TD 2: Sémantique : environnement, mémoire et pile

Dans les quatre exercices ci-dessous, vous devez donner les *tableaux d'activation* des fonctions puis les états successifs de l'*environnement* et de la *mémoire* pour les principales étapes du programme et préciser les affichages à l'écran lorsqu'il y en a. On suppose que le segment de pile est constitué des adresses de 0 à 99 et que le segment de données statiques est constitué des adresses de 100 à 199.
Note: *Il est conseillé pour chaque exercice de recopier le code au tableau afin de pouvoir simuler son exécution de façon plus démonstrative.*

Ce TD passe plus ou moins bien... Pour les motiver, il faut leur dire au début du TD que quand on passera à des programmes plus complexes avec des références, ils auront besoin de la pile pour comprendre ce qu'il se passe. Comme on ne veut pas tout leur faire d'un coup, on commence par la pile dans une version simple puis on ajoute les paramètres et les références.

Si des élèves vous demandent si à l'examen il faudra redessiner la pile à chaque petit changement, la réponse est la suivante: en général à l'examen on leur demande de dessiner l'état de la pile à un moment précis de l'exécution du programme (par exemple en donnant un numéro de ligne du programme). Sur leur copie ils ont donc uniquement à dessiner l'état de la pile à ce moment là, mais pour réussir à le trouver et avoir une réponse parfaitement juste, il faut qu'ils aient fait sur leur brouillon l'état de la pile à chaque moment du début de l'exécution du programme jusqu'au moment demandé. A eux de voir comment ils gèrent leur brouillon de façon à être surs de ne pas se tromper.

1 – Affectations simples

Note: *Un exercice très simple pour commencer, l'objectif ici est surtout d'être sûr que tout le monde comprend bien la manière de représenter la pile et l'importance de réaliser les opérations une par une en séquence de la même manière que l'ordinateur. Le fun commence plus tard.*

```
1 #include <iostream>
2 using namespace std;
3 int main () {
4     int n1, n2;
5     cout << "Saisissez n1:" << endl;
6     cin >> n1;    // on suppose que l'on tape 2
7     cout << "Saisissez n2:" << endl;
8     cin >> n2;    // on suppose que l'on tape 3
9     n1 = n2;
10    n2 = n1;
11    cout << "n1 vaut " << n1 << endl;
12    cout << "n2 vaut " << n2 << endl;
13    return 0;
14 }
```

Correction:

On commence par l'état du programme à la compilation. Le tableau d'activation construit pour cette fonction `main()` *prévoit 4 cases:*

`main`
`n1`
`n2`
`return`

Lors de l'exécution du programme, l'espace nécessaire prévu par le tableau d'activation de la fonction est alloué dans la mémoire aux premières places disponibles de celle-ci: On initialise ADM (données administratives, adresse de retour), ici on retient que à la fin du main on quittera le programme.

	4	?
main	3	ADM
n1	2	?
n2	1	?
return	0	?

Les effets de ce programme sont les suivants : on affecte à `n1` *la valeur saisie par l'utilisateur en ligne (6), de même pour n2, ligne (8). Par exemple 2 et 3.*

	4	?
main	3	ADM
n1	2	**2**
n2	1	**3**
return	0	?

Puis on affecte à `n1` *la valeur de* `n2`.

	4	?
main	3	ADM
n1	2	**3**
n2	1	3
return	0	?

Enfin, on affecte à `n2` *la valeur de* `n1`.

	4	?
main	3	ADM
n1	2	3
n2	1	**3**
return	0	?

Après les impressions, on a `n1=3` *et* `n2=3`, *la valeur 0 est affectée au return puis on dépile le tout.*

Modifiez ensuite ce code pour que les variables `n1` et `n2` échangent effectivement leur contenu, puis refaire les tableaux d'activation correspondant à cette nouvelle version du code.

Correction: *On introduit une variable temporaire pour réaliser l'échange :*

```
1  #include <iostream>
2  using namespace std;
3  int main () {
4      int n1, n2, temp;
5      cout << "Saisissez n1:" << endl;
6      cin >> n1;    // on suppose que l'on tape 2
7      cout << "Saisissez n2:" << endl;
8      cin >> n2;    // on suppose que l'on tape 3
9      temp = n1;
10     n1 = n2;
11     n2 = temp;
12     cout << "n1 vaut " << n1 << endl;
```

```
13      cout << "n2 vaut " << n2 << endl;
14      return 0;
15  }
```

Le tableau d'activation construit pour cette fonction `main()` *prévoit 5 cases:*

main
n1
n2
temp
return

Toujours avec 2 et 3 comme valeurs saisies par l'utilisateur

	5	?
main	4	ADM
n1	3	**2**
n2	2	**3**
temp	1	?
return	0	?

Puis on affecte à `temp` *la valeur de* `n1`*.*

	5	?
main	4	ADM
n1	3	2
n2	2	3
temp	1	**2**
return	0	?

Ensuite on affecte à `n1` *la valeur de* `n2`*.*

	5	?
main	4	ADM
n1	3	**3**
n2	2	3
temp	1	2
return	0	?

Enfin, on affecte à `n2` *la valeur initiale de* `n1` *que l'on a sauvegardée dans* `temp`*.*

	5	?
main	4	ADM
n1	3	3
n2	2	**2**
temp	1	2
return	0	?

L'affichage obtenu est `n1=3` *et* `n2=2`*, le résultat attendu. On place ensuite la valeur 0 dans* `return` *et on dépile le tout.*

2 – Première fonction

Note: *Ici on introduit deux notions: le segment statique et une fonction. Faire remarquer aux étudiants que les deux fonctions ont une variable* `x` *mais que ce sont deux variables différentes avec des portées différentes et qui seront stockées dans des cases différentes.*

Insister aussi sur le fait que même si dans le code il y a des fonctions définies avant le main, à l'exécution on commence par empiler le bloc du main uniquement. Si une fonction n'est pas appelée, elle ne sera pas exécutée.

```
1 #include <iostream>
```

```
 2 using namespace std;
 3 int y;
 4 int P1() {
 5     int x = y * 3;
 6     return x;
 7 }
 8 int main() {
 9     int x;
10     x = 5;
11     y = 2 * x;
12     x = P1() + 3;
13     cout << "x = " << x << endl;
14     return 0;
15 }
```

Note: *Cet exercice à pour objectif de montrer le principe d'appel d'une fonction à l'aide de la pile et la visibilité des variables. Pensez à bien discuter avec les étudiants de chaque utilisation de la variable* x *en particulier et de comment le mécanisme de la pile permet d'obtenir des variables avec une portée lexicale.*

Correction:

On commence par l'état du programme à la compilation. Les tableaux d'activation construits pour les fonctions sont très similaires. La variable y *est une variable globale et sera stockée au début du segment de données statiques à l'adresse 100.*

P1
x
return

main
x
return

Puis on démarre l'exécution en commencant par la fonction `main`*. Au début du programme l'état de la mémoire est:*

y	100	?
	0	?

À l'entrée dans la fonction `main` *on empile son tableau d'activation. ADM en case mémoire 2 denote les informations administratives, ici on retient que à la fin du* `main` *on quittera le programme.*

y	100	?
	3	?
main	2	**ADM**
x	1	?
return	0	?

Ligne 10: affectation de la variable x *locale à la fonction* `main`*.*

y	100	?
	3	?
main	2	ADM
x	1	**5**
return	0	?

Ligne 11: affectation de la variable y *globale donc dans le segment de données statiques.*

y	100	**10**
	3	?
main	2	ADM
x	1	5
return	0	?

Ligne 12: appel de la fonction `P1`*, l'espace nécessaire prévu par le tableau d'activation de la fonction est alloué sur la pile. Le bloc du* `main` *n'est plus visible. ADM en case mémoire 5 retient que à la fin de* `P1`*, il faudra revenir à la ligne 15 du* `main` *exécuter l'instruction* `x=P1()+3;` *après avoir remplacé* `P1` *par la valeur retournée.*

y	100	10
	6	?
P1	5	**ADM**
x	4	?
return	3	?
	2	ADM
	1	5
	0	?

Dans `P1` *en ligne 5: affectation de la variable* `x` *locale à la fonction* `P1`*. La valeur de* `y` *est lue depuis le segment statique.*

y	100	10
	6	?
P1	5	ADM
x	4	**30**
return	3	?
	2	ADM
	1	5
	0	?

Dans `P1` *en ligne 6: préparation de la fin de la fonction, la valeur retournée est placée dans la case prévue.*

y	100	10
	6	?
P1	5	ADM
x	4	30
return	3	**30**
	2	ADM
	1	5
	0	?

Ligne 12: le return nous ramène dans le main où l'on a remplacé l'appel à `P1()` *par la valeur retournée. On exécute donc* `x=30+3;`*.*

y	100	10
	6	?
	5	ADM
	4	30
	3	30
main	2	ADM
x	1	**33**
return	0	?

Ligne 13: on affiche `x = 33`*.*

Ligne 14: On met en place la valeur de retour de la fonction `main`*.*

y	100	10
	6	?
	5	ADM
	4	30
	3	30
main	2	ADM
x	1	33
return	0	**0**

Fin du programme

y	100	10
	6	?
	5	ADM
	4	30
	3	30
	2	ADM
	1	33
	0	0

3 – Plus compliqué

```
1  #include <iostream>
2  using namespace std;
3  int y = 10;
4  int P1() {
5      int x;
6      x = 7;
7      return x + 1;
8  }
9  void P2() {
10     int x, y;
11     x = 3;
12     y = P1();
13     x = y;
14 }
15 void main() {
16     int x;
17     x = 1;
18     y = P1() + 4;
19     P2();
20     cout << "x = " << x << endl;
21 }
```

Correction:

On commence par l'état du programme à la compilation. La variable y *est une variable globale et sera stockée au début du segment de données statiques à l'adresse 100.*

P1	P2
x	x
return	y

main
x

Puis on démarre l'exécution en commencant par la fonction `main`*. On note que cette fois ci la variable* y *est initialisée. Au début du programme l'état de la mémoire est:*

y	100	10
	0	?

À l'entrée dans la fonction `main` *on empile son tableau d'activation.*

y	100	10
	2	?
main	1	**ADM**
x	0	?

Ligne 17 : rien de particulier.

y	100	10
	2	?
main	1	ADM
x	0	**1**

Ligne 18 : appel de la fonction `P1`*. Son tableau d'activation est alloué sur la pile, le bloc de la fonction* `main` *n'est plus visible.*

y	100	10
	5	?
P1	4	**ADM**
x	3	?
return	2	?
	1	ADM
	0	1

Ligne 6 et 7 dans P1 : affectation de la variable `x` *puis calcul de la valeur de retour.*

y	100	10
	5	?
P1	4	ADM
x	3	**7**
return	2	**8**
	1	ADM
	0	1

Ligne 18 : l'appel à `P1()` *est maintenant terminé, on retourne donc au bloc de* `main` *pour finir le calcul de la variable* `y`*. Les cases utilisées par l'appel à* `P1()` *conservent leurs valeurs même si elles ne sont maintenant plus accessibles.*

y	100	**12**
	5	?
	4	ADM
	3	7
	2	8
main	1	ADM
x	0	1

Ligne 19 : appel de la fonction `P2()`*. On empile son bloc qui occupe donc les mêmes case que le précédent appel à* `P1()`*. Attention, tant que les variables n'ont pas été initialisées elles ont pour valeur les valeurs qui étaient là avant ! La variable* `y` *globale n'est plus visible car masquée par la variable locale de* `P2()`*.*

	100	**12**
	5	?
P2	4	**ADM**
x	3	7
y	2	8
	1	ADM
	0	1

Lignes 11 et 12 dans P2. Après l'affectation de 3 à `x`*, on passe à l'appel de* `P1()` *et on alloue son bloc au sommet de la pile. Le bloc de* `P2()` *n'est plus visible et le* `y` *global redevient visible. On remarque que la variable* `x` *de la fonction* `P1()` *est à une adresse différente du premier appel à la fonction.*

y	100	12
	8	?
P1	7	**ADM**
x	6	?
return	5	?
	4	ADM
	3	**3**
	2	8
	1	ADM
	0	1

Lignes 6 et 7 dans P1. On affecte à x *la valeur 7 puis on calcule la valeur de retour.*

y	100	12
	8	?
P1	7	ADM
x	6	**7**
return	5	**8**
	4	ADM
	3	3
	2	8
	1	ADM
	0	1

Lignes 12 et 13 dans P2. Sortie de P1 *et retour dans* P2*. On dépile le bloc de* P1 *donc le bloc de* P2 *redevient visible ce qui de nouveau cache la variable globale* y*. La valeur de retour est donc affectée au* y *de* P2 *puis* y *est affectée à* x*.*

	100	12
	8	?
	7	ADM
	6	7
	5	8
P2	4	ADM
x	3	**8**
y	2	**8**
	1	ADM
	0	1

Lignes 19 et 20. Sortie de la fonction P2 *et retour au* main*. Le* x *qui est affiché est celui du* main *qui vaut 1.*

y	100	12
	8	?
	7	ADM
	6	7
	5	8
	4	ADM
	3	8
	2	8
main	1	ADM
x	0	1

4 – Avec des paramètres

```
1 #include <iostream>
2 using namespace std ;
3 int pgcd(int a, int b){
4     int reste;
5     do {
6         reste = a % b;
7         a = b;
8         b = reste;
9     } while (reste != 0);
10    return a;
11 }
12 void main() {
13    int a = 75, b = 60;
```

```
14      int r = pgcd(a, b);
15      cout << "pgcd(" << a << ", " << b << ")=" << r << endl;
16  }
```

Correction:

On commence par l'état du programme à la compilation. Les tableaux d'activation construits sont similaires à ceux des exercices précédents. On note juste que les paramètres de la fonction `pgcd` *sont traités comme des variables locales.*

pgcd
a
b
reste
return

main
a
b
r

On démarre l'exécution en commencant par la fonction `main` *avec une mémoire vide.*

0	?

À l'entrée dans la fonction `main` *on empile son tableau d'activation et les variables* `a` *et* `b` *sont initialisées à la ligne 13.*

	4	?
main	3	**ADM**
a	2	**75**
b	1	**60**
r	0	?

Ligne 14: appel de la fonction `pgcd`*. On empile son tableau d'activation et on affecte les valeurs des paramètres. Bien noter ici que les valeurs sont copiées dans de nouvelles cases.*

	9	?
pgcd	8	**ADM**
a	7	**75**
b	6	**60**
reste	5	?
return	4	?
	3	ADM
	2	75
	1	60
	0	?

Ligne 6: premier calcul du reste, on accède aux copies de `a` *et* `b` *et non pas aux valeurs originales dans la fonction* `main` *dont l'environnement n'est plus visible.*

	9	?
pgcd	8	ADM
a	7	75
b	6	60
reste	5	**15**
return	4	?
	3	ADM
	2	75
	1	60
	0	?

Ligne 7 et 8: Affectation des nouvelles valeurs de `a` *et* `b`*, on travaille ici sur les paramètres qui dans la fonction se comportent comme des variables locales. On peut donc les modifier et cela n'a aucune influence sur les valeurs d'origine stockées dans la fonction* `main`*.*

	9	?
pgcd	8	ADM
a	7	**60**
b	6	**15**
reste	5	15
return	4	?
	3	ADM
	2	75
	1	60
	0	?

Ligne 9: Le test est vrai, on va donc repartir pour un deuxième tour de boucle. En fonction de la compréhension des étudiants, détailler plus ou moins le calcul ici. Après l'exécution des trois lignes de la boucle on obtient:

	9	?
pgcd	8	ADM
a	7	**15**
b	6	**0**
reste	5	**0**
return	4	?
	3	ADM
	2	75
	1	60
	0	?

Ligne 9 et 10: Le test est maintenant faux, on arrête donc la boucle et on passe à l'exécution du return.

	9	?
pgcd	8	ADM
a	7	15
b	6	0
reste	5	0
return	4	**15**
	3	ADM
	2	75
	1	60
	0	?

Ligne 14: La fonction `pgcd` *se termine, on dépile donc son enregistrement d'activation et on revient dans le* `main` *pour affecter la valeur de retour.*

	9	?
	8	ADM
	7	15
	6	0
	5	0
	4	15
main	3	ADM
a	2	75
b	1	60
r	0	**15**

Ligne 15: On réalise l'affichage du résultat. L'affichage de `a` *et* `b` *se fait bien avec les valeurs initiales qui n'ont pas été modifiées par l'appel de la fonction.*

```
pgcd(75, 60)=15
```

Note: *S'il vous reste du temps et que vous avez fini tous les exercices du TD précédent, pour faire la transition avec le prochain TD, faites les réfléchir à une fonction qui permettrait de simplifier une fraction.*

Quelles sont les données (numérateur et dénominateur) et les résultats (numNew et denNew). Combien de résultat et donc fonction qui n'en renvoie qu'un seul ou procédure avec des paramètres résultats...

```
1  #include <iostream>
2  using namespace std ;
3  int pgcd(int a, int b){
4      int reste;
5      do {
6          reste = a % b;
7          a = b;
8          b = reste;
9      } while (reste != 0);
10     return a;
11 }
12 void simplifier( int n, int d, int &nNew, int &dNew) {
13     int div = pgcd(n, d);
14     nNew = n / div;
15     dNew = d / div ;
16 }
17 int main(){
18     int a, b, c, d;
19     cout << " numerateur et denominateur ?" ;
20     cin >> a >> b ;
21     simplifier(a, b, c, d);
22     cout << c << " " << d << endl;
23     return 0;
24 }
```