Langage C

Les fonctions

L2 Mathématique et Informatique

Université de Marne-la-Vallée

Syntaxe

La définition d'une fonction

```
1 typeRetour nomFt(type1 nom1,type2 nom2,typen nomn){
2 instructions
3 }
```

- `nomFt` est le nom de la fonction;
- `nom1,nom2,nomn` sont les paramètres de la fonction. Ce sont des variables locales à la fonction (inconnues ailleurs que dans le bloc de la fontion). Elles sont créées à l'appel et prennent la valeur des arguments fournis à l'appel. Si la fonction n'a pas de paramètre la liste contient `void`
- le bloc de la fonction peut contenir ses propres déclarations de variables (locales);
- Une seule valeur de type `typeRetour` peut être renvoyée par la fonction. Si la fonction ne renvoie pas de valeur, `typeRetour` devra être `void`

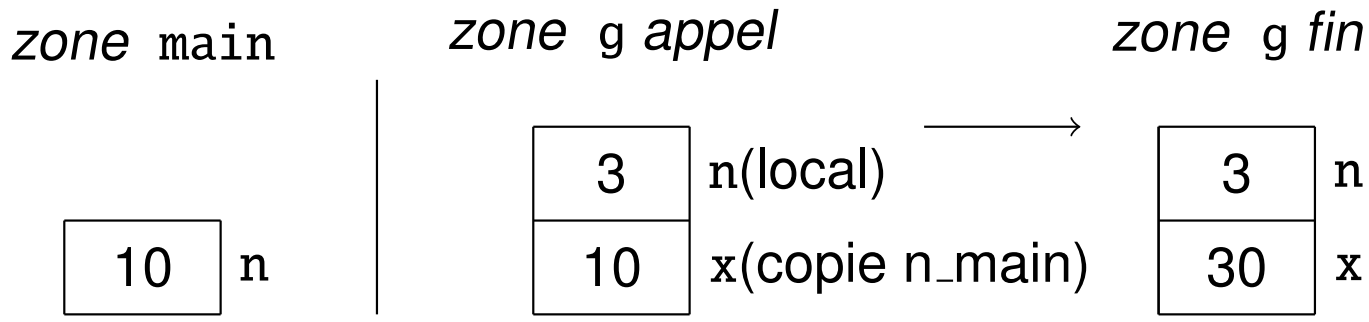
Comme tout élément manipulé par le programme, une fonction doit avoir été déclarée avant d'être utilisée.

Appel de fonction

- l'exécution de la fonction est lancée par le nom de la fonction suivi de parenthèses contenant la liste des arguments.
- les arguments sont des expressions dont la valeur est transmise
- si la fonction n'a pas de paramètre les parenthèses ne contiennent rien `f()`

```
1  #include <stdio.h>
2  int f(void){
3    return 5;
4  }
5  int g(int x){
6    int n=3;
7    x=n*x;
8    return x;
9  }
10 int main(void){
11   int n=10;
12   printf("f() renvoie %d\n",f());
13   printf("g(%d) renvoie %d\n",n,g(n));
14   printf("dans main n=%d\n",n);
15   return 0;
16 }
```

en mémoire




```
1 f() renvoie 5
2 g(10) renvoie 30
3 dans main n=10
```

Changer la valeur d'une variable par une fonction

Une fonction travaille toujours sur des copies des arguments qui lui ont été fournit.Ces arguments ne sont donc jamais changés par la fonction.
Toutes les transmissions de paramètres à des variables sont des **transmissions par valeur**
Mais on peut cependant changer la valeur d'une variable à l'aide d'une fonction.
C'est ce que fait `scanf`.
On peut fournir à une fonction l'adresse d'une variable.
On demande alors à la fonction de changer ce qu'il y a à l'adresse qui lui a été fournie.

Utilisation de l'adresse

- l'opérateur & fournit l'adresse d'un variable;
- L'opérateur * permet de
  - dans une déclaration, définir un élément de type *adresse de type*

```
1        int *truc; /* truc est de type adresse d'un int
2        double *machin;/* machin est de type adresse d'
3        /*truc et machin ne sont pas initialises*/
```

- dans une expression, accèder à ce qu'il y a à une adresse:

```
1        *p=5;/*met la valeur 5 a l'adresse p*/
2        a=*p+1 /* a vous*/
3        /* si p est une adresse valide!*/
```

```c
#include<stdio.h>
int main(void){
   int x;
   int *p;/* p est une variable de type adresse d'int*/
   p=&x;
   x=3;
   printf("␣adresse␣de␣x␣en␣hexadecimal␣:%p,%p\n",&x,p);
   printf("␣valeur␣de␣x=%d=%d\n'",x,*p);
   *p=5; /*mettre 5 a l'adresse p (donc la ou est x)*/
   printf("␣valeur␣de␣x=%d=%d\n'",x,*p);
   return 0;
}
```

une sortie possible

```
 adresse de x en hexadecimal:0xbfa7ea48,0xbfa7ea48
 valeur de x=3=3
 valeur de x=5=5
```

%p permet d'afficher une adresse en base 16. La valeur de l'adresse dépend de l'exécution.

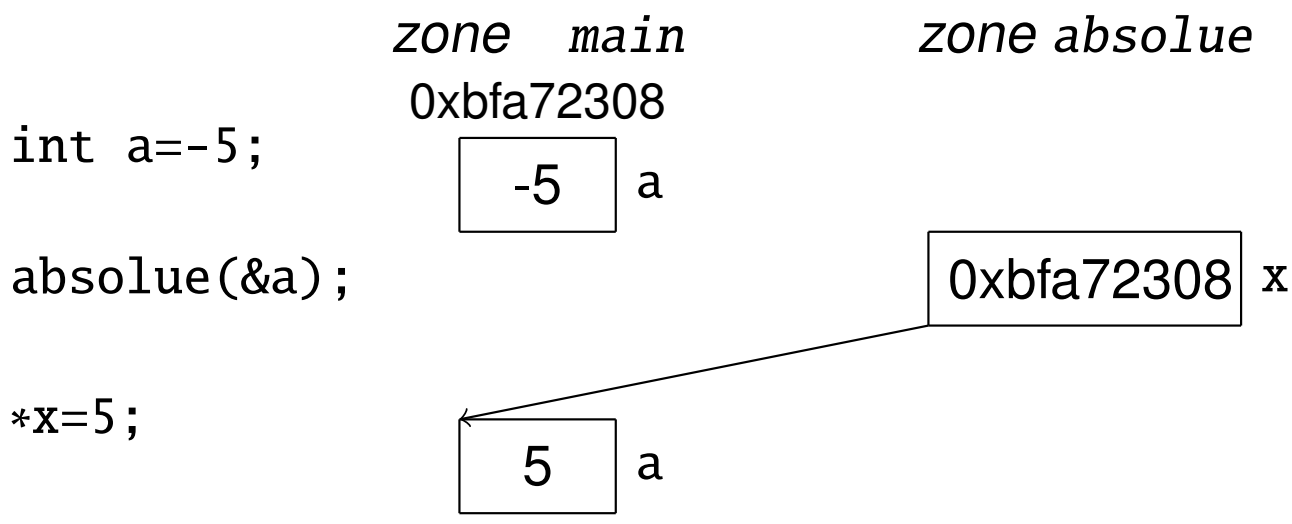
Utilisation par une fonction transmission par adresse

On transmet à la fonction l'adresse de la variable à manipuler. La fonction agit à l'adresse indiquée et donc transforme ce qui est à cette adresse.

```
1  #include<stdio.h>
2  /* transforme l'entier a l'adresse x en sa valeur absolue*/
3  void absolue(int *x){
4     if (*x < 0)
5     *x=-*x;
6  }
7  main(void){
8     int a=-5;
9     printf("␣valeur␣de␣a␣avant:␣%d␣,"a);
10    absolue(&a);
11    printf("␣apres␣:␣%d\n",a);
12    return 0;
13 }
```

résultat

```
1   valeur de a avant: -5 ,apres: 5
```


zone main
zone absolue
0xbfa72308
int a=-5;
-5
a
absolue(&a);
0xbfa72308
x
*x=5;
5
a

transmission par adresse

On parle de transmission par adresse, ce qui est un abus de langage.
Toute transmission de paramètre en C est une transmission par valeur.
On a passé la valeur de l'adresse d'un élément ce qui nous a donné, indirectement, accès à cet élément.
Fonction d'échange de la valeur de 2 `int`

```
1  void echangeInt(int *a,int *b){
2     int tmp;
3     tmp=*a;
4     *a=*b;
5     *b=tmp;
6  }
```

appel

```
1  int x,y;
2  ...
3  echangeInt(&x,&y);
```

Et pour échanger des variables de type `double`?

Transmettre plusieurs valeurs

On veut écrire une fonction qui calcule les racines réelles d'une équation du secont degré. Une fonction C ne peut renvoyer qu'une seule valeur (de type `void`, numérique, adresse, structure, ...).
On fournit donc à la fonction l'adresse des deux variables où placer le résultat. La valeur de retour indiquera à la partie appellante le nombre de racines affectée.

```
1  /* BUT : resoud une equation du second degre    */
2  /* PARAMETRES : valeur :les coefficients entiers*/
3  /*              adresse : 2 solutions possibles */
4  /* RETOUR : le nombre de solution               */
5  /* -1 :equation degeneree, infinite de solution */
6  /* 0 :pas de solution                           */
7  /* 1: un racine placee a l'adresse x            */
8  /* 2: deux racines placees aux adresses x et y  */
9
10 int resoudre(int a,int b,int c,double *x,double *y){
11 ...
12 }
```

Utilisation de la valeur de retour comme code d'erreur

Très utilisé dans les fonctions systèmes.

- si le résultat correct appartient à un ensemble bien défini, on peut en cas d'erreur renvoyer une valeur qui n'est pas dans cet ensemble

```
1  /* BUT : calcule la factorielle          */
2  /* PARAMETRES : un entier positif n      */
3  /*                                       */
4  /* RETOUR: n! ou                         */
5  /*         0 si n<0 ou n! trop grand     */
6  int fact(int n){
7  ...
```

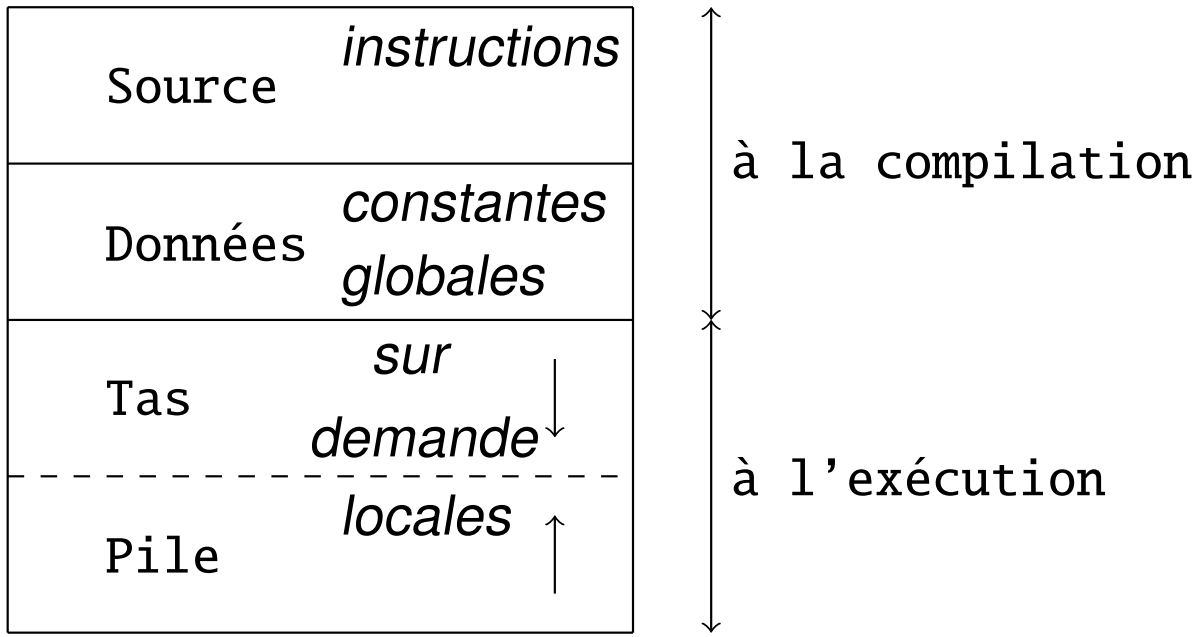
- Si le résultat correct peut prendre toutes les valeurs possibles du type, on transmet l'adresse où placer le résultat et la valeur de retour ne sert quà indiquer le bon fonctionnement

```
1  /* BUT          :                                          */
2  /* calcule la puissance entiere                            */
3  /* PARAMETRES:                                             */
4  /*    un entier n : la base                                */
5  /*    un entier positif p :l'exposant                      */
6  /*    une adresse d'entier puis : le resultat              */
7  /* RETOUR       :                                          */
8  /*     1 si p >=0 0 sinon                                  */
9  /* METHODE :                                               */
10 /*     exponentielle rapide en lg(p)                       */
11 /* ********************************************************* */
12 int exponantielle(int n, int p,int * puis );
```

On a ici donné le prototype de la fonction qui permet aux compilateur de vérifier que les appels sont correctement écrits. La définition doit être effectuée ailleurs.

Comment ça marche

Le compilateur traduit le programme C en suite d'instructions en langage machine. Ce sont des instructions très simples exécutables par le processeur.
Lorsque l'on lance un programme, le système réserve une zone mémoire pour le processus créé. Cette zone est formée de 4 parties



Gestion de la pile

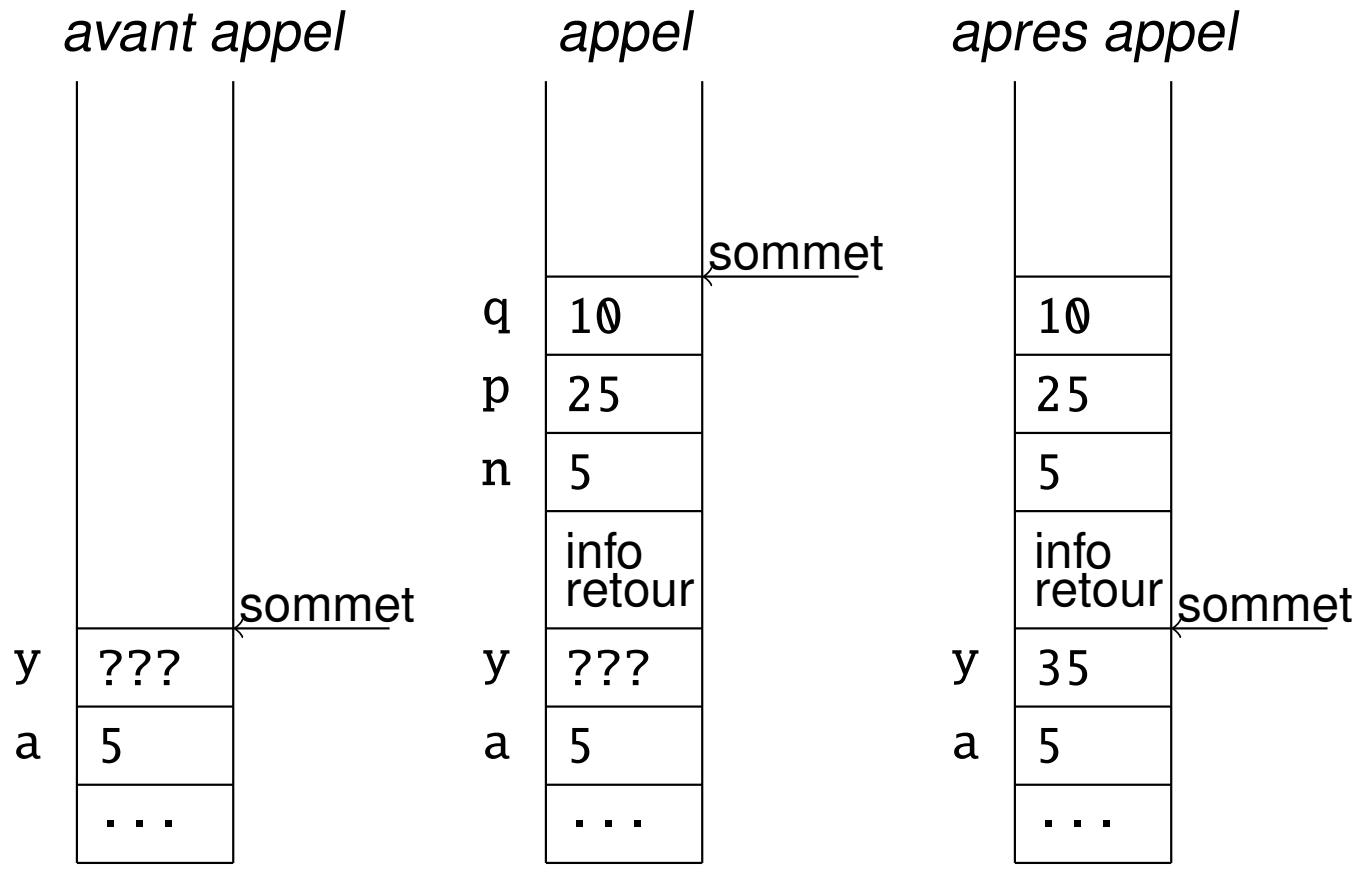
Lorsqu'une variable de bloc (variable locale) est déclarée, elle est allouée sur la pile. Le sommet de pile augmente.
A la sortie du bloc, le sommet redescend, les variables ne sont plus accessibles.
Une autre variable locale réutilisera peut-être le même emplacement

LES VARIABLES LOCALES DOIVENT ÊTRE INITIALISÉS
LEUR ADRESSE NE DOIT PAS ÊTRE RENVOYÉE.

Gestion du code source, appel de fonctions

Les instructions sont numérotées.
Le processeur dispose du numéro de l'instruction à exécuter.
Il analyse, exécute une instruction puis passe à la suivante.
Un appel de fonction correspond à un branchement dans le code source sur les instructions de cette fonction. Lorsque l'exécution de la fonction est terminée, le processeur doit revenir à l'instruction ayant effectué l'appel.
Au moment d'un appel de fonction, toutes les informations nécessaires au retour sont placées sur la pile.
Les valeurs des arguments de la fonction sont empilés.
On empile ensuite les variables locales au bloc de la fonction.

```
int f(int n){
 int p,q;
p =n*n;
 q=2*n;
 return p+q;
}
...
{
int a=5,y;
y=f(a);
```



Récursivité

- Pour le système, un appel récursif est un appel de fonction comme un autre;
- cela prend de la place sur la pile d'exécution;
- pas de limitation logicielle au nombre d'appels: sans condition d'arrêt `erreur de segmentation`;
- ça facilite la vie du programmeur.

Conseils

- Une tache $\rightarrow$ une fonction;
- ne pas mélanger saisie, calcul et affichage (sauf triviaux);
- donner des noms significatifs;
- une fonction courte est plus simple à lire et à vérifier;
- commentez (commencez avant même d'écrire le code)
  - BUT,
  - PARAMETRES,
  - RETOUR,
  - EFFET DE BORD,
  - METHODE,
  - ERREURS,
  - ...

Gestion des erreurs

- sortir d'une fonction dès qu'une erreur est détectée évite des `else` inutiles.
- la valeur de retour choisie pour indiquer l'erreur ne doit pas partie des valeurs valides, préférez un passage par adresse du résultat:

```
1 int quotient(int a,int b,int *quotient){
2   if(b==0)
3     return 0;
4   *quotient=a/b;
5   return 1;
6 }
```

- éviter l'interruption du programme `exit`($\neq 0$);
- remonter la valeur d'erreur jusqu'à la partie pouvant la traiter, par exemple en sauvegardant les résultats partiels avant de passer au traitement suivant;
- Pour une fonction récursive ne tester qu'une seule fois les erreurs (avec une fonction appelant la fonction récursive

```
1 int Hanoi(int n){
2   if(n<1 || n>40)
3     return 0;
4   Hanoi_aux(n);
```