# Simulation du `for`

L'équivalent du pseudo-code

```
Pour a = 1 à b
    BLOC
FinPour
```

est

```
mov eax, 1
for:
    cmp eax, ebx
    jg end_for
    BLOC
    add eax, 1
    jmp for
end_for:
```

On peut simuler ce pseudo-code de manière plus compacte grâce à l'instruction

```
loop ETIQ
```

Celle-ci décrémente `ecx` et saute vers l'étiquette d'instruction `ETIQ` si `ecx` est non nul.

On obtient la suite d'instructions suivante :

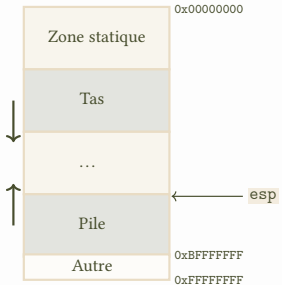
```
mov ecx, ebx
boucle:
    BLOC
    loop boucle
```

# Pile

La pile est une zone de la mémoire dans laquelle des données peuvent être empilée ou dépilées.

La zone qu'elle occupe en mémoire est de taille variable mais elle se situe toujours avant l'adresse `0xBFFFFFFF`.

La pile est de type **LIFO** : les données sont dépilées de la plus récente à la plus ancienne.




On place et on lit dans la pile uniquement des **doubles mots**.

Le registre `esp` contient l'adresse de la **tête de pile**.

Le registre `ebp` est utilisé pour sauvegarder une position dans la pile (lorsque `esp` est susceptible de changer).

# Pile

On dispose de deux opérations pour manipuler la pile :

1. **empiler** une valeur ;
2. **dépiler** une valeur.

Pour **empiler** une valeur `VAL` à la pile, on utilise

```
push VAL
```

Ceci décrémente `esp` de 4 et écrit à l'adresse `esp` la valeur `VAL`.

Pour **dépiler** vers le registre `REG` la valeur située en tête de pile, on utilise

```
pop REG
```

Ceci recopie les 4 octets à partir de l'adresse `esp` vers `REG` et incrémente `esp` de 4.

**Attention** : l'ajout d'éléments dans la pile fait décroître la valeur de `esp` et la suppression d'éléments fait croître sa valeur, ce qui est peut-être contre-intuitif.

# Pile

## – Exemple –

Observons l'effet des quatre instructions

```
push 0x3
pop eax
pop ebx
push eax
```

avec la pile dans l'état

| Adresses | Pile |
|---|---|
| 1000 | 0x01010101 |
| 1004 | 0x20202020 |
| 1008 | 0xAA55AA55 |
| 1012 | 0xFFFFFFFF |
| 1016 | 0x00000000 |

`esp` = `1008`

| |
|---|
| 0x01010101 |
| 0x00000003 |
| 0xAA55AA55 |
| 0xFFFFFFFF |
| 0x00000000 |

`esp` = `1004`

| |
|---|
| 0x01010101 |
| 0x00000003 |
| 0xAA55AA55 |
| 0xFFFFFFFF |
| 0x00000000 |

`esp` = `1008`
`eax` = `0x3`

| |
|---|
| 0x01010101 |
| 0x00000003 |
| 0xAA55AA55 |
| 0xFFFFFFFF |
| 0x00000000 |

`esp` = `1012`
`ebx` = `0xAA55AA55`

| |
|---|
| 0x01010101 |
| 0x00000003 |
| 0x00000003 |
| 0xFFFFFFFF |
| 0x00000000 |

`esp` = `1008`

# Instruction `call`

On souhaite maintenant établir un mécanisme pour pouvoir **écrire des fonctions et les appeler**.

L'un des ingrédients pour cela est l'instruction

```
call ETIQ
```

Elle permet de sauter à l'étiquette d'instruction `ETIQ`.

La différence avec l'instruction `jmp ETIQ` réside dans le fait que `call ETIQ` **empile**, avant le saut, l'**adresse de l'instruction qui la suit** dans le programme.

Ainsi, les deux suites d'instructions suivantes sont équivalentes :

```
call cible
suite:
...
```

```
push suite
jmp cible
suite:
...
```

L'adresse de retour `suite` est connue même dans le cas où cette adresse n'est pas positionnée par le programmeur.

# Instruction `ret`

L'intérêt d'enregistrer l'adresse de l'instruction qui suit un `call ETIQ` repose sur le fait que l'exécution peut **revenir** à cette instruction.

Ceci est offert par l'instruction (sans opérande)

```
ret
```

Elle dépile la donnée en tête de pile et saute à l'adresse spécifiée par cette valeur.

Ainsi, les deux suites d'instructions suivantes sont équivalentes (excepté pour la valeur de `eax` qui est modifiée dans la seconde) :

```
call cible

suite:
    ...
cible:
    ...

ret
```

```
push suite
jmp cible

suite:
    ...
cible:
    ...

pop eax
jmp eax
```

# Exemple d'utilisation de `call` / `ret`

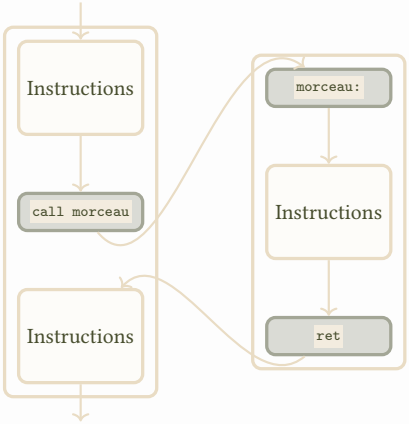
## – Exemple –

Considérons la suite d'instructions suivante et observons l'état de la pile et du pointeur d'instruction au fil de l'exécution.

```
mov ecx, 8          ; Adresse a1.
call loin           ; Adresse a2.
suiv: add ecx, 24   ; Adresse a3.
loin: add ecx, 16   ; Adresse a4.
ret                 ; Adresse a5.
...                 ; Adresse a6.
fin:                ; Adresse a7.
```

# Instructions `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret`




**Attention** : le retour à l'endroit du code attendu par l'instruction `ret` n'est correct que si l'état de la pile à l'étiquette `morceau` est le même que celui juste avant le `ret`.

# Fonctions — conventions d'appel du C

L'écriture de fonctions respecte des conventions indiquant comment sont passés les arguments et comment la valeur de retour peut être récupérée.

Celle majoritairement employée sur l'architecture x86 est la conventions d'appel du C. Celle-ci consiste en le respect des points suivants :

1. les **arguments** d'une fonction sont passés, avant son appel, dans la **pile** en les empilant ;
2. le **résultat** d'une fonction est renvoyé en l'écrivant dans le registre `eax` ;
3. la **pile** doit être dans le **même état** avant l'appel et après l'appel d'une fonction.

   Ceci signifie que l'état des pointeurs `esp` et `ebp` sont conservés et que le contenu de la pile qui suit l'adresse `esp` est également conservé ;
4. les **valeurs** des registres **différents de** `eax`, `ecx` et `edx` doivent être dans le **même état** avant l'appel et après l'appel de la fonction.

# Fonctions — écriture et appel

L'**écriture** d'une fonction suit le squelette

```
NOM_FCT:
    push ebp
    mov ebp, esp

    INSTR

    pop ebp
    ret
```

Ici, `NOM_FCT` est une étiquette d'instruction qui fait d'office de nom pour la fonction. De plus, `INSTR` est un bloc d'instructions.

Il est primordial que `INSTR` **conserve l'état de la pile**.

L'**appel** d'une fonction se fait par

```
push ARG_N
...
push ARG_1

call NOM_FCT

add esp, 4 * N
```

Ici, `NOM_FCT` est le nom de la fonction à appeler. Elle admet `N` arguments qui sont **empilés du dernier au premier**.

Après l'appel, on incrémente `esp` pour dépiler d'un coup les `N` arguments de la fonction.

# Fonctions — appel et état de la pile

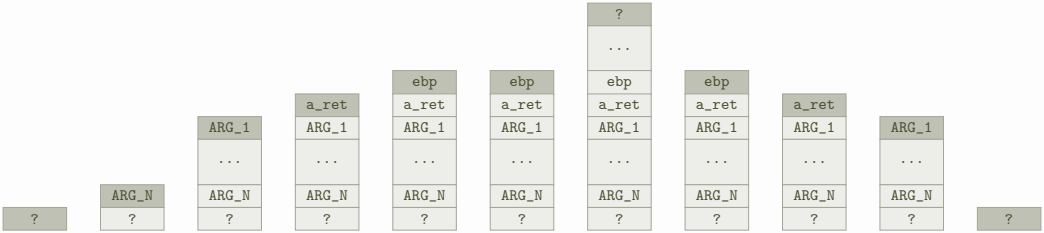
Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
; Appel de la fonction NOM_FCT.
push ARG_N
...
push ARG_1
call NOM_FCT
ADD esp, 4 * N     ; Adresse a_ret.
```

```
; Definition de la fonction NOM_FCT.
NOM_FCT:
    push ebp
    mov ebp, esp
    INSTR
    pop ebp
    ret
```

À partir du 6e dessin de pile, `ebp` pointe vers une zone fixée de la mémoire.

Il s'agit de la zone de la pile située à deux cases au dessus de là où figure le 1er argument.



# Fonctions — préservation de l'état de la pile

L'**état** de la **pile** doit être **préservé** par le bloc d'instructions `INSTR`.

Cela signifie que l'état de la pile juste avant d'exécuter `INSTR` et son état juste après son exécution sont les mêmes. En d'autres termes,

- `esp` doit posséder la même valeur ;
- toutes les données de la pile d'adresses plus grandes que `esp` ne doivent pas été modifiées.

C'est bien le cas si

- `INSTR` contient autant de `push` que de `pop` ;
- à tout instant, il y a au moins autant de `push` que de `pop` qui ont été exécutés.

Il faut bien respecter ces deux conditions dans la pratique.

# Fonctions — rôle de `ebp`

La **valeur** de `esp` est susceptible de **changer** dans `INSTR`. C'est pour cela que l'on **sauvegarde** sa valeur, à l'entrée de la fonction, dans `ebp`.

Le registre `ebp` sert ainsi, dans `INSTR`, à **accéder aux arguments**. En effet, l'adresse du 1er argument est `ebp + 8`, celle du 2e est `ebp + 12` et plus généralement, celle du `i`e argument est

```
ebp + 4 * (i + 1)
```

On **sauvegarde** et on **restaure** tout de même, par un `push ebp` et `pop ebp` l'état de `ebp` à l'entrée et à la sortie de la fonction.

Ce même mécanisme doit être utilisé pour sauvegarder/restaurer l'état des registres de travail `eax` (sauf si la fonction renvoie une valeur), `ebx`, `ecx` et `edx`.

# Fonctions — exemple 1

## – Exemple –

```
; Fonction qui renvoie la somme de deux entiers.
; Arguments :
;       (1) une valeur entière signée sur 4 octets
;       (2) une valeur entière signée sur 4 octets.
; Renvoi : la somme des deux arguments.

somme:
    push ebp
    mov ebp, esp

    mov eax, [ebp + 8]
    add eax, [ebp + 12]

    pop ebp
    ret
```

Pour calculer dans `eax` la somme de $(43)_{\text{dix}}$ et $(1996)_{\text{dix}}$, on procède par

```
push 1996
push 43
call somme
add esp, 8
```

**Rappel 1** : on empile les arguments dans l'ordre inverse de ce que la fonction attend.

**Rappel 2** : on ajoute `8` à `esp` après l'appel pour dépiler d'un seul coup les deux arguments ($8 = 2 \times 4$).

# Fonctions — exemple 2

## – Exemple –

```
; Fonction qui affiche un caractere.
; Arguments :
;       (1) valeur du caractere a afficher.
; Renvoi : rien.

print_char:
    ; Debut.
    push ebp
    mov ebp, esp

    ; Sauv. des registres.
    push eax
    push ebx
    push ecx
    push edx

    ; Affichage.
    mov ebx, 1
    mov ecx, ebp
    add ecx, 8
    mov edx, 1
    mov eax, 4
    int 0x80

    ; Rest. des registres.
    pop edx
    pop ecx
    pop ebx
    pop eax

    ; Fin.
    pop ebp
    ret
```

# Fonctions — exemple 2 bis

## – Exemple –

```
; Fonction qui affiche un caractere.
; Arguments :
;       (1) adresse du caractere a afficher.
; Renvoi : rien.

print_char_2:
    ; Debut.
    push ebp
    mov ebp, esp

    ; Sauv. des registres.
    push eax
    push ebx
    push ecx
    push edx

    ; Affichage.
    mov ebx, 1
    mov ecx, [ebp + 8]
    mov edx, 1
    mov eax, 4
    int 0x80

    ; Rest. des registres.
    pop edx
    pop ecx
    pop ebx
    pop eax

    ; Fin.
    pop ebp
    ret
```

# Fonctions — exemples 2 et 2 bis

Les deux fonctions `print_char` et `print_char_2` ne s'utilisent pas de la même manière : la 1re prend son argument **par valeur**, tandis que la 2e prend son argument **par adresse**.

## – Exemple –

Pour afficher p.ex. le caractère `'W'`, on appelle `print_char` par

```
push 'W'
call print_char
add esp, 4
```

La fonction `print_char_2` s'appelle par

```
push c1
call print_char_2
add esp, 4
```

où `c1` est l'adresse d'un caractère en mémoire. Celle-ci a été définie par exemple par `c1 : db 'W'` dans la section de données.

# Fonctions — Déclaration C

En C, les deux fonctions `print_char` et `print_char_2` se différencient par leur prototype :

`void print_char(char c)` VS `void print_char2(char *c)`

En suivant la convention d'appel, la déclaration d'une fonction suffit à savoir comment l'utiliser.

## – Exemple –

Pour appeler la fonction `nom_fun` de signature `int nom_func(int n, int *tab, char *str)`, il faut :

- Empiler l'adresse d'un `byte`
- Empiler l'adresse d'un `dword`
- Empiler la valeur d'un `dword`
- Faire un `call` sur l'étiquette de la fonction

La valeur renvoyée par la fonction se trouvera dans le registre `eax`.

Inversement, pour accéder au *i*-ième argument dans le code d'une fonction, on a la méthode `mov REG, [ebp + 4 * (i + 1)]`.

# Chaînes de caractères

Une chaîne de caractères est une **suite contiguë d'octets** en mémoire se terminant par l'octet nul.

On accède à une chaîne de caractères via l'**adresse** `x` de son **1er caractère**. Par abus de langage, on parle de « la chaîne `x` ». Les adresses des autres caractères s'obtiennent par incrémentation.

## – Exemple –

La chaîne `x` de valeur `"nasm"` est représentée en mémoire par


La valeur de `[x]` est `'n'`, celle de `[x + 1]` est `'a'`, …, celle de `[x + 4]` est `0`.

On **lit** le `i`e caractère d'une chaîne `x` en le rangeant dans `al` par `mov al, [x + i - 1]`.

On **écrase** le `i`e caractère d'une chaîne `x` en la valeur contenue dans `al` par `mov [x + i - 1], al`.

Toute fonction ayant un argument chaîne de caractère travaille à partir de l'adresse de cette chaîne. Il s'agit d'un **passage par adresse**.

# Fonctions — exemple 3

## – Exemple –

```
; Fonction qui affiche une chaine de carac.
; Arguments :
;       (1) adresse de la chaine de carac.
; Renvoi : nombre caracteres affiches.

print_string:
    ; Debut.
    push ebp
    mov ebp, esp

    ; Sauv. des registres.
    push ebx

    ; Adresse du debut de la chaine.
    mov ebx, [ebp + 8]

    ; Init. compteur
    mov eax, 0
```

```
    ; Boucle d'affichage.
    boucle:
        cmp byte [ebx], 0
        je fin_boucle

        ; Appel de print_char.
        push dword [ebx]
        call print_char
        add esp, 4

        add ebx, 1
        add eax, 1
        jmp boucle
    fin_boucle:
```

```
    ; Rest. des registres.
    pop ebx

    ; Fin.
    pop ebp
    ret
```