Mise à niveau en C

Laurence Pierre

*Ce document se présente sous la forme de rappels de **cours** suivis d'**exercices** illustratifs*

*Il est nécessaire de lire les **rappels de cours** attentivement*
*Et les **exercices** associés seront faits au moins jusqu'à l'exercice 16*
(faire en priorité les exercices non optionnels)

Avertissement. *Le langage C est normalisé par l'ANSI (American National Standards Institute). Il existe plusieurs versions du standard, notamment C89 (souvent appelé le C ANSI, et reconnu par l'option `-ansi` du compilateur gcc) et C99 (reconnu par l'option `-std=c99` du compilateur gcc). C99 contient diverses caractéristiques nouvelles (voir par ex. `https://en.wikipedia.org/wiki/C99`). C89 sert encore couramment de référence, il peut être prudent de le respecter. C'est essentiellement lui que nous verrons ici.*

I. PREAMBULE

I.1 Rappels sur quelques bases

✓ Dans la plupart des langages de programmation, les *variables* doivent être *déclarées* préalablement à leur utilisation, et doivent être *typées*. C'est le cas dans le langage C.

Une ***déclaration de variable*** se fait de la façon suivante :

```
type-de-la-variable nom-de-la-variable ;
```

Exemples :

```
int nb;
float y,z;
```

Si on fait précéder la déclaration du *mot-clé* ***const***, on déclare en fait une *constante*, dont on doit fixer la valeur, et celle-ci ne doit pas changer en cours d'exécution du programme.

Exemple :

```
const float pi = 3.14;
```

L'instruction d'*affectation* de variable est notée = en C (attention à ne pas confondre avec l'opérateur de comparaison ! voir plus loin…)

Exemple :

```
x = 3.5;  // notons que toute instruction C se termine par ;
```

✓ Voyons ci-dessous un **premier programme** élémentaire. Il ne comporte que la fonction "principale", qui doit être nommée *main*.
Dans cet exemple, nous allons utiliser 3 **variables** locales à cette fonction *main* :

- une variable `premiernb` pour stocker le premier nombre saisi
- une variable `deuxiemenb` pour stocker le deuxième nombre saisi
- une variable `resultat` pour stocker le résultat de la multiplication


```c
#include <stdio.h>

int main() {                    // notre premier programme C
  // Declarations des variables
  float premiernb, deuxiemenb, resultat;
  // Saisies au clavier
  printf("Saisir le premier nombre\n");
  scanf("%f", &premiernb);
  printf("Saisir le deuxieme nombre\n");
  scanf("%f", &deuxiemenb);
  // Calcul
  resultat = premiernb * deuxiemenb;
  // Affichage du resultat
  printf("Le resultat de la multiplication est : %f\n", resultat);
  return 0;
}
```

NOTE. Cet exemple permet aussi d'introduire des **fonctions pour les entrées/sorties** : fonctions pour la *saisie au clavier* (scanf) et l'*affichage à l'écran* (printf).

La fonction ***printf*** permet l'écriture de texte à l'écran, avec format. Sa syntaxe est la suivante :
printf(format, exp_1, exp_2, … exp_n)

où

- *format* (chaîne de caractères, entre "…") contient des caractères à afficher tels quels, et des codes de format (i.e. un % suivi d'un code de conversion qui indique le type de l'information à afficher),
- les exp_i sont des expressions qui seront affichées à la place des codes de format.

Parmi les principaux codes de conversion, le *d* permet l'affichage d'une valeur entière, et le *f* permet l'affichage d'une valeur réelle.

La fonction ***scanf*** permet la lecture au clavier, avec format. Sa syntaxe est la suivante :
scanf(format, ad_1, ad_2, … ad_n)

où

- *format* (chaîne de caractères, entre "…") indique la manière de convertir les caractères qui vont être lus (les codes de format sont ici aussi représentés par un % suivi d'un code de conversion),
- les ad_i sont les adresses (emplacement mémoire) des variables qui vont recevoir les données lues (on a recours à l'opérateur &).

Les codes de conversion sont similaires à ceux utilisés avec *printf*, notamment le *d* pour un entier, et le *f* pour un réel.

I.2 Quelques mots sur la compilation

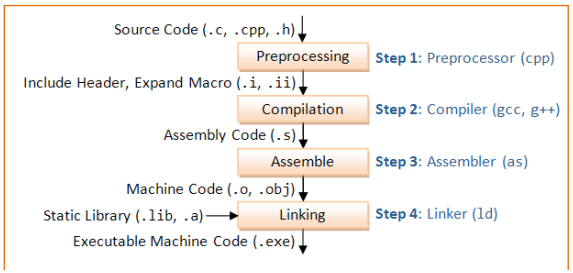
✓ Une fois écrit, votre programme tel que dans l'exemple ci-dessus doit pouvoir être *exécuté*. Il est écrit dans un langage "de haut niveau" qui doit être traduit en code machine (le programme *exécutable*). Cette opération s'appelle la *compilation*, un compilateur C couramment utilisé est **`gcc`** (https://gcc.gnu.org/).

✓ Procédure :

- le code source du programme est placé dans un fichier, le nom de ce fichier aura l'extension `.c` s'il s'agit d'un programme C (par exemple, plaçons le programme de multiplication ci-dessus dans un fichier de nom `mult.c`),
- grâce au *compilateur* associé au langage utilisé, le programme est compilé et transformé en code machine. Un autre fichier est alors produit, qui contient ce *code exécutable*.

✓ En fait, la "compilation" C correspond en réalité à plusieurs phases, comme représenté par le schéma ci-dessous :

- **Preprocessing** (*cpp*)
- **Compilation** en code assembleur, puis traduction en code machine
- **Edition de liens** (*ld*)



https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

Avant la compilation à proprement parler, le code source est prétraité par le ***préprocesseur C***.
Il traite les *directives* commençant par un # (qui peuvent se trouver en début de fichier souvent) : il procède à des *transformations syntaxiques dans le code source*, avant de transmettre ce code "préprocessé" au compilateur lui-même.
Le programme donné en exemple contient une telle directive :

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

Les directives `#include` sont des *directives d'inclusion*. Elles indiquent qu'il faut **inclure** le contenu du fichier dont le nom suit (ce nom est placé la plupart du temps entre `<...>`, ce qui indique que le fichier est placé dans un répertoire d'installation standard).
Le préprocesseur doit ici inclure le fichier `stdio.h`. Ce fichier est nécessaire ici car il contient des **déclarations** relatives aux fonctionnalités d'entrées/sorties (`printf` et `scanf` utilisées dans le programme).
Nous revenons sur les directives dans la prochaine section.

La ***compilation*** produira un ou des **fichiers objets** (plusieurs fichiers seront produits si le code source est organisé modulairement, voir plus tard). Ces fichiers contiennent des **définitions** des fonctions.
Durant la phase finale d'***édition de liens***, les fichiers objets sont combinés pour créer le **code exécutable** (pour faire simple, disons que cette étape établit les liens entre les appels de fonctions et les définitions de ces fonctions, le tout étant potentiellement dans des fichiers différents).
C'est à cette étape finale d'éditions de liens qu'il pourra de plus être nécessaire d'indiquer des *bibliothèques* dans lesquelles récupérer d'autres définitions de fonctions utilisées dans le programme (fonctions mathématiques, graphiques,...). Nous verrons plus tard les options relatives à l'utilisation de bibliothèques.

NB. Pour la suite, nous vous conseillons de **compiler avec l'option `-Wall`** de gcc (qui active le niveau le plus élevé de warnings - corriger tous les warnings avant exécution !), voire également avec l'option `-ansi` si vous souhaitez être sûrs de vous conformer à la norme C89.

II. Structure d'un programme C

II.1 Généralités - Fonction `main`

✓ Un programme C peut être réparti dans plusieurs fichiers, pour l'instant nous n'en utiliserons qu'un. Chaque fichier peut contenir :

- des **directives** pour le préprocesseur,
- des constructions de **types**,
- des **déclarations** de variables globales, et de fonctions (externes),
- des définitions de **fonctions**.

Parmi les *fonctions*, il doit en exister une dont le nom est ***main***, c'est par elle que l'exécution commencera.

```
#include <stdio.h>
... et/ou autres directives...

int main() {
   ...
   return 0;
}
```

La fonction `main` renvoie un résultat de type `int` (code de retour du processus, détails plus tard), qui vaudra 0 si tout s'est bien passé.

II.2 Directives pour le préprocesseur

✓ Le ***préprocesseur*** transforme le texte source avant la *compilation*. Les directives destinées au préprocesseur commencent par un # en début de ligne.
Nous avons déjà parlé de la directive `#include` : elle permet d'inclure dans le texte source, avant compilation, le texte figurant dans d'autres fichiers, typiquement des fichiers *en-têtes* (donnant principalement des déclarations de fonctions), c'est à dire des fichiers de suffixe `.h`

✓ La directive `#define` permet de *définir des symboles*, ou "macros" (nous verrons beaucoup plus loin qu'une telle macro peut être utilisée par exemple pour les dimensions d'un tableau).
Le préprocesseur **remplace** textuellement, partout où il apparaît (sauf dans les commentaires et chaînes littérales), **le symbole par sa valeur**, placée à côté de lui dans le `#define`.

Exemple (**attention**, incomplet - voir exercice 1) :

```
#include <stdio.h>

#define PUISSANCE 3

int main() {
  float nb, resultat;
  // Saisie au clavier
  printf("Saisir le nombre\n");
  scanf("%f", &nb);

  // Calcul
  resultat = pow(nb, PUISSANCE);

  // Affichage du resultat
  printf("Le res de l'elevation a la puissance est : %f\n", resultat);
  return 0;
}
```

Exercice 1.

1. Saisir le programme ci-dessus dans un fichier `exercice1.c` puis exécuter la commande

```
gcc -Wall -c exercice1.c
```

Vous obtenez un warning, mais on rappelle qu'il ne faut laisser passer aucun warning, car certains d'entre eux peuvent en réalité conduire à de véritables erreurs à l'exécution.
Le warning obtenu est-il produit par le préprocesseur, le compilateur, ou l'éditeur de liens ? que signifie-t-il ? que devons-nous modifier pour le supprimer ?

2. Maintenant que le problème a été réglé, exécuter la commande

```
gcc -E exercice1.c -o exercice1.i
```

qui a pour effet de ne pas appeler le compilateur mais seulement le préprocesseur, et de placer le résultat dans le fichier `exercice1.i` (voir section "Options Controlling the Kind of Output" ici `http://linux.die.net/man/1/gcc`)

Examiner le contenu du fichier `exercice1.i` :

- les directives `#include` et `#define` sont-elles toujours présentes ? Pourquoi ?
- pourquoi y a-t-il des lignes avant le main ?
- qu'y a-t-il à la place de la ligne `resultat = pow(nb, PUISSANCE);` ? Pourquoi ?
- c'est ce code "préprocessé" qui est fourni au compilateur. En conséquence, le compilateur aura-t-il connaissance de l'existence originelle des directives `#include` et `#define` ?

3. Nous tentons maintenant de générer l'exécutable : exécuter la commande

```
gcc -Wall exercice1.c -o exercice1
```

Vous obtenez une erreur de compilation, cette erreur est-elle produite par le préprocesseur, le compilateur, ou l'éditeur de liens ? que signifie-t-elle ?

Note. Pour l'instant vous ne savez pas encore comment gérer ce problème. Nous verrons plus tard comment avoir recours à des bibliothèques de fonctions. Pour l'instant notons seulement que, la fonction incriminée étant dans la bibliothèque mathématique, il suffira de l'indiquer comme suit dans la commande de compilation :

```
gcc -Wall exercice1.c -o exercice1 -lm
```

II.3 Types de données fondamentaux

✓ Les nombres ***entiers*** : on rencontre 3 types d'entiers (attention, la taille requise pour leur représentation peut dépendre de la machine utilisée)

- *int*
- *short* (ou *short int*)
- *long* (ou *long int*)

Notons qu'on peut préciser qu'on considère un *unsigned int*, un *unsigned short* ou un *unsigned long* (l'entier est alors non signé, i.e. il n'y a pas de bit de signe).

✓ Les ***réels*** : il y a aussi 3 types de nombres flottants

- *float* (simple précision)
- *double* (double précision)
- *long double* (précision étendue)

✓ Les ***caractères*** et *chaînes de caractères* : les caractères ASCII (imprimables ou non) correspondent au type *char*, représenté sur 8 bits (voir https://man7.org/linux/man-pages/man7/ascii.7.html)

Remarques :

- C ne fait pas de différence entre un caractère et *l'entier* correspondant à son code ASCII.
  Exemple : char c;
  …
  c = 'w' + 1;
- le caractère de fin de fichier (voir plus loin) correspond à la constante entière *EOF*, définie dans `stdio.h`

Une *chaîne de caractères* est en fait un ***tableau de caractères*** **se terminant par le caractère spécial '\0'.**
Une *constante de type caractère* est notée entre quotes, et une *constante de type chaîne de caractères* est notée entre guillemets

Exemples : 'a', '4', "a", "bonjour"

✓ NB. C99 propose d'autres types de base, en particulier nous pouvons trouver dans `stdint.h` des types d'*entiers dont les tailles sont fixées*, comme :

- *uint8_t* : entier non signé, sur 8 bits
- *uint32_t* : entier non signé, sur 32 bits
- *uint64_t* : entier non signé, sur 64 bits

III. LES OPERATEURS

III.1 Opérateurs arithmétiques

✓ *Opérateurs binaires* :

\+
\-
/ *le quotient de 2 entiers est un entier, le quotient de 2 réels est un réel*
% *reste de la division entière entre 2 entiers*

Opérateur unaire : -

✓ Remarque : les fonctions exponentiel, logarithme, puissance, etc… ne sont pas des opérateurs, mais des fonctions de la bibliothèque mathématique (inclure `math.h` et compiler avec l'option `-lm` pour l'éditeur de liens).

III.2 Opérateurs relationnels et logiques

✓ Opérateurs *relationnels* (comparaisons) :

== et !=
<
<=
>
>=

La valeur rendue par la comparaison de deux expressions à l'aide d'un opérateur relationnel est :

- 0 si le résultat est faux
- 1 si le résultat est vrai

NB. les booléens (type à deux valeurs de vérité *true* et *false*) n'existent pas en C. De façon générale, l'expression d'une condition satisfaite ou non se fera toujours avec des entiers : 0 est assimilé à "faux" (condition non satisfaite), et tout entier différent de 0 est assimilé à "vrai" (condition satisfaite). C'est l'interprétation faite par les instructions conditionnelles et itératives

(voir section IV).
Noter que même C99 ne supporte pas les booléens (il propose juste, artificiellement, la définition de 2 macros `true` et `false` qui sont en fait 1 et 0). C++, lui, différencie les booléens des entiers.

✓ Opérateurs *logiques* : les trois opérateurs logiques sont &&, || et !
- le résultat de *exp1 && exp2* est 1 si *exp1* et *exp2* valent 1, et 0 sinon
- le résultat de *exp1 || exp2* est 1 si au moins l'une des deux expressions vaut 1, et 0 sinon
- le résultat de *!exp* est 1 si *exp* vaut 0, et 0 sinon.

Remarques :
- en fait, ces opérateurs acceptent n'importe quels opérandes numériques
- *!exp* est équivalent à *exp==0*
- *exp* est équivalent à *exp!=0*
- *!(exp1==exp2)* est équivalent à *exp1!=exp2*

Les opérateurs && et || sont implémentés de telle façon que leur second opérande n'est évalué que si c'est indispensable (évaluation "paresseuse").

III.3 Opérateur d'affectation et dérivés

✓ L'opérateur d'affectation est le =, il réalise l'affectation et l'expression correspondante a pour valeur la valeur de la variable après affectation

✓ Incrémentation ++ et décrémentation --
Ces opérateurs sur entiers permettent de réaliser l'incrémentation (+1) ou la décrémentation (-1) de l'entier.

Exemples : i++; pourra s'écrire au lieu de i = i + 1;
j--; pourra s'écrire au lieu de j = j - 1;

Nous illustrons ci-dessus la version postfixée de ces opérateurs (la valeur est d'abord utilisée dans l'expression, puis incrémentée/décrémentée), une version préfixée existe également (la valeur est d'abord incrémentée/décrémentée, puis utilisée dans l'expression).

✓ Autres opérateurs d'affectation élargie :
+= -= *= /= %= |= ^= &= <<= >>=

La signification de *var op= exp* est *var = var op exp*

Exemples : *a -= b* est équivalent à *a = a - b*
*x *= 5+y* est équivalent à *x = x * (5+y)*

III.4 Taille et conversion de type

✓ L'opérateur *sizeof* permet de connaître la *taille* d'un type ou d'une expression, en octets.

Exemples : *sizeof(char)* vaut toujours 1, *sizeof(int)* vaudra généralement 4

✓ L'opérateur de *conversion de type* (ou cast) permet de forcer la conversion d'une expression dans le type voulu. Le nom du type est placé avant l'expression, entre parenthèses.

Exemples : *(int) x*
*(double) (a*b)*

III.5 Priorité des opérateurs

✓ Le langage C possède quelques autres opérateurs que nous ne présenterons pas pour l'instant, notamment les opérateurs bit à bit et opérateurs de décalage : & (et), | (ou), ^ (ou exclusif), ~ (complément à 1), >> (décalage à droite), << (décalage à gauche)

La table ci-dessous donne la priorité ainsi que l'associativité de tous les opérateurs :

Priorité	Opérateur(s)	Associativité
15	() [] . ->	GD
14	! ~ ++ -- - * & sizeof cast	DG
13	* / %	GD
12	+ -	GD
11	<< >>	GD
10	< <= > >=	GD
9	== !=	GD
8	&	GD
7	^	GD
6	|	GD
5	&&	GD
4	||	GD
3	?:	GD
2	= *= /= %= += -= <<= >>= &= ^= |=	DG
1	,	GD

IV. LES INSTRUCTIONS

IV.1 Notions de fonctions, instructions, blocs

✓ A part d'éventuelles constructions de types et déclarations de variables globales, un programme C est essentiellement constitué de *fonctions*. Parmi celles-ci, nous avons vu que la fonction `main` est indispensable.

✓ Chaque fonction peut elle-même contenir des déclarations locales, mais elle contient surtout une suite d'*instructions*

✓ Un ***bloc*** est une suite de déclarations et d'instructions placées entre `{…}`. Les blocs nous seront utiles en particulier dans les instructions conditionnelles et itératives (voir plus loin).
Il peut contenir ses propres déclarations de variables locales au bloc. Il peut figurer partout où une instruction simple peut figurer.
Attention, une instruction simple se termine par un ";" mais il n'y a pas de ";" après l'accolade fermante d'un bloc !

IV.2 Instructions conditionnelles

✓ L'instruction *if..else* a la syntaxe suivante :

if (expr) $inst_1$ *else* $inst_2$

où $inst_1$ et $inst_2$ sont soit des instructions simples soit des blocs. La partie *else* est optionnelle.
Si $inst_1$ n'est formé que d'une seule instruction, celle-ci n'a pas besoin d'être placée entre accolades (bloc), mais s'il y a plusieurs instructions, elle doivent obligatoirement être placées

entre accolades. Même chose pour $inst_2$.
Le fonctionnement de cette instruction est le suivant : si la valeur de expr est différente de 0 alors $inst_1$ est exécutée, sinon $inst_2$ est exécutée.
Chacune des $inst_1$ et $inst_2$ peut contenir *n'importe quelle instruction*. En particulier, elle peut à son tour contenir une instruction conditionnelle *if*, on parle de "if imbriqués".

Exemples :

```
if (a>2) {
   x = y;
   a = b;
}
else x = a;

if (c>=32 && c<=127)
   if (c>='0' && c<='9')
      printf("le caractère est un chiffre\n");
   else printf("le caractère n'est pas un chiffre\n");

if (c>=32 && c<=127) {
    if (c>='0' && c<='9')
       printf("le caractère est un chiffre\n");
}
else printf("le caractère n'est pas imprimable\n");
```

✓ L'instruction *switch* permet de considérer un ensemble de choix, elle a la syntaxe suivante :

switch (expr) {
case exp-constante$_1$: suite-inst$_1$
case exp-constante$_2$: suite-inst$_2$
…
case exp-constante$_k$: suite-inst$_k$
default : suite-inst$_{k+1}$
}

où les suite-inst$_i$ peuvent être vides, et l'énoncé *default* est optionnel.
Le fonctionnement de cette instruction est le suivant : s'il existe une exp-constante$_i$ qui égale la valeur de expr, alors suite-inst$_i$ est exécutée, ainsi que toutes les instructions qui suivent. Sinon, si l'énoncé *default* est présent, suite-inst$_{k+1}$ est exécutée, et sinon aucune instruction n'est exécutée.
Dans le cas d'un branchement à suite-inst$_i$, si on ne souhaite pas que toutes les instructions qui suivent soient exécutées, on utilise l'instruction *break*.

Exemple :

```
int x;
switch (x) {
  case 0 : printf("x est nul\n");
           break;
  case 1 :
  case 2 : y++;
  case 3 : y++;
           printf("x est <= 3 \n");
           break;
  default : printf("x est >= 4\n");
}
```


IV.3 Instructions itératives (boucles)

✓ Les instructions *while* et *do..while* se présentent comme suit :

while (expr) inst
do inst *while* (expr);

Le fonctionnement de l'instruction *while* est le suivant : l'instruction simple ou le bloc inst est exécuté répétitivement tant que la valeur de l'expression expr est différente de 0. L'exécution de inst doit modifier la valeur de expr.
Le fonctionnement de l'instruction *do…while* est similaire, mais la valeur de expr est testée en fin de boucle. La différence essentielle entre les deux instructions est donc que le *do…while* exécute au moins une fois la partie inst.
Si inst n'est formé que d'une seule instruction, celle-ci n'a pas besoin d'être placée entre accolades (bloc), s'il y a plusieurs instructions, elle doivent obligatoirement être placées entre accolades.

Exemples :

```
x = 0;
i = 0;
while (i<k) {
   x = x+y;
   i++;
}

x = 0;
i = 0;
do {
   x = x+y;
   i++;
}
while (i<k);
```

✓ L'instruction *for* a la syntaxe suivante :

for (expr$_1$; expr$_2$; expr$_3$) inst

où expr$_1$ effectue des initialisations (avant l'entrée dans la boucle), expr$_2$ constitue le test de continuation de la boucle, et expr$_3$ est une expression évaluée à la fin de chaque itération.
Comme ci-dessus, si inst n'est formé que d'une seule instruction, celle-ci n'a pas besoin d'être placée entre accolades (bloc), s'il y a plusieurs instructions, elle doivent obligatoirement être placées entre accolades.

En fait cette instruction se comporte de la même façon que la suite d'instructions ci-dessous :

expr$_1$;
while (expr$_2$) {
inst;
expr$_3$;
}

Exemple :

```
res = 0;
for (i = 1; i<=n; i++) {
   res = res + i;
   printf("Resultat partiel : %d\n", res);
}
printf("Resultat final : %d\n", res);
```

NB. La variable de boucle `i` doit être déclarée dans le bloc englobant. Toutefois C99 autorise la syntaxe suivante pour cette variable de boucle :

```
for (int i = 1; i<=n; i++) {
   res = res + i;
   printf("Resultat partiel : %d\n", res);
}
printf("Resultat final : %d\n", res);
```

c'est à dire une déclaration localisée à l'instruction `for`. Attention, `i` est alors inconnue hors de cette instruction `for`.

IV.4 Instructions *break* et *continue*

✓ Employée dans une instruction *for*, *while*, *do…while* ou *switch*, l'instruction *break* provoque la sortie de cette instruction et le passage à l'instruction située immédiatement après.

✓ Employée dans une instruction *for*, *while*, ou *do…while*, l'instruction *continue* provoque l'abandon de l'itération courante, et le passage à l'itération suivante si le test de continuation est toujours vrai.

Exercice 2. *** Dans les exercices 2 à 6, il n'est **pas demandé** d'écrire des fonctions ***

1. Ecrire un programme C qui saisit au clavier un entier positif x et qui calcule puis affiche la somme des entiers de 1 à x, i.e. 1 + 2 + 3 +… + (x-1) + x

- au moyen d'une boucle for
- au moyen d'une boucle while

On pourra compiler par :

```
gcc -Wall exercice2.c -o exercice2
```

et on aura par exemple à l'exécution :

```
$./exercice2
Saisir la borne pour le calcul de la somme :  12
La somme des entiers de 1 a 12 est 78
```

2. Adapter ce programme pour qu'il saisisse (dans l'ordre) deux entiers positifs x et y au clavier, et calcule et affiche la somme x + (x+1) + … + (y-1) + y
On voit ici clairement que x doit être inférieur à y, réaliser deux programmes utilisant deux façons différentes de s'en assurer :

- en échangeant les valeurs saisies si l'utilisateur n'a pas respecté la consigne (i.e. donner d'abord le plus petit puis le plus grand)
- en obligeant l'utilisateur à re-saisir tant qu'il se trompe.

Dans ce deuxième cas, on aura par exemple à l'exécution :

```
Saisir la borne inf puis la borne sup pour votre somme :  12  3
Attention saisir la borne inf puis la borne sup :  12  3
Attention saisir la borne inf puis la borne sup :  3  12
La somme des entiers de 3 a 12 est 75
```

Exercice 3. Ecrire un programme C qui saisit au clavier un entier positif x et qui calcule itérativement puis affiche x! (la factorielle de x).

Exercice 4.

1. Ecrire un programme C qui saisit au clavier 10 entiers positifs et calcule et affiche le nombre d'entiers pairs saisis.

2. Adapter ce programme pour qu'il fasse ce même travail sur un nombre quelconque d'entiers

saisis au clavier, tant que l'utilisateur annonce vouloir continuer.

Dans ce deuxième cas, on aura par exemple à l'exécution :

```
Saisir un entier positif :  3
Encore un ?  1
Saisir un entier positif :  8
Encore un ?  1
Saisir un entier positif :  14
Encore un ?  1
Saisir un entier positif :  5
Encore un ?  0
Vous avez saisi 2 entiers pairs
```

Exercice 5. Ecrire un programme C qui saisit au clavier deux entiers positifs i et j et affiche tous les multiples de 3 compris entre i et j. Attention, il faudra veiller à optimiser le raisonnement, en particulier votre boucle ne devra pas contenir d'instruction conditionnelle.

On aura par exemple à l'exécution :

```
Saisir la borne inf puis la borne sup pour la recherche :  4  40
Voici les multiples de 3 :
6  9  12  15  18  21  24  27  30  33  36  39
```

Exercice 6. Ecrire un programme C qui saisit au clavier 10 entiers positifs et calcule et affiche le plus grand de tous ces éléments. Puis reprendre en calculant le plus petit.
On pourra également faire une variante de ce programme comme dans l'exercice 4 (*optionnel*).
On pourra également faire une variante en ajoutant de plus une obligation pour l'utilisateur de re-saisir chaque entier s'il n'est pas positif, cf Exercice 2 (*optionnel*).

V. LES FONCTIONS

V.1 Définitions de fonctions

✓ La syntaxe pour procéder à la *définition* d'une fonction est :

type-res nom-fonction (type-par$_1$ par$_1$, …, type-par$_n$ par$_n$) {
déclarations-var-locales;
instructions;
}

où *type-res* est le type du résultat de la fonction,
par$_1$, *par$_2$*, …, *par$_n$* sont les *paramètres formels* (les valeurs passées à l'appel de la fonction sont dites *paramètres effectifs*).

Remarques :

- si la fonction ne doit pas fournir de résultat, on lui donnera le type *void*,
- attention, si *type-res* est absent, la fonction est supposée être de type *int*,

✓ L'instruction `return` permet l'abandon de la fonction en cours d'exécution et le retour à la fonction appelante, en ramenant éventuellement une valeur de retour (dont le type doit être compatible avec le type de la fonction).

Exemple :

```
int sommejusqua(int n) {
  int i, res;
  res = 0;
  for (i = 1; i<=n; i++)
    res = res + i;
```

```
  return res;
}
```

V.2 Utilisation des fonctions

✓ L'*appel d'une fonction* se fait en écrivant son nom suivi des paramètres effectifs, entre parenthèses, dans le bon ordre.

Exemple :

```
int main() {
  int somme;

  // calcul, par appel de la fonction
  somme = sommejusqua(10);

  // affichage du resultat
  printf("Resultat de la somme : %d\n", somme);
  return 0;
}
```

NB. Notons qu'en C le passage de paramètres se fait toujours ***par valeur***.

V.3 Visibilité et durée de vie des variables

✓ Remarques préliminaires : Les fonctions C ne peuvent pas être imbriquées les unes dans les autres. D'autre part, tout "bloc" peut comporter ses propres définitions de variables locales.

✓ Une *variable globale* (extérieure à tout bloc) peut être utilisée n'importe où entre sa déclaration et la fin du fichier dans laquelle elle est déclarée.
Les variables globales existent pendant toute la durée de l'exécution du programme, et l'espace mémoire correspondant est alloué lors de l'activation du programme.
Si la valeur de la variable n'est pas initialisée lors de sa déclaration (obligatoirement par une expression constante), l'espace mémoire correspondant est rempli de zéros.

✓ Une variable locale (déclarée au début d'un bloc) n'est visible *que* dans ce bloc.
Une variable locale masque toute variable globale de même nom ou toute variable locale de même nom déclarée dans un bloc englobant.
L'allocation mémoire pour une variable locale n'est valable que pendant l'exécution de son bloc.
Attention, si la valeur de la variable n'est pas initialisée lors de sa déclaration, la valeur prise est imprévisible.

Exercice 7.

1. Reprendre les exercices 3 et 2 en faisant réaliser les calculs de factorielle et de somme par des fonctions (la fonction renvoie le résultat du calcul, qui est affiché dans le main).

2. *Optionnel* - Reprendre ces mêmes fonctions et en faire des versions récursives (bien entendu, le main ne doit pas changer).

Attention : bien réfléchir aux entêtes des fonctions avant tout.

Exercice 8.

1. On considère la suite définie par : $u_0 = 1$
$u_{n+1} = 3*u_n + 2$

Ecrire une fonction qui calcule et renvoie en résultat le n^{ième} terme de la suite u.

2. Ecrire un programme utilisant cette fonction, qui demande à l'utilisateur de saisir des entiers jusqu'à ce qu'il entre -1 et qui affiche à chaque fois le terme de la suite correspondant.

3. On veut maintenant laisser le choix du premier terme k de la suite. Ecrire une fonction qui calcule et renvoie en résultat le n^{ième} terme de la suite :
$u_0 = k$
$u_{n+1} = 3*u_n + 2$

Exercice 9. Saisir et exécuter le code ci-dessous.
Observer les valeurs de `k` avant et après l'appel à la fonction `compte` : quel lien faites-vous entre cette observation et le mode de passage de paramètres en C ? (cf section V.2)

```
#include <stdio.h>

int compte(int x) {
  int i;
  for (i=1; i<=4; i++)
     x = x * i;   /* qu'arrive-t-il a x ? */
  return x;
}

int main(void) {
  int k;
  printf("Saisir la valeur pour le calcul :  ");
  scanf("%d", &k);
  printf("La fonction renvoie %d\n", compte(k));
  printf("La valeur de k est maintenant %d\n", k);  /* resultat observé ? */
  return 0;
}
```

Exercice 10.

1. Que vous attendez-vous à avoir comme résultat à l'exécution du code ci-dessous ?

```
#include <stdio.h>

void g (void){
  int n;
  n = 22;
}

void f (void){
  int compteur;
  compteur ++;
  printf ("%d\n",compteur);
}

int main (void){
  g();
  f();
  return 0;
}
```

2. Saisir ce code dans un fichier exo10.c, compiler puis exécuter. Obtenez-vous ce que vous aviez prévu ? Pourquoi ? Conclusion quant aux initialisations de variables locales (placées sur la pile) ?

Exercice 11. *optionnel*

1. Ecrire un programme C qui saisit au clavier un nombre entier positif et affiche à l'écran sa valeur en binaire (affichée à partir du bit de poids faible, c'est à dire au fur et à mesure de son calcul), puis qui saisit au clavier une suite de caractères '0' et '1' (terminée par '#') représentant une valeur binaire donnée à partir du bit de poids fort et qui calcule et affiche la valeur décimale correspondante. On ne demande pas d'utiliser des fonctions.

On aura par exemple à l'exécution :

```
-- Essai de conversion decimal -> binaire --
Saisir la valeur a convertir en binaire :  47
Conversion en binaire (a partir du bit de poids faible) :
111101

-- Essai de conversion binaire -> decimal --
Saisir a partir du bit de poids fort (et terminer par #) : 101111#
Valeur lue = 47
```

2. Ecrire un programme C qui saisit au clavier une suite de caractères compris entre les caractères '0' et '9' (terminée par '#') représentant un nombre entier donné à partir de son chiffre de poids fort (attention, on lira caractère par caractère, comme ci-dessus) et qui calcule et affiche la valeur entière correspondante.

On aura par exemple à l'exécution :

```
Saisir une suite de caracteres (et terminer par #) 4532#
Valeur lue = 4532
```

VI. LES TABLEAUX

VI.1 Déclaration et initialisation de tableaux statiques

✓ Les ***tableaux*** sont des structures pouvant contenir *plusieurs éléments du même type*.
Tout tableau doit être *dimensionné*, c'est à dire qu'il faut déterminer le nombre d'éléments qu'il pourra contenir.

✓ Un tableau d'éléments d'un type *T*, de taille *s* sera déclaré de la façon suivante :

T nom-tableau[s];

Il est alors **indexé de 0 à s-1**, et ses éléments sont référencés par *nom-tableau[0], nom-tableau[1], … nom-tableau[s-1]*.

Exemple :

```
int tab[10];
```

Le tableau `tab` est indexé de 0 à 9, et ses éléments sont référencés par `tab[0]`, `tab[1]`, ..., `tab[9]`.

✓ En l'absence d'initialisation, comme pour les variables scalaires, un tableau variable globale est rempli de zéros, et l'initialisation d'un tableau variable locale est imprévisible.
Il est possible d'*initialiser un tableau lors de sa déclaration*, uniquement avec des valeurs qui correspondent à des expressions constantes. Pour cela, il suffit de donner les valeurs entre accolades, séparées par des virgules.

Exemple :

```
float tableau[4] = { 4.7, 8.2, 12., -3.9 };
```

VI.2 Affectation et parcours

✓ *Affectation* : un tableau dimensionné statiquement ne peut pas apparaître en partie gauche d'une instruction d'affectation (mais ses éléments le peuvent), il n'est donc *pas possible d'affecter directement un contenu à tout un tableau*, il faut le faire élément par élément.

✓ Pour *parcourir* globalement un tableau (afin de le remplir, consulter l'ensemble de son contenu, faire subir un traitement à l'ensemble de ses valeurs,...) on aura généralement recours à une boucle *for*.

Exemple :

```
#include <stdio.h>

int main() {
  float tableau[4] = { 4.7, 8.2, 12., -3.9 };
  float tab2[4];
  int i;
  for (i = 0; i<4; i++)
    tab2[i] = tableau[i];
  for (i = 0; i<4; i++)
    printf("tab2[%d] = %f\n", i, tab2[i]);
  return 0;
}
```

qui produit à l'exécution :

```
tab2[0] = 4.700000
tab2[1] = 8.200000
tab2[2] = 12.000000
tab2[3] = -3.900000
```

✓ Comme les variables scalaires, les tableaux peuvent être passés en paramètres aux fonctions.

Exemple :

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

void remplissage(int taille, double tab[]) {
  int i;
  for (i = 0; i<taille; i++) {
    printf("Saisissez tab[%d] : ", i);
    scanf("%lf", &tab[i]);
  }
}

void ajoutelog(int taille, double tab[]) {
  int i;
  for (i = 0; i<taille; i++)
    tab[i] = tab[i] + log10(tab[i]);
}

int main() {
  double tableau[4];
  int i;
  remplissage(4, tableau);
  ajoutelog(4, tableau);
  printf("Voici le contenu final : \n");
  for (i = 0; i<4; i++)
    printf("tableau[%d] = %f\n", i, tableau[i]);
  return 0;
}
```

VI.3 Tableaux à deux dimensions

✓ Les tableaux peuvent être multi-dimensionnels, voyons le cas des tableaux à deux dimensions. Un tableau d'éléments d'un type T, de taille s_1 dans la première dimension et s_2 dans la deuxième dimension, sera déclaré de la façon suivante :

T nom-tableau[s_1][s_2];

Ses éléments sont référencés par *nom-tableau[0][0], nom-tableau[0][1], ... nom-tableau[0][s_2-1], nom-tableau[1][0], nom-tableau[1][1], ... nom-tableau[1][s_2-1], ..., nom-tableau[s_1-1][0], nom-tableau[s_1-1][1], ... nom-tableau[s_1-1][s_2-1].*

Pour les initialiser lors de la déclaration, on pourra procéder de la même façon que pour les tableaux à une dimension.

Exemple :
```
char Ctab[3][2] = { { 'a', 'b' } , { 'c', 'd' }, { 'e', 'f' } };
```

✓ Pour leur faire subir des traitements, on a généralement recours à des *boucles for imbriquées* (attention, il faut deux variables de boucles !).

Exemple :
```
#include <stdio.h>
#define N 3
#define M 2

int main() {
  int letab[N][M];
  int i,j;
  for (i = 0; i<N; i++)
    for (j = 0; j<M; j++) {
      printf("Saisissez letab[%d][%d] : ", i, j);
      scanf("%d", &letab[i][j]);
    }
  printf("Voici ce que vous avez saisi : \n");
  for (i = 0; i<N; i++)
    for (j = 0; j<M; j++)
      printf("letab[%d][%d] = %d\n", i, j, letab[i][j]);
  return 0;
}
```

VII. LES POINTEURS

✓ Une variable de type ***pointeur*** *sur un élément de type T* (repère cet élément) est déclarée de la façon suivante :

*T *nom-de-la-variable;*

Exemples :
```
int *px;    // px est un pointeur sur int
char *pc;   // pc est un pointeur sur char
```

En outre, l'opérateur `&` permet de récupérer **l'adresse** d'une variable.
Et l'opérateur `*` permet d'indiquer **l'objet pointé** (repéré) par le pointeur.

Exemples :
```
int x,y;
int *p = &x;   // p est déclaré et reçoit l'adresse de x
int *q;        // q déclaré mais non initialisé
x = 5;
y = 25;
*p = 3;   // l'objet pointé par p (c'est à dire x) reçoit 3 (x a donc
          // changé de valeur !)
p = &y;   // p est maintenant l'adresse de y
*p += 1;  // y a été incrémenté !
```

✓ Ce type de variable permet notamment de faire des tableaux **dimensionnés dynamiquement** (à l'exécution) - ce qui est interdit par les tableaux "classiques".

Pour ce faire, on a recours à la fonction d'***allocation mémoire*** `malloc` (inclure `stdlib.h`)

Exemple :
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void remplissage(int taille, double *tab) {  // remplit le tableau tab
  int i;
  for (i = 0; i<taille; i++) {
    printf("Saisissez tab[%d] : ", i);
    scanf("%lf", &tab[i]);
  }
}

void ajoutelog(int taille, double *tab) {  // modifie le contenu de tab
  int i;
  for (i = 0; i<taille; i++)
    tab[i] = tab[i] + log10(tab[i]);
}

int main() {
  double *tableau;   // déclaration de la variable "tableau"
  int n,i;
  // pour une allocation dynamique, on peut ne connaître la taille
  // qu'à l'exécution (n ici) :
  printf("Quelle est la taille du tableau ?\n");
  scanf("%d", &n);
  // allocation mémoire (pour n éléments de la taille d'un double) :
  tableau = (double *)malloc(n * sizeof(double));
  // seulement maintenant que l'allocation est faite, on peut remplir,
  // modifier, etc… le tableau :
  remplissage(n, tableau);
  ajoutelog(n, tableau);
  printf("Voici le contenu final : \n");
  for (i = 0; i<n; i++)
    printf("tableau[%d] = %f\n", i, tableau[i]);
  return 0;
}
```

Sauf dans le cas où un pointeur reçoit l'adresse d'un élément déjà existant, et donc déjà alloué (comme dans l'exemple `p = &y;` plus haut), un élément de type pointeur **doit faire l'objet d'une allocation mémoire** par `malloc`.

Exemple :
```
int main() {
  double *tableau;   // déclaration de la variable "tableau"
  double *ptelem1, *ptelem2;
  int n;
  double d = 12.345;

  // Cas de l'exemple précédent, on veut utiliser un tableau de
  // n éléments :
  printf("Quelle est la taille du tableau ?\n");
  scanf("%d", &n);
  tableau = (double *)malloc(n * sizeof(double));
  // etc...
  // Cas où on ne veut manipuler qu'un seul élément :
  ptelem1 = &d;
  ptelem2 = (double *)malloc(sizeof(double));
  *ptelem2 = 567.89;
  // etc...
}
```

La fonction **`free`** permet de désallouer un bloc alloué par `malloc` (voir par exemple http://www.manpagez.com/man/3/free/).
Un objet de type pointeur (contrairement à un tableau - qui est en fait une adresse *constante*, l'adresse du premier élément du tableau) peut apparaître en partie gauche d'une instruction d'*affectation*, car c'est une *variable*. C'est le cas des variables `tableau`, `ptelem1` et `ptelem2` dans l'exemple ci-dessus.

Il est possible de se déplacer dans un tableau (statique ou dynamique) en ayant recours à l'arithmétique sur pointeurs : une ***incrémentation*** de pointeur correspond à un déplacement en mémoire de la taille (nombre d'octets) d'un élément du tableau.

Exemple :

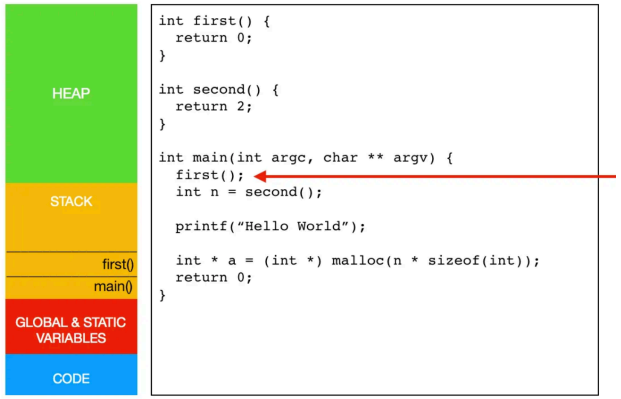
```
int main() {
  double *tableau, *t;
  int n;

  printf("Quelle est la taille du tableau ?\n");
  scanf("%d", &n);
  // allocation mémoire (pour n éléments de la taille d'un double) :
  tableau = (double *)malloc(n * sizeof(double));
  remplissage(n, tableau);
  ajoutelog(n, tableau);
  printf("Voici le contenu final : \n");
  for (t=tableau; t<tableau+n; t++)
      printf("%f\n", *t);
  return 0;
}
```

✓ Quelques mots sur la **gestion mémoire** :
Les **variables locales** des fonctions sont stockées sur la **pile** (stack), un cadre de pile étant utilisé pour chaque appel de fonction (il permet de sauvegarder les arguments de la fonction, l'adresse de retour, les variables locales).

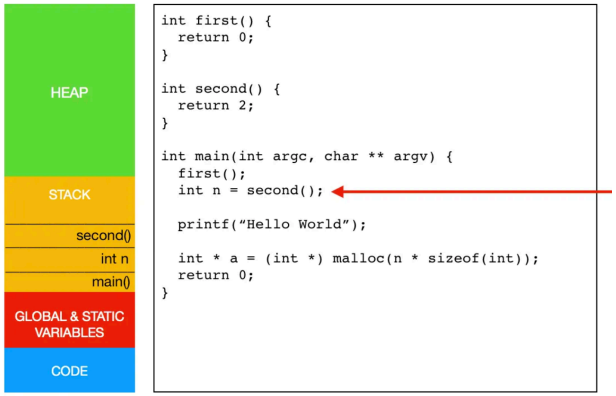
Les blocs alloués par **allocation dynamique** (malloc) se trouvent dans le **tas** (heap).

Voir des explications complémentaires sur ces deux segments de mémoire par exemple ici :
https://medium.com/@wireless.patrick/a-tale-of-memories-stack-and-heap-7528f49aea6e

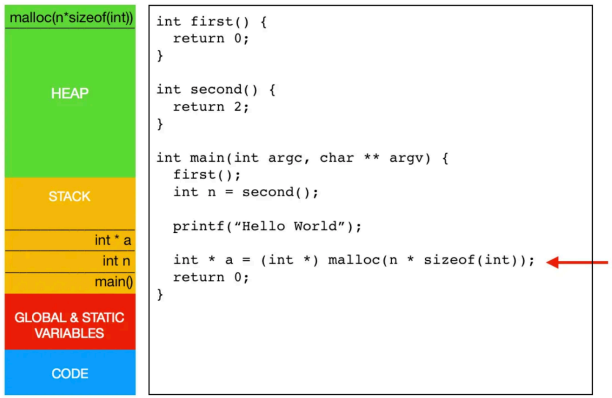
Exemple : Voyons un exemple simple proposé sur cette même page



Après la sortie de la fonction *first()*, son cadre de pile n'est plus utilisé, le pointeur de pile "redescend". La variable *n* est empilée, et l'appel à *second()* donne naissance à un nouveau cadre de pile pour cet appel :



Après la sortie de la fonction *second()*, son cadre de pile n'est plus utilisé, le pointeur de pile "redescend". La variable *a* est empilée et le bloc mémoire est, lui, alloué dans le tas. La valeur que reçoit *a* est l'adresse de ce bloc :



✓ Notons enfin qu'il est possible de manipuler tableaux et pointeurs conjointement.

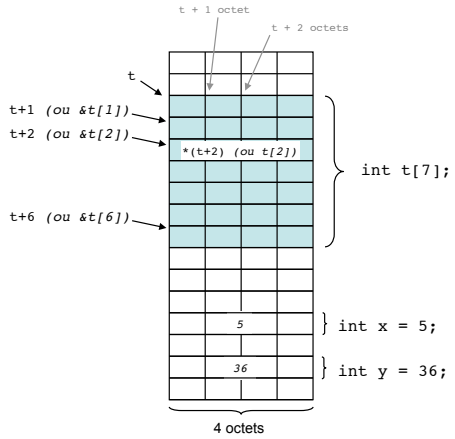
Dans l'exemple ci-dessous, le pointeur p peut se déplacer dans le tableau t (ou ailleurs), même si t a été déclaré comme un tableau "classique".

Mais attention, avec la déclaration de t comme un tableau "classique" :

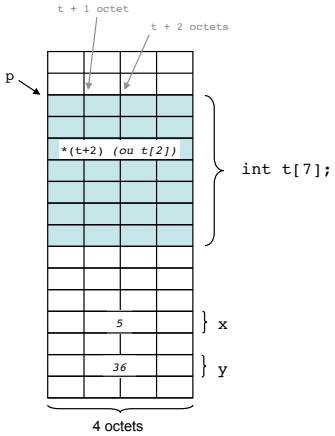
```
int t[7];
```

il est impossible de modifier (affecter) t, qui a la nature d'une constante.

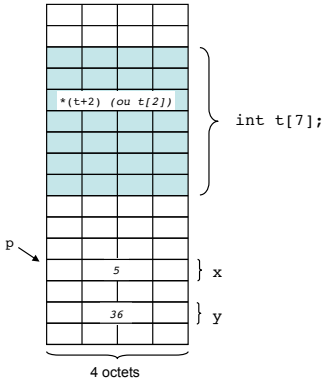
Exemple :
L'objectif est d'illustrer l'utilisation de pointeurs, et l'arithmétique sur pointeurs.
A l'origine nous avons le tableau t de 7 entiers, la variable x et la variable y (nous ne représentons que ces variables ici, et sans hypothèse quant à l'ordre de leurs déclarations et leur positionnement exact sur la pile). Nous supposons que les entiers sont codés sur 4 octets.



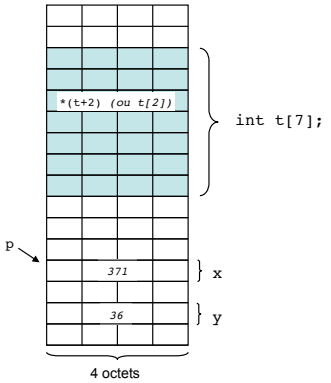

```
int *p = t;  // affectation de pointeur
```



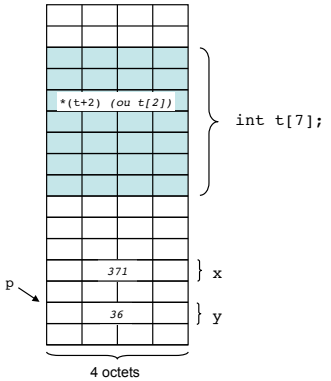

```
p = &x;   // autre affectation de pointeur
```


```
*p = 371;    // Modification de la valeur référencée
```




```
p = &y;    // Re-déplacement du pointeur
```



Exercice 12.

1. Que vous attendez-vous à avoir comme résultat à l'exécution du code ci-dessous ?

```
#include <stdio.h>
#define N 50

char *tableau_rempli (char c) {
  char tabl[N];
  int i;
  for (i=0; i < N; i++)
    tabl[i] = c;
  return tabl;
}

int main(void) {
  char x;
  char *t;
  int j;
  printf("Quel caractere met-on dans le tableau ? ");
  scanf("%c", &x);
  t = tableau_rempli(x);
  printf("Voici le tableau rempli : \n");
  printf(" [ ");
  for (j=0; j < N; j++) {
    printf("%c ", t[j]);
  }
  printf(" ] \n");
  return 0;
}
```

2. Saisir ce code dans un fichier exo12.c, compiler puis exécuter. Obtenez-vous ce que vous aviez prévu ? Expliquer très précisément en rappelant l'utilisation des variables allouées sur la pile ou dans le tas.

3. Quelle variable doit être allouée (dynamiquement) dans le tas ici ? Corriger la fonction tableau_rempli en conséquence.

Exercice 13.

1. Que vous attendez-vous à avoir comme résultat à l'exécution du code ci-dessous ?

```
#include <stdio.h>

int main() {
    int tab[10] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
    int *p;

    p = tab;
    printf("int %lu ; int* %lu ; long %lu ; long* %lu ; tab %lu ; p %lu\n",
           sizeof(int), sizeof(int *), sizeof(long), sizeof(long *),
           sizeof(tab), sizeof(p));

    char *q = (char *)p;
    int i;
    for (i = 0; i < 10; i++) {
        printf("tab[%d] = %d ; *(p+%d) = %d ; *(q+%i) = %d\n",
               i, tab[i], i, *(p+i), i, *(q+i));
    }
    return 0;
}
```

2. Saisir ce code dans un fichier exo13.c, compiler puis exécuter. Obtenez-vous ce que vous aviez prévu ? Expliquer très précisément en analysant selon les tailles des éléments pointés par p

et par q.

VIII. LES STRUCTURES

✓ Les ***structures*** permettent de rassembler un nombre quelconque de données, de types quelconques. Les éléments d'une structure s'appelent des champs, ils ont un identifiant et un type. Une structure se définit comme suit :

struct nom-structure {
type-champ1 nom-champ1;
type-champ2 nom-champ2;
…
};

Pour créer une variable du type de cette structure, il faut alors la typer *struct nom-structure*.
On accède à ses champs par notation pointée.

Exemple :

```
struct Livre {
   int numero;    // numero d'identification du livre à la bibliothèque
   char auteur[30];    // nom de l'auteur
   char titre[100];    // titre du livre
};

int main(){
  struct Livre l;
  struct Livre *pl;

  l.numero = 1;
  strcpy(l.auteur, "H.G.Wells");
  strcpy(l.titre, "La guerre des mondes");

  pl = (struct Livre *)malloc(sizeof(struct Livre)); // allocation d'un
                              // emplacement mémoire de taille suffisante
                              // pour contenir un struct Livre
  (*pl).numero = 2;     // ou pl->numero = 2;
  strcpy((*pl).auteur, "V.Hugo");
  strcpy(pl->titre, "Hernani");
  ...
}
```

✓ On utilisera fréquemment le mot-clé *typedef* pour renommer le type de façon plus concise (i.e., sans le mot-clé *struct*).

Exemple :

```
struct Livre {
   int numero;    // numero d'identification du livre à la bibliothèque
   char auteur[30];    // nom de l'auteur
   char titre[100];    // titre du livre
};
typedef struct Livre _Livre;   // le type _Livre est maintenant synonyme
                               // de struct Livre

int main(){
  _Livre l;

  l.numero = 1;
  strcpy(l.auteur, "H.G.Wells");
  ...
}
```

✓ Attention, ne pas oublier qu'il peut être nécessaire de procéder à une **allocation** mémoire pour

des **champs** de la structure.

Exemple :

```
struct Livre {
   int numero;    // numero d'identification du livre à la bibliothèque
   char *auteur;    // nom de l'auteur
   char *titre;     // titre du livre
};
typedef struct Livre _Livre;

int main(){
  _Livre *pl;

  pl = (_Livre *)malloc(sizeof(_Livre)); // l'allocation mémoire n'est faite
                    // que pour un entier et 2 pointeurs (entiers non signés)
  pl->numero = 2;
  // Le champ auteur est juste un pointeur, il faut faire l'allocation
  // pour la chaine de caractères qui sera reçue par ce champ :
  pl->auteur = (char *)malloc(strlen("V.Hugo")+1);
  strcpy(pl->auteur, "V.Hugo");
  // Même chose pour le champ titre :
  pl->titre = (char *)malloc(strlen("Hernani")+1);
  strcpy((*pl).titre, "Hernani");
}
```

IX. COMPLEMENT

✓ Complément - Passage de **paramètres à la fonction `main`** : pour pouvoir lire des arguments sur la ligne de commande, on utilisera l'en-tête suivant pour la fonction `main`

*int main(int argc, char *argv[]);*

où

- *argc* correspond au nombre d'arguments +1 (c'est à dire le nombre de mots sur la ligne de commande, nom de commande compris),
- *argv* est un tableau de chaînes de caractères, la première est le nom de la commande, et les autres sont les arguments de la commande.

Exemple : `$ monexec -n fic1`
alors argc=3, argv[0] = "monexec",
argv[1] = "-n", et argv[2] = "fic1"

X. QUELQUES MOTS SUR GDB…

✓ L'utilisation de ***gdb*** (GNU debugger) est très utile pour identifier rapidement les erreurs dans vos programmes, en particulier lorsqu'elles sont liées aux pointeurs ou allocations mémoire.

Attention, pour pouvoir l'utiliser, le programme doit être compilé avec l'option `-g`

✓ Ce debugger propose un large ensemble de commandes, en particulier :

- *break* pour positionner des breakpoints, et des commandes telles que *step*, *next*, *continue* pour réaliser des exécutions pas à pas
- *print* (ou *p*) pour afficher la valeur courante d'une variable (noter que *p/a* par exemple permet d'afficher comme une adresse hexadécimale)
- *backtrace* pour afficher une trace de la pile d'exécution (empilement des appels de fonctions), *up* et *down* pour se déplacer dans cette pile (aller dans le contexte d'une fonction donnée)

(voir ici par exemple : http://www.yolinux.com/TUTORIALS/GDB-Commands.html)

✓ En particulier, lorsque l'exécution de votre programme échoue avec un *Segmentation fault*, vous pouvez facilement identifier la raison de cet échec, comme illustré par l'exercice 14…

Exercice 14.

Dans cet exercice, nous considérons le programme élémentaire ci-dessous :

```
#include <stdio.h>

struct example {     // structure avec deux champs
  int field1;
  float field2;
};

void init1(int x, struct example *e){     // affectation de field1
   e->field1=x*2;
}

void init2(float x, struct example *e){   // affectation de field2
  e->field2=x+10;
}

void initialization(int x, float y, struct example *e){
  init1(x,e);
  init2(y,e);
}

void print_structure(struct example *e){
  printf("In this structure, we have %d and %f\n", e->field1, e->field2);
}

struct example *s;

int main() {
  int i;
  float f;
  printf("which integer for the structure? ");
  scanf("%d", &i);
  printf("which real number for the structure? ");
  scanf("%f", &f);
  initialization(i,f,s);
  printf("after initialization, the structure is as follows:\n");
  print_structure(s);
  return 0;
}
```

1. Compiler et exécuter ce programme. Vous allez obtenir quelque chose comme :

```
$ ./ex
which integer for the structure? 8
which real number for the structure? 15.3
Segmentation fault
```

Vous allez maintenant **utiliser *gdb*** pour **identifier la raison de cet échec** (l'objectif principal n'est pas de trouver l'erreur dans cet exemple trivial, mais de **comprendre comment gdb peut être utile dans un tel cas**).
Tout d'abord, recompiler le programme avec l'option appropriée, puis l'exécuter dans *gdb* :

```
$ gcc -g ex.c -o ex
$ gdb ./ex
GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
Reading symbols from ./ex...done.
(gdb)
```

Utiliser la commande *run* pour exécuter le programme :

```
(gdb) run
Starting program: ex
which integer for the structure? 8
which real number for the structure? 15.3

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005cb in init1 (x=8, e=0x0) at ex.c:10
10       e->field1=x*2;
```

Bien sûr, vous obtenez le même échec. De plus, nous savons maintenant l'instruction sur laquelle l'exécution a échoué. Nous voyons que cette instruction utilise les variables `e`, `e->field1` et `x`.
Utiliser la commande *print* pour afficher leurs valeurs, qu'observez-vous ?

Nous allons maintenant identifier le niveau des appels de fonctions à partir duquel la valeur est devenue erronée, cela nous permettra de savoir dans quelle fonction il faut corriger l'erreur.
Utiliser la commande *backtrace* et observer la **pile d'exécution** :

```
(gdb) backtrace
#0  0x00000000004005cb in init1 (x=8, e=0x0) at ex.c:10
#1  0x000000000040061f in initialization (x=8, y=15.3000002, e=0x0) at ex.c:18
#2  0x00000000004006db in main () at ex.c:37
```

Remonter d'un niveau dans la pile d'exécution (up). Dans quelle fonction êtes-vous ? Afficher les valeurs des variables `e`, `e->field1` et `x`, qu'observez-vous ?
Remonter d'un niveau encore (up). Dans quelle fonction êtes-vous ? Afficher les valeurs des variables `i` et `s`, qu'observez-vous ? Quelle est votre conclusion ?

2. Modifier ce code pour corriger l'erreur. Vérifier que l'exécution n'échoue plus.
Rappelez-vous que cette démarche peut permettre de **détecter très rapidement la raison d'une erreur d'exécution** !

Exercice 15.

0. Déclarer des macros N de valeur 30, BORNEINF de valeur 100 et BORNESUP de valeur 150.

1. Ecrire une fonction d'entête

```
void init(int *tabl, int taille);
```

permettant de remplir un tableau d'une taille donnée avec des valeurs entières générées pseudo-aléatoirement entre BORNEINF et BORNESUP.

Ecrire une fonction d'entête

```
int recherche(int x, int *tabl, int taille);
```

permettant de rechercher la présence de l'élément x dans un tableau de taille donnée. Cette fonction renverra -1 si l'élément est absent, et l'indice de l'élément s'il est présent.

Ecrire le programme C dans lequel vous déclarez un tableau `tabint` de N entiers, vous le remplissez avec des valeurs générées pseudo-aléatoirement entre BORNEINF et BORNESUP, vous générez aussi un entier `cherche` entre BORNEINF et BORNESUP, et vous faites afficher un message indiquant si cet entier se trouve dans le tableau `tabint`.

2. Reprendre ce même programme avec une fonction d'initialisation ayant maintenant l'entête

```
int *init(int taille);
```

On rappelle que la **génération de nombres pseudo-aléatoires** selon une loi de probabilité uniforme consiste à construire une suite u_k et nécessite donc l'initialisation de la "graine" (c'est à dire de u_0).

Afin que la suite construite ne soit pas toujours la même, il convient de faire des initialisations différentes de la graine d'une exécution sur l'autre; pour cela on utilise habituellement comme valeur de graine le temps courant (utilisation de la fonction `time`).
En C, cette initialisation de la graine se fera grâce à la fonction (à utiliser 1 seule fois, au début) :

```
void srand (unsigned int seed);
```

Puis la génération de nombres pseudo-aléatoires successifs se fera par la fonction :

```
int rand (void);
```

On pourra consulter par exemple `http://perso.iut-nimes.fr/wdesrat/faq01.html` pour plus de détails.

Exercice 16. *optionnel*

Reprendre l'exercice 11 en considérant maintenant que les entiers codés en binaire (pour la question 1) ou les suites de caractères (pour la question 2) sont rangés dans des **tableaux**. Attention, on définira maintenant des **fonctions** pour les conversions, qu'on appellera depuis la fonction *main* :

- une fonction *convert_vers_bin* qui renvoie en résultat un vecteur contenant le codage binaire d'un nombre entier positif x, sur un nombre donné de bits,
- une fonction *convert_vers_dec* qui renvoie en résultat un nombre entier positif correspondant à un vecteur de bits d'une taille donnée,
- une fonction *transcript* qui renvoie en résultat un nombre entier positif correspondant à une chaîne de caractères (les caractères étant des chiffres).

On définira de plus une fonction pour l'affichage d'un vecteur de bits, afin d'afficher le résultat de *convert_vers_bin*.

Exercice 17. *Supplément, pour ceux qui sont arrivés jusqu'ici en ayant tout fait…*

Ce problème consiste à traiter des grilles de mots croisées partiellement ou entièrement remplies, comme celle de l'illustration ci-dessous :

On va considérer que la grille est codée comme un tableau carré à 2 dimensions, de caractères. Le paramètre taille du tableau (9 sur l'exemple) sera passé via la ligne de commande. Les caractères contenus dans ce tableau peuvent être : une * pour représenter une case noire, un blanc (caractère espace) pour représenter une case non encore remplie, une lettre minuscule si la case a été remplie.

1. Définir le type `grille` permettant de représenter un tel tableau (de taille indéfinie).
Ecrire l'ébauche de la fonction `main`, qui déclare une variable G destinée à représenter la grille de mots croisés, s'assure qu'elle a reçu un paramètre (taille du tableau - on ne demande pas de vérifier que le paramètre représente bien un entier positif) et dans ce cas réalise l'allocation mémoire correspondante pour G

2. Ecrire une fonction `affiche` pour l'affichage à l'écran du contenu d'une grille qu'elle reçoit en paramètre (penser à espacer pour avoir une représentation claire). Par exemple :

```
Contenu de la grille :
a r c e n c i e l
p * a n * u n * u
p a r e n t * n l
o r * m o i s * u
n * e a u * b r *
d a n * b i * h o
e v o c a t i o n
n e u f * o c * c
t u e * t u i l e
```

3. En supposant une grille entièrement remplie, écrire une fonction

```
void compte(grille g, int n, int *nbv, int *nbc);
```

qui renvoie par l'intermédiaire de ses paramètres nbv et nbc le nombre de voyelles et le nombre de consonnes contenues dans la grille de taille `n` (on rappelle que les voyelles sont : a, e, i, o, u, y); on utilisera une instruction "switch".

4. Ecrire une fonction `compte_mots` qui renvoie en résultat le nombre de mots contenus dans une grille entièrement remplie (à l'horizontale et à la verticale). Attention une seule lettre ne représente pas un mot !

5. On suppose maintenant que le joueur n'a pas fini de remplir la grille, on est par exemple dans l'état :

```
a r c e n c i   l
p * a n * u n * u
p a r e n t * n l
o r * m   i s * u
n * e a u * b   *
    n * b i * h
e v o c a t i o n
n e u f * o c *
t u e * t u     e
```

Ecrire une fonction `compte_mots_finis` qui renvoie en résultat le nombre de mots complets, c'est à dire dans lesquels il n'y a plus de blanc (à l'horizontale et à la verticale).

6. En supposant l'existence de deux fonctions

```
void grille_en_cours(grille g, int n);
```

et

```
void grille_finie(grille g, int n);
```

qui remplissent respectivement la grille avec une grille en cours de jeu et une grille complètement finie, compléter le `main` de la question 1. de telle façon qu'il réalise les opérations suivantes : remplissage de G avec la grille en cours de jeu, affichage de G, affichage du nombre de mots complets dans G, puis remplissage de G avec la grille finie, affichage de G, affichage du nombre de voyelles et de consonnes dans G et affichage du nombre de mots contenus dans G.

7. On souhaite maintenant stocker chaque mot de la grille finie comme une liste chaînée de caractères, et on rassemblera tous les mots dans un tableau de listes chaînées (de taille maximale NMAXMOTS). Une liste chaînée sera caractérisée par un pointeur sur le premier maillon et un pointeur sur le dernier maillon. Définir :

- la constante symbolique (macro) NMAXMOTS de valeur 100,
- le type `maillon` pour représenter un maillon de la liste (voir la déclaration de *structures*),
- le type `mot` pour représenter la liste.

Ecrire une fonction `mot_vide` qui crée et renvoie en résultat une liste vide (pas de caractère encore).
Ecrire une fonction `nouveau` qui crée et renvoie en résultat un nouveau maillon contenant un caractère `c`
Ecrire une fonction `insertion` qui insère en fin d'un mot `m` un maillon portant un caractère `c`

8. Ecrire une fonction

```
void remplir_mots(grille g, int n, mot** tm);
```

qui remplit un tableau `tm` avec toutes les listes chaînées représentant tous les mots de la grille `g` (à l'horizontale et à la verticale).

En supposant l'existence d'une fonction `affiche_mots` permettant d'afficher le contenu d'un tel tableau, compléter le `main` de la question 6. par l'ajout de la déclaration d'une variable tabmots représentant un tableau pouvant contenir au maximum NMAXMOTS mots, du remplissage de ce tableau avec les mots contenus dans G, puis l'affichage du contenu de ce tableau.