Information, Calcul et Communication

Composante Pratique: Programmation C++

Semaine 12: consolidation et analyse

Pointeur et chaine-à-la-C: quelques precisions sur main()

Projet: outils et analyse de structures de données C et C++

Pointeur et chaine-à-la-C: la vérité sur main()

```
int main(int argc, char* argv[])
```

Il est possible d'utiliser une forme de la fonction **main()** avec deux paramètres formels:

- Un compteur entier argc valant au moins 1.
- Un **tableau** argv de argc **pointeurs** vers des **chaînes à-la-C**

C'est le mécanisme qui permet de fournir un nombre variable de paramètres aux commandes Linux.

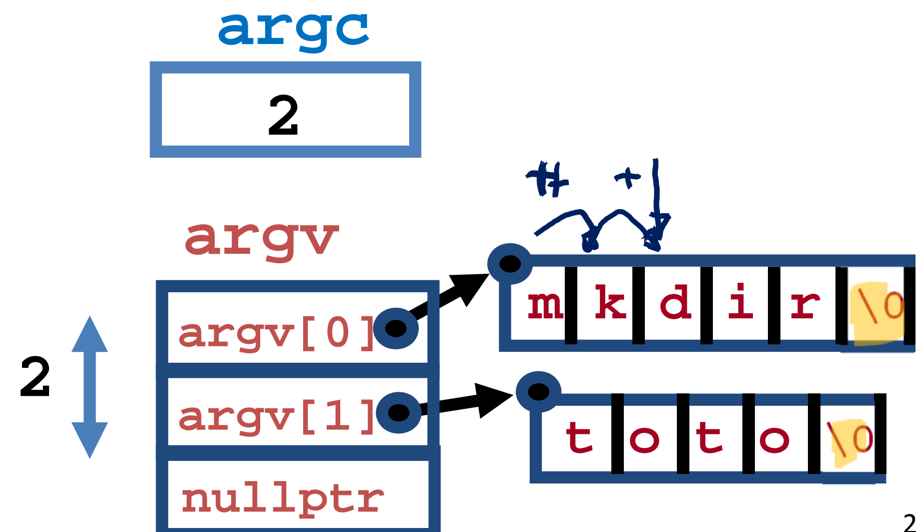
Ex: **mkdir toto**

main() de la commande **mkdir** reçoit **2** pointeurs vers les chaînes à-la-C **mkdir** et **toto**

Comment utiliser argv ?

Chaque élément **argv[i]** du tableau **argv** pour **i** compris entre **0** et **argc-1** est un pointeur qui contient l'adresse d'une chaîne à-la-C :

argv[0] pointe vers le nom de l'exécutable



Outil 1: l'opérateur sizeof(type *ou* expr)

```
sizeof(int) sizeof(char*)  sizeof(argv[0])
```

L'opérateur **sizeof** évalue, au moment de la compilation, **l'espace mémoire** occupé par un élément d'un **type** donné ou par une **variable** ou par le résultat d'une **expression**

Le résultat est exprimé en **octet**.

Exemples (sur machine 64 bits):

sizeof(char)	renvoie	**1**
sizeof(bool)	renvoie	1
sizeof(char*) (adresse)	renvoie	8
sizeof(double)	renvoie	8
sizeof(argv[0]) (char*)	renvoie	8

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

sizeof1.cc

Outil 2: les opérateurs de conversion explicite (cast)

```
(type)    static_cast<type>
```

L'opérateur de conversion permet de forcer un changement de type pour l'évaluation d'une expression. Il permet de contourner des cas de conversions automatiques indésirables. Par exemple, qu'est-ce qui est affiché ?

```
int a(5), b(6);
double x(3/4);
cout << x  << " " << a/b << endl;
```

x(3./4) -> x(0.83..)

L'opérateur de **conversion à-la-C** indique le type désiré entre parenthèses: **(type)**
En C++ on préfère l'écrire sous la forme plus visible **static_cast<type>**
Sa priorité est supérieure aux opérateurs arithmétiques

```
int a(5), b(6);
cout << (double) a/b << static_cast<double> (a)/b << endl;
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

cast_prog.cc

Outil 2: les opérateurs de conversion explicite (cast)

```
(type)    reinterpret_cast<type>
```

L'opérateur de conversion à-la-C est utilisé dans une grande variété de contextes, comme par exemple cette conversion de pointeur dans main() :

```
cout << argv[0]  << " " << (int*)argv[0] << endl;
```

En C++, pour une meilleure clarté du code, on préfère utiliser **reinterpret_cast<type>** pour effectuer des conversions de type sur un pointeur. Pour le même exemple:

```
cout << argv[0]  << " "
     << reinterpret_cast<int*> (argv[0]) << endl;
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

cast_prog.cc

Mieux comprendre `vector` à l'aide de `sizeof()` et du cast

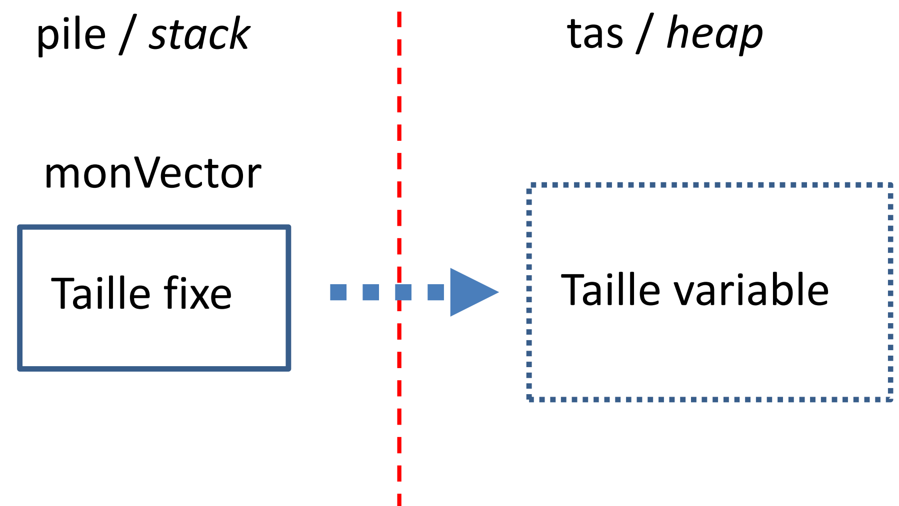
Lorsqu'un vector est déclaré localement, un espace mémoire minimum est réservé sur la pile, même si ce vector est vide. sizeof() peut nous indiquer cette taille.

```
vector<double> monVector;
cout << sizeof(monVector) << endl;
```

remarque: **sizeof()** fournit toujours la même taille quel que soit l'espace réservé initialement pour **monVector**, ceci même si monVector change ensuite dynamiquement de taille.

Comment est-ce possible ?

→ L'espace mémoire de monVector se partage entre la **pile**/stack (taille fixe) et le **tas**/heap (taille nulle pour un vector vide et qui change par allocation/libération dynamique).



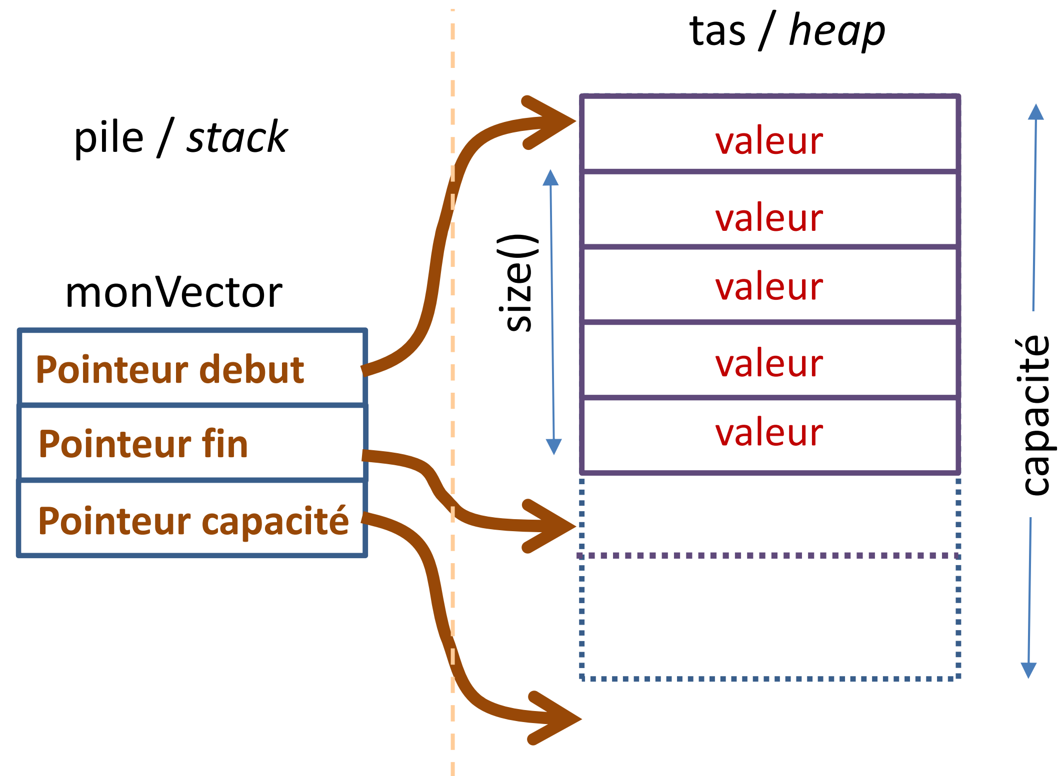
sizeof_vector.cc

Comment un `vector` est-il mis en oeuvre ?

En fait le standard du C++ ne documente pas *comment* un vecor est réalisé. Il documente seulement son *interface* = l'ensemble des tâches qu'il doit réaliser correctement.

De nombreux ouvrages suggèrent qu'il lui suffit de *3 pointeurs* pour réaliser efficacement ses objectifs. Ces pointeurs pointent respectivement vers:

- Le **début des données**
- Juste après la **fin des données**
- Juste après la fin de la **capacité** réservée

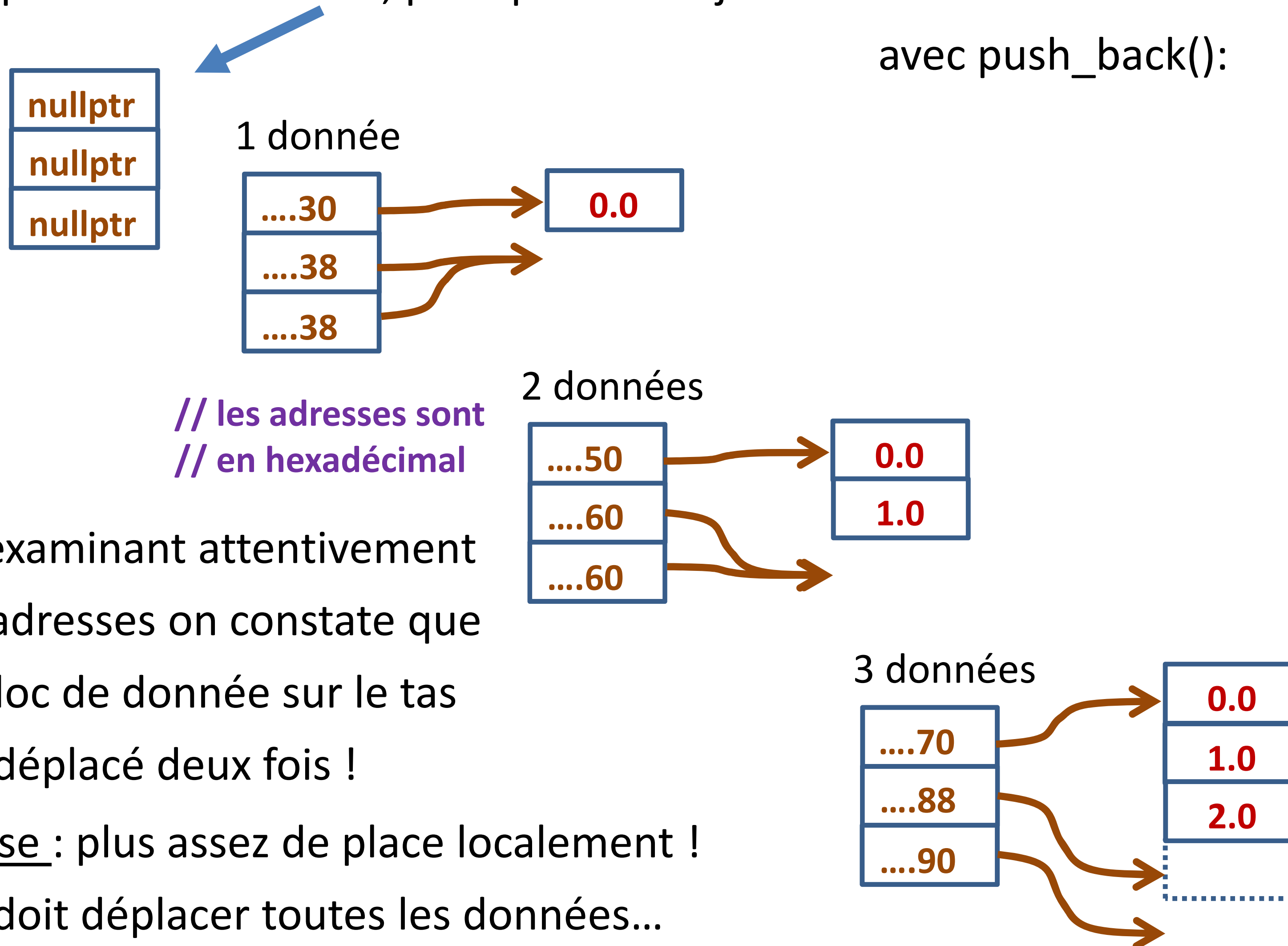


Nous pouvons le vérifier à l'aide de l'opérateur reinterpret_cast.

adresse_vector.cc

Comment un `vector` est-il mis en oeuvre ? (2)

Exemple d'exécution du code adresse_vector.cc affichant les 3 pointeurs pour un vector vide, puis quand on ajoute une donnée à la fois avec push_back():



En examinant attentivement les adresses on constate que le bloc de donnée sur le tas est déplacé deux fois !

Cause : plus assez de place localement !

On doit déplacer toutes les données…

```
0x7ffc7c16aff0
0
0
0
0x153d030
0x153d038
0x153d038

0x153d050
0x153d060
0x153d060

0x153d070
0x153d088
0x153d090

0x153d070
0x153d090
0x153d090

0x153d0a0
0x153d0c8
0x153d0e0

0x153d0a0
0x153d0d0
0x153d0e0

0x153d0a0
0x153d0d8
0x153d0e0

0x153d0a0
0x153d0e0
0x153d0e0

0x153d0f0
0x153d138
0x153d170

0x153d0f0
0x153d140
0x153d170
```

Comment optimiser les performances d'un `vector` ?

Le déplacement des données implique une **libération** et une **ré-allocation dynamique** de mémoire. Ces opérations ne sont pas gratuites en termes de coût calcul.

Par exemple, voici la comparaison de l'estimation (en ns) de quelques opérations en langage C sur un laptop avec un processeur CORE i7:

```
k = i+j                    2 ns
k = i*j                    2 ns
k = i/j                    3 ns
f = sqrt(i+j)              5 ns
f = sin(i+j)             103 ns
free(malloc(16))          26 ns
free(malloc(100))         26 ns
free(malloc(2000))        38 ns
```

// équivalent à la libération
// d'un espace alloué dyn.
// exprimé en nb octets

temps obtenus avec timemod.c de John Bentley (Programming Pearls 1999)

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

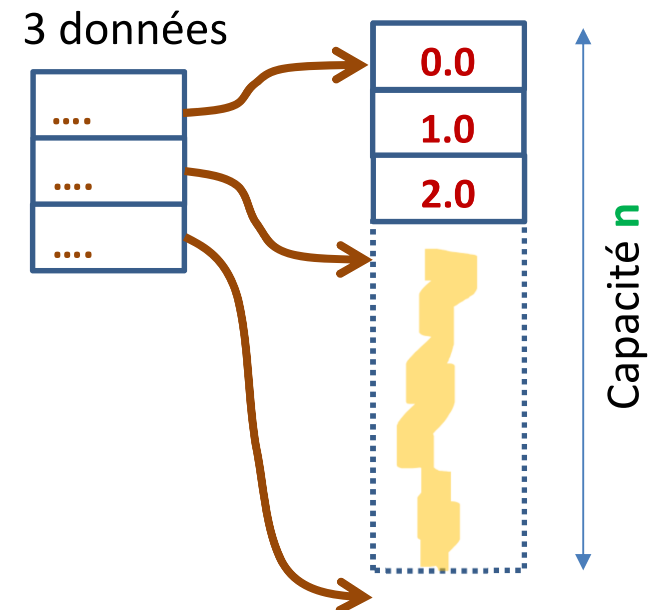
Comment optimiser les performances d'un `vector ? (2)`

1) Ne pas utiliser de vector si on connait la taille nécessaire au moment de l'écriture du programme ; utiliser array à la place

2) Déclarer le vector avec sa dimension dès qu'elle est connue à l'exécution puis travailler avec des indices sur l'espace réservé. Cela évite les push_back qui causent des ré-allocations.

3) Si la dimension du vector **v** peut évoluer en cours d'exécution, faire une demande de réservation d'une certaine capacité de **n** éléments avec **v.reserve(n)**.

Cela évite les réallocations tant que les push_back travaillent dans l'espaces de la capacité réservée.



4) Pour échanger les valeurs de deux vector **a** et **b**, appeler **a.swap(b)** car cela échange seulement les 3 pointeurs de la partie fixe sur la pile.

5) Passer un vector par référence constante aux fonctions

Performance liée à l'accès aux données en mémoire

Lien avec le cours ICC-théorie M3 L2: Hiérarchie de mémoires

Conseil valable pour toutes les structures de données et en particulier les tableaux à plusieurs indices (**vector**, **array**, à-la-C):

Traiter les données **ligne par ligne** pour bénéficier du très faible coût de l'accès séquentiel aux données consécutives en mémoires (***localité spatiale*** = forte probabilité d'être dans un même bloc). Cela diminue les défauts de cache mémoire. Illustration avec le programme :

perf_som_mat_ligne_par_ligne_puis_col_par_col.cc

Outil de mesure temps d'exécution d'une partie d'un programme:
(**série6** moodle): fonction **clock()** avec **#include <ctime>**

Outil de mesure temps d'exécution d'un programme entier **prog:**
Sur la ligne de commande: **time prog**
Retenir le temps «user»

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Résumé

vector dispose d'une interface riche permettant de gérer des structures de données dynamiques en minimisant les risques d'erreurs.

Cette interface est robuste = elle n'expose pas des éléments de type pointeurs qui pourraient être la cause de nombreux bugs.

Si la performance est un critère important pour le fonctionnement du programme:

→ **array** pour toutes les données dont on connait la taille a priori

→ Utiliser le passage par référence, reserve(), swap()…

→ Tirer parti de la *localité spatiale* des données