Récursivité

Objectifs

- Rappels : pile d’exécution
- Notion de récursivité
- Notion de récursivité terminale

Rappels : pile d’exécution

- La **pile d’exécution** d’un programme est un emplacement **mémoire** qui contient l’ensemble des **paramètres** d’une fonctions, **ses variables locales** et sa **valeur de retour**. Elle contient aussi **l’adresse de retour** de la fonction appelante
  - À chaque appel d’une fonction / méthode sont empilés :
    - un emplacement pour la valeur de retour
    - l’adresse de retour
    - les paramètres
  - Généralement, dans nos représentations schématiques (Pile / Tas), nous ne représentons pas la valeur de retour ni l’adresse de retour
- Comme vous avez pu le constater, la **pile a une capacité limitée**. **Si** vous la **dépassez**, vous aurez une erreur de type « **StackOverflow** »

Notion de récursivité

- Toute entité est dite récursive si elle se définit à partir d'elle-même
- Plus spécifiquement :
  - Une structure est dit récursive si elle contient un ou des membres de son propre type
    - Exemple : la structure nœud de la liste chainée
  - **Un fonction est dite récursive si elle s'appelle elle-même**
    - Exemple : la fonction partitionner du tri rapide

Notion de récursivité

- Pour définir une fonction récursive, vous avez généralement deux parties :
  - Condition d'arrêt : comment arrêter les appels à soi-même ?
  - Rappel de nous-même avec une variation dans les paramètres
- Factorielle :

$$\begin{cases} f(0) = 1 \\ f(n) = n * f(n-1) \end{cases}$$

- Des variantes existent :
  - Condition de continuation à la place d'une condition d'arrêt
  - Variations des paramètres dépendants de conditions
  - Etc.

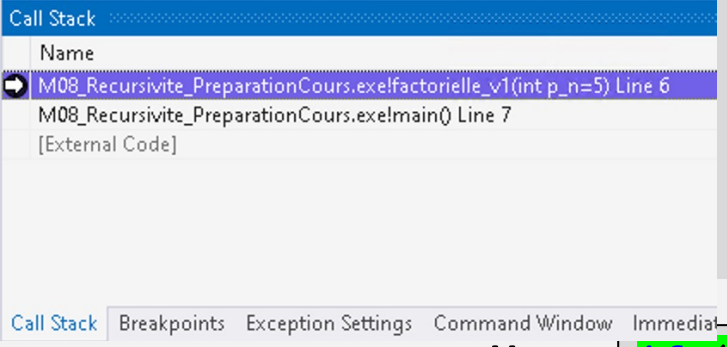
Exemple classique #1

- Factorielle :

$$\begin{cases} f(0) = 1 \\ f(n) = n * f(n - 1) \end{cases}$$

```
int factorielle_v1(int p_n) { // Fonction publique
    if (p_n < 0) {
        throw std::invalid_argument("La valeur ne doit pas être négative");
    }

    // f(0) = 1
    if (p_n == 0) {
        return 1;
    }

    // f(n) = n * f(n - 1)
    return p_n * factorielle_v1(p_n - 1);
}
```

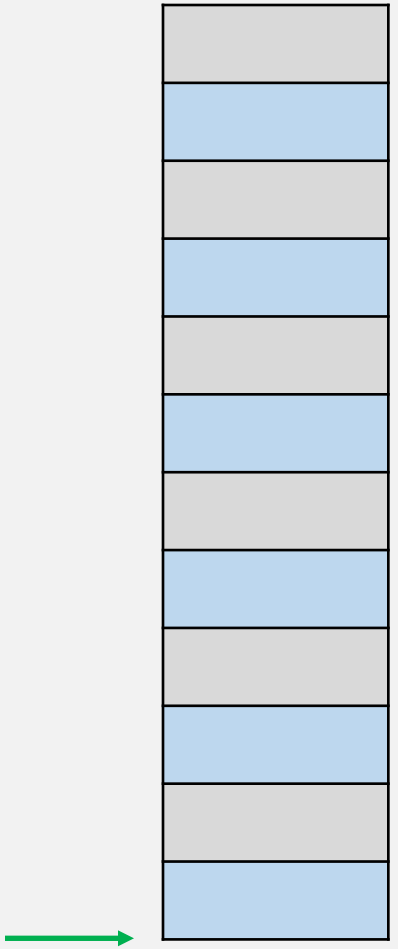
Exemple classique #1



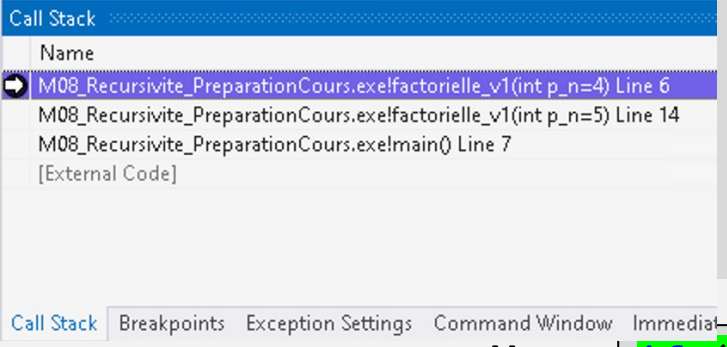
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

f(5) :



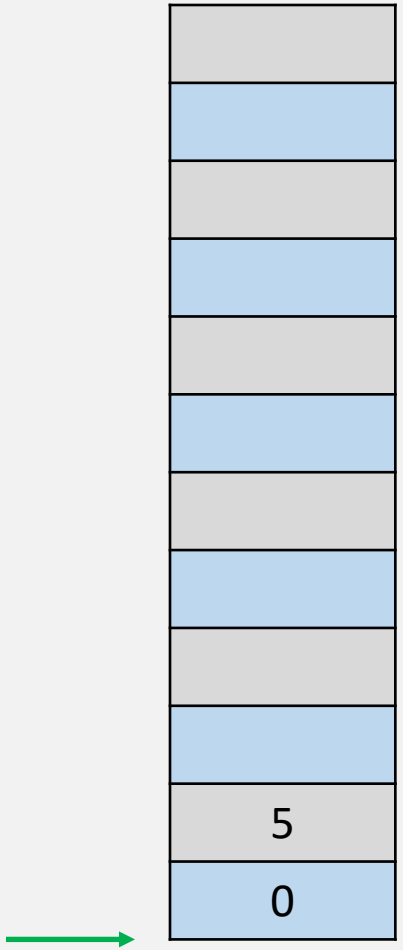
Exemple classique #1



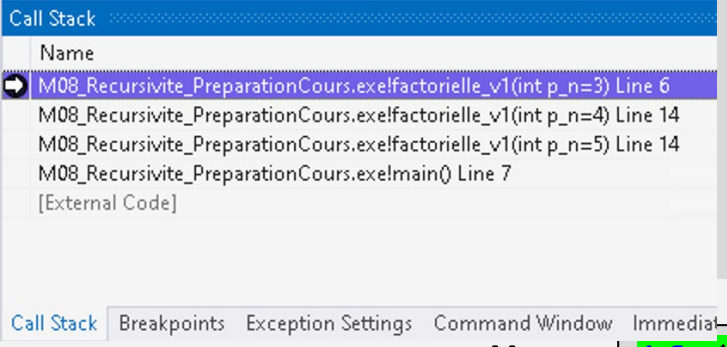
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
```



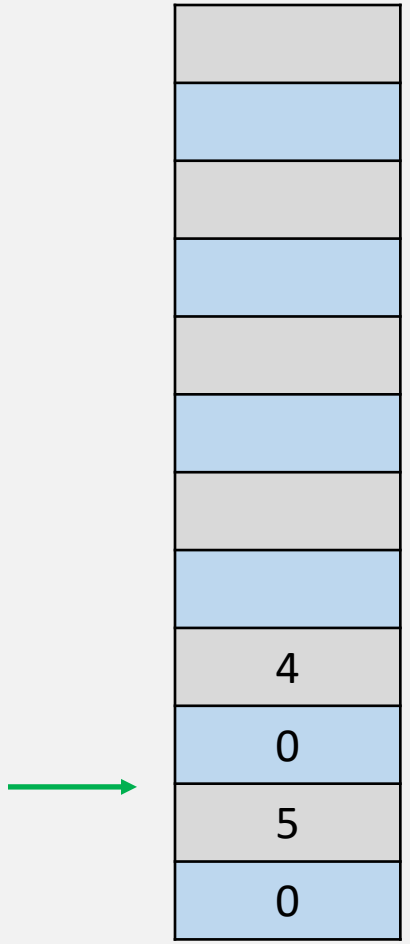
Exemple classique #1



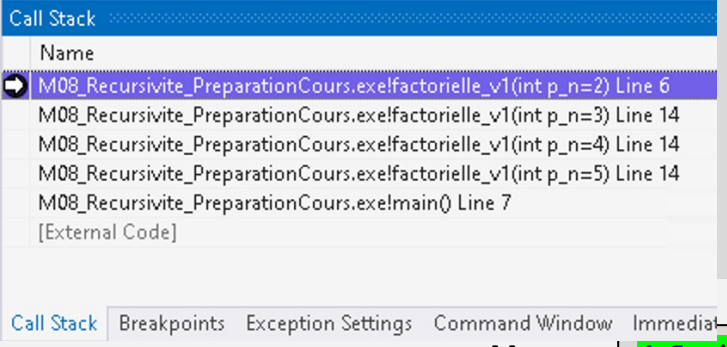
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
              4 * f(3)
```



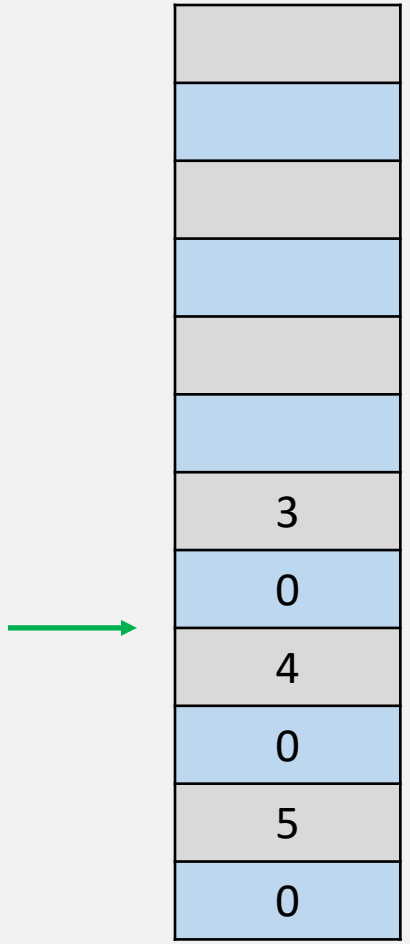
Exemple classique #1



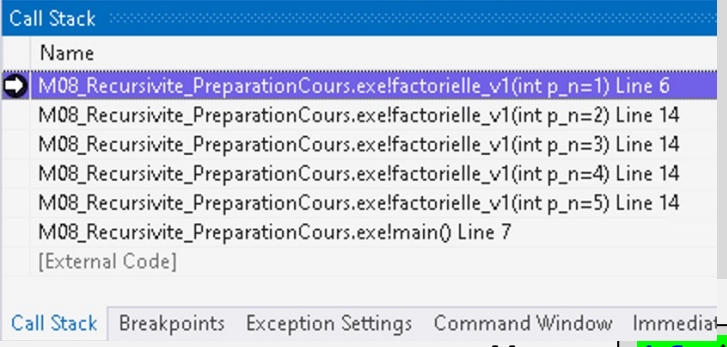
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
              4 * f(3)
                    3 * f(2)
```



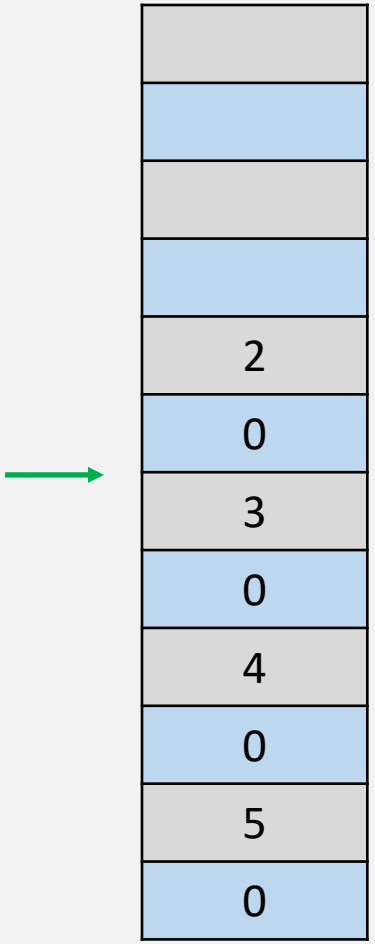
Exemple classique #1



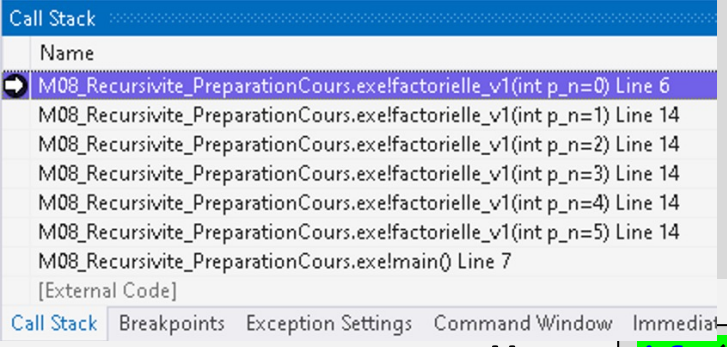
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
              4 * f(3)
                    3 * f(2)
                          2 * f(1)
```



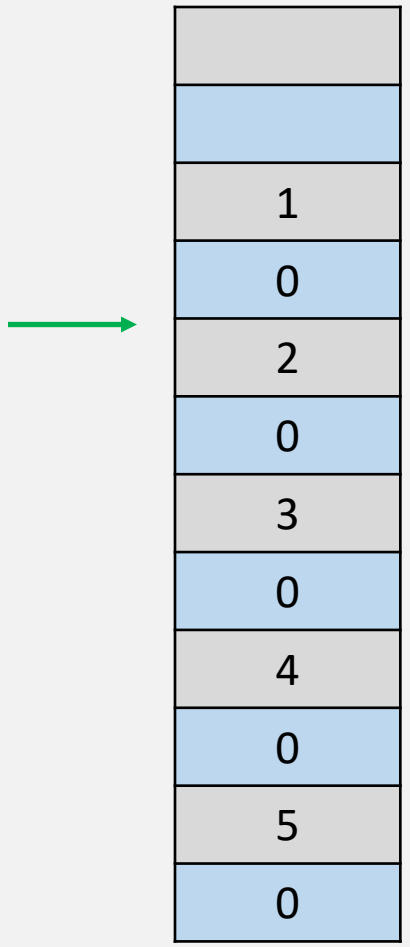
Exemple classique #1



- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
              4 * f(3)
                    3 * f(2)
                          2 * f(1)
                                1 * f(0)
```



Exemple classique #1

```
Call Stack
Name
M08_Recursivite_PreparationCours.exe!factorielle_v1(int p_n=0) Line 6
M08_Recursivite_PreparationCours.exe!factorielle_v1(int p_n=1) Line 14
M08_Recursivite_PreparationCours.exe!factorielle_v1(int p_n=2) Line 14
M08_Recursivite_PreparationCours.exe!factorielle_v1(int p_n=3) Line 14
M08_Recursivite_PreparationCours.exe!factorielle_v1(int p_n=4) Line 14
M08_Recursivite_PreparationCours.exe!factorielle_v1(int p_n=5) Line 14
M08_Recursivite_PreparationCours.exe!main() Line 7
[External Code]
Call Stack  Breakpoints  Exception Settings  Command Window  Immediat
```

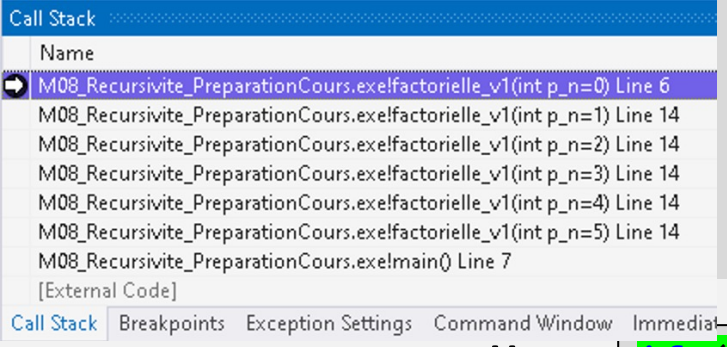
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
              4 * f(3)
                    3 * f(2)
                          2 * f(1)
                                1 * f(0)
```


0
0
1
0
2
0
3
0
4
0
5
0

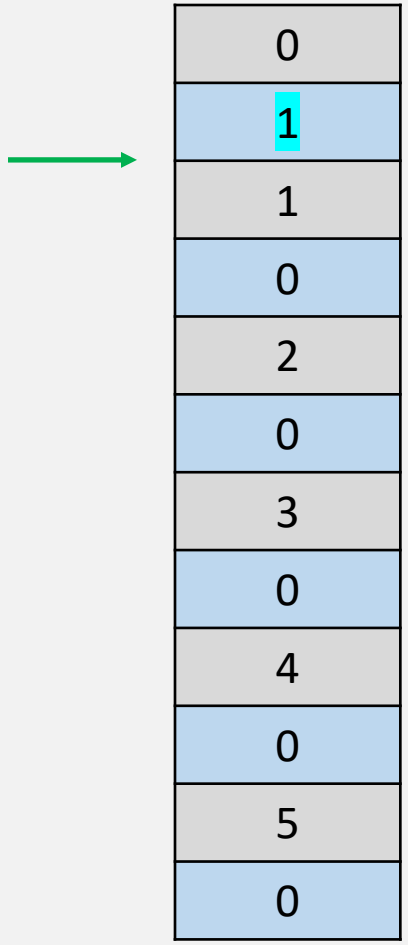
Exemple classique #1



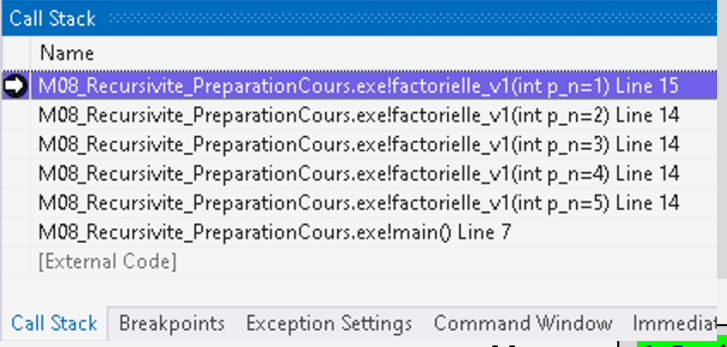
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
              4 * f(3)
                    3 * f(2)
                          2 * f(1)
                                1 * f(0)
                                      1
```



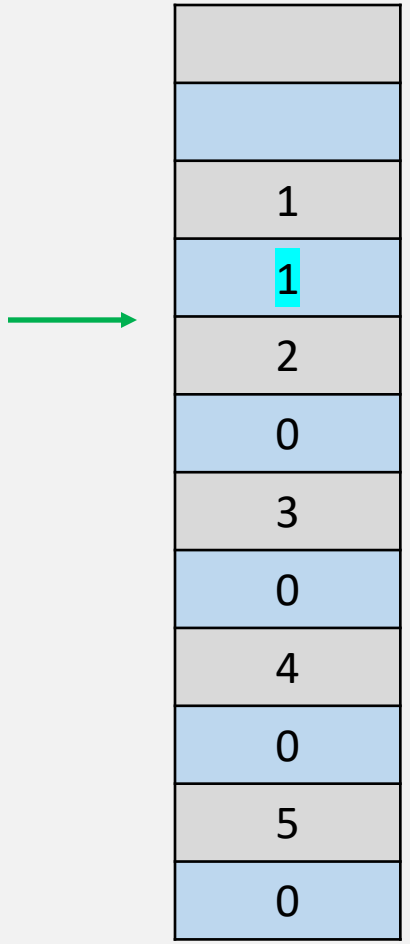
Exemple classique #1



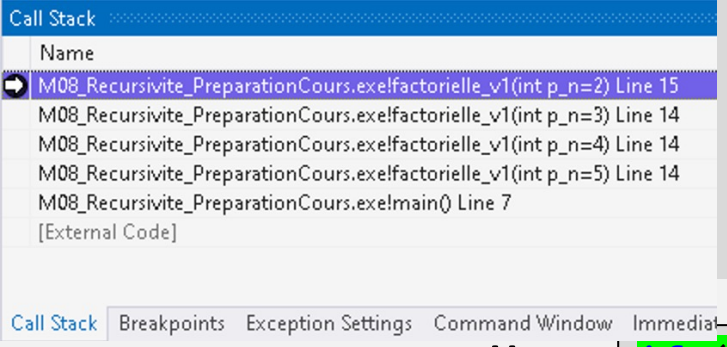
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
              4 * f(3)
                    3 * f(2)
                          2 * f(1)
                                1 * f(0)
                                      1
                                1 * 1 = 1
```



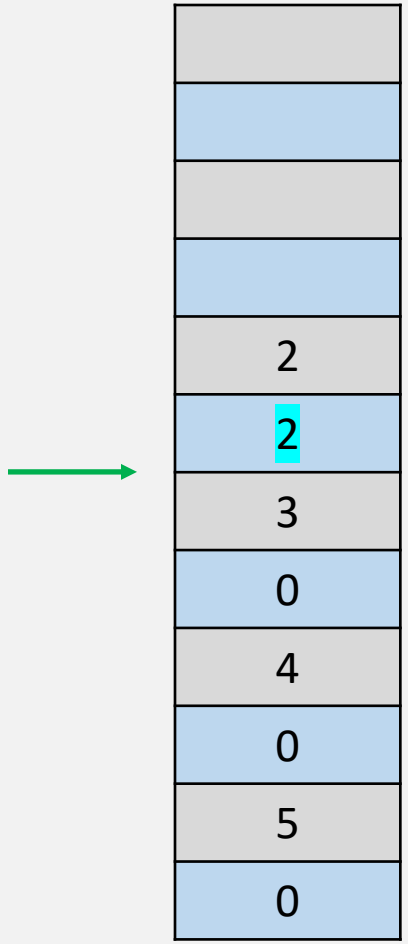
Exemple classique #1



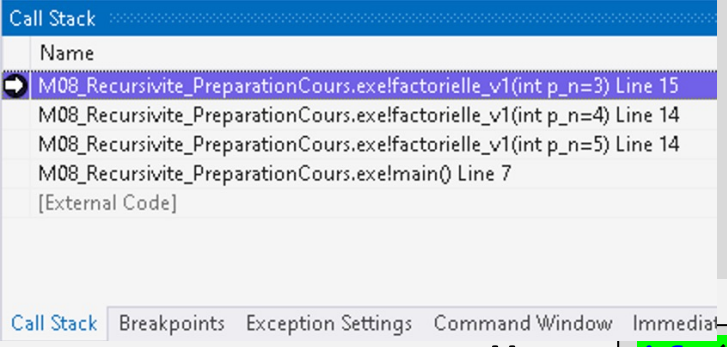
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
              4 * f(3)
                    3 * f(2)
                          2 * f(1)
                                1 * f(0)
                                      1
                                1 * 1 = 1
                          2 * 1 = 2
```



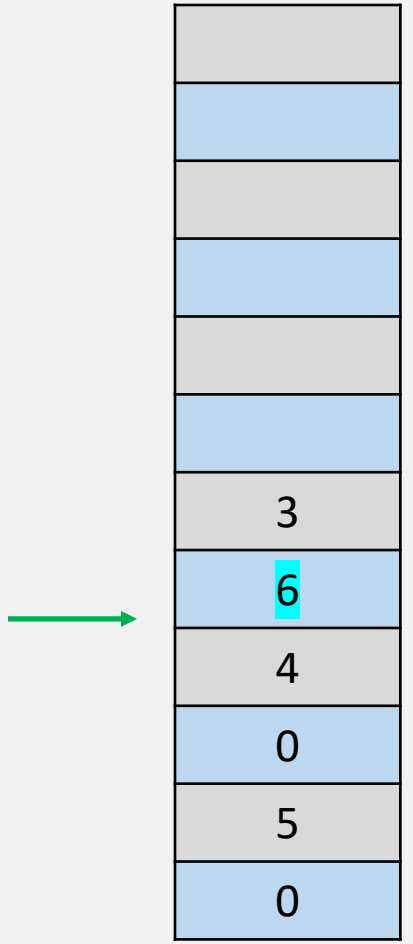
Exemple classique #1



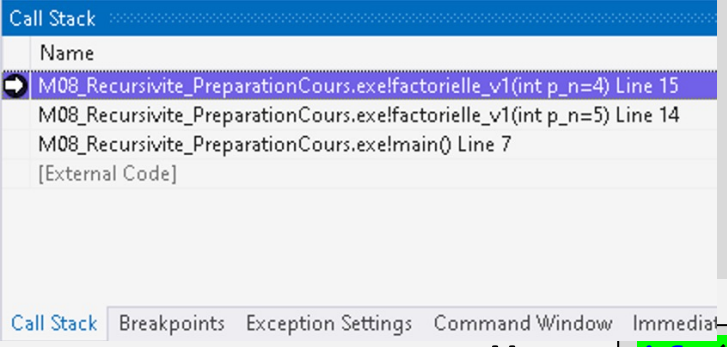
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
            4 * f(3)
                3 * f(2)
                    2 * f(1)
                        1 * f(0)
                            1
                        1 * 1 = 1
                    2 * 1 = 2
                3 * 2 = 6
```



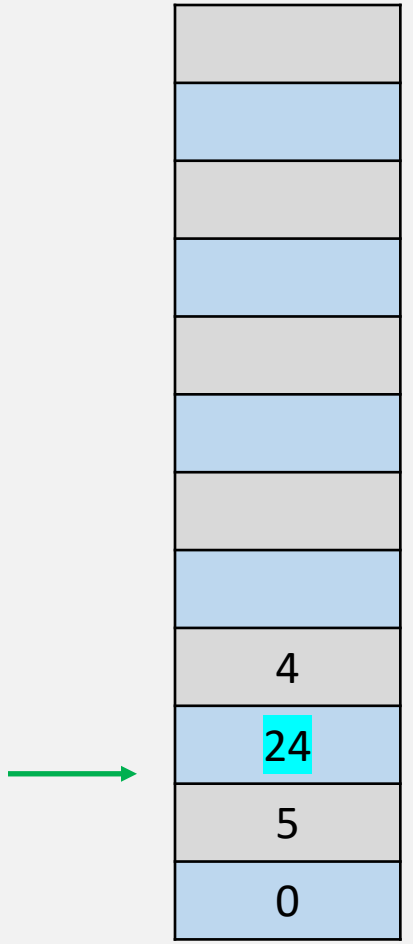
Exemple classique #1



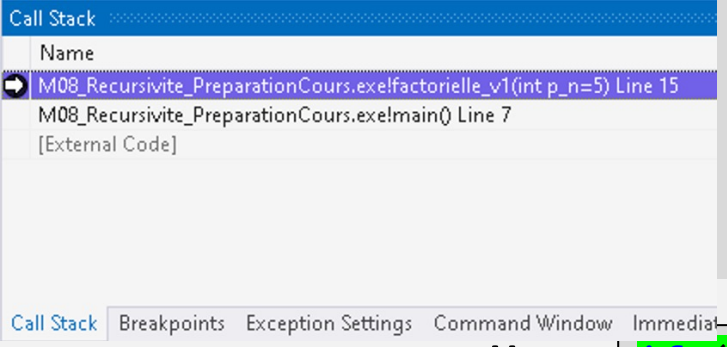
- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
            4 * f(3)
                3 * f(2)
                    2 * f(1)
                        1 * f(0)
                            1
                        1 * 1 = 1
                    2 * 1 = 2
                3 * 2 = 6
            4 * 6 = 24
```



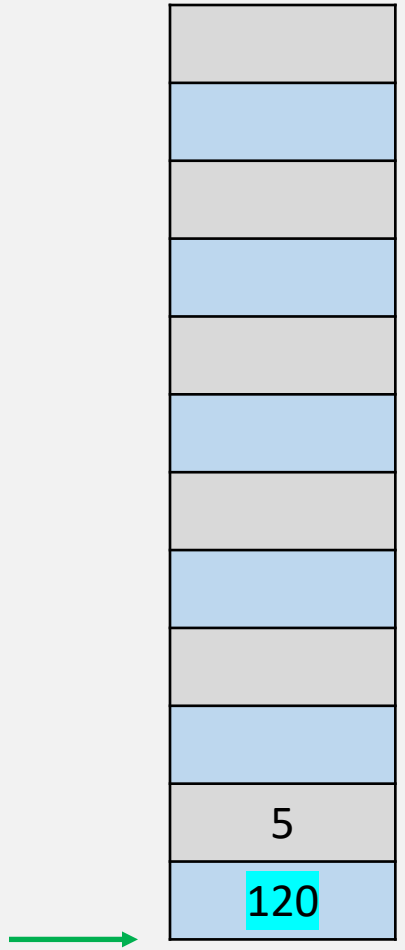
Exemple classique #1



- Factorielle :

```
if (p_n == 0) {
        return 1;
}

return p_n * factorielle_v1(p_n - 1);
```

```
f(5) :
        5 * f(4)
            4 * f(3)
                3 * f(2)
                    2 * f(1)
                        1 * f(0)
                            1
                        1 * 1 = 1
                    2 * 1 = 2
                3 * 2 = 6
            4 * 6 = 24
        5 * 24 = 120
120
```



Exemple classique #1 – V2

- Factorielle :

$$\begin{cases} f(0) = 1 \\ f(n) = n * f(n-1) \end{cases}$$

```
int factorielle_v2(int p_n) { // Fonction publique
    if (p_n < 0) {
        throw std::invalid_argument("La valeur ne doit pas être négative");
    }

    return factorielle_v2_rec(p_n);
}

int factorielle_v2_rec(int p_n) { // Fonction privée
    if (p_n == 0) {
        return 1;
    }

    return p_n * factorielle_v2_rec(p_n - 1);
}
```

Exemple classique #2

- Remplacer une boucle – calculer la somme des nombres de 1 à n

```
int calculerSomme1AN_v1(int p_n) { // Fonction publique
    if (p_n < 1) {
        throw std::invalid_argument("La valeur doit être supérieure à 0");
    }

    return calculerSomme1AN_v1_rec(p_n);
}

int calculerSomme1AN_v1_rec(int p_n) { // Fonction privée
    if (p_n == 1) {
        return 1;
    }

    return p_n + calculerSomme1AN_v1_rec(p_n - 1);
}
```

Exemple classique #3

- Remplacer une boucle – afficher les nombres de 1 à n

```
void afficherNombresDe1AN_v1(int p_n) { // Fonction publique
    if (p_n < 1) {
        throw std::invalid_argument("La valeur doit être supérieure à 0");
    }

    afficherNombresDe1AN_v1_rec(1, p_n);
}

void afficherNombresDe1AN_v1_rec(int p_de, int p_a) { // Fonction privée
    std::cout << p_de << std::endl;

    if (p_de < p_a) {
        afficherNombresDe1AN_v1_rec(p_de + 1, p_a);
    }
}
```

Exemple classique #3

- Remplacer une boucle – afficher les nombres de 1 à n

```
void afficherNombresDe1AN_v2(int p_n) { // Fonction publique
    if (p_n < 1) {
        throw std::invalid_argument("La valeur doit être supérieure à 0");
    }

    afficherNombresDe1AN_v2_rec(p_n);
}

void afficherNombresDe1AN_v2_rec(int p_n) { // Fonction privée
    if (p_n > 1) {
        afficherNombresDe1AN_v2_rec(p_n - 1);
    }

    std::cout << p_n << std::endl;
}
```

Récursivité terminale

- Une fonction est dite **récursive terminale** si la valeur de retour **ne dépend que de l'appel subséquent**, c'est-à-dire si le retour est de la forme « `return f(…)` » ou « `f(…)` » où f est le nom de la fonction récursive.

```
void afficherNombresDe1AN_v3_rec(int p_de, int p_a) {
    std::cout << p_de << std::endl;

    if (p_de < p_a) {
        afficherNombresDe1AN_v3_rec(p_de + 1, p_a);
    }
}
```

Récursivité terminale

```
// Fonction privée
```

```
int factorielle_v2_rec(int p_n) {
    if (p_n == 0) {
        return 1;
    }

    return p_n * factorielle_v2_rec(p_n - 1);
}
```

Non récursivité terminale

```
// Fonction privée
```

Récursivité terminale – Transformation

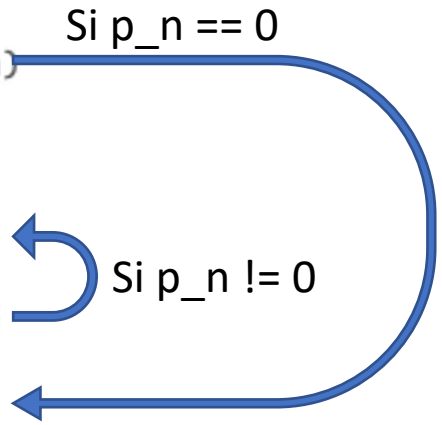
```
int factorielle_v2_rec(int p_n) { // Fonction privée
    if (p_n == 0) {
        return 1;
    }

    return p_n * factorielle_v2_rec(p_n - 1);
}
```

```
int factorielle_v3(int p_n) { // Fonction publique
    if (p_n < 0) {
        throw std::invalid_argument("La valeur ne doit pas être négative");
    }

    return factorielle_v3_rec(p_n, 1);
}

int factorielle_v3_rec(int p_n, int p_res) { // Fonction privée
    if (p_n == 0) {
        return p_res;
    }
    return factorielle_v3_rec(p_n - 1, p_res * p_n);
}
```

Récursivité terminale – Assembleur (optimisé)

Ici, le compilateur a décidé d'omettre le code de la fonction publique (le code a des appels avec des valeurs constantes et positives)

```
--- C:\repos\420-W31-SF-CPP\Module08_Recursivite\M08_Recursivite_PreparationCours\M08_Recursivite_PreparationCours\factorielle.cpp
    if (p_n == 0) {
00007FF752061320  test        ecx,ecx
00007FF752061322  je          factorielle_v3_rec+0Ch (07FF75206132Ch)
        return p_res;
    }
    return factorielle_v3_rec(p_n - 1, p_res * p_n);
00007FF752061324  imul        edx,ecx
00007FF752061327  sub         ecx,1
00007FF75206132A  jne         factorielle_v3_rec+4h (07FF752061324h)
}
00007FF75206132C  mov         eax,edx
00007FF75206132E  ret
```



- En version optimisée :
  - le registre **ECX** contient **p_n**
  - le registre **EDX** contient **p_res**
  - **retour** le registre **EAX**
- « **`test ecx, ecx`** » permet de positionner ZF (Zero flag) à 0 si **ECX (p_n)** == 0 (il fait un ET logique bit à bit)
- « **`sub ecx, 1`** » permet de positionner ZF (Zero flag) à 0 si **ECX (p_n)** == 0 après sa décrémentation

Références

- Version wikipedia : https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9cursif
- Passage de paramètres : https://stackoverflow.com/questions/60261705/why-functions-locals-and-arguments-are-pushed-to-the-stack
- Instructions assembleurs de contrôle de flux : https://en.wikibooks.org/wiki/X86_Assembly/Control_Flow
- Culture générale ALU : https://en.wikibooks.org/wiki/X86_Assembly/Arithmetic
- Culture générale FPU : https://en.wikibooks.org/wiki/X86_Assembly/Floating_Point