

Les fonctions ¹

Salem BENFERHAT

Centre de Recherche en Informatique de Lens (CRIL-CNRS)
email : benferhat@cril.fr

¹Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

Premier contrôle continu
La semaine du 13 octobre
ou
La semaine du 20 octobre

Points bonus
pendant
les TPs

- Mercredi 8 octobre - 17h30 > 19h
- Atrium du Louvre-Lens Vallée - 84 rue Paul Bert - 62300 LENS
- Les réseaux de neurones à travers une histoire ...

- En C, les sous-programmes sont appelées des fonctions.
- La décomposition d'un programme en fonctions offre plusieurs avantages :
 - Améliorer la lisibilité des programmes.
 - Faciliter la détection d'erreurs. Chaque fonction peut-être vérifiée et testée séparément.
 - Eviter la duplication des codes.

- Un autre avantage est la possibilité de ré-utiliser des codes pour d'autres programmes.
 - Par exemple, vous pouvez écrire une fonction qui vérifie si un nombre est premier ou non.
 - Puis utiliser cette fonction dans différents programmes.
 - D'ailleurs c'est le principe des bibliothèques C.
 - Par exemple, les fonctions :
 - ▶ printf,
 - ▶ scanf,
 - ▶ etc.
- sont utilisés dans différents programmes.

Une fonction dispose :

- d'un identifiant.
- Des déclarations de paramètres ou d'arguments.
- Le type de la valeur retournée.
- Le corps de la fonction.

Les fonctions : plus précisément

La syntaxe d'une fonction est :

```
Type_de_la_fonction identifiant (arg1, arg2, ...)  
{  
    Corps de la fonction [avec un return]  
}
```

- La convention pour la définition des noms de fonctions est la même que celle utilisée pour les variables.
- La valeur retournée peut-être :
 - de type quelconque, y compris :
 - ▶ les pointeurs, et
 - ▶ les types complexes que nous verrons plus tard.
- Lorsque la fonction ne retourne aucune valeur, on utilisera le mot clef "void".

Les fonctions : plus précisément

La syntaxe d'une fonction est :

```
Type_de_la_fonction identifiant (arg1, arg2, ...)  
{  
    Corps de la fonction [avec un return]  
}
```

- Chaque argument (appelé aussi paramètre d'entrée) est simplement déclaré sous la forme :

type_argument identifiant_argument

- Lorsqu'aucun argument n'est utilisé, le mot clef "void" est utilisé.
- Ce mot clef "void" précise, explicitement, que la fonction ne contient pas d'arguments

Les fonctions : plus précisément

La syntaxe d'une fonction est :

```
Type_de_la_fonction identifiant (arg1, arg2, ...)  
{  
    Corps de la fonction [avec un return]  
}
```

Le corps de la fonction contient :

- des déclarations de variables, dites locales.
- Une suite d'instructions qui contient, au moins une fois, l'expression :

return valeur (ou expression);

- La valeur retournée est de même type que celui de la fonction.
- Bien sûr, si le type retournée est "void", alors l'instruction "return" ne doit pas être utilisée.

**Prenons
des exemples**

Un premier exemple

Exemple de fonctions avec un seul argument

- Ecrire une fonction C qui :
 - qui prend en entrée un entier relatif a (positif ou négatif) et
 - retourne sa valeur absolue.

Un premier exemple

Commençons par l'entête

- Il nous faut un nom pour identifier la fonction.
 - Soyons originaux, appelons cette fonction : `absolu`.
- Ensuite, il faut déclarer les paramètres ou les arguments.
 - Dans notre exemple, nous avons un seul paramètre à déclarer :

`int a`

- Finalement, il faut préciser le type de retour. Ca sera :
 - type `"int"`
 - on peut aussi utiliser `"unsigned int"`.

Un premier exemple

Commençons par l'entête

Pour résumer, l'entête de la fonction est :

```
int absolu (int a)
```

Un premier exemple

Corps de la fonction

- Il nous reste maintenant à spécifier le corps de la fonction.
- Nous n'avons besoin d'aucune variable locale pour réaliser l'objectif de la fonction.
- Comme notre fonction est de type int, il faudra alors qu'elle contienne au moins une instruction qui retournerait une valeur (ou une expression) de type int.

Un premier exemple

Voici donc un exemple d'écriture de notre fonction "absolu" :

```
int absolu (int a)
{
    if (a<0)
        return -a;
    else return a;
}
```


Un premier exemple

Donner une écriture de notre fonction "absolu" sans utiliser le "else".

Un premier exemple

Voici donc un autre exemple d'écriture de notre fonction "absolu" (sans utiliser le "else"):

```
int absolu (int a)
{
    if (a<0)
        return -a;
    return a;
}
```

Un premier exemple

Voici donc un autre exemple d'écriture de notre fonction "absolu" (sans utiliser le "else"):

```
int absolu (int a)
{
    if (a<0)
        return -a;
    return a;
}
```

- En effet, le else n'est pas obligatoire.

Un premier exemple

Voici donc un autre exemple d'écriture de notre fonction "absolu" (sans utiliser le "else"):

```
int absolu (int a)
{
    if (a<0)
        return -a;
    return a;
}
```

- En effet, le else n'est pas obligatoire.
- Car, en langage C, le "return" provoque la fin de l'exécution de la fonction.

Un premier exemple

Donner une écriture de notre fonction "absolu" sans utiliser le "if".

Un premier exemple

Voici encore une autre écriture de notre fonction "absolu" :

```
int absolu (int a)
{
    return (a<0) ? -a: a;
}
```

Un premier exemple

Voici encore une autre écriture de notre fonction "absolu" :

```
int absolu (int a)
{
    return (a<0) ? -a: a;
}
```

- Ici on utilise le conditionnement ternaire.

Un autre exemple

Exemple de fonctions avec plusieurs arguments

- Ecrire une fonction C qui :
 - qui prend en entrée deux entiers a et b, et
 - retourne la valeur maximale de a et b.

Un autre exemple

Solution

```
int max (int a, int b)
{
    if (a>=b)
        return a;
    else return b;
}
```

Un autre exemple

Solution

```
int max (int a, int b)
{
    if (a>=b)
        return a;
    else return b;
}
```

Remarques

- Le "else" n'est pas obligatoire et

Un autre exemple

Solution

```
int max (int a, int b)
{
    if (a>=b)
        return a;
    else return b;
}
```

Remarques

- Le "else" n'est pas obligatoire et
- on peut aussi utiliser le conditionnement ternaire.

- Une fois une fonction est définie (écrite), elle peut-être utilisée aussi bien dans le programme principale que dans le corps d'une autre fonction.
- On parle alors d'appels de fonctions.
- Dans un premier temps, on se contentera de recopier son entête dans les fonctions (comme le main) où elle sera utilisée.

- La valeur retournée peut-être
 - affichée (imprimée),
 - affectée à une variable ou
 - utilisée dans une expression.
- Bien sûr, chaque argument doit-être au préalable renseigné (par une expression ou une valeur).

Exemples d'utilisation des fonctions

La fonction "absolu"

L'exemple suivant montre l'appel à la fonction "absolu" par l'instruction "printf" au niveau du programme principal.

```
int main (void)
{
    int absolu (int a);
    int i=-20;
    printf ("La valeur absolue de i est : %d.\n ", absolu(i));
    return(0);
}
```

Exemples d'utilisation des fonctions

La fonction "absolu"

L'exemple suivant mémorise le résultat de l'appel à la fonction "absolu" dans une variable).

```
int main (void)
{
    int absolu (int a);
    int x=-23, y;
    y=absolu(x);
    printf ("La valeur absolue de %d est : %d.\n ", x,y);
    return(0);
}
```

Exemples d'utilisation des fonctions

La fonction "absolu"

L'exemple suivant utilise l'appel à la fonction "absolu" dans une expression (qui calcule une distance dite de Manhattan).

```
int main (void)
{
    int absolu (int a);
    int x1, y1, x2, y2, distance;
    scanf("%d %d %d %d", x1, y1, x2, y2);
    distance=absolu(x1-y1)+absolu(x2-y2);
    return(0);
}
```


Passage des paramètres

Regardons de près l'appel à la fonction "absolu"

```
#include <stdio.h>
int absolu (int a)
{
    if (a<0)
        return -a;
    else return a;
}
```

Regardons de près l'appel à la fonction "absolu"

```
#include <stdio.h>
int absolu (int a)
{
    if (a<0)
        return -a;
    else return a;
}
```

```
int main (void)
{
    int absolu (int a);
    int x=-23, y;
    y=absolu(x);
    return(0);
}
```

Regardons de près l'appel à la fonction "absolu"

```
#include <stdio.h>
int absolu (int a)
{
    if (a<0)
        return -a;
    else return a;
}
int main (void)
{
    int absolu (int a);
    int x=-23, y;
    y=absolu(x);
    return(0);
}
```

Voici donc l'état de la mémoire après les deux déclarations des variables x et y.

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	-23
y	0xF2346	?
	0xF2347	
	0xF2348	
	0xF2349	
	0xF234A	
	0xF234B	
	0xF234C	

Regardons de près l'appel à la fonction "absolu"²

```
#include <stdio.h>
int absolu (int a)
```

```
{
    if (a<0)
        return -a;
    else return a;
}
```

```
int main (void)
{
    int absolu (int a);
    int x=-23, v;
    v=absolu(x);
    return(0);
}
```

A ce niveau là, la fonction "absolu" est appelée et sera exécutée.

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	-23
y	0xF2346	?
	0xF2347	
	0xF2348	
	0xF2349	
	0xF234A	
	0xF234B	
	0xF234C	

²Normalement, je dois laisser 4 octets minimum pour un entier

Regardons de près l'appel à la fonction "absolu"

```
#include <stdio.h>
int absolu (int a)
{
    if (a<0)
        return -a;
    else return a;
}

int main (void)
{
    int absolu (int a);
    int x=-23, y;
    v=absolu(x);
    return(0);
}
```

- Un espace mémoire est réservée pour *a*
- La valeur de x est recopiée dans *a*
- La valeur 23 est retournée

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	-23
y	0xF2346	?
	0xF2347	
	0xF2348	
a	0xF2349	-23
	0xF234A	
	0xF234B	
	0xF234C	

Regardons de près l'appel à la fonction "absolu"

```
#include <stdio.h>
int absolu (int a)
{
    if (a<0)
        return -a;
    else return a;
}
int main (void)
{
    int absolu (int a);
    int x=-23, y;
    y=absolu(x);
    return(0);
}
```

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	-23
y	0xF2346	23
	0xF2347	
	0xF2348	
a	0xF2349	-23
	0xF234A	
	0xF234B	
	0xF234C	

Finalement, la valeur (23) retournée par la fonction "absolu" est stockée dans la variable y.

- Lors de l'appel :

$y = \text{absolu}(x);$

Le compilateur crée :

- un espace mémoire pour le paramètre a , et
- recopie la valeur de x dans a
- Ce qui signifie que :
 - x et a sont stockées dans deux espaces mémoires différents,
 - et que toute modification apportée sur le paramètre a n'aura aucun effet sur la variable x déclarée dans la fonction "main".

Remarque importante sur le passage des paramètres

Le passage des paramètres dans les fonctions se fait par valeur.

Passage des paramètres : question

Question

Comment faire alors pour modifier le contenu d'une variable passée en paramètre?

Réponse

Utiliser dans ce cas l'adresse de la variable.

Ecrire une fonction C, appelée `echange`, qui :

- prend deux entiers positifs `A` et `B` en paramètres et les échange (en utilisant une variable intermédiaire).

Passage par valeur : petit soucis

Une première solution serait :

```
void echange (unsigned int a, unsigned int b)
{
    unsigned int c;
    c=a;
    a=b;
    b=c;
}
```

Passage par valeur : petit soucis

Une première solution serait :

```
#include <stdio.h>
int main (void)
{
    void echange (unsigned int a, unsigned int b);
    unsigned int x, y;
    x=5;
    y=10;
    echange(x, y);
    printf ("Après l'appel à la fonction echange
           x=%u et y=%u !!!!!\n", x,y);
    return(0);
}
```

- Après exécution de ce programme,

Passage par valeur : petit soucis

Une première solution serait :

```
#include <stdio.h>
int main (void)
{
    void echange (unsigned int a, unsigned int b);
    unsigned int x, y;
    x=5;
    y=10;
    echange(x, y);
    printf ("Après l'appel à la fonction echange
           x=%u et y=%u !!!!!!\n", x,y);
    return(0);
}
```

- Après exécution de ce programme, on affiche :
 - et après l'appel à la fonction "echange",
 - on affiche x=5 et y=10 !!!!!

Passage par valeur : petit soucis

Une première solution serait :

```
#include <stdio.h>
int main (void)
{
    void echange (unsigned int a, unsigned int b);
    unsigned int x, y;
    x=5;
    y=10;
    echange(x, y);
    printf ("Après l'appel à la fonction echange
           x=%u et y=%u !!!!!\n", x,y);
    return(0);
}
```

- Après exécution de ce programme, on affiche :
 - et après l'appel à la fonction "echange",
 - on affiche x=5 et y=10 !!!!!
- C'est-dire les valeurs de x et y n'ont pas été échangées.

Fonction echange :
Regardons le passage
des paramètres en mémoire

Regardons de près l'appel à la fonction "echange"³

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
    unsigned int c;
    c=a;
    a=b;
    b=c;
}
```

```
int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;

    echange(x, y);
    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Voici l'état de la mémoire après l'exécution des deux premières instructions de la fonction main.

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	5
y	0xF2346	10
	0xF2347	
	0xF2348	
	0xF2349	
	0xF234A	
	0xF234B	
	0xF234C	

³Normalement, nous devons réserver 4 octets pour un entier.

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
    unsigned int c;
    c=a;
    a=b;
    b=c;
}
```

```
int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;
    echange(x, y);

    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Etat de la mémoire au début de l'appel
de la fonction échange et après pas-
sage des paramètres.

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	5
y	0xF2346	10
	0xF2347	
	0xF2348	
a	0xF2349	5
b	0xF234A	10
	0xF234B	
	0xF234C	

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
    unsigned int c;
```

```
    c=a;
    a=b;
    b=c;
}
int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;
    echange(x, y);

    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Etat de la mémoire après la déclaration de la variable locale c de la fonction échange.

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	5
y	0xF2346	10
	0xF2347	
	0xF2348	
a	0xF2349	5
b	0xF234A	10
c	0xF234B	??
	0xF234C	

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
    unsigned int c;
    c=a;

    a=b;
    b=c;
}
int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;
    echange(x, y);

    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	5
y	0xF2346	10
	0xF2347	
	0xF2348	
a	0xF2349	5
b	0xF234A	10
c	0xF234B	5
	0xF234C	

Etat de la mémoire après la première affectation de la fonction échange.

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
    unsigned int c;
    c=a;
    a=b;
```

```
    b=c;
}
int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;
    echange(x, y);

    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Etat de la mémoire après la deuxième affectation de la fonction échange.

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	5
y	0xF2346	10
	0xF2347	
	0xF2348	
a	0xF2349	10
b	0xF234A	10
c	0xF234B	5
	0xF234C	

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
    unsigned int c;
    c=a;
    a=b;
    b=c;
}

int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;
    echange(x, y);

    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	5
y	0xF2346	10
	0xF2347	
	0xF2348	
a	0xF2349	10
b	0xF234A	5
c	0xF234B	5
	0xF234C	

Etat de la mémoire après la dernière affectation de la fonction échange.

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
    unsigned int c;
    c=a;
    a=b;
    b=c;
}
int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;
```

```
    echange(x, y);
    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

- Etat de la mémoire à la fin de l'appel de la fonction échange.
- Et malheureusement les valeurs n'ont pas été échangées.

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	5
y	0xF2346	10
	0xF2347	
	0xF2348	
a	0xF2349	10
b	0xF234A	5
c	0xF234B	5
	0xF234C	

**Fonction echange :
passons les adresses
des variables**

Fonction echangé : passons les adresses

Principe et première conséquence

- L'idée est lors de l'appel à la fonction "echangé" (depuis le main), de remplacer

`echangé (x, y) ;`

par

Fonction echangé : passons les adresses

Principe et première conséquence

- L'idée est lors de l'appel à la fonction "echangé" (depuis le main), de remplacer

```
echangé (x, y) ;
```

par

```
echangé (&x, &y) ;
```

Fonction echange : passons les adresses

Principe et première conséquence

- L'idée est lors de l'appel à la fonction "echange" (depuis le main), de remplacer

```
echange (x, y);
```

par

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

- C'est-à-dire au lieu de passer la valeur des variables x et y , on passera les adresses où se trouvent les variables x et y .

Fonction echange : passons les adresses

Principe et première conséquence

- L'idée est lors de l'appel à la fonction "echange" (depuis le main), de remplacer

```
echange (x, y);
```

par

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

- C'est-à-dire au lieu de passer la valeur des variables x et y , on passera les adresses où se trouvent les variables x et y .
- De cette façon, on peut accéder au contenu des variables x et y .

Deuxième conséquence

- Changer la déclaration des paramètres dans la fonction "echangé".

Fonction echange : passons les adresses

Deuxième conséquence

- Changer la déclaration des paramètres dans la fonction "echange".
- Précisément, remplacer

```
1 void echange (unsigned int a, unsigned int b)
```

Fonction échange : passons les adresses

Deuxième conséquence

- Changer la déclaration des paramètres dans la fonction "échange".
- Précisément, remplacer

```
1 void échange (unsigned int a, unsigned int b)
```

par

```
1 void échange (unsigned int *a, unsigned int *b)
```

Deuxième conséquence

- Changer la déclaration des paramètres dans la fonction "échange".
- Précisément, remplacer

```
1 void échange (unsigned int a, unsigned int b)  
   par
```

```
1 void échange (unsigned int *a, unsigned int *b)
```

- C'est-à-dire, les paramètres *a* et *b* ne sont plus déclarés comme des entiers mais plutôt des pointeurs vers des entiers.

Troisième conséquence

- Changer les opérations arithmétiques dans la fonction "echange".

Troisième conséquence

- Changer les opérations arithmétiques dans la fonction "echange".
- Plus, précisément, remplacer

```
1    c=a;  
2    a=b;  
3    b=c;
```

Troisième conséquence

- Changer les opérations arithmétiques dans la fonction "echange".
- Plus, précisément, remplacer

```
1   c=a;  
2   a=b;  
3   b=c;
```

par

```
1   c=*a;  
2   *a=*b;  
3   *b=c;
```

Troisième conséquence

- Changer les opérations arithmétiques dans la fonction "échange".
- Plus, précisément, remplacer

```
1   c=a;  
2   a=b;  
3   b=c;
```

par

```
1   c=*a;  
2   *a=*b;  
3   *b=c;
```

- C'est-à-dire, on a plus un accès direct aux données des variables ou des paramètres (ici a, b), mais seulement un accès indirect à ces données via les pointeurs.

Fonction échange : passons les adresses

```
#include <stdio.h>

void échange (unsigned int *a, unsigned int *b)
{
    unsigned int c;
    c=*a;
    *a=*b;
    *b=c;
}

int main (void)
{
    void échange (unsigned int *a, unsigned int *b);
    unsigned int x, y;
    x=5;
    y=10;
    échange(&x, &y);
    printf ("Après l'appel à la fonction échange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Regardons de près l'appel à la fonction "echange"

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

```
void echange(unsigned int *a, unsigned int *b)
```

```
{  
    unsigned int c;  
    c=*a;  
    *a=*b;  
    *b=c;  
}
```

```
int main (void)
```

```
{  
    unsigned int x, y;  
    x=5;  
    y=10;
```

```
    echange(&x, &y);  
    printf ("Après l'appel à  
la fonction echange x=%u et y=%u.\n", x,y);  
    return(0);  
}
```

Voici l'état de la mémoire après l'exécution des deux premières instructions de la fonction main.

Variables



x

y

Adresses



0xF2345

0xF2346

0xF2347

0xF2348

0xF2349

0xF234A

0xF234B

0xF234C

Mémoire



5

10

Regardons de près l'appel à la fonction "echange"

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

```
void echange(unsigned int *a, unsigned int *b)
```

```
{  
    unsigned int c;  
    c=*a;  
    *a=*b;  
    *b=c;  
}
```

```
int main (void)
```

```
{  
    unsigned int x, y;  
    x=5;  
    y=10;  
    echange(&x, &y);  
}
```

```
printf ("Après l'appel à  
la fonction echange x=%u et y=%u.\n", x,y);  
return(0);  
}
```

Etat de la mémoire au début de l'appel
de la fonction échange et après pas-
sage des paramètres.

Variables

Adresses

Mémoire



x

0xF2345

5

y

0xF2346

10

0xF2347

0xF2348

a

0xF2349

0xF2345

b

0xF234A

0xF2346

0xF234B

0xF234C

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
    unsigned int c;
```

```
    c=*a;
    *a=*b;
    *b=c;
}
int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;
    echange(&x, &y);

    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Etat de la mémoire après la déclaration de la variable locale c de la fonction échange.

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	5
y	0xF2346	10
	0xF2347	
	0xF2348	
a	0xF2349	0xF2345
b	0xF234A	0xF2346
c	0xF234B	??
	0xF234C	

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
    unsigned int c;
    c=*a;

    *a=*b;
    *b=c;
}
int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;
    echange(&x, &y);

    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	5
y	0xF2346	10
	0xF2347	
	0xF2348	
a	0xF2349	0xF2345
b	0xF234A	0xF2346
c	0xF234B	5
	0xF234C	

Etat de la mémoire après la première affectation de la fonction échange.

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
    unsigned int c;
    c=*a;
    *a=*b;
```

```
    *b=c;
}
int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;
    echange(&x, &y);

    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Etat de la mémoire après la deuxième affectation de la fonction échange.

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	10
y	0xF2346	10
	0xF2347	
	0xF2348	
a	0xF2349	0xF2345
b	0xF234A	0xF2346
c	0xF234B	5
	0xF234C	

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
    unsigned int c;
    c=*a;
    *a=*b;
    *b=c;
}

int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;
    echange(&x, &y);

    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	10
y	0xF2346	5
	0xF2347	
	0xF2348	
a	0xF2349	0xF2345
b	0xF234A	0xF2346
c	0xF234B	5
	0xF234C	

Etat de la mémoire après la dernière affectation de la fonction échange.

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
    unsigned int c;
    c=*a;
    *a=*b;
    *b=c;
}
int main (void)
{
    unsigned int x, y;
    x=5;
    y=10;

    echange(&x, &y);
    printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
    return(0);
}
```

Variables	Adresses	Mémoire
↓	↓	↓
x	0xF2345	10
y	0xF2346	5
	0xF2347	
	0xF2348	
a	0xF2349	0xF2345
b	0xF234A	0xF2346
c	0xF234B	5
	0xF234C	

- Etat de la mémoire à la fin de l'appel de la fonction échange.
- Et on affiche bien que les valeurs ont été échangées.

Notions de Variables locales

- Une variable est dite locale si elle a une portée (ou accessibilité) limitée.
- Toute variable déclarée à l'intérieur d'un bloc n'est accessible que dans ce bloc.
- Ceci est particulièrement vraie pour les fonctions :
 - La variable "c" n'est accessible qu'à l'intérieur de la fonction "echange".
 - Sa durée de vie = la durée de vie de la fonction.
 - De manière générale, dès qu'une fonction termine son exécution toutes les variables locales sont libérées et ne sont plus accessibles (sauf cas particuliers que nous discuterons plus tard).

Rappels : adresse des variables

Grâce à l'opérateur `&`, on peut récupérer les adresses des variables locales et paramètres utilisés dans la fonction `Estpremiernaif`.

```
int Estpremiernaif( int n)
{
    int i;
    if( n <= 1) return 0;
    printf ("Dans la fonction Estpremiernaif :\n");
    printf ("la variable n se trouve a l'adresse %p :\n", &n);
    printf ("la variable local i se trouve a l'adresse %p :\n", &i);
    for(i = 2; i <= n; ++i)
    {
        if(n % i == 0) return 0;
    }
    return 1;
}
```

Adresse

- L'exécution de la fonction donne :

Dans la fonction Estpremiernaif :

la variable n se trouve a l'adresse : 0x7fff61f5cb2c

la variable local i se trouve a l'adresse : 0x7fff61f5cb20

- Les adresses sont données en hexadécimal

Adresse des variables

Grâce à l'opérateur &, on peut également récupérer les adresses des fonctions.

```
int main ()
{
    int nombre;
    printf ("Introduire un nombre : ");
    scanf ("%d", &nombre);
    printf ("l'adresse de la variable nombre est : %p \n", &nombre);
    printf ("l'adresse de la fonction Estpremiernaif est : %p \n",
            &Estpremiernaif);
    printf ("l'adresse de la fonction main est : %p \n", &main);
    if (Estpremiernaif(nombre)==1)
        printf ("Le nombre %d est premier : ", nombre);
    else printf ("Le nombre %d n est pas premier : ", nombre);
    return(0);
}
```

L'adresse

- L'exécution de la fonction donne :

l'adresse de la variable nombre est :	0x7fff6d03db44
l'adresse de la fonction Estpremiernaif est :	0x10d43eba0
l'adresse de la fonction main est :	0x10d43ec50

Remarque

- La variable *nombre* utilisée dans le programme principal et le paramètre *n* de la fonction *Estpremiernaif* n'ont pas la même adresse.
- De ce fait, toute modification sur *n* n'affectera pas la variable *nombre*.

Les fonctions récursives

La Récursivité

- La récursivité est essentielle dans de nombreux langages de programmation
- Elle permet d'offrir des solutions élégantes à de nombreux problèmes
- Elle s'applique particulièrement pour les problèmes qui se décomposent en sous-problèmes similaires mais de tailles plus petites. Elle reprend le principe de "diviser pour régner".

Définition

Une fonction est dite récursive si elle utilise une référence à elle-même dans sa définition.

3 principes ...

3 principes ...

- Une fonction récursive doit évidemment avoir un appel récursif.
- Une fonction doit contenir des cas de "base" qui ne contiennent pas d'appels récursifs. Ca représente l'équivalent de conditions d'arrêt dans des procédures itératives. Sans ces situations de base, la fonction "bouclera à l'infini"...
- Les fonctions récursives doivent converger vers ces cas de base pour toutes les valeurs données en entrée.

Hypothèses de travail

- Etablir les cas de base (que l'on sait traiter facilement).
 - Ces cas de bases représentent les conditions d'arrêt où on ne fait plus d'appels récursifs
- Pour écrire une fonction récursive avec un paramètre donné (exemple un entier n), faites l'hypothèse que la fonction est disponible pour les paramètres plus petits ($n-1$, $n-2$, etc.)

Types de récursivité

Récursivité simple

Un algorithme est dit simplement récursif s'il ne fait qu'un seul appel à lui-même.

Exemple de récursivité simple

La factorielle de n

$$n! = n * (n - 1)!,$$

et

$$0! = 1.$$

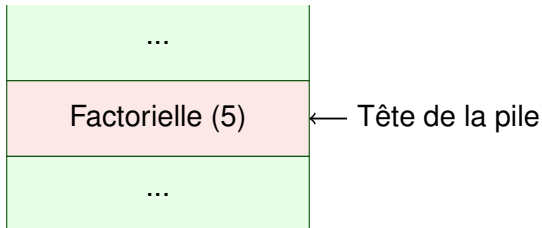
```
int factorielle(int n)
{
    if (n==0) return 1;
    else return factorielle(n-1)*n;
}
```

Exécution de la fonction factielle

La Pile d'exécution (stack) du programme en cours est un emplacement mémoire destiné à mémoriser les paramètres, les variables locales ainsi que les adresses de retour des fonctions en cours d'exécution.

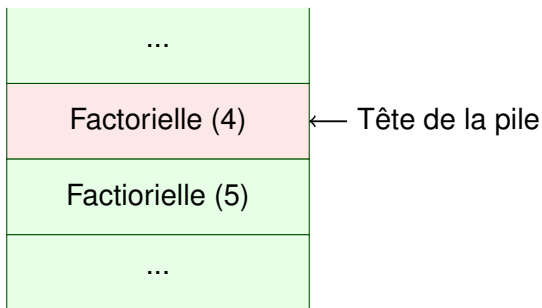
Appel à la fonction récursive : Etape = 1

On empile la fonction factorielle de 5



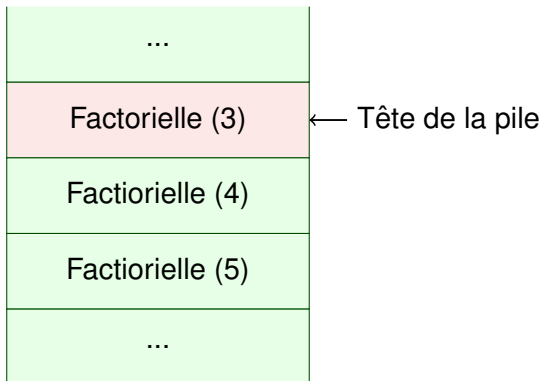
Appel à la fonction récursive : Etape = 2

On empile la fonction factorielle de 4



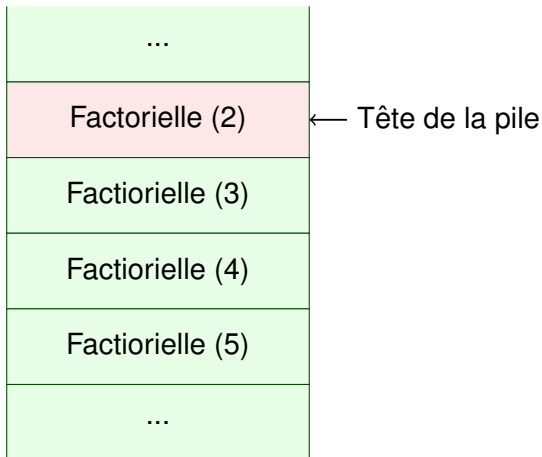
Appel à la fonction récursive : Etape = 3

On empile la fonction factorielle de 3



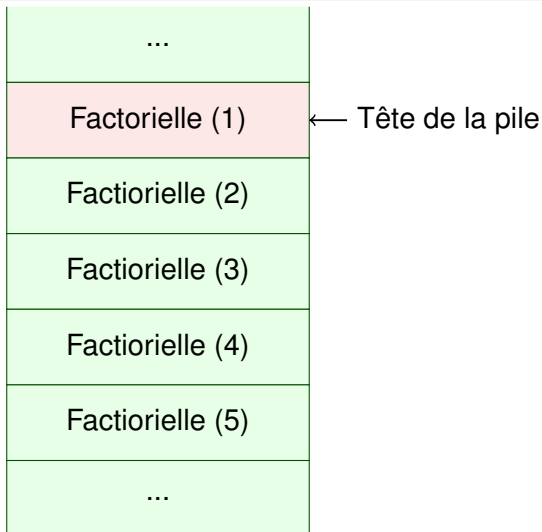
Appel à la fonction récursive : Etape = 4

On empile la fonction factorielle de 2



Appel à la fonction récursive : Etape = 5

On empile la fonction factorielle de 1



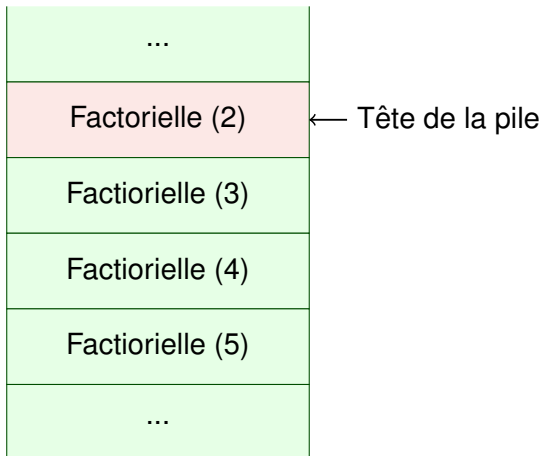
Résultat de la fonction récursive : Factiorelle(1)

On calcule la fonction factorielle de 1

$$\text{Factiorelle}(1) = 1 * \text{Factiorelle}(0) = 1.$$

Retour de la fonction récursive

On depile la fonction factorielle (1)



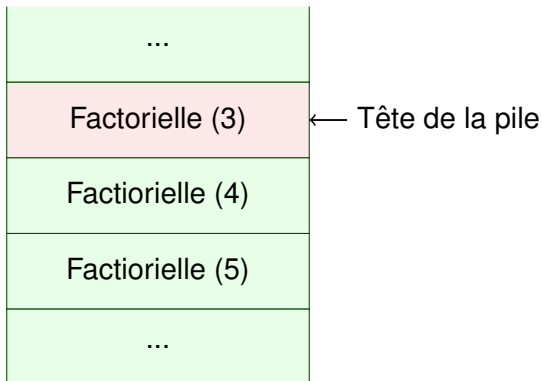
Résultat de la fonction récursive : Factiorelle(2)

On calcule la fonction factorielle de 2

$$\text{Factiorelle}(2) = 2 * \text{Factiorelle}(1) = 2.$$

Retour de la fonction récursive

On depile la fonction factorielle (2)



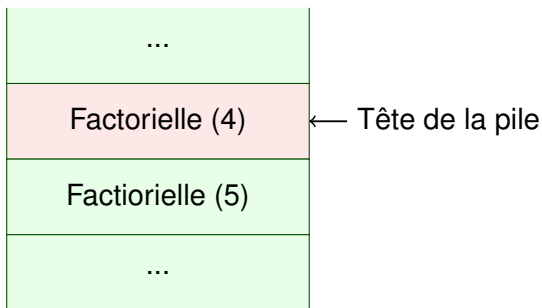
Résultat de la fonction récursive : Factiorelle(3)

On calcule la fonction factorielle de 3

$$\text{Factiorelle}(3) = 3 * \text{Factiorelle}(2) = 6.$$

Retour de la fonction récursive

On depile la fonction factorielle (3)



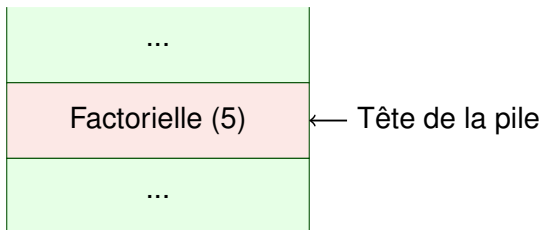
Résultat de la fonction récursive : Factiorelle(4)

On calcule la fonction factorielle de 4

$$\text{Factiorelle}(4) = 4 * \text{Factiorelle}(3) = 24.$$

Retour de la fonction récursive

On depile la fonction factorielle (4)



Résultat de la fonction récursive : Factiorelle(5)

On calcule la fonction factorielle de 5

$$\text{Factiorelle}(5) = 5 * \text{Factiorelle}(4) = 120.$$

Retour de la fonction récursive

On depile la fonction factorielle (5)

