# TP2 : Vie et mort des variables en mémoire

## Exercice 1 : Modèle de pile pour l'adressage

Reprenez le fichier `Vecteur3D.cpp` que vous avez créé au TP précédent et ajoutez-y des « cout » pour visualiser les adresses mémoires des paramètres et des variables locales des différents sous-programmes.

**Aide** : pour afficher l'adresse d'une variable, vous pouvez la présenter en tant que long int :

```
cout << "Adresse de monFloat : " << (long int) &monFloat;
```

Vous pouvez aussi demander la taille en octets occupée par une variable ou un type avec l'opérateur sizeof :

```
cout << "Taille occupée par monTab : " << sizeof(monTab);
```

Comparez l'évolution théorique de la pile avec ce qui se passe en réalité :

a. Dans quel ordre sont empilés les éléments d'un tableau ? La case 0 a-t-elle l'adresse la plus haute ou la plus basse ?

La case 0 a l'adresse la plus basse. Ainsi, on vérifie bien l'équivalence vue en cours :
tab[i] ⇔ *(tab + i).

b. Dans quel ordre sont empilés les paramètres d'une fonction ou d'une procédure ?

Cela dépend de la machine. Les paramètres sont généralement empilés de la droite vers la gauche (i.e. avec les premiers paramètres avec les adresses les plus basses), contrairement à ce que l'on fait en TD.

c. Dans quel ordre sont empilées les données membres d'un objet ?

Les données sont empilées du premier au dernier, comme ce que l'on fait en TD.

d. Les variables locales d'un même sous-programme sont-elles, comme en TD, dans l'ordre dans lequel elles étaient déclarées dans le code ?

Pas forcément. Le compilateur peut réorganiser les variables d'un même bloc. De plus, le compilateur peut insérer des octets « blancs » pour aligner les adresses mémoire. Ces octets « blancs » peuvent être insérés, par exemple, entre les champs d'une structure, ou au bout d'un tableau (par exemple un tableau de 30 char occupera en réalité 32 octets, aligné sur un multiple d'un mot c'est-à-dire 4 octets). Ces octets de « padding » ne sont pas pris en compte dans le modèle théorique de pile vu en CM et en TD. cf http://en.wikipedia.org/wiki/Data_structure_alignment

e. Lorsque l'un des paramètres d'un sous-programme est un tableau, combien d'octets ce paramètre occupe-t-il dans la frame du sous-programme ? Le tableau est-il recopié dans cette frame ?

Le tableau n'est pas recopié, seule son adresse de base est transmise au sous-programme. Le paramètre qui correspond au tableau n'occupe donc que 8 octets. Attention sur une machine 32 bits, cette valeur sera de 4 octets.

f. Quel écart observez-vous entre l'adresse la plus haute et l'adresse la plus basse, parmi les adresses affichées ? Cet écart correspond-il à l'écart théorique (celui de votre trace « papier ») ? Demandez à votre encadrant d'où vient cette différence.

L'écart réel sera plus grand que l'écart prévu sur papier. Cette différence vient du fait que le modèle théorique de pile utilisé en CM et en TD ne prend pas tout en compte. On ignore notamment : les objets temporaires utilisés pour le calcul des expressions, l'adresse de retour du pointeur d'instruction, le « frame

pointer », la sauvegarde des registres, et les octets de « padding » pour l'alignement des adresses. Ainsi on aura des « trous » :
- entre les variables locales de la fonction appelante et les paramètres de la fonction appelée,
- entre les paramètres d'une fonction et ses variables locales.

## Exercice 2 : Modèle de pile pour les appels récursifs et visualisation des frames avec gdb

a. Tapez le programme ci-dessous qui calcule le nombre de combinaisons de p parmi n dans un nouveau fichier. Compilez-le et exécutez-le pour vérifier que tout fonctionne bien.

```
1  #include <iostream>
2  using namespace std;
3
4  /* Calcul d'un coefficient binomial à l'aide du triangle de Pascal */
5  int comb(int n, int p) {
6    int tmp1, tmp2;
7    cout <<"Calcul du nb de combinaisons de "<<p<<" elts parmi "<<n<<endl;
8
9    if ((p==0) || (n==p))
10     return 1;
11   tmp1 = comb(n-1, p-1);  /* premier appel récursif */
12   tmp2 = comb(n-1, p);    /* second appel récursif */
13   return tmp1 + tmp2;
14 }
15
16 int main() {
17   int c;
18   c = comb(4, 3);
19   cout << "c vaut " << c << endl;
20   return 0;
21 }
```

b. Puis recompilez-le en ajoutant à GCC l'option `-g` (cette option ajoute à l'exécutable des informations de débogage), et lancez le débugger gdb avec la commande `gdb nomDeVotreExecutable`. Placez un point d'arrêt sur la ligne 10 (correspondant au `return 1;`) en tapant `break 10`. gdb vous informe que ce breakpoint est le numéro 1, on pourrait en mettre d'autres si on le souhaitait. Lancez ensuite l'exécution du programme en tapant `run` ou juste `r`. L'exécution va s'arrêter lorsqu'on va entrer dans le if, juste avant d'exécuter le return.

c. Demandez à gdb la liste des « frames » actives en mémoire à ce moment en tapant `backtrace`. Combien de frames voyez-vous pour comb ? Que se passe-t-il donc quand une fonction s'appelle elle-même : réutilise-t-on la même frame ou en empile-t-on une nouvelle à chaque appel ?

En principe, on voit 4 frames pour comb. On empile donc une nouvelle frame à chaque appel, conformément à ce qui a été fait en TD.

d. gdb identifie les frames par des numéros (frame #0, frame #1, etc…). Sélectionnez la frame 0 pour l'examiner, en tapant `select-frame 0`. Demandez ensuite à gdb quels sont les paramètres de cette frame et quelles sont les valeurs courantes de ses variables locales en tapant `info args` puis `info locals`. Pourquoi les valeurs de tmp1 et tmp2 sont-elles étranges ?

Parce qu'au moment du breakpoint, dans cette instance de comb, tmp1 et tmp2 ont été déclarées mais pas initialisées. Leurs valeurs sont donc arbitraires.

e. Recommencez pour une autre frame. Les valeurs des paramètres sont-elles les mêmes pour des frames différentes d'une même fonction ?

Non, chaque instance de comb a sa propre frame indépendante des autres, avec ses propres valeurs de paramètres.

f. Quittez gdb en tapant `quit`.

## Exercice 3 : Allocation dynamique de mémoire dans le main

Ecrivez dans un nouveau fichier un programme qui demande à l'utilisateur de taper la taille qu'il souhaite pour son tableau, alloue un tableau de réels de la taille demandée, et demande à l'utilisateur les valeurs des réels à stocker dans le tableau. Pour la lecture des saisies clavier, vous pourrez utiliser cin. Le programme affichera ensuite le tableau et se terminera proprement, c'est-à-dire en libérant la mémoire allouée dynamiquement.

```
#include <iostream>
using namespace std;

int main() {
  int tailleTab = 0;
  double * tab = NULL;
  cout << "Quelle est la taille du tableau? ";
  cin >> tailleTab;
  if (tailleTab <= 0) return 0;
  tab = new double [tailleTab];
  for (int i=0; i<tailleTab; i++) {
    cout << "Saisir la valeur pour tab["<<i<<"] : ";
    cin >> tab[i];
  }
  cout << "Affichage du tableau:" << endl;
  for (int i=0; i<tailleTab; i++) cout << "tab["<<i<<"] = " << tab[i] << endl;
  delete [] tab;
  return 0;
}
```

## Exercice 4 : Trois entêtes pour une même fonction

Reprenez la fonction comb de l'exercice 2. Vous allez faire deux implémentations supplémentaires de cette fonction de combinaison qui vont différer par leurs entêtes, c'est-à-dire par la façon dont les entrées et les sorties sont gérées.

a. Dans le fichier écrit à l'exercice 2, indiquez les pré- et post-conditions et résultat en commentaires de la fonction déjà réalisée.

b. Ajouter une procédure de même nom (comb) qui prend en paramètre n et p mais aussi le résultat du calcul de la combinaison en mode donnée-résultat. Indiquez en commentaires les pré- et post-conditions de cette procédure. Ajouter au main un appel à cette procédure avec les mêmes valeurs de n et p que précédement et vérifier que vous obtenez le même résultat (l'afficher).

c. Ajouter une autre procédure de même nom (comb) qui prend en paramètre n et p et un pointeur sur un entier alloué sur le tas dans lequel vous mettrez le résultat du calcul. Indiquez en commentaires les pré- et post-conditions de cette procédure. Ajouter au main un appel à cette procédure avec les mêmes valeurs de n et p que précédement et vérifier que vous obtenez le même résultat (l'afficher).

```
int comb (int n, int p) {
// Pré-conditions : n et p >= 0
// Post-conditions : aucune
// Résultat : le nombre de combinaisons de p éléments parmi n
    int tmp1, tmp2;
```

```
    if (p==0 || n==p) return 1;
    tmp1 = comb(n-1,p-1);
    tmp2 = comb(n-1,p);
    return tmp1+tmp2;
}

void comb (int n, int p, int& resultat) {
// Pré-conditions : n et p >= 0
// Post-conditions : resultat contient le nombre de combinaisons de p éléments parmi n
    int tmp1, tmp2;
    if (p==0 || n==p) {resultat=1; return;}
    comb(n-1,p-1,tmp1);
    comb(n-1,p,tmp2);
    resultat = tmp1+tmp2;
}

void comb(int n, int p, int* resultat) {
// Pré-conditions : n et p >= 0, resultat est un pointeur valide sur int
// Post-conditions : *resultat contient le nombre de combinaisons de p éléments parmi n
    int tmp1, tmp2;
    if (p==0 || n==p) {*resultat=1; return;}
    comb(n-1,p-1,&tmp1);
    comb(n-1,p,&tmp2);
    *resultat = tmp1+tmp2;
}

int main () {
    int c = comb(4,3);  // version 1
    cout << "comb(4,3)= " << c << endl;
    comb(4,3,c);        // version 2
    cout << "comb(4,3)= " << c << endl;
    int * cTas = new int;
    comb(4,3,cTas);     // version 3
    cout << "comb(4,3)= " << *cTas << endl;
    delete cTas;
    return 0;
}
```

## Exercice 5 : Arithmétiques des pointeurs

Ecrivez un programme principal exécutant les instructions suivantes :

- Afficher la taille d'un Vecteur3D en mémoire et la taille d'un pointeur sur Vecteur3D.
- Allouer un tableau `tabVecteurPile` de 3 Vecteur3D sur la pile et un autre tableau `tabVecteurTas` de même taille sur le tas.
- Remplir les deux tableaux avec les mêmes valeurs, tirées aléatoirement entre -10.0 et 10.0 avec exactement un chiffre après la virgule.
- Afficher la taille des variables `tabVecteurPile` et `tabVecteurTas`.
- Afficher l'adresse du premier élément des deux tableaux.
- Afficher les adresses des champs x,y,z du deuxième élément des deux tableaux.
- Afficher le contenu de `*(tabVecteurPile+2)` et `*(tabVecteurTas+2)`. A quoi cela correspond-t-il ?
- Affecter au champ x du premier élément de `tabVecteurPile` la différence entre l'adresse du champ y du deuxième élément de `tabVecteurTas` et l'adresse du champ z du troisième élément de `tabVecteurTas`. Faire une version où les adresses sont converties en `long int` avant soustraction, et une version où elles ne le sont pas. Afficher et expliquer les valeurs obtenues.

```
cout << "taille d'un Vecteur3D : " << sizeof(Vecteur3D) << " octets." << endl;
```

```
cout << "taille d'un pointeur sur Vecteur3D : " << sizeof(Vecteur3D*) << " octets." << endl;

Vecteur3D tabVecteurPile [3];
Vecteur3D * tabVecteurTas = new Vecteur3D [3];

srand((unsigned int)time(NULL));
for (unsigned int i=0; i<3; i++) {
   tabVecteurPile[i].x = tabVecteurTas[i].x = ((rand() % 201)/10.0)-10.0;
   tabVecteurPile[i].y = tabVecteurTas[i].y = ((rand() % 201)/10.0)-10.0;
   tabVecteurPile[i].z = tabVecteurTas[i].z = ((rand() % 201)/10.0)-10.0;
}

cout << "taille de la variable tabVecteurPile : " << sizeof(tabVecteurPile) << " octets." << endl;
cout << "taille de la variable tabVecteurTas : " << sizeof(tabVecteurTas) << " octets." << endl;

cout << "adresse du premier element de tabVecteurPile : " << (long int) &tabVecteurPile << endl;
cout << "adresse du premier element de tabVecteurTas : " << (long int) tabVecteurTas << endl;

cout << "adresse des champs x,y,z du deuxieme element de tabVecteurPile : " << (long int)
&(tabVecteurPile[1].x) << " " << (long int) &(tabVecteurPile[1].y) << " " << (long int)
&(tabVecteurPile[1].z) << endl;
cout << "adresse des champs x,y,z du deuxieme element de tabVecteurTas : " << (long int)
&(tabVecteurTas[1].x) << " " << (long int) &(tabVecteurTas[1].y) << " " << (long int)
&(tabVecteurTas[1].z) << endl;

cout << "valeur de *(tabVecteurPile+2) : "; Vecteur3DAfficher(*(tabVecteurPile+2)); cout << endl;
cout << "valeur de *(tabVecteurTas+2) : "; Vecteur3DAfficher(*(tabVecteurTas+2)); cout << endl;

tabVecteurPile[0].x = &(tabVecteurTas[1].y) - &(tabVecteurTas[2].z);
cout << "valeur du champ x du premier element de tabVecteurPile (sans cast long int) : " <<
tabVecteurPile[0].x << endl;
tabVecteurPile[0].x = (long int)(&(tabVecteurTas[1].y)) - (long int)(&(tabVecteurTas[2].z));
cout << "valeur du champ x du premier element de tabVecteurPile (avec cast long int) : " <<
tabVecteurPile[0].x << endl;
```

---

```
Trace écran attendue :

taille d'un Vecteur3D : 12 octets.
taille d'un pointeur sur Vecteur3D : 8 octets.
taille de la variable tabVecteurPile : 36 octets.
taille de la variable tabVecteurTas : 8 octets.
adresse du premier element de tabVecteurPile : 140735031870112
adresse du premier element de tabVecteurTas : 37118656
adresse des champs x,y,z du deuxieme element de tabVecteurPile : 140735031870124 140735031870128
140735031870132
adresse des champs x,y,z du deuxieme element de tabVecteurTas : 37118668 37118672 37118676
valeur de *(tabVecteurPile+2) : (-3.3,7.5,-6.2)
valeur de *(tabVecteurTas+2) : (-3.3,7.5,-6.2)
valeur du champ x du premier element de tabVecteurPile (sans cast long int) : -4
valeur du champ x du premier element de tabVecteurPile (avec cast long int) : -16
```

Dans la version sans la conversion en long int, l'arithmétique des pointeurs est utilisée, il y a bien une différence de 4 éléments (float) entre [1].y et [2].z.
Dans la version avec la conversion, les calculs (soustraction ici) sont faits directement sur les valeurs des adresses comme étant des nombres (des long int), et il y a bien une différence de 32 octets (4 floats de 8 octets).

## Exercice 6 : Pointeurs et références

Soient les trois instructions :

```
int x = 1;
int & rx = x;
int * px = &x;
```

a. Afficher la valeur de x, la valeur de rx et la valeur de px.
b. Afficher l'adresse mémoire de la variable x, celle de rx et celle de px. Expliquer ces valeurs.
c. Ecrire trois procédures `procedureAvecPointeur (int * ptr)`, `procedureAvecReference (int & rf)` et `procedureAvecInt (int val)` qui affichent la valeur et l'adresse du paramètre.
d. Appeler ces trois procédures avec comme paramètre px pour la première, x puis rx pour la deuxième et la troisième. Expliquer les résultats affichés.

```
void procedureAvecPointeur (int * ptr) {
    cout << "Dans procedureAvecPointeur: ptr = " << (long int) ptr << " et @ptr = " << (long int) &ptr <<
" et *ptr = " << *ptr << endl;
}

void procedureAvecReference (int & rf) {
    cout << "Dans procedureAvecReference: rf = " << rf << " et @rf = " << (long int) &rf << endl;
}

void procedureAvecInt (int val) {
    cout << "Dans procedureAvecInt : val = " << val << " et @val = " << (long int) &val << endl;
}

int main ()  {
   int x = 1;
   int &rx = x;
   int *px = &x;

   cout << "x = " << x << " et rx = " << rx << " et px = " << (long int) px << endl;
   cout << "@x = " << (long int) &x << " et @rx = " << (long int) &rx << " et @px = " << (long int) &px <<
endl;

   procedureAvecPointeur(px);
   procedureAvecReference(x);
   procedureAvecReference(rx);
   procedureAvecInt(x);
   procedureAvecInt(rx);

   return 0;
}
```

---

```
Trace écran attendue :

x = 1 et rx = 1 et px = 68702702540
@x = 68702702540 et @rx = 68702702540 et @px = 68702702544
Dans procedureAvecPointeur: ptr = 68702702540 et @ptr = 68702702504 et *ptr = 1
Dans procedureAvecReference: rf = 1 et @rf = 68702702540
Dans procedureAvecReference: rf = 1 et @rf = 68702702540
Dans procedureAvecInt : val = 1 et @val = 68702702508
Dans procedureAvecInt : val = 1 et @val = 68702702508
```

@x et @rx ont la même valeur puisqu'une référence est la même chose que la valeur référée. px est stocké à une adresse mémoire différente puisque c'est un pointeur. Dans procedureAvecPointeur, ptr pointe sur l'adresse de x mais le paramètre ptr lui-même est local à la procédure (i.e. une copie de l'adresse mémoire de x, de px dans ptr, est faite à l'appel). Dans procedureAvecReference, rf est x (même adresse sur la pile du main), et non un paramètre local (et donc passer x ou rx à l'appel est strictement identique). Dans procedureAvecInt, le paramètre val est local à la procédure, une copie de la valeur (ici 1) est faite à l'appel (et donc passer x ou rx à l'appel est aussi strictement identique).