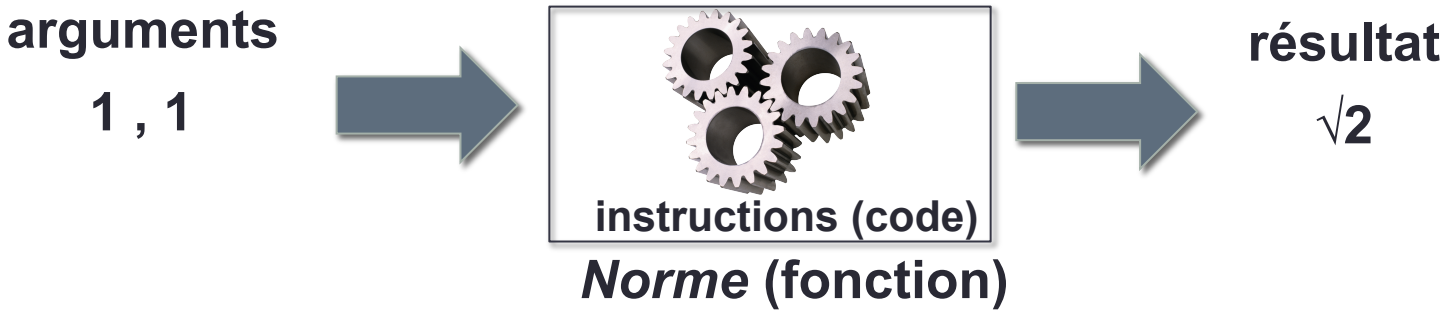
5. Fonctions et paramètres

- Fonctions
- Passage des arguments
- Passage de tableaux et de structures
- Récursivité
- La fonction main()

VERSION FINALE

Fonctions dans un programme C

- Une fonction : un élément du programme qui réalise un traitement des arguments (données) en entrée et renvoie un résultat



- Usage des fonctions :
  - Décomposition d'un problème en sous-problèmes plus simples

⬅➡ Décomposition d'un programme complexe en sous-programmes ou fonctionnalités plus simples

Fonction C : définition

- Définition d'une fonction :

```
type_retour nom_fonction ( type1 arg_1, type2 arg_2,… )
   {
       /* Déclaration variables locales */
       /* Instructions*
    }
```

- `type_retour` : le type de la fonction - type de la valeur retournée.
- **`nom_fonction`** : son identifiant (comme pour variables: pas d'accent, ni espace, peut pas commencer par un chiffre)
- `type1 arg_1,..` : les arguments formels ou paramètres formels de la fonctions avec leur type

Fonctions C : le type void

- **void** :
  - pour le type de retour - si la fonction ne retourne aucune valeur
  - pour les arguments - si la fonction ne prends pas d'arguments/paramètres

Exples:

```
int message( void )
    {  int i;  for (i=0; i<1;i++) printf("Ok, here. \n");
       return(1);
    }

void insiste (int n, char * texte)
   {
      int i;
      for (i=0; i < n; i++)  printf("%s \n", texte);
   }
```

Appel par `insiste(3,"Non!");` ou `char t[]="Non!"; insiste(4,t);`

Fonctions C : return

- **`return`** : instruction de retour à la fonction appelante
  avec en argument/paramètre la valeur à retourner

1. **`return(value);`** ou **`return value;`**
   avec **`value`**, la valeur à retourner du type annoncé pour la fonction
2. **`return ;`** (facultatif) sans argument pour une fonction de type void

- Fin d'exécution des instructions d'une fonction:
  - au premier **`return`** rencontré !
  - ou à la fin de sa définition (`}`)

Fonction C avec return

```
float  norme (int a , int b)
{
    float resultat ;
    if (a==0 && b==0)
            return(0);
    a=a*a;
    b=b*b;
    resultat = sqrt(a+b) ;
    return(resultat);
}
```

Arguments formels

Si condition vérifiée :

sortie de la fonction avec valeur 0.

Le code dessous n'est pas exécuté.

Retour final

Appel d'une fonction

- Appel d'une fonction par :

```
nom_fonction (arg1, arg2,…);
```

avec **arg1, arg2**: les arguments effectifs – des variables initialisées ou des valeurs constantes (2, 3.4, "toto",…)

La valeur renvoyée peut être :

- affectée à une variable

```
float x, int a=1;
x=norme(a, 2);
```

- utilisée en argument d'une autre fonction

```
printf("norme : %d", norme(2,a));
```

- utilisée dans un test

```
if ( norme (2,a) == 1.0 )  {
```

Passage de paramètres par valeur

- En C, arguments/paramètres sont toujours passés par *valeur* :

les arguments/paramètres formels d'une fonction sont initialisés avec une copie de la valeur des arguments effectifs.

Passage de paramètres par valeur

```
{
        int i = 3, k = 4;
        float res;
        . . .
        res = norme(k,i);
        . . .


}
```

0

```
float norme(int i, int j)
{
        float result;
        i * = i;          //i=i*i
        j * = j;
        return ( sqrt(i+j) );
}
```

Variables locales à la fonction appelante (rappel):

- création et destruction à l'entrée et sortie du bloc { }
- visibles que de l'intérieur du bloc

Passage de paramètres par valeur

```
{
        int i = 3, k = 4;
        float res;
        . . .
1       res = norme(k,i);
        . . .


}
```

```
float norme(int i, int j)
{
        float result;
        i * = i;          //i=i*i
        j * = j;
        return ( sqrt(i+j) );
}
```

1. Appel à la fonction:
   - création des variables **locales pour arguments**
   - copie des valeurs

Passage de paramètres par valeur

```
{
        int i = 3, k = 4;
        float res;
        . . .
        res = norme(k,i);
        . . .


}
```

2

```
float norme(int i, int j)
{
        float result;
        i * = i;          //i=i*i
        j * = j;
        return ( sqrt(i+j) );
}
```

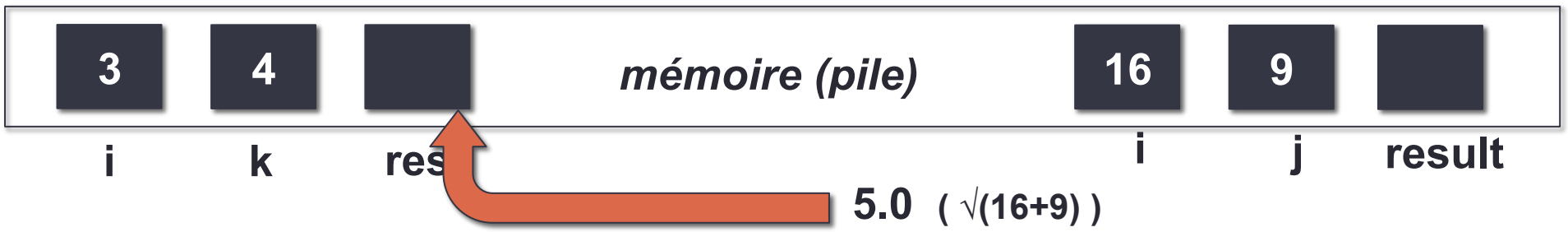
2. Exécution de la fonction:
   - création des variables **locales**
   - exécution des instructions ici (`i* = i;`)

Passage de paramètres par valeur

```
{
        int i = 3, k = 4;
        float res;
        . . .
        res = norme(k,i);
        . . .


}
```

```
float norme(int i, int j)
{
        float result;
        i * = i;          //i=i*i
        j * = j;
        return ( sqrt(i+j) );
}
```



3. Retour de la valeur:
   - copie de la valeur retournée vers la fonction appelante

Passage de paramètres par valeur

```
{
        int i = 3, k = 4;
        float res;
        . . .
        res = norme(k,i);
        . . .


}
```

4

```
float norme(int i, int j)
{
        float result;
        i * = i;          //i=i*i
        j * = j;
        return ( sqrt(i+j) );
}
```

4. Retour à la fonction appelante:
   - sortie de la fonction norme
   - **destruction** de ses variables locales

Passage de paramètres par valeur

- Avantages :
  - variables locales

    (arguments formels i et j invisibles en dehors de la fonction)

  - moins de variables locales

    (arguments i et j utilisés pour stocker des résultats de calculs intermédiaires)

  - protections des paramètres effectifs

    (i et k de la fonction appelante ne peuvent pas être modifiés)

- Inconvénients :
  - **une fonction ne peut modifier les paramètres effectifs**

    par passage par valeur

Passage de paramètres par valeur

- Que faire s'il faut modifier les arguments ?

Exemple:
```
void echange(int i, int j)
    {   int tmp;
        tmp = i ;
        i = j   ;
        j = tmp ;
    }
```

- Que se passe-t-il pour x et y dans un programme qui appelle `echange (x, y)` ?

Passage de paramètres par valeur

```
{
        int x = 1, y = 2;

        . . .
1       echange(x,y);
        . . .

}
```

```
void echange(int i, int j)
{
        int tmp;
        tmp = i ;
        i = j ;
        j = tmp ;
}
```

1. Appel à la fonction:
   - création des variables **locales pour arguments**
   - copie des valeurs

Passage de paramètres par valeur

```
{
        int x = 1, y = 2;

        . . .
        echange(x,y);
        . . .

}
```

2

```
void echange(int i, int j)
{
        int tmp;
        tmp = i ;
        i = j ;
        j = tmp ;
}
```

2. Exécution de la fonction:
   - création des variables **locales**
   - exécution des instructions ici (`tmp = i;`)

Passage de paramètres par valeur

```
{
        int x = 1, y = 2;

        . . .
        echange(x,y);
        . . .

}
```

3

```
void echange(int i, int j)
{
        int tmp;
        tmp = i ;
        i = j ;
        j = tmp ;
}
```

3. Retour de la valeur:

- i et j échangés
- pas de valeur à retourner (type void)

Passage de paramètres par valeur

```
{
        int x = 1, y = 2;

        . . .
        echange(x,y);
4       . . .

}
```

```
void echange(int i, int j)
{
        int tmp;
        tmp = i ;
        i = j ;
        j = tmp ;
}
```

4. Retour à la fonction appelante: - sortie de la fonction echange
   -destruction de ses variables locales

➢ **x et y n'ont pas été modifiés ni échangés**

Passage de paramètres par adresse

- Solution pour modifier les arguments effectifs avec une fonction :
  passer par **valeur** l'adresse de la variable

➢ la fonction initialise des pointeurs avec la copie de l'adresse

➢ et accède directement en mémoire aux paramètres effectifs par indirection sur les pointeurs

- On parle de **passage de paramètres par adresse**.

Exemple:

```
void echange(int *i, int *j)
{   int tmp;
    tmp = *i ;
    *i = *j  ;
    *j = tmp ;
}
```

Appel avec echange (&x, &y)

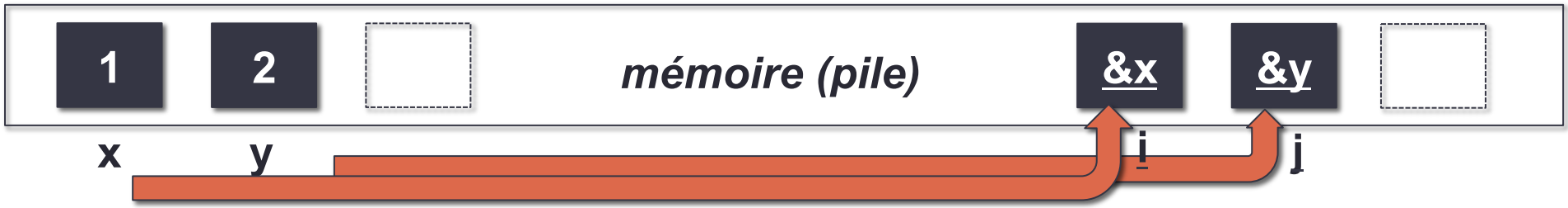
Passage de paramètres par valeur

```
{
        int x = 1, y = 2;

        . . .
1       echange(&x,&y);
        . . .

}
```

```
void echange(int *i, int *j)
{
        int tmp;
        tmp = *i ;
        *i = *j ;
        *j = tmp ;
}
```



1. Appel à la fonction: - création des variables (pointeurs) locales pour arguments
   - **copie des valeurs des adresses**

Passage de paramètres par valeur

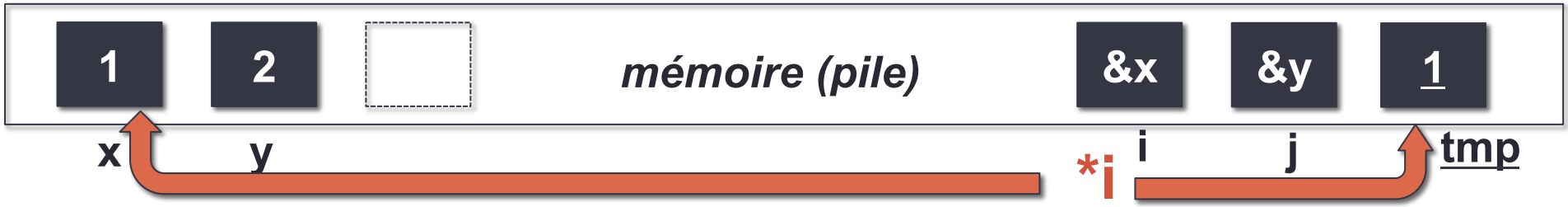
```
{
        int x = 1, y = 2;

        . . .
        echange(&x,&y);
        . . .

}
```

```
void echange(int *i, int *j)
{
        int tmp;
        tmp = *i ;
        *i = *j ;
        *j = tmp ;
}
```

2



2. Exécution de la fonction:
   - création des variables locales (tmp)
   - `tmp = *i;`

Passage de paramètres par valeur

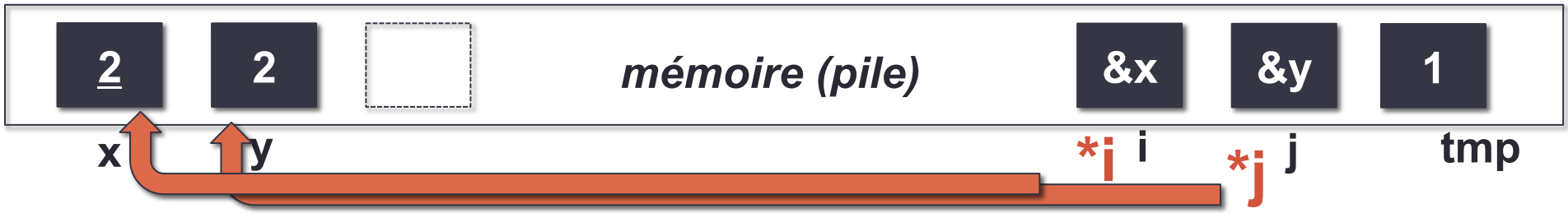
```
{
        int x = 1, y = 2;

        . . .
        echange(&x,&y);
        . . .

}
```

```
void echange(int *i, int *j)
{
        int tmp;
        tmp = *i ;
        *i = *j ;
        *j = tmp ;
}
```

2



2. Exécution de la fonction: - `*i = *j;`

(affecte au contenu de l'adresse pointée par **i**, le contenu de l'adresse pointée par **j**)

Passage de paramètres par adresse

```
{
        int x = 1, y = 2;

        . . .
        echange(&x,&y);
        . . .

}
```

4

```
void echange(int *i, int *j)
{
        int tmp;
        tmp = *i ;
        *i = *j ;
        *j = tmp ;
}
```

3. et 4. Retour valeur et fonction :

-pas de valeur à retourner
- destruction variables locales de échange

➢ **x et y ont vu leur valeur échangée**

En Résumé:

- Passage par valeur des paramètres :
  - `type function (type i)`
  - paramètre effectifs (passé) **non modifiable** par la fonction (copie des valeurs)
  - ➢ variables locales et protection des paramètres effectifs

- Passage par adresse des paramètres :
  - `type function (type *i)`
  - paramètre effectif **modifiable** par la fonction en utilisant indirection (*) sur les pointeurs

Note: D'où l'utilité des pointeurs en plus de l'allocation dynamique ☺

- Passage d'un entier par valeur :

(variable **x** passée **non modifiable** par la fonction – copie de la valeur)

```
int x = 1;
fonction(x);
```

```
fonction(int i)
```

- Passage d'un entier par adresse :

(variable **x** passée **modifiable** par la fonction – copie de son adresse et utilisation de l'indirection (*)

```
int x = 1, *p_x=&x;
fonction(&x); ou  fonction (p_x);
```

```
fonction(int *i)
```

Tableaux et structures

- Tableaux
  - on ne passe pas la "valeur" d'un tableau
  - ➢ **Passage par adresse**
  - • on fournit l'adresse du premier élément (ou idem. le nom du tableau)
  - conseil : toujours fournir la taille
    sauf pour les chaînes de caractère où on a le “\0“.

Tableau et structure

```
int somme_elts ( int n, int* tab )
 {     int somme=0;                          /*initialisation à 0! */
       for (n-=1; n >= 0 ; n--)              /*utilise directement n ☺*/
           somme += tab[n];                  /* idem.  = *(tab+n)  */
       return (somme);
 }

int main(void)
{    int u[3]={1,-2,7};
     int sum_u;

     sum_u=somme_elts(3,u);                 /*  sum_u=somme_elts(3, &u[0] ) */
     return(0);
}
```

Tableaux et structures

- Structure
  - passer une structure par valeur possible mais pas recommandé
    - pas rapide
    - encombrement mémoire
  - **Passage par adresse**
    - On passe l'adresse de la structure
    - On privilégie alors l'utilisation de l'opérateur " -> " pour accéder au membre

Structure et fonction :Example

```
struct vect {int N; int x[3];};                    /* definition structure*/


int somme_elts ( struct vect *tab )
{     int somme=0;                                          /*initialisation !! */
        int n = tab -> N;                          /*attention de pas modifier N*/
       for (n-- ; n >= 0 ; n--)
         somme += tab -> x[n];
     return (somme);
}

int main(void)
{
     struct vect u = {3,{1,-2,7}};
     int sum_u;

     sum_u=somme_elts( &u );
     return(0);
}
```

Tableaux et structures

- **Si une fonction doit recevoir beaucoup de paramètres :**
  - lister les paramètres: fonction (int param1, int param2, char param3,…)
  - mettre ces paramètres dans une structure ou tableau
  - et utiliser un passage par adresse

- Si une fonction doit retourner beaucoup de paramètres:
  - passer une structure/tableau non initialisé/e par adresse est aussi une idée.

Récursivité

- Un programme est dit récursif quand il contient un appel à lui-même

- Exemple 1 :

```
void recursivite()
{ static count = 1 ;
  printf("%d\n",count++) ;
  recursivite();
}
```

Programmation récursive

test d'arrêt

```
if (count > 10) return;
```

- Exemple 2 :

```
int fact(int i)
{ if (i ==1)
    return(1);
  else
    return(fact(i-1)*i);
}
```

Récursivité

- Exemple 2 :

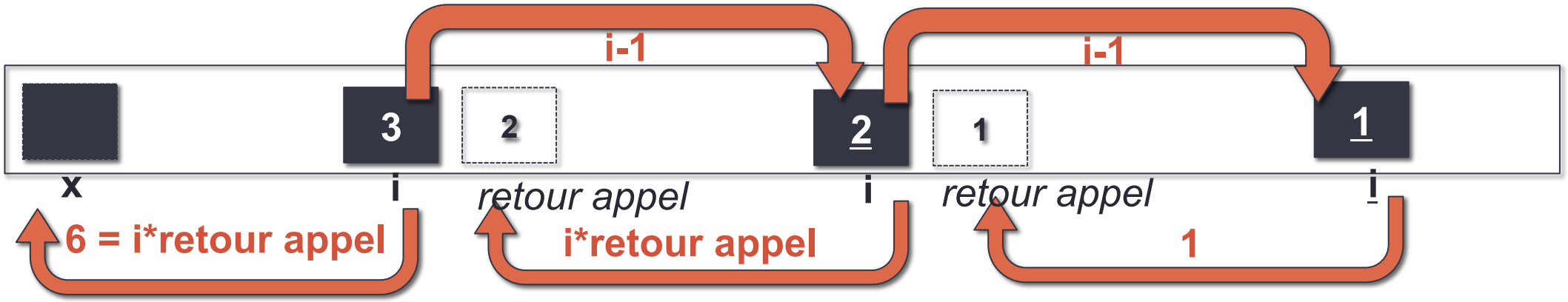
```
int fact(int i)
{ if (i ==1)
    return(1);
  else
    return(fact(i-1)*i);
}
```

```
x=fact(3);
```

```
fact(i)
{ ..
 return(fact(i-1)*i);
}
```

```
fact(i)
{ ..
 return(fact(i-1)*i);
}
```

```
fact(i)
{ if (i ==1 )
 return (1);..
}
```



Récursivité

- Esthétique

- Peut être un outil puissant
  - tours de Hanoï
  - tri rapide

- Mais version itérative de factorielle est ici meilleure :

```
f = 1 ;
for(i=1 ; i<=n ; i++) f *= i ;
```

  - gain mémoire
    - taille proportionnelle à n
  - gain en temps d'exécution
    - pas allocation de place sur la pile
    - pas mise à jour des pointeurs

Le main( ) : type de retour

- Le point d'entrée du programme, la fonction principale
- Type retourné au choix, dont `void`.

- En norme ANSI, retourne un **`int`** au système d'exploitation
  - si cette valeur est 0 = fin normale du programme pour le système
  - sinon une erreur

- Cette valeur peut être utilisée dans des scripts shell

(l'interpréteur de lignes de commande du terminal)

Exemple: *if my_program ; then halt; fi*

*(*si le programme réussi, éteindre le système)

La fonction Main : arguments

- Aucun : `Void` .
- Ou les arguments transmis par le système d'exploitation via la ligne de commande:

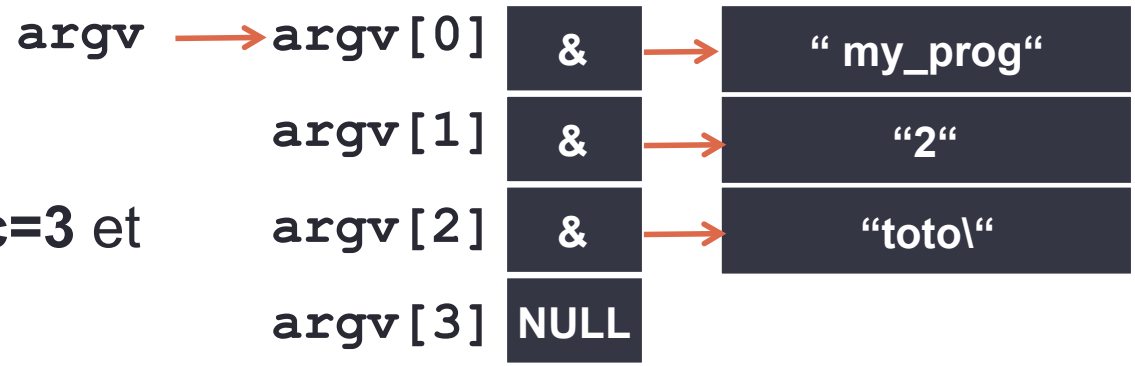
```
int main (int argc, char *argv[])
```

Avec argc : le nombre de mots sur la ligne de commande

argv : (de type char **) un tableau de pointeur sur chaque mot de la ligne de commande

- Exple : $> *my_prog 2 toto*

alors **argc=3** et



Exemple de main:

my_prog.c:

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  printf( "nombre arguments: %i\n",
                                argc);
  if (argc > 1 )
  printf("1er arg: %s (%i)",
            argv[1],atoi(argv[1]));

  return(0);
}
```

ligne de commande:

- *./my_prog*
- *nombre argument : 1*

- *./im_pair 2*
- *nombre arguments : 2*
- *1er arg: 2 (2)*

Note: **atoi()** convertit un chiffre (char) en nombre (entier)