Méthodes informatiques pour physiciens

introduction à C++ et
résolution de problèmes de physique par ordinateur

Leçon # 6 : Les pointeurs

Alessandro Bravar Alessandro.Bravar@unige.ch
tél.: 96210 bureau: EP 206

assistants

Johanna Gramling Johanna.Gramling@unige.ch
tél.: 96368 bureau: EP 202A

Mark Rayner Mark.Rayner@unige.ch
tél.: 96263 bureau: EP 219

http://dpnc.unige.ch/~bravar/C++2015/L6
pour les notes du cours, les exemples, les corrigés, …

Plan du jour #6

Récapitulatif et corrigé de la leçon #5

Adresse mémoire

texte Micheloud et Rieder
chap. 8 et 12

Les pointeurs

Déclaration et utilisation des pointeurs

Tableaux et pointeurs

Arithmétique des pointeurs

Fonctions et pointeurs

Allocation dynamique de la mémoire

Récapitulatif de la leçon #5

Intégration numérique

- Formule du rectangle
- Formule du trapèze
- Formule de Simpson

Modélisation

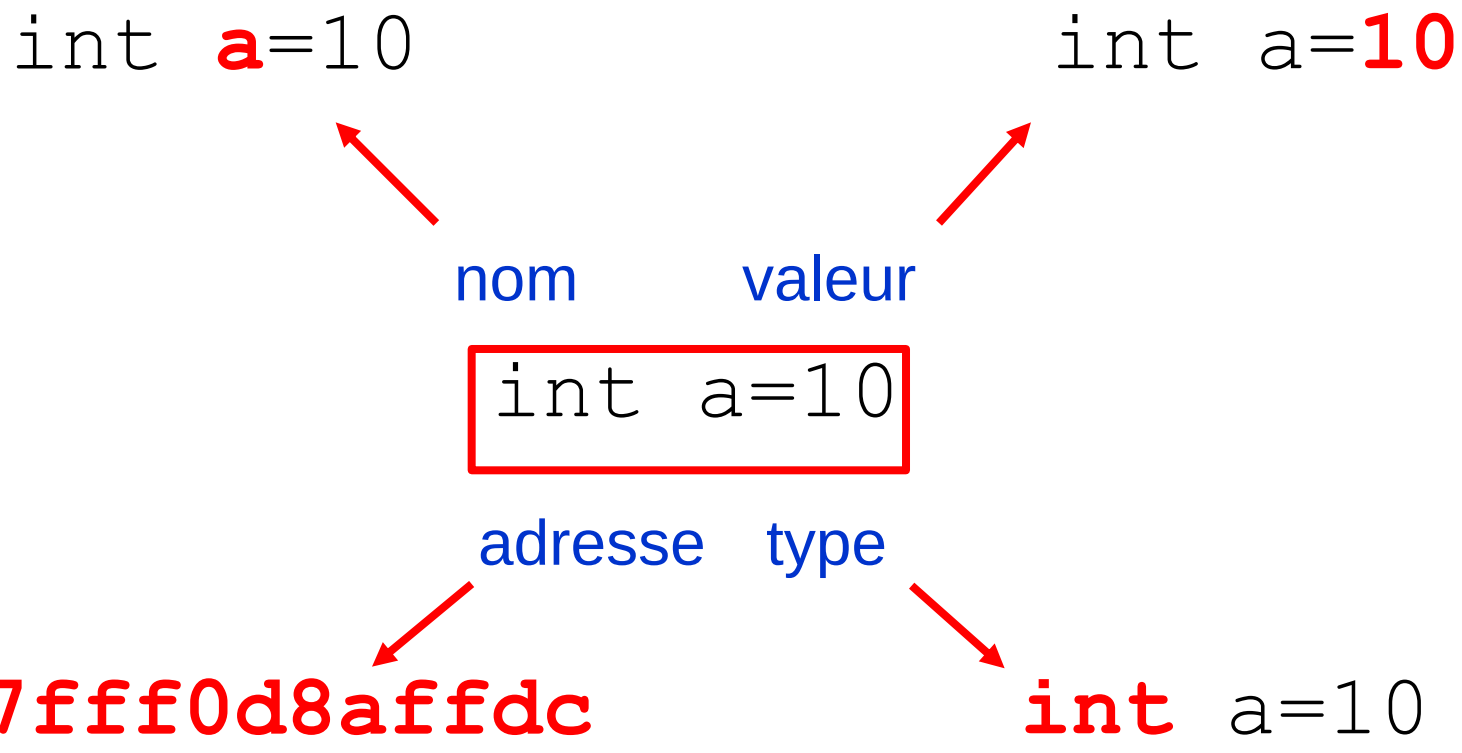
* Différentiation numérique

- différence finie
- différence finie centrale
- dérivée première et seconde

Variable et ses attributes

Lors de sa déclaration, quatre attribuées sont associés à une variable :

- nom
- type
- adresse mémoire
- si initialisée, sa valeur (si pas initialisée, aussi une valeur non définie !)



p. ex. : `cout << &a << endl;`

attention : l'adresse mémoire peut changer
chaque fois que vous lancez le programme

Memoire d'un ordinateur

La mémoire d'un ordinateur est divisée en
emplacements numérotés séquentiellement
et pourrait être représentée comme un très long tableau constitué d'octets.

Par exemple :

Ordinateur avec mémoire vive (RAM) de 1 GBy
→ tableau de 2^{30} octets

Indice de chaque octet : adresse mémoire
formée par 4 (8) octets sur ordinateur à 32 (64) bit
en format hexadecimal (léçon 7) de `0x00000000` à `0x3fffffff`.

Pour accéder à l'adresse mémoire utilisez l'opérateur d'adresse `&`.

64 bit

```
cout << &a << endl;
```

0x7fff0d8affdc

32 bit

```
cout << &a << endl;
```

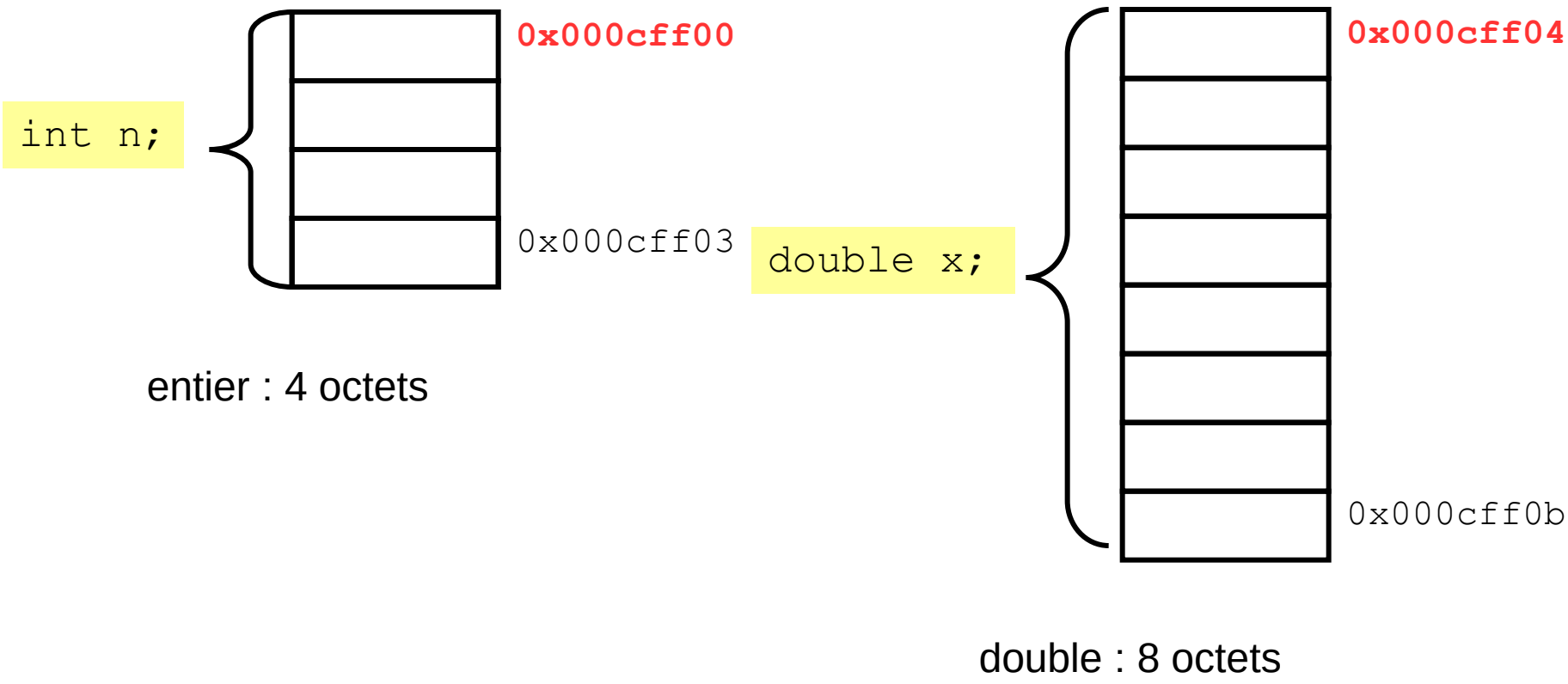
0x10454ac8

Combien de bits a votre ordinateur ? Affichez l'adresse mémoire d'une variable...

voir Adresse.cpp

Adresse mémoire

L'adresse mémoire est
l'**adresse du premier octet** du bloc de mémoire où l'objet (variable) est stocké
(et il change d'ordinateur à ordinateur / d'exécution en exécution du programme)



entier : 4 octets

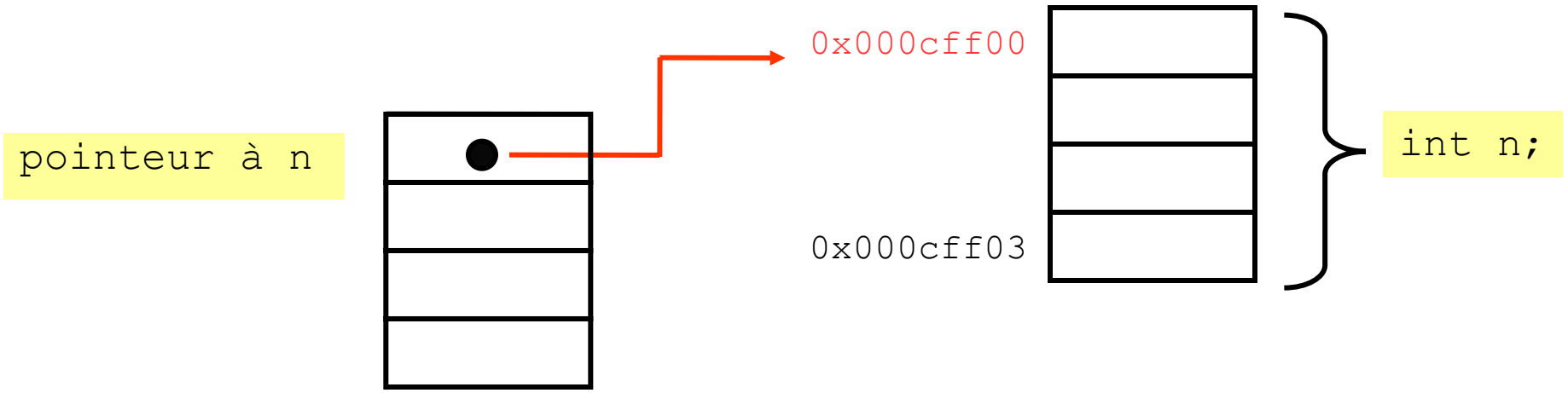
double : 8 octets

En général, connaître la valeur exacte de l'adresse mémoire n'est pas très intéressant, par contre nous voulons accéder à l'adresse pour **utiliser** la variable en question.

Pointeurs

On peux accéder à l'adresse mémoire directement (et à l'objet ou variable) avec un pointeur

un pointeur est une variable (du type entier) qui contient
l' adresse mémoire d'une autre variable



les pointeurs sont des variables typées :
ils sont liés au type de la variable sur laquelle ils pointent.

Pointeurs

déclaration d'une variable :

```
int n;
```

déclaration d'un pointeur :

```
int *p_n;
```

affectation d'un pointeur :

```
p_n = &n;
```

L'opérateur de **référence** `&` renvoie **l'adresse** de la variable à laquelle il s'applique.

déréférencement d'un pointeur :

```
int nCopy = *p_n;
```

(pour accéder à la variable)

L'opérateur de **déréférencement** `*` renvoie la **valeur** de la variable

```
int main() {
  int j = 4;
  int *ptr = 0;      //declaration de pointeur
  ptr = &j;    //l'adresse de j est stockee dans p

  int k = *ptr;    //la valeur de j est affectee a k
  cout << k << "\t" << *ptr << endl;

  *ptr = 5;      //la valeur 5 est affectee a j
  cout << *ptr << "\t"  << j << endl;
  return 0; }
```

voir Adresse.cpp

Les règles (et les dangers) des pointeurs

Il faut toujours initialiser le pointeur lors de la déclaration :
à 0 ou NULL si l'on n'affecte pas l'adresse d'une variable

```
int *ptr = 0;
//ou
int *ptr = NULL;
```

Ces deux instructions généreront un erreur :

```
int *q;
cout << *q << endl;    //erreur !!
```

le pointer `q` est déclaré, mais il **n'est pas initialisé** → il ne pointe sur rien !

Il faut toujours vérifier si le pointeur est valide (pas NULL) avant son utilisation

```
if (ptr == 0) {
  //ne faire rien avec ce pointeur !
}
```

Si vous essayez d'accéder ou de déréférencier un pointeur nul pendant l'exécution du programme cela provoquera un *erreur fatale courante* (run time error).
→ le compilateur ne vérifie pas si vous essayez d'accéder à un pointeur nul !

Cet instruction, par contre, générera une erreur pendant la compilation

```
ptr = 5;    //erreur !
```

on ne peut pas simplement affecter des adresses mémoire !

Arithmétique des pointeurs

Les seuls opérations admises sur les pointeurs eux-mêmes sont :

1. les affectations,
2. les comparaisons (comparaisons d'adresses mémoire),
3. l'addition (ou la soustraction) des constantes.

Ce dernière opération tient compte du type d'objet pointé :
le pointeur pointe à l'adresse mémoire N octets après l'adresse courante
avec N = nombre of octets pour le type de la variable

```
int *pti = &n;    //pointeur sur un entier
double *ptd = &x;    //pointeur sur un double

pti++;    //l'adresse pti est incrementee de 4 (int = 4 octets)
ptd++;    //l'adresse ptd est incrementee de 8 (double = 8 octets)
```

Après l'incrément, nous ne connaissons plus
le contenu à l'adresse mémoire du pointeur
→ dangereux d'additionner/soustraire à un pointeur
une constante en dehors d'un tableau !

Pointeurs et constantes

Le mot-clé `const` peut modifier la déclaration d'un pointeur :

déclaration d'un pointeur constant

```
int *const ptr1 = &x;
```

`ptr1` ne peut pas changer de valeur
(i.e. l'adresse mémoire stocké dans `ptr1`)

déclaration d'un pointeur sur une constante

```
int const *ptr2 = &C;
```

`ptr2` point sur une variable constante
`ptr2` peut changer de valeur, mais pas `C`

déclaration d'un pointeur constant sur une constante

```
const int *const ptr3 = &C;
```

`ptr3` ne peut pas changer de valeur
`C` ne peut pas changer de valeur

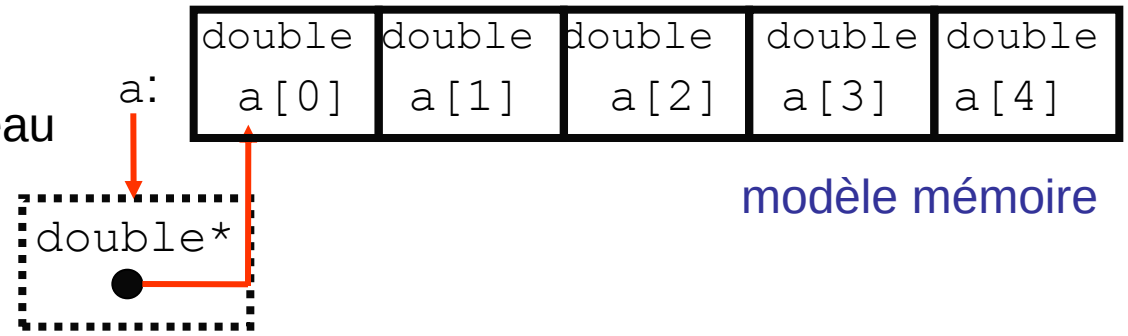
Pointeurs et Tableaux

Déclaration d'un tableau :

```
double a[5];
```

`a` est le nom du tableau et contient l'adresse mémoire du premier élément du tableau `a`

→ `a` est aussi un pointeur au premier élément du tableau



Comparez : `cout << *a;` et `cout << a[0];`

Comparez : `cout << a;` et `cout << &a[0];` (adresse du tableau)

On peux aussi parcourir le tableau avec des pointeurs :

```
double a[5];
   double *y = a;  (ou double *y = &a[0];)
   y pointe à a[0]      a[0] == *y
   y+1 pointe à a[1]    a[1] == *(y+1)
   y+2 pointe à a[2]    a[2] == *(y+2
```

voir PointeursEtTableaux.cpp

Qu'affichera t'il ? Et pourquoi ?

```
double a = 5.;
cout << a;    ?
cout << &a;    ?

double b = 7.;
double &c = b;
cout << c;    ?

int i = 4;
int *k = &i;
int l = *k;
i++;
cout << k;    ?
cout << *k;    ?
cout << l;    ?

int *n;
cout << n;    ?
cout << *n;    ?
```

```
int x[5]={1,2,3,4,5};
cout << x;    ?
cout << &x[0];    ?
cout << x[1];    ?

double x[5]={1,2,3,4,5};
double *y = x;    ?
double *z = &x[0];    ?
cout << *y;    ?
cout << y;    ?

y = y + 3;
cout << *y;    ?
cout << y;    ?

y = y - 1;
cout << *y;    ?
cout << y;    ?
```

Exemples

somme des éléments d'un tableau :
le tableau est parcouru avec un indice

```
double x[5];    //remplissez x !

double sum = 0.;
for (int i=0; i<5; i++) {
    sum = sum + x[i];
}
```

même somme avec un pointeur :
le tableau est parcouru avec un pointeur

```
double x[5];
double *y = x; //ou &x[0]
double sum = 0.;
for (int i=0; i<5; i++) {
   sum = sum + *y;
   y++; }
```

inversion de l'ordre des éléments dans un tableau
on utilise deux pointeurs :
- le premier pour parcourir le tableau en sens direct
- le deuxième pour parcourir le tableau en sens inverse

```
double x[10];
double *left = &x[0];    //adresse premier element
double *right = &x[9];    //adresse dernier element
while (left < right) {
  double temp = *left;    //variable temporaire
  *left = *right;    //ou *left++ = *right;
  left++;
  *right = temp;    //ou *right-- = temp;
  right--; }
```

voir Inversion.cpp

Passage des arguments aux fonctions

Nous avons vu que les fonctions ont 2 limitations de base :

1. les arguments sont passés par valeur (non modifiables) et
2. l'instruction `return` ne peut retourner qu'une seule valeur.

Pour remédier à ces limitations, on peut également passer les arguments par *référence* en utilisant les pointeurs ou les références :

1) valeur :

```
void swap(int a, int b);
swap(a, b);
```

N.B. : on passe les valeurs de a et de b !

2) référence :

```
void swap(int &a, int &b);
swap(a, b);
```

N.B. : on passe les variables a et de b !

3) pointeur :

```
void swap(int *a, int *b);
swap(&a, &b);
```

N.B. : on passe l'adresse mémoire de a et de b !

voir Swap1.cpp, Swap2.cpp, Swap3.cpp

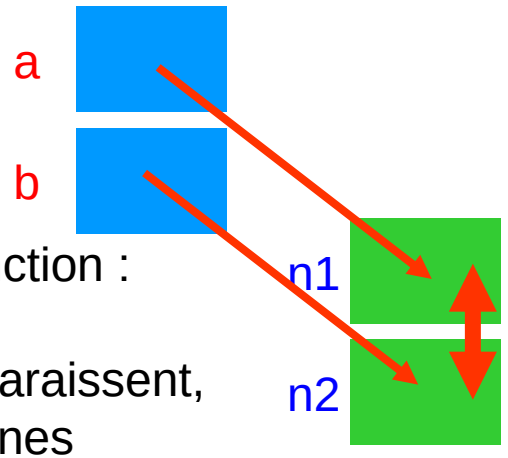
Le type renvoyé par la fonction peut aussi être un pointeur ou une référence :
`int &fonct1()` ou `int *fonct2()`

Passage par valeur

Ce programme échange deux valeurs dans la fonction swap.

```
#include <iostream>

using namespace std;

void swap(int n1, int n2);

int main() {
  int a = 10;
  int b = 20;
  cout << "avant swap : " << a << "  " << b << endl;
  swap(a, b);
  cout << "apres swap : " << a << "  " << b << endl;
  return 0;
}

void swap(int n1, int n2) {
  int temp = n1;
  n1 = n2;
  n2 = temp;
  cout << "dans swap : " << n1 << "  " << n2 << endl;
  return;
}
```

On passe des valeurs à la fonction (pas les variables).
Les arguments passés à la fonction sont locaux à la fonction : une fois sorti de la fonction, les arguments n1 et n2 disparaissent, donc on retrouve les anciennes valeurs de a et de b.



a = 10 et b = 20 !
c'est correct ?

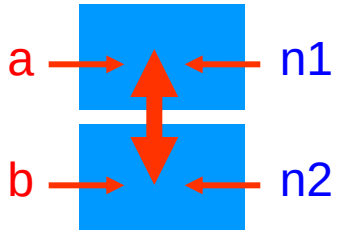
la fonction ne renvoie aucune valeur, elle est de type `void` et `return` n'est pas suivi par une valeur à renvoyer (dans ce cas, on peut omettre `return`)

voir Swap1.cpp

Passage par référence

Ce programme échange aussi deux valeurs dans la fonction `swap`, mais le variables sont passées à la fonction `swap` par référence.

```
#include <iostream>

using namespace std;

void swap(int &n1, int &n2);

int main() {
  int a = 10;
  int b = 20;
  cout << "avant swap : " << a << "  " << b << endl;
  swap(a, b);
  cout << "apres swap : " << a << "  " << b << endl;
  return 0;
}

void swap(int &n1, int &n2) {
  int temp = n1;
  n1 = n2;
  n2 = temp;
  cout << "dans swap : " << n1 << "  " << n2 << endl;
  return; }
```



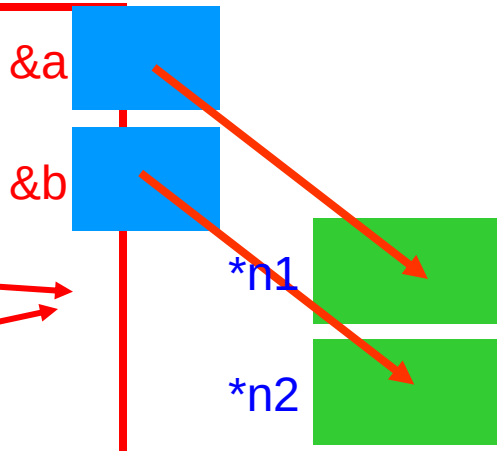
a = 20 et b = 10 ! c'est ça que vous vous attandez ?

voir Swap2.cpp

Passage par pointeur

Ce programme échange aussi deux valeurs dans la fonction `swap`, mais le variables sont passées à la fonction `swap` par pointeur.

```
#include <iostream>

using namespace std;

void swap(int *n1, int *n2);

int main() {
  int a = 10;
  int b = 20;
  cout << "avant swap : " << a << "  " << b << endl;
  swap(&a, &b);
  cout << "apres swap : " << a << "  " << b << endl;
  return 0;
}

void swap(int *n1, int *n2) {
  int temp = *n1;
  *n1 = *n2;
  *n2 = temp;
  cout << "dans swap : " << *n1 << "   " << *n2 << endl;
  return; }
```



adresse de a et b

a = 20 et b = 10 !
c'est ça que vous vous attandez ?

voir Swap3.cpp

Pointeurs, tableaux et fonctions

Lors du passage d'un tableau à la fonction,
c'est l'adresse mémoire du tableau qui est transmis
c'est-à-dire le pointeur sur le premier élément du tableau
(syntaxe : identique passage par valeur)

→ la taille du tableau n'est pas disponible dans la fonction
(il faut la passer avec un autre paramètre à la fonction)

déclaration de la fonction

```
double somme(double x[], int n);
```

appel de la fonction (dans `main`)

```
const int taille = 5;
double tableau[taille] = {1,2,3,4,5};
somme(tableau, taille);
```

Pour passer un tableau multidimensionnel,
il faut spécifier toutes les dimensions sauf la première, p. ex. :

déclaration de la fonction

```
void multParConst(double a[][5], . . .);
```

Pointeurs et tableaux multidimensionels

Alternativement

on peut passer un pointeur sur le tableau et spécifier toutes ses dimensions

déclaration de la fonction:

```
double somme(double *adrtab, int dim1, int dim2);
```

pointeur sur le tableau

appel de la fonction (dans main)

```
somme(&tab[0][0], dim1, dim2);
```

adresse mémoire du premier élément

Dans ce cas, il faut parcourir le tableau avec des pointeurs pour accéder aux valeurs du tableau.

définition de la fonction

pointeurs sur les *matrices*

dimensions des *matrices*

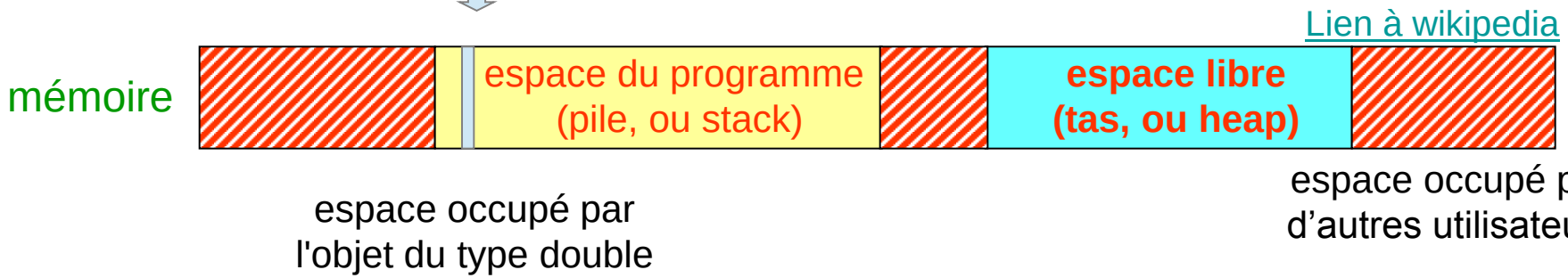
```
void somme(double *matA, double *matB, double *matC, int m, int n) {
  for (int i=0; i<m; i++)
    for (int j=0; j<n; j++) {
      *matC = *matA + *matB;   //dereferenciement des pointeurs
      matA++;   //on increment le pointeur de 8 octet
      matB++;   //on increment le pointeur de 8 octet
      matC++;   //on increment le pointeur de 8 octet
  }
}
```

voir PointeursEtMatrices.cpp

Allocation statique de la mémoire

Declaration d'un objet ⟶ **emplacement en mémoire** pour y stocker sa valeur

```
Double cinq = 5
```



. La **taille** de cet objet doit être **connue à la compilation**
et **ne peut pas changer** pendant l'exécution d'un programme
(p.ex. la dimension d'un tableau)

La memoire occupé par cet object va être libérée automatiquement
quand l'object va **dehors de sa portee**

Portée de **b**

```
{
int a = 10 ;
for (int i=0 ; i<10 ; i++) {
int b=a+i ;
cout << b << endl ;
}
```

Emplacement du memoire pour **b**

La memoire pour **b** est liberée

Allocation dynamique de la mémoire

Et si nous **ne savons pas** p. ex. la **taille de notre tableau avant la compilation ?**

L'allocation dynamique de la mémoire permet de créer des espaces dans la mémoire libre pour stocker des données pendant l'exécution d'un programme. Leur taille est déterminée pendant l'exécution.

```
cin >> n;
int *tab = new int[n];
```

L'espace réservé sur le tas ne va pas être libérée automatiquement !
On faut utiliser le mot-clé **delete** pour détruire la variable
et eviter les fuites de memoire

Pour chacun **new**,
il doit y avoir un **delete** !

```
delete[] tab;
```

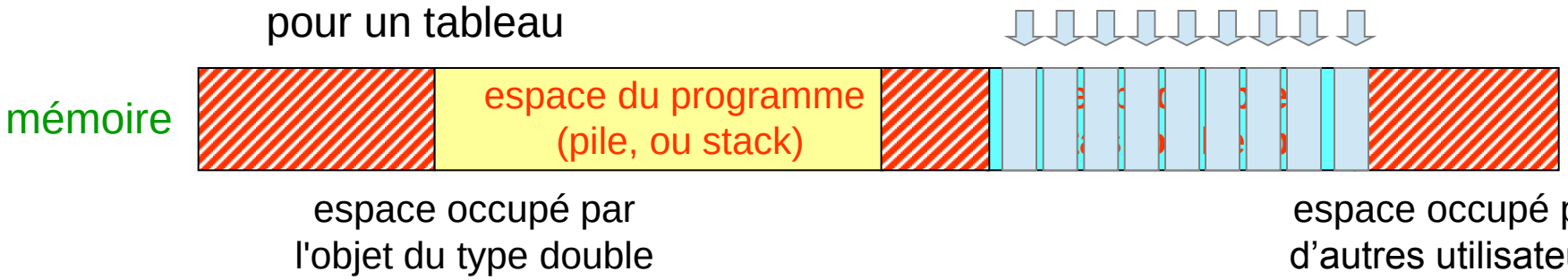
Fuites de memoire et saturation

Est-ce que ce program marche bien (ne l'essayez pas ...) ?

```
for (int i=0; i<100000000; i++ ) {
  int n = 1000000;
  double *tab = new double[n] ;
}
```

A chaque itération dans la boucle, on va allouer (beaucoup) d'espace pour un tableau

Oups ! Il n'y a plus de memoire dans le tas!



Il y a deux problèmes ici :
1. le pointeur tab est dehors sa portée chaque fois, mais la mémoire n'est pas libérée automatiquement → fuite de mémoire
2. si nous allouons trop de mémoire, la mémoire va être saturée et le pointeur sera initialisé à NULL

1. Pour chacun **new**, il doit y avoir un **delete**

2. Chacune fois qu'il y a une allocation dynamique, il faut vérifier le pointeur

Déclaration d'un tableau : résumé

Avant de déclarer un tableau, il faut connaître sa dimension. La dimension du tableau doit être connue avant l'exécution du programme (norme ANSI):

p.ex.:
```
const int N = 10;
double x[N];
```

La plupart des compilateurs récents ne requièrent pas la dimension avant l'exécution du programme:

p.ex.:
```
int n;    cin >> n;
double x[n];
```
interdit dans ce cours !

En réalité, il s'agit d'une allocation dynamique de la mémoire implicite.
Si vous imposez les normes ANSI à votre compilateur, la compilation échouera (p.ex. `g++ -ANSI test.cpp`).

Pour créer des objets pendant l'exécution du programme, on utilise l'allocation dynamique de la mémoire de manière explicite.

p.ex.:
```
int n;    cin >> n;
double *x = new double [n];
```
Pour vérifier si la mémoire à été allouée correctement, on teste le pointeur `x` :
```
if (x==0) exit(0);
```
Le tableau ainsi défini peut être utilisé comme un tableau ordinaire;
on utilise un indice pour parcourir le tableau,

Enfin, pour libérer la mémoire occupée par le tableau après son utilisation, on efface ce tableau: `delete[] x;`

Exemple

Dans cet exemple, la longueur du tableau est inconnue au début ; l'espace pour le tableau sera créé pendant l'exécution du programme. C'est un exemple d'*allocation dynamique* de la mémoire.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
  cout << "Combien d'elements ? \n";
  int n;    cin >> n;
  int *a = new int [n];
  if (a==0) exit(0);

  cout << "Entrez " << n << " nombres entiers : \n";
  for (int i=1; i<=n; i++) {
    cout << i << ": ";
    cin >> a[i-1];
  }

  cout << "Ils sont dans l'ordre inverse: \n";
  for(int i=n; i>0; i--)
    cout << a[i-1] << endl;

  delete[] a;

  return 0;
}
```

`int *a` est un pointeur sur un tableau d'entiers, i.e. il contient l'adresse mémoire du premier élément du tableau

mémoire insuffisante

Le tableau dynamique est traité de la même façon qu' un tableau ordinaire !

on n'a plus besoin de `a`;

mieux vaut libérer la mémoire !
a chaque instruction `new` devrait correspondre une instruction `delete` .

voir Allocation.cpp

Mots-clés du langage C++

asm	do	if	return	try
auto	double	inline	short	typedef
bool	dynamic_cast	int	signed	typeid
break	else	long	sizeof	typename
case	enum	mutable	static	union
catch	explicit	namespace	static_cast	unsigned
char	export	new	struct	using
class	extern	operator	switch	virtual
const	false	private	template	void
const_cast	float	protected	this	volatile
continue	for	public	throw	wchar_t
default	friend	register	true	while
delete	goto	reinterpret_cast		

mots-clés déjà vus

mots-clés rencontrés aujourd'hui

Résumé

Ce qu'il faut retenir / savoir faire à la fin de cette leçon :

Adresse mémoire (`&`)

Déclaration et initialisation des pointeurs

```
int n;
int *ptr = &n;
```

Parcourir un tableau avec un pointeur

Passage de paramètres aux fonctions :
passage par valeur
passage par pointeur
passage par référence

Allocation dynamique de la mémoire

`new`
`delete`

Memorisez ces deux programmes :

Inversion des éléments d'un tableau

Inversion.cpp

```
int main() {
  double x[10];
  double *left = &x[0];    //adresse premier element
  double *right = &x[9];    //adresse dernier element
  while (left < right) {
    double temp = *left;
    *left++ = *right;    //ou *left = *right; left++;
    *right-- = temp;      //ou *right = temp; right--;
  return 0; }
```

Allocation dynamique de la mémoire

Allocation.cpp

```
int main() {
  cout << "Combien d'elements ? ";
  int n;    cin >> n;
  int *a = new int [n];
  if(a==0) exit(0);    //allocation échue

  cout << "Entrez " << n << " nombres entiers : \n";
  for(int i=1; i<=n; i++) {
    cout << i << ": ";
    cin >> a[i-1];
  }

  delete[] a;
  return 0; }
```

Exercices – série 6

Questions

1. Comment accédez-vous à l'adresse mémoire d'une variable ?

2. Comment accédez-vous au contenu d'une location mémoire dont l'adresse est stockée dans un pointeur ?

3. Quelle est la différence entre : `int &r = n;` et `p = &n;` ?

4. Peut-on déréférencer un pointeur NULL ?

5. Pourquoi vaut-il mieux initialiser un pointeur lors de sa déclaration ?

6. Cherchez l'erreur dans ces fragments de code:

```
int main() {
  int *pInt;
  *pInt = 9;
  cout << *pInt;
}
```

```
int main() {
  double x[10];
  double *y = &a;
}
```

Exercices

1. Utilisez des pointeurs pour passer des paramètres aux fonctions. P. ex. écrivez un programme avec une fonction pour calculer la surface et le volume d'une sphère. Le rayon de la sphère est passé à la fonction par valeur; la surface et le volume de la sphère sont *renvoyés* par pointeur.

2. Développez le programme Adresse.cpp (page 5).

3. Répondez aux questions de la page 13.

4. Développez le programme Allocation.cpp (page 25).

5. Ecrivez un programme pour calculer la valeur moyenne des éléments d'un tableau. La dimension du tableau n'est connue que pendant l'exécution du programme. Utilisez l'allocation dynamique de la mémoire pour créer le tableau pendant l'exécution.

6. Ecrivez un programme pour ordonner les éléments d'un tableau de dimension N en ordre croissant. Utilisez l'allocation dynamique pour réserver la mémoire nécessaire pour stocker le tableau. Utilisez des pointeurs pour parcourir le tableau.

7. Des mesures sont stockées dans un fichier (ou entrées par le clavier). Ecrivez un programme pour lire ces données. Enregistrez les mesures dans un tableau de dimension appropriée. La première donnée dans le fichier décrit le nombre N de mesures. Le tableau est créé de façon dynamique pendant l'exécution du programme. Calculez la moyenne μ et l'écart quadratique moyen σ des données.

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N} \qquad \sigma = \sqrt{\frac{\sum_{i=1}^{N} \left(x_i - \mu\right)^2}{N-1}}$$

Imprimez le résultat sur l'écran.

8. Ecrivez des fonctions pour additionner, multiplier, etc. deux matrices de dimension quelconque. Dans les fonctions, les matrices sont parcourues avec des pointeurs. (voir ex. 7 et 9 de la leçon 4; cette fois utilisez les pointeurs !)