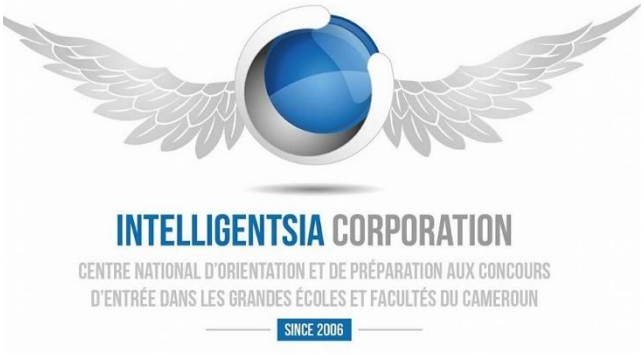


CENTRE NATIONAL D'ORIENTATION ET DE PREPARATION AUX CONCOURS D'ENTREE DANS LES GRANDES ECOLES ET FACULTES DU CAMEROUN

Préparation au Concours d'Entrée en Troisième Année de l'ENSP et FGI

Support de Cours

PROGRAMMATION EN LANGAGE C

Avec Intelligentsia Corporation, Il suffit d'y croire !!!

☎ 698 222 277 / 671 839 797
fb : Intelligentsia Corporation
email : contact@intelligentsia-corporation.com

" Vous n'êtes pas un passager sur le train de la vie, vous êtes l'ingénieur. "

-- Elly Roselle --

Instructions :

Il est recommandé à chaque étudiant de traiter les exercices de ce recueil (du moins ceux concernés par la séance) avant chaque séance car ***le temps ne joue pas en notre faveur.***

I. INTRODUCTION & OBJECTIFS

Les programmes, il y'en a bon nombre et tout genre dans un ordinateur. De la simple calculatrice, en passant par les éditeurs de textes, les jeux vidéo, le système d'exploitation etc. tous sont à la base des programmes informatiques.

Un **programme** exécutable est une suite d'instructions pouvant être exécutées par le processeur. Ces instructions sont très difficiles à comprendre, il n'est donc pas envisageable de créer des programmes en écrivant des suites de chiffres et de lettres. Pour pouvoir donc créer des programmes, on se sert des langages de programmation.

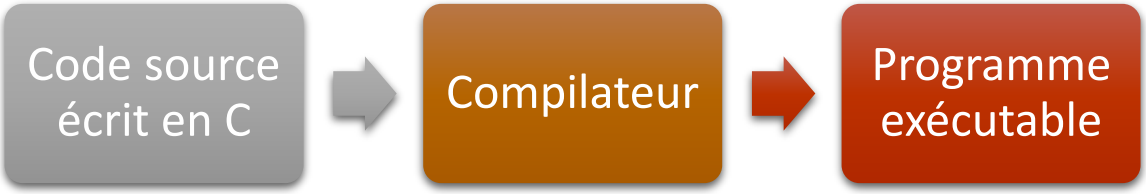
Un **programme en C** est un ensemble d'instructions qui se saisit dans un fichier **.c** à l'aide d'un éditeur (ex : Notepad), ce type de fichier s'appelle source. Les instructions qui y sont écrites s'appellent du code ou encore le code source.

Un **compilateur** est un logiciel qui lit le code source et le convertit en un code exécutable, c'est-à- dire un ensemble d' instructions compréhensible par le processeur.

La compilation d'une source se fait en deux étapes : la compilation proprement dite et le linkage; on utilise en général le terme compilation en englobant les deux étapes précédemment citées.

La compilation à proprement parler produit un fichier objet, dont l'extension est **.obj**. Les sources peuvent être réparties dans plusieurs fichiers, ceux- ci doivent tous être compilés séparément. Ensuite, le linkage des différents fichiers objets est assemblage produisant un exécutable **.exe**.

Certains environnements de développement (IDE) servent d'éditeur, et prennent en charge la compilation et le linkage (CodeBlocks pour C/C++, eclipse pour Java etc.



II. LES BASES DU LANGAGE C

1. Structure d'un programme

```
//Inclusion des bibliothèques
#include <stdio.h>


int main(int argc, char *argv[])
{
      //Corps du programme
      printf("Hello World !!!\n");
      return 0;
}
```

Selon ce que l'on souhaite faire dans notre programme, on peut avoir besoin de différentes fonctions. Celles - ci sont disponibles dans des bibliothèques.

Par exemple : « **stdio.h** » propose des fonctions de lecture et d'écriture.

Ensuite on a la fonction **main**() : C'est le point d'entrée de notre programme, c'est la fonction qui est appelée lorsqu'on exécute un programme.

Un programme en langage C est constitué de fonctions, il ne contient quasiment que ça. Pour le moment, notre programme ne contient donc qu'une seule fonction.

Une **fonction** permet grosso modo de rassembler plusieurs commandes à l'ordinateur. Regroupées dans une fonction, les commandes permettent de faire quelque chose de précis. Par exemple, on peut créer une fonction "ouvrir_fichier" qui contiendra une suite d'instructions pour l'ordinateur lui expliquant comment ouvrir un fichier. L'avantage, c'est qu'une fois la fonction écrite, on n'aura plus qu'à dire "ouvrir_fichier", et l'ordinateur saura comment faire sans qu'on ait à tout répéter.

Une fonction a un début et une fin, délimités par des accolades { et }. Toute la fonction main se trouve donc entre ces accolades.

Les lignes à l'intérieur d'une fonction sont appelées **instructions**, elles se terminent toujours par un point-virgule.

La première ligne : printf("Hello world!\n"); demande à afficher le message "Hello world!" à l'écran, tandis que la seconde : return 0; marque la fin de la fonction main() et renvoie le code

0 ; En fait chaque programme une fois terminé renvoie une valeur (0 = tout s'est bien passé et toute autre valeur = erreur). La plupart du temps cette valeur n'est pas utilisée mais il faut quand même en renvoyer une.

Il est possible d'insérer des commentaires dans le code en utilisant un double slash (//) pour un commentaire mono ligne et /* */ pour un commentaire multi ligne.

2. Les variables

Une **variable** est un emplacement de la mémoire dans lequel est stockée une valeur. Chaque variable porte un nom et c'est ce nom qui sert à identifier l'emplacement de la mémoire représentée par cette variable.

Pour utiliser une variable, la première étape est la déclaration.

a) Déclaration des variables :

En C, on déclare les variables à n'importe quel endroit du programme, mais surtout avant son utilisation dans une instruction.

On place les instructions à exécuter à la suite de la déclaration de variables.

Pour déclarer une variable, la syntaxe est la suivante :

```
type nom_de_la_variable [= valeur_initiale];
```

NB : L'assignation d'une valeur initiale (entre crochet) est facultative.

Exemples :

```
int i;
float x, y = 2.5;
char c = 'a';
```

Les principaux types de variables en C sont :

Type	Plage de valeurs	Description
char	-128 à 127	Caractère
int	-2 147 483 648 à 2 147 483 647	Entier signé
long	-2 147 483 648 à 2 147 483 647	Entier signé
float	-3.4 x 10 puissance 38 à 3.4 x 10 puissance 38	Nombre décimal à virgule flottante

double	-1.7 x 10 puissance 308 à 1.7 x 10 puissance 308	Nombre décimal à virgule flottante codé avec double précision

Pour les types stockant des entiers (**char**, **int**, **long**...), il existe d'autres types dits "**unsigned**" (non signés) qui eux ne peuvent stocker que des nombres positifs. Pour les utiliser, il suffit d'écrire le mot "unsigned" devant le type :

Type	Plage de valeurs	Description
unsigned char	0 à 255	Octet
unsigned int	0 à 4 294 967 295	Entier non signé
unsigned long	0 à 4 294 967 295	Entier non signé

b) Affectation :

L'**affectation** est une instruction qui a pour but d'assigner une valeur à une variable. Elle se fait via l'opérateur « = ».

Exemple :

```
i = 2;
x = 1.05;
y = (2*x)/5;
```

On dispose ainsi des opérateurs arithmétiques « +, -, *, /, % » qui représentent respectivement l'addition, la soustraction, le produit, la division et le modulo (reste de la division euclidienne).

c) Les constantes :

Il arrive parfois que l'on ait besoin d'utiliser une variable dont on voudrait qu'elle garde la même valeur pendant toute la durée du programme. C'est-à-dire qu'une fois déclarée, vous voudriez que votre variable conserve sa valeur et que personne n'ait le droit de changer ce qu'elle contient.

Ces variables particulières sont appelées **constantes**, justement parce que leur valeur reste constante.

Pour déclarer une constante, c'est en fait très simple : il faut utiliser le mot "**const**" juste devant le type quand on déclare la variable.

Par ailleurs, il faut obligatoirement lui donner une valeur au moment de sa déclaration.

Ex :

```
const float PI = 3.14159;
```

d) Afficher le contenu d'une variable :

La fonction **printf()** est utilisée pour afficher du texte à l'écran, mais elle peut aussi être utilisée pour afficher le contenu d'une variable. Il suffit d'ajouter un symbole spécial à l'endroit où on souhaite afficher le contenu d'une variable, puis spécifier la variable en question.

Ex :

```
int a = 5;
printf("La variable a vaut %d", a);
```

Il est possible d'afficher le contenu de plusieurs variables en même temps, il suffit d'indiquer tous les endroits où on souhaite afficher une variable et toutes les variables en question.

Ex :

```
int a = 5, b = -1;
printf("a vaut %d et b vaut %d", a, b);
```

Ci-dessous les symboles à utiliser pour différents types de variables :

Symbole	Type de variable
%c	**char**
%d	**int** et **long**
%f	**float** et **double**

e) Lire les entrée du clavier :

Il est important de pouvoir lire les entrées du clavier car ceci de rendre son programme interactif. On peut récupérer une valeur à l'utilisateur et la stocker dans une variable.

Pour ce faire, on utilise la fonction **scanf()** qui ressemble beaucoup à printf(), on met un « **%d** » pour indiquer que l'utilisateur doit saisir un nombre.

Ex :

```
int age ;
printf("Quel est votre âge ? ");
scanf("%d", &age);
```

Par ailleurs, on note le symbole « **&** » devant le nom de la variable qui va recevoir la valeur venant du clavier. Celui-ci désigne en fait l'adresse de la variable « age », la notion d'adresse sera mieux expliquée dans la partie consacrée aux pointeurs.

Il est aussi à noter que pour lire un « **double** » au clavier on utilise « **%lf** » au lieu de « **%f** » comme avec la fonction printf.

3. Les opérateurs

a) Généralités : Opérandes et arité :

Si l'opérateur s'applique à 2 opérandes, on dit qu'il s'agit d'un **opérateur binaire**, ou bien d'**arité 2**.

Un opérateur d'**arité 1**, dit aussi **unaire**, s'applique à un seul opérande, (par exemple –x qui est équivalent à -1 * x).

- Les opérateurs unaires (les opérateurs « **-** » et « **~** »)
- Les opérateurs binaires (« **>>** », « **<<** », « **&** », « **|** » et « **^** »)

b) Priorité :

En fait tous les opérateurs n'ont pas la même priorité, lorsque le compilateur est en face d'une expression non parenthesée il choisit les opérations à effectuer en fonction de leurs priorités.

Ceux de priorité la plus forte sont les opérateurs arithmétiques (*, /, %, +, -), puis les opérateurs de décalage de bit (<<, >>), les opérateurs de bit (&, ^, |), et enfin l'affectation =.

c) Forme contractée :

Il est possible de simplifier certaines expressions en utilisant les formes contractées.

Exemple :

```
a += 2; //a = a + 2;

b = ++a + (c = 3); //a = a + 1; c = 3; b = a + c;
```

4. Les traitements conditionnels

On appelle **traitement conditionnel**, un traitement qui est effectué selon ou non qu'une condition est remplie.

En C, les traitements conditionnels se font de la manière suivante :

```
if( <condition> ){
      //Traitement si la condition est remplie
}else{
      //Traitement sinon
}
```

Il est à noter que le bloc « **else** { … } » est facultatif, et qu'il y'a la possibilité d'avoir des « sinon si » qui se traduisent par « **else if** ([alternative1]) { //Traitement alternatif 1 } » autant de fois qu'on le souhaite.

Exemple :

```
if(age < 2){
      printf("Bébé");
}else if(age < 14){
      printf("Enfant");
}else if(age < 18){
      printf("Adolescent");
}else{
      printf("Adulte");
}
```

- **Les conditions ternaire :**

Une **condition ternaire** est une affectation conditionnée. Il s'agit ici e fait d'assigner telle ou telle valeur en fonction d'une condition.

Syntaxe :

```
<variable> = <condition> ? <valeur_si_vrai> : <valeur_si_faux>;
/* Ce qui est équivalent à :
if( <condition> )
      <variable> = <valeur_si_vrai>;
else
      <variable> = <valeur_si_faux>;
*/
```

Exples :

```
abs = x >= 0 ? x : -x; //Valeur absolue
max = a > b ? a : b; //Max
```

- **L'énumération des cas :**

Ce type de traitement conditionnel est utilisé pour simplifier l'écriture (rendre mieux lisible) de certains traitements conditionnels avec beaucoup de « else if ».

Syntaxe :

```
switch( <variable> )
{
      case <valeur_1> :
            //Traitement 1
            [break;]
      /* Autant de cas que nécessaire */
      case <valeur_n> :
            //Traitement n
            [break;]
      default :
            //Traitement par défaut
}
```

Le « break » est facultatif, il permet d'arrêter de tester lorsqu'un cas a été repéré.

Le « default » permet de renseigner le traitement qui est appliqué lorsqu'on n'a repéré aucun cas.

Exemple :

Supposons qu'on veuille écrire en toutes lettres un chiffre (compris entre 0 et 3).

On a à gauche l'écriture avec des « if…else » et à droite avec un « switch ».

```
switch( nombre )
{
      case 0 :
            printf(“Zero”); break;
      case 1 :
            printf(“Un”); break;
      case 2 :
            printf(“Deux”); break;
      case 3 :
            printf(“Trois”);   break;
      default :
            printf(“Inconnu”);
}
```

```
if( nombre == 0){
      printf(“Zero”);
}else if( nombre == 1){
      printf(“Un”);
} else if( nombre == 2){
      printf(“Deux”);
}else if( nombre == 3){
      printf(“Trois”);
} else{
      printf(“Inconnu”);
}
```

- Les booléens :

Conceptuellement, une condition est une valeur booléenne. Mais en C il n’existe pas de type booléens, les conditions sont en fait des entiers où **0** représente « **faux** » et toute autre valeur entière représente « **vrai** ». Habituellement, on prend **1** comme vrai.

Les opérateurs logiques (**!**, **&&**, **||** et **^**) s’appliquent donc aux entiers (en les assimilant à des booléens).

5. Les boucles

Une **boucle** permet d'exécuter plusieurs fois de suite une même séquence d'instructions. Chaque exécution du corps d'une boucle s'appelle une itération, ou plus informellement un passage dans la boucle. Lorsque l'on s'apprête à exécuter la première itération, on dit que l'on rentre dans la boucle, lorsque la dernière itération est terminée, on dit qu'on sort de la boucle.

Il existe trois types de boucle :

- **while**
- **do … while**
- **for**

Chacune de ces boucles a ses avantages et ses inconvénients. Nous les passerons en revue ultérieurement.

a) La boucle « while » :

Elle correspond à « tantque…faire » en algorithmique, sa syntaxe est la suivante :

Syntaxe :

```
while( <condition> ){
      //Traitement à répéter
}
```

Exemple : Calcul de la somme des n premiers entiers :

```
int somme = 0; i = 0;
while( i <= n ){
      somme += i;
}
```

b) La boucle « do…while » :

Elle correspond à « faire…tantque » en algorithmique, sa syntaxe est la suivante :

Syntaxe :

```
do{
      //Traitement à répéter
}while( <condition> );
```

Exemple : Calcul de la somme des n premiers entiers :

```
int somme = 0; i = 0;
do{
      somme += i;
} while( i <= n );
```

c) La boucle « for » :

Elle correspond à « pour » en algorithmique, sa syntaxe est la suivante :

```
for([initialisation]; [condition de continuation]; [incrémentation]){
      //Traitement à répéter
}
```

Exemple : Calcul de la somme des n premiers entiers :

```
int somme = 0; i = 0;
for(i=0; i<=n; ++i){
      somme += i;
}
```

6. Les fonctions

Comme vu précédemment, un programme en C commence par une fonction appelée "main".

Seulement, jusqu'ici nous sommes restés à l'intérieur de la fonction main(). Nous n'en sommes jamais sortis.

Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction "main". Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes, mais en imaginant des plus gros programmes qui font des milliers de lignes de code, Si tout était concentré dans la fonction main, le code serait peu maintenable et très touffus.

Une fonction exécute des actions et renvoie un résultat. C'est un morceau de code qui sert à faire quelque chose de précis.

On dit qu'une fonction possède une entrée et une sortie. Schématiquement, cela revient à:

Le but des fonctions est donc de simplifier le code source, pour ne pas avoir à retaper le même code plusieurs fois d'affilée.

La syntaxe de création d'une fonction est la suivante :

```
<type_de_retour> <nom_fonction> ( [params] ){
     //Corps de la fonctions
}
```

- **<type_de_retour>** est le type de la sortie
- « **params** » représente l'entrée

Exple :

```
int addition(int a, int b){
     return a + b;
}
```

Les procédures :

Une procédure est une fonction qui ne renvoie pas de valeur. En C cela se traduit par une fonction dont le type de retour est « **void** ».

Exple :

```
void dire_bonjour(){
     printf("Bonjour !");
}
```

Appel de fonction :

L'appel d'une fonction se fait par son nom suivi des parenthèses et éventuellement le passage de valeurs.

Exple :

```
int x = addition(18, 63);
dire_bonjour();
```

Prototype d'une fonction :

On appelle **prototype** (ou **signature**) d'une fonction, l'entête qui comporte le type de retour de la fonction, son nom et ses différents paramètres.

Ainsi pour nos fonctions précédentes, on a les prototypes suivants :

```
int addition(int a, int b);
void dire_bonjour();
```

III. TECHNIQUES AVANCEES

1. Les tableaux

Une Un **tableau** est un regroupement de variables de même type, il est identifié par un nom. Chacune des variables du tableau est numérotée, ce numéro s'appelle un indice. Chaque variable du tableau est donc caractérisée par le nom du tableau et son indice.

Si par exemple, T est un tableau de 10 variables, alors chacune d'elles sera numérotée et il sera possible de la retrouver en utilisant simultanément le nom du tableau et l'indice de la variable.

Les différentes variables de T porteront des numéros de 0 à 9, et nous appellerons chacune de ces variables un élément de T.

a) Déclaration :

Comme les variables d'un tableau doivent être de même type, il convient de préciser ce type au moment de la déclaration du tableau. De même, on précise lors de la déclaration du tableau le nombre de variables qu'il contient.

Syntaxe :

```
<type> <nomdutableau> [<taille>];
```

Exple :

```
int tab [5]; //Déclare un tableau de 5 entiers
```

b) Initialisation :

Il est possible d'initialiser les éléments d'un tableau à la déclaration, on fait celà comme pour des variables scalaires :

```
<type> <nom_du_tableau> [<taille>] = <valeur_initiale>;
```

La seule chose qui change est la façon d'écrire la valeur d'initialisation, on écrit entre accolades tous les éléments du tableau, on les dispose par ordre d'indice croissant en les séparant par des virgules.

La syntaxe générale de la valeur d'initialisation est donc :

```
<type> <nom> [<taille>] = { <valeur_0>, <valeur_1>, ..., < valeur_n> };
```

Par exemple, on crée un tableau contenant les 5 premiers nombres impairs de la sorte :

```
int  T[ 5 ] = {1 , 3 , 5 , 7 , 9 } ;
```

c) Accès aux éléments :

Les éléments d'un tableau à n éléments sont indicés de 0 à n- 1. On note T[i] l'élément d'indice i du tableau T. Les cinq éléments du tableau de l'exemple ci- avant sont donc notés T[0], T[1], T[2], T[3] et T[4].

2. Les chaînes de caractères

Une **chaîne de caractères** est un tableau de char contenant un caractère nul. Le caractère nul à 0 pour code ASCII et s'écrit '\0'. Les valeurs significatives de la chaîne de caractères sont toutes celles placées avant le caractère nul.

On remarque donc que si le caractère nul est en première position, on a une chaine de caractères vide.

Par exemple, la phrase "Toto" sera codée de la sorte : 'T' 'o' 't' 'o' '\0'

Prenez bien note du fait que le dernier caractère de la chaine est suivi d'un caractère nul.

a) Déclaration :

Comme une chaîne de caractères est un tableau de char, on le déclare :

```
char  <nom_chaine> [<taille_chaine>] ;
```

Par exemple, on déclare une chaine c de 200 caractères de la sorte :

```
char  mot[100] ;
```

Attention !!! Le nombre maximal de lettres qu'il sera possible de placer dans c ne sera certainement pas 200 mais 199, car il faut placer après le dernier caractère de la chaîne un caractère nul !

b) Initialisation :

On initialise une chaîne à la déclaration, et seulement à la déclaration de la sorte :

```
char <nom_chaine> [<taille_chaine>] = <valeur_initiale>;
```

Ou la valeur d'initialisation contient la juxtaposition de caractères formant la chaîne entourée de guillemets (double quotes).

Par exemple :

```
char mot [ 50 ] = "Toto";
```

Cette instruction déclare une chaîne de caractères « mot » initialisée à "Toto". Les 5 premiers éléments du tableau seront occupés par les 4 caractères de la chaîne ainsi que par le caractère nul, les autres contiendront des valeurs non significatives. Observez bien l'exemple suivant :

```
char mot[4] = "Toto";
```

Cette déclaration engendrera un warning a la compilation et probablement une erreur à l' exécution car l'affectation du caractère nul à la 5ème position du tableau donnera lieu à un débordement d'indice.

c) Accès aux éléments :

Du fait qu'une chaîne de caractère est un tableau, il est aisé d'en isoler un élément. Ainsi c[i] est le (i+1)ième élément de la chaîne c.

On teste donc si le premier caractère de c est une majuscule de la sorte :

```
if ( c[0] >= 'A' && c[0] <= 'Z ' )
     printf ("Cette phrase commence par une majuscule \n") ;
else
     printf ("Cette phrase ne commence pas  par une majuscule \n") ;
```

Cette propriéte permet aussi d'afficher une chaîne caractère par caractère :

```
int i = 0 ;
while( c[i] != 0 )
     printf( "%c" , c[i++]);
```

Notez que le corps de la boucle **while** est itéré jusqu'à ce que le caractère nul soit rencontré. Il est donc impératif que votre chaîne se termine par le caractère nul et que le caractère nul se trouve dans la plage d'indices du tableau.

Est- ce que le code suivant est correct ?

```
char c[26];
int i;
for ( i = 0; i < 2 6 , i++)
     c[i] = 'a' + i;
```

Si la question vous est posée, vous pouvez présumer que ce code n'est pas correct. Le fait que chaque élément du tableau contienne un caractère non nul ne peut que corroborer cette présomption…

Si l'on souhaite placer l'alphabet dans une chaîne, on procède de la sorte :

```
char c[27];
int i;
for ( i = 0; i < 2 6; i++)
      c[i] = 'a' + i ;
c[26] = 0;
```

Notez bien que le tableau contient 27 éléments (si l'on compte le caractère nul), et que celui-ci est place à la fin du tableau juste après la boucle.

d) Affichage :

Nous avons vu qu'il était possible d'utiliser le fait qu'une chaîne est un tableau pour l'afficher. Il existe une méthode plus simple, en utilisant printf avec la chaîne de format "%s". Par contre soyez attentifs au fait que si votre chaîne ne contient pas de caractère nul ou que le caractère nul se trouve en dehors de la plage d'indice de votre chaîne, il faudra vous attendre aux pires horreurs à l'exécution !

Dans le code donné en exemple nous pouvons donc écrire l'instruction d'affichage de la chaîne saisie par l'utilisateur :

```
printf( "Vous avez  saisi :  \n%s", mot) ;
```

Notez que lors de l'affichage des chaînes, on ne met pas le caractère « & » comme pour les types primitifs.

e) La bibliothèque « string.h » :

Cette bibliothèque propose des fonctions de maniement de chaînes de caractères, à savoir :

- **strcmp** : comparer deux chaînes.
- **strlen** : longueur d'une chaîne de caractère
- **strsubs** : rechercher une sous-chaîne
- **strcat** : concaténer (juxtaposer) deux chaînes
- **strcpy** : copier une chaîne

3. Les types structurés

Soit Une **structure**, appelé **enregistrement** dans d'autres langages, est une variable contenant plusieurs variables, appelé es champs. Si une structure t contient un char et un int, chacun de ces champs portera un nom, par exemple c et i. Dans ce cas, **t.c** désignera le char de t et **t.i** le int de t.

a) Création :

Pour créer un type structuré, on utilise la syntaxe suivante :

```
struct <nom_du_type>
{
      <typechamp_1> <nomchamp_1>;
      <typechamp_2> <nomchamp_2>;
      ...
      <typechamp_n> <nomchamp_n>;
};
```

On précise donc le nom de ce type structure, et entre les accolades, la liste des champs avec pour chacun d'eux son type. Vous remarquez que le nombre de champs est fixe d'avance. On n'accède pas à un champ avec un indice mais avec un nom. Considérons par exemple le type structure suivant :

```
struct Point
{
      double abscisse;
      double ordonnee;
};
```

Ce type permet de représenter un point dans IR^2, avec respectivement une abscisse et une ordonnée.

« **struct** Point » est maintenant un type, il devient possible de déclarer un point P comme toutz autre variable :

```
struct Point P;
```

b) Accès aux champs :

On accède aux champs d'une variable structurée à l'aide de la notation pointée « nom_variable.nom_du_champ ».

Ainsi, le champ « ordonnee » de notre variable P sera accessible avec « P.ordonnee » et le champ « abscisse » de notre variable P sera accessible avec « P.abscisse ».

Voici un exemple de programme illustrant ce principe :

*

```
#include <stdio.h>
struct Point
{
      double abscisse;
      double ordonnee;
} ;

main ( )
{
      struct Point P;
      P.ordonnee = 2;
      p.abscisse = p.ordonnee + 1;
      printf ( "P = (%f , %f )  \n" , p.abscisse, p.ordonnee ) ;
}
```

Ce code affiche :

```
P = (3.000000, 2.000000)
```

c) Les « typedef » :

On se débarrasse du mot clé « struct » en renommant le type, on utilisera par exemple la syntaxe suivante :

```
#include <stdio.h>
typedef struct Point
{
      double abscisse;
      double ordonne;
}  Point ;
```

```
main ( )
{
      Point P;
      P.ordonnee = 2;
      P.abscisse = p.ordonnee + 1;
      printf ("p = (%f, %f)" , p.abscisse , p.ordonnee);
}
```

« Point » est donc le nom de ce type structuré.

4. Les pointeurs

Un **pointeur** est une variable qui contient l'adresse mémoire d'une autre variable.

Déclaration : « **T*** » est le type d'une variable contenant l'adresse mémoire d'une variable de type T.

Si une variable p de type T* contient l'adresse mémoire d'une variable x de type T, on dit alors que « p pointe vers x » (ou bien sur x). « **&x** » est l'adresse mémoire de la variable x.

Exposons cela dans un exemple :

```
#include <stdio.h>
void main () {
      int x = 3 ;
      int* p;
      p = &x;
}
```

x est de type int. p est de type int*, c'est à dire de type pointeur de int, p est donc faite pour contenir l'adresse mémoire d'un int. &x est l'adresse mémoire de la variable x, et l'affectation p = &x place l'adresse mémoire de x dans le pointeur p. A partir de cette affectation, p pointe sur x.

- Accès à la variable pointée :

Si p pointe sur x, alors il est possible d'accéder à la valeur de x en passant par p. Pour le compilateur, *p est la variable pointée par p, cela signifie que l'on peut, pour le moment du moins, utiliser indifféremment *p ou x. Ce sont deux façons de se référer à la même variable, on appelle cela de l'aliasing.

Explicitons cela sur un exemple :

```
#include <stdio.h >
void main  ( )
{
      int x = 3 ;
      int * p ;
      p = &x;
      printf( "x = %d  \n" , x) ;
      *p = 4;
      printf( "x = %d  \n" , x);
}
```

- Application aux tableaux :

Il est possible d'aller plus loin dans l'utilisation des pointeurs en les employant pour manipuler des tableaux.

Commençons par observer l'exemple suivant :

```
#include <stdio.h>
main ( )
{
      char  t[10] ;
      char* p ;
      t[0] = 'a' ;
      p = t ;
      printf ("Le premier élément du tableau est %c .  \n", *p) ;
}
```

La variable t contient l'adresse mémoire du premier élément du tableau t. p est un pointeur de char, donc l'affectation p = t place dans p l'adresse mémoire du premier élément du tableau t. Comme p pointe vers t[0], on peut indifféremment utiliser *p ou t[0].

Donc ce programme affiche : « Le premier élément du tableau est a »

On rappelle qu'une variable de type char occupe un octet en mémoire. Considérons maintenant les déclarations : **char** t[10]; et **char*** p = t;

Nous savons que si p pointe vers t[0], il est donc aisé d'accéder au premier élément de t en utilisant le pointeur p. Mais comment accéder aux autres éléments de t ? Par exemple T [1] ?

Souvenez- vous que les éléments d'un tableau sont juxtaposés, dans l'ordre, dans la mémoire. Par conséquent, si p est l'adresse mémoire du premier élément du tableau t, alors (p + 1) est l'adresse mémoire du deuxième élément de ce tableau. Vous conviendrez que *p est la variable dont l'adresse mémoire est contenue dans p. Il est possible, plus généralement, d' écrire *(p + 1) pour désigner la variable dont l'adresse mémoire est (p + 1), c'est-à-dire la valeur contenue dans (p + 1).

Illustrons cela dans un exemple, le programme :

```
# include <stdio. h >
main ( )
{
      char t[10];
      char* p ;
      t [1] = 'b';
      p = t;
      printf("Le deuxième élément du tableau est %c. \n", *(p +1)) ;
}
```

Ce programme affiche le deuxième élément du tableau est b.

- **Arithmétique des pointeurs :**

Supposons que le tableau t contienne des int, sachant qu'un int occupe 2 octets en mémoire. Est-ce que (t + 1) est l'adresse du deuxième élément de t ?

On ne pondèrera donc pas les indices ! Cela signifie, plus explicitement, que quelque soit le type des éléments du tableau p , l'adresse mémoire du $i^{ème}$ élément de p est p + i.

On le vérifie expérimentalement en exécutant le programme suivant :

```
#include <stdio.h>
#define N 30
void initTab( int* k , int n )
 {
      int i ;
      *k = 1;
      for( i = 1; i < n; i++)
      *(k + i) = *(k + i - 1) + 1 ;
}
void  afficheTab (int* k, int n)
{
      int i;
      for( i = 0; i < n; i++)
            printf ("%d ", *( k + i )) ;
      printf ("\n") ;
}
main()
{
      int t[N] ;
      initTab(t, N);
      afficheTab(t, N);
}
```

5. Allocation dynamique de mémoire

a) La fonction malloc() :

Lorsqu'on déclare un pointeur p, on alloue un espace mémoire pour y stocker l'adresse mémoire d'un entier. Et p, jusqu'à ce qu'on l'initialise, contient n'importe quoi. On peut donc ensuite faire pointer p sur l'adresse mémoire qu'on veut (choisissez de préférence une zone contenant un int…).

Il faut donc dire au compilateur qu'on sait ce qu'on fait, et qu'on est sûr que c'est bien un int qu'on va mettre dans la variable pointée. Pour ce faire, il convient d'effectuer ce que l'on appelle un « cast », en ajoutant, juste après l'opérateur d'affectation, le type de la variable se situant à gauche de l'affectation entre parenthèses. Dans l'exemple ci -avant, cela donne : int* p = (int*) malloc(2).

Voici un exemple illustrant l'utilisation de malloc() :

```
#include <stdio.h>
#include <malloc.h>
main()
{
      int* p;
      p = (int*) malloc(2);
      *p = 28;
      printf("%d\n", *p) ;
}
```

b) La fonction free() :

Lorsque que l'on effectue une allocation dynamique, l'espace réservé ne peut pas être alloué pour une autre variable. Une fois que vous n'en avez plus besoin, vous devez le libérer explicitement si vous souhaitez qu'une autre variable puisse y être stockée. La fonction de libération de la mémoire est free(). free(v) où v est une variable contenant l'adresse mémoire de la zone à libérer. A chaque fois qu'on alloue une zone mémoire, on doit la libérer !

Un exemple classique d'utilisation est :

```
#include <stdio.h>
#include <malloc.h>
main ()
{
      int * p ;
      p = (int*) malloc( 2 ) ;
      *p = 28;
      printf( "%d \n" , *p ) ;
      free(p) ;
}
```

Notons bien que la variable p, qui a été allouée au début du main(), a été libérée par le free(p).

c) La valeur NULL :

Le pointeur p qui ne pointe aucune adresse a la valeur NULL. Attention, il n'est pas nécessaire d'initialiser à NULL, NULL est la valeur que, conventionnellement, on décide de donner à p s'il ne pointe sur aucune zone mémoire valide.

Par exemple, la fonction malloc() retourne NULL si aucune zone mémoire adéquate n'est trouvée. Il convient, à chaque malloc(), de vérifier si la valeur retournée par malloc est différente de NULL.

Par exemple :

```
#include <stdio.h>
#include <malloc.h>
main ()
{
      int* p ;
      p = (int*) malloc(2) ;
      if(p == NULL)
        exit(0);
      *p = 28;
      printf( "%d \n" , *p ) ;
      free(p) ;
}
```

d) Allocation dynamique des tableaux :

Le compilateur met à notre disposition la fonction « sizeof() », qui nous permet de calculer la place prise en mémoire par la variable d'un type donné. Par exemple, soit T un type, la valeur sizeof(T) est la taille prise en mémoire par une variable de type T. Si par exemple on souhaite allouer dynamiquement un int, il convient d'exécuter l'instruction « malloc(sizeof(int)) ».

Attention, sizeof prend en paramètre un type !

Si on souhaite allouer dynamiquement un tableau de n variables de type T, on exécute l'instruction « malloc(n * sizeof (T)) ».

Par exemple, pour allouer un tableau de 10 int, on exécute malloc(10 * sizeof (int)).

Voici une variante du programme d'un programme précédent :

```c
#include <stdio.h>
#include <malloc.h>
#define N 26
void initTab( int* k , int n )
{
      int i ;
      for ( i = 0 ; i < n ; i++)
      *( k + i ) = i + 1 ;
}
void afficheTab ( int* k , int n )
{
      int i ;
      for (  i = 0 ; i < n ; i++)
      printf ( "%d " , *( k + i ) ) ;
      printf ( "  \n" ) ;
}
int main ( )
{
      int* p ;
      p = ( int* ) malloc (N * sizeof(int)) ;
      if(p == NULL)
            return  1;
      initTab (p, N);
      afficheTab(p, N);
      free(p);
      return 0;
}
```

e) Passage de valeur par référence :

« Pour le compilateur, *p est la variable pointé e par p, cela signifie que l'on peut, pour le moment du moins, utiliser indifféremment *p ou x ».

Observons le programme suivant :

```
#include <stdio.h>
void echange ( int x , int y )
{
      int t = x ;
      x = y ;
      y = t ;
}
void main ()
{
      int a = 1 ;
      int b = 2 ;
      printf ( " a = %d , b = %d  \n", a, b ) ;
      echange ( a, b ) ;
      printf ( " a = %d, b = %d  \n", a, b ) ;
}
```

Qu'affiche-t-il ?

```
a = 1, b = 2
a = 2, b = 1
ou bien
a = 1, b = 2
```

Méditons quelque peu: la question que l'on se pose est "Est- ce que le sous-programme échange échange bien les valeurs des deux variable s a et b" ? Il va de soi qu'il échange bien les valeurs des deux variables x et y, mais comme ces deux variables ne sont que des copies de a et b, cette permutation n'a aucun effet sur a et b. Cela signifie que la fonction echange() ne fait rien.

Ce programme affiche donc :

```
a = 1, b = 2
```

- Remarques :

Il existe un moyen de passer des paramètres par référence.

On peut d'ores et déjà retenir que nom_sous_programme(&x) sert à passer en paramètre la variable x par référence. Et finalement, c'est plutôt logique, l'instruction nom_sous_programme(x) passe en paramètre la valeur de x, alors que nom_sous_programme(&x) passe en paramètre l'adresse de x, c'est-à- dire un moyen de retrouver la variable x depuis le sous-programme et de modifier sa valeur.

Cependant, si on écrit « echange(&a, &b) », le programme ne compilera pas… En effet, le sous-programme echange() prend en paramètre des int et si on lui envoie des adresses mémoire à la place, le compilateur ne peut pas "comprendre" ce qu'on veut faire… On va donc devoir modifier le sous-programme echange() si on veut lui passer des adresses mémoire en paramètre.

- Utilisation des pointeurs :

On arrive à la question suivante : dans quel type de variable puis-je mettre l'adresse mémoire d'une variable de type entier ?

La réponse est int*, un pointeur sur int.

Observons le sous - programme suivant :

```
void echange ( int* x , int* y )
{
      int t = *x;
      *x = y;
      *y = t;
}
```

x et y ne sont pas des int, mais des pointeurs sur int. De ce fait le passage en paramètre des deux adresses &a et &b fait pointer x sur a et y sur b. Donc *x est un alias de a et *y est un alias de b.

Il suffit donc, pour écrire un sous-programme prenant en paramètre des variables passées par référence, de les déclarer comme des pointeurs, et d'ajouter une « * » devant à chaque utilisation.

6. Les listes chaînées

a) Pointeurs et structures :

Soit T un type définit comme suit :

```
typedef struct T
{
      int i;
      char c;
} T;
```

Si p est un pointeur de type T*, alors p contient l'adresse mémoire d'un élément de type T.

Soit t une variable de type T, et soit l'affectation p = &t. Alors, p pointe sur t. De ce fait *p et t sont des alias, et nous pourrons indifférement utiliser t.i (resp. t.c) et (*p) .i (resp (*p). c).

Par exemple, réécrivons le programme de l'exemple du cours sur les structures :

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h >
#define N 10

/* déclaration du type */
typedef struct Point {
      double abs;
      double ord;
} Point;

/* fonction nextoint */
Point nextPoint( Point* previous )
{
      Point result ;
      result.ord = (*previous).ord + 1;
      result.abs = (*previous).abs + 2;
      return result;
}
```

```
void initTableauPoints( Point* p, int n ){
     int i ;
     (*p).ord = 0;
     (*p).abs = 1;
     for ( i = 1; i < n; i++)
          *(p + i) = nextPoint ( p + i + 1 );
}
void affichePoint ( Point* p ) {
  printf( "(%f , %f ) ", (*p).abs, (*p).ord ) ;
}
void afficheTableauPoints  ( Point p, int n ) {
     int i ;
     for ( i = 1 ; i < n ; i++)
     {
          printf ( "p[%d] = ", i );
          affichePoint ( p + i );
          printf ( " \n" );
     }
}
int main  ( ) {
     Point* p;
     p = ( Point* ) malloc (N * sizeof( Point ));
     if ( p == NULL)
          exit( 0 );
     initTableauPoints( p, N) ;
     afficheTableauPoints( p, N) ;
     free( p ) ;
     return 0 ;
}
```

L'écriture (*p).i permet de désigner le champ i de la variable pointée par p. Cette écriture, peu commode, peut être remplacée par « p -> i », par exemple :

```
#include <stdio.h >
#include <malloc.h>
#include <stdlib.h>
#define N 10

typedef struct Point
{
      double abs ;
      double ord ;
} Point;

Point nextPoint( Point* previous )
{
      Point result;
      result.ord = previous->ord + 1;
      result.abs = previous->abs + 2;
      return result;
}

void iniTableauPoints ( Point p , int n )
{
      int i ;
      p->ord = 0 ;
      p->abs = 1 ;
      for ( i = 1 ; i < n ; i++)
            *( p + i ) = nextPoint ( p + i - 1 ) ;
}
```

```
void initTableauPoints ( Point* p , int n )  {
     int i ;
     p->ord = 0;
     p->abs = 1;
     for ( i = 1 ; i < n ; i++)
          * ( p + i ) = nextPoint( p + i - 1  );
}

void affichePoint ( Point* p ){
     printf( "(%f, %f ) ", p->abs, p->ord );
}

void afficheTableauPoints ( Point* p, int n ) {
     int i;
     for ( i = 1; i < n; i++)
     {
          printf( "p[%d] = ", i ) ;
          affichePoint ( p + i ) ;
          printf("\n" ) ;
     }
}
main ( )
{
     Point* p;
     p = (Point*) malloc (N *sizeof(Point) ) ;
     if(p == NULL)
          exit( 0 ) ;
     initTableauPoints( p, N) ;
     afficheTableauPoints ( p, N) ;
     free( p ) ;
}
```

Remarques :

Les opérateurs d'accès aux champs « . » et « -> » ont une priorité supérieure à celles de tous les autres opérateurs du langage !

Si on manipule un tableau de structures « t » et qu'on souhaite accéder au champ « c » du $i^{ème}$ élément, est- il intelligent d'écrire *(i + t).c ? Absolument pas !

« . » est prioritaire sur « * », donc le parenthésage implicite est *((i + t).c), essayez de vous répresenter ce que cela fait, et vous comprendrez pourquoi votre programme plante !

En écrivant « (i + t)->c », vous obtenez une expression équivalente à (*(i + t)).c, qui est déjà bien plus proche de ce que l'on souhaite faire.

b) Un premier exemple :

Considérons le type suivant :

```
typedef struct maillon
{
      int data ;
      struct maillon  *next;
} maillon;
```

On appelle cette forme de type un type récursif, c'est -à- dire qui contient un pointeur vers un élément du même type.

Observons l'exemple suivant :

```c
#include <stdio.h>
typedef struct maillon
{
     int data ;
     struct maillon  *next;
} maillon;

int main  ( )
{
     maillon m, p;
     maillon* ptr;
     m.data = 1;
     m.next = &p;
     p.data = 2 ;
     p.next = NULL;
     for ( ptr = &m; ptr != NULL; ptr = ptr->next )
          printf ( " data = %d \n" , ptr->data ) ;
     return 0;
}
```

m et p sont deux maillons, et m->next pointe vers p.

Lorsqu'on initialise ptr à &m, ptr pointe sur m. Donc, lors de la première itération de la boucle for , la valeur « ptr->data » est « m.data », à savoir 1. Lorsque le pas de la boucle est exécuté, ptr reçoit la valeur ptr->next , qui n'est autre que m->next , ou encore &p. Donc ptr pointe maintenant sur p.

Dans la deuxième itération de la boucle, ptr->data est la valeur p.data, à savoir 2. Ensuite le pas de la boucle est exécute, et ptr prend la valeur ptr->next , à savoir « p->next » , ou encore NULL. Comme ptr == NULL, alors la boucle s'arrête.

Ce programme affiche donc :

```
data = 1
data = 2
```

c) Le chaînage :

La façon de renseigner les valeurs des pointeurs « next » observée dans l'exemple précédent s'appelle le chaînage. Une liste de structures telle que chaque variable structurée contienne un pointeur vers une autre variable du même type s'appelle une liste chaînée.

d) Utilisation de malloc() :

Pour le moment, nous avons stocké les éléments dans un tableau, les maillons étaient donc regroupés dans des zones mémoire contigües. Il est possible de stocker les maillons dans les zones non contigües, tous peuvent être retrouvés à l'aide du chaînage.

Par exemple, pour plus de clarté, on placera l'initialisation de la liste dans une fonction:

```
#include <stdio.h >
#include <stdlib.h >
#include <malloc. h>
#define N 10

typedef struct maillon
{
      int data ;
      struct   maillon* next ;
} maillon;

void printLL   ( maillon *  ptr)
{
      for (; ptr != NULL; ptr = ptr->next )
      printf( " data = %d \n", ptr- >data ) ;
}
```

```
maillon* initLL( int n ) {
     maillon *first;
     maillon *current ;
     maillon *previous ;
     int i ;
     first = (maillon*) malloc ( sizeof ( maillon ) ) ;
     if(first == NULL)
          exit( 0 ) ;
     first->data = 0 ;
     previous = first ;
     for ( i = 1 ; i < n ; i++)
     {
          current = (maillon*) malloc(sizeof(maillon)) ;
          if ( current == NULL)
               exit( 0 ) ;
          current->data = i ;
          previous->next =  current ;
          previous = current ;
     }
     current->next = NULL;
     return first ;
}
void freeLL ( maillon* l ){
     maillon * n ;
     while( l != NULL)
     {
         n = l->next ;
         free( l ) ;
         l = n ;
     }
}
```

```
int main  ( )
{
   maillon* l ;
   l = initLL(N) ;
   printLL(l)  ;
   freeLL (l) ;
   return 0 ;
}
```

Ce programme affiche :

```
data = 0
data = 1
data = 2
data = 3
data = 4
data = 5
data = 6
data = 7
data = 8
data = 9
```

e) Opérations sur les listes :

Observons de quelle façon il est possible d'effectuer certaines opérations sur une liste chaînée.

Les sous-programmes ci -dessous ont été pensés et combinés pour être les plus simples possibles…

- Création d'un maillon :

```
maillon* creeMaillon( int n )
{
      maillon* l;
      l = (maillon *) malloc( sizeof(maillon) );
      if( l == NULL)
            exit(0) ;
      l->data = n;
      l->next = NULL;
      return l;
}
```

- Insertion au début d'une liste chaînée :

```
maillon* insereDebutLL (maillon* l, int n )
{
      maillon* first = creeMaillon ( n );
      first- >next = l;
      return first;
}
```

- Création d'une liste chaînée :

```
maillon* initLL ( int n )
{
      maillon * l = NULL;
      int i ;
      for ( i = n - 1 ; i > = 0 ; i-- )
            l = insereDebut( l , i ) ;
      return l ;
}
```

f) Listes doublement chaînée :

Un maillon d'une liste doublement chaînée contient deux pointeurs, un vers le maillon précédent, et un vers le maillon suivant.

```
typedef struct dmaillon
{
      int data;
      struct dmaillon* previous;
      struct d maillon* next;
} maillon;
```

Aussi paradoxal que cela puisse paraître, bon nombre d'opérations sont davantage aisées sur des listes doublement chaînées.

Pour se faciliter la tâche, nous manipulerons les listes chaînées à l'aide de deux pointeurs, un vers son premier élément, et un vers son dernier :

```
typedef struct dLinkedList
{
      struct dmaillon* first;
      struct dmaillon* last;
} dLinkedList;
```