Programmation des appels de procédure et fonction (fin)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

17 mars 2020

Plan

1. Rappels
2. Gestion du résultat
3. Variables locales et temporaires
4. Passage par *adresse*
5. Conclusions

Plan

1 Rappels

2 Gestion du résultat

3 Variables locales et temporaires

4 Passage par *adresse*

5 Conclusions

Organisation du code (rappel)

La gestion des appels de fonction est partagée entre l'appelant et l'appelé :

Organisation du code (rappel)

La gestion des appels de fonction est partagée entre l'appelant et l'appelé :

appelant `P` :
préparer et empiler les paramètres
appeler `Q` : `BL Q`
libérer la place allouée aux paramètres

Organisation du code (rappel)

La gestion des appels de fonction est partagée entre l'appelant et l'appelé :

appelant `P` :
préparer et empiler les paramètres
appeler `Q` : `BL Q`
libérer la place allouée aux paramètres

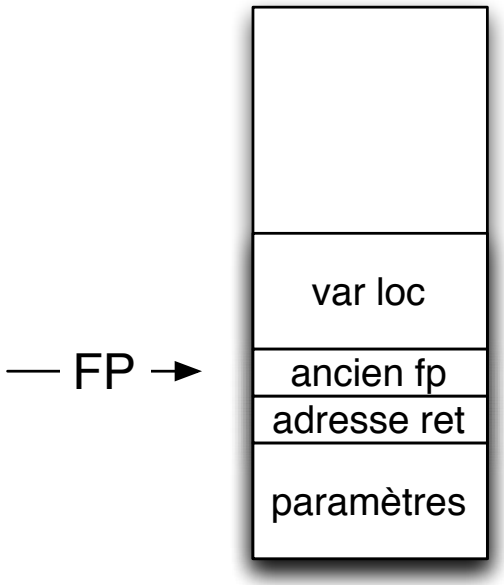
appelé `Q` :
empiler l'adresse de retour
empiler la valeur de `fp`
placer `fp` pour repérer les nouvelles variables
allouer la place pour les variables locales
corps de la fonction `Q`
libérer la place allouée aux variables locales
dépiler `fp`
dépiler l'adresse de retour
retour à l'appelant (`P`) : `MOV PC, LR` ou `BX LR`

Organisation de la pile lors de l'exécution (rappel)

En cours d'exécution du corps de la fonction appelée :

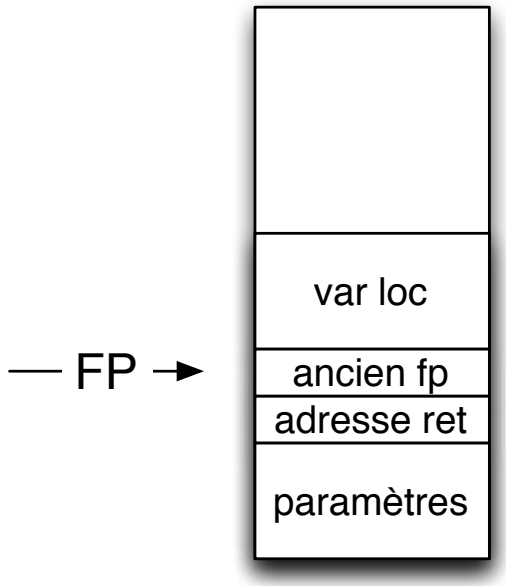
Organisation de la pile lors de l'exécution (rappel)

En cours d'exécution du corps de la fonction appelée :



Organisation de la pile lors de l'exécution (rappel)

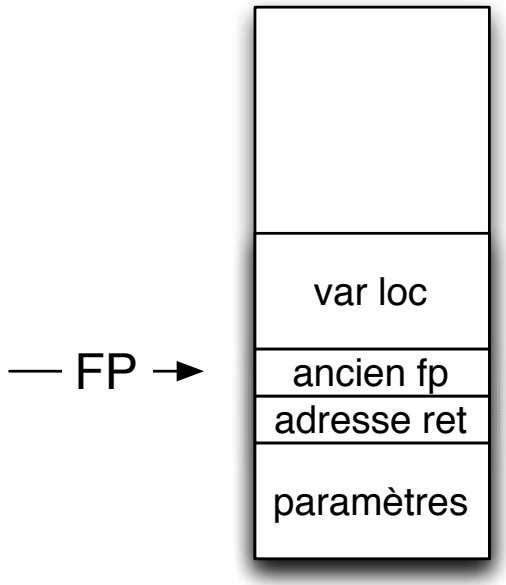
En cours d'exécution du corps de la fonction appelée :



- `fp` − 4 − *déplacement* : Accès aux variables locales

Organisation de la pile lors de l'exécution (rappel)

En cours d'exécution du corps de la fonction appelée :



- `fp` $-$ 4 $-$ *déplacement* : Accès aux variables locales
- `fp` $+8+$ *déplacement* : Accès aux paramètres

Plan

1 Rappels

2 Gestion du résultat

3 Variables locales et temporaires

4 Passage par *adresse*

5 Conclusions

Le resultat : questions

- Qui s'occupe(nt) du résultat ?

Le resultat : questions

- Qui s'occupe(nt) du résultat ?
- Quand s'occuper du résultat ?

Le resultat : questions

- Qui s'occupe(nt) du résultat ?
- Quand s'occuper du résultat ?
- Où mettre le résultat ?

Rappel : les paramètres données (Qui ? Quand ? Où ?)

Les valeurs des paramètres effectifs sont calculées avant l'appel

C'est la procédure appelante qui connait les informations.

Rappel : les paramètres données (Qui ? Quand ? Où ?)

Les valeurs des paramètres effectifs sont calculées avant l'appel

C'est la procédure appelante qui connait les informations.

Deux étapes avant l'appel :

- Evaluation des paramètres,
- Stockage des valeurs des paramètres dans la pile.

Résultat d'une fonction (Qui ? Quand ? Où ?)

Résultat d'une fonction (Qui ? Quand ? Où ?)

1. Le résultat d'une fonction est calculé par l'appelée

Résultat d'une fonction (Qui ? Quand ? Où ?)

1. Le résultat d'une fonction est calculé par l'appelée
2. Le résultat doit être rangé à un emplacement accessible par l'appelante de façon à ce que cette dernière puisse le récupérer.

Il faut donc utiliser une zone mémoire commune à l'appelante et l'appelée.

Résultat d'une fonction (Qui ? Quand ? Où ?)

1. Le résultat d'une fonction est calculé par l'appelée
2. Le résultat doit être rangé à un emplacement accessible par l'appelante de façon à ce que cette dernière puisse le récupérer.

Il faut donc utiliser une zone mémoire commune à l'appelante et l'appelée.

Par l'exemple, **la pile**.

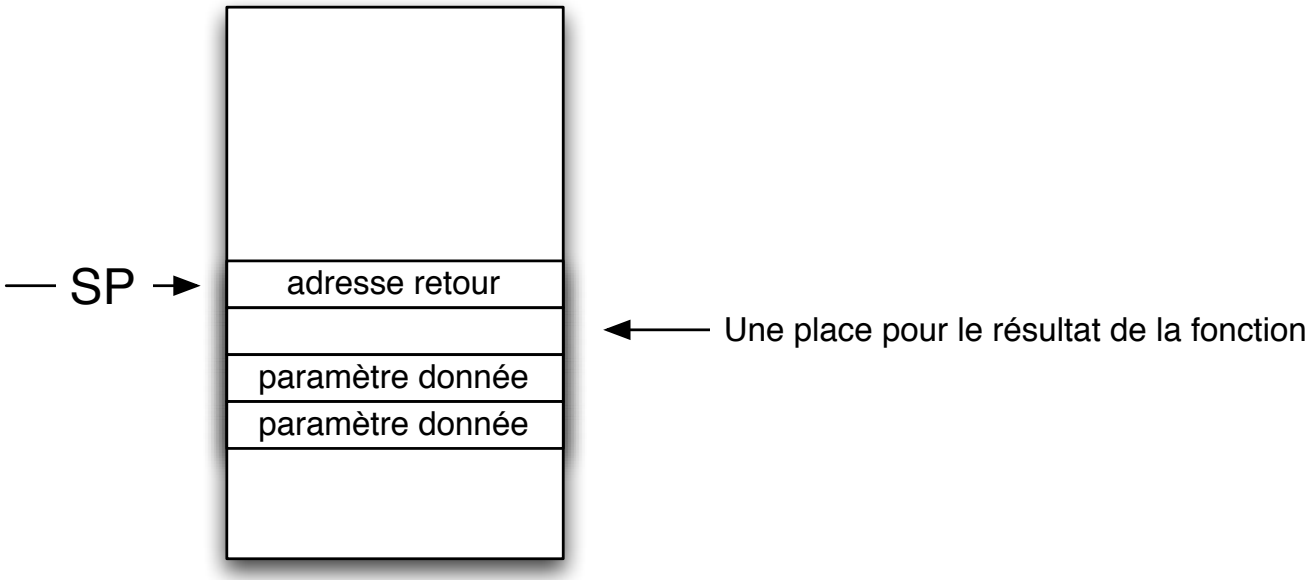
Résultat dans la pile (1/3)

1. avant l'appel, **L'appelant** réserve une place pour le résultat dans la pile
2. **L'appelée** rangera son résultat dans cette case dont le contenu sera récupéré par l'appelant après le retour

Résultat dans la pile (2/3)

Avant l'appel d'une fonction qui a deux paramètres données

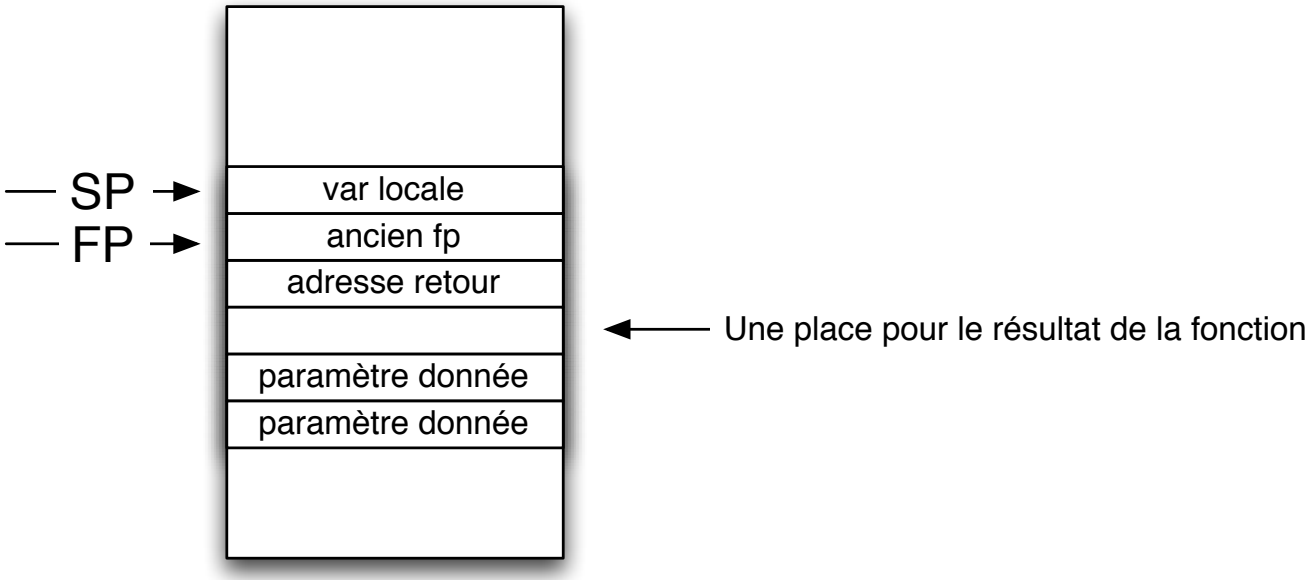
- Les valeurs des deux paramètres sont empilés
- Une case est réservée pour le résultat de la fonction



Résultat dans la pile (3/3)

Lors de l'exécution du corps de la fonction.

1. Les variables locales sont accessibles par une adresse de la forme : $fp - 4 - depl$ avec $depl \geq 0$,
2. Les paramètres données par les adresses : $fp + 8 + 4$ et $fp + 8 + 8$ et
3. La case résultat par l'adresse $fp + 8$.



Structure du code de l'appel de la fonction et du corps de la fonction

appelant `P` :
préparer et empiler les paramètres
réserver la place du résultat dans la pile
appeler `Q` : `BL Q`
récupérer le résultat
libérer la place allouée au résultat
libérer la place allouée aux paramètres

Structure du code de l'appel de la fonction et du corps de la fonction

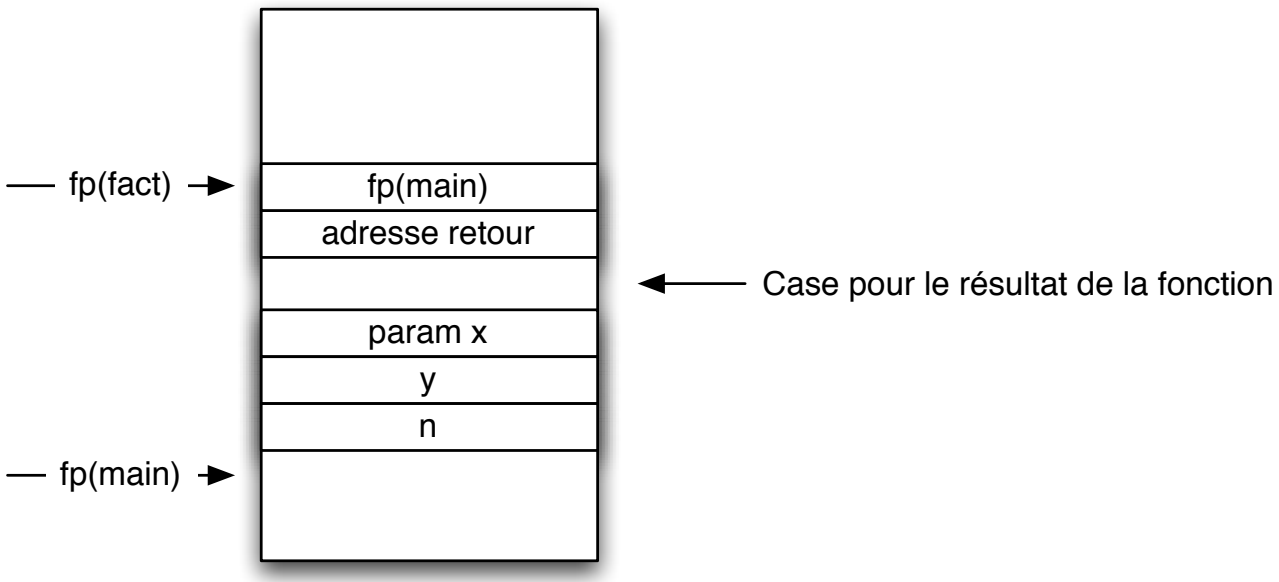
appelant `P` :
préparer et empiler les paramètres
réserver la place du résultat dans la pile
appeler `Q` : `BL Q`
récupérer le résultat
libérer la place allouée au résultat
libérer la place allouée aux paramètres

appelé `Q` :
empiler l'adresse de retour
empiler la valeur de `fp`
placer `fp` pour repérer les nouvelles variables
allouer la place pour les variables locales
corps de la fonction `Q`
le résultat est rangé en **fp+8**
libérer la place allouée aux variables locales
dépiler `fp`
dépiler l'adresse de retour
retour à l'appelant (`P`) : `MOV pc,lr` ou `BX lr`

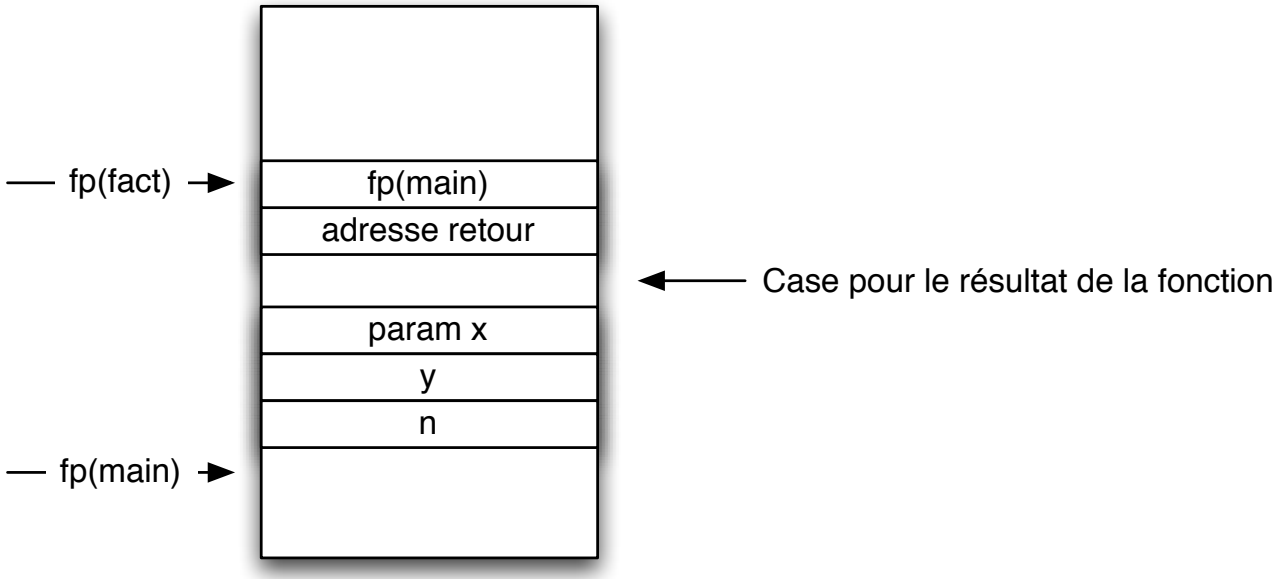
Application : codage de la fonction factorielle

```
int fact (int x) {
    if (x==0) then return 1
    else return x * fact(x-1);}

main(){
int n, y;
    ...
    y = fact(n);
    ...
}
```

Application : état de la pile lors de l'exécution du corps de fact



Application : état de la pile lors de l'exécution du corps de fact



Remarque : Dans cet exemple comme il n'y a pas de variables locales on pourrait se passer de mettre en place `fp` pour la fonction appelée. On l'a tout de même utilisé de façon à adresser les paramètres toujours de la même façon.

Application : la fonction `main`

```
main:
        ...
                                                    @ r1 ← n
        @ sequence d'appel de fact(n)

        bl fact                                     @ appel
                                                    @ recuperation du resultat dans r1
                                                    @ on range le resultat dans y
                                                    @ on recupere la place allouee
                                                    @ au resultat et au parametre
```

Application : la fonction `main`

```
main:
        ...
        ldr r1, [fp, #-4]          @ r1 ← n
        @ sequence d'appel de fact(n)

        bl fact                    @ appel
                                   @ recuperation du resultat dans r1
                                   @ on range le resultat dans y
                                   @ on recupere la place allouee
                                   @ au resultat et au parametre
```

Application : la fonction `main`

```
main:
        ...
        ldr r1, [fp, #-4]                @ r1 ← n
        @ sequence d'appel de fact(n)
        @ empiler la valeur de n, parametre donnee
        sub sp, sp, #4
        str r1, [sp]


        bl fact                          @ appel
                                         @ recuperation du resultat dans r1
                                         @ on range le resultat dans y
                                         @ on recupere la place allouee
                                         @ au resultat et au parametre
```

Application : la fonction `main`

```
main:
        ...
        ldr r1, [fp, #-4]                @ r1 ← n
        @ sequence d'appel de fact(n)
        @ empiler la valeur de n, parametre donnee
        sub sp, sp, #4
        str r1, [sp]
        @ reserver la place pour le resultat de fact
        sub sp, sp, #4
        bl fact                          @ appel
                                         @ recuperation du resultat dans r1
                                         @ on range le resultat dans y
                                         @ on recupere la place allouee
                                         @ au resultat et au parametre
```

Application : la fonction `main`

```
main:
        ...
        ldr r1, [fp, #-4]                @ r1 ← n
        @ sequence d'appel de fact(n)
        @ empiler la valeur de n, parametre donnee
        sub sp, sp, #4
        str r1, [sp]
        @ reserver la place pour le resultat de fact
        sub sp, sp, #4
        bl fact                          @ appel
        ldr r1, [sp]                     @ recuperation du resultat dans r1
                                         @ on range le resultat dans y
                                         @ on recupere la place allouee
                                         @ au resultat et au parametre
```

Application : la fonction `main`

```
main:
        ...
        ldr r1, [fp, #-4]                @ r1 ← n
        @ sequence d'appel de fact(n)
        @ empiler la valeur de n, parametre donnee
        sub sp, sp, #4
        str r1, [sp]
        @ reserver la place pour le resultat de fact
        sub sp, sp, #4
        bl fact                          @ appel
        ldr r1, [sp]                     @ recuperation du resultat dans r1
        str r1, [fp, #-8]                @ on range le resultat dans y
                                         @ on recupere la place allouee
                                         @ au resultat et au parametre
```

Application : la fonction `main`

```
main:
        ...
        ldr r1, [fp, #-4]                @ r1 ← n
        @ sequence d'appel de fact(n)
        @ empiler la valeur de n, parametre donnee
        sub sp, sp, #4
        str r1, [sp]
        @ reserver la place pour le resultat de fact
        sub sp, sp, #4
        bl fact                          @ appel
        ldr r1, [sp]                     @ recuperation du resultat dans r1
        str r1, [fp, #-8]                @ on range le resultat dans y
        add sp, sp, #8                   @ on recupere la place allouee
                                         @ au resultat et au parametre
```

Application : la fonction `fact`

```
fact:
        @ empiler lr

        @ sauvegarde ancien fp et place fp(fact)


        @ recuperer la valeur du parametre x : r0=x

        cmp r0, #0
        bne sinon
alors:  @ ranger le resultat (1)

        b fsi
sinon:  @ r0 contient x
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1

        @ empiler x-1


        @ place pour resultat

        bl fact
        @ recuperer le resultat de l'appel r1=fact(x-1)

        @ liberation parametre et resultat

        @ r0=x

        @ r0 * r1 = x * fact(x-1)
        mul r3, r0, r1
        @ ranger le resultat

fsi:
        @ recuperer fp


        @ retour
```

Application : la fonction `fact`

```
fact:
        @ empiler lr
        sub sp, sp, #4
        str lr, [sp]
        @ sauvegarde ancien fp et place fp(fact)

        @ recuperer la valeur du parametre x : r0=x

        cmp r0, #0
        bne sinon
alors:  @ ranger le resultat (1)

        b fsi
sinon:  @ r0 contient x
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1

        @ empiler x-1

        @ place pour resultat

        bl fact
        @ recuperer le resultat de l'appel r1=fact(x-1)

        @ liberation parametre et resultat

        @ r0=x

        @ r0 * r1 = x * fact(x-1)
        mul r3, r0, r1
        @ ranger le resultat

fsi:
        @ recuperer fp

        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr
```

Application : la fonction `fact`

```
fact:
        @ empiler lr
        sub sp, sp, #4
        str lr, [sp]
        @ sauvegarde ancien fp et place fp(fact)
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        @ recuperer la valeur du parametre x : r0=x

        cmp r0, #0
        bne sinon
alors:  @ ranger le resultat (1)

        b fsi
sinon:  @ r0 contient x
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1
        @ empiler x-1

        @ place pour resultat

        bl fact
        @ recuperer le resultat de l'appel r1=fact(x-1)

        @ liberation parametre et resultat

        @ r0=x

        @ r0 * r1 = x * fact(x-1)
        mul r3, r0, r1
        @ ranger le resultat

fsi:
        @ recuperer fp
        ldr fp, [sp]
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr
```

Application : la fonction `fact`

```
fact:
        @ empiler lr
        sub sp, sp, #4
        str lr, [sp]
        @ sauvegarde ancien fp et place fp(fact)
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        @ recuperer la valeur du parametre x : r0=x
        ldr r0, [fp, #+12]
        cmp r0, #0
        bne sinon
alors:  @ ranger le resultat (1)

        b fsi
sinon:  @ r0 contient x
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1
        @ empiler x-1

        @ place pour resultat

        bl fact
        @ recuperer le resultat de l'appel r1=fact(x-1)

        @ liberation parametre et resultat

        @ r0=x

        @ r0 * r1 = x * fact(x-1)
        mul r3, r0, r1
        @ ranger le resultat

fsi:
        @ recuperer fp
        ldr fp, [sp]
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr
```

Application : la fonction `fact`

```
fact:
        @ empiler lr
        sub sp, sp, #4
        str lr, [sp]
        @ sauvegarde ancien fp et place fp(fact)
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        @ recuperer la valeur du parametre x : r0=x
        ldr r0, [fp, #+12]
        cmp r0, #0
        bne sinon
alors:  @ ranger le resultat (1)
        mov r1, #1
        str r1, [fp, #+8]
        b fsi
sinon:  @ r0 contient x
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1
        @ empiler x-1


        @ place pour resultat

        bl fact
        @ recuperer le resultat de l'appel r1=fact(x-1)

        @ liberation parametre et resultat

        @ r0=x

        @ r0 * r1 = x * fact(x-1)
        mul r3, r0, r1
        @ ranger le resultat

fsi:
        @ recuperer fp
        ldr fp, [sp]
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr
```

Application : la fonction `fact`

```
fact:
        @ empiler lr
        sub sp, sp, #4
        str lr, [sp]
        @ sauvegarde ancien fp et place fp(fact)
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        @ recuperer la valeur du parametre x : r0=x
        ldr r0, [fp, #+12]
        cmp r0, #0
        bne sinon
alors:  @ ranger le resultat (1)
        mov r1, #1
        str r1, [fp, #+8]
        b fsi
sinon:  @ r0 contient x
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1
        sub r1, r0, #1
        @ empiler x-1


        @ place pour resultat

        bl fact
        @ recuperer le resultat de l'appel r1=fact(x-1)

        @ liberation parametre et resultat

        @ r0=x

        @ r0 * r1 = x * fact(x-1)
        mul r3, r0, r1
        @ ranger le resultat

fsi:
        @ recuperer fp
        ldr fp, [sp]
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr
```

Application : la fonction `fact`

```
fact:
        @ empiler lr
        sub sp, sp, #4
        str lr, [sp]
        @ sauvegarde ancien fp et place fp(fact)
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        @ recuperer la valeur du parametre x : r0=x
        ldr r0, [fp, #+12]
        cmp r0, #0
        bne sinon
alors:  @ ranger le resultat (1)
        mov r1, #1
        str r1, [fp, #+8]
        b fsi
sinon:  @ r0 contient x
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1
        sub r1, r0, #1
        @ empiler x-1
        sub sp, sp, #4
        str r1, [sp]
        @ place pour resultat
        sub sp, sp, #4
        bl fact
        @ recuperer le resultat de l'appel r1=fact(x-1)

        @ liberation parametre et resultat

        @ r0=x

        @ r0 * r1 = x * fact(x-1)
        mul r3, r0, r1
        @ ranger le resultat

fsi:

        @ recuperer fp
        ldr fp, [sp]
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr
```

Application : la fonction `fact`

```
fact:
        @ empiler lr
        sub sp, sp, #4
        str lr, [sp]
        @ sauvegarde ancien fp et place fp(fact)
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        @ recuperer la valeur du parametre x : r0=x
        ldr r0, [fp, #+12]
        cmp r0, #0
        bne sinon
alors:  @ ranger le resultat (1)
        mov r1, #1
        str r1, [fp, #+8]
        b fsi
sinon:  @ r0 contient x
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1
        sub r1, r0, #1
        @ empiler x-1
        sub sp, sp, #4
        str r1, [sp]
        @ place pour resultat
        sub sp, sp, #4
        bl fact
        @ recuperer le resultat de l'appel r1=fact(x-1)
        ldr r1, [sp]
        @ liberation parametre et resultat
        add sp, sp, #8
        @ r0=x

        @ r0 * r1 = x * fact(x-1)
        mul r3, r0, r1
        @ ranger le resultat

fsi:
        @ recuperer fp
        ldr fp, [sp]
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr
```

Application : la fonction `fact`

```
fact:
        @ empiler lr
        sub sp, sp, #4
        str lr, [sp]
        @ sauvegarde ancien fp et place fp(fact)
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        @ recuperer la valeur du parametre x : r0=x
        ldr r0, [fp, #+12]
        cmp r0, #0
        bne sinon
alors:  @ ranger le resultat (1)
        mov r1, #1
        str r1, [fp, #+8]
        b fsi
sinon:  @ r0 contient x
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1
        sub r1, r0, #1
        @ empiler x-1
        sub sp, sp, #4
        str r1, [sp]
        @ place pour resultat
        sub sp, sp, #4
        bl fact
        @ recuperer le resultat de l'appel r1=fact(x-1)
        ldr r1, [sp]
        @ liberation parametre et resultat
        add sp, sp, #8
        @ r0=x
        ldr r0, [fp, #+12]
        @ r0 * r1 = x * fact(x-1)
        mul r3, r0, r1
        @ ranger le resultat
fsi:
        @ recuperer fp
        ldr fp, [sp]
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr
```

Application : la fonction `fact`

```
fact:
        @ empiler lr
        sub sp, sp, #4
        str lr, [sp]
        @ sauvegarde ancien fp et place fp(fact)
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        @ recuperer la valeur du parametre x : r0=x
        ldr r0, [fp, #+12]
        cmp r0, #0
        bne sinon
alors:  @ ranger le resultat (1)
        mov r1, #1
        str r1, [fp, #+8]
        b fsi
sinon:  @ r0 contient x
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1
        sub r1, r0, #1
        @ empiler x-1
        sub sp, sp, #4
        str r1, [sp]
        @ place pour resultat
        sub sp, sp, #4
        bl fact
        @ recuperer le resultat de l'appel r1=fact(x-1)
        ldr r1, [sp]
        @ liberation parametre et resultat
        add sp, sp, #8
        @ r0=x
        ldr r0, [fp, #+12]
        @ r0 * r1 = x * fact(x-1)
        mul r3, r0, r1
        @ ranger le resultat
        str r3, [fp, #+8]
fsi:
        @ recuperer fp
        ldr fp, [sp]
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr
```

Exercice

Dérouler l'exécution avec $n = 3$ en dessinant les différents états de la pile, c'est-à-dire la zone de mémoire repérée par `sp`.

Plan

1 Rappels

2 Gestion du résultat

3 Variables locales et temporaires

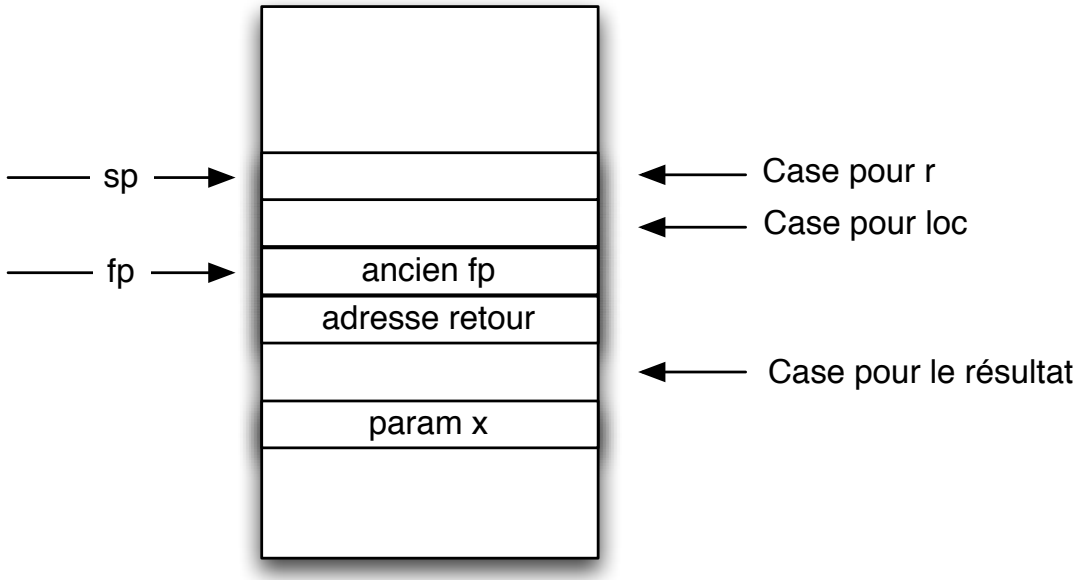
4 Passage par *adresse*

5 Conclusions

Application : codage d'une fonction factorielle avec des variables locales

```
int fact (int x) {
int loc, r;
   if x==0 { r = 1; }
   else {
      loc = fact (x-1); r = x * loc; }
   return r;
}

main () {
int n, y;
   ...
   y = fact(n);
   ...
}
```

Etat de la pile lors de l'exécution du corps de factorielle juste après l'appel dans main



Nouvelle version de la fonction `fact`

```
fact:   @ empiler adr retour
        @ mise en place fp
        @ place pour loc et r

        @ if x==0 ...
                                @ r0=x
alors:
                                @ r = 1
sinon:  @ appel fact(x-1)
        @ preparer param et resultat

                                @ r1=x-1

finsi:
        @ apres l'appel

        @ recuperer place var loc

        @ retour

        @ case resultat
        @ appel
        @ recuperer resultat
        @ desallouer param et res

        @ loc=fact(x-1)
        @ r0=x
        @ r1=loc
        @ x*loc
        @ r=x*loc

        @ return r

        @ recuperer fp
```

Nouvelle version de la fonction `fact`

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp
        @ place pour loc et r

        @ if x==0 ...
                        @ r0=x
alors:
                        @ r = 1
sinon:  @ appel fact(x-1)
        @ preparer param et resultat

                        @ r1=x-1

        @ apres l'appel

        @ recuperer place var loc

        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr

finsi:  @ case resultat
        @ appel
        @ recuperer resultat
        @ desallouer param et res

        @ loc=fact(x-1)
        @ r0=x
        @ r1=loc
        @ x*loc
        @ r=x*loc

        @ return r

        @ recuperer fp

        @ ou bx lr
```

Nouvelle version de la fonction `fact`

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp
        @ place pour loc et r
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp

        @ if x==0 ...
                                @ r0=x
alors:
                                @ r = 1
sinon:  @ appel fact(x-1)
        @ preparer param et resultat

                                @ r1=x-1
                                @ case resultat
                                @ appel
                                @ recuperer resultat
                                @ desallouer param et res
        @ apres l'appel

                                @ loc=fact(x-1)
                                @ r0=x
                                @ r1=loc
                                @ x*loc
                                @ r=x*loc
finsi:
                                @ return r
        @ recuperer place var loc

        ldr fp, [sp]            @ recuperer fp
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr              @ ou bx lr
```

Nouvelle version de la fonction `fact`

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp
        @ place pour loc et r
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        sub sp, sp, #8
        @ if x==0 ...
                                @ r0=x
alors:
                                @ r = 1
sinon:  @ appel fact(x-1)
        @ preparer param et resultat
                                @ r1=x-1
                                                finsi:
                                                        @ apres l'appel
                                                        @ recuperer place var loc
                                                        add sp, sp, #8
                                                        ldr fp, [sp]
                                                        add sp, sp, #4
                                                        @ retour
                                                        ldr lr, [sp]
                                                        add sp, sp, #4
                                                        mov pc, lr
                                                                @ case resultat
                                                                @ appel
                                                                @ recuperer resultat
                                                                @ desallouer param et res

                                                                @ loc=fact(x-1)
                                                                @ r0=x
                                                                @ r1=loc
                                                                @ x*loc
                                                                @ r=x*loc

                                                                @ return r

                                                                @ recuperer fp

                                                                @ ou bx lr
```

Nouvelle version de la fonction `fact`

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp
        @ place pour loc et r
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        sub sp, sp, #8
        @ if x==0 ...
        ldr r0, [fp, #+12]      @ r0=x
        cmp r0, #0
        bne sinon
alors:  mov r2, #1
        str r2, [fp, #-8]       @ r = 1
        b finsi
sinon:  @ appel fact(x-1)
        @ preparer param et resultat

                                @ r1=x-1
        @ case resultat
        @ appel
        @ recuperer resultat
        @ desallouer param et res
        @ apres l'appel

        @ loc=fact(x-1)
        @ r0=x
        @ r1=loc
        @ x*loc
        @ r=x*loc
finsi:

        @ return r
        @ recuperer place var loc
        add sp, sp, #8
        ldr fp, [sp]            @ recuperer fp
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr              @ ou bx lr
```

Nouvelle version de la fonction `fact`

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp
        @ place pour loc et r
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        sub sp, sp, #8
        @ if x==0 ...
        ldr r0, [fp, #+12]      @ r0=x
        cmp r0, #0
        bne sinon
alors:  mov r2, #1
        str r2, [fp, #-8]       @ r = 1
        b finsi
sinon:  @ appel fact(x-1)
        @ preparer param et resultat
        sub sp, sp #4
        sub r1, r0, #1          @ r1=x-1
        str r1, [sp]
        sub sp, sp, #4          @ case resultat
        bl fact                 @ appel
        ldr r1, [sp]            @ recuperer resultat
        add sp, sp, #8          @ desallouer param et res
        @ apres l'appel

                                @ loc=fact(x-1)
                                @ r0=x
                                @ r1=loc
                                @ x*loc
                                @ r=x*loc
finsi:

                                @ return r
        @ recuperer place var loc
        add sp, sp, #8
        ldr fp, [sp]            @ recuperer fp
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr              @ ou bx lr
```

Nouvelle version de la fonction `fact`

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp
        @ place pour loc et r
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        sub sp, sp, #8
        @ if x==0 ...
        ldr r0, [fp, #+12]      @ r0=x
        cmp r0, #0
        bne sinon
alors:  mov r2, #1
        str r2, [fp, #-8]       @ r = 1
        b finsi
sinon:  @ appel fact(x-1)
        @ preparer param et resultat
        sub sp, sp #4
        sub r1, r0, #1          @ r1=x-1
        str r1, [sp]
        sub sp, sp, #4          @ case resultat
        bl fact                 @ appel
        ldr r1, [sp]            @ recuperer resultat
        add sp, sp, #8          @ desallouer param et res
        @ apres l'appel
        str r1, [fp, #-4]       @ loc=fact(x-1)
        ldr r0, [fp, #+12]      @ r0=x
        ldr r1, [fp, #-4]       @ r1=loc
        mul r2, r0, r1          @ x*loc
        str r2, [fp, #-8]       @ r=x*loc
finsi:
                                @ return r
        @ recuperer place var loc
        add sp, sp, #8
        ldr fp, [sp]            @ recuperer fp
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr              @ ou bx lr
```

Nouvelle version de la fonction `fact`

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp
        @ place pour loc et r
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        sub sp, sp, #8
        @ if x==0 ...
        ldr r0, [fp, #+12]      @ r0=x
        cmp r0, #0
        bne sinon
alors:  mov r2, #1
        str r2, [fp, #-8]       @ r = 1
        b finsi
sinon:  @ appel fact(x-1)
        @ preparer param et resultat
        sub sp, sp #4
        sub r1, r0, #1          @ r1=x-1
        str r1, [sp]
        sub sp, sp, #4          @ case resultat
        bl fact                 @ appel
        ldr r1, [sp]            @ recuperer resultat
        add sp, sp, #8          @ desallouer param et res
        @ apres l'appel
        str r1, [fp, #-4]       @ loc=fact(x-1)
        ldr r0, [fp, #+12]      @ r0=x
        ldr r1, [fp, #-4]       @ r1=loc
        mul r2, r0, r1          @ x*loc
        str r2, [fp, #-8]       @ r=x*loc
finsi:
        ldr r2, [fp, #-8]
        str r2, [fp, #+8]       @ return r
        @ recuperer place var loc
        add sp, sp, #8
        ldr fp, [sp]            @ recuperer fp
        add sp, sp, #4
        @ retour
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr              @ ou bx lr
```

Variables temporaires

Problème :

- Les registres utilisés par une procédure ou une fonction pour des calculs intermédiaires locaux sont modifiés
- Or il serait sain de les retrouver inchangés après un appel de procédure ou fonction

Solution :

- Sauvegarder les registres utilisés : `r0, r1, r2`... dans la pile.
- Et cela doit être fait avant de les modifier donc en tout début du code de la procédure ou fonction.

Application à l'exemple de la fonction `fact`

Le code de la fonction `fact` utilise les registres `r0, r1, r2`.

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp et allocation loc et r
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        sub sp, sp, #8
        @ sauvegarde de r0, r1, et r2 (empiler)

        @ if x==0 ...
        ...
```

```
        @ restaurer les registres r0, r1, r2 (depiler)

        @ desallouer var locales
        add sp, sp, #8
        ldr fp, [sp] @ ancien fp
        add sp, sp, #4
        @ depiler adr retour dans lr
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr retour
```

Application à l'exemple de la fonction `fact`

Le code de la fonction `fact` utilise les registres `r0, r1, r2`.

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp et allocation loc et r
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        sub sp, sp, #8
        @ sauvegarde de r0, r1, et r2 (empiler)
        sub sp, sp, #4
        str r0, [sp]
        sub sp, sp, #4
        str r1, [sp]
        sub sp, sp, #4
        str r2, [sp]
        @ if x==0 ...
        ...
```

```
        @ restaurer les registres r0, r1, r2 (depiler)
        @ desallouer var locales
        add sp, sp, #8
        ldr fp, [sp] @ ancien fp
        add sp, sp, #4
        @ depiler adr retour dans lr
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr retour
```

Application à l'exemple de la fonction `fact`

Le code de la fonction `fact` utilise les registres `r0, r1, r2`.

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp et allocation loc et r
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        sub sp, sp, #8
        @ sauvegarde de r0, r1, et r2 (empiler)
        sub sp, sp, #4
        str r0, [sp]
        sub sp, sp, #4
        str r1, [sp]
        sub sp, sp, #4
        str r2, [sp]
        @ if x==0 ...
        ...
```

```
        @ restaurer les registres r0, r1, r2 (depiler)
        ldr r2, [sp]
        add sp, sp, #4
        ldr r1, [sp]
        add sp, sp, #4
        ldr r0, [sp]
        add sp, sp, #4
        @ desallouer var locales
        add sp, sp, #8
        ldr fp, [sp] @ ancien fp
        add sp, sp, #4
        @ depiler adr retour dans lr
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr retour
```

Structure générale du code d'un appel et du corps de la fonction ou procédure

appelant `P` :
1) préparer et empiler les paramètres
2) si fonction, réserver une place dans la pile pour le résultat
3) appeler Q : `BL Q`
4) si fonction, récupérer le résultat
5) si fonction, libérer la place allouée au résultat
6) libérer la place allouée aux paramètres

appelée `Q` :
1) empiler l'adresse de retour (`lr`)
2) empiler la valeur `fp` de l'appelant
3) placer `fp` pour repérer les variables de l'appelée
4) allouer la place pour les variables locales
5) empiler les variables temporaires (registres) utilisées
6) **corps de la fonction**
7) si fonction, le résultat est rangé en **fp+8**
8) dépiler les variables temporaires (registres) utilisées
9) libérer la place allouée aux variables locales
10) dépiler `fp`
11) dépiler l'adresse de retour (`lr`)
12) retour à l'appelant : `MOV pc,lr` ou `BX` lr

Plan

1 Rappels

2 Gestion du résultat

3 Variables locales et temporaires

4 Passage par *adresse*

5 Conclusions

Situation : comment faire +1 par programme ?

Situation : comment faire +1 par programme ?

- Directe :

```
n : entier
    n = n+1;
```

Situation : comment faire +1 par programme ?

- Directe :

```
n : entier
    n = n+1;
```

- Par procédure :

```
procedure inc (x : entier)
    x = x+1;

n : entier
  inc(n);
```

Situation : comment faire +1 par programme ?

- Directe :

```
n : entier
    n = n+1;
```

- Par procédure :

```
procedure inc (x : entier)
    x = x+1;

n : entier
  inc(n);
```

- Catastrophe, cela ne marche pas

Situation : comment faire +1 par programme ?

- Directe :

```
n : entier
    n = n+1;
```

- Par procédure :

```
procedure inc (x : entier)
    x = x+1;

n : entier
  inc(n);
```

- Catastrophe, cela ne marche pas
- Le +1 s'effectue pour l'élément situé sur la pile, pas sur l'original !

Situation : comment faire +1 par programme ?

- Directe :

```
n : entier
    n = n+1;
```

- Par procédure :

```
procedure inc (x : entier)
    x = x+1;

n : entier
  inc(n);
```

- Catastrophe, cela ne marche pas
- Le +1 s'effectue pour l'élément situé sur la pile, pas sur l'original !
- C'est le drame du passage de paramètre par valeur

Situation : comment faire +1 par programme ?

- Directe :

```
n : entier
   n = n+1;
```

- Par procédure :

```
procedure inc (x : entier)
   x = x+1;

n : entier
  inc(n);
```

- Catastrophe, cela ne marche pas
- Le +1 s'effectue pour l'élément situé sur la pile, pas sur l'original !
- C'est le drame du passage de paramètre par valeur
- **Solution :** passage de paramètre par référence, ou par adresse (paramètre donnée *vs.* paramètre résultat)

Remarque : des fois, ça marche !

Comment faire +1 sur le premier élément d'un tableau

Remarque : des fois, ça marche !

Comment faire +1 sur le premier élément d'un tableau

- Par procédure :

```
procedure inc (t : tableau d'entiers)
   t[0] = t[0]+1;

Ns : tableau d'entiers
  inc(Ns);
```

Remarque : des fois, ça marche !

Comment faire +1 sur le premier élément d'un tableau

- Par procédure :

```
procedure inc (t : tableau d'entiers)
   t[0] = t[0]+1;

Ns : tableau d'entiers
  inc(Ns);
```

- Cette fois cela marche :-)

Remarque : des fois, ça marche !

Comment faire +1 sur le premier élément d'un tableau

- Par procédure :

```
procedure inc (t : tableau d'entiers)
   t[0] = t[0]+1;

Ns : tableau d'entiers
  inc(Ns);
```

- Cette fois cela marche :-)
- `Ns` et `t` sont des références ...

Remarque : des fois, ça marche !

Comment faire +1 sur le premier élément d'un tableau

- Par procédure :

```
procedure inc (t : tableau d'entiers)
   t[0] = t[0]+1;

Ns : tableau d'entiers
  inc(Ns);
```

- Cette fois cela marche :-)
- `Ns` et `t` sont des références ...
- C'est la suite du drame du passage de paramètre par valeur

Référence d'une variable ?

Comment obtenir la référence d'une variable ?

Référence d'une variable ?

Comment obtenir la référence d'une variable ?

- cela dépend des langages ...

Référence d'une variable ?

Comment obtenir la référence d'une variable ?

- cela dépend des langages ...
- parfois c'est impossible, déconseillé, difficile, ...

Référence d'une variable ?

Comment obtenir la référence d'une variable ?

- cela dépend des langages ...
- parfois c'est impossible, déconseillé, difficile, ...
- en C, c'est son adresse : l'adresse de `x` se note `&x` ;

Référence d'une variable ?

Comment obtenir la référence d'une variable ?

- cela dépend des langages ...
- parfois c'est impossible, déconseillé, difficile, ...
- en C, c'est son adresse : l'adresse de `x` se note `&x` ;
  $\rightarrow$ accès à une variable à partir de son adresse stockée dans `y` : `*y` ;

Référence d'une variable ?

Comment obtenir la référence d'une variable ?

- cela dépend des langages ...
- parfois c'est impossible, déconseillé, difficile, ...
- en C, c'est son adresse : l'adresse de `x` se note `&x` ;
  $\rightarrow$ accès à une variable à partir de son adresse stockée dans `y` : `*y` ;
- en javascript (pour les variables globales), son nom peut servir de référence : `"n"` ;

Référence d'une variable ?

Comment obtenir la référence d'une variable ?

- cela dépend des langages ...
- parfois c'est impossible, déconseillé, difficile, ...
- en C, c'est son adresse : l'adresse de `x` se note `&x` ;
  → accès à une variable à partir de son adresse stockée dans `y` : `*y` ;
- en javascript (pour les variables globales), son nom peut servir de référence : `"n"` ;
  → accès à la valeur à partir de son nom : `window["n"]` ;

Autre solution

Si on ne peut pas accéder à une référence ...

Autre solution

Si on ne peut pas accéder à une référence ...

- Par fonction (et confier l'affectation à l'appelant) :

```
fonction inc (x : entier)
    retourne x+1;

n : entier
  n=inc(n);
```

Autre solution

Si on ne peut pas accéder à une référence ...

- Par fonction (et confier l'affectation à l'appelant) :

```
fonction inc (x : entier)
   retourne x+1;

n : entier
  n=inc(n);
```

- Par macro (si disponible)

Réalisation, vocabulaire

On se place maintenant dans le cas d'une procédure ayant **des paramètres de type donnée** et **des paramètres de type résultat**.

```
procedure XX (donnees x, y : entier; resultat z : entier)
u,v : entier
   ...
   u=x;
   v=y+2;
   ...
   z=u+v;
   ...
```

Réalisation, vocabulaire

On se place maintenant dans le cas d'une procédure ayant **des paramètres de type donnée** et **des paramètres de type résultat**.

```
procedure XX (donnees x, y : entier; resultat z : entier)
u,v : entier
   ...
   u=x;
   v=y+2;
   ...
   z=u+v;
   ...
```

- Les paramètres données **ne doivent pas être modifiés par l'exécution de la procédure** : les paramètres effectifs associés à *x* et *y* sont des expressions qui sont évaluées avant l'appel, les valeurs étant substituées aux paramètres formels lors de l'exécution du corps de la procédure.

Réalisation, vocabulaire

On se place maintenant dans le cas d'une procédure ayant **des paramètres de type donnée** et **des paramètres de type résultat**.

```
procedure XX (donnees x, y : entier; resultat z : entier)
u,v : entier
   ...
   u=x;
   v=y+2;
   ...
   z=u+v;
   ...
```

- Les paramètres données **ne doivent pas être modifiés par l'exécution de la procédure** : les paramètres effectifs associés à *x* et *y* sont des expressions qui sont évaluées avant l'appel, les valeurs étant substituées aux paramètres formels lors de l'exécution du corps de la procédure.
- Le paramètre effectif associé au paramètre formel résultat est une variable **dont la valeur n'est significative qu'après l'appel de la procédure** ; cette valeur est calculée dans le corps de la procédure et affectée à la variable passée en argument.

Notations

Il existe différentes façons de gérer le paramètre `z`. Nous n'en étudions qu'une seule : la méthode dite du passage par adresse.

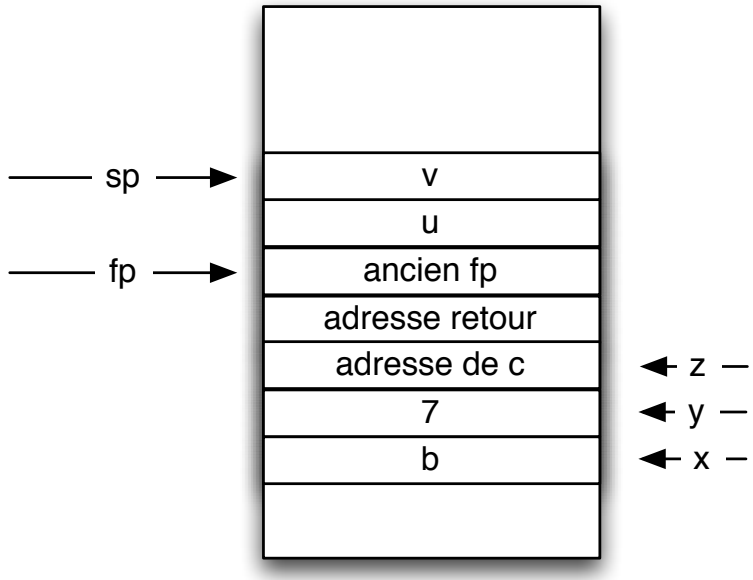
Nous utilisons la notation suivante :

```
procedure XX (donnees x, y : entier; adresse z : entier)
u,v : entier

   ...
   u=x;
   v=y+2;
   ...
   mem[z]=u+v;  @ mem[z] designe le contenu de la memoire d'adresse z
   ...
```

L'exemple d'appel traité

```
a,b,c : entier

    b=3;
    ....
    XX (b, 7, adresse de c);
```

Solution : état de la pile lors de l'exécution de la procédure XX



main

```
        .bss
a:      .skip 4
b:      .skip 4
c:      .skip 4
        .text
main:
        ...
```

@ *r*0 ← valeur de *b*

@ empiler b

@ *r*0 ← 7

@ empiler 7

@ *r*0 ← adresse de *c*

@ empiler adresse de c

```
        bl XX
```

...

```
        ...
        b exit
ptr_a:  .word a
ptr_b:  .word b
ptr_c:  .word c
```

main

```
        .bss
a:      .skip 4
b:      .skip 4
c:      .skip 4
        .text
main:
        ...
        ldr r0, ptr_b
        ldr r0, [r0]        @ r0 ← valeur de b

                            @ empiler b

        mov r0, #7          @ r0 ← 7

                            @ empiler 7

        ldr r0, ptr_c       @ r0 ← adresse de c

                            @ empiler adresse de c
        bl XX               ...

                            ...
        b exit
ptr_a:  .word a
ptr_b:  .word b
ptr_c:  .word c
```

main

```
        .bss
a:      .skip 4
b:      .skip 4
c:      .skip 4
        .text
main:
        ...
        ldr r0, ptr_b
        ldr r0, [r0]        @ r0 ← valeur de b
        sub sp, sp, #4
        str r0, [sp]        @ empiler b

        mov r0, #7          @ r0 ← 7
        sub sp, sp, #4
        str r0, [sp]        @ empiler 7

        ldr r0, ptr_c       @ r0 ← adresse de c
        sub sp, sp, #4
        str r0, [sp]        @ empiler adresse de c
        bl XX               ...
```

```
        ...
        b exit
ptr_a:  .word a
ptr_b:  .word b
ptr_c:  .word c
```

Procédure XX

```
XX:
        ...
                                @ u ← x

                                @ v ← y + 2

        ...

                                @ calcul de u + v

                                @ r2 ← z, i.e., adresse c
                                @ mem[z] ← u + v, i.e., mem[adresse c] ← u + v
        ...
```

Procédure XX

```
XX:
        ...
        ldr r0, [fp, #+16]    @ u ← x
        str r0, [fp, #-4]

                              @ v ← y + 2


        ...

                              @ calcul de u + v

                              @ r2 ← z, i.e., adresse c
                              @ mem[z] ← u + v, i.e., mem[adresse c] ← u + v
        ...
```

Procédure XX

```
XX:
        ...
        ldr r0, [fp, #+16]    @ u ← x
        str r0, [fp, #-4]

        ldr r0, [fp, #+12]    @ v ← y + 2
        add r0, r0, #2
        str r0, [fp, #-8]
        ...

                              @ calcul de u + v

                              @ r2 ← z, i.e., adresse c
                              @ mem[z] ← u + v, i.e., mem[adresse c] ← u + v
        ...
```

Procédure XX

```
XX:
        ...
        ldr r0, [fp, #+16]    @ u ← x
        str r0, [fp, #-4]

        ldr r0, [fp, #+12]    @ v ← y + 2
        add r0, r0, #2
        str r0, [fp, #-8]
        ...
        ldr r0, [fp, #-4]
        ldr r1, [fp, #-8]
        add r0, r0, r1        @ calcul de u + v

                              @ r2 ← z, i.e., adresse c
                              @ mem[z] ← u + v, i.e., mem[adresse c] ← u + v
        ...
```

Procédure XX

```
XX:
        ...
        ldr r0, [fp, #+16]    @ u ← x
        str r0, [fp, #-4]

        ldr r0, [fp, #+12]    @ v ← y + 2
        add r0, r0, #2
        str r0, [fp, #-8]
        ...
        ldr r0, [fp, #-4]
        ldr r1, [fp, #-8]
        add r0, r0, r1        @ calcul de u + v

        ldr r2, [fp, #+8]     @ r2 ← z, i.e., adresse c
        str r0, [r2]          @ mem[z] ← u + v, i.e., mem[adresse c] ← u + v
        ...
```

Exercice

Codez en ARM le programme suivant :

```
procédure swap(adresse x : entier, adresse y : entier)
  z : entier
  z=mem[x]
  si z > mem[y]
                    mem[x]=mem[y]
                    mem[y]=z
fin procédure

procédure main()
  a,b : entier
  a=9
  b=7
  afficher a
  afficher b
  swap(adresse a,adresse b)
  afficher a
  afficher b
fin procédure
```

Solution : la procédure principale

```
.text
.global main
main :  mov fp,sp                @ nouveau fp

        sub sp,sp,#8             @ variables a et b

        mov r0,#9                @ a←9
        str r0,[fp,#-4]
        mov r0,#7                @ b←7
        str r0,[fp,#-8]

        ldr r1,[fp,#-4]          @ affiche a
        bl EcrZdecimal32
        ldr r1,[fp,#-8]          @ affiche b
        bl EcrZdecimal32

        sub r0,fp,#4             @ stockage
        sub sp,sp,#4             @ de l'adresse
        str r0,[sp]              @ de a sur la pile

        sub r0,fp,#8             @ idem pour b
        sub sp,sp,#4
        str r0,[sp]

        bl swap                  @ appel
        add sp,sp,#8             @ dépilement paramètres

        ldr r1,[fp,#-4]          @ affiche a et b
        bl EcrZdecimal32
        ldr r1,[fp,#-8]
        bl EcrZdecimal32

        add sp,sp,#8             @ dépile a et b
        b exit
```

Solution : la procédure `swap`

```
swap :  sub sp,sp,#4      @ sauvegarde fp
        str fp,[sp]

        mov fp,sp         @ nouveau fp

        sub sp,sp,#4      @ variable z

        sub sp,sp,#4      @ var. temporaires
        str r0,[sp]
        sub sp,sp,#4
        str r1,[sp]
        sub sp,sp,#4
        str r2,[sp]
        sub sp,sp,#4
        str r3,[sp]

        ldr r0,[fp,#8]    @ adresse x
        ldr r1,[fp,#4]    @ adresse y

        ldr r2,[r0]       @ z←mem[x]
        str r2,[fp,#-4]

        ldr r3,[r1]       @ r3 := mem[y]
        cmp r2,r3         @ comparaison z,mem[y]
        ble fin

                          @ z>y
        str r3,[r0]       @ échange
        str r2,[r1]

fin :   ldr r3,[sp]       @ restauration
        add sp,sp,#4      @ registres
        ldr r2,[sp]
        add sp,sp,#4
        ldr r1,[sp]
        add sp,sp,#4
        ldr r0,[sp]
        add sp,sp,#4

        add sp,sp,#4      @ dépile z

        ldr fp,[sp]       @ restaure ancien fp
        add sp,sp,#4
        mov pc,lr         @ ou bx lr
```

remarque : Swap en javascript

remarque : Swap en javascript

- Pour objets globaux :

remarque : Swap en javascript

- Pour objets globaux :
- function swap(a,b) { // a, b : noms (references)
  var tmp = window[a] ;
  window[a] = window[b] ;
  window[b] = tmp ; }
  var x=0 ;
  var y=1 ;
  swap(”x”,”y”)

remarque : Swap en javascript

- Pour objets globaux :
- function swap(a,b) { // a, b : noms (references)
  var tmp = window[a] ;
  window[a] = window[b] ;
  window[b] = tmp ; }
  var x=0 ;
  var y=1 ;
  swap(”x”,”y”)
- Sans référence :

remarque : Swap en javascript

- Pour objets globaux :
- function swap(a,b) { // a, b : noms (references)
  var tmp = window[a] ;
  window[a] = window[b] ;
  window[b] = tmp ; }
  var x=0 ;
  var y=1 ;
  swap(”x”,”y”)
- Sans référence :
- function swap(a,b) {
  return [b, a] ; }
  var x=0 ;
  var y=1 ;
  ([x,y]) = swap([x,y]) ;

Autre utilisation du passage par adresse (en C).

Proposition pour le calcul de la longueur d'un texte :

```
#include <stdio.h>

typedef struct {  char txt[100];  int lng;} t_texte;

int longueur(t_texte p, int i) {
  if ((i==100) || (p.txt[i]==0) || (p.txt[i]==10) || (p.txt[i]==13)) {
    return 0;}
  else {
    return 1+longueur(p,i+1);}}

int main() {
t_texte unTexte;
  printf("Entree ?\n");
  scanf("%s",unTexte.txt);
  unTexte.lng = longueur(unTexte,0);
  printf("\nSortie :%d\n",unTexte.lng);
  return 0;}
```

Autre utilisation du passage par adresse (en ARM).

```
longueur: sub sp, sp, #16
stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #96
ldr r3, [fp, #108]
cmp r3, #100
beq .L2
[...]
.L2: mov r3, #0
b .L4
.L3: ldr r3, [fp, #108]
add r3, r3, #1
str r3, [sp, #88]
mov r1, sp
add r2, fp, #20
mov r3, #88
mov r0, r1
mov r1, r2
mov r2, r3
bl memcpy
add r3, fp, #4
ldmia r3, {r0, r1, r2, r3}
bl longueur
mov r3, r0
add r3, r3, #1
.L4: mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
add sp, sp, #16
```

```
main: stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #200
ldr r0, .L6
bl puts
sub r3, fp, #108
ldr r0, .L6+4
mov r1, r3
bl scanf
mov r3, #0
str r3, [sp, #88]
mov r1, sp
sub r2, fp, #92
mov r3, #88
mov r0, r1
mov r1, r2
mov r2, r3
bl memcpy
sub r3, fp, #108
ldmia r3, {r0, r1, r2, r3}
bl longueur
mov r3, r0
str r3, [fp, #-8]
ldr r3, [fp, #-8]
ldr r0, .L6+8
mov r1, r3
bl printf
mov r3, #0
mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
```

Autre utilisation du passage par adresse (en x86).

```
longueur:
.LFB0: pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl %edi, -4(%rbp)
cmpl $100, -4(%rbp)
je .L2
[...]
cmpb $13, %al
jne .L3
.L2: movl $0, %eax
jmp .L4
.L3: movl -4(%rbp), %eax
addl $1, %eax
subq $8, %rsp
pushq 112(%rbp)
pushq 104(%rbp)
pushq 96(%rbp)
[...]
pushq 24(%rbp)
pushq 16(%rbp)
movl %eax, %edi
call longueur
addq $112, %rsp
addl $1, %eax
.L4: leave
ret
```

Autre utilisation du passage par adresse (en x86).

```
longueur:
.LFB0: pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl %edi, -4(%rbp)
cmpl $100, -4(%rbp)
je .L2
[...]
cmpb $13, %al
jne .L3
.L2: movl $0, %eax
jmp .L4
.L3: movl -4(%rbp), %eax
addl $1, %eax
subq $8, %rsp
pushq 112(%rbp)
pushq 104(%rbp)
pushq 96(%rbp)
[...]
pushq 24(%rbp)
pushq 16(%rbp)
movl %eax, %edi
call longueur
addq $112, %rsp
addl $1, %eax
.L4: leave
ret
```

```
main:
.LFB1: pushq %rbp
movq %rsp, %rbp
subq $112, %rsp
movl $.LC0, %edi
call puts
leaq -112(%rbp), %rax
movq %rax, %rsi
movl $.LC1, %edi
movl $0, %eax
call __isoc99_scanf
subq $8, %rsp
pushq -16(%rbp)
pushq -24(%rbp)
pushq -32(%rbp)
[...]
pushq -104(%rbp)
pushq -112(%rbp)
movl $0, %edi
call longueur
addq $112, %rsp
movl %eax, -12(%rbp)
movl -12(%rbp), %eax
movl %eax, %esi
movl $.LC2, %edi
movl $0, %eax
call printf
movl $0, %eax
leave
ret
```

Autre utilisation du passage par adresse (en C).

Proposition 2 pour le calcul de la longueur d'un texte (avec passage par adresse) :

```
#include <stdio.h>

typedef struct {  char txt[100];  int lng;} t_texte;

int longueur(t_texte *p, int i) {
  if ((i==100) || (p->txt[i]==0) || (p->txt[i]==10) || (p->txt[i]==13)) {
    return 0;}
  else {
    return 1+longueur(p,i+1);}}

int main() {
t_texte unTexte;
  printf("Entree ?\n");
  scanf("%s",unTexte.txt);
  unTexte.lng = longueur(&unTexte,0);
  printf("\nSortie : %d\n",unTexte.lng);
  return 0;}
```

Autre utilisation du passage par adresse (en ARM).

```
longueur:
stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #8
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r3, [fp, #-12]
cmp r3, #100
beq .L2
[...]
.L2: mov r3, #0
b .L4
.L3: ldr r3, [fp, #-12]
add r3, r3, #1
ldr r0, [fp, #-8]
mov r1, r3
bl longueur
mov r3, r0
add r3, r3, #1
.L4: mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
bx lr
```

Autre utilisation du passage par adresse (en ARM).

```
longueur:
stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #8
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r3, [fp, #-12]
cmp r3, #100
beq .L2
[...]
.L2: mov r3, #0
b .L4
.L3: ldr r3, [fp, #-12]
add r3, r3, #1
ldr r0, [fp, #-8]
mov r1, r3
bl longueur
mov r3, r0
add r3, r3, #1
.L4: mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
bx lr
```

```
main: stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #104
ldr r0, .L6
bl puts
sub r3, fp, #108
ldr r0, .L6+4
mov r1, r3
bl scanf
sub r3, fp, #108
mov r0, r3
mov r1, #0
bl longueur
mov r3, r0
str r3, [fp, #-8]
ldr r3, [fp, #-8]
ldr r0, .L6+8
mov r1, r3
bl printf
mov r3, #0
mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
bx lr
```

Autre utilisation du passage par adresse (en x86).

```
longueur: .LFB0:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movq %rdi, -8(%rbp)
movl %esi, -12(%rbp)
cmpl $100, -12(%rbp)
je .L2
[...]
cmpb $13, %al
jne .L3
.L2: movl $0, %eax
jmp .L4
.L3: movl -12(%rbp), %eax
leal 1(%rax), %edx
movq -8(%rbp), %rax
movl %edx, %esi
movq %rax, %rdi
call longueur
addl $1, %eax
.L4:
leave
ret
```

Autre utilisation du passage par adresse (en x86).

```
longueur: .LFB0:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movq %rdi, -8(%rbp)
movl %esi, -12(%rbp)
cmpl $100, -12(%rbp)
je .L2
[...]
cmpb $13, %al
jne .L3
.L2: movl $0, %eax
jmp .L4
.L3: movl -12(%rbp), %eax
leal 1(%rax), %edx
movq -8(%rbp), %rax
movl %edx, %esi
movq %rax, %rdi
call longueur
addl $1, %eax
.L4:
leave
ret
```

```
main:
.LFB1: pushq %rbp
movq %rsp, %rbp
subq $112, %rsp
movl $.LC0, %edi
call puts
leaq -112(%rbp), %rax
movq %rax, %rsi
movl $.LC1, %edi
movl $0, %eax
call __isoc99_scanf
leaq -112(%rbp), %rax
movl $0, %esi
movq %rax, %rdi
call longueur
movl %eax, -12(%rbp)
movl -12(%rbp), %eax
movl %eax, %esi
movl $.LC2, %edi
movl $0, %eax
call printf
movl $0, %eax
leave
ret
```

Plan

1 Rappels

2 Gestion du résultat

3 Variables locales et temporaires

4 Passage par *adresse*

5 Conclusions

Question : hors la pile ?

Si le programme est l'exécution d'une fonction principale (main) et que les paramètres (coté appelant, coté appelé), variables (locales, temporaires), résultat, sont dans la pile, que reste-t-il hors de la pile ?

Question : hors la pile ?

Si le programme est l'exécution d'une fonction principale (main) et que les paramètres (coté appelant, coté appelé), variables (locales, temporaires), résultat, sont dans la pile, que reste-t-il hors de la pile ?

Réponses :

- pas grand chose

Question : hors la pile ?

Si le programme est l'exécution d'une fonction principale (main) et que les paramètres (coté appelant, coté appelé), variables (locales, temporaires), résultat, sont dans la pile, que reste-t-il hors de la pile ?

Réponses :

- pas grand chose
- les variables globales

Question : hors la pile ?

Si le programme est l'exécution d'une fonction principale (main) et que les paramètres (coté appelant, coté appelé), variables (locales, temporaires), résultat, sont dans la pile, que reste-t-il hors de la pile ?

Réponses :

- pas grand chose
- les variables globales
- le tas !

Question : hors la pile ?

Si le programme est l'exécution d'une fonction principale (main) et que les paramètres (coté appelant, coté appelé), variables (locales, temporaires), résultat, sont dans la pile, que reste-t-il hors de la pile ?

Réponses :

- pas grand chose
- les variables globales
- le tas !
- (et des constantes [par exemple des textes])

Conclusion / Rappel : Structure générale du code d'un appel et du corps de la fonction ou procédure

appelant `P` :
1) préparer et empiler les paramètres (valeurs et/ou adresses)
2) si fonction, réserver une place dans la pile pour le résultat
3) appeler Q : `BL Q`
4) si fonction, récupérer le résultat
5) si fonction, libérer la place allouée au résultat
6) libérer la place allouée aux paramètres

appelée `Q` :
1) empiler l'adresse de retour (`lr`)
2) empiler la valeur `fp` de l'appelant
3) placer `fp` pour repérer les variables de l'appelée
4) allouer la place pour les variables locales
5) empiler les variables temporaires (registres) utilisées
6) **corps de la fonction**
7) si fonction, le résultat est rangé en **fp+8**
8) dépiler les variables temporaires (registres) utilisées
9) libérer la place allouée aux variables locales
10) dépiler `fp`
11) dépiler l'adresse de retour (`lr`)
12) retour à l'appelant : `MOV pc,lr` ou `BX lr`