la fonction que nous avons définie.

Quelle que soit l'utilisation dans la fonction, on transmet toujours l'adresse du 1er élément du tableau. Dans le corps de la fonction (le bloc), nous pouvons donc utiliser par exemple v[i_local] ce qui, nous l'avons vu, permet d'accéder au contenu de la case et de le modifier.

Puisque le paramètre formel "v" contient une copie de "a" le pointeur constant qui est l'adresse du début de la zone mémoire allouée au sein de fonction principale « main », nous modifions directement le tableau pointé par "a".

b) Cas des tableaux à deux dimensions

Exemple

Soit :

```
#define MAX_L 2
#define MAX_C 3
int t[MAX_L][MAX_C];
int i,j;
```

Si vous voulez utiliser les notations simples t[i][j] dans la fonction il faut indiquer au compilateur au minimum le nombre maximum de colonnes pour qu'il puisse générer les calculs d'adresses « ad hoc » (décalages de lignes).

A cet effet, nous vous rappelons que tab[i][j] est transformé.

Le compilateur calcule l'expression mathématique suivante pour avoir l'adresse :

```
t + ( i x MAX_C + j) x Taille avec Taille = sizeof(int)
```

Le nombre de colonnes MAX_C est donc absolument nécessaire pour que la fonction puisse se compiler.

Ci-dessous, des exemple d'en-tête de fonctions avec passage d'un tableau 2D en paramètre :

Exemple

```
void aff(double b[MAX_L][ MAX_C])
void aff(double b[][ MAX_C])
```

Les deux notations sont équivalentes et suffisantes. Ensuite nous avons le bloc suivant :

```
{
  int i,j;
  for (i=0;i< MAX_L;i++)
      for (j=0;j<MAX_C;j++)
      {
          printf("\n tableau[%d][%d] = %4.3f",i,j,b[i][j]);
      }
}
```

Ci-dessous, des exemple de l'appel de la fonction précédente avec les déclarations suivantes :

Exemple

```
#define MAX_L 2
#define MAX_C 3
double td[MAX_L][MAX_C] = {{37.2,37.5, 20.8},{38.4, 40.5,
43.2}};
```

L'appel de la fonction est :

```
aff(td);
```

7. Prototypage de fonction

Considérons le programme suivant :

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

```
int plus_un(int a)
{
    int inter;
/* point point_fonc */
     inter = a+1;
    return inter;
}
void main()
{
    int x,y;
/* point 1 */
    x = plus_un(7);
/* point 2 */
     y = plus_un(x);
/* point 3 */
}
```

Complément

Nous avons vu précédemment les vérifications effectuées par le compilateur au moment de l'invocation de la fonction « plus_un(7) » et « plus_un(x) » sur :

- le nombre de paramètres réels qui doit être le même que le nombre de paramètres formels déclarés
- les types des paramètres réels utilisés pour l'appel de la fonction qui doivent être compatibles avec les types des paramètres formels déclarés. C'est la vérification de la concordance de types.

La compilation échoue sinon.

C'est une vérification syntaxique.

Ensuite et seulement ensuite, les mécanismes de création de contexte, de recopie, de branchements, de récupération d'informations, et de branchement de retour sont mis en place par le compilateur.

En fait, la compilation effectue d'abord la vérification syntaxique, puis après avoir fait toutes les vérifications elle ré-effectue une « passe » sur le code pour la mise en place des mécanismes. Cette technique est plus efficace que de faire du travail qui sera éventuellement annulé s'il y a une erreur de syntaxe qui fait échouer la compilation.

Remarque

Nous pouvons remarquer que seul l'en-tête de la fonction est nécessaire pour les vérifications. Nous pouvons donc inverser l'ordre d'écriture des fonctions « main » et « plus_un » à condition que l'on fournisse au compilateur un « prototype » de la fonction qui lui permet de faire son travail de vérification préalable.

Exemple

Voici le code utilisant cette technique :

```
#include <stdio.h>
int plus_un(int a);
                    // prototypage de la fonction, c'est
juste l'en-tête
                    // suivi d'un délimiteur « ; »
void main()
{
     int x,y;
/* point 1 */
```

```
    x = plus_un(7);
// ici le compilateur peut vérifier
// la syntaxe de l'invocation
/* point 2 */
    y = plus_un(x);
// ici le compilateur peut vérifier
// la syntaxe de l'invocation
/* point 3 */
}
int plus_un(int a)
                    // ici nous déclarons réellement la
fonction
                    // le compilateur « recollera » les
morceaux pour vous.
{
int inter;
/* point point_fonc */
    inter = a+1;
    return inter;
}
```

Complément

La règle « on ne peut utiliser quelque chose que si cette chose a été préalablement déclarée » est ainsi respectée.

Une fonction doit être déclarée avant d'être utilisée, nous avons désormais deux solutions :

- déclarer entièrement une fonction avant son invocation
- déclarer le prototype de la fonction avant son invocation, ce qui rend possible la vérification, puis déclarer la fonction plus tard.

L'inclusion d'un fichier comme « stdio.h » permet de déclarer les prototypes des fonctions. Ces fichiers d'en-tête (nous les avons d'ailleurs appelés ainsi dans le chapitre 1) contiennent les prototypes de fonctions (des en-têtes avec le « ; ») dont les codes exécutables sont dans les bibliothèques de codes pré-compilées.

8. Exemples divers

Complément

Voici quelques exemples de fonctions utiles sur les caractères disponibles dans « string.h » :

- conversion : atoi(s), toupper(c) , tolower(c)
- test : isalpha(c) , isupper(c), isdigit(c)

« s » est une chaîne de caractères, et « c » un caractère.

- atoi(s), signifie ascii to integer, atoi("123") retourne l'entier 123 codé sur un int.
- toupper(c) retourne le caractère majuscule si la conversion est possible
- tolower(c) retourne le caractère minuscule si la conversion est possible
- isupper(c) retourne vrai si c est dans 'A' .. 'Z'
- isdigit(c) retourne vrai si c est dans '0' ... '9'
- isalpha(c) retourne vrai si c est dans 'A' .. 'Z', 'a' ..., 'z', ou '0' ... '9'

Et il y en a beaucoup d'autres ...

Exemple : Exemples de fonction

Minimum de deux nombres

```
#include < stdio. h >
int min(int i, int j)
{
  if (i < j )
      return i ;
  else
      return j ;
}
void main( )
{
  int x, y, z ;
  printf ("entrez deux nombres : " ) ;
  scanf("%d", & y) scanf("%d", & x);
  z = min(x, y) ;
}
```

Puissance d'un nombre positif ou nul :

```
double puiss(double x, int y)
{
  double p ;
  int i ;
  p = 1;
  for (i = 0 ; i < y ; i ++)
      p = p * x; return p ;
}
void main( )
{
  double m ;
  m = puis(10.5, 3) ;
}
```

Remarque

La fonction « pow(x,y) » existe dans « math.h » et elle est plus efficace.

9. Portée des identificateurs, scope lexical

a) Introduction

Rappel

Nous avons vu que l'on pouvait déclarer des identificateurs de variables ou de paramètres formels en différents endroits du code. La question qui se pose est : « est-ce qu'un identificateur déclaré à un endroit du code peut être utilisée à un autre ? »

Cette question est importante en particulier quand une fonction appelle elle-même une fonction.

Définition

Nous introduisons un ensemble de notations qui permettent de mieux comprendre le déroulement d'un programme.

A tout identificateur, nous attribuons un quadruplet qui définit son état (visible, nature, type, valeur) :

Ce quadruplet prend différentes valeurs tout au long du déroulement du

programme, ce qui permet de tracer et suivre l'état d'un identificateur à différents points d'observation au sein du code.

NB : par la suite {} signifiera ensemble vide, ou rien, ou aucune valeur.

Complément

Le premier élément "Visible" prend ses valeurs dans l'ensemble {V, NV} :

- "V" signifie "visible" : en un point d'observation précis du code pendant son fonctionnement, si l'on essaie d'insérer un "printf" qui affiche le contenu de cet identificateur alors le compilateur "connaîtra" cet identificateur et acceptera de compiler l'appel de "printf" avec cet identificateur comme argument ;
- NV" signifie "non visible" : au point d'insertion du "printf" dans code, le compilateur refusera de compiler, il génèrera une erreur de compilation car l'identificateur ne sera pas connu.

Complément

Le deuxième élément "Nature" prend ses valeurs dans { {}, VG, VL de quoi, PF de quoi, PR de quoi}

- "{}" signifie que l'identificateur n'a pas de nature, cela ne peut être qu'un identificateur non visible
- "VG" signifie variable globale, elle est a priori connue en tout point du code après sa définition, le code est vu comme le "bloc" principal, qui englobe tous les autres blocs
- "VL de quoi" signifie que l'identificateur est celui d'une variable locale et il FAUT OBLIGATOIREMENT préciser à quel bloc ou fonction cet identificateur est local. Dans ce cas, l'identificateur est connu UNIQUEMENT au sein du bloc où se trouve sa déclaration ET dans tous les blocs ou appels de fonctions inclus dans le bloc où se trouve sa déclaration. Cependant, si un identificateur est défini dans un bloc qui utilise une fonction alors cet identificateur ne sera pas visible dans la fonction.
- "PF de quoi" signifie identificateur de paramètre formel. Ce type d'identificateur ne peut se trouver qu'au niveau de la déclaration d'une fonction, c'est-à-dire dans l'en-tête de la fonction et dans le bloc de la fonction :
  - dans l'en-tête de la fonction : l'identificateur donne formellement un nom à une information, une donnée qu'il faudra transmettre pour pouvoir utiliser la fonction. Il permet de donner un identificateur à une donnée dont on ne connaît pas encore la valeur, c'est donc un identificateur "formel"
  - dans le bloc de la fonction : l'identificateur est utilisable dans des instructions et expressions au sein du bloc de la fonction. Il faut donc préciser "de quoi", c'est à dire à quelle fonction correspond le paramètre formel.
- "PR de quoi" signifie identificateur de paramètre réel. Au niveau de l'utilisation d'une fonction, c'est soit la valeur du littéral, soit la valeur de la variable, soit la valeur de l'expression, soit la valeur d'un paramètre formel (cas d'une fonction qui en appelle une autre) qui sera recopiée dans le PF associé en respectant l'ordre de la déclaration de l'en-tête de la fonction.

Complément

Le troisième élément "Type" prend ses valeurs dans { {}, type simple, constructeur de type, type défini par le programmeur}. A ce stade seul "{}" et "type simple" ont été vus dans ce cours, les autres points seront vus dans les chapitre suivants :

- "{}" signifie que l'identificateur n'a pas de type, cela ne peut être qu'un identificateur non visible
- "type simple" : exemple : int, float, double, char, etc.
- "constructeur de type", "type défini par le programmeur" : attendre le cours sur struct et typedef pour revenir sur cette partie

Complément

Le quatrième élément "Valeur" : prend ses valeurs dans { {}, < contenu du réceptacle associé à l'identificateur > }

- "{}" signifie que l'identificateur n'a pas de valeur, soit l'identificateur est non visible, soit la variable n'a pas été initialisée, dans ce dernier cas, on ne sait pas ce qu'elle contient, ce que l'on assimilera à "{}" pour simplifier
- "contenu du réceptacle associé à l'identificateur" au point d'observation de l'identificateur, c'est la valeur qu'il a, c'est au programmeur (vous) de savoir exactement quelle est cette valeur en ce point précis du code.

b) Règles très importantes

Attention

R1 : C'est la déclaration la plus locale de l'identificateur qui est dominante et qu'il faut considérer en un point d'observation donné.

Attention

R2 : un identificateur est connu au sein du bloc où se trouve sa déclaration ET dans tous les blocs inclus dans le bloc où se trouve sa déclaration.

Exemple : Exemple 1

C'est un exemple qui utilise uniquement le bloc principal de la fonction main Code :

```
1 --> #include < stdio.h >
2 --> int un_entier = -3; // déclaration d'une variable
globale (VG)
3 --> void main()
4 --> { /* p1 */
5 --> char carac; // déclaration de la variable locale à la
fonction main "carac" (VL)
6 --> /* p2 */
7 --> carac = getchar();
8 --> un_entier = 1; /* p3 */
9 --> printf("\nle caractere est %c et l'entier est
%d\n",carac,un_entier);
10--> /* p4 */ }
```

Nous supposerons que l'utilisateur tape 'A'.

Voici le contenu des quadruplets aux points p1, p2, p3, p4:

	un_entier	**carac**
p1	(V, VG, int, -3)	(NV, {} , {} , {})
p2	(V, VG, int, -3)	(V, VL fonction main, char, {})
p3	(V, VG, int, 1)	(V, VL fonction main, char, 'B')
p4	(V, VG, int, 1)	(V, VL fonction main, bloc principal, char, 'B')

Tableau 6 Quadruplets aux points p1, p2, p3 et p4

i - Explications

Au point p1, seule la variable globale 'un_entier' est déclarée et est visible, la

variable dont l'identificateur est 'carac' n'est pas encore définie, elle est non visible. Une variable globale est définie et visible partout à partir de l'emplacement où elle est déclarée.

Au point p2, l'identificateur de variable 'carac' est défini mais la variable n'a pas de valeur, elle n'a pas encore été initialisée.

Au point p3, 'un_entier' vient d'être modifié et 'carac' aussi.

Au point p4, juste avant de sortir du programme, les variables conservent l'état précédent, elles ont juste été affichées.

c) Autres exemples plus compliqués

Exemple

```
#include <stdio.h>
float y ; // c'est une variable globale définie en dehors
de tout bloc
                    // de fonction
void plus_un(float * y) // "y" est un PF de la fonction
plus_un
                                                    // qui
contiendra la valeur d'une adresse
                                                    // vers un
flottant
{
// point P :
*y = *y + 1;
                    // printf("%f",x); // a dé-commenter
pour essayer, cela génère une erreur
                    // l'identificateur "x" n'est pas connu
en ce point. C'est une VL de main.
}
Float cube(float x) // "x" est un PF de type float de la
fonction cube
                                          // qui contiendra la
valeur d'un flottant
{
int y ;                     // "y" est une variable locale
de la fonction cube
                                    // tout identicateur "y"
défini avant n'existe plus
                                    // seule la déclaration
la plus locale compte donc la VG "y"
                                    // n'est plus visible,
c'est la définition locale qui domine.
// point C1
plus_un(&x);         // "x" est ici le PF de cube, c'est un
réceptacle,
                                    // son adresse existe, il
est utilisé comme PR
                                    // de la fonction
"plus_un". Le quadruplet de "&x" est donc
                                    // ici (V, PR plus_un,
float, adresse de x PF de cube)
                                    // la valeur de "adresse
de x PF de cube" ne peut être
                                    // déterminée, elle
dépend de l'exécution, cela n'a pas
                                    // d'importance car ce
n'est pas l'adresse de "x" PF de cube
```

```
                                    // que l'on modifie mais
"x" la valeur du PF de cube
                                    // grâce à l'utilisation
de son adresse.
y = x*x*x;              // il y a un cast automatique
// point C2
         //printf("%f",z); // a dé-commenter pour essayer,
cela génère une erreur
         // car l'identificateur "z" n'est pas défini en ce
point, c'est une VL
         // de la fonction "main" qui n'existe pas ici
return (float)y;    // il y a un cast explicite
}
int main(void)
{
float x,z;          // x est une variable locale de la
fonction main
                                // z est une variable locale
de la fonction main
                                // point M1 x = cube(3.0);
                                // le PR paramètre réel de
cube est le littéral constant 3.0
                                // point M2 y = cube(x);
                                // le PR paramètre réel de
cube est la valeur de la VL "x" de main
// point M3
{ int t;            // un bloc dans un bloc avec "t" une
nouvelle VL du nouveau bloc
z= 4.0;
printf("%f",z); // fonctionne car "z" est définie dans le
bloc englobant
}
         // printf("%d",t); a dé-commenter pour essayer,
cela génère une erreur
         // car l'identificateur "t" n'est plus défini en
ce point du code
}
```

Comme on peut redéfinir localement un identificateur, on peut donc changer son type.

Quand le programme s'exécute nous passons dans l'ordre par les points :

M1 : fonction main

C1 : fonction cube

P : fonction plus_un

C2 : fonction cube

M2 : fonction main

C1 : fonction cube

P : fonction plus_un

C2 : fonction cube

M3 : fonction main

Nous ne nous préoccupons pas des identificateurs "z" et "t" pour l'exécution de ce code, ils ont été ajoutés pour mettre en évidence la notion de "visible", si l'on dé-commente les lignes qui les affichent le compilateur génère une erreur et refuse de compiler. Ils ne sont en effet pas visibles au niveau des "printf" mis en commentaires.

Nous nous intéressons aux deux identificateurs "x" et "y". Quel quadruplet leur associer en chacun des points d'observation ?

Le tableau suivant se lit de haut en bas et suit l'exécution du code :

	x	y
M1	(V, VL main, float, {})	(V, VG , float , {})
C1	(V, PF cube, float, 3.0)	(V, VL cube, int, {})
P	(NV, {}, {}, {})	(V, PF plus_un, float, &x PF de cube)
C2	(V, PF cube, float, 4.0)	(V, VL cube, int, 64)
M2	(V, VL main, float, 64.0)	(V, VG , float , {})
C1	(V, PF cube, float, 64.0)	(V, VL cube, int, {})
P	(NV, {}, {}, {})	(V, PF plus_un, float, &x PF de cube)
C2	(V, PF cube, float, 65.0)	(V, VL cube, int, 274625)
M3	(V, VL main, float, 64.0)	(V, VG , float , 274625.0)

Tableau 7 Exécution du Code source

A vous de suivre pas à pas le déroulement du code en appliquant les règles.

Remarque : Indication

Au point P, "y" est le PF de "plus_un" qui contient une valeur qui est un "pointeur vers flottant".

Au point C1, "x" est l'identificateur du PF de "cube" qui contient une valeur flottante.

Le réceptacle de "x" existe, il est créé dynamiquement au moment de l'appel de "cube", donc on peut calculer son adresse.

Attention

nous rappelons que les PF et VL d'une fonction n'existent qu'au moment où la fonction est invoquée et pendant son déroulement, ni avant l'appel de la fonction, ni après l'appel de la fonction (car ils sont dynamiquement créés puis détruits).

Conseil

- Il est très dangereux de modifier la valeur d'un identificateur qui n'est pas localement défini. Par exemple, modifier dans une fonction la valeur d'une variable globale. Cela peut conduire à des erreurs graves de programmation. Cette pratique est à proscrire.
- Les fonctions doivent être conçues comme des entités de traitement indépendantes, elles reçoivent des paramètres, utilisent éventuellement des variables locales pour effectuer leurs traitements et retournent une valeur.
- Si vous avez, en cours de programmation, un doute sur l'existence d'un identificateur en un point du programme, insérez un "printf" pour essayer de voir la valeur de cet identificateur. Si ce dernier n'est pas visible alors le compilateur refusera de compiler le code. En effet, cet identificateur ne sera alors pas connu et la règle "on ne peut utiliser quelque chose que si cette chose est préalablement déclarée" sera alors violée. Ce que détectera le compilateur

10. Exercices de Révision (QCM)

a) Partie 1

Exercice 1

1. Une fonction renvoie toujours une valeur.

- ○ Vrai
- ○ Faux

Exercice 2

2. Lesquelles de ces définitions de fonctions sont correctes si elles sont sensées calculer « x » élevé à la puissance « N » entière. ?

- ☐ float puissance (float X , int N)
- ☐ puissance (float X ; int N)
- ☐ puissance (float x , int N) float
- ☐ puissance() ;
- ☐ puisance : float ;
- ☐ void puissance (float X , int N)

Exercice 3

3. Considérons le programme suivant :

```
#include <stdio.h>
/*Program essai */
float x ;
void Affectation() {
float x =5 ;
}
void main() {
x = 3 ;
Affectation ;
printf("%d",x) ;
}
```

Quelle est la sortie à l'écran ?

- ○ 3
- ○ 5
- ○ Rien

Exercice 4

4. Il préférable d'utiliser des variables:

- ○ Locales.
- ○ Globales.

Exercice 5

5. Où sont déclarées les variables locales ?

- ○ En début du programme principal.
- ○ Au début de chaque fonction où elles interviennent.
- ○ Elles sont passées en paramètres dans l'énoncé de la fonction.

Exercice 6

6. Quelles sont les manières correctes pour passer le tableau « tab » en paramètre ?

- ☐ void proc(int *tab)
- ☐ void proc(int tab[])
- ☐ void proc(int &tab)

Exercice 7

7. Dans un programme, x est définie globalement. Dans un bloc (fonction) de ce même programme, x est définie localement. Lorsqu'on se trouve dans ce dernier bloc, quelle variable prédomine ?

- ○ La variable x locale.
- ○ La variable x globale.

Exercice 8

8. Une fonction renvoie toujours une valeur.
Comment cela se programme-t-il, à l'intérieur de la fonction ?

- ○ En précédant la valeur à retourner par le mot-clé return.
- ○ En affectant à l'identificateur de fonction la valeur à retourner.
- ○ langage C renvoie toujours la dernière valeur calculée dans la fonction.

b) Partie 2

Exercice 1

1. Peut-on donner un même identificateur à une variable locale et à une variable globale au cours d'un même programme ?

- ○ Oui
- ○ Non

Exercice 2

2. Choisissez le mode de transmission de X et Y à partir de l'expression suivante :
*void Fiche(float *X, float *Y, int i, char Z, char R) ;*

- ○ Transmission de valeur flottantes
- ○ Transmission d'adresse de flottantes

Exercice 3

3. Une variable globale est-elle active dans une fonction ?

- ○ Oui, toujours.
- ○ Oui, sauf si une variable locale du même nom a été définie dans ce bloc.
- ○ Non, jamais.

Exercice 4

4. On considère l'appel suivant :

echange(A, B) ;

Pour qu'il y ait inversion des valeurs des variables A et B dans le programme principal, on utilise :

- ○ Un passage par valeur.
- ○ Un passage par adresse.

Exercice 5

5. Où sont déclarées les fonctions?

- ☐ Après la fonction principale « main », si les prototypes sont définis avant.
- ☐ Au cours du programme principal.
- ☐ Avant le programme principal et la règle "une fonction doit être déclarée avant usage" doit être respectée.

Exercice 6

*6. Considérons l'en-tête de fonction suivante : void Fiche(float *X, float *Y, int i, char Z, char R) ;*

Considérons les déclarations suivantes :

float A, C ;

int J ;

char B, H ;

Quels sont les appels de fonction justes ?

- ☐ Fiche (A,C ; J ; B, H) ;
- ☐ Fiche (&A, &B, C, J, H) ;
- ☐ Fiche (&A,&C, 3, 'b', B) ;
- ☐ Fiche (&A, &C, J, B, H) ;
- ☐ Fiche (A ; J ; B, H) ;

Exercice 7

7. On considère les déclarations suivantes :

```
#include <stdio.h>
/*Program calcul */
int x , y ;
```

int triple(int z) {
*return z*3 ;*
}
main() {
(Programme)
}
Est-ce que "z" est une variable ?

○	Oui
○	Non

Exercice 8

8. L'appel d'une fonction de type « void » et d'une fonction de type « int » dans le programme principal se fait-il de la même manière ?

○	Oui
○	Non

H. Les structures

1. Définition

Définition

Une structure est un agglomérat hétérogène de composants de types différents regroupés sous un même nom. Chaque composant de la structure est appelé champ (ou encore attribut) et est accessible par un nom (identificateur).

L'ensemble est accessible par le nom de la structure.

Exemple : Exemple 1 :

Une structure Date est un agglomérat du jour, du mois et de l'année nom de la structure : date

- champ 1 jour : 3 (une des valeurs possibles)
- champ 2 mois : 12 (une des valeurs possibles)
- champ 3 année : 1999 (une des valeurs possibles)

Exemple : Exemple 2 :

Une structure Etudiant est un agglomérat de son Nom, son Prénom et de sa moyenne nom de la structure : etudiant

- champ 1 Nom : DUPOND (une des valeurs possibles)
- champ 2 Prénom : Eric (une des valeurs possibles)
- champ 3 moyenne : 17.8 (une des valeurs possibles)

2. Différence entre une structure et un tableau

Fondamental

Un tableau permet de regrouper et manipuler des éléments de mêmes types, c'est-à-dire codés sur le même nombre d'octets et de la même façon.

Les structures regroupent des champs (ou attributs) de natures différentes au sein d'une entité repérée par un seul nom de variable.

Cette variable structurée occupe une place mémoire fixe en mémoire.

3. Déclaration d'une structure

Syntaxe

Pour illustrer la déclaration, nous reprenons les exemples ci-dessus. Nous déclarons ainsi :

```
#define MAX 20
struct date
{
                                    int jour, mois, annee;
};
struct etudiant
{
                                    char nom[MAX] ;
                                    char prenom[MAX];
                                    float moyenne;
} ;
```

A partir de ces deux exemples, nous pouvons déduire que la déclaration d'une structure s'écrit ainsi :

```
struct identificateur_de_la_structure
{
              type_champ1 Nom_Champ1;
              type_champ2 Nom_Champ2;
              . . .
};
```

Les champs sont vus comme des variables liées à la structure. Ils correspondent donc à des couples (identificateur, réceptacle). Les identificateurs de champs doivent être construits comme des identificateurs classiques.

Vous noterez qu'après la '}' fermante il y a obligatoirement un ';' qui termine la déclaration.

Attention

ces déclarations de structures ne sont pas des déclarations de variables. Nous déclarons en fait la façon de construire une variable structurée.

Cette déclaration explique au compilateur comment réserver l'espace mémoire pour le réceptacle correspondant.

Nous pouvons dire que ces déclarations de structures sont des déclarations de « constructeurs de variables »

4. Déclaration de variables structurées et initialisation à la déclaration

Exemple

Ci suivent des exemples possibles de déclarations de variables structurées :

```
struct date d1, d2={01, 01, 2009};
                                    // d1 et d2 sont deux
variables structurées construites avec la
                                   // déclaration de
structure
                                    // seule d2 est
initialisée avec une date
struct date *pt_d ; // variable de type pointeur vers
structure
struct etudiant genie = {"vous","non ?",18.7}, cancre =
{"l'autre","c'est normal",3.2}, * pt_etu;
```

Nous utilisons des listes d'initialisation comme pour les tableaux, mais vous remarquez bien dans ces exemples que les valeurs sont de types différents. Vous remarquerez également que chaque liste comporte une valeur par champ. Champs et valeurs sont appariés dans l'ordre des champs. A ce stade, "pt_d" et "pt_etu", les deux pointeurs vers des variables structurées, ne sont pas initialisés. Nous pouvons ensuite écrire les instructions :

```
pt_d = &d1;
pt_etu = &genie;
```

Ce qui initialise les pointeurs avec des adresses de variables compatibles avec leurs déclarations. Le compilateur vérifie là encore la concordance de type. Nous pouvons déclarer la structure et déclarer en même temps des variables comme dans l'exemple qui suit :

```
struct etudiant
{
             char nom[MAX], prenom[MAX];
             float moyenne;
} jean, * paul, philippe, amir;
```

Ainsi les variables structurées "jean", "philippe" et "amir" seront définies, de même que la variable "pointeur vers" une variable structurée "paul".

Conseil

Nous vous déconseillons cette déclaration simultanée structure-variable, vous risquez de confondre la déclaration de structure (la façon de construire une variable structurée) et la déclaration de variables structurées pour lesquelles le compilateur va utiliser la première déclaration.

5. Accès aux champs d'une structure et affectation

Syntaxe

Nous pouvons accéder aux champs d'une structure soit en utilisant l'identificateur d'une variable structurée soit en utilisant l'identificateur d'un pointeur vers une variable structurée à condition que ce dernier ait été correctement initialisé.

Pour accéder au champ <truc> d'une variable strcuturée <identi_var>, il faut utiliser la syntaxe :

```
<identi_var>.<truc>
```

C'est le "." qui permet d'accéder au champ à partir de l'identificateur <identi_var>.

Si on accède au champ <truc> à partir d'un identificateur <identi_pt> de variable qui pointe vers une variable structurée, alors il faut utiliser la syntaxe :

```
<identi_pt> -> <truc>
```

C'est le "->" qui permet d'accéder au champ à partir de l'identificateur <identi_pt>.

Exemple

La syntaxe se différentie donc, considérons le code suivant :

```
#define MAX 20
struct etudiant
{
      char nom[MAX], prenom[MAX];
      float moyenne;
} ;
void main()
{
// déclaration de "e" et "a" : variables structurées, et
"pt_e" comme variable
// de type pointeur vers variable structurée de nature
"etudiant".
// seule "e" est initialisée à la déclaration.
struct etudiant e = {"jean","seigne",18.8}, a, *pt_e;
printf("%s", e.nom);
printf("%s", e.prenom);       // syntaxe avec "." car nom
de variable printf("%f", e.moyenne);
pt_e = &e;
       //l'adresse de la variable structurée « e » est
rangée dans
      // la variable « pointeur vers variable structurée »
« pt_e ».
printf("%s", pt_e->nom);
printf("%s", pt_e->prenom);       // syntaxe avec "->" car
« pointeur vers »
printf("%f", pt_e->moyenne);
}
```

Exemple

La saisie des champs suit les mêmes règles que la saisie de variables. Exemple d'instructions :

```
e.moyenne = 10.0;
strcpy(e.nom, "poubot");
scanf("%s", e.prenom);       // on suppose que "paul" soit
saisi
```

A ce stade, la variable structurée « e » contiendra comme valeurs de champs : "poubot" pour le nom, "paul" pour le prénom, et 10.0 pour moyenne.

Si on effectue ensuite les instructions suivantes :

```
pt_e->moyenne = 19.0;       // sachant que pt_e=&e comme
dans le code ci-dessus
scanf("%s", pt_e->nom);  // on suppose que "snoopy" soit
saisi
```

A ce stade, la variable structurée "e" contiendra comme valeurs de champs : "snoopy" pour le nom, "paul" pour le prénom, et 19.0 pour moyenne.

Vous remarquerez que le champ "nom" correspond à l'identificateur d'un pointeur constant car c'est un tableau de caractères. Pour saisir une chaîne de caractères la fonction "scanf" attend un pointeur, et c'est bien un pointeur qu'on lui transmet.

Peu importe que l'on écrive :

```
« e.nom » ou « pt_e->nom »
```

Ce qui importe c'est ce que représente le champ.

Par contre, si l'on veut modifier caractères par caractères, alors on accède individuellement à des cases du tableau de caractères. Ainsi :

```
e.nom[3] = 'v';
pt_e->nom[1] = 'y';
scanf("%c", &(e.nom[2]);        // on saisie 'l'
scanf("%c", &(pt_e->nom[4])     // on saisie 'e'
pt_e->nom[5] = '\0';
```

Si l'on affiche :

```
printf("%s", e.nom);
```

Nous obtiendrons "sylve"

En résumé, si un des champs est un tableau, il se manipule comme tel.

On peut aussi affecter une variable structurée à une autre de même nature. L'instruction :

```
a = e;
```

est possible. Elle recopie tous les champs de la variable "e", dans les champs de la variable "a" (Cf déclaration de "a" ci-dessus).

Attention

Si « a » et « e » ne sont pas des variables structurées de même nature cela provoquera une erreur. Ne pas oublier que le compilateur vérifie la concordance des types.

On peut s'interroger sur le pourquoi des deux syntaxes "." et "->" ? La raison est qu'il est plus lisible d'écrire :

```
pt_e->moyenne
```

que

```
(*pt_e).moyenne
```

Surtout dans le cas des saisies :

Il est plus lisible d'écrire :

```
scanf("%d",&( pt_e -> moyenne);
```

Que :

```
scanf("%d",&((*pt_e).moyenne));
```

Bien que ces écritures désignent la même chose pour le compilateur.

6. Erreurs à ne pas commettre

La déclaration de structure suivante est incorrecte :

```
struct personne
{
     int age;
     char age;
     struct personne pere;
};
```

Pour deux raisons :

- Premièrement le nom de variable "age" n'est pas unique, deux champs ont

pour identificateur "age". Là, je ne vois pas trop l'intérêt ? Corrigeons cette première erreur :

```
struct personne
{
     int age;
     struct personne pere;
};
```

- Il y a toujours une erreur car "struct personne pere" n'est pas autorisé. La déclaration d'une structure ne fait que donner la façon de construire la structure. On cherche à utiliser "struct personne pere", c'est-à-dire que l'on veut un champ "pere" lui-même avec un champ "age" et un champ "pere". Puisque la déclaration n'est pas terminée, le compilateur ne peut pas savoir comment construire le champ "pere".

Par contre cette écriture est autorisée :

```
struct personne
{
     int age;
     struct personne * pt_pere;
};
```

Dans cette écriture, le champ "pt_pere" est un pointeur, et le compilateur sait construire un réceptacle pour un pointeur. Nous avons vu au chapitre 3, que toutes les informations de type "pointeur vers" ont la même taille de réceptacle, car les valeurs qui y sont rangées sont des adresses mémoire. Il en est donc de même pour les pointeurs vers les variables structurées.

Attention : Erreur plus difficile à comprendre

Si on utilise un pointeur "identi_pt" pour accéder au champ "truc" d'une variable structurée la syntaxe est :

```
identi_pt -> truc
```

"indenti_pt" est l'identificateur d'une variable de type pointeur vers structure. Elle contient donc une valeur qui l'adresse de la variable structurée.

Nous pouvons donc a priori utiliser l'opérateur * et la syntaxe :

```
(* identi_pt).truc
```

est équivalente à :

```
identi_pt -> truc
```

Attention : la paire de parenthèses est nécessaire. En effet, l'écriture :

```
*identi_pt.truc
```

ne donne pas le résultat escompté, car l'opérateur '.' s'évalue de gauche à droite et il est plus prioritaire que l'opérateur '*' qui s'évalue lui de droite à gauche. Ce qui veut dire que la syntaxe

```
* identi_pt.truc
```

est en faite équivalente à

```
* (identi_pt.truc)
```

La syntaxe '->' a été introduite pour simplifier les écritures et éviter les erreurs.

Exemple

L'exemple suivant a été conçu pour provoquer des erreurs de compilation, il illustre

le problème :

```
#include <stdio.h>
struct machin
{
                        int truc;
};
int main(){
struct machin * identi_pt, variable;
identi_pt = &variable;
identi_pt -> truc = 7;
      // les deux ecritures suivantes sont evidemment
equivalentes
printf("\n variable.truc = %d",variable.truc);
printf("\n identi_pt->truc = %d",identi_pt->truc);
      // autre forme equivalente en utilisant l'operateur *
      // (* identi_pt).truc c'est la meme chose que
identi_pt->truc
      // les parentheses sont obligatoires
printf("\n (* identi_pt).truc = %d",(* identi_pt).truc);
      // la ligne suivante genere une erreur a la
compilation
      // car ce n'est pas equivalent a identi_pt->truc
printf("\n *identi_pt.truc = %d",*identi_pt.truc);
      // la ligne suivante genere aussi la meme erreur car
       // *identi_pt.truc est equivalent a
*(identi_pt.truc)
      // a cause de la priorite des operateurs
printf("\n *(identi_pt.truc) = %d",*(identi_pt.truc));
return 0;
}
```

7. Variables structurées et passage de paramètres

Méthode

Si l'on désire afficher les champs d'une variable de type structure « etudiant » à l'aide d'une fonction, il suffit de déclarer :

```
void afficher_etudiant( struct etudiant un_etu)
{
      printf("\n nom : %s ", un_etu.nom);
      printf("\n prenom : %s ",un_etu.prenom);
      printf("\n moyenne : %f ",un_etu.moyenne);
}
```

Et ensuite si on utilise :

```
afficher_etudiant(a); // la variable est supposée contenir
ici "paul" "sylve" et 19.0
```

Nous afficherons :

nom : sylve

prenom : paul

moyenne : 19.0

Comme nous l'avons vu dans le chapitre précédent sur les fonctions, la variable "a" est recopiée dans le paramètre formel "un_etu".

Par contre, si l'on veut modifier le contenu d'une variable structurée dans le corps

de la fonction, il faut transmettre un pointeur comme expliqué dans le chapitre 7.

```
void saisie_etudiant( struct etudiant * un_pt_sur_etu)
{
       printf("\n nom : ");
       scanf("%s ", un_pt_sur_etu->nom);
       printf("\n prenom : ");
       scanf("%s ",un_pt_sur_etu->prenom);
       printf("\n moyenne : ");
       scanf("%f ",&(un_pt_sur_etu->moyenne));
}
```

Syntaxe

Nous appellerons la fonction de la manière suivante :

```
saisie_etudiant (&e) ;
```

Ou encore :

```
saisie_etudiant (pt_e) ;
```

Complément

Une autre solution est possible pour la saisie :

```
// la fonction saisie_II retourne une valeur de type «
struct etudiant » et n'a pas de paramètre. struct etudiant
saisie_II (void)
{
      struct etudiant etu ; //déclaration de la variable
locale « etu »
      printf("\n nom : ");
      scanf("%s ", etu.nom);
      printf("\n prenom : ");
      scanf("%s ", etu.prenom);
      printf("\n moyenne : ");
      scanf("%f ",&( etu.moyenne));
      return etu ;
}
```

Dans ce cas, le contenu de la variable locale « etu » de la fonction « saisie_II » sera retournée par la fonction.

Puisque l'on peut affecter une variable structurée à une autre de même nature, l'usage en sera :

```
e= saisie_II () ;
```

La fonction retourne une valeur qui correspond à une variable structurée, et cette valeur est affectée à la variable structurée « e ».

8. Déclaration de type

Nous pouvons constater qu'à chaque fois que nous déclarons une variable structurée ou un pointeur vers une variable structurée, nous sommes obligés de préciser le mot réservé "struct" devant l'identificateur du nom du constructeur, ce qui peut être fastidieux. Heureusement, le langage C permet de construire des types définis par le programmeur.

Exemple : Exemple de départ

```
#define MAX 20
struct etudiant
{
       char nom[MAX], prenom[MAX];
       float moyenne;
} ;
```

Syntaxe

Nous pouvons déclarer ensuite :

```
typedef struct etudiant T_etudiant;
```

La syntaxe qui utilise le mot réservé "typedef" ci-dessus permet au programmeur de définir un type.

Ce type pourra ensuite être utilisé comme les autres types de variables.

Par la suite il suffira d'écrire :

```
void main()
{
       struct etudiant e = {"jean","seigne",18.8}, a,
*pt_e;
       T_etudiant etu_1, etu_2, * pt_etu;
       ...
}
```

"T_etudiant" est synonyme de "struct etudiant", les variables "etu_1", "etu_2" et "pt_etu" sont correctement définies. Elles s'utiliseront comme les variables "e", "a" et "pt_etu"

De même, cela simplifie l'écriture des fonctions, les fonctions deviennent alors:

```
void afficher_etudiant( T_etudiant un_etu)
{
      printf("\n nom : %s ", un_etu.nom);
      printf("\n prenom : %s ",un_etu.prenom);
      printf("\n moyenne : %f ",un_etu.moyenne);
}
void saisie_etudiant( T_etudiant * un_pt_sur_etu)
{
      printf("\n nom : ");
      scanf("%s ", un_pt_sur_etu->nom);
      printf("\n prenom : ");
     scanf("%s ",un_pt_sur_etu->prenom);
      printf("\n moyenne : ");
     scanf("%f ",&(un_pt_sur_etu->moyenne));
}
```

Syntaxe

De façon générale, "typedef" s'utilise ainsi :

```
typedef <type_constructeur> <nouveau_type_contructeur>
```

Exemple

Exemple de l'utilisation de la déclaration de type avec des tableaux

```
#define MAX 3
typedef float T_coord[MAX];
```

Peut être utilisé ainsi :

```
void main(void)
{
      T_coord point1, point2;
}
```

Les variables « point1 » et « point2 » sont ainsi déclarées comme des tableaux de 3 flottants (des coordonnées géométriques par exemple).

Il est aussi possible d'effectuer les déclarations "struct" et "typedef" simultanément, ainsi les écritures suivantes sont possibles :

```
typedef struct etudiant
{
      char nom[MAX], prenom[MAX];
       float moyenne;
} T_etudiant;
```

Ou ne faire que la déclaration du "typedef" ainsi :

```
typedef struct
{
      char nom[MAX], prenom[MAX];
      float moyenne;
} T_etudiant;
```

Dans ce dernier cas l'identificateur "etudiant" pour le constructeur de structure ne sera pas connu car il n'est pas déclaré.

Il sera donc impossible de déclarer les variables ainsi :

```
struct etudiant alpha;
```

Car l'identificateur "etudiant" sera inconnu.

Par contre dans les deux cas on peut écrire les déclarations :

```
T_etudiant alpha, beta;
```

Car l'identificateur de type « T_etudiant » est déclaré.

9. Structure et tableau dans une structure

Exemple

Considérons les déclarations :

```
#define MAX 20
typedef struct date {int jour, mois, annee;} T_date;
typedef struct etudiant {
      char nom[MAX], prenom[MAX];
      float moyenne;
      T_date date_naissance;
                  //date_naissance est un champ qui est une
variable structurée
} T_etu;
```

Nous avons ajouté des définitions de type ainsi qu'un champ de type "T_date" dans le constructeur de structure "etudiant" auquel nous avons donné comme nom de type "T_etu".

Si l'on déclare :

```
T_etu vous;
```

La variable "vous" contiendra les champs "nom", "prenom", "moyenne" et "date_naissance". Ce dernier étant un champ qui est une variable structurée.

La fonction de saisie deviendra alors :

```
void saisie_etudiant( T_etudiant * un_pt_sur_etu)
{
      printf("\n nom : ");
      scanf("%s", un_pt_sur_etu->nom);
      printf("\n prenom : ");
      scanf("%s",un_pt_sur_etu->prenom);
      printf("\n moyenne : ");
      scanf("%f",&(un_pt_sur_etu->moyenne));
      printf("\n jour de naissance : ");
      scanf("%d",&(un_pt_sur_etu->date_naissance.jour));
      printf("\n mois de naissance : ");
      scanf("%d",&(un_pt_sur_etu->date_naissance.mois));
      printf("\n annee de naissance : ");
      scanf("%d",&(un_pt_sur_etu->date_naissance.annee));
}
```

En effet, « un_pt_sur_etu -> date_naissance.jour» correspond à la variable « jour » qui est un champ, le « & » devant permet d'avoir son adresse, ce qui est nécessaire à la fonction « scanf ».

```
void affiche_etudiant( T_etudiant un_etu)
{
       printf("\n nom : %s ", un_etu.nom);
       printf("\n prenom : %s ",un_etu.prenom);
       printf("\n moyenne : %f ",un_etu.moyenne);
       printf("\n jour de naissance : %d
",un_etu.date_naissance.jour);
       printf("\n mois de naissance : %d
",un_etu.date_naissance.mois);
       printf("\n annee de naissance : %d
",un_etu.date_naissance.annee);
}
```

En effet, le champ "date_naissance" est vu comme une variable structurée. Nous pouvons donc utiliser comme nature de champ une autre structure.

Nous allons maintenant déclarer une structure comprenant un tableau de flottants :

```
#define MAX 500
typedef struct tableau_en_partie_rempli
{
       int nombre_cases_utilisees; float t[MAX];
} T_tableau_en_partie_rempli;
```

Et la déclaration de variable :

```
T_tableau_en_partie_rempli notes, *pt_sur_paquet_note;
```

Nous pouvons décider de remplir des notes, avec les instructions :

```
notes.t[0] = 15.7;
scanf'("%f",&(notes.t[1]));
notes.t[2] = (notes.t[0] + notes.t[1])/2;
notes.nombre_cases_utilisees = 3;
pt_sur_paquet_note = &notes;
```

```
pt_sur_paquet_note -> t[3] = 10.2;
scanf("%f",&(pt_sur_paquet_note->t[4]));
pt_sur_paquet_note->t[5] = (pt_sur_paquet_note->t[3] +
pt_sur_paquet_note->t[4])/2;
pt_sur_paquet_note->nombre_cases_utilisees = 6;
```

ainsi nous aurons 6 notes dans le tableau qui correspond au champ 't' de la variable 'notes', et le champ 'nombre_cases_utilisees' vaut 6.

Un champ qui est un tableau s'utilise comme un tableau, il suffit de faire attention à la façon dont on accède à ce champ (par un identificateur de variable structurée ou par un identificateur de pointeur vers variable structurée).

10. Tableaux de structures

Une variable structurée est composée d'éléments hétérogènes mais de taille fixe.

Considérons la déclaration :

```
# define MAX 20 struct etudiant
{
        char nom[20] ;
        char prenom[20] ;
        float moyenne ;
} ;
```

L'instruction :

```
sizeof(struct etudiant) ;
```

retournera la valeur 44 pour 44 octets (=20+20+4), à supposer qu'un float soit sur 4 octets.

De même, si le type « T_etudiant » est défini :

```
Sizeof (T_etudiant)
```

retourne aussi 44 pour 44 octets Il est donc possible de créer un tableau contenant des éléments du type d'une structure donnée.

Il suffit de créer un tableau dont le type est celui de la structure et de le repérer par un nom de variable.

Exemple

```
#define MAXETU 500
#define MAX 20
typedef struct date
{
        int jour,mois, annee;
} T_date;
typedef struct etudiant
{
        char nom[MAX], prenom[MAX];
         float moyenne;
         T_date date_naissance;
} T_etu;
typedef T_etu T_tab_etu[MAXETU ];
```

Lors de la déclaration de variable :

```
T_tab_etu module;
```

On a défini le tableau "module", où chaque case représente alors une structure du

type T_etu. En considérant les dernières fonctions écrites, nous pourrons écrire les instructions :

```
saisie_etudiant(&(module[i]));
affiche_etudiant(module[i]);
```

Si l'on a déclaré "i" comme un entier et initialisé ce dernier avec un indice licite.

Remarque

vous avez remarqué que les identificateurs de type que nous avons défini commencent toujours par le suffixe "T_", ce n'est pas obligatoire. Un identificateur valide suffit.

Cependant, cette convention facilite la lecture du code. Nous avons utilisé comme convention : le suffixe "T_" est utilisé pour débuter l'identificateur d'un type déclaré par le programmeur.

A vous d'utiliser, les conventions qui semblent les plus claires.

11. Exercices de Révision (QCM)

Exercice 1

1. Soient les déclarations suivantes :

struct LIVRE {

int nb_pages ;

char langue[20], auteur[20], titre[20] ;

float prix; } L ;

Comment accéder à un champ d'un enregistrement ?

○	auteur.L
○	L.auteur
○	L : auteur
○	auteur.livre
○	livre.auteur

Exercice 2

2. Quelle est la différence entre un tableau et un enregistrement ?

○	Un tableau peut contenir des données de types différents, tandis qu'un enregistrement ne le peut pas.
○	Un enregistrement peut contenir des données de types différents, tandis qu'un tableau ne le peut pas.
○	Tous deux peuvent contenir des données de types différents, mais l'enregistrement occupe moins de place mémoire.
○	Tous deux peuvent contenir des données de types différents, mais l'enregistrement permet un accès mémoire plus rapide.

Exercice 3

3. On considère la déclaration suivante :

#define MAX maxint
struct TIMBRE{
int prix ;
char origine[20] ;
int annee ;
char image[20] ;
char couleur[20];
} ;
struct TIMBRE COLLECTION [MAX];
// Une collection est un tableau de timbres
Comment accède-t-on à l'année du 3ème timbre de la collection?

○	COLLECTION [2, 2]
○	COLLECTION [2]. annee
○	COLLECTION [2, annee]
○	COLLECTION. annee [2]
○	COLLECTION. annee

Exercice 4

4. Quelle est la déclaration correcte d'un enregistrement ?

○	struct LIVRE (int nb_pages ; char langue[20], auteur[20], titre[20] ; float prix;)
○	struct LIVRE { int nb_pages ; char langue[20], auteur[20], titre[20] ; float prix; }
○	struct LIVRE { int nb_pages ; char langue[20], auteur[20], titre[20] ; float prix; } ;

Exercice 5

5. Quelles sont les différentes affectations que l'on peut réaliser sur ces variables structurées ?
Sachant que :
struct VOITURE { char Marque[20]; float Cylindree ; char Couleur[20] ; char Nom[20] ; int Prix ; } ;
struct VELO { char Marque[20]; char Couleur[20] ; char Nom[20] ; int Prix ; };
struct VOITURE V1 ;
struct VELO V2 ;

☐	V1 =V2 ;
☐	V2 = V1 ;
☐	V2.Marque = V2.Nom ;
☐	V1.Cylindree := 1.5 ;
☐	Aucune de celles proposées

I. Allocation dynamique

1. Allocation dynamique

Rappel : Un petit rappel avant de commencer

La mémoire centrale d'un ordinateur peut être vue comme un long ruban d'octets. Chaque octet possède une adresse unique. Dans les chapitres précédents, les variables ont été présentées comme un couple (identificateur, réceptacle), le réceptacle étant constitué de plusieurs octets contigus en mémoire centrale.

Nous avons vu que, connaissant la taille (le nombre d'octets) d'un réceptacle et l'adresse de son premier octet, il était possible de lire et d'écrire de l'information comme si c'était une variable : c'est une des utilisations possibles des pointeurs. Cependant, nous n'avons pas précisé où se trouvaient rangées les variables en mémoire centrale. Nous dirons simplement qu'il existe deux zones de la mémoire où on peut ranger les variables :

- Une zone 1 qui correspond à ce que nous avons vu jusqu'à présent
- Une zone 2 qui peut être utilisée dynamiquement par le programmeur

La première zone correspond par exemple à la déclaration ci-dessous :

```
void main(void)
{
      int i;
      float t[50];
}
```

Nous avons utilisée ce type de déclaration dans tous nos exemples, depuis le début de ce cours introductif. Nous allons vous présenter succinctement comment utiliser la zone 2 que l'on appelle le "tas" (heap en anglais).

Attention

Nous avons volontairement simplifié la présentation en scindant la mémoire en deux zones, la réalité est plus complexe. Les explications fournies dans ce cours introductif sont suffisantes pour l'appréhender. Néanmoins, la « carte » d'un programme qui s'exécute en mémoire centrale est plus complexe.

Ce cours introductif n'a pas pour objectif de prendre en considération des concepts trop complexes. A ce stade, ce qu'il faut retenir, c'est que l'on peut dynamiquement demander la création de nouveaux réceptacles pour des données. Ceci signifie que l'on peut allouer de l'espace pour les données au cours de l'exécution du programme.

Cette création s'effectuera dans la zone de tas. De même, on peut récupérer dynamiquement les octets qui ont été alloués. Pour cela, nous allons utiliser deux fonctions :

- malloc
- free

Ces fonctions sont définies dans le fichier "en-tête" « stdlib.h » qui fait partie de la bibliothèque standard définie par la norme ANSI. Ce fichier contient aussi les déclarations de fonctions traitant de conversion de chaînes de caractères en types numériques ("itoa" par exemple), de tirages aléatoires (« srand » et « rand ») et d'autres fonctions d'allocation mémoire.

2. La fonction « malloc »

Rappel

Nous vous rappelons qu'il est impossible de déclarer une variable de type « void » et qu'une fonction de type « void » ne retourne aucune valeur.

Par contre, « void * » correspond à une donnée de type pointeur vers octet. Nous pouvons donc déclarer une variable de type « void * ».

Si une fonction est de type « void * » elle retournera une valeur de type « adresse d'un octet » (voir chapitre 3).

Syntaxe

L'en-tête de la fonction « malloc » est le suivant : void *malloc(size_t taille) ;

Définition

La fonction malloc sert à faire une allocation dynamique de mémoire dans la zone du « tas ». Le paramètre « taille » est de type « size_t » (type entier), c'est la taille de l'espace-mémoire que l'on veut obtenir en nombre d'octets.

Elle retourne une adresse de type pointeur vers « void » (pointeur vers octet), qui nécessite l'utilisation de la conversion explicite de type (cast voir chapitre 4).

Exemple

Prenons l'exemple suivant :

```
void main()
{
char * s;
       // une variable de type pointeur sur caractères
int * t;
      // une variable de type pointeur sur entier
float * td;
       // une variable de type pointeur sur flottant
int i;
s = (char *) malloc(10 * sizeof(char));
t = (int *) malloc(12 * sizeof(int));
td = (float *) malloc(51 * sizeof(float));
scanf("%s",s);
for (i = 0;i<10;i++)
      scanf("%d",&(t[i]));
for (i = 0;i<10;i++)
      scanf("%f",&(td[i]));
}
```

Définition

L'utilisation de la fonction « sizeof » permet de retourner la taille du réceptacle (en octets) associé à un type (voir chapitre 3). Si on multiplie ensuite par une valeur positive entière, nous avons une valeur entière qui correspond à un nombre

d'octets. La fonction « malloc » demande alors au gestionnaire (fictif) de la mémoire centrale de lui réserver le nombre d'octets consécutifs demandés en mémoire.

Si cette allocation est impossible par manque de place, « malloc » retourne la valeur de pointeur « NULL » (Voir chapitre 4), sinon elle retourne l'adresse du premier octet de la zone allouée dans le tas. Toutes les valeurs retournées par la fonction « malloc » sont de type « void * ».

Pour que le compilateur puisse considérer autre chose qu'une zone d'octets, il faut effectuer une conversion explicite de type (un cast). La conversion est réalisée ici grâce aux (char *), (int *), (float *) qui transforment la valeur retournée par « malloc » en « une adresse vers » respectivement : un char, un int et un float.

Après les trois premières instructions qui utilisent "malloc", "s" est une variable qui contient l'adresse d'une zone de caractères, "t" est une variable qui contient l'adresse d'une zone d'entiers et enfin "td" est une variable qui contient l'adresse d'une zone de float.

En conséquence, nous pouvons accéder aux éléments comme avec des tableaux.

Remarque

En effet, nous vous rappelons que :

- s[2] est syntaxiquement la même chose que *(s+2)
- t[4] est syntaxiquement la même chose que *(t+4)
- td[11] est syntaxiquement la même chose que *(td+11)

Sauf que cette fois-ci "s", "t" et "td" sont des variables de type "pointeur vers".

Nous pouvons donc les manipuler comme un tableau, la seule différence étant que "s", "t", et "td" sont des variables et non des constantes (voir chapitre 6). Nous les utilisons ensuite comme des tableaux dans les instructions qui suivent.

L'allocation dynamique est très utile en programmation pour créer des réceptacles temporaires pour des données.

Exemple : Prenons l'exemple

Cet exemple permet de réserver en mémoire la place pour un tableau d'élèves dont on demande la taille à l'utilisateur.

```
#define MAX 50
struct eleve
{
      char nom[MAX],
      prenom[MAX],
      float note_moyenne;
};
typedef struct eleve T_eleve;
int * allouer_tableau_eleve(int nb)
{
      T_eleve * inter;
// variable locale de la fonction allouer_tableau_eleve
// qui contient une valeur qui est l'adresse d'un tableau
alloué dynamiquement
      inter = (T_eleve *)malloc(nb*sizeof(T_eleve));
//allocation de « nb » cases, chaque case étant de
// type « T_eleve ».
// La fonction « malloc » retourne un « void* », il ne faut
// donc pas oublier l'opération de « cast ».
      return inter;
// on retourne la valeur du pointeur : elle n'est pas
perdue
```

```
// bien que la variable locale soit détruite après l'appel
de la fonction.
}
int main(void)
{
       T_eleve * tab_eleve;
      int nombre;
       prinf("\n entrer le nombre d'eleves : ");
      scanf("%d",&nombre);
       tab_eleve = allouer_tableau_eleve(nombre);
// appel de la fonction « allouer_tableau_eleve » qui va
//dynamiquement créer
// un tableau de « nombre » cases. Chaque case étant
//une variable de type « T_eleve »
      if (tab_eleve == NULL)
             prinf("\n probleme d'allocation dynamique");
      else
            {
                  // la suite de votre code
            }
}
```

Attention

Ce n'est pas un tableau de taille variable, vous avez alloué un nombre fixe de cases.

Si vous ne souhaitez pas utiliser l'ensemble des cases, il vaudrait mieux utiliser la technique algorithmique vue au chapitre 6.

3. La fonction "free"

Définition

La fonction "free" sert à restituer l'espace que l'on avait alloué avec "malloc".

Conseil

Nous vous conseillons d'utiliser cette fonction autant de fois que la fonction "malloc" pour libérer la place allouée dynamiquement. Si "p" pointe sur le début d'une zone mémoire allouée dynamiquement alors l'appel de la fonction « free(p) » libère cette zone mémoire.

Le gestionnaire (fictif) de mémoire centrale récupère cette place et peut éventuellement l'allouer à nouveau.

Pour comprendre le fonctionnement de la fonction « free », il peut être utile de savoir que, au moment d'une allocation dynamique, le gestionnaire (fictif) de mémoire réserve en fait une petite place supplémentaire juste avant la zone allouée au programme pour y noter la longueur de la zone allouée. Lorsque l'instruction « free(p) » est effectuée, il suffit alors au gestionnaire d'aller relire la longueur de la zone allouée pour connaître la taille de la zone à "désallouer".

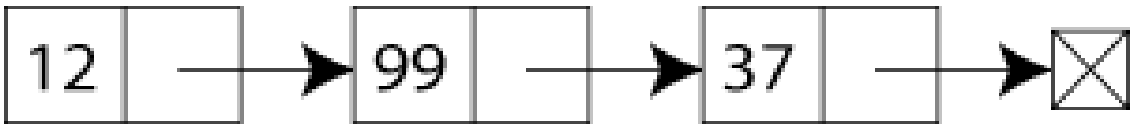
Si la mémoire allouée avec « malloc » n'est pas libérée avec « free », elle est quand même libérée à la fin du programme, mais ce n'est pas de la programmation « propre ». Si l'on cherche à libérer une zone mémoire non allouée avec « malloc », cela peut provoquer une erreur à l'exécution du code.

4. Un exemple de gestion de listes chaînées

Le code complet qui suit résume ce cours. Une liste chaînée est constituée de

cellules. Une cellule est une donnée structurée qui comporte une information que l'on stocke et une information sur l'adresse de la cellule qui suit.

Nous avons, dans l'exemple qui suit, une structure avec deux champs : un entier et un pointeur vers une structure de type cellule qui suit la cellule courante.



Graphique 8 Liste chainée

Nous utilisons les techniques d'allocation dynamique pour créer dynamiquement les zones mémoires (réceptacles) qui correspondent aux cellules.

La liste est repérée simplement par un pointeur vers la tête de la liste. La tête de liste correspond à l'adresse de la première cellule de la liste. Quand la liste est vide, ce pointeur vaut NULL. Sur le dessin qui précède, la cellule de tête est celle qui contient la valeur 12.

Et la dernière cellule est celle qui contient la valeur 37. Le pointeur qui suit la dernière cellule pointe sur une X, ce qui symbolise graphiquement la valeur particulière NULL. NULL est une valeur particulière qui correspond à une adresse mémoire qui n'existe pas.

Il est donc facile de tester si la liste est vide. Nous comparons la valeur du pointeur de la liste avec cette valeur particulière (voir chapitre 4, opérateurs & et *).

Dans le code ci-dessous nous utilisons le prototypage de fonction.

Ce code permet :

- d'ajouter des cellules en tête de la liste,
- d'afficher les cellules de la liste,
- de rechercher une valeur particulière dans la liste et
- de récupérer tous les réceptacles alloués dynamiquement quand on désire détruire toute la liste.

C'est une base, qui peut vous servir pour vos propres applications.

Exemple

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
struct cellule {
   int valeur;
   struct cellule * succ;
         // pointeur vers la cellule suivante
};
typedef struct cellule T_cellule;
   // definition de type, permet d'eviter
   // d'avoir a ecrire a chaque fois "struct cellule"
   // les prototypes des fonctions
char bon_choix(char *s);
int affiche_menu_et_saisie_choix(void);
void afficher_cellule(T_cellule * c);
void affiche_liste(T_cellule * c);
T_cellule * detruire_liste(T_cellule * c);
T_cellule * ajouter(T_cellule * c);
void rechercher(T_cellule * c);
// la fonction principale
int main(void) {
```

```
   char choix;
   T_cellule * tete;
   tete = NULL;
   do {
         switch (choix = affiche_menu_et_saisie_choix())
         {
               case 'A' : tete = ajouter(tete); break;
               case 'V' : affiche_liste(tete); break;
               case 'S' : tete = detruire_liste(tete);
break;
               case 'R' : rechercher(tete); break;
               case 'Q' : { tete = detruire_liste(tete);
                                    // desallouer toujours
proprement avant de sortir
                                    printf("\n\nAu
revoir\n");
                                  }
          }
      } while (choix !='Q');
   printf("\n");
}
// les declarations completes des fonctions
char bon_choix(char *s) {
/* cette fonction teste si le premier caractère d'une
chaîne correspond à un caractère parmi 'A','a', 'V', 'v',
'S', 's', 'R', 'r', 'Q', 'q', elle retourne un unique
caractère majuscule dans ce cas et le caractère '\0' sinon.
*/
char r;
switch (s[0])
{
   case 'A' :;
   case 'a' : r = 'A';break;
   case 'V' :;
   case 'v' :r = 'V';break;
   case 'S' :;
   case 's' :r = 'S';break;
   case 'R' :;
   case 'r' :r = 'R';break;
   case 'Q' :;
   case 'q' : r = 'Q';break;
   default: r='\0';
}
return r;
}
int affiche_menu_et_saisie_choix(void)
{
/* Permet d'afficher un menu, de lire une chaîne et de
vérifier si le premier caractère de cette chaîne correspond
à un des choix possibles du menu.*/
// Elle affiche un message d'erreur si le choix n'est pas
bon et ré-affiche ensuite le menu.
char s_choix[2];
do {
   printf("\n ajouter -> A");
   printf("\n voir liste -> V");
   printf("\n supprimer liste -> S");
   printf("\n rechercher -> R");
   printf("\n quitter -> Q");
```

```
   printf("\n");
   printf("\n votre choix : ");
   scanf("%s",s_choix);
   printf("\n");
   if (!bon_choix(s_choix))
      {
         printf("\n\nVotre choix n'est pas bon ,recommencez
SVP\n");
      }
} while (!bon_choix(s_choix));
return (bon_choix(s_choix));
}
void afficher_cellule(T_cellule * c) {
// Fonction qui affiche le contenu d'une cellule
if (c==NULL)
{
   printf("\rien a afficher, cellule vide\n");
}
else
{
   printf("\nla cellule rangee a l'adresse %p contient la
valeur %d ",c,c->valeur);
   printf("\nla cellule qui suit est rangee a l'adresse %p
",c->succ);
   printf("\n");
}
}
void affiche_liste(T_cellule * c) {
// Fonction qui affiche le contenu d'une liste cellule par
cellule
if (c==NULL)
{
   printf("\nrien a afficher, liste vide\n");
}
else {
   do {
         afficher_cellule(c);
         c = c -> succ;
        } while (c!=NULL);
  }
}
T_cellule * detruire_liste(T_cellule * c) {
// Fonction qui libère toute la place mémoire occupée par
une liste que l'on veut détruire
//initialement c contient la valeur du pointeur vers la
cellule de tête de liste à détruire
T_cellule * inter;
if (c==NULL)
{
   printf("\nrien a detruire, liste vide\n");
}
else
{
   do {
         inter = c -> succ; //inter pointe sur la prochaine
cellule a détruire
         printf("\nrecuperation de la memoire de la cellule
a l'adresse %p",c);
         free(c);
```

```
         c = inter;
        } while (c!=NULL);
}
return NULL; //puisque toute la liste a été détruite on
renvoie NULL
}
T_cellule * ajouter(T_cellule * c) {
// Fonction qui crée une cellule (allocation dynamique avec
« malloc »).
T_cellule * nouvelle;
nouvelle = ( T_cellule *)malloc(sizeof( T_cellule));
printf("\nun entier SVP ");
scanf("%d",&(nouvelle->valeur));
nouvelle -> succ = c; return nouvelle;
}
void rechercher(T_cellule * c) {
// Fonction qui recherche une valeur particulière dans une
liste.
// Affiche l'adresse de la cellule qui contient la valeur
si cette dernière est dans la liste et un message
// disant que la valeur n'est pas dans la liste sinon.
int val,trouve;
trouve = 0;
if (c==NULL)
{
   printf("\nrien a rechercher, liste vide\n");
}
else {
   printf("\nl'entier recherche SVP ");
   scanf("%d",&val);
   do {
         if (c->valeur == val)
        {
          printf("\n\nl'entier recherche %d",val);
           printf("\nse trouve dans la cellule a l'adresse
%p",c);
           c = NULL;
// pour stopper la boucle do while
          trouve = 1;
// pour memorise le fait que l'on a trouve
      }
        else
      {
         printf("\npanparcours cellule a l'adresse %p,
valeur %d",c,c->valeur);
         c = c-> succ;
      }
} while ((c!=NULL)&& (trouve !=1));
if (trouve ==0)
{
   printf("\n\nrecherche entier %d ",val);
   printf("ne se trouve pas dans la liste");
   printf("\n");
}
}
}
```