Table des matières

A. Introduction dans la programmation C

1. Qu'est ce que programmer ?

Un programme est une suite finie d'opérations réunies et organisées dans un ordre précis en vue de l'accomplissement d'une ou de plusieurs tâches _les tâches que l'ordinateur accomplit lorsque le programme est lancé. Ces opérations s'appellent des instructions. Elles sont écrites et définies avec un langage de programmation et il existe des milliers de langages de programmation. Dans chaque langage les instructions reposent sur des données qui sont des informations stockées en mémoire sous forme de ce que l'on appelle les types et structures de données.

Une suite finie d'opérations réunies et organisées dans un ordre précis en vue de l'accomplissement d'une ou de plusieurs tâches est aussi la définition d'un algorithme. Un programme est donc un algorithme ou un ensemble d'algorithmes réunis entre eux par... un algorithme. Toute combinaison d'instructions en vue de l'accomplissement d'une tâche est un algorithme.

Programmer consiste donc à définir et ordonner dans un langage de programmation les instructions que la machine devra exécuter, c'est à dire trouver le bon algorithme et les bonnes structures de données pour atteindre un objectif donné.

2. Le positionnement par rapport à l'ordinateur

L'ordinateur est construit sur plusieurs niveaux superposés et solidaires entre eux. Chaque niveau s'appuie sur le niveau précédent. Il en réduit la complication et permet d'élaborer des opérations plus complexes. Nous pouvons le schématiser de la façon suivante :

NIVEAUX / CORRESPONDANCE

1 / Programmes applicatifs

2 / Langage de programmation <- langage C, nous sommes ici

3 / Langage assembleur

4 / Noyau du système d'exploitation

5 / Langage machine

6 / Microprogramme

7 / Logique numérique

1. **Logique numérique**, circuits électroniques de l'ordinateur, portes logiques (ET, OU) univers binaire 0/1
2. **Microprogramme**, premier niveau de langage, tous les ordinateurs ne le possèdent pas (des séquences d'étapes utilisées par le niveau 2 du langage machine)

3. **Langage machine** : à ce niveau, ajouter 2 nombres, déplacer des données d'un emplacement vers un autre, déterminer si un nombre est égal à 0 sont des instructions élémentaires suffisantes pour exécuter n'importe quel programme ou application des niveaux plus élevés.
4. **Noyau système exploitation** : ordonnancer et allouer les ressources d'un ordinateur aux différents programmes s'exécutant sur la machine. Il peut être programmé dans un langage de programmation de haut niveau qui a été traduit en langage machine, c'est-à-dire compilé. Rappelons que Le langage C a été créé initialement pour écrire le système d'exploitation UNIX. En plus du noyau le système d'exploitation complet comprend des programmes applicatifs (niveau 7), avec le plus souvent un rôle d'interface (gestion des fichiers, gestion des fenêtres etc.)
5. **Le langage d'assemblage** est une représentation symbolique des instructions rencontrées aux niveaux inférieurs. Un programme en langage d'assemblage est converti en instructions de niveau inférieur par un traducteur appelé un assembleur
6. **Les langages de hauts niveaux** grâce auxquels les applications peuvent être écrites plus facilement qu'en langage assembleur. Il existe des milliers de langages à ce niveau (les plus connus, pascal, Cobol, Fortan, Lisp, Basic, C, C++, java, prolog, C#, …) Ils font eux-mêmes l'objet d'une classification en strates C, C++, JAVA, JAVASCRIPT… par exemple)
7. **Les applications (logiciels)** ou collections de programmes dans des domaines multiples et variés.

3. Les logiciels de la programmation

a. Le compilateur

Le compilateur est un traducteur qui lit un programme écrit dans un premier langage, le langage source et le traduit dans un programme équivalent écrit dans un autre langage, le langage cible. Au passage il signale s'il y a des erreurs dans le programme source. Lorsque nous écrivons un programme en C, le langage C est le source et la compilation réalise sa traduction en langage machine, c'est à dire un exécutable.

En général le compilateur est accompagné d'une interface de type traitement de texte qui permet l'écriture des programmes en langages source.

b. L'IDE : Environnement de développement

Pour être utilisable le compilateur est complété par un environnement de développement : EDI pour Environnement de Développement Intégré ou en anglais IDE pour Integrated Development Environment. C'est un programme ou un ensemble de programmes corrélés entre eux. Il regroupe un traitement de texte spécialisé pour le code source avec des options spécifiques automatisées (indentation automatique, coloration des mots clés, recherche et remplacement d'expressions etc.), des outils automatiques de paramétrage, diverses options, et souvent un débogueur. Par exemple le compilateur libre MingW peut être utilisé

avec plusieurs IDE : DevC++, CodeBlock, NetBeans. Souvent l'environnement est fourni avec le compilateur mais ce n'est pas toujours le cas.

c. Installation

A voir selon le compilateur et l'environnement de développement.

4. Base d'un programme

a. Des données et des instructions

Pour écrire un programme nous disposons dans tous les langages de deux grandes catégories d'outils :

- ce qui a trait au stockage des informations et concerne la mémoire de l'ordinateur. C'est que l'on appelle "les structures de données". Elles reposent sur différentes espèces de variables. Chaque type de variable est déterminé par un mot clé. En C nous avons au total les variables de type char, short, int, long, float, double, struct, tableaux et pointeurs.

- ce qui a trait aux traitements (calculs) appliqués aux données, ce sont les instructions. Comme chaque programme est une création nous ne pouvons pas savoir à l'avance quels seront les traitement à faire. Le langage de programmation fournit donc des instructions de base à partir desquelles nous pouvons construire nos propres instructions c'est à dire nos fonctions. Les instructions natives dans un langage sont désignées par un mot clé. Le C fourni les instructions suivantes : if, if-else, if-else if-else, switch, while, do-while, for ainsi que différentes variétés d'opérateurs : affectation, arithmétiques, bits à bits, comparaisons, pointeurs, tableau, structures, fonction.

En définitive, il y a peu de chose dans un langage, mais avec il y a une infinité de choses possibles à faire. Un peu comme en musique avec treize notes nous pouvons inventer une infinité de musiques.

b. Des "librairies" de fonctions

Par ailleurs le programmeur n'invente pas la roue à chaque nouveau programme et il est possible d'utiliser des instructions sophistiquées déjà écrites pour des opérations fréquentes par exemple ce qui concerne les fichiers, les affichages etc. En C ces instructions se présentent sous la forme de fonctions et nous disposons en standard de librairies de fonctions classées par thèmes : entrées-sorties, traitements de chaines de caractères, des utilitaires, des fonctions mathématiques etc.

Il y a de nombreuses autres possibilités de librairies de fonction notamment pour la création de jeux video, le traitement d'images, le son... Certaines, en "open source" sont libres de droit. D'autres, propriétaires, peuvent être vendues très chères.

Lors de la réalisation d'un projet conséquent nous créons nous-mêmes nos propres librairies.

5. Premier programme

a. La fonction main() : entrée du programme

(Une fonction est toujours désignée par un nom et des parenthèses à droite. Par exemple la fonction maFonct : maFonct() ou la fonction main : main())

Pour le système d'exploitation, un exécutable se traduit par une pile d'instructions avec une entrée, un début, qui est appelé la "tête" du programme. Pour le programmeur c'est la fonction main(). Ainsi tout programme commence par un main().

Selon le système d'exploitation ou l'environnement de développement, le main() peut avoir des caractéristiques spécifiques. Voici un exemple de programme avec un main() standard. Ce programme est complet, une fois compilé il marche mais il ne fait rien :

```
int main()                    // tête ou entrée du programme,
{                                        // ouverture bloc des
instructions du programme



                                                  /* valeur de
retour de la fonction main qui indique
                          un bon déroulement du programme. */
          return 0;

}                                        // fermeture bloc des
instructions du programme
```

Les commentaires // et /...*/*

Le signe // dans un programme indique que tout ce qui suit sur la ligne (et sur la ligne uniquement) est un commentaire et ne sera pas pris en compte au moment de la compilation, c'est à dire lors de la fabrication de l'exécutable en langage machine. Cela permet d'introduire des explications dans le code source pour le rendre plus clair et compréhensible.

De même tous le texte ou le code qui se trouve entre /* et */ est mis en commentaire.

b. Afficher du texte avec la fonction printf()

La fonction printf() est dans la librairie dans <stdio.h>. Pour pouvoir l'utiliser il faut inclure la librairie dans notre projet. Pour ce faire nous utilisons l'instruction #include avant le commencement du programme au dessus du main() ce qui donne dans le programme :

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

Les chevrons indiquent à la machine que le fichier correspondant se trouve dans le dossier include du compilateur. Si ce n'est pas le cas il y aura un message d'erreur à la compilation.

La fonction printf() permet d'afficher du texte dans une fenêtre console. Un ou plusieurs mots, une ou plusieurs phrases : le texte à afficher doit être spécifié entre

les parenthèses à droite de la fonction. Par exemple pour afficher le texte "il fait beau" je devrai écrire dans mon code :

```
printf("il fait beau");
```

Le texte entre parenthèse doit obligatoirement être entre guillemets : "le texte que je veux afficher", et après la parenthèse fermante il faut un point-virgule.

Le texte entre guillemets est appelé une chaine de caractères et le point-virgule signale une instruction.

```
#include <stdio.h>           // pour avoir accès à la fonction
printf()

int main()                   // entrée du programme,
{                            // ouverture bloc d'instructions

   /* appel de la fonction printf() qui affiche la chaîne
      de caractères passée en  paramètre */
   printf("bonjour");

   /* arrêter le programme pour avoir le temps de
   lire le résultat */
   system("PAUSE");

   /* valeur de retour de la fonction main qui indique un bon
   déroulement du programme.*/
   return 0;

}                            // fermeture bloc d'instructions
```

6. Mise en pratique : découverte du compilateur

Exercice 1
Installer le compilateur et IDE codeBlock s'il n'est pas encore installé. Le fichier d'installation est téléchargeable sur http://fdrouillon.free.fr
Une fois l'installation terminée, repérer sur le disque l'emplacement des dossiers include et lib du compilateur. Attention à ne rien modifier. Lancer le programme codeBlock.

Exercice 2
Pour repérer les étapes de la création d'un projet, faire un projet "console application". Trouvez les commandes adéquates dans le programme. Vous spécifierez un dossier sur votre disque dur pour votre projet, et vous lui attribuerez un nom.

Exercice 3
Découverte du main() : compiler un programme qui ne fait rien

Exercice 4
Premier programme : afficher "ce que vous voulez" avec la fonction printf()
Recommencer en modifiant quelque chose à chaque fois, par exemple :

- ajouter \n à la fin du message, des accents ô à, é, è, ê etc.
- supprimer le point-virgule, une ou deux guillemets, une parenthèse...

Dans quels cas y a t-il un message d'erreur et quel est-il ?

Exercice 5

Faire un dessin en utilisant plusieurs fois de suite la fonction printf() et plusieurs affichages successifs de chaînes de caractères.

B. Variables simples

1. Qu'est ce qu'une variable en informatique ?

Pour la machine une variable c'est un espace mémoire réservé et accessible qui permet de stocker et d'utiliser des valeurs numériques. Il y a deux actions possibles

- donner une valeur à la variable
- utiliser la valeur de la variable (dans un calcul par exemple)

Quelque soit le langage il y a toujours plusieurs types de variables. Chaque type est identifié par un mot clé. Il définit une taille en mémoire pour la variable et certaines propriétés. Pour les variables simples il s'agit uniquement de savoir s'il y a une virgule ou pas, si elles sont signées ou non (+, -).

2. Avoir des variables dans un programme

a. Les types de variables élémentaires en C

Chaque type est déterminé par un mot clé du langage, il y en a sept en tout :

Mots clé	Taille en octet	Nombre de codes possibles	Plage en non signé	Plage en signé	Propriétés
char	1	2^8	0 à 2^8-1	-2^7 à 2^7-1	Entiers
short	2	2^{16}	0 à 2^{16}-1	-2^{15} à 2^{15}-1	entiers
int	2 ou 4 (selon environnement)	2^{16} ou 2^{32}	0 à 2^{16}-1 ou 0 à 2^{32}-1	-2^{15} à 2^{15}-1 ou -2^{31} à 2^{31}-1	entiers
long	4	2^{32}	0 à 2^{32}-1	-2^{31} à 2^{31}-1	entiers
float	4	2^{32}	0 à 2^{32}-1	-2^{31} à 2^{31}-1	Réels (virgule)
double	8	2^{64}	0 à 2^{64}-1	-2^{31} à 2^{31}-1	Réels (virgule)
Pointeur (opérateur *)	4	2^{32}	0 à 2^{32}-1	Contient des adresses mémoires	Contient des adresses mémoires

Par défaut toutes les variables sont signées. Le mot clé signed est utilisé implicitement par la machine. Pour avoir des variables non signées il faut le spécifier explicitement avec le mot clé unsigned.

Le cas de la variable pointeur est différent, les pointeurs sont des variables qui contiennent uniquement des adresses mémoire. Ils ne sont pas utilisés pour des

calculs mais pour des opérations spécifiques en relation avec la mémoire (allocation dynamique, passage par référence en paramètre de fonction, construction de listes chainées et d'arbres) .

b. Déclarer ses variables dans un programme

Pour avoir une variable dans un programme il faut premièrement définir sa variable avec un type et un identificateur (c'est à dire un nom pour la variable). Par exemple :

```
int toto
```

Définit une variable de type int qui s'appelle toto

Et deuxièmement il faut ajouter un point-virgule à l'expression :

```
int toto;
```

Le point-virgule transforme l'expression en une instruction qui sera exécutée par le programme. En l'occurrence le programme réserve un espace mémoire selon le type de la variable afin de pouvoir y écrire et y lire des valeurs.

La forme générale est donc :

```
<type>  <identificateur> <;>
```

C'est ce que nous appelons une déclaration de variable. En C toute variable doit être déclarée avant son utilisation sinon il y a une erreur à la compilation.

Voici un programme complet qui marche : il ne fait que déclarer des variables de types différents :

```
int main()
{
   char c;
   short sh;
   int i;
   long l;
   float f;
   double d;

   // fin
   return 0;
}
```

c. Contraintes pour le choix des noms

En C l'identificateur, le nom de la variable doit respecter les règles suivantes :

- le premier caractère est obligatoirement une lettre ou le caractère _
- ensuite on peut utiliser des lettres majuscules, des lettres minuscules, des chiffres et le caractère de soulignement _
- les espaces et tous les opérateurs (-, +, /, *, =, %) ne sont pas autorisés

Le non respect de ces règles produit des erreurs de compilation et empêche la réalisation de l'exécutable.

Par ailleurs il est important de choisir des noms significatifs pour ses identificateurs afin de rendre plus compréhensible son programme.

3. Mise en pratique : avoir des variables dans un programme

Exercice 1
Combien y a t-il de type en C et quels sont-ils ?
Qu'est ce qui les différencie ? Par défaut sont-ils signés ou non signés ?
Quelles sont les fourchettes de valeurs ?

Exercice 2
Quels sont les types à choisir pour coder les nombres
45.876, 56.0, 77, 650987, 32769, - 32765, 450009996 ?

Écrivez un programme dans lequel vous avez des variables susceptibles de recevoir ces valeurs.

Exercice 3
Soit dans un programme les déclarations suivantes :

int 0t, ti, p0;
freste float;
double div-total;
float tata, t2345, char c, cc :
short Err_, _E_,

Indiquez, expliquez et corrigez les erreurs (vous pouvez utiliser le compilateur).

4. Manipulations de base sur les variables

a. Affecter une valeur à une variable

L'opérateur d'affectation est l'opérateur =

Dans un programme les instructions :

```
float toto;     // déclare une variable int
   toto=1.5;    // affecte la valeur 1.5 à toto (toto vaut 1.5)
```

Il y a aussi la possibilité d'initialiser la variable à la déclaration, ce qui donne :

```
float toto = 1.5; // La virgule est indiquée avec un point
```

Que fait le programme suivant ?

```
int main()
{
int a,b;
    a=10;
    b=0;
    a=b;
    b=555;
    b=a;
    a=3;
    b=a;
   return 0;   // ici combien bien valent a et b ?
}
```

Pour le savoir il faut simuler intellectuellement le fonctionnement du programme.

Les instructions (expressions closes par un point-virgule) sont exécutées les unes à la suite des autres, de façon linéaire, dans l'ordre où elles sont présentées et sans jamais revenir en arrière.

Dans ce programme il y a neuf instructions. Du haut vers le bas chaque ligne donne

```
ligne 1 : Au départ déclaration de deux variables de type int a et b
ligne 2 : a prend la valeur 10, a vaut 10
ligne 3 : b prend la valeur 0, b vaut 0
ligne 4 : a prend la valeur de b, a vaut 0,
ligne 5 : b prend la valeur 555, b vaut 555,
ligne 6 : b prend la valeur de a, b vaut 0
ligne 7 : a prend la valeur 3, a vaut 3
ligne 8 : b prend la valeur de a, b vaut 3
ligne 9 : à l'issue, au moment du return, a vaut 3 et b  vaut 3.
```

Autre exemple avec des flottants :

```
int main()
{
float f1=8.78654, f2=7.77;
   f1=f2;
   f1=0.999;
   f2=f1;
   return 0;
}
```

A la fin f1 vaut 0.999 et f2 vaut 0.999.

Attention !

Dans un programme en C l'expression a = b ne veut jamais dire que je compare a et b du point de vue de leur égalité, mais que je copie la valeur numérique de la variable b dans la variable a, que j'affecte la valeur de la variable b à la variable a.

b. Des valeurs de type caractère (codage ASCII)

Remarquons au passage que certaines valeurs codées en 8 bits correspondent à des caractères. Les correspondances entre ces valeurs numériques et les caractères sont répertoriées dans une table, la table ascii.

Par exemple :

la lettre A correspond à la valeur numérique 65.
la lettre B correspond à la valeur numérique 66.
la lettre C correspond à la valeur numérique 67.
etc.
la lettre a correspond à la valeur numérique 97,
la lettre b correspond à la valeur numérique 98,
la lettre c correspond à la valeur numérique 99.
etc.
le caractère 0 correspond à la valeur numérique 48,
le caractère 1 correspond à la valeur numérique 49
(...)
le caractère 9 correspond à la valeur numérique 57 (48+9)

Les lettres sont dans l'ordre alphabétique et croissent de 1 en 1 et les nombres sont en ordre croissant et croissent de 1 en 1. Pour les ponctuations, les opérateurs arithmétiques et différents autres signes comme les parenthèses, les accolades, les crochets etc. il n'y a pas d'ordre particulier :

! correspond à la valeur 33

(correspond à la valeur 40
; correspond à la valeur 59
l'espace, qui est une ponctuation, correspond à la valeur 32
[correspond à la valeur 91
] correspond à la valeur 93
{ correspond à la valeur 123
} correspond à la valeur 125
etc.

Dans un programme lorsque l'on a besoin d'un caractère, par exemple pour un affichage ou savoir si une touche du clavier a été appuyée, le langage C permet d'obtenir directement la lettre sans avoir à passer par le nombre. Il suffit d'encadrer la lettre par deux apostrophes 'A' 'B' 'C' ... 'a' 'b' 'c' ... '0' '1' '2'... ' . ' ';' '[' '}' etc.

Mais en réalité pour la machine c'est un nombre (par exemple 'A' est équivalent à 65) et elle peut être utilisée exactement comme un nombre, par exemple :

```
int a1 = 'B', a2 = 66;      // a1 et a2 valent 66
```

c. printf() pour afficher des valeurs

La fonction printf() permet d'afficher des chaines de caractères et une chaine de caractères est un ensemble de caractères présenté entre guillemets : "ceci est une chaine de caractères", "estbngiuzqgjn \n vkjd1234567890" est une autre chaine de caractères. Dans un programme l'instruction :

```
printf("estbngiuzqgjn \n vkjd1234567890");
```

affiche dans une fenêtre console :

```
estbngiuzqgjn
  vkjd1234567890
```

Le caractère '\n' dans une chaine de caractères correspond au retour chariot et provoque un passage à la ligne.

Pour pouvoir afficher une valeur numérique il faut la convertir en chaine de caractères. En effet dans un programme la valeur 50 n'est pas du même type que la chaine de caractères "50", ce sont deux objets différents. Le premier est un nombre de type int le second une chaine de caractère, à savoir un tableau de char ce qui est autre chose et ils n'ont pas les mêmes propriétés techniques. Ainsi, pour pouvoir afficher la valeur 50 il faut qu'elle soit convertie en la chaine de caractères "50".

La fonction printf() peut traduire en chaine de caractères des valeurs numériques pour les afficher. Pour ce faire elle a besoin des valeurs numériques à traduire et d'une chaine résultat dans laquelle ces valeurs traduites vont prendre place. Cette chaine résultat utilise ce que l'on appelle des formats. Les formats sont indiqués dans la chaine par le signe % (pour cent). Il y a plusieurs formats, en général un pour chaque type de valeur à afficher.

```
%d  est le format pour afficher une valeur entière
%i  est un autre format pour afficher un entier, une variable de
type int
%ld  est le format pour afficher un long
%f   est le format pour afficher un float
%lf, %g, %e sont des formats pour afficher un double
%x  est le format pour afficher une valeur héxadécimale
%c  est le format pour afficher un caractère
%s  est le format pour afficher une chaine de caractères
```

```
%p  est le format pour afficher une adresse mémoire (en fait une
valeur héxadécimale comme pour %x)
```

Ensuite l'utilisation de la fonction printf() est simple. En premier vient la chaine de caractères avec le ou les formats et en second viennent la ou les valeurs que l'on veut afficher ou les variables dont nous voulons afficher les valeurs. Par exemple pour afficher la valeur 50 et la valeur de la variable test :

```
int test = 0;
          printf("%d",50);
    printf("%d",test);
```

Lors de l'affichage dans la fenêtre console le format %d est remplacé par la chaine de caractères qui correspond à la valeur indiquée après la chaine formatée. Dans cet appel la valeur numérique 50 est remplacée par la chaine "50" et cette chaine est intégrée à la place du %d dans la chaine à afficher. De même pour la valeur de la variable test, la chaine "0" est intégrée à la place du %d dans le second appel.

Il est possible d'ajouter du texte avec les formats de la chaine formatée :

```
printf("val1=%d, val2=%c, val3=%d, val4=%f", 66, 66, 231,30.3333);
```

Les valeurs à la fin sont converties chacune en chaine de caractères et viennent remplacer successivement, dans le même ordre de gauche à droite le %d, le %c, le %d et le %f ce qui donne dans une fenêtre console l'affichage suivant :

```
val1=66, val2=B, val3=231, val4=30.3333
```

La valeur 2 est affichée selon le format %c réservé aux caractères. Ainsi la valeur 66 est convertie dans le caractère correspondant dans la table ascii puis ce caractère est intégré dans la chaine finale.

Pour ce qui concerne l'affichage d'une valeur flottante avec la fonction printf() il est possible de spécifier le nombre de chiffres après la virgule que nous voulons voir. Ce nombre est précédé d'un point, et s'intercale entre le signe % et le format. Par exemple pour n'avoir que deux chiffres après la virgule le format est %.2f :

```
printf("test=%.2f\n",1.234567);  // affiche : 1.23
```

D'une façon générale il est possible de cette façon de spécifier la taille minimum pour l'espace d'affichage selon n'importe quel format :

```
printf("[%4d][%-4d]\n", 10,20);
```

Cet appel affiche : [10][20] et le signe – provoque un alignement à gauche.

Pour un flottant ou une chaine de caractère nous pouvons spécifier la taille minimum totale et le nombre de chiffres après la virgule ou le nombre de caractères pour une chaine :

```
printf("[%5.2f][%-5.2s]\n", 1.567,"2.896");
```

Cet appel affiche : [1.56][2.]

d. Obtenir et afficher la taille en mémoire d'une variable

Pour avoir la taille en mémoire d'un objet quelconque il y a un opérateur spécifique nommé sizeof(). Cet opérateur marche comme une fonction : il retourne la taille en octet de l'objet dont le nom est placé entre les parenthèses. Par exemple soit une déclaration de variable :

```
int ma_variable;
```

L'expression

```
sizeof(ma_variable )
```

vaut la taille mémoire en octet de l'objet à savoir 4 parce que ma_variable est un int. Si ma_variable est déclarée double ce sera 8. Qu'imprime le programme suivant ?

```
int main()
{
int i ;
float f;
double d;
char c;

   printf("taille en mémoire de i : %d\n",sizeof(i) );
   printf("taille en mémoire de f : %d\n",sizeof(f) );
   printf("taille en mémoire de d : %d\n",sizeof(d) );
   printf("taille en mémoire de c : %d\n",sizeof(c) );
   return 0;
}
```

Réponse :

```
taille en mémoire de i : 4
taille en mémoire de f : 4
taille en mémoire de d : 8
taille en mémoire de c : 1
```

e. Obtenir et afficher l'adresse mémoire d'une variable

Il existe un opérateur qui permet d'obtenir l'adresse mémoire d'une variable, c'est l'opérateur & dit "adresse de" placé à gauche de la variable dont on veut récupérer l'adresse. Soit par exemple dans un programme la déclaration :

```
int i=99;
```

l'adresse en mémoire de la variable i est donnée par l'expression

```
&i
```

Pour afficher cette adresse il reste à appeler la fonction printf avec le bon format. Deux sont possibles, le %x parce que les adresses sont des nombres hexadécimaux ou le %p qui est spécifiquement prévu pour les adresses mémoire, ce qui donne :

```
printf("l'adresse de i est %p \n", &i);
//ou
printf("l'adresse de i est %x \n", &i);
```

f. scanf() pour récupérer une valeur entrée par l'utilisateur

La fonction scanf() permet de récupérer une ou plusieurs valeurs entrées au clavier par l'utilisateur. Cette fonction utilise un système de chaine formatée avec des formats identiques à ceux de la fonction printf(). Elle fonctionne en quelque sorte à l'envers : il ne s'agit pas d'afficher du texte mais de récupérer des valeurs que l'utilisateur va entrer au clavier sous forme de texte. Ce texte, une chaine de caractères, est ensuite converti et affecté à une variable ou plusieurs variables.

Le premier paramètre de scanf() est la chaine de caractère formatée. Elle spécifie ce qui est attendu. Le second paramètre est la liste des adresses des variables dans lesquelles les résultats sont stockés. Le type de chacune de ces variables doit coïncider avec le format spécifié. Les valeurs sont stockées à l'adresse des variables, c'est à dire directement dans la zone mémoire des variables utilisées.

Par exemple, le programme suivant permet de récupérer une valeur pour un entier :

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

int main()
{
int rec;

   scanf("%d",&rec);
   return 0;
}
```

Lorsque ce programme est lancé le curseur en écriture clignote dans le coin haut gauche de la fenêtre console. Il attend qu'une valeur soit saisie par l'utilisateur. Il suffit de taper un nombre et ensuite de valider en tapant sur la touche enter. Il est possible de faire plusieurs demandes à la fois avec une succession de formats et les adresses des variables de destination :

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

int main()
{
int rec1,rec2;
float rec3;
   scanf("%d%d%f",&rec1,&rec2,&rec3);
   return 0;
}
```

L'utilisateur doit alors entrer chaque valeur séparée par un espace et taper enter à la fin.

A la différence de la fonction printf() la chaine formatée contient les formats de ce qui est attendu, mais rien d'autre sous peine d'erreurs. Seuls les espaces sont indifférents entre les formats :

```
scanf("%d%d%d", &a, &b, &c);
```

est équivalent à

```
scanf("%d %d            %d", &a, &b, &c);
```

En revanche

```
scanf("%d, %d, %d", &a, &b, &c);
```

provoque une erreur à cause des virgules.

Également, l'utilisateur doit taper des valeurs qui correspondent à ce qui est demandé sinon il y a erreur. Par exemple :

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

int main()
{
int rec1, rec2;
   printf("entrer deux valeurs entiéres :\n");
   scanf("%d%d",&rec1,&rec2);
   printf("les valeurs entrées sont : %d et %d\n",rec1,rec2);
   return 0;
}
```

Si l'utilisateur entre ici autre chose que des entiers, par exemple un float ou une ou plusieurs lettres il y aura erreur, aucune valeur ne sera placée dans la variable où l'erreur se produit et les suivantes ne seront plus prises en compte.

g. Les pièges de scanf()

Utiliser rewind(stdin) si plusieurs appels successifs

Lorsqu'il y a plusieurs appels successifs de scanf() il est nécessaire de réinitialiser le buffer d'entrée (un fichier nommé stdin où sont stockées toutes les entrées du clavier) avec la fonction rewind(). Sinon la fonction a un comportement parfois incompréhensible parce qu'elle conserve des informations des collectes précédentes.

```
int rec1, rec2;
    printf("entrer deux valeurs entiéres :\n");
    scanf("%d%d",&rec1,&rec2);
    rewind(stdin);
    printf("les valeurs entrées sont : %d et %d\n",rec1,rec2);
```

Récupération d'une lettre

Attention lors de l'utilisation de scanf() pour récupérer une lettre dans un int. scanf() stocke la valeur de la lettre sur le premier octet du int mais ne touche pas aux autres octets. Si la variable n'est pas initialisée à 0 il y aura des valeurs aléatoires qui correspondent à ce qui traîne dans la mémoire sur les trois octets restants. Il faut soit initialiser la variable à 0 ou utiliser un char. Le test suivant permet de le constater :

```
int c;
    scanf("%c",&c);     // entrer ici la lettre 'A'
    printf("erreur : %d ne vaut pas %d ", c, 'A');
           // erreur !!
           // ce n'est pas la lettre voulue
```

Ne pas oublier &, "adresse de", pour la ou les variables réceptrices

Attention également, sous peine d'erreur, de bien spécifier les adresses des variables utilisées avec l'opérateur & "adresse de " accolé à la gauche du nom de la variable. Les paramètres de cette liste sont ce que l'on appelle des pointeurs. Ils reçoivent une adresse mémoire comme valeur et permettent d'écrire directement à cette adresse mémoire. Ces points seront abordés et approfondis ultérieurement avec les pointeurs en paramètre de fonction.

5. Mise en pratique : manipulations de variables

Exercice 1
Faire un programme qui déclare trois int, deux float, un double et deux char, affecter une valeur à chaque variable, afficher le résultat.

Exercice 2
Dans un programme, déclarer un char et lui affecter une valeur choisie au hasard. Afficher la valeur entrée en utilisant les deux formats %d et %c, qu'est ce que ça donne ? Réessayer en demandant cette fois une valeur comprise entre 97 et 122.

Que remarquez-vous ? Quelles valeurs permettent d'afficher les caractères *, @, ♠, ♣, ♥, ♦. ?

Exercice 3
La fonction putchar(char c) permet d'afficher un caractère tout seul.
Dans un programme, utiliser cette fonction pour afficher le mot "TataFaitDuZele" caractère par caractère, en allant à la ligne au moins trois fois.

Exercice 4
Afficher lettre par lettre le mot "fourchette" en ajoutant ou soustrayant ce qu'il faut à chaque lettre du mot "contrainte" (utiliser opérateur arithmétique + ou -)

Exercice 5
Dans un programme crypter un mot à partir d'une clé (une valeur) entrée par l'utilisateur. Le mot crypté est affiché lettre par lettre (utiliser opérateur arithmétique + ou -).

Exercice 6
Dans un programme déclarer quatre variables de type int. Afficher pour chacune son adresse mémoire dans l'ordre où vous les avez déclarées. Entre chaque affichage intercalez la commande : system("PAUSE"); Cette commande arrête le programme et il attend que vous pressiez une touche pour continuer son exécution.
Pouvez-vous prédire à chaque arrêt quel sera l'adresse de la variable suivante ?

Exercice 7
Faire un programme qui déclare cinq variables chacune d'un type différent. Pour chacune demandez à l'utilisateur d'entrer une valeur. Ensuite afficher toutes les informations obtenues (valeur, taille, adresse) en commençant par les variables qui prennent le moins de place en mémoire.

Exercice 8
Dans un programme il y a 5 commandes possibles : commande 1, commande 2 ... commande 5. Imaginer des commandes si vous préférez. L'objectif est de faire un menu qui permette de récupérer le choix de l'utilisateur. Faire un programme qui demande à l'utilisateur la commande qu'il veut exécuter et qui ensuite affiche la commande choisie par l'utilisateur.

Exercice 9
Faire un dessin avec des lettres choisies au début du programme par l'utilisateur.

Exercice 10
Le programme à faire récupère avec deux variables v1 et v2 deux valeurs entrées par l'utilisateur. Ensuite il inverse ces valeurs (v1 prend la valeur de v2 et v2 prend la valeur de v1) et réaffiche les variables pour vérifier que les valeurs ont bien été inversées.

Exercice 11
Soit cinq variables v1, v2, v3, v4, v5 contenant chacune une valeur entrée par l'utilisateur ou choisie dans le programme. Permuter les valeurs en sorte que v1 prend la valeur de v2, v2 prend la valeur de v3, v3 prend la valeur de v4, v4 prend la valeur de v5 et v5 prend la valeur de v1.

6. Pour comprendre les variables

a. Codage et mesure de l'information

La RAM ou mémoire vive est nécessaire pour le fonctionnement d'un programme. Elle repose sur des puces mémoires ou "puces RAM". Une puce est un circuit intégré, d'environ un centimètre carré qui rassemble un grand nombre de conducteurs et composants électriques. C'est là où sont rangés les fameux "bits" en grande quantité. Le bit est un micro-interrupteur interprété comme 0 lorsqu'il est fermé et 1 lorsqu'il et ouvert. C'est la plus petite unité de mesure de l'information.

```
1 bit      peut stocker 2 informations :
           (0) ou (1) soit 2¹ informations

2 bits     peut stocker 4 informations :
           (0,0)(0,1)(1,0)(1,1) soit 2² informations

3 bits     peut stocker 8 informations :
           (0,0,0)(0,0,1)(0,1,0)(0,1,1)
           (1,0,0)(1,0,1)(1,1,0)(1,1,1) soit 2³ informations

n bits     peut stocker 2ⁿ informations
```

Si le bit est l'unité élémentaire d'information, l'octet est la plus petite unité de mémoire adressable c'est à dire qui a sa propre adresse (un char en C) . Et c'est à partir de l'octet (8 bits, 2^8 informations) que sont construites les quantités d'information :

```
1 octet    (o)    vaut    2⁸    Bits
```

En C les variables simples ne dépassent pas 8 octets (un double).

Par ailleurs la mesure des grandes quantités d'information pour les fichiers en tout genre est établie de la façon suivante :

```
1 kilo-octet (Ko)vaut   2¹⁰  octets   soit   1024 octets
1 Mega-octet (Mo)vaut   2²⁰  octets   soit   1024 Ko  et 1 048 576  o
1 Giga-octet (Go)vaut   2³⁰  octets   soit   1024 Mo
1 Tera-octet (To)vaut   2⁴⁰  octets   soit   1024 Go
```

b. Plages de valeurs en décimal

Soit dans un programme une variable codée sur un octet (8 bits). Elle permet donc de coder un maximum de 2^8 informations c'est à dire 256 valeurs binaires différentes qui vont de 00000000 à 11111111.

Le passage du binaire au décimal se fait sur le principe de la notation étendue. Pour mémoire, un nombre décimal, par exemple 8542, peut s'écrire :

```
      8542   = 8*10³ + 5*10² + 4*10¹ + 2*10⁰
             = 8*1000 + 5*100 + 4*10 + 2*1
             = 8000 + 500 + 40 + 2 .
```

De même un nombre binaire peut-être représenté en notation étendue. Par exemple la notation étendue du nombre binaire 110101 donne :

```
      110101 = 1*2⁵ + 1*2⁴ + 0*2³ + 1*2² + 0*2¹ + 1*2⁰
```

et il n'y a plus qu'à effectuer le calcul pour obtenir sa valeur en décimal :

```
      110101 = 32 + 16 + 0 + 4 + 0 + 1
             = 53
```

On constate que le nombre décimal le plus grand codé sur un octet est 255. C'est :

```
11111111   =  2⁷ *1+  2⁶ *1+  2⁵ *1+  2⁴ *1+ 2³ *1+ 2² *1+ 2¹ *1+ 2⁰  *1
           = 128  +   64  +   32 +   16 +   8 +   4  +  2  +  1
```

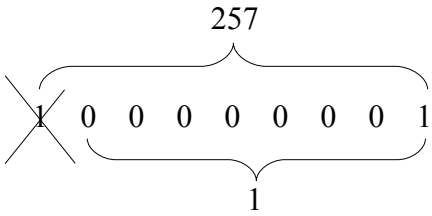
= 255

Sur un octet la plage de valeur non signée (sans signe – et +) va de 0 à 255 compris soit 256 valeurs ou 2^8 valeurs.

Le principe de cette addition est le même pour n bits, la plage de valeur non signée va alors de 0 à n -1 ce qui donne un total de 2^n cas.

c. Troncature

Si le nombre à coder dépasse la possibilité de codage qu'offre l'espace mémoire, il y a un dépassement de capacité. Par exemple un seul octet ne peut pas restituer des nombres supérieurs à 255. Pour ces nombres l'octet se comporte comme si un modulo 2^8 était effectué, et la partie qui dépasse est tronquée. Affecter la valeur 257 à un octet donne 257 modulo 256 ce qui est égal à 1 (le reste de la division par 256):



d. Codage des nombres négatifs

En machine sur chaque octet les bits sont disposés de droite à gauche de la position 0 à la position 7 comprises. Par exemple le nombre 110101 (53) de l'exemple ci-dessus donne l'octet suivant :

bit 8	**bit 7**	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**
pos 7	pos 6	pos 5	pos 4	pos 3	pos 2	pos 1	pos 0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	1	0	1	0	1
0	+ 0	+ 32	+ 16	+ 0	+ 4	+ 0	+ 1

Pour avoir des nombres signés c'est à dire avec des valeurs négatives ou positives le principe est d'avoir un bit de signe, celui le plus à gauche, le bit 8 sur un octet.

Si ce bit est à 1 le nombre est négatif sinon il est positif. Mais ce n'est pas suffisant car il y a alors deux 0 (un négatif et un positif) et les opérations d'addition ou de soustraction entre nombres négatifs et positifs sont un peu compliquées (Si x et y

ont des signes différents le signe de x+y est celui du plus grand en valeur absolue. Il faut donc commencer par connaître lequel des deux est le plus grand pour connaître le signe du résultat, puis soustraire du plus grand le plus petit).

La méthode généralement retenue est celle du complément à deux. Cette méthode permet de transformer une soustraction en une addition. Rappelons que parmi le nombre limité d'instructions simples que le microprocesseur effectue, il y a l'addition de bits. C'est une solution rapide.

Principe du complément à deux en binaire

Le complément à 2 d'un nombre binaire c'est son inverse +1, par exemple :

```
Nombre binaire          11010011
inverse                 00101100
complément à deux       00101101
```

Complément à deux dans un système de signes

Si le nombre est négatif (bit de signe à 1) ça donne le nombre négatif du complément à deux de ce nombre. Par exemple sur un octet :

```
nombre binaire          10000000    |  10000001   | 10001111
valeur non signée       128         |    129      |  143
complément à deux       10000000    |   01111111  |  01110001
traduction signée       -128        |  -127       |  -113
```

Sur un octet les valeurs positives vont de 0 à 127 et toutes les valeurs de 128 à 255 sont interprétées comme les valeurs négatives de -128 à 0

Pour n bits, la plage possible des valeurs signées couvre la fourchette :

$-(2^{n-1})$ à $+(2^{n-1}-1)$

Le problème du dépassement de capacité subsiste. Par exemple, sur un octet signé il n'est pas possible de coder les nombres inférieurs à -128 ni les nombres supérieurs à 127. Pour un octet signé additionner 120 et 9 donne -127.

7. Mise en pratique : codage des informations numériques

Exercice 1
Combien d'informations pouvez-vous coder sur 2 bits ? sur 4 ? sur 8 bits ?
Quels sont les codes binaires associés à ces informations ?
Quelles sont les valeurs décimales associées à ces codes ? En signé ? En non-signé ?
Combien de bits faut-il pour coder un alphabet (chinois) de 4344 lettres ? Combien d'octets ?

Exercice 2
Donner en binaire le codage des nombres -32, -77, 104, 258 sur 8 bits.

Exercice 3
Si dans un programme :

- j'affecte la valeur 266 à un char et que je l'affiche quel est le résultat ?
- j'affecte la valeur 384 à un char et que je l'affiche quel est le résultat ?
- j'affecte la valeur 768 à un char et que je l'affiche quel est le résultat ?
- j'affecte la valeur 23277 à un char et que je l'affiche, résultat ?
- combien donne sur un char -120+250 ?

- combien donne sur un char -120-120 ?

Testez ou trouvez vos réponses avec un programme et expliquez les résultats.

8. Expérimentation : Variables simples, déclaration, affectation, affichage, saisie

```
// des librairies de fonctions
#include <stdio.h>
#include <stdlib.h>

int main()
{
   //-------------------------------------------------------
//avoir des variables : <type> <identificateur> < ; >
//le type définit une taille et des propriétés (virgule ou pas)

// les instructions
int toto; // reserve un emplacement mémoire (une adresse) pour un
int
   toto = 10; // affecte de la valeur 10 à la variable toto,

   // remarque :
   // expression (int toto, toto=10) et instruction (int toto;
toto=10;)

   //-------------------------------------------------------
// les autres types en variables simples du C
char c = 'A';     // 1octet, codage ascii des caractères
short s = 10;     // 2 octets
int i = 20;       // 2 ou 4 octets
long l = 456;     // 4 octets
float f = 3.89;   // 4 octets
double d = 45678890.876543;// 8 octets

   //-------------------------------------------------------
   // pour afficher une valeur :
   //    chaque type a son format :
   //    char %c, short, int, long %d, float %f, double %lf,
   //    adresse mémoire (variable complexe de type pointeur) %p
   printf("i vaut : %d\n", i);

   //-------------------------------------------------------
   // codage ascii des caractères : à une valeur numérique
   // correspond un caractère
   printf("%c : %d\n", 112,112);

   //-------------------------------------------------------
   // taille en mémoire d'une variable : opérateur sizeof
   printf("double : %d, int :%d\n",sizeof(double), sizeof(i));

   //-------------------------------------------------------
   // récupérer l'adresse mémoire d'une variable : opérateur &
   printf("%p, %p, %p, %p\n", &i, &l,&f,&d);

   //-------------------------------------------------------
   // entrée utilisateur : scanf
   printf ("entrer une valeur entiere\n");
   scanf("%d",&i);
```

```
   printf("i vaut : %d\n",i);

   //-------------------------------------------------------
   // remarque : les instructions sont exécutées dans l'ordre les
   // unes à la suite des autres de façon linéaire

   //-------------------------------------------------------
   // valeur de retour pour communication avec le système
   return 0;

}
```

C. Les opérations

1. La notion d'expression

Dans le code informatique tout élément ou ensemble d'éléments qui fait l'objet d'une évaluation numérique est appelé une expression :

```
a + b,  a / b,  a = b,  &b, !b sont des expressions.
```

Dans ces expressions +, /, =, & et ! sont des opérateurs et les variables a et b sont les opérandes.

Ces combinaisons de variables avec des opérateurs sont appelées des expressions plutôt que des opérations car il n'y a pas que des opérateurs arithmétiques. De plus, une valeur constante, une variable ou un appel de fonction seuls sont également considérées comme des expressions. Ce sont des expressions élémentaires. Les expressions qui font appel à des opérateurs et plusieurs arguments sont dites expressions composées.

La valeur numérique d'une expression, composée ou élémentaire est le résultat de l'expression elle-même. S'il s'agit d'une opération arithmétique ou d'une affectation, la valeur numérique de l'expression est le résultat de cette opération. S'il s'agit d'un appel de fonction c'est la valeur de retour de la fonction. L'expression vaut cette valeur, elle est cette valeur, et à ce titre une expression peut être intégrée dans une expression plus large, dans une affectation ou encore comme opérande dans une opération. Par exemple, le programme :

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

int main()
{
int a, b=20;
   printf("%d--",a=10);      // affectation
   printf("%d--",a=a*b);     // multiplication
   printf("%d\n",a%b);       // modulo ( reste de la division)
   return 0;
}
```

imprime :

```
      10--200--0
```

2. Opérations arithmétiques

a. Les opérateurs +, -, *, /, %

Les opérateurs arithmétiques sont les suivants :

```
+   plus   :
le résultat de l'expression   a+b   est la somme

-   moins  :
le résultat de l'expression   a-b    est la soustraction

*   multiplier   :
le résultat de l'expression   a*b   est la multiplication

/   diviser :
le résultat de l'expression   a/b    est la division

% modulo :
le résultat de l'expression   a%b  est  le reste de la division de a
par b
```

Que donne le programme suivant ?

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

int main()
{
int a=10,b=20,c=0;                          // 1
   c=a+b;                                   // 2
   a=b/c;                                   // 3
   b=a*c;                                   // 4
   printf("a=%d, b=%d, c=%d\n",a,b,c);      // 5
   printf("res=%d\n",c%4);                  // 6
   printf("c=%d\n",c);                      // 7
   return 0;
}
```

```
ligne 1 : a vaut 10, b vaut 20, c vaut 0
ligne 2 : a vaut 10, b vaut 20, c vaut 30
ligne 3 : a vaut   0, b vaut 20, c vaut 30
ligne 4 : a vaut   0, b vaut   0, c vaut 30
ligne 5 : affichage :  a=0, b=0, c=30
ligne 6 : affichage : res=2
ligne 7 : affichage : c=30
```

Attention !

A la ligne 6 le résultat de l'opération est affiché mais il n'y a pas d'affectation qui modifie la valeur de c et c conserve sa valeur qui est affichée à la ligne 7.

b. Les affectations combinées

Les opérateurs suivants permettent d'associer une opération avec une affectation, l'opération est faite en premier ensuite l'affectation :

```
a += b       est une contraction de    a = a+b
a -=  b      est une contraction de    a = a-b
a *= b       est une contraction de    a = a*b
a /=  b      est une contraction de    a = a/b
a %= b       est une contraction de    a = a%b
```

Qu'imprime le programme suivant ?

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

int main()
{
int a=7,b=232,c=4;                                  // 1

          c+=a;                                     // 2
          b-=c;                                     // 3
          a%=10;                                    // 4
          printf("a=%d, b=%d, c=%d\n",a,b,c);       // 5
          c+=(b-a);                                 // 6
          printf("c=%d\n",c);                       // 7
}
```

```
ligne 1 : a vaut 7, b vaut 232, c vaut 4
ligne 2 : c vaut 11
ligne 3 : b vaut  221
ligne 4 : a vaut  7
ligne 5 : affichage :  a=7 b=221 c=11
ligne 6 : c vaut 225
ligne 7 : affichage : c=225
```

c. Post et pré incrémentations ou décrémentations

A maîtriser également les opérateurs ++ et --. Ils sont très souvent utilisés. Ils s'emploient à la gauche ou à la droite de leur opérande.

Soit par exemple un entier i :

```
      i++ et ++i   sont  équivalents à i = i+1, de même
      i-- et --i   sont équivalents à  i = i-1
```

Mais attention ! Pour ces deux opérateurs il y a toutefois une différence entre l'opérateur placé avant la variable et l'opérateur placé après la variable. Cette différence est sensible lorsque i++ ou ++i sont utilisés dans des expressions par exemple dans un appel de fonction ou une opération :

```
   printf("%d %d",++i, i++);
```

Si l'opérateur ++ est placé avant l'opérande : ++i, alors, dans une expression, la valeur de i est la valeur de i+1, c'est à dire que i est incrémenté de 1 avant d'être utilisé dans l'expression.

En revanche si ++ est placé après l'opérande : i++ alors, dans une expression, la valeur de i reste i et i est incrémenté de 1 après que l'expression ait été évaluée.

Par exemple qu'affiche la séquence suivante ?

```
int i=0;
   printf("%d",i++);   // resultat donne  0
   printf("%d",i);     // mais i est incrémenté après avoir été
                       // utilisé et vaut 1  après
   printf("%d",++i);   // i est incrémenté avant son utilisation,
                       // le résultat donne  2
```

Le principe est le même avec l'opérateur -- .S'il est placé avant la variable une soustraction de 1 est opérée avant que la variable soit utilisée dans l'expression. S'il

est placé après la variable, la variable est utilisée sans modification dans l'expression puis est décrémentée de 1 après.

d. Opérations entre type différents, opérateur de "cast"

Que se passe t-il si des opérations avec des variables de types différents sont effectuées ? Par exemple :

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

int main()
{
int entier = 10;
float flottant = 34.9;
int eRes;
float fRes;
   eRes=entier+flottant;
   fRes=entier+flottant;
   printf("eRes : %d, fRes : %f\n", eRes, fRes);        // 1

   eRes=entier / (entier+5);
   fRes=entier / (entier+5);
   printf("eRes : %d, fRes : %f\n", eRes, fRes);        // 2

   eRes=flottant / entier;
   fRes=flottant / entier;
   printf("eRes : %d, fRes : %f\n", eRes, fRes);        // 3
   return 0;
}
```

Voici la règle à connaître pour le savoir :

L'opération arithmétique s'effectue dans le type de l'opérande le plus fort
L'affectation s'effectue dans le type du membre de gauche, celui de la variable qui reçoit la valeur.

1) Le premier affichage donne eRes : 44, fRes : 44.900000.
L'opération est faite en float, le type le plus fort. Avec l'affectation, le résultat est ramené dans le type int à 44 pour eRes et reste en float avec fRes.

2) Le second affichage donne eRes : 0, fRes : 0.000000.
L'opération est faite en int le type le plus fort. Il n'y a donc pas de chiffre après la virgule et le résultat est 0 dans les deux cas.

3) Le troisième affichage donne eRes : 3, fRes : 3.490000.
L'opération est faite en float, le résultat est converti en int dans le premier cas et reste en float dans le second.

Opérateur de cast

Parfois il est nécessaire de pouvoir forcer une opération à s'effectuer dans un type particulier. Pour ce faire il existe un opérateur qui permet de convertir le type d'une expression dans un autre type. C'est l'opérateur dit "cast". Par exemple dans le cas //2 ci-dessus :

```
fRes = entier / (entier+5);
```

si l'on a besoin du résultat en float de l'opération il faut forcer cette opération à s'effectuer en float. Pour cela il suffit de spécifier entre parenthèses le type souhaité à gauche d'une des deux variables :

```
fRes = (float)entier / (entier+5);
ou
fRes = entier / (float)(entier+5);
```

Attention toutefois, ce n'est pas le résultat de l'opération qui est converti mais une des deux variables afin que l'opération s'effectue dans un type plus fort. Ca marche aussi dans l'autre sens, si l'on souhaite par exemple forcer une opération faite en float à se faire dans le type int :

```
fRes= (int)flottant / entier;
```

e. Priorités entre opérateurs

La machine exécute toujours les instructions les unes à la suite des autres et elle n'exécute jamais deux opérations en même temps. Pour pouvoir effectuer une opération composée de plusieurs calculs il y a un ordre de priorité entre les opérateurs. Par exemple l'opérateur * est prioritaire par rapport à l'opérateur + et une expression comme :

```
a + b * c
```

est interprétée comme :

```
a + (b * c)
```

Voici un tableau des priorités pour les opérateurs que nous avons abordés jusqu'ici (Le tableau complet comprenant tous les opérateurs du C est donné en annexe et nous nous y référerons dans les chapitres suivants). Chaque ligne correspond à un niveau. Nous avons quatre niveaux et ils sont classés en ordre décroissant de la priorité la plus forte à la priorité la plus faible.

opérateurs	fonction de l'opérateur	associativité
++ -- + - (type) sizeof	pré et post incrémentation pré et post décrémentation signe plus et signe moins cast taille	droite
* / %	multiplication, division, modulo	gauche
+ -	addition et soustraction	gauche
= += - = *= /= %=	affectation et affectations combinées	droite
,	évaluation séquentielle (virgule)	gauche

Mais que ce passe t-il si les opérateurs ont le même niveau de priorité ? Il y a une règle d'associativité qui détermine l'ordre d'évaluation de l'expression et des sous-expressions s'il y en a. Par exemple * et / sont associatifs à gauche, c'est à dire que le calcul est décomposé en partant de la gauche :

```
a * b / c
```

est interprété comme

```
(a * b) / c
```

Il n'est pas obligatoire de s'appuyer exclusivement sur les priorités définies avec les opérateurs et il est possible de forcer une priorité en ajoutant des parenthèses, par exemple avec :

```
( a + b ) * c
```

c'est a + b qui est effectué en premier et ensuite la multiplication par c.

Dans le doute il est bon de clarifier une expression avec les parenthèses appropriées.

Qu'imprime le programme suivant ?

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

int main()
{
int x,y,z;

          x= -3+4*5-6;
          printf("%d\n",x);

          x= 3+4%5-6;
   printf("%d\n",x);

          x=-3*4%-6/5;
          printf("%d\n",x);

          x= (7+6)%5/2;
          printf("%d\n",x);

          x=2;
          x*=3+2;
          printf("%d\n",x);

          x*=y=z=4;
          printf("%d\n",x);
}
```

Il affiche :

```
11
1
0
1
10
40
```

Pour trouver le plus simple est d'ajouter les parenthèses correspondant aux priorités, et décomposer progressivement l'expression dans l'ordre exécuté par la machine

Premier exemple :

```
x = -3+4*5-6
x = (-3)+4*5-6                - comme signe
x = (-3)+(4*5)-6              * multiplier
x = ((-3)+(4*5))-6             + addition cause associativité gauche
x = (((-3)+(4*5))-6)          - moins avant affectation
(x = (((-3)+(4*5))-6))        = affectation
```

```
ce qui donne :
(x= ((-3+20)-6))
(x= (17-6))
(x= 11)
11
```

Décomposition deuxième exemple

```
x = 3+4%5-6
x = 3+(4%5)-6
x = (3+(4%5))-6
x = ((3+(4%5))-6)
(x = ((3+(4%5))-6))
```

Décomposition troisième exemple

```
x = -3*4%-6/5
x = (-3)*4%(-6)/5
x = ((-3)*4)%(-6)/5
x = (((-3)*4)%(-6))/5
x = ((((-3)*4)%(-6))/5)
(x = ((((-3)*4)%(-6))/5))
```

Décomposition quatrième exemple

```
x = (7+6)%5/2           // le calcul entre parenthèses est effectué
                        // en premier
x = ((7+6)%5)/2
x = (((7+6)%5)/2)
(x = (((7+6)%5)/2))
```

Décomposition cinquième exemple

```
x vaut 2
x *= 3+2
x *= (3+2)
(x *= (3+2))
```

Décomposition sixième exemple

```
x vaut 10
x *= y = z = 4
x *= y = (z = 4)        associativité à partir de la droite
x *= (y = (z = 4))
(x *= (y = (z = 4)))
```

3. Mise en pratique : opérations arithmétiques, cast

Exercice 1
Soient les déclarations :

```
char c ='\x01' ; /* '\xhh' pour un nombre hexadécimal
                 (0...9, a...f ou A...F)sur un octet max*/
short p=10;
```

Quels sont le type et la valeur de chacune des expressions suivantes :

```
p + 3
c + 1
p + c
3 * p + 5 * c
6.8 /7 + 'a' * 560
```

Exercice 2
Soient les déclarations :

```
char c='\x05';
int n=5;
long p=1000;
float x=1.25;
double z=5.5;
```

Quels sont les types et les valeurs des expressions suivantes :

```
n + c + p
2 * x + c
(char) n + c
(float) z + n / 2
```

Exercice 3
Écrire un programme qui saisit un nombre entier, affiche son opposé et sa moitié exacte.

Exercice 4
Faire un convertisseur francs/euros... (un euro = 6,55957 francs)

- □ quelles sont les étapes du programme ?
- □ programmez

Exercice 5
Pour convertir des degrés Fahrenheit en degrés Celsius, on a la formule suivante :

```
C= 5/9 * (F-32)
```

Pour convertir des degrés Celsius en degrés Fahrenheit on a :

```
F= ( (9*C ) / 5)+32
```

où F est une température en degrés Farenheit et C la température correspondante en degrés Celsius

- □ Faire un programme qui convertit en degré Celsius une température entrée au clavier exprimée en degré Farenheit.
- □ Même question avec une température exprimée en degré Celsius à convertir en degré Farenheit.

Exercice 6
Ecrire un programme pour tester tous les cas de figure de la division en C afin d'illustrer le problème des cast :

- □ division d'un int par un int, rangé dans un int et affiché en %d
- □ division d'un int par un int, rangé dans un float et affiché en %f
- □ division d'un int par un float et rangé dans un float et affiché en %d et %f

Dans quel cas un cast est-il nécessaire ? tester avec cast.
Attention distinguez bien entre d'une part la division et son résultat et d'autre part l'affichage du résultat selon tel ou tel format. Ce sont deux problèmes différents.

Exercice 7
Un magasin d'informatique annonce une réduction de 10% sur les ordinateurs portables. Écrire un programme qui lit le prix d'un ordinateur entré au clavier et affiche le nouveau prix avec la réduction.

Exercice 8
Soit une fonction mathématique f définie par f(x) = (2x+3)(3x^2+2).
Ecrire le programme qui calcule l'image par f d'un nombre saisi au clavier.

Exercice 9
Ecrire un programme qui lit un nombre au clavier, affiche 1 s'il est pair et 0 s'il est impair.

Exercice 10
Écrire un programme qui affiche le nombre des dizaines, puis des centaines et des milliers d'un nombre saisie au clavier. Par exemple pour 31345 dizaine c'est le nombre 4, centaine 3 et millier 1.

Exercice 11
Ecrire un programme qui arrondi un nombre réel entré au clavier à deux chiffres uniquement après la virgule.

Exercice 12
Ecrire un programme qui lit une valeur entière entrée par l'utilisateur dans une variable i et qui affiche i, i++ et ++i. Qu'est ce que ça donne ? Pourquoi ?

Exercice 13
Quels résultats fournit le programme suivant :

```
#include <stdio.h>

int main()
{
int i, j, n;
   i=0;
   n=i++;
   printf("A : i = %d,  n = %d \n", i, n);

   i=10;
   n=++i;
   printf("B : i = %d,  n = %d \n", i, n);

   i=20;
   j=5;
   n=i++ * ++j;
   printf("C : i = %d,  j = %d, n = %d \n", i, j, n);

   i=15;
   n= i += 3;
   printf("D : i = %d,  n = %d \n", i, n);

   i = 3;
   j = 5;
   n = i *= --j;
   printf("E : i = %d,  j = %d, n = %d \n", i, j, n);
   return 0;
}
```

4. Obtenir des valeurs aléatoires

a. Principe du pseudo aléatoire

La notion de hasard pose problème en mathématiques et l'aléatoire n'existe pas pour un ordinateur. Toutefois il est possible de simuler de l'aléatoire avec un ordinateur en produisant des valeurs imprévisibles.

Le principe est de construire une suite de nombres. On prend un nombre comme point de départ, on lui applique un calcul bizarre qui produit une valeur impossible à prédire et à chaque fois on applique le même calcul sur le résultat obtenu. En fonction du point de départ et du calcul on obtient une suite imprédictible de nombres comme s'ils étaient produits par hasard. Par exemple une suite peu prévisible peut être obtenue avec le calcul suivant : $(x+3,14159)^8$

A chaque étape il y a un nombre intuitivement improbable. Évidement la suite sera toujours la même avec un nombre de départ et un calcul identiques. Le problème alors est d'avoir à volonté un point de départ qui change.

Pour résoudre ces deux problèmes, une suite et une valeur de départ, et obtenir à volonté des nombres (pseudo) aléatoires la librairie standard stdlib.h fournit deux fonctions :

- la fonction rand() s'occupe du calcule et renvoie un nombre imprédictible
- la fonction srand() initialise la valeur de départ.

b. La fonction rand()

Le calcul opéré par la fonction rand() est le suivant, à partir de la variable etape de type unsigned long (ou int en 32 bits sur 4 octets) et par défaut initialisée à 1 au départ :

```
etape = etape * 1103515245 + 12345
etape = (etape / 65536) % 32768
```

Chaque nouvel appel de la fonction accomplit une étape de plus dans la suite à partir de la valeur contenue dans la variable etape. La valeur de la variable etape est renvoyée au contexte d'appel de la fonction, c'est à dire qu'il faut récupérer cette valeur dans une variable de la façon suivante :

```
int test;

   (...)

    test=rand();        // quelque part dans le programme
```

Dans le calcul, le modulo 32768 fait que la valeur maximum possible est 32767. Cette valeur est également définie dans la librairie par la macro constante RAND_MAX (une macro constante est une valeur fixe remplacée par un texte, nous verrons ce point dans un autre module).

Le programme suivant teste les cinq premiers nombres renvoyés par la fonction rand() :

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

int main()
{
int test;

   test=rand();
```

```
    printf("etape 1 : %d\n",test);
    test=rand();
    printf("etape 2 : %d\n",test);
    test=rand();
    printf("etape 3 : %d\n",test);
    test=rand();
    printf("etape 5 : %d\n",test);
    test=rand();
    printf("etape 1 : %d\n",test);
    return 0;
}
```

A chaque nouveau lancement du programme, ce sont toujours les mêmes nombres :

```
41
18467
6334
26500
19169
```

c. La fonction srand()

Pour éviter d'avoir toujours la même suite il faut commencer la suite avec des nombres différents. C'est le rôle de la fonction srand(), initialiser le premier nombre de la suite. Pour ce faire il faut passer à srand() la valeur que nous voulons pour démarrer la suite. Par exemple pour générer une suite à partir du nombre 2 nous appelons la fonction de la façon suivante :

```
    srand(2);
```

Dans un programme avec un nombre quelconque ça donne :

```
int main()
{
int val;

    srand(689);               // initialisation du départ

    val = rand();
    printf("val=%d\n",val);   // tirage du premier nombre
    return 0;
}
```

Mais à chaque lancement le point de départ sera toujours le même : 689 dans cet exemple... Comment faire pour initialiser la valeur de départ avec une valeur différente à chaque lancement ?

Le truc c'est de prendre l'heure avec une fonction de la librairie time.h, la fonction time(). Cette fonction retourne la valeur de temps selon un codage qui varie d'un système à l'autre ou la valeur -1 si cette fonctionnalité n'est pas disponible. Sous windows l'heure est retournée en milliseconde. L'appel de la fonction est fait de la façon suivante dans le programme :

```
int depart;
    depart=time(NULL);
```

Il est nécessaire d'indiquer la valeur NULL entre les parenthèses, en paramètre de la fonction. Ensuite il reste à passer la variable départ à la fonction srand()

```
srand(depart);
```

Ce qui donne dans un programme :

```
#include <stdio.h>     // pour utiliser la fonction printf()
#include <stdlib.h>    // pour utiliser les fonctions rand() et
                       // srand()
#include <time.h>      // pour utiliser la fonction time()

int main()
{
int depart=time(NULL); // initialisation dés la déclaration de la
variable
   srand(depart);

   (...)

   printf("première valeur : %d\n",rand());
   printf("deuxieme valeur : %d\n",rand());
   printf("troisieme valeur : %d\n",rand());
   return 0;
}
```

Avec ce programme la suite est différente à chaque nouveau lancement du programme.

Remarque :

La variable depart n'est pas nécessaire, vous pouvez placer directement l'appel de la fonction time() en argument de la fonction srand() ce qui donne la formule :

```
srand( time(NULL));
```

d. Valeurs aléatoires dans des fourchettes

Obtenir des valeurs entières dans une fourchette

L'opérateur modulo donne le reste d'une division entière, a%b vaut le reste de la division de a par b, soit une valeur entière qui va de 0 à b-1, par exemple, le programme suivant permet d'afficher deux valeurs aléatoires entre 0 et 9 :

```
int main()
{
int test;
   srand(time(NULL));

   test=rand();
   test%=10;  // équivalent à test = test%10;;
   printf("test=%d\n",test);

   // il est possible d'écrire directement
   test=rand()%10;
   printf("test=%d\n",test);
   return 0;
}
```

Pour obtenir une valeur dans une fourchette mais avec deux bornes, une pour la valeur minimum "min" et une autre pour la valeur maximum "max", il suffit de tirer une valeur aléatoire dans l'interval max-min et d'ajouter min. Par exemple pour une valeur entre 20 et 49 ça donne :

```
int test;

   test= (rand()%(50-20))+20;
```

```
            // C'est à dire :
    test = 20 + rand()%30;
```

Si rand() %30 donne 0, alors 20 est bien la plus petite valeur, si rand() retourne 29, 49 est bien la plus grande valeur.

Obtenir des valeurs flottantes dans une fourchette

La fonction rand() retourne une valeur entière de type int et la plus grande valeur possible est la valeur RAND_MAX de type int, définie sous forme de macro constante dans la librairie stdlib.h. Pour obtenir une valeur flottante entre 0 et 1 il suffit de diviser la valeur de retour de rand() par la valeur maximum possible RAND_MAX, sans oublier de caster l'opération en float qui sinon s'effectue dans le type int (int divisé par int) ce qui donne :

```
float test;      // attention prendre un float !
    test= (float) rand() / RAND_MAX;
```

Pour avoir un nombre à virgule entre 0 et 50 il suffit de multiplier le nombre obtenu entre 0 et 1 par 50. La borne inférieure est 0 avec 0*50 et la borne supérieure est 50 avec 1*50. Sinon il y a toutes les valeurs intermédiaires possibles.

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

int main()
{
float test;
   srand(time(NULL));

   test=(float) rand() / RAND_MAX;   // entre 0 et 1
   printf("test1=%f\n",test);

   test=((float) rand() / RAND_MAX)*50; // entre 0 et 50
   printf("test2=%f\n",test);

   test=20+ ((float) rand() / RAND_MAX)*30; // entre 20 et 50
   printf("test3=%f\n",test);

   return 0;
}
```

5. Mise en pratique : opérations et nombres aléatoires

Exercice 1
Choisissez pile ou face dans votre tête, lancer un programme qui annonce au hasard 0 pour face et 1 pour pile... gagné ?

Exercice 2
Choisissez dans votre tête un nombre entre 2 et 12 et lancer votre programme qui simule un tirage avec deux dés à six faces... gagné ?

Exercice 3
Ecrire un programme qui génère et affiche 7 nombres aléatoires selon les contraintes suivantes :

- □ afficher un nombre aléatoire selon la plage maximum du générateur aléatoire.
- □ afficher une valeur aléatoire comprise entre 0 et 367

- afficher un nombre aléatoire compris entre 0 et une valeur "seuil haut" entrée par l'utilisateur
- afficher un nombre aléatoire compris entre 678 et 7354
- afficher un nombre aléatoire compris entre une valeur seuil bas et une valeur seuil haut entrées par l'utilisateur
- afficher un nombre aléatoire compris entre 0 et 1
- afficher un nombre aléatoire à deux décimales compris entre 0 et 50

6. Opérations bits à bits

En C nous avons six opérateurs qui permettent des opérations au niveau des bits. Ils s'utilisent uniquement avec les types entiers signés ou non : char, short, int, long.

a. ET - opérateur &

L'opérateur ET fonctionne bit à bit de la façon suivante :

```
i & 0  donne toujours 0
i & 1  donne i, c'est à dire soit 0 soit 1
```

Par exemple, le résultat de 12 & 10 donne 8 :

```
            1 1 0 0      (12 en binaire)
      &     1 0 1 0      (10 en binaire)
donne       1 0 0 0      (8 en binaire)
```

Il sert pour accéder à un ou plusieurs bits particuliers sur un entier. Pour ce faire nous prenons une valeur de masque dans laquelle seule les bits à 1 sont visibles, par exemple :

pour connaître la valeur du premier bit d'un entier, masque à 1 :

```
            10111001
      &     00000001
donne       00000001
```

Pour connaître la valeur du second bit, masque à 2 :

```
            10111001
      &     00000010
donne       00000000
```

Pour connaître la valeur du troisième bit, masque à 4 :

```
            10111001
      &     00000100
donne       00000100
```

Pour connaître la valeur du quatrième bit, masque à 8 :

```
            10111001
      &     00001000
donne       00001000
```

Pour connaître la valeur résultante des quatre premiers bits de l'octet :

```
            10111001
      &     00001111
donne       00001001
```

Pour connaître la valeur résultante des quatre derniers bits de l'octet :

```
            10111001
      &     11110000
donne       10110000
```

L'intérêt est de pouvoir disposer sur un seul entier de plusieurs interrupteurs booléens distincts.

b. OU exclusif - opérateur ^

Cet opérateur met à 1 ce qui diffère et à 0 ce qui est identique :

```
0^0    donne 0
1^0    donne 1
0^1    donne 1
1^1    donne 0
```

Il est utilisé dans différentes occasions par exemple dans le programme suivant les valeurs des deux variables sont échangées :

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

int main()
{
int a=345, b=678;

   printf("a: %d, b: %d\n,a,b);
   a^=b;
   b^=a;
   a^=b;
   printf("a: %d, b: %d\n,a,b);
   return 0;
}
```

c. OU inclusif - opérateur |

Met à 1 ce qui est à 1 :

```
0 | 0        donne 0
1 | 0        donne 1
0 | 1        donne 1
1 | 1        donne 1
```

Il permet d'accumuler des bits à 1 sur une variable. C'est utilisé notamment dans certaines fonctions pour spécifier avec une seule variable quelles traitements utiliser (les bits à 1) et quelles traitements ignorer (les bits à 0). Dans une telle implémentation chaque bit est associé symboliquement à un traitement s'il est à 1 le traitement est effectué et s'il est à 0 il n'est pas effectué.

d. COMPLEMENT - opérateur ~

Le complément est un opérateur unaire (un seul opérande à droite) qui inverse la valeur des bits d'une variable :

```
~1100  donne  0011
```

Il met à 1 ce qui est à 0 et à 0 ce qui est à 1.

e. DECALAGES gauche et droite - opérateurs >> et <<

Les décalages permettent de décaler les bits d'une variable soit vers la gauche, soit vers la droite, par exemple un décalage à droite de 2 bits :

```
10110011 >> 2       donne  00101100
```

Les deux bits à 1 de droite sont sortis et ont été remplacés par des 0 entrés à gauche. Cette opération équivaut à une division par 2^2. Pour n bits décalés à droite c'est une division par 2^n.

Dans l'autre sens un décalage à gauche de 4 bits :

```
11110011 << 4       donne  00110000
```

Les quatre bits à 1 de gauche sont sortis et ont été remplacés par des 0 entrés à droite. Cette opération équivaut à une multiplication par 2^4. Pour n bits décalés à gauche c'est une multiplication par 2^n.

f. Priorités des opérateurs bits à bits

Leur niveau de priorité est entre l'affectation et les soustraction et l'addition. Ils n'ont pas tous la même priorité(Annexe 1 Priorité et associativité des opérateurs).

7. Mise en pratique : opérations bits à bits

Exercice 1

Qu'imprime le programme suivant ?

```
int main()
{
   int x,y,y;

   x = 3;
   y = 2;
   z = 1
   printf("%d\n", x | y & z);
   printf("%d\n", x | y & ~z);
   printf("%d\n", x ^ y & ~z);
   printf("%d\n", x & y & z);

   x = 1;
   y = -1;
   printf("%d\n", ~x | x);
   printf("%d\n", x & ~x);
   printf("%d\n", x ^ x);
   x <<= 3;
   y <<= 3;
   y >>= 3;
}
```

8. Expérimentation : Opérations arithmétiques, valeurs aléatoires

```
/*
   2. OPERATIONS ARITHMETIQUES
*/
// des librairies de fonctions
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
```

```
int a=90, b=10;

   // les opérateurs arithmétiques sont +, -, *, /, %

   //-------------------------------------------------------
   // a+b est une expression
   // la valeur numérique d'une expression est le résultat
   // de l'opération (arithmétique ou non)
   printf("%d\n",a+b);

   //-------------------------------------------------------
   // priorité des opérateurs arithmétiques :
   printf("%d\n",a/2+b*8%3); // (a/2)+((b*8)%3)

   //-------------------------------------------------------
   // type d'une expression arithmétique :
   // si les opérandes sont de types différents, l'expression
   // est du type de l'opérande le plus fort
char c=12, res2;
float f=4.5, res1;
   printf("%f\n",c+f);

   //-------------------------------------------------------
   // récupérer un résultat : l'affectation est faite dans le type
   // de la variable qui reçoit,  troncature éventuelle
   res1=c+f;
   printf("%f\n",res1);

   res2=c+f;
   printf("%d\n",res2);

   //-------------------------------------------------------
   // L'opération est faite dans le type le plus fort :
int i1=10, i2=5;

   i1=i2/i1;
   printf("res i1 : %d\n",i1);

float res;
   i1=10;  // (attention div par 0)
   res=i2/i1;
   printf("res float : %f\n",res);

   //-------------------------------------------------------
   // forcer une opération à se faire dans un autre type que celui
   // des opérandes : opérateur de cast
   res= (float)i2/i1;
   printf("res float : %f\n",res);

   //-------------------------------------------------------
   //possibilité de combiner affectation et opération arithmétique
   // +=, -=, *=, /=, %=

   res+=10; // équivalent à res= res+10
   printf("res float : %f\n",res);

   //-------------------------------------------------------
   // opérateurs d'incrémentation/décrémentation
int i=0;
   printf("%d\n",i++); // post incrémentation
```

```
   printf("%d\n",++i); // pré incrémentation

   //-------------------------------------------------------
   // obtenir une valeur aléatoire entière : fonction rand()
   i=rand();
   printf("%d\n",i);

   // contraindre dans une fourchette
   i=rand()%10;  // entre 0 et 9 compris
   printf("%d\n",i);

   i=5+rand()%10;// entre 5 et 14

   //-------------------------------------------------------
   // initialiser la suite des nombre aléatoires :
   srand(time(NULL));

   //-------------------------------------------------------
   // obtenir une valeur aléatoire entre 0 et 1
   res= (float)rand()/RAND_MAX;
   printf("%f\n",res);

   // obtenir une valeur aléatoire en réelle dans une fourchette
   res= ((float)rand()/RAND_MAX)*10; // entre 0 et 10 compris
   printf("%f\n",res);

   // fin prog
   return 0;

}
```