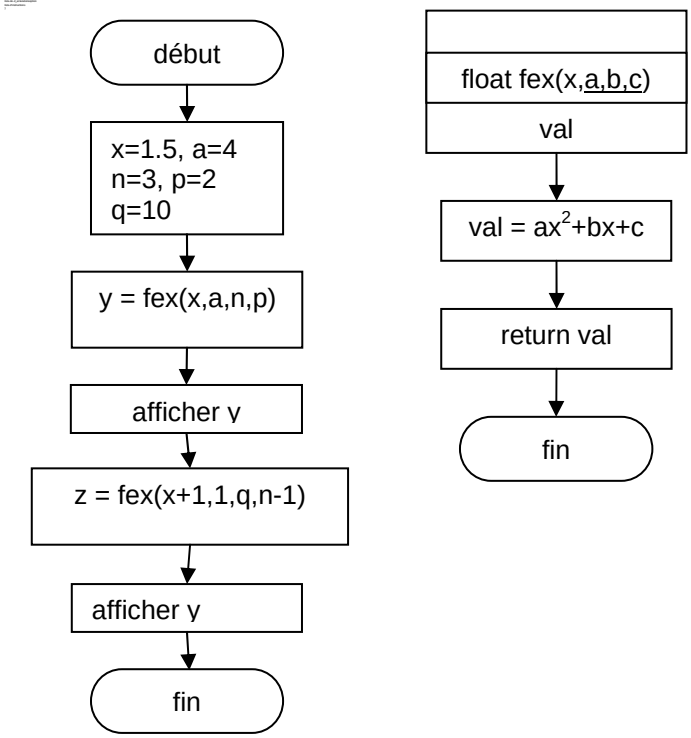
# Les Sous-programmes

## *Les sous-programmes en général*

Le sous-programme est une partie de programme presque indépendante qui a un nom et peut être appelée d'un autre sous-programme ou du programme principal. Il y a deux sortes de sous-programmes – procédures et fonctions. L'appel d'une fonction est une expression, c'est-à-dire que la fonction renvoie un résultat (valeur). L'appel d'une procédure est une instruction, c'est-à-dire que la procédure ne renvoie rien, Elle exécute certain code. En C on voie que certaines instructions sont des expressions à la fois et vice versa, C'est pourquoi en C il n'y a que des fonctions mais il y a des fonctions qui ne renvoient rien.

Exemple: Calculer la fonction $f(x) = ax^2 + bx + c$ pour quelques ensembles de valeurs pour les arguments a,b et c (fig.5.1)



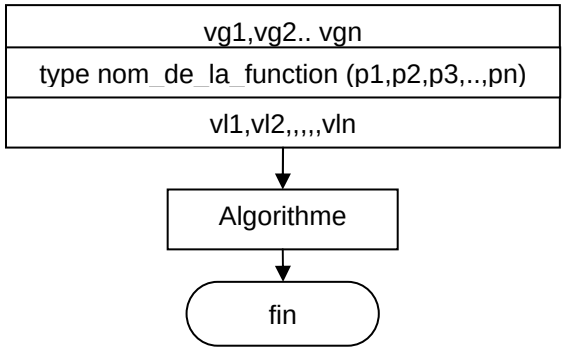

```
#include <stdio.h>

void main (void) {
float x = 1.5;
float y,z;
int a=4, n=2, q = 7, p=3;
float fex(float, int,int,int);

 y = fex(x,a,n,p);
 printf ("y = %8.3f\n",y);
 z = fex(x+1,1,q,n-1);
 printf ("z = %8.3f\n",z);
}
float fex(float x, int a,int b,int c){
   float val, val1=3.4;
   val = a*x*x+b*x+c;
  return val;
}
```

Fig,5.1 Exemple d'une fonction

## Passage des paramètres




Fig,5.2 L'algorithme de la fonction

```
type-de-retour nom-de-la-fonction (
   type par1, type par 2, ..., type parn)
{
   déclaration des variables locales ;
   instructions avec au moins une
            instruction return
}
```

Fig,5.3a La syntaxe de la définition d'une fonction en C

```
nom-de-la-fonction (   par1, par 2, ..., parn)
```

Fig,5.3b La syntaxe de l'appel d'une fonction en C

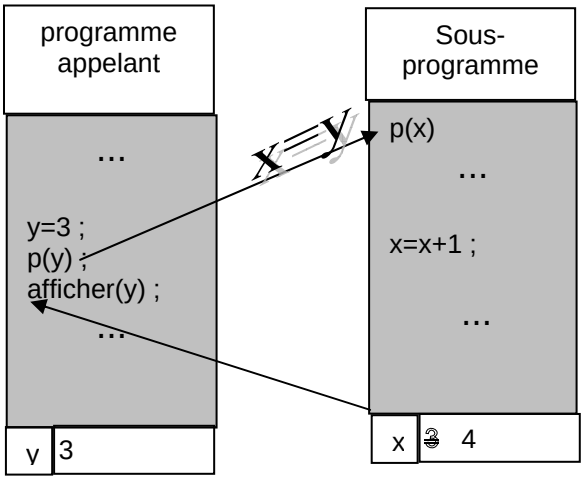
Dans tous les langages de programmation le sous-programme peut avoir des paramètres à l'aide de lesquels on décrit l'algorithme (fig. 5.2) Ces paramètres sont appelées **paramètres formels**. Les paramètres sont nécessaires pour que le sous-programme puisse communiquer avec le programme qui l'appelle, Dans un autre sous-programme (dit appelant) on peut exécuter le sous-programme à l'aide d'une construction spéciale dite appel du sous-programme. L'appel toujours comporte le nom du sous-programme et une liste des **paramètres effectifs**, qui vont être passés à place des paramètres formels selon des règles bien élaborées, pour que le sous-programme soit exécuté avec eux. Le nombre et les types des

paramètres de l'appel doivent correspondre aux ceux du sous-programme. Quand le l'appel est effectué le programme appelant passe les paramètres effectifs aux sous-programme et passe lui le contrôle. Le sous-programme s'exécute et quand il termine son travail il rends au programme appelant le contrôle au point suivant l'appel.

Il y a deux mécanismes de passage des paramètres :

### Passage par valeur

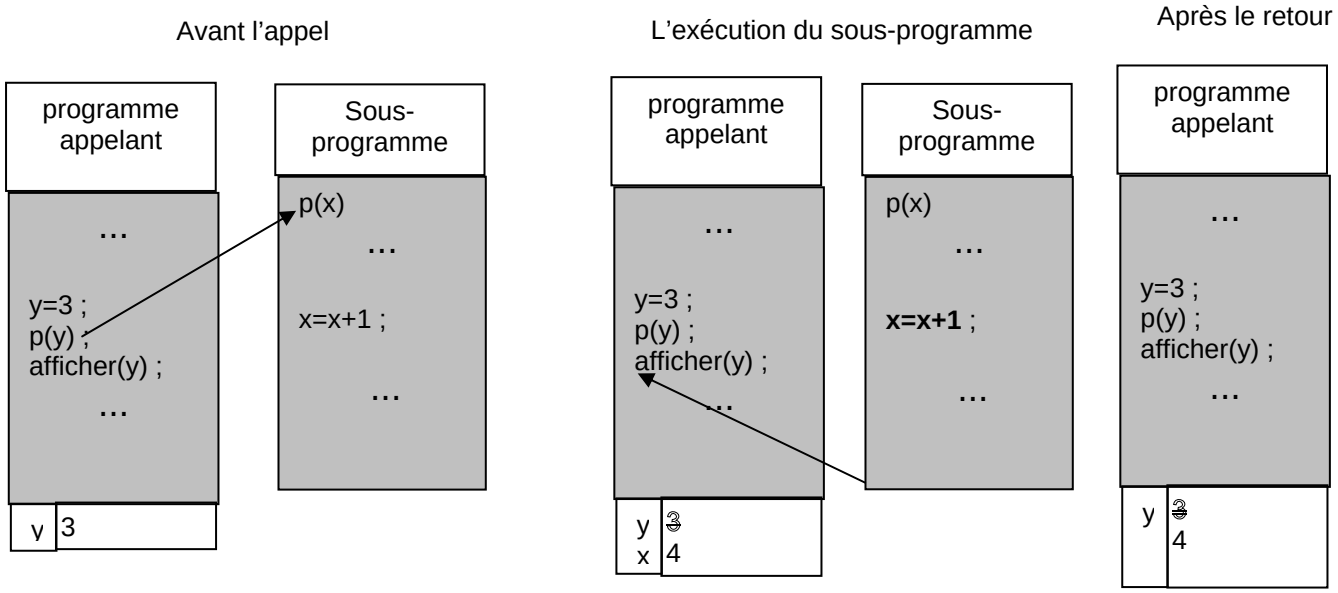
Quand le paramètre est passé par valeur la valeur du paramètre effectif est affecté au paramètre formel. Quand le sous-programme termine il ne rend que le contrôle. Donc le paramètre effectif ne peut pas être affecté. On ne peut que utiliser sa valeur. (fig.5.4)




Fig,5.4 Passage par valeur

### Passage par adresse

Dans ce cas le programme appelant passe l'adresse du paramètre effectif au sous-programme. (fig. 5.5). Cette adresse temporellement devient l'adresse du paramètre formel. Tant que le sous-programme est exécuté les deux paramètres se trouve à la même endroit de la mémoire, c'est-à-dire pour ce temps ***x*** devient pseudonyme d'***y***. Après le retour du sous-programme ***x*** cette liaison est détruite, mais la dernière valeur de ***x*** reste comme une valeur d'***y.*** De telle façon on peut obtenir des résultats du sous-programme***,***



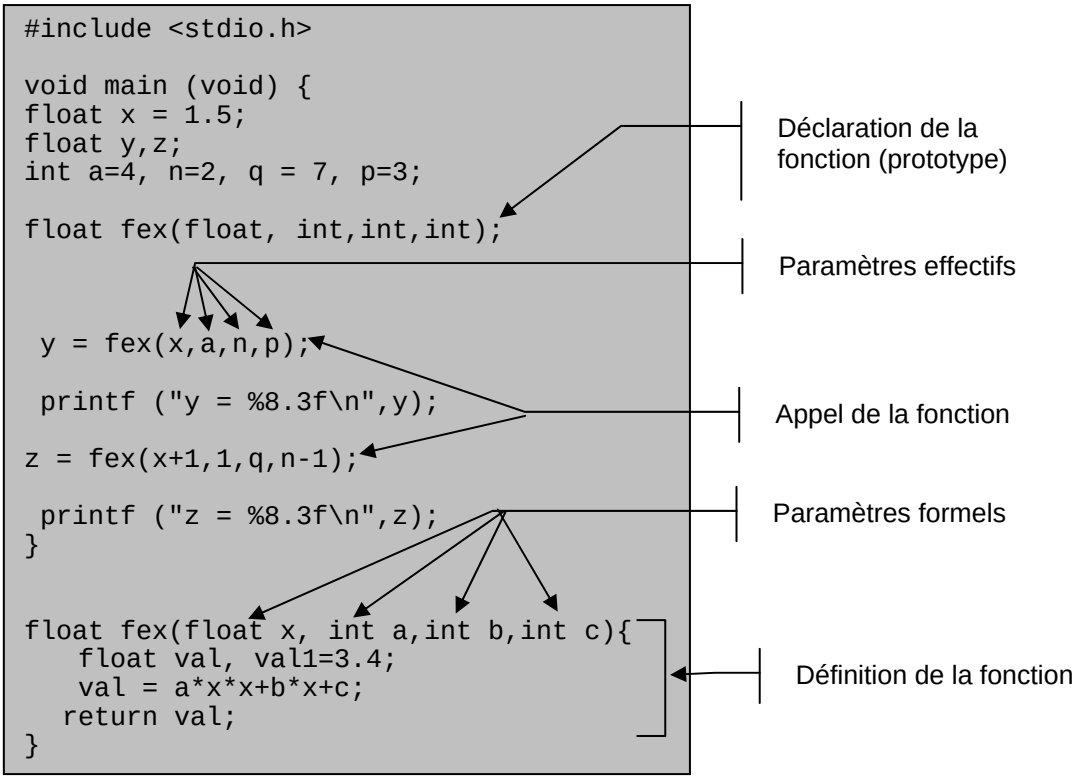

Fig,5.5 Passage par adresse

## *Les fonctions en C*

La fonction a un nom, type du résultat et certains paramètres, appelés paramètres formels (muets). La fonction est dénotée dans l'organigramme avec un pavé initial (au lieu de l'élément « début » et au dessus on dessine l'algorithme de la fonction. Dans le pavé on écrit le nom de la fonction, son type, les types et les noms des paramètres formels, les

variables globales et les variables locales. (Fig.5.2). cette construction se traduit comme la définition de la fonction (Fig.5.3).

La définition peut être n'importe où dans le fichier mais au-dessus du premier appel de la fonction doit exister une déclaration qui ne contient que le titre (le type, le nom et les types des paramètres (fig 5.6). La définition décrit l'algorithme de la fonction en utilisant des paramètres formels Cette fonction peut être appelée par depuis une autre fonction (dite appelante) avec un ensemble de paramètres effectifs qui remplacent les paramètres formels et dans ce cas la fonction est exécutée. Le nombre des paramètres effectifs doit être égal au nombre des paramètres formels et leurs types doivent conformer aux types formels. Si la définition est située dans le même fichier et est avant le premier appel elle joue le rôle d'une déclaration aussi. La déclaration est nécessaire pour que le compilateur puisse vérifier que le nombre et les types des paramètres effectifs conforment au nombre et aux types des paramètres formels.




Fig,5.6 Déclaration, définition, appel

**Le passage des paramètres en C ne se produit que par valeur.** Le type de résultat peut être un type scalaire, un pointeur ou une structure. L'appel peut apparaître à chaque place où on une valeur du type de résultat est permise, c'est-à-dire dans expressions. La valeur rendue par la fonction est la valeur de l'expression appel de fonction. Mais on peut ne pas utiliser le résultat et dans ce cas on peut écrire l'appel comme une instruction séparée.

Mais, en C, la fonction pourra prendre des aspects différents, pouvant complètement dénaturer l'idée qu'on se fait d'une fonction. Par exemple:

- La valeur d'une fonction pourra très bien ne pas être utilisée; c'est ce qui se passe fréquemment lorsque vous utilisez printf ou scanf. Bien entendu, cela n'a d'intérêt que parce que de telles fonctions réalisent une action (ce qui, dans d'autres langages, serait réservé aux sous-programmes ou procédures).
- Une fonction pourra ne fournir aucune valeur.

```
void nom(paramètres)
```

- Une fonction peut ne pas avoir de paramètres

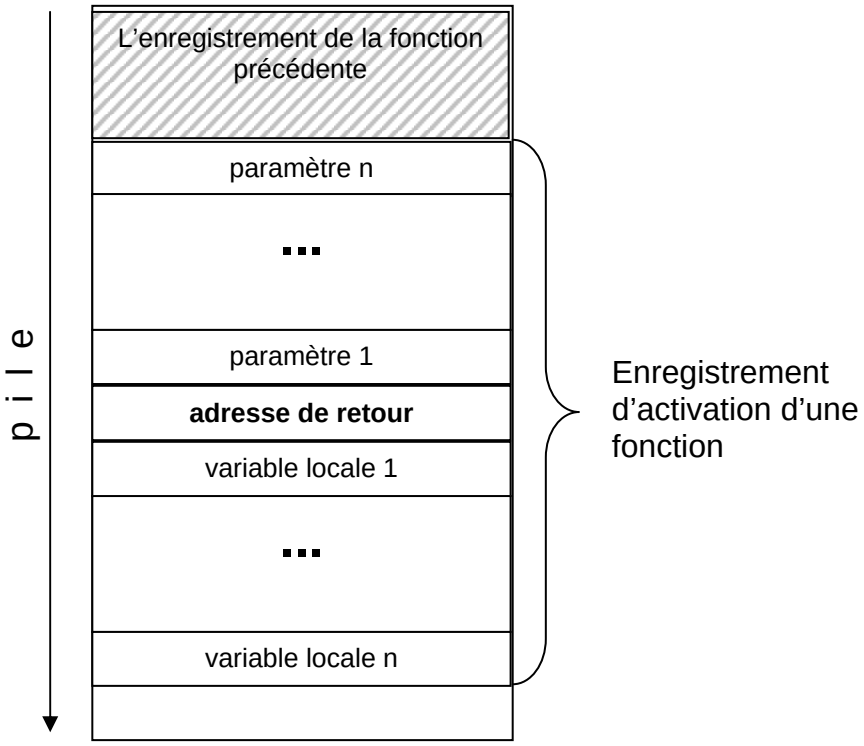
```
type nom(void)
```

- Ainsi, donc, malgré son nom, en C, la fonction pourra jouer un rôle aussi général que la procédure ou le sous-programme des autres langages.

## Exécution d'une fonction

Les données d'un programme sont mis en deuz endroits – la pile et le tas. La structure pile est formée des données automatiques (voir plus bas) des fonctions en train d'exécution. Sa propriété la plus importante est que les membres les plus récentes doivent être supprimées les premières. Les action d'appel sont :

- Un enregistrement d'activation avec la structure montrée à la fig. 5.7 est créé et placé dans la pile (les variables locales commencent à exister). Il contient les variables locales, les paramètres appelés par valeur comme des variables locales.
- Les expressions correspondant aux paramètres appelés par valeur sont calculées et les valeurs sont affectées aux variables locales définies par les paramètres formels.
- Les instructions de la procédure sont exécutées; toute instruction décrite comme agissant sur un paramètre formel appelé par valeur agit sur la variable avec ce nom de l'enregistrement d'activation. L'instruction **return** termine l'exécution ;
- L'enregistrement d'activation est détruit (les variables locales n'existent plus).
- L'expression continue d'être évaluée avec le résultat de la fonction à la place d'appel.




Fig,5.7 Enregistrement d'activation

**Exemple ;**

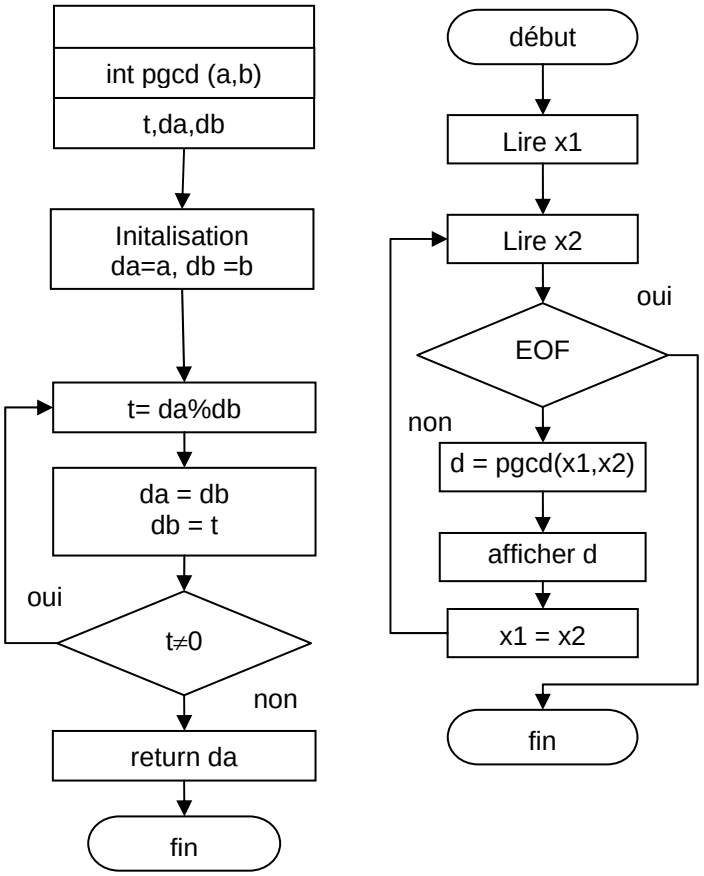
**Faire une fonction qui trouve le pgcd de deux nombres entiers positifs. Ecrire un programme qui lit une suite de nombres et calcule le pgcd de chaque 2 nombres voisins.** L'algorithme de pgcd on a examiné aux TD2. Les algorithmes sont dessinés à la fig. 5.8.

## Les paramètres résultats

L'utilisation normale des paramètres passés par valeur signifie que les seul résultat qu'on peut obtenir par un appel c'est le résultat de la fonction elle-même, parce qu'on ne peut pas changer les valeurs des paramètres effectifs. Mais comment procéder quand on a besoin de plusieurs valeurs comme résultats ? Par exemple si on veut d'écrire une fonction qui échange les valeurs de ces deux paramètres, on a besoin de changer leurs valeurs. Pour ce but en C on utilise les pointeurs comme des paramètres formels et on passe comme les adresses des variables dont les valeurs on veut changer. Dans ce cas la valeur du pointeur n'est pas changée (passée par valeur), mais la valeur de la variable désignée par ce pointeur est changée (fig. 5.9). Et c'est la valeur de cette variable dont l'adresse était passée comme paramètre effectif.

**Exemple**

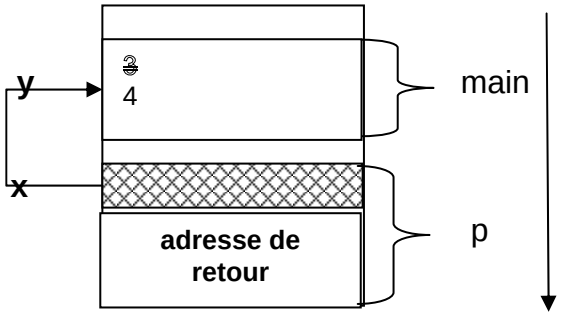
1. **Faire une fonction qui échange les valeurs de ces deux paramètres entiers. Montrer qu'elle marche bien en écrivant un programme qui lit deux nombres entiers et échange leurs valeurs.** La solution est montrée à la fig. 5.10. Noter bien que dans la fonction on utilise l'opérateur d'indirection pour dénoter les variables dont les adresses étaient passées. L'instruction **return** peut être omise dans ce cas parce que il n'y a pas une valeur à rendre et la fin de la fonction termine aussi son exécution.




```c
#include <stdio.h>
int pgcd( int ,int );
void main (void) {
  int x1, x2,r,d;
  do {
   printf ("tapez un nombre positif:");
       scanf("%d", &x1);
  } while (x1 <= 0);
  while(1){
   do {
   printf ("tapez un nombre positif:");
       r = scanf("%d", &x2);
    } while (x2 <= 0);
       if ( r<1) break;
       d = pgcd(x1,x2);
printf ("Le pgcd de %d et %d est %d\n",
                 x1,x2,d);
       x1 = x2;
      }
}
int pgcd( int a,int b){
  int da, db, t;
            da = a; db = b;
  do{
        t = da % db;
        da = db;
        db = t;
  }while (t!=0);
      return da;
}
```

Fig,5.8 Le pgcd d'une suite de pairs

```c
#include <stdio.h>
void p(int *x){
   *x+=1 ;
}
  void(main(void){
  int y=3 ;
  p(&y) ;
  printf("%d\n",y) ;
}
```




Fig,5.9 Paramètre de type pointeur

```c
#include <stdio.h>
void echange( int * ,int * );
void main (void) {
  int x1, x2;
  printf ("tapez deux nombres entiers:");
  scanf("%d%d", &x1, &x2);
  printf ("Avant l'echange x1= %d et x2 = %d \n",x1,x2);
  echange(&x1,&x2);
  printf ("Apres l'echange x1= %d et x2 = %d \n",x1,x2);
}
void echange( int * a,int *b){
  int  t;
  t = *a ;
  *a = *b;
  *b = t;
   return ;
}
```

Fig,5.10 L'exemple échange

2. On peut noter que dans la fonction main de l'exemple pgcd nous avons utilisé deux fois les mêmes instructions pour lire x1 et x2 et de vérifier que le nombre soit positif. On pet faire une fonction qui lit un nombre et vérifie qu'il soit positif. Le paramètre doit être un pointeur car quand on lit la variable obtienne une nouvelle valeur. On peut éecrire d'autre côté une fonction qui lit un nombre entier positif et renvoie sa valeur comme résultat. Les deux approches sont montrées aux la fig 5.11a et fig.5.11b Le résultat de la fonction lirepositif de 5.11a est le résultat du scanf qui présent le nombre des nombres lus. Le résultat égal à zéro signifie que la fin du fichier était atteinte. Le résultat de la fonction lirepositif du programme de 5.11b est la valeur du nombre lu. Si la fin du fichier est atteinte le résultat est -1.

```
#include <stdio.h>
int pgcd( int ,int );
int lirepositif(int *);
void main (void) {
  int x1, x2,r,d;
  lirepositif(&x1);
  while(1){
   r=lirepositif(&x2);
   if ( r<1) break;
   d = pgcd(x1,x2);
   printf ("Le pgcd de %d et %d est
%d\n", x1,x2,d);
    x1 = x2;
 }
}
int lirepositif(int *x) {
  int r;
  do {
    printf ("tapez un nombre positif:");
    r = scanf("%d", x);
  } while (*x <= 0);
  return r;

}
int pgcd( int a,int b){
  int da, db, t;
  da = a; db = b;
  do{
        t = da % db;
        da = db;
        db = t;
  }while (t!=0);
      return da;
}
```

Fig,5.11a pgcd1

```
#include <stdio.h>
int pgcd( int ,int );
int lirepositif(void);
void main (void) {
  int x1, x2,d;
  x1 = lirepositif();
  while(1){
  x2=lirepositif();
       if ( x2<0) break;
       d = pgcd(x1,x2);
       printf ("Le pgcd de %d et %d est
%d\n",x1,x2,d);
       x1 = x2;
      }
}
int lirepositif(void) {
  int r,x;
  do {
       printf ("tapez un nombre
positif:");
       r = scanf("%d", &x);
  } while (x <= 0);
  if (r>0)  return x;
       else return -1;
}
int pgcd( int a,int b){
  int da, db, t;
            da = a; db = b;
  do{
        t = da % db;
        da = db;
        db = t;
  }while (t!=0);
      return da;
}
```

Fig,5.11b pgcd2

## *Classes d'allocation (classes de mémoire)*

Lors de l'exécution d'un programme C il y a trois zones de mémoire différentes, correspondant aux trois durées de vies possibles :

- la zone contenant les variables statiques – ils sont allouées aux début du programmes et sont libérées à sa fin.
- la zone contenant les variables automatiques (cette zone est gérée en *pile*);
- la zone contenant les variables dynamiques (cette zone est généralement appelée le *tas*).

## Blocs et types de variables

- Un bloc en C ce sont les instructions renfermées entre accolades {}. On peut voir que chaque fonction présent un block. Dans chaque block on peut imbriquer autres blocks. Mais dans une fonction on ne peut pas imbriquer une autre fonction.

```
{
  déclarations
  instructions exécutables
}
```

Fig,5.12 Structure du bloc

La structure d'un bloc est montrée à fig. 5.12.

Les déclarations de variables peuvent se trouver :

- en dehors de toute fonction, il s'agit alors de variables globales – ils sont statiques
- à l'intérieur d'un bloc, il s'agit alors de *variables locales* ;
- dans l'en-tête d'une fonction, il s'agit alors d'*arguments formels*,

## Visibilité des variables

La question de la visibilité des identificateurs (c'est-à-dire « quels sont les identificateurs auxquels on peut faire référence en un point d'un programme ? ») est réglée en C comme dans la plupart des langages comportant la structure de bloc, avec une simplification : les fonctions ne peuvent pas être imbriquées les unes dans les autres, et une complication : tout bloc peut comporter ses propres définitions de variables locales.

**Variables locales.** Tout bloc peut comporter un ensemble de déclarations de variables, qui sont alors dites *locales* au bloc en question. Une variable locale ne peut être référencée que depuis l'intérieur du bloc où elle est définie ; en aucun cas on ne peut y faire référence depuis un point extérieur à ce bloc. Dans le bloc où il est déclaré, le nom d'une variable locale ***masque*** toute variable de même nom définie dans un bloc englobant le bloc en question.

Toutes les déclarations de variables locales à un bloc doivent être écrites au début du bloc, avant la première instruction.

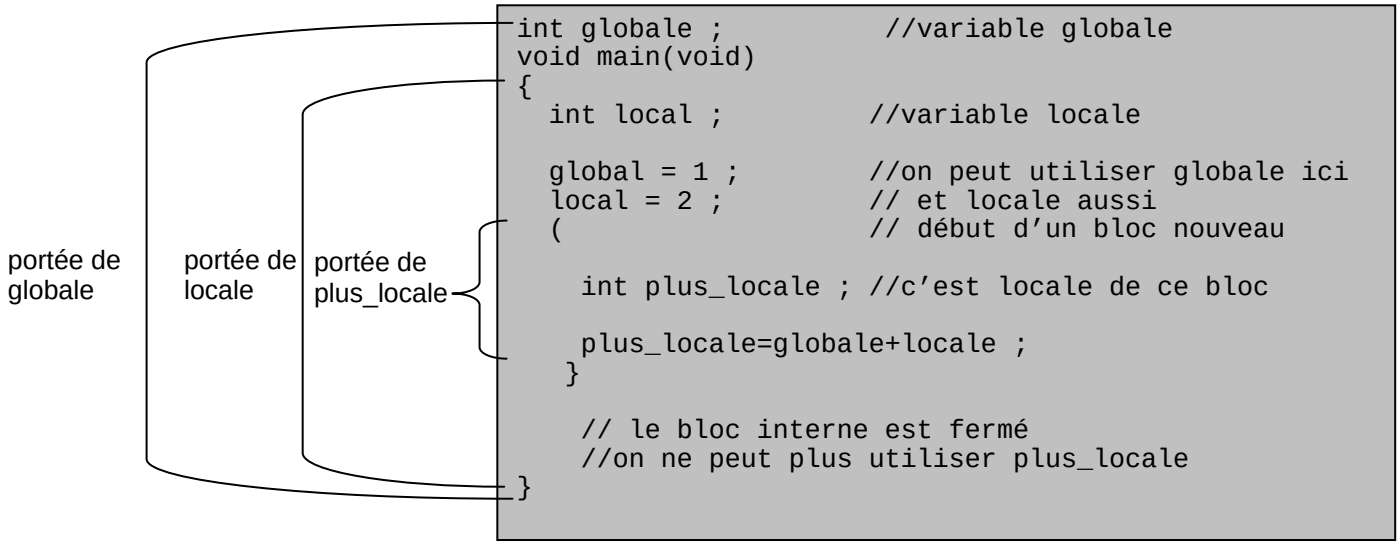
**Arguments formels.** Pour ce qui concerne leur visibilité, les arguments formels des fonctions sont considérés comme des variables locales du niveau le plus haut, c'est-à-dire des variables déclarées au début du bloc le plus extérieur9. Un argument formel est accessible de l'intérieur de la fonction, partout oµu une variable locale plus profonde ne le masque pas. En aucun cas on ne peut y faire référence depuis l'extérieur de la fonction (fig.5.14).

**Variables globales.** Le nom d'une variable globale ou d'une fonction peut être utilisé depuis n'importe quel point compris entre sa déclaration (pour une fonction : la fin de la déclaration de son en-tête) et la fin du fichier où la déclaration figure, sous réserve de ne pas être masquée par une variable locale ou un argument formel de même nom.

La question de la visibilité inter-fichiers sera examinée plus tard. On peut noter d'ores et déjà qu'elle ne se pose que pour les variables globales et les fonctions, et qu'elle concerne l'édition de liens, non la compilation, car le compilateur ne traduit qu'un fichier source à la fois et, pendant la traduction d'un fichier, il ne « voit » pas les autres.

## Allocation et durée de vie des variables

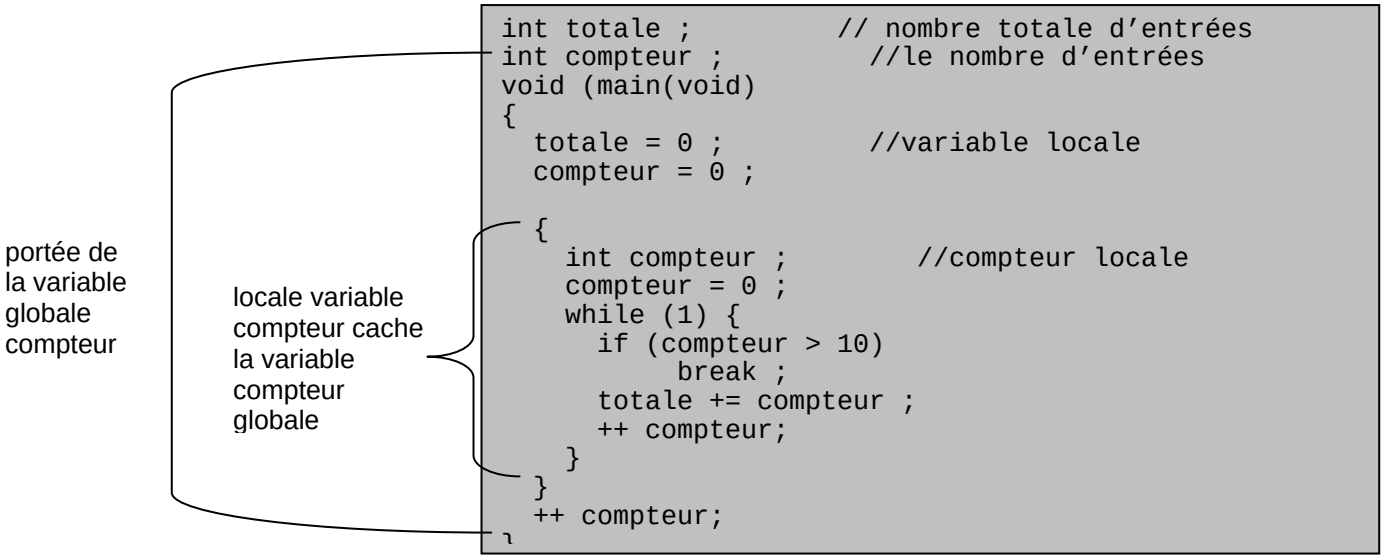
Les variables globales sont toujours *statiques*, c'est-à-dire permanentes : elles existent pendant toute la durée de l'exécution. Le système d'exploitation se charge, immédiatement avant l'activation du programme, de les allouer dans un espace mémoire de taille adéquate, éventuellement garni de valeurs initiales.




Fig,5.13. Variables globales et variables locales

A l'opposé, les variables locales et les arguments formels des fonctions sont *automatiques* : l'espace correspondant est alloué lors de l'activation de la fonction ou du bloc en question et il est rendu au système lorsque le contrôle quitte cette fonction ou ce bloc. Certains qualifieurs (static, register) permettent de modifier l'allocation et la durée de vie des variables locales.

On note une grande similitude entre les variables locales et les arguments formels des fonctions : ils ont la même visibilité et la même durée de vie. En réalité c'est presque la même chose : les arguments formels sont de vraies variables locales avec l'unique particularité d'être automatiquement initialisés (par les valeurs des arguments effectifs) lors de l'activation de la fonction.




Fig,5.14. Variables masquées (cachées)

## Classes de mémoire

On a vu que par défaut les variables globales sont *statiques* et leur durée de vie est la durée de l'exécution du programme. Les variables locales (définies en blocs et les paramètres des fonctions) sont automatiques, elles sont crées dans la pile chaque fois que l'exécution du bloc commence et elles sont détruites à la sortie du bloc. Mais on peut définir explicitement la classe de la mémoire, notamment statique par l'attribut **static.**

- variable locale – la portée de cette variable est le bloc où elle est définie mais sa durée de vie et toute l'exécution.

```
static int p ;
```

à chaque entrée dans le bloc la valeur de la variable statique sera cette valeur que la variable avait obtenue à la fin de l'exécution précédente du même bloc.

- variable globale – pour cette variable la différence est qu'elle est locale pour le fichier où elle est définie. (cette question sera examinée le semestre suivant).

### Exemples

1. Un exemple simple qui vous illustre la différence entre la variable locale automatique et la variable locale statique (fiq.5.15)

```
#include <stdio.h>
int main() {
  int counter; /* compteur d'itérations */
  for (counter = 0; counter < 3; ++counter) {
       int temporary = 1; /*  variable temporelle*/
       static int permanent = 1; /* variable permanente */
       printf("Temporelle %d Permanente %d\n",
       temporary, permanent);
       ++temporary;
       ++permanent;
  }
  return (0);
}
```

Fig,5.15. Variables automatiques et statiques

2. Faire une fonction qui compte le nombre de ses appels (fig. 5.16).

### Initialisation des variables

Chaque variable peut être initialisée dans l'instruction de définition par une expression qui dépend de la classe de mémoire

```
int p = 3;
```

```
#include <stdio.h>
int appels(void){
  static int app;
  app++;
  return app;
}

int main(void) {
  int counter; /* compteur d'itérations */
  for (counter = 0; counter < 3; ++counter) {
       printf ("appel No %d \n", appels());
  }
  printf("Quel est ce nombre ? %d\n", appels()-appels());
  return (0);
}
```

Fig,5.16. Compteur d'appels

- les variables statiques sont initialisées avant l'exécution du programme. Donc l'expression doit être une expression constante, qui peut être calculée par le compilateur. Par défaut elles sont initialisées par zéro.
- les variables automatiques sont initialisées au moment d'entée dans le bloc et leur initialisation est une simple affectation. Par défaut (s'il n'y a pas une initialisation explicite) la valeur d'une variable automatique n'est pas définie. Elle prend la valeur existante de l'endroit de la mémoire qui lui est alloué.

Exemples

```
#include <stdio.h>
int appels(int p1){
  static int app = 2; // expression contante;
  int loc = p1 +1;   // initialisation d'une variable automatique
  static int st1 = loc-1; // erreur – l'expression n'est pas constante
  static int j = N;  // expression contante;  app++;
  return app;
}

int main(void) {
  int counter ; /* compteur d'itérations  ici sa valeur n'est pas définie*/
  for (counter = 0; counter < 3; ++counter) {
       printf ("appel No %d \n", appels());
  }
  printf("Quel est ce nombre ? %d\n", appels()-appels());
```

Fig,5.16. Compteur d'appels

**Tableau récapitulatif**

| Type de variable | Déclaration | Portée | Classe de mémoire | Initialisation |
|---|---|---|---|---|
| Globale | En dehors de toute fonction | • la partie du fichier source suivant sa déclaration<br>• n'importe quel fichier source avec **extern** | Statique | Implicite – 0<br>Explicite – expression constante |
| Globale cachée | En dehors de toute fonction, avec attribut **static** | la partie du fichier source suivant sa déclaration | | |
| Locale « remanente » | Au début d'un bloc avec attribut **static** | Le bloc | | |
| Locale à une fonction | Au début d'une fonction | La fonction | Automatique | Implicite – il n'y a pas<br>Explicite - Expression |
| Locale à un bloc | Au début d'un bloc | Le bloc | | |

## L’attribut const

Une variable dont le type est qualifiée par **const** ne peut pas être modifiée. Le programmeur entend ainsi se protéger contre une erreur de programmation. Ceci est utile pour les paramètres d'une fonction, lorsqu'on désire protéger les paramètres effectifs de la fonction en les mettant en « lecture seulement » pour la fonction ou de définir une constante par une méthode alternative de #define.

```
#include <stdio.h>
const int N=19;

int appels(const int M){
  static int app =M; //ici on peut initialiser – M est une constante
  int i;
  for (i=0; i<M; i++) printf("*");
  M++; // erreur on ne peut pas modifier une constante
  printf("\n");
  app++;
  return app;
}

int main(void) {
  N++; // erreur on ne peut pas modifier une constante
  printf("Quel est ce nombre ? %d\n", appels(3)-appels(5));
  return (0);
}
```

Si la variable est de type pointeur la déclaration suivante déclare comme constante la valeur désignée par le pointeur.

```
int i=2;
int const * p = &i;
```

Et celle ci-dessous – un pointeur constant qui ne peut pas changer l’adresse qu’il désigne.

```
int i;
int  * const p = &i;
```