Aix*Marseille université

Licence SPI – parcours AGE

Semestre 5

Automatisme et...

Informatique Industrielle

Bases en langage C

Julien Marot
Institut Fresnel

Mél. : julien.marot@fresnel.fr

Zouhair Haddi
LSIS

Mél. : zouhair.haddi@lsis.org

Premiers programmes

1 – Ne fait rien

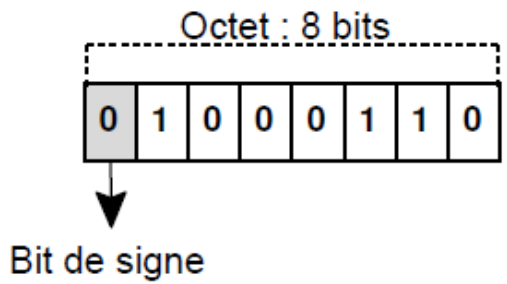
```c
/*
    Programme minimum
    ne faisant rien
*/

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

2 – « Hello World ! »

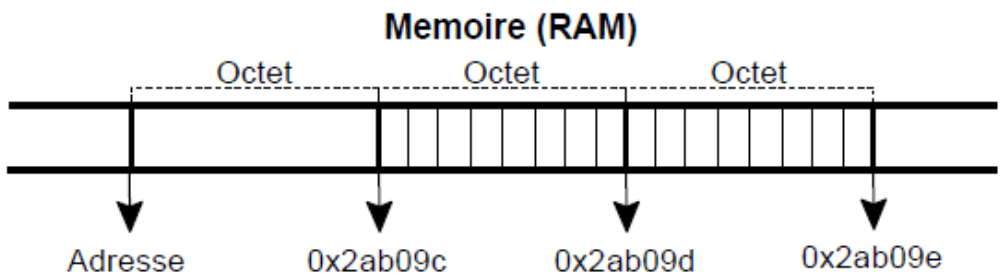
```c
#include <stdio.h>

int main()
  {
    // Affiche le message
    printf("\nHello World!");

    return 0;
  }
```

Élément mémoire : bit et octet

- Un octet (ou « byte ») est composé de 8 « bits »,
- Un bit est un élément mémoire de base : valeurs 0 ou 1.
- Un octet : $2^8 = 256$ possibilités, valeurs de -128 à +127.



- Mémoire (RAM) : suite d'octets adressés.



Incrémentation et décrémentation

Incrémentation

- `i++;` ou `++i;` signifie `i = i + 1;`
- `i += n;` signifie `i = i + n;`
- Même chose avec `i--`, `--i` et `i -= n`

Pré-incrémentation

- `int i = 5; int a = ++i;` donne `i=6` et `a=6`
- `int i = 5; int a = --i;` donne `i=4` et `a=4`

Post-incrémentation

- `int i = 5; int a = i++;` donne `i=6` et `a=5`
- `int i = 5; int a = i--;` donne `i=4` et `a=5`

Opérateurs relationnels et logiques

➽ **Opérateurs relationnels** (`true` ou `false`)

- `==`, `!=` : identité, différent . Ex : `A == 5`
- `>`, `<`, `>=`, `<=` : supérieur, inférieur

Exemples `int A = 10, B = -5;`

- `A == B` ➽ `0` : `false`
- `A >= B` ➽ `1` : `true`

Attention Les deux opérandes doivent avoir le même type arithmétique, sinon une conversion de types est effectuée.

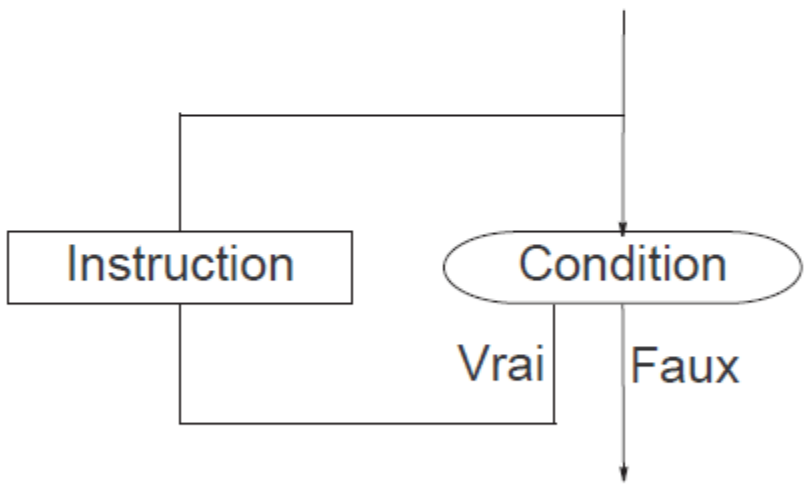
➽ **Opérateurs logiques** :

- `!` : négation unaire d'une valeur logique. Ex : `!true == false`
- `&&` : « ET » de deux valeurs logiques. Ex : `true && false == false`
- `||` : « OU » de deux valeurs logiques. Ex : `true || false == true`

Exemple de combinaison d'opérateurs : `(K>5) && (L<10)`.

Inst. répétitive : `while`

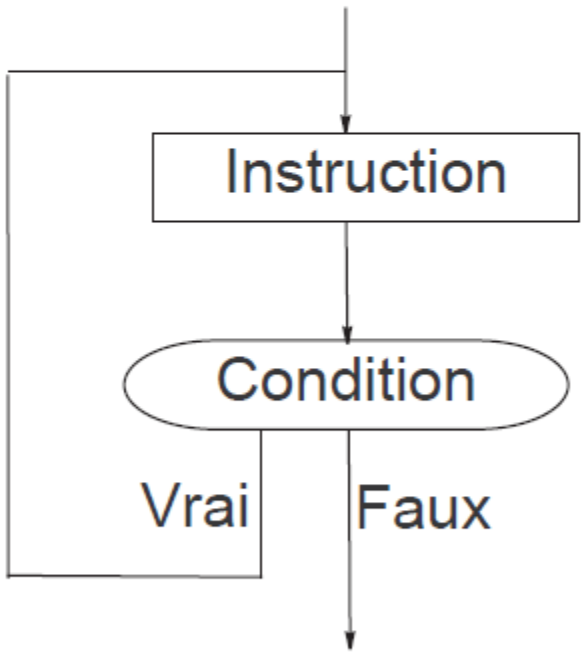
`while` (expression de fin) instruction




```
int nb, i;
printf("\nEntrez un nombre : " );
scanf("%d", &nb);

i = 0;
while (i<nb)
{
   printf("Iteration : %d", i);
   i++;
}
```

Inst. répétitive : do ... while

```
do instruction   while ( expression de fin ) ;
```

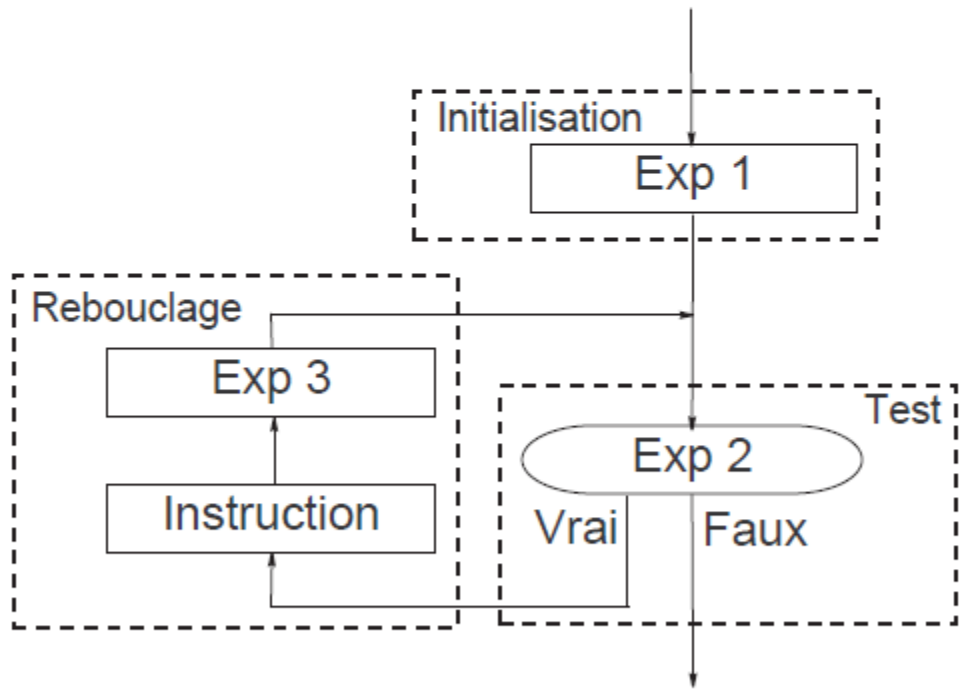


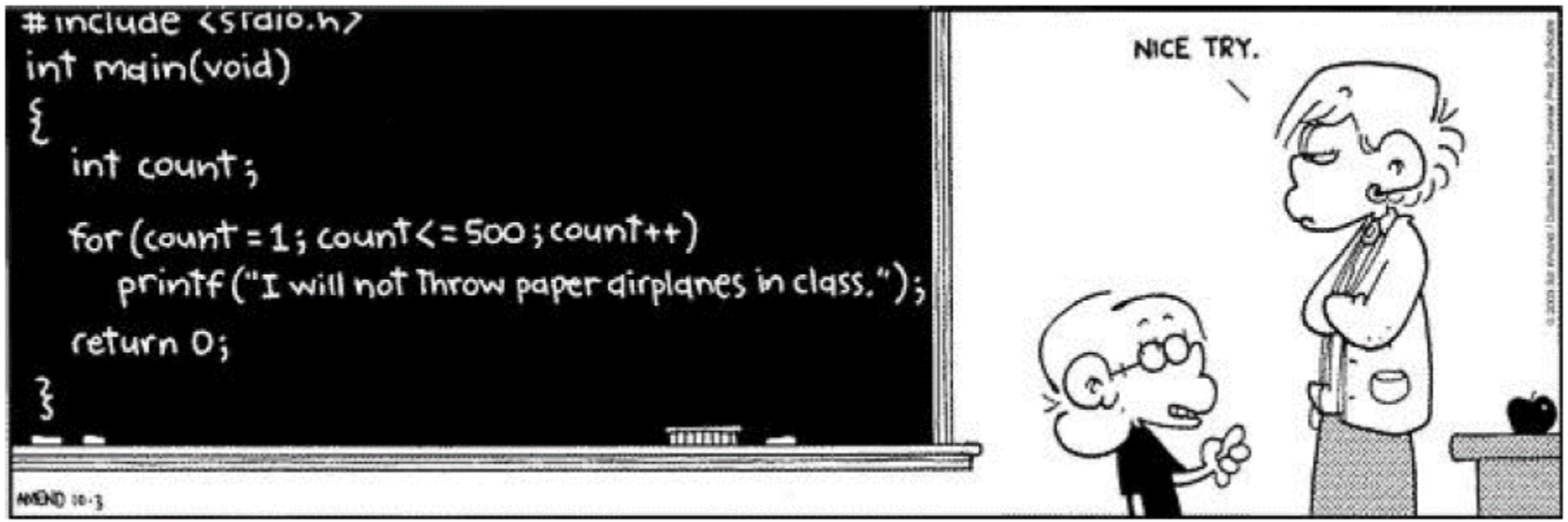
démo 9

```
int s[10], i=0, n=5634;
do {
   s[i] = n % 10;
   i++;
   n /= 10;
} while ( n > 0 );
```

Inst. répétitive : `for`

```
for ( exp1 ; exp2 ; exp3 )   instruction
```




#include <stdio.h>
int main(void)
{
int count;
for (count = 1; count<=500; count++)
printf("I will not throw paper airplanes in class.");
return 0;
}
NICE TRY.

Fonctions : rôles et intérêts

- **Objectifs** : Découper un programme en petites entités :
  - Indépendantes ;
  - Réutilisables ;
  - Plus lisibles.
- **Où** : Les fonctions peuvent être définies :
  - Dans le fichier principal ;
  - Dans d'autres fichiers source (compilation séparée - Tr. 55) ;
- **Syntaxe d'une fonction**

```
type_de_retour nom_fonction ( arguments ) {
    declarations des variables locales ;
    instructions ;
}
```

Fonction sans argument

```c
// Déclaration de la fonction
void Affiche() {
    printf("Message");
}

// fonction principale
int main() {
    // Appel de la fct
    Affiche();
    return 0;
}
```

```c
// En-tête de la fonction
void Affiche();

// fonction principale
int main() {
    // 1ppel de la fct
    Affiche();
    return 0;
}

// Déclaration de la fonction
void Affiche() {
    printf("Message");
}
```

Fonction : valeur de retour

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

// Déclaration de la fonction
double Sinus() {
    double angle;
    scanf("%lf", &angle);
    return sin(angle);
}

int main() {
    double res;
    // Appel de la fonction
    res = Sinus();
    printf("\nResultat = %lf", res);
    return 0;
}
```

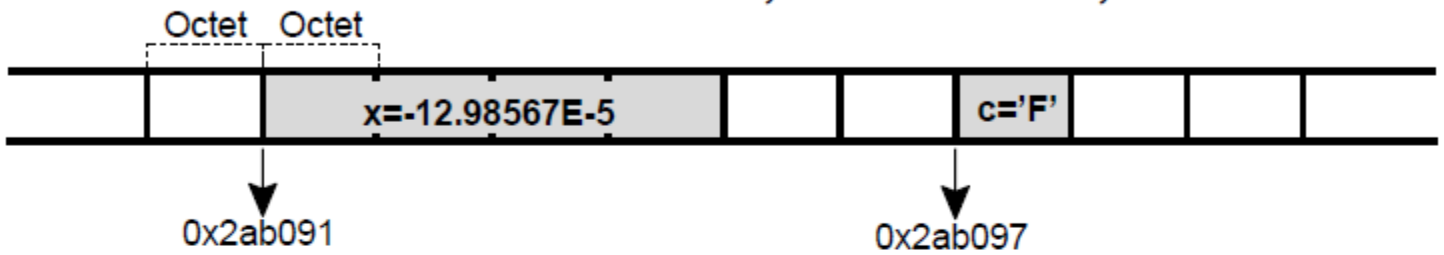
Fonction : passage d'argument par valeur

```c
#include<stdio.h>

// Déclaration de la fonction
int Add(int val1, int val2) {
    int result;                 // dec var locale
    result = val1 + val2;       // calcul
    return result;             // retour
}

int main() {
    int res, a = 5, b = 8;
    //Appels de la fonction
    res = Add(a, b);           printf("\nResultat = %d", res);
    res = Add(-8, 15);         printf("\nResultat = %d", res);
    res = Add(-8*2-5, 6*a);    printf("\nResultat = %d", res);

    return 0;
}
```

➥ Les arguments transmis sont recopiés dans les variables locales de la fonction.

Type pointeur

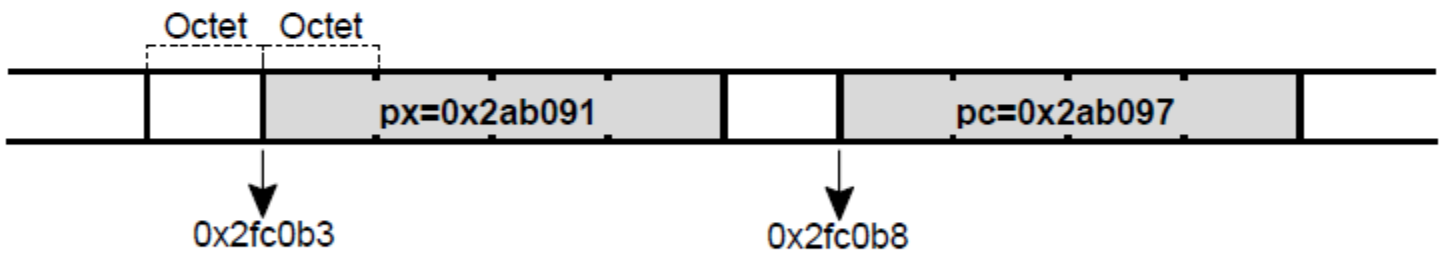
- **Variables :** `float x=-12.98587E-5; char c='F';`



- **Pointeurs :**

  Déclaration : `float* px;` et `char* pc;`

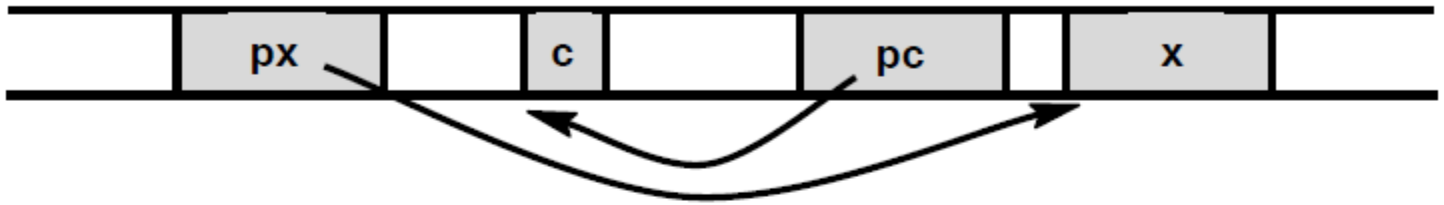
  Initialisation : `px = &x;` et `pc = &c;`



- **Remarque :** La taille d'un pointeur, c'est à dire la quantité de mémoire réservée au stockage d'une adresse, est de 4 octets (quel que soit le type sur lequel il pointe).

  `sizeof(char*)==sizeof(double*)==4` `sizeof(char)==1; sizeof(double)==8`

Schéma global :



Autre exemple :

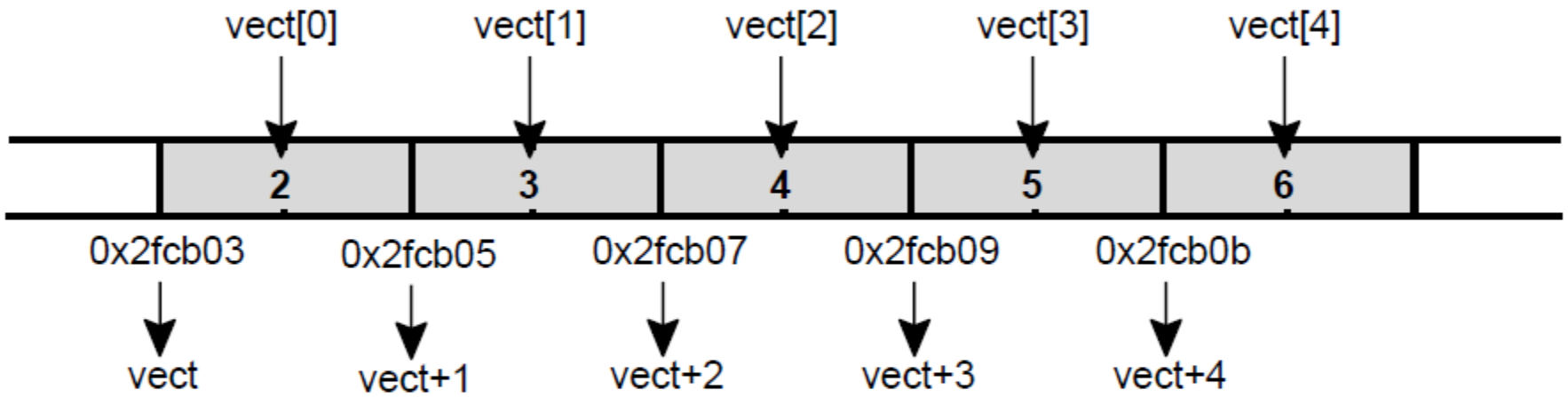
```
int x=1, y=2;
int* px;     // pi pointe sur un int
px  = &x;    // pi pointe sur x
y   = *px;   // y reçoit la valeur 1 (x)
*px = 0;     // x vaut 0
```

Remarques :

- `&*p` est identique à `p` (pour `p` un pointeur)
- Opérateurs sur les pointeurs : `+`, `-`, `++`, `--`, `+=`, `-=`, `==`, `!=` ...
- `p=NULL` : identique à l'initialisation à `0` d'une variable entière ou flottante.

Pointeurs et Tableaux

- **Tableau 1D (Vecteur) :** `int vect[5] = {2, 3, 4, 5, 6};`



- `&vect[0] == vect` : Adresse du 1er élément de `vect`.
- `&vect[i] == vect+i` : Adresse du ième élément de `vect`.
- `vect[i] == *(vect+i)` : Valeur du ième élément de `vect`.

```c
#include <stdio.h>
void main( ) {
   int i = 0;
   int *p;
   float x = 3.14;
   float *f;

   p = &i;
   *f = 666;
   f = &x;
   *f = *p;
   *p = 34;
   p = f;
   *p = *p + 1;

   printf("\ni=%d; *f=%f", i, *f);
}
```

Supprimer les lignes invalides - Que s'affiche-t'il ?

```c
#include <stdio.h>
void main( ) {
   int i = 0;
   int *p;
   float x = 3.14;
   float *f;

   p = &i;
   *f = 666;
   f = &x;                  // *f=3.14
   *f = *p;
   *p = 34;                 // i=34
   p = f;
   *p = *p + 1;             // i=35

   printf("\ni=%d; *f=%f", i, *f);
}
```

➽ **À vous de deviner ... :**

```
float *p, *q;
float tab[5] = {1, 2, 3, 4, 5};

p = &tab[2];
p = tab+2;
*p= 0.0;
q = p++;          \\Post-incrémentation !
*(q+2) = -10;
```

Supprimer les lignes invalides - État du vecteur ?

À vous de deviner ... :

```
float *p, *q;
float tab[5] = {1, 2, 3, 4, 5};

p = &tab[2];
p = tab+2;          \\ idem ligne au-dessus
*p= 0.0;            \\ tab[2] = 0          \\ tab = {1,2,0,4,5}
q = p++;            \\Post-incrémentation !  \\ *q=tab[2] donc
                                             \\ *q=0
                                             et
                                             \\ *p=tab[2+1]=tab[3]=4
*(q+2) = -10;
```

***q=tab[2]=0**

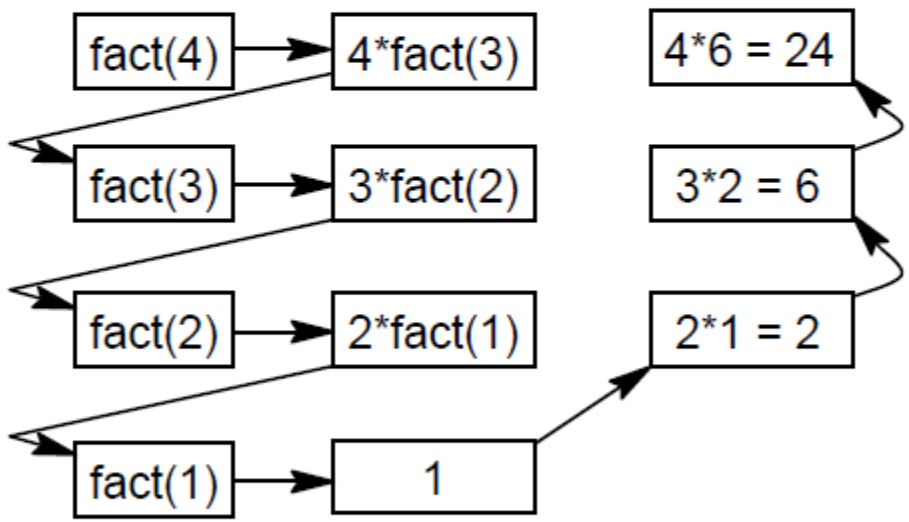
***(q+2) == tab[4]=-10**

bilan: {1,2,0,4,-10}

Fonctions récursives

➫ La fonction s'appelle elle-même ! ➫ Exemple : $4! = 4 * 3!$.

```c
#include<stdio.h>

// Déclaration de la fonction
int fact(int n) {
    if (n<=1)
        return 1;
    else
        return n*fact(n-1);
}

int main() {
    // Appel de la fonction
    printf("\nFactoriel 4 = %d", fact(4));
    return 0;
}
```



Attention au test de fin de récursivité !

FIN