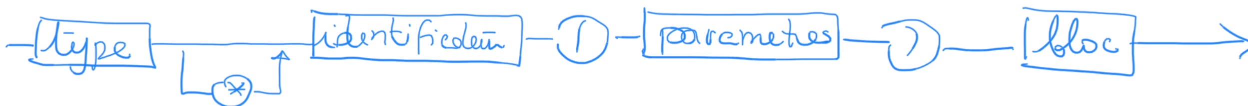
Chapitre 4 : les fonctions

Objectif :
- Décomposer une longue séquence de traitement en un ensemble de petits traitements
- Fournir des éléments de base de haut niveau
  lisibilité
  facilité de maintenance
- Eviter la répétition de morceaux programmes identiques.

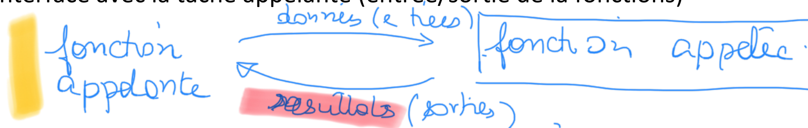
1) Déclaration et définition d'une fonction



type : type de la valeur retournée par la fonction
dans le cours de lo01 nous verrons principalement des fonctions qui retournent des types simples : (int, char, float, …

si pas de retour => void (procédure).
Identificateur : nom de la fonction, il doit décrire le type de traitement effectué par la fonction.

paramètres : liste de paramètres séparés par une ,
variables qui effectuent l'interface avec la tâche appelante (entrée/sortie de la fonctions)



bloc :
- déclarations des variables locales à la fonction
- Séquence d'instruction
  Les variables utilisées dans les instructions du bloc sont les variables locales et les paramètres de la fonction

Important : Une fonction doit être déclarée avant d'être utilisée => 2 solutions.

Déclaration et définition avant l'utilisation

```
int ma_fonction(…)
{/ * bloc * / }

int main( )
{
/* appel de ma fonction * /
}
```

Déclaration avant utilisation (prototype) et définition après

```
int ma_fonction();

void main ( )
{/* appel de ma fonction */ }

int ma fonction(…)
{ / * bloc * / }
```

Retour de l'appel :

Une fonction retourne une valeur à la tâche qui l'a appelée par l'instruction return qui doit être placée à la fin de la fonction.

=> `return expression ;`
le type de l'expression doit être identique au type de la fonction

Exemples :

```
return i;
return 0;
return 3*i+1;
```

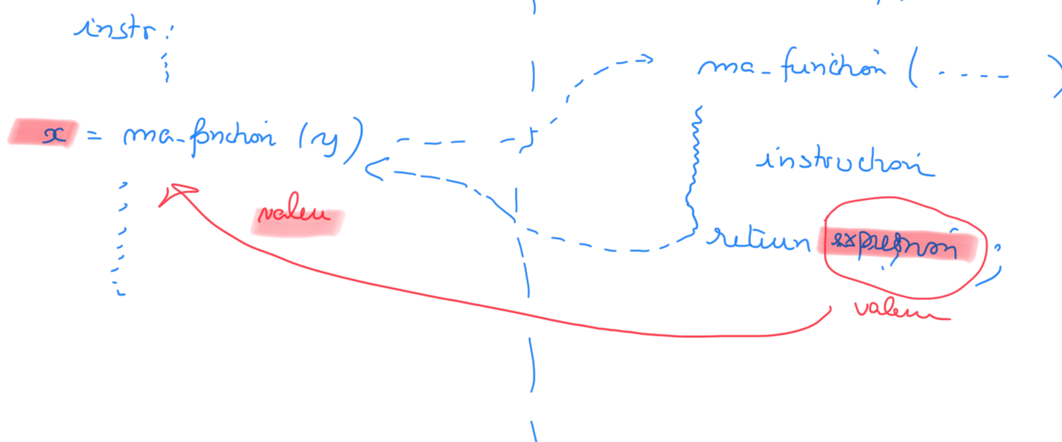
si void => `return ;`

2) Appel d'une fonction

Une fonction peut être appelée dans les instructions d'autres fonctions (ou par elle-même , vu plus tard).
Le nombre et le type des arguments doit correspondre au nombre et type de paramètres déclarés dans la fonction

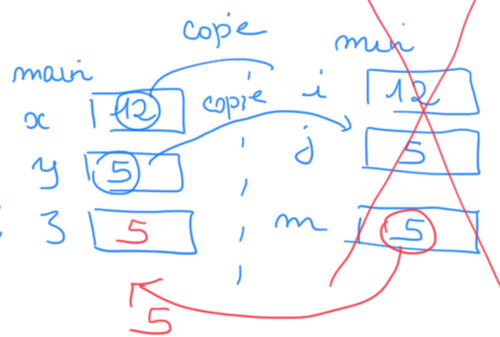
Si la fonction retourne une valeur d'un type différent de void alors l'appel doit être inséré dans une expression.
Exemple d'appel : fonctions prédéfinies dans math.h :

`x = sin(y)` ; fonction retournant un double
`x = pow(8,3)` => 8^3 fonction avec 2 paramètres
`printf(" i = " , i)` => fonction utilisée comme une procédure avec 2 arguments
`c = getchar()` => retourne 1 caractère sans argument.



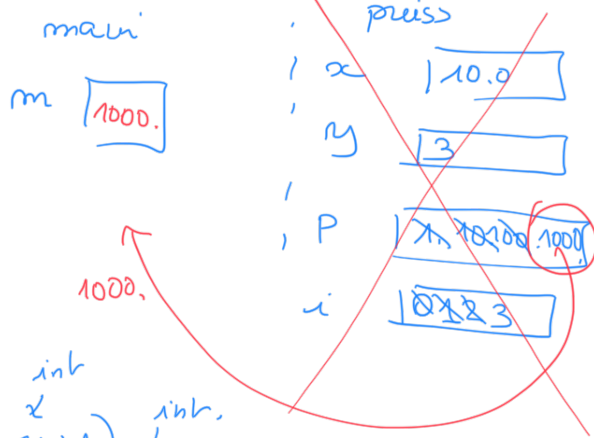
Exemples d'écritures de fonctions :

```
#include <stdio.h> / * minimum * /


int min(int i, int j)
{
        if (i < j)
                return i;
        else
                return j;

}

int main( )
{
        int x, y, z ;
        printf("entrez deux nombres : " ) ;
        scanf(" %d %d ", &x, &y) ;
        z = min(x, y) ;
}
```



Puissance d'un nombre positif ou nul.

```
double puiss(double x, int y)
{
        double p ;
        int i ;

        p = 1;
        for (i=0; i<y; i++)
                p = p * x;
        return p;

}

int main( )
{
        double m ;
        m = puis (10.5, 3) ;
        …
}
```



Les bibliothèques de fonctions prédéfinies :

Il existe de très nombreuses bibliothèques de fonctions prédéfinies en C. Inutile de ré-inventer la roue sauf pour des raisons liées à l'entraînement à la programmation.

exemples de bibliothèques :

`string.h` : pour les chaînes de caractères (abordées dans un autre chapitre) :
`strlen(chaine), strcpy(chaine1, chaine2), strcmp(chaine1, chaine2), strcat(chaine1, chaine2).`

math.h : fonctions mathématiques de bases
`exp(x), log(x), log10(x), pow(x,y)`
`sqrt(x), sin(x), cos(x), tan(x), asin(c), acos(x), atan(x), fabs(x).`

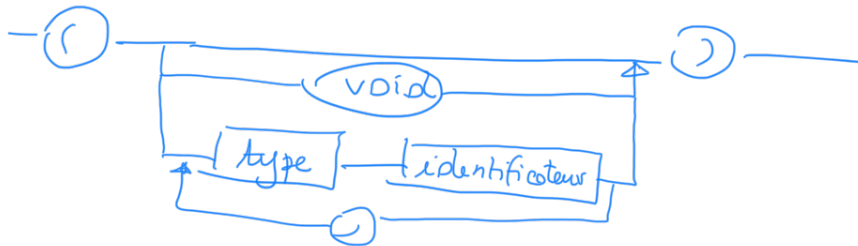
conversion : `atoi(carac), atol(carac), itoa(i)`
Test : `isalpha(c) , isupper(c), isascii(c), isdigit(c) , isprint(c)`

Conversion : `toupper(c) , tolower(c)`

3) Passage des paramètres

Les paramètres d'une fonction :

* Déclaration : liste de paramètres entre ( ) séparés par, liste vide => void.



* Appel : - liste d'arguments entre ( ) -> variables , expressions.
* Paramètre est un objet (variable) de la routine.

-> Il a une valeur initiale par le fait de l'appel qui copie la valeur de l'argument dans le paramètre

* ~~La déclaration de la variable~~ le paramètre est valable jusqu'à la fin du bloc de la fonction.
* La correspondance entre paramètres et arguments se fait suivant l'ordre dans la liste.

-> concordance de types
-> même nombre

Exemple :

```
void maFonction (int i, float k, char c)
{ ......}
```

Appel :
k est un entier , g est un float, i est un char

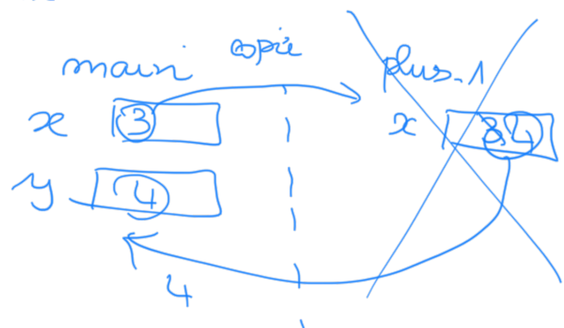
```
ma_fonction (k, g, i)

ma_fonction (-3, 12.78, 'e')
```

int float char

Passage de paramètres par valeur : (paramètre d'entrée)

- lors de l'appel -> copie de la valeur de l'argument dans le paramètre.
- exécution de la fonction -> dans le paramètre = valeur de l'argument. Le paramètre est une variable, on peut donc modifier sa valeur.
- retour à la routine appelante -> la valeur de l'argument n'est pas affectée par les modifications éventuelles du paramètre,
- utilisé lorsque le rôle du paramètre est une donnée qui n'a pas vocation à être modifiée par la fonction

```
int plus_1 (int x)
{ x = x+1; //modifie la valeur de x
  return x;
}
int main ()
{ int x, y;
  x=3
  y = plus_1 (x);
```



```
printf("x=%d, y=%d", x,y);
return 0;
}
```

Exemple :

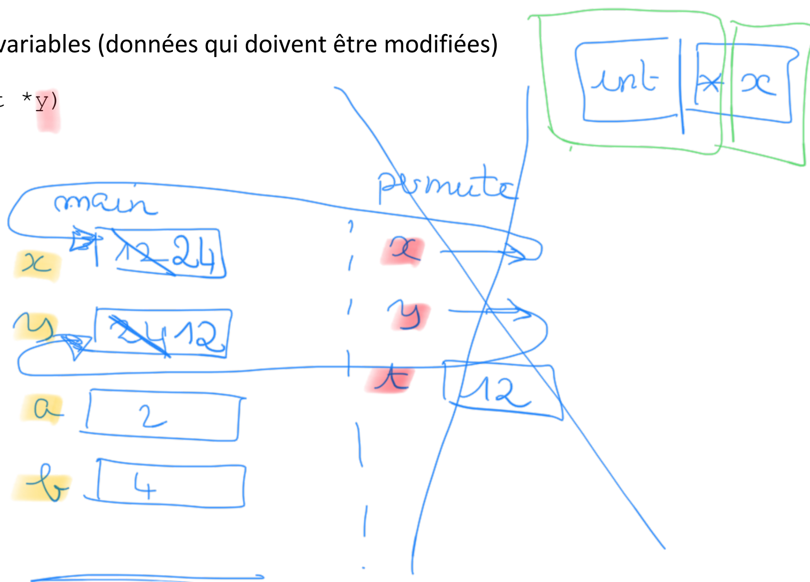
```
int plus_1 (int x)
{ x = x + 1  ;
return x ;
}


void main ( )
{
int x, y ;
x = 3 ;
y = 0 ;
y = plus (x) ;
printf(" x= %d  y=%d ",x, y);
…
```

Passage de paramètres par adresse : (par référence)

- transmettre l'adresse de l'argument de telle sorte qu'elle devienne la valeur du paramètre. Les mêmes règles que pour le passage par valeur sont appliquées.
- différence : le contenu de la mémoire à l'adresse indiquée peut-être modifié.
- utilisé lorsque le rôle de la fonction est de modifier l'état de ses paramètres : donnée qui doit être modifiée ou résultat de la fonction.

Exemple : permutation de deux variables (données qui doivent être modifiées)

```
void permute( int *x, int *y)
{ int t;

  t=*x ;
  *x= *y ;
  *y=t
  return;
}


int main()
{ int x, y, a, b ;

x = 12 ; y = 24 ;
a = 2 ; b = 4 ;
permute (&x, &y) ;
permute (&a, &b) ;
printf("x = %d , y = %d ", x,y);
}
```



Important : Lorsque la valeur du paramètre doit changer dans la fonction, il faut alors impérativement passer les paramètres par adresse.
Les modifications du contenu des objets désignés par leur adresse en paramètre ont des répercussions dans la routine appelante.

Exemple (à comparer avec l'exemple donné pour la passage par valeur:

```
int plus(int * x)
{
*x = *x + 1;
return (*x);
}

void main()
{
int x, y
 y = plus(&x) ;
printf(" x= %d  y=%d ",x, y);
}
```

Equivalence : résultat retourné par la fonction ou résultat passé en paramètre

résultat retourné par la fonction

```
typef f(…)
{
typef resultat ;

resultat = ……. . ;
return resultat ;
}
```

equivalent

résultat passé en paramètre

```
void calculef(……., typef *f)
{
typef resultat ;
…
resultat = …;
*f = resultat ;
return ;

}
```

Appel :

```
typef valeur ;
```

cas 1 `valeur =f(…) ;`

cas 2 `calculef(…, &valeur) ;`