

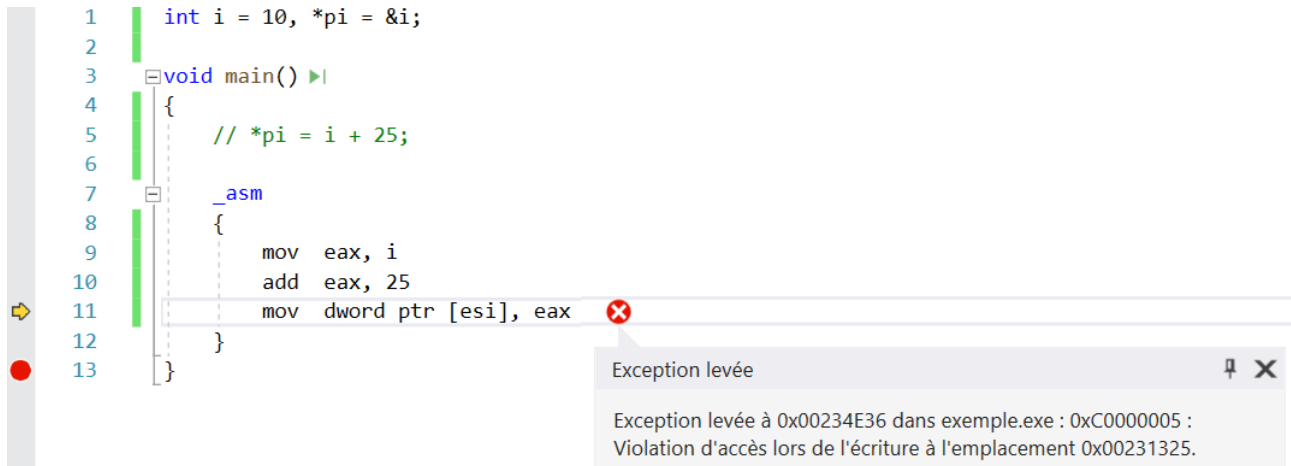
Liste d'exercices 9 : les appels de fonction (2 séances)

Notions à considérer pour pouvoir faire les exercices

Remarque concernant les pointeurs :

Il peut arriver que vous rencontriez l'erreur suivante durant l'exécution d'un programme :
"Exception levée ... Violation d'accès ...".

Exemple :



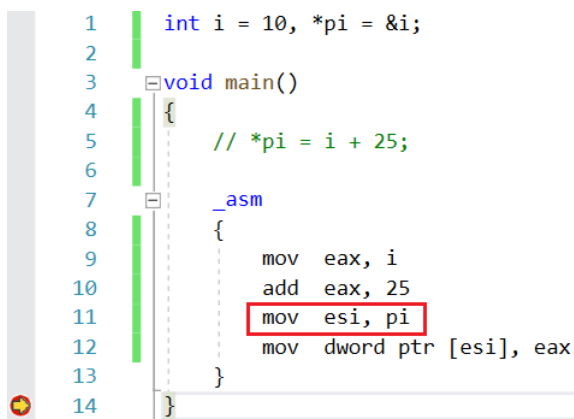
```
1  int i = 10, *pi = &i;
2
3  void main()
4  {
5      // *pi = i + 25;
6
7      _asm
8      {
9          mov  eax, i
10         add  eax, 25
11         mov  dword ptr [esi], eax
12     }
13 }
```

Exception levée

Exception levée à 0x00234E36 dans exemple.exe : 0xC0000005 : Violation d'accès lors de l'écriture à l'emplacement 0x00231325.

C'est le processeur qui signale qu'il ne veut pas exécuter l'instruction sur laquelle figure la flèche jaune. Ce problème survient quand on oublie de charger une adresse correcte dans le registre utilisé en tant que pointeur, ici ESI, avant de faire l'accès indirect en mémoire.

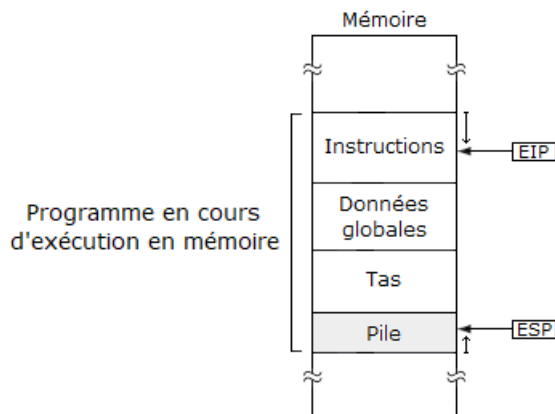
Solution :



```
1  int i = 10, *pi = &i;
2
3  void main()
4  {
5      // *pi = i + 25;
6
7      _asm
8      {
9          mov  eax, i
10         add  eax, 25
11         mov  esi, pi
12         mov  dword ptr [esi], eax
13     }
14 }
```

La pile :

La pile est une zone particulière de mémoire appartenant au programme en cours d'exécution dont le mode d'accès est LIFO (Last In First Out). ESP est le registre de 32 bits du processeur qui contient l'adresse du dernier élément empilé, c'est-à-dire l'adresse de l'élément situé au sommet de la pile.



Instructions utiles :

- *PUSH op* empile la valeur de l'opérande *op*. En supposant un opérande *op* d'une taille de 4 octets, lors de l'exécution de cette instruction, le processeur diminue l'adresse stockée dans ESP de 4 unités. Cette diminution du contenu de ESP permet de réserver les 4 octets dans la pile pour le nouveau sommet de pile. Ensuite, le processeur copie la valeur de l'opérande *op* dans ce nouveau sommet de pile.

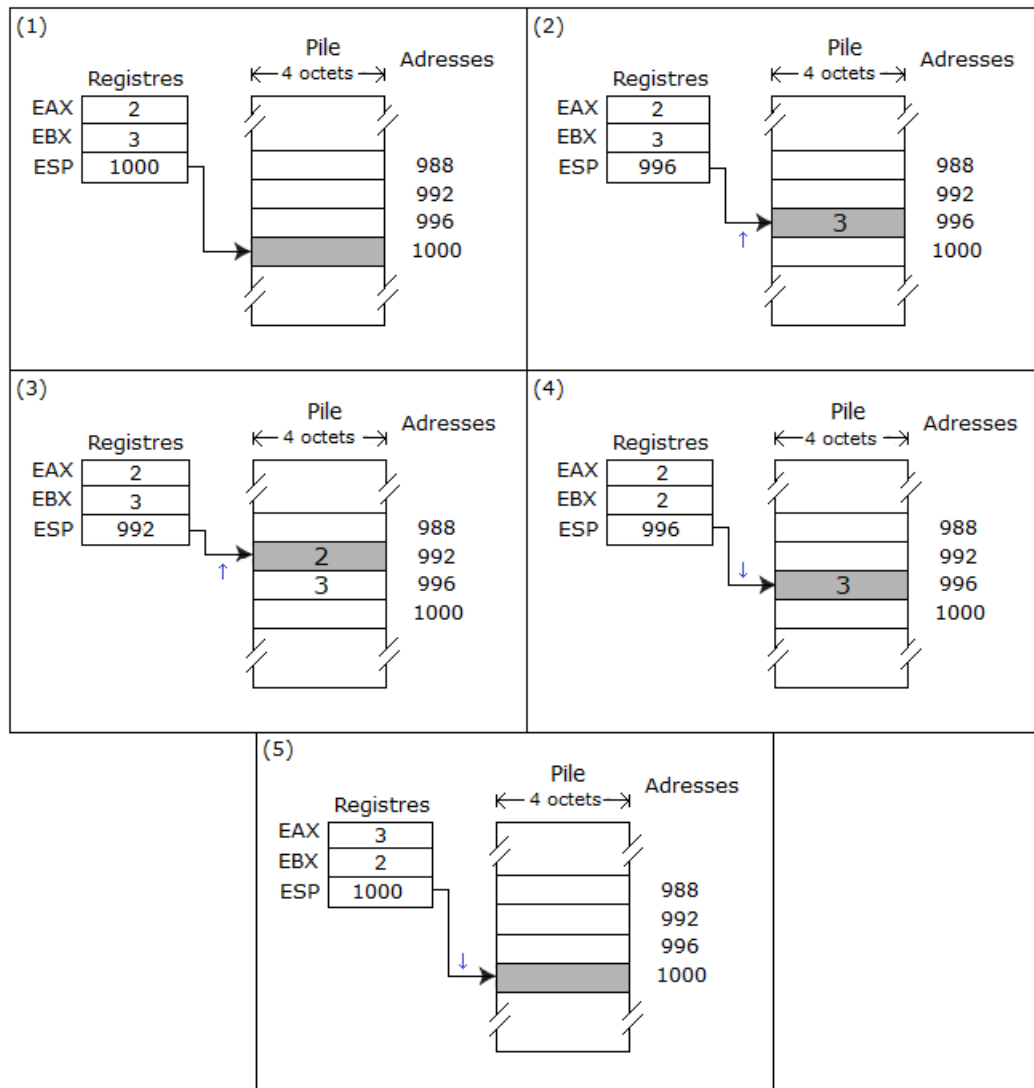
On doit parfois écrire *push dword ptr op* pour indiquer explicitement que l'opérande *op* a une taille de 4 octets. On utilise *dword ptr* lorsqu'on empile 4 octets appartenant à un double (voir les exemples à la suite) ou lorsqu'on empile une valeur immédiate (exemple : *push dword ptr 4*).

- *POP op* dépile vers l'opérande *op*. En supposant un opérande *op* d'une taille de 4 octets, lors de l'exécution de cette instruction, le processeur copie la valeur du sommet de pile dans l'opérande *op*. Ensuite, le processeur augmente l'adresse stockée dans ESP de 4 unités. Cette augmentation du contenu de ESP permet de libérer les 4 octets en mémoire occupé par le sommet de pile et donc de supprimer celui-ci.

Exemple : échanger le contenu de EAX avec celui de EBX en passant par la pile:

```
mov  eax, 2  // (1)
mov  ebx, 3
push ebx     // (2)
push eax     // (3)
pop  ebx     // (4)
pop  eax     // (5)
```

Les diagrammes suivants montrent, pendant l'exécution de ces 5 instructions, comment évolue le contenu de la pile et le contenu des registres EAX, EBX et ESP. On suppose, au départ, l'adresse 1000 dans ESP (ce qui est en gris représente le sommet de pile) :



Fonctions de bibliothèques du C :

Nous allons utiliser en assembleur les fonctions suivantes.

- Fichier **stdlib.h** à inclure :

<i>Prototype</i>	<i>Fonctionnement</i>
int abs(int x)	Retourne la valeur absolue de x

- Fichier **math.h** à inclure :

<i>Prototype</i>	<i>Fonctionnement</i>
double pow(double x, double y)	Retourne x^y
double sqrt(double x)	Retourne la racine carrée de x
double cos(double x)	Retourne le cosinus de x
double sin(double x)	Retourne le sinus de x
double exp(double x)	Retourne e^x (e correspond à 2.718)
double log(double x)	Retourne le logarithme en base e de x
double ceil(double x)	Retourne la valeur entière supérieure ou égale à x qui est la plus proche de x
double floor(double x)	Retourne la valeur entière inférieure ou égale à x qui est la plus proche de x
double fabs(double x)	Retourne la valeur absolue de x

Les appels de fonction en assembleur :

Instructions utiles :

- *CALL dword ptr fct* appelle la fonction *fct*. Lors de l'exécution de cette instruction, le processeur diminue l'adresse stockée dans ESP de 4 unités pour réserver un nouveau sommet de pile. Puis, le processeur copie dans ce nouveau sommet de pile l'adresse se trouvant dans EIP (EIP est le registre de 32 bits du processeur qui contient à ce moment l'adresse de l'instruction située juste après l'instruction call dans le programme). Ensuite, le processeur place dans EIP l'adresse de la première instruction de la fonction *fct*, ce qui provoque alors un branchement vers le début de la fonction *fct*.
- *FSTP op* copie dans l'opérande *op* qui est une variable du type float ou double la valeur stockée dans la FPU (Floating-Point Unit, unité de traitement sur flottants). FSTP n'accepte pas un registre xmm comme opérande. C'est avec FSTP qu'on va récupérer un flottant retourné par une fonction.

Avant d'invoquer le code de la fonction avec CALL, on va d'abord transmettre via la pile les paramètres attendus par la fonction. Après le retour au code appelant, on doit supprimer de la pile ces paramètres. Ensuite, on récupère la valeur retournée par la fonction.

Exemple 1 : usage de la fonction `abs()` dont le prototype est `int abs(int x)` :

```
int i;
```

<i>Langage C</i>	<i>Assembleur</i>
<code>i = abs(-5);</code>	<code>push dword ptr -5</code> <code>call dword ptr abs</code> <code>add esp, 4</code> <code>mov i, eax</code>

- Avec `push dword ptr -5`, on passe la valeur -5 en paramètre. Cette valeur est un entier, car `abs()` attend un entier en paramètre.
- Avec `call dword ptr abs`, on invoque la fonction `abs()`.
- Avec `add esp, 4`, on supprime le paramètre -5 occupant 4 octets dans la pile. La valeur numérique *n* dans `add esp, n` correspond ainsi à la taille en octets du/des paramètres passés à une fonction.
- Avec `mov i, eax`, on transfère dans la variable *i* la valeur entière retournée par la fonction. En C, une fonction retourne une valeur entière avec le registre EAX.

Exemple 2 : usage de la fonction `sqrt()` dont le prototype est `double sqrt(double x)` :

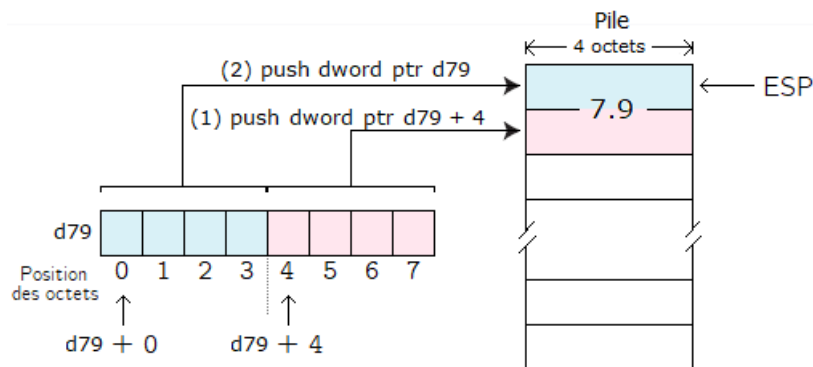
```
double d;
const double d79 = 7.9;
```

<i>Langage C</i>	<i>Assembleur</i>
<code>d = sqrt(7.9);</code>	<pre>push dword ptr d79 + 4 push dword ptr d79 call dword ptr sqrt add esp, 8 fstp d</pre>

- Avec `push dword ptr d79 + 4` et `push dword ptr d79`, on passe la valeur 7.9 du type double en paramètre. Cette valeur est du type double, car `sqrt()` attend un double en paramètre.

Pour passer une valeur du type double en paramètre :

1. Tout d'abord, on place la valeur 7.9 dans une constante, ici la constante d79.
2. Ensuite, on passe le contenu de cette constante en paramètre en utilisant à 2 reprises l'instruction `push`, car un double a une taille de 8 octets et l'instruction `push` ne permet d'empiler que 4 octets à la fois.



`push dword ptr d79 + 4` empile les 4 octets les plus éloigné du double et `push dword ptr d79` empile les 4 octets au début du double.

- Avec `call dword ptr sqrt`, on invoque la fonction `sqrt()`.
- Avec `add esp, 8`, on supprime le paramètre 7.9 occupant 8 octets dans la pile.
- Avec `fstp d`, on transfère dans la variable d la valeur du type double retournée par la fonction qui se trouve dans la FPU du processeur. En C, une fonction retourne une valeur du type float ou du type double avec la FPU du processeur (Floating-Point Unit, unité de traitement sur flottants).

Exemple 3 : passage en paramètre de la valeur d'un registre xmm0 :

```
double d;  
const double d79 = 7.9;
```

<i>Langage C</i>	<i>Assembleur</i>
<code>d = sqrt(7.9 + d);</code>	<code>movsd xmm0, d79 addsd xmm0, d sub esp, 8 movsd qword ptr [esp], xmm0 call dword ptr sqrt add esp, 8 fstp d</code>

On ne peut empiler la valeur d'un registre xmm avec l'instruction push. Pour contourner ce problème, on peut implémenter manuellement l'instruction push ainsi :

1. Avec `sub esp, 8`, on réserve tout d'abord 8 octets dans la pile.
2. Avec `movsd qword ptr [esp], xmm0`, on copie ensuite la valeur de xmm0 dans les 8 octets réservés dans la pile.

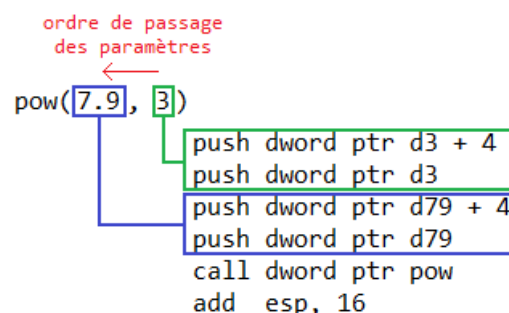
Ces 2 instructions permettent ainsi de passer en paramètre la valeur de xmm0 à la fonction `sqrt()`.

Exemple 4 : usage de la fonction `pow()` dont le prototype est *double pow(double x, double y)* :

```
double d;  
const double d79 = 7.9, d3 = 3.0;
```

<i>Langage C</i>	<i>Assembleur</i>
<code>d = pow(7.9, 3);</code>	<code>push dword ptr d3 + 4 push dword ptr d3 push dword ptr d79 + 4 push dword ptr d79 call dword ptr pow add esp, 16 fstp d</code>

En C, quand il y a plusieurs paramètres à passer à une fonction, l'ordre de passage de ces paramètres est inversé par rapport à l'ordre dans lequel ils sont inscrits entre les parenthèses. Ceci explique pourquoi on transmet en paramètre à la fonction `pow()` d'abord la valeur 3, puis la valeur 7.9.



Les paramètres occupent 16 octets dans la pile. On les supprime avec `add esp, 16` après le retour au code appelant.

Problèmes courants :

- Il peut arriver que vous deviez stocker des résultats intermédiaires dans la pile ou dans des variables temporaires.

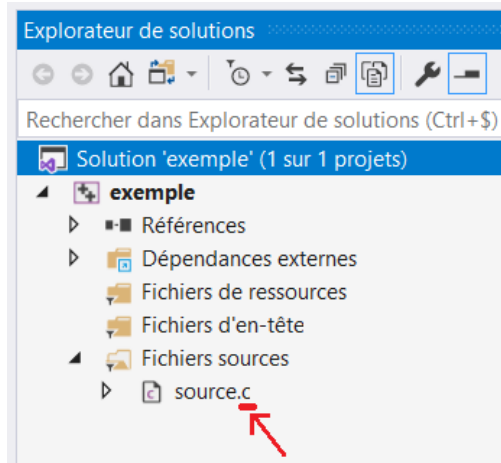
Exemple :

```
1  #include <math.h>
2
3  int i = 56, *pi = &i;
4  double d, dtmp;
5  const double d35 = 3.5;
6
7  void main()
8  {
9      //d = *pi + sqrt(3.5);
10
11      _asm
12      {
13          mov ebx, pi
14          mov eax, dword ptr[ebx]
15          push eax
16          push dword ptr d35 + 4
17          push dword ptr d35
18          call dword ptr sqrt
19          add esp, 8
20          fstp dtmp
21          pop eax
22          cvtsi2sd xmm0, eax
23          addsd xmm0, dtmp
24          movsd d, xmm0
25      }
26  }
```

L'instruction `push eax` à la ligne 15 est utilisée pour sauver la valeur de `eax`, car la fonction `sqrt()` utilisée à la ligne 18 modifie `eax`. La valeur d'origine de `eax` est récupérée ensuite à la ligne 21 avec `pop eax`. La variable `dtmp` sert à la ligne 20 à récupérer la valeur retournée par la fonction `sqrt()`, puis cette valeur est utilisée à la ligne 23.

- Quand vous ajoutez au projet le fichier source dans lequel vous allez faire les appels de fonction, veillez bien à indiquer `.c` comme extension.

Exemple :



Exercices

Écrivez la séquence d'instructions en assembleur qui correspond à chaque instruction d'affectation en langage C suivante :

```
#include <stdlib.h>
#include <math.h>
```

```
char    b = -15, *pb = &b;
int     i = 0xff1, *pi = &i;
float   f = 352.318, *pf = &f;
double  d = 975.24, *pd = &d;
```

- `*pd = abs(b);` // d = 15.0
- `f = *pi * (float)sqrt(d);` // f = 127444.867
- `*pd = log(f) / log(2);` // d = 8.460734346...
- `f = sqrt(d) + pow(b, 2) * exp(4);` // f = 12315.8125
- `*pi = -((float)*pi / 2) * (f - 5.2 * sqrt(-*pb));` // i = -677810
- `f = pow(*pd, 3) / (*pb * 15 - sqrt(36));` // f = -4015342.00
- `*pi = ((int)floor(*pd) | 0xf00f) - 2 + fabs(b * f);` // i = 67697
- `d = -sin(b * 3 + 5) + pow(*pi + *pd + f, 3);` // d = 158213840345.31961
- `f = cos(-i) * 100 - exp(b / 5 + 6) + *pf * d;` // f = 343474.781
- `d = ceil(*pd * 3 + -i) / (cos(2.9) * 6);` // d = 198.25776939662740