# Information, Calcul et Communication

## Composante Pratique: Programmation C++

## Semaine 12: consolidation et analyse

Outils et analyse de structures de données: le cas de **vector**

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

sizeof1.cc

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

```
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*)** renvoie **8**

**sizeof(double)** renvoie **8**

**sizeof(argv[0])** renvoie **8**

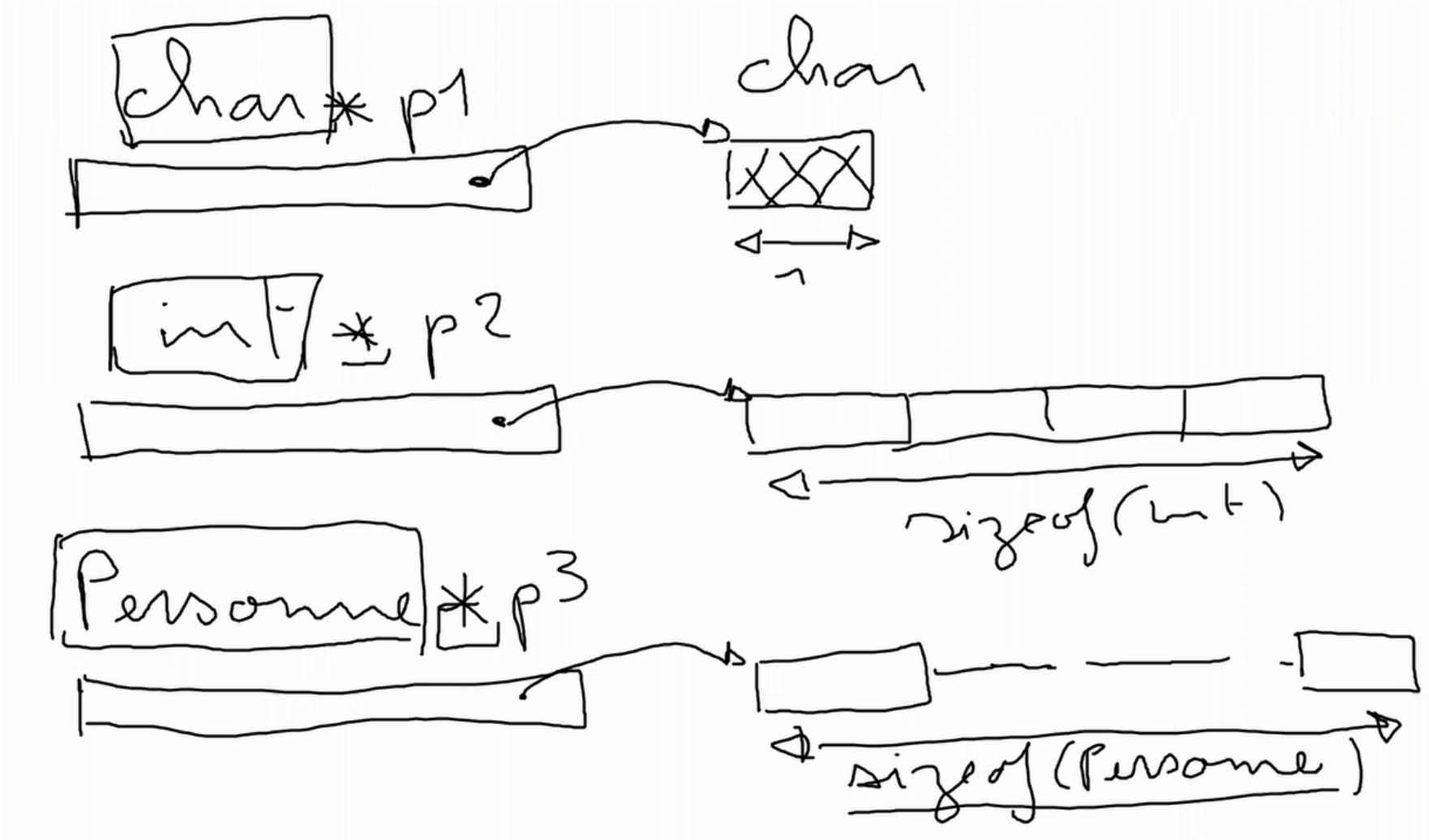
# SpeakUp: Which of the 4 letters A, B, C or D is displayed by the program sizeofPointer.cc ?

A. A

B. B

C. C

D. D



# 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;
```

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;
```

# 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;
```

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

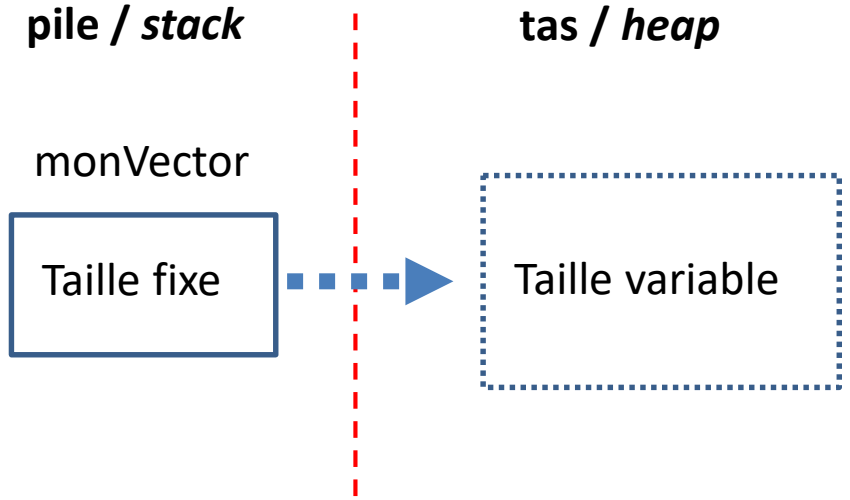
Lorsqu'un vector est déclaré localement, un *espace mémoire minimum* est réservé **sur la pile**, même si ce vector est vide. **`sizeof()`** peut nous indiquer cette taille.

```
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).

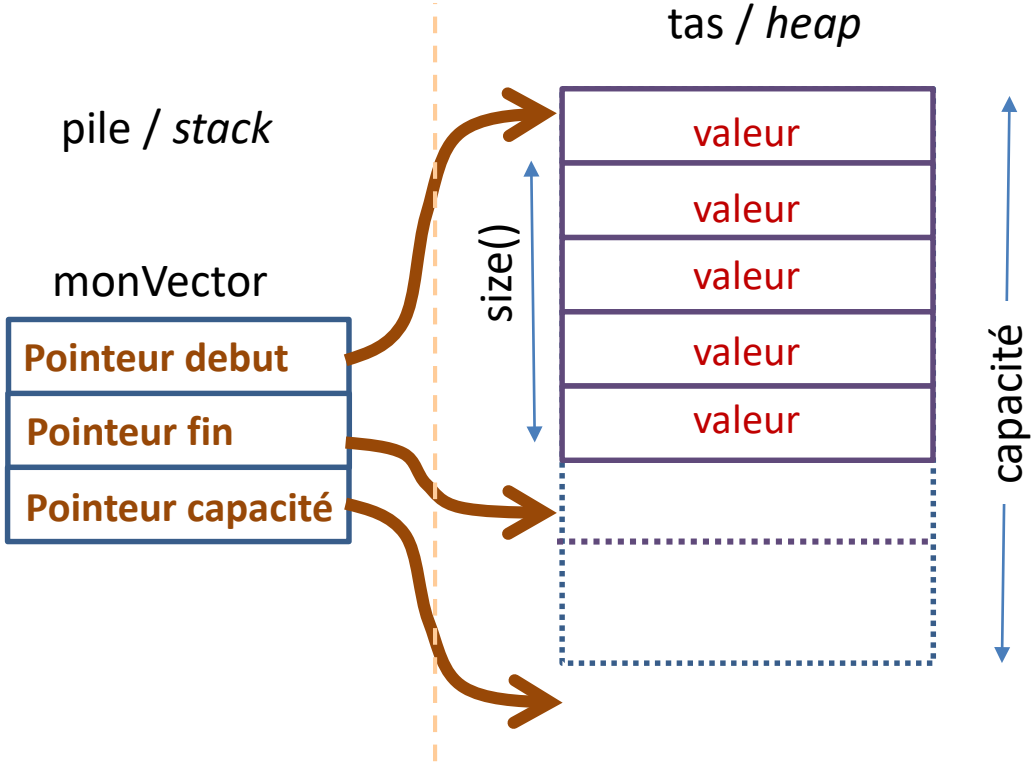


# Comment un `vector` est-il mis en oeuvre ?

En fait le standard du C++ ne documente pas *comment* un vector 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:

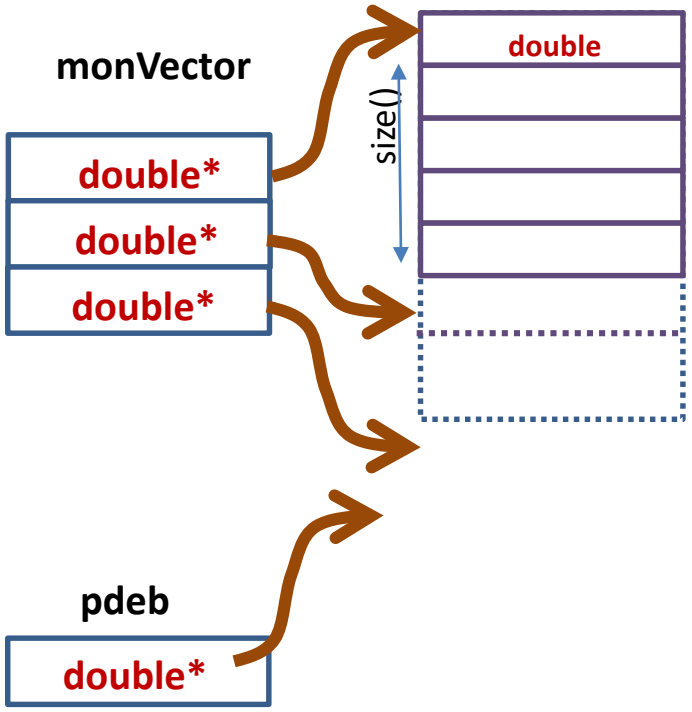
- Le **début des données**
- Juste après la **fin des données**
- Juste après la fin de la **capacité** réservée

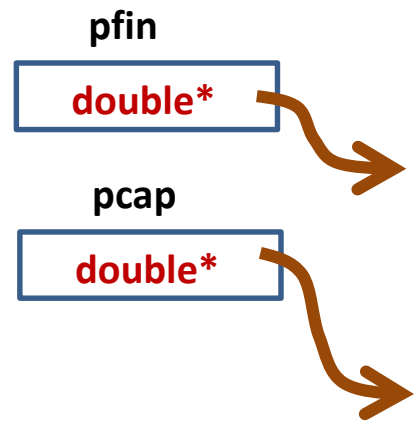



Nous pouvons le vérifier à l'aide de l'opérateur reinterpret_cast.

adresse_vector.cc

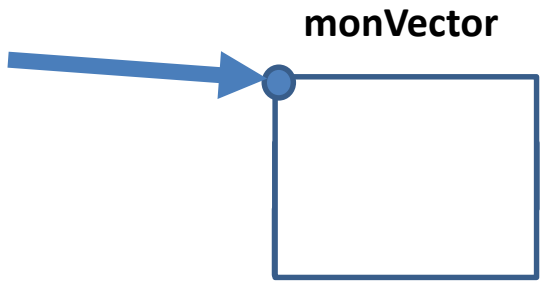
# Types des variables et expressions utilisées dans le programme adresse_vector.cc



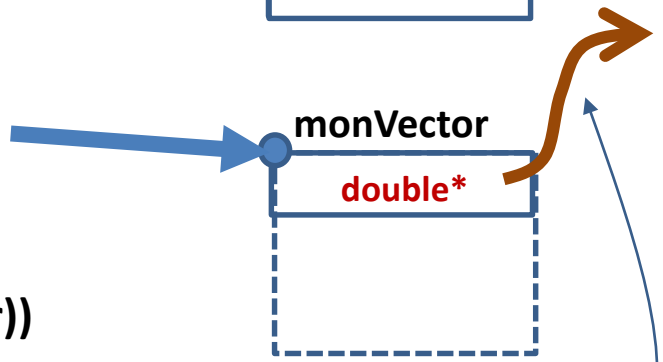

**&monVector**

*est une adresse de type* **vector<double>***




**reinterpret_cast<double**>(&monVector)**

*est une adresse de type* **double****

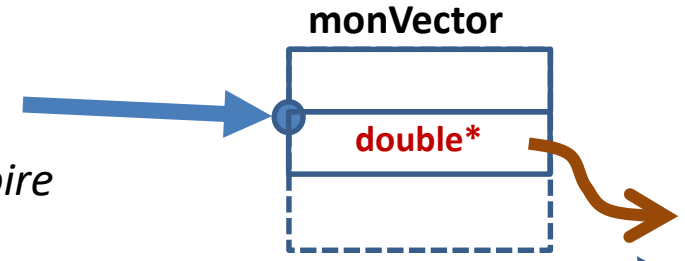



***(reinterpret_cast<double**>(&monVector))**

*est la valeur rangée à cette adresse*

**reinterpret_cast<double**>(&monVector) +1**

*est l'adresse de type* **double**** *qui suit en mémoire*

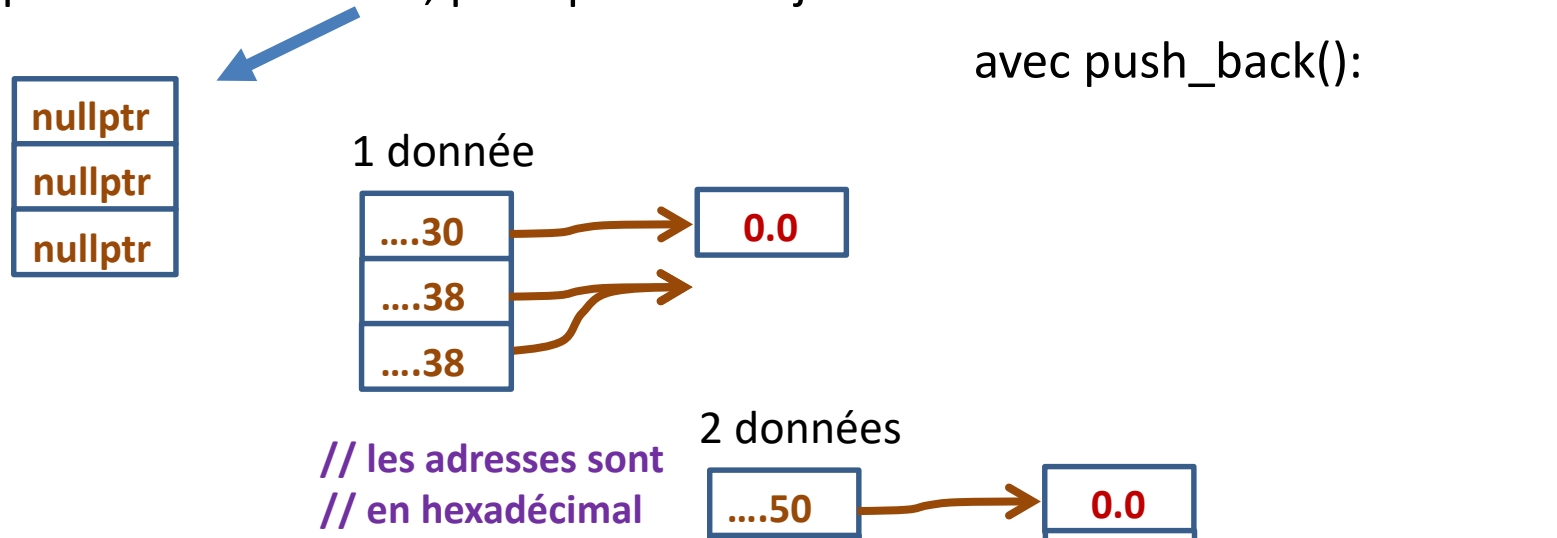



***(reinterpret_cast<double**>(&monVector) +1)**

*est la valeur rangée à cette adresse*

# 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
```

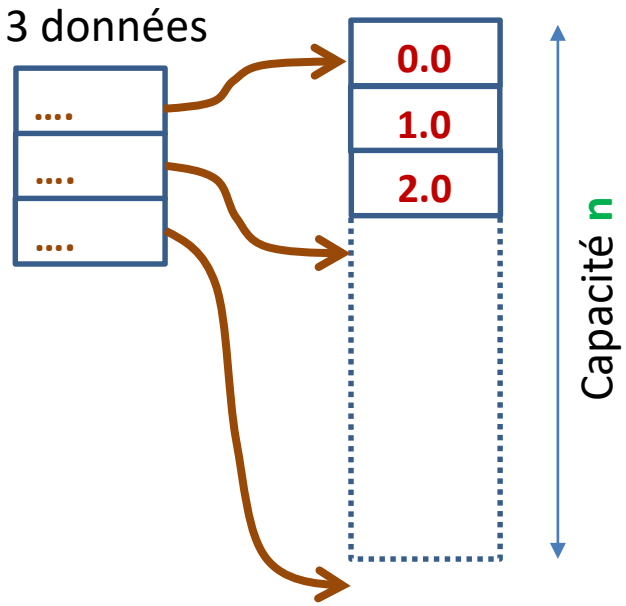
# Comment optimiser les performances d'un `vector` ? (2)

1) Ne pas utiliser de vector **si on connait la taille nécessaire au moment de l'écriture du programme** et que **cette taille reste invariable pendant l'exécution** ; => utiliser **array** à la place de vector

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 centrale

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.

**Outil de mesure** temps d'exécution d'un programme entier **prog:**

Sur la ligne de commande: **time prog**

Retenir le temps «user»

# 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