# Mémoire, tableaux et pointeurs en C/C++

**Nicolas Gazères**

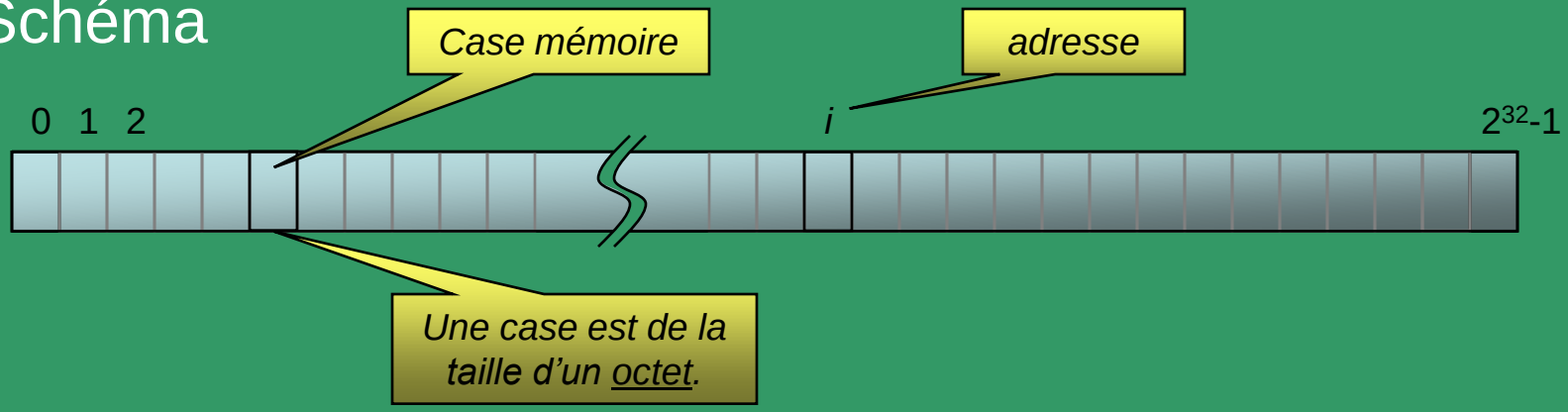
Dassault Systèmes
DS Research, Life Sciences
ngs@3ds.com

# La mémoire virtuelle

***C'est ce type de mémoire qu'utilise un process.***

- Modèle mental simplifié
  - La mémoire mise à disposition d'un *process* peut être vue comme une suite de $2^{32}$ *octets* numérotés (machine *32-bits*)
  - Le numéro de l'octet s'appelle *l'adresse*.

- Schéma

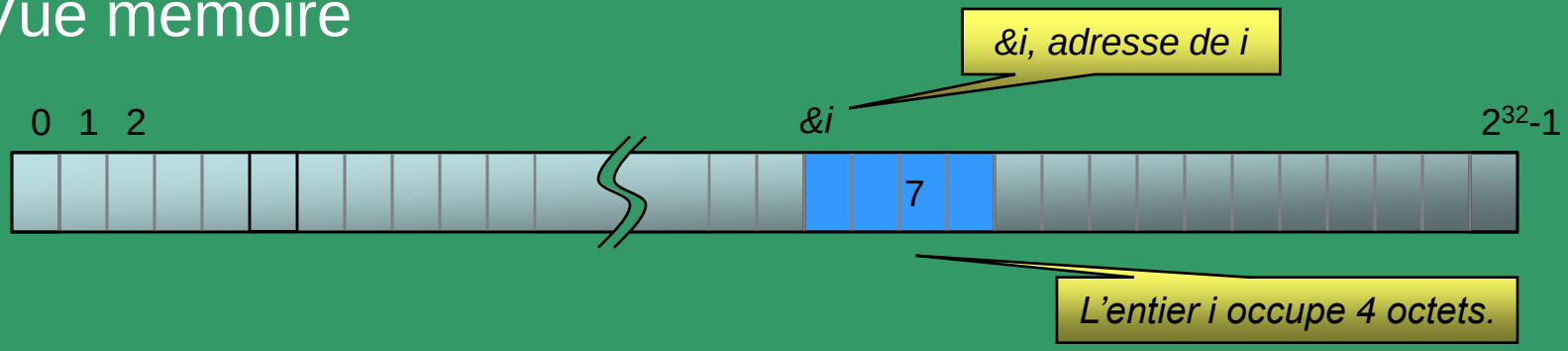


# Variable, Adresse, Opérateur &

- Toute variable en C est stockée « quelque part en mémoire » et possède de ce fait une adresse.
  - Seule exception: les variables avec le qualificateur *register*.
- L'opérateur d'adressage *&*
  - Il permet d'obtenir l'adresse d'une variable.
- Syntaxe

```
int i = 7;
printf( "L'adresse de i est %x",&i );
```

- Vue mémoire



# Les tableaux statiques

# Tableaux statiques

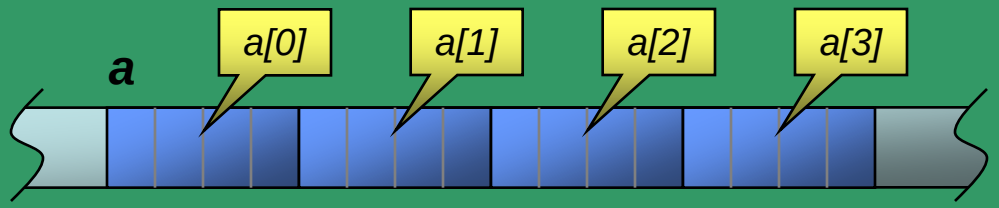
- Définition
  - C'est une séquence de variables
    - de même type (short, float, int, char…)
    - qui occupent en mémoire des positions consécutives et contiguës.
  - Un tableau statique possède un nombre d'éléments qui doit être connu dès la compilation.
- Exemples

```
int tab1[ 64 ] = {1,3,5,7,9 };      // déclaration + définition de tab1
                                    // le compilateur remplit le début du tableau
                                    // avec les valeurs de la liste d'initialisation
                                    // et complète par des zéros.

double tab2[ ] = {3.14, 127, 22};   // longueur omise volontairement par le programmeur.
                                    // elle est calculée automatiquement par le compilateur (3)

char tab3[ 0 ] ;                    // définition de tab3: erreur: taille nulle interdite
char tab4[ ] = {} ;                 // définition de tab4: erreur: taille nulle interdite

int tab5[ ], tab6[ 64 ] ;           // ce sont des déclarations, pour tab5 et tab6.
```

# Tableaux statiques

- Accès en lecture aux membres du tableau:
  - Si un tableau *a* possède *N* éléments:
    - le premier élément est *a[0]* (attention: pas *a[1]* !)
    - le dernier élément est *a[N-1]* (attention: pas *a[N]* !)
  - Exemple:
    - ***a*, ci-dessous,** est un tableau formé des *quatre* entiers *a[0]*, *a[1]*, *a[2]* et *a[3]*.
    - Attention: *a[4]* n'est pas membre du tableau !

```
{
   int a[4];
   …
```




- Accès en écriture aux éléments du tableau (affectation)

```
a[ 2 ] = 58;
```

# Les chaînes de caractères en C

- Une ***chaîne de caractères*** est
  - un tableau de *char*,
  - qui se termine obligatoirement par un caractère *nul*.
    - Attention: la *taille complète occupée en mémoire* par une chaîne C est son nombre de caractères *plus un* (ie. *sizeof*)
    - Ne pas confondre avec la *longueur de la chaîne*, qui ne prend pas en compte le caractère nul terminal (ie. *strlen*).
- Exemples:
  - Remarque: le caractère nul terminal est implicite
    - Il est ajouté par le compilateur…

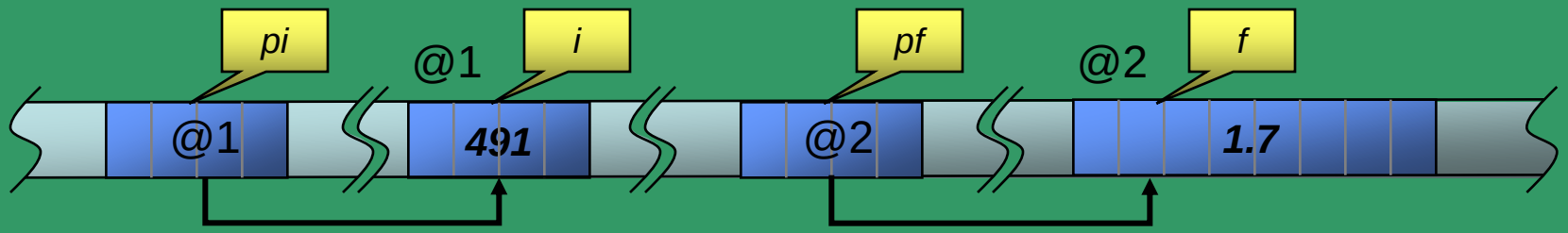
```
char t1[ ] = "toto";                    // Donne «toto»
char t2[ ] = "";                        // La chaîne vide
char t3[ ] = "Le guillemet est \"";     // Donne «Le guillemet est "»
char t3[ ] = "le debut " "la fin";      // Donne «le debut la fin»
```

# Les pointeurs

# Pointeur

- Définition
  - Un *pointeur* est une *variable* qui peut contenir l'adresse d'une autre variable.
    - *Remarque:* un pointeur peut aussi contenir l'adresse d'une *fonction* (voir plus loin…)
- Conséquence
  - Un pointeur occupe donc de l'espace mémoire
    - *sizeof(ptr)* vaut 4 octets sur une machine 32 bits (8 octets sur 64 bits)
- Exemples

```
// Exemples de déclarations de pointeurs
int i = 491 ;          // i est un entier
int *pi = &i;          // pi est un pointeur sur l'entier i
double f = 1.7 ;       // f est un flottant double précision
double *pf = &f;       // pf est un pointeur sur le flottant f
```

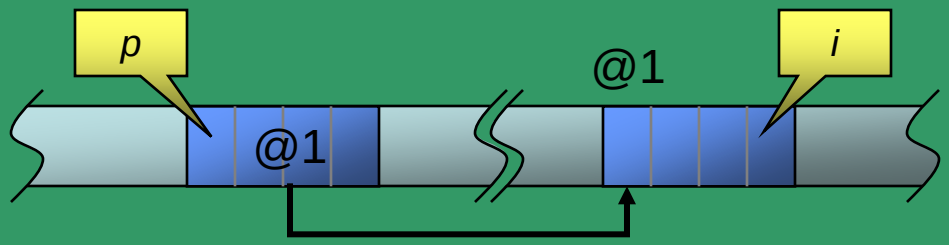


# Déréférencement de pointeur

- Terminologie
  - « *Déréférencer le pointeur* » = accéder à l'objet pointé.

- L'opérateur * permet d'accéder à l'objet pointé par un pointeur
  - …pour *lire* ou pour *modifier* sa valeur.
  - L'expression **p* désigne l'objet pointé par le pointeur *p*.
  - Attention à ne pas confondre cet * unaire avec l'opérateur de multiplication.

```
int i=7, *p=&i;        // déclarations+définitions

int j = *p + 1 ;       // accès à i en lecture, via p
*p = 8 ;               // accès à i en écriture, via p
```

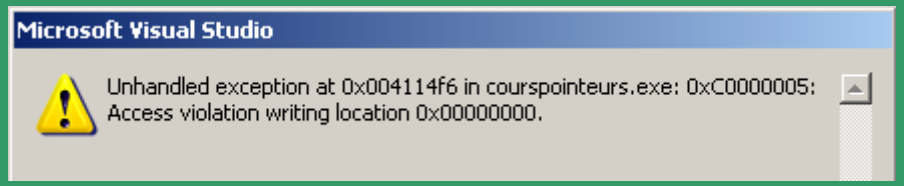


# Attention: erreur classique

- Exemple

```
int *p ;     // déclaration de pointeur non-initialisé
…
*p = 1 ;     // accès à *p en écriture → ???
```

  - En C, un pointeur non-initialisé peut contenir n'importe quoi.
    - En C++, ce pointeur serait initialisé à 0 (ce qui n'arrange rien !)
- Conséquences possibles du déréférencement de *p* (lecture ou écriture)
  - L'arrêt brutal du programme (*Access Violation, exception de code 5*).
    - Si l'adresse contenue dans *p* est dans une zone mémoire « *inaccessible* »

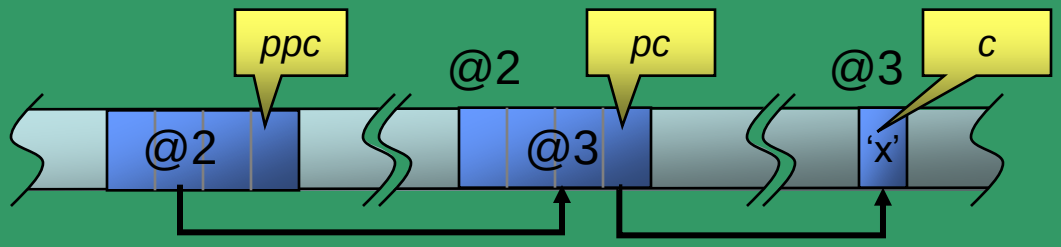



  - Lire une information n'importe où, donc signifiant n'importe quoi,
    - Si l'adresse contenue dans *p* est dans une zone mémoire « *accessible* »
  - L'effet peut être *non-reproductible* d'une exécution à l'autre !

# Pointeur sur Pointeur

- Par définition, un *pointeur* est une *variable* qui peut contenir l'adresse d'une autre variable.
  - Or, un pointeur est une variable comme une autre.
  - On peut donc définir un *pointeur sur un pointeur*.
    - Et récursivement…
- Syntaxe
  - on rajoute une étoile par niveau de pointeur.

```
char c = 'x';          // c = caractère
char *pc = &c ;        // pc = pointeur sur caractère (pointe sur c).
char **ppc = &pc;      // ppc = pointeur sur pointeur sur caractère (pointe sur pc)
```



# Alignement

- En général, une variable de type natif ne peut pas se trouver à une adresse mémoire totalement libre.
  - ***L'adresse en mémoire d'une variable d'un type natif doit être un multiple de son* sizeof.**
    - Les *double* ont des adresses multiples de 8 ( =*sizeof(double)* )
    - Les *int* ont des adresses multiples de 4 ( =*sizeof(int)* )
    - Les *short* ont des adresses multiples de 2 ( =*sizeof(short)* )
  - Le seul type indifférent à cette contrainte est le *char.*
    - Forcément, il n'occupe qu'un octet.
- Cette contrainte est un reflet des optimisations de l'architecture des micro-processeurs.
  - Cela permet de réduire le nombre de transistors.
- Conséquence
  - Si on veut qu'il soit déréférençable, un « pointeur sur T » doit forcément contenir une adresse multiple de *sizeof(T)*.
    - Sinon, arrêt brutal de l'exécution.
    - Sur certaines machines, ça peut passer quand même, même si ce n'est pas le cas (parce que l'OS corrige), mais c'est beaucoup plus lent; donc **à éviter**.

# Le pointeur nul

- C'est un pointeur dont la valeur est 0.
  - La constante symbolique ***NULL*** a été définie pour lui (*stdio.h*).
- Un pointeur nul ne pointe sur aucun objet.
  - on ne peut pas déréférencer le pointeur nul.
  - L'adresse nulle est « protégée »
    - il y a une *erreur à l'exécution* si on tente de la déréférencer (pas à la compilation !)
- Mais une fonction peut très bien renvoyer un pointeur nul.
  - Comme le pointeur nul est une valeur « anormale », il peut donc être utilisé pour signaler une condition d'erreur.

# lvalue

- Définition
  - une ***lvalue*** est une expression qui réfère à un emplacement-mémoire.

- Interprétation intuitive
  - Une *lvalue* est « quelque chose qui peut être à gauche d'un signe = »
  - Attention, une *lvalue* peut aussi être à droite d'un signe =
    - Mais dans ce cas, c'est la *valeur* de la variable qui est prise en compte, pas son emplacement-mémoire.

- L'opérateur d'adressage & permet d'obtenir l'adresse d'une variable.
  - Cet opérateur prend nécessairement une *lvalue* en entrée.
    - L'adresse renvoyée n'est pas une *lvalue*.

- Exemples (en C)
  - i, p[i+3], *p, … sont des *lvalues*
  - i+1, i+j, f(i), a++, a && b, tab ne sont pas des *lvalues*.
  - une variable avec le qualificateur *register* n'est pas une *lvalue*.

# Quizz

- Les expressions suivantes ont-elles un sens ?
  - &&x
    - n'a pas de sens (adresse d'une adresse)
    - ne compile pas
  - **p
    - compile si p est un pointeur et *p aussi
    - c'est une lvalue.
  - &*p
    - compile, si p est un pointeur, car *p est une lvalue.
    - c'est une adresse, donc pas une lvalue
  - *&p
    - compile, si p est une variable
    - c'est une lvalue.

# Opérations sur les pointeurs
## (autres que le déréférencement)

- Opérations autorisées
  - ***Comparaison*** de pointeurs: >, <, >=, <=, ==, !=
  - ***Négation logique*** de pointeur (*not*): !
    - Si p est le pointeur nul, renvoie quelque chose de non-nul, donc condition *true.*
    - Si p est non-nul, renvoie 0, donc condition *false*.
  - Exemple

```
if (!p) {
   …
}
```

- Opérations interdites
  - Les ***opérations binaires*** avec *deux opérandes de type pointeur* (autres que la soustraction et les comparaisons) sont *interdites* par le compilateur.
  - Exemples:
    - p+q, p*q, p/q, p%q, …
    - p+=q, p-=q, p*=q, p/=q, p%=q,…

# Pointeur sur *void*

- Propriétés
  - 1/ Peut pointer sur tout type de données.
    - On peut lui affecter tout type de pointeur !
  - *2/ N'est pas déréférençable.*
  - 3/ L'arithmétique de pointeur ne s'applique pas sur lui.
    - Le compilateur ne sait pas calculer le *sizeof* de l'objet pointé.
- Exemples

```
int     i = 3;
float   f = 1.7 ;
void    *p = 0;          // pointeur sur void

p = &i;                  // OK (cf. 1/)
p = &f;                  // OK (cf. 1/)

… *p … ;                 // Erreur de compilation (cf. 2/)
p++ ;                    // Erreur de compilation (cf. 3/)
```

# *Cast* de pointeurs
## (*conversion de type explicite*)

- En C, l'affectation d'un pointeur à un autre n'est autorisée que lorsque les pointeurs sont du même type.
  - *Remarque: cela sera un peu différent en C++…*
- **Principe du *cast***
  - Lorsqu'une expression est précédée d'un nom de type entre parenthèses, sa valeur est convertie dans le type indiqué.

```
int    *pi = … ;
float * pf1 = pi ;                  // erreur de compilation
float * pf2 = (float*) pi;          // Compile, mais risqué.
void  * pv = pi ;                   // OK sans cast ! Un void* peut être valué
                                    // par tout type de pointeur
float  * pf3 = pv ;                 // erreur de compilation
float  * pf4 = (float*) pv ;        // OK avec cast !
```

- On peut forcer l'affectation par un *cast* quand on sait ce que l'on fait
  - pointeur sur *void*
  - c'est le seul moyen pour *malloc()* (voir plus loin)

# Pointeur sur fonction

- Une fonction est un objet pointable
  - tout comme les variables de types plus classiques (*int*, *float*…)
- Un pointeur sur fonction
  - possède un *type*
    - 1/ qui est vérifié par le compilateur lors des appels de fonction
    - 2/ qui se prête à certaines conversions standards
  - sert à passer une fonction comme un paramètre du calcul.

*Déclaration d'un argument « fonction »*

**monSource.cpp**

```
void evaluePerformance( double (*s)(double) , double x ) {
    for (int i=0; i<10000; i++ )
        s(x);       // on peut aussi écrire (*s)(x)
}

int main() {
   evaluePerformance( sin, PI/4 );
   evaluePerformance( mySin, PI/4 ) ;
   evaluePerformance( printf, PI/4 ) ;          // Compile KO (cf. 1/)
}
```

**math.h**

```
// sinus standard
double sin( double );
```

**mesFonctions.h**

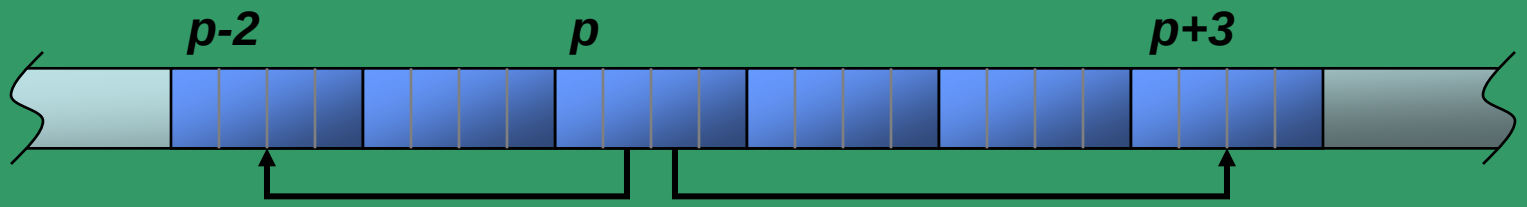
```
// mon sinus optimisé
double mySin( double );
```

# Arithmétique de pointeur (1)

- Règle
  - A un pointeur, on peut *ajouter* une valeur « entière » (ie. *char*, *short*, *int*, *long, long long/__int64* et leurs versions *unsigned*).

- Exemple:
  - Si p est un « *pointeur sur T* »:
    - « p+3 » représente l'adresse du 3ème objet *de type T* après p.
    - Le décalage entre p et p+3
      - est de 3 *éléments* de type *T*
      - Est de *3 sizeof(T) octets*.
    - « p-2 » représente l'adresse du 2ème objet *de type T* avant p.

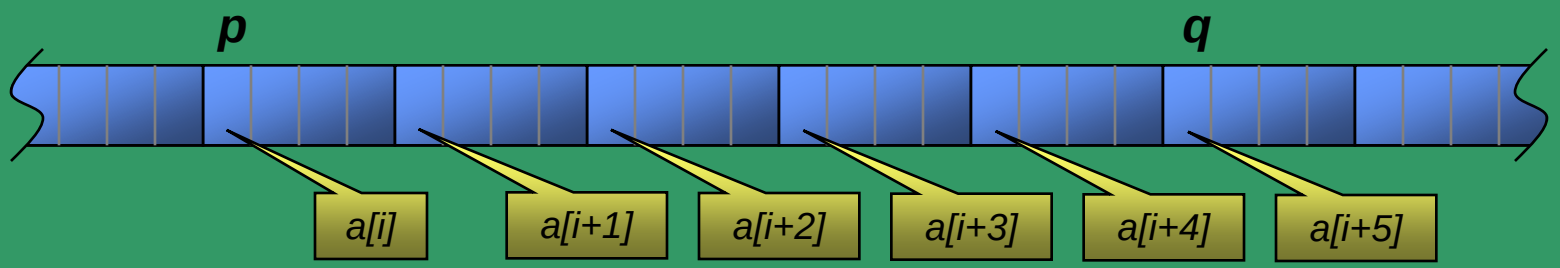
- Illustration (ici, *T* est un type dont le *sizeof* vaut 4, par exemple *int*)



# Arithmétique de pointeur (2)

- Règle
  - On peut *calculer la différence de deux pointeurs* de *même type* pointant vers les éléments d'un *même tableau*.
  - Le résultat est un entier qui correspond au *nombre d'éléments* de tableau pour passer du premier pointeur au deuxième.
    - Attention: Ce n'est pas la différence des adresses en octets !

- Exemple:
  - Si p et q sont des *pointeurs* du même tableau d'*int* (cf. schéma)
    - « *q-p* » représente le nombre d'*int* à sauter pour passer de *p* à *q*.
    - « *q-p* » ne vaut pas 20, mais 5 !
    - En effet, 5 *sizeof(int)*=20.



# Arithmétique de pointeur (3)

- Attention
  - On ne peut pas ajouter (ni soustraire) un *double* ou un *float* à un pointeur.
  - L'arithmétique de pointeur ne s'applique pas sur le pointeur *void*.
    - Le compilateur ne sait pas calculer le *sizeof* de l'objet pointé.

# Allocation dynamique

**Voir présentation séparée**

# Utilité des pointeurs lors d'appels de fonctions qui modifient leurs arguments.

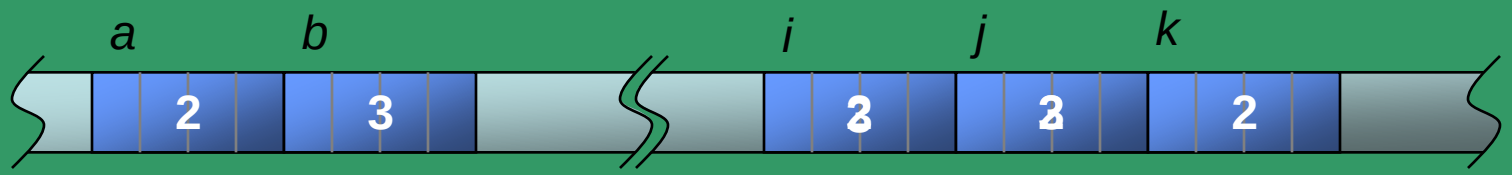
# Passage d'arguments de fonction
## *Exemple (passage par valeur)*

- Nous voulons coder une fonction *swap()* qui échange les valeurs de deux variables entières:

```
void swap( int i, int j )
{
   int k = i ;
   i = j;
   j = k;
}
```

```
int main()
{
   int a=2, b=3;
   swap( a, b );
   … // ← que valent a et b ?
}
```

- Interprétation



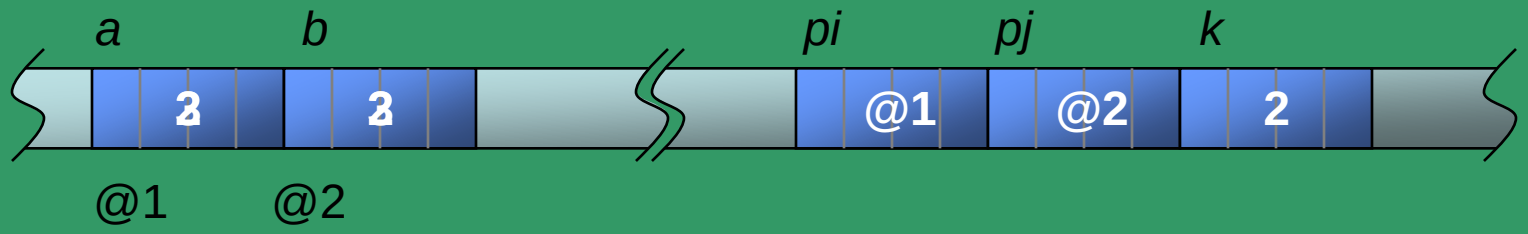
# Exemple corrigé *(passage par adresse)*

- Il faut passer l'adresse d'une variable pour que cette variable soit modifiable dans la fonction.

```
void swap( int *pi, int *pj )
{
   int k = *pi ;
   *pi = *pj;
   *pj = k;
}
```

```
int main()
{
   int a=2, b=3;
   swap( &a, &b );
   // OK
}
```

- On dit que la variable *a* est passée *par adresse*.
  - C'est le pointeur (sur a) qui est passé par valeur.



# Les entrées simples: *scanf()*

- Entrée au clavier
  - par la fonction *scanf()*, déclarée dans <*stdio.h*>
- Syntaxe
  - une chaine de formatage (avec un certain nombre de codes %_)
  - autant de *pointeurs* sur variables d'accueil que de codes %_.
- Exemple
  - ***Il faut passer les variables d'accueil par addresse !***

```
int i1 = 0 ;
float f1 = 0.0;
char tmp[40];
scanf( "%d %lf %s“,  & i1,  & f1, & tmp[0] );
```

On spécifie les *adresses* des variables d'accueil !

  - Cet exemple signifie:
    - lire un entier au clavier et le stocker dans *i1*,
    - *puis* lire un flottant au clavier et le stocker dans *i2.*
    - *puis* lire une chaîne de caractères au clavier et la stocker dans *tmp.*
      - *pourvu qu'il y en ait moins de 40…*

# Quizz sur *scanf()*

```
{
   int n ;
   scanf( "%d", n );
   printf( "%d", n );
   …
```

L'entier n contient une valeur aléatoire en C, 0 en C++.

Le nombre saisi au clavier est stocké à l'adresse *n*:

Pour *scanf()*:

**→ Comportement aléatoire en C; plantage en C++ (au niveau du scanf).**

```
{
   int n ;
   scanf( "%d", &n );
   printf( "%d", n );
   …
```

L'entier *n* est à un emplacement bien défini en mémoire.

C'est l'adresse de *n* qui est passée à *scanf()*.

→ **OK**

```
{
   int *pn ;
   scanf( "%d", pn );
   printf( "%d", pn );
   …
```

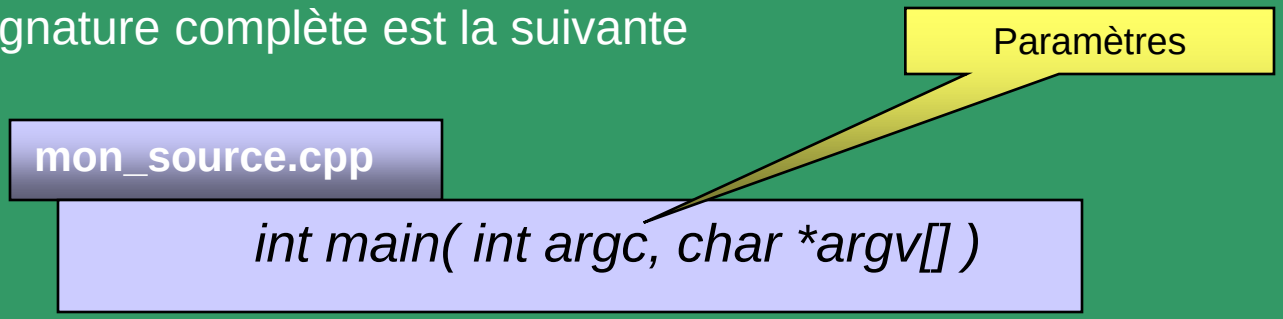
Pour *scanf()*: Le pointeur sur entier *pn* contient une valeur aléatoire en C, 0 en C++.

**→ Comportement aléatoire en C; plantage en C++.**

Pour *printf()*: l'argument est de type pointeur: *printf()* ne plante pas mais affiche une valeur incompréhensible.

# Retour sur la fonction *main()*

- C'est le **point d'entrée** de l'exécutable:
  - La signature complète est la suivante




  - Cette signature permet de récupérer les paramètres du programme:
    - soit sur la ligne de commande
    - soit par drag-and-drop
  - Description
    - *argc* Nombre d'arguments de ligne de commande+1
    - *argv* Tableau de pointeurs *char** tel que:
      - *argv[0]* contient le nom du programme.
      - *argv[i]* contient le i$^{ème}$ argument du programme (*0<i<argc*)
      - *argv* se termine par un pointeur nul ( *argv[argc]==0* )

# Recommandations

- Un pointeur non-initialisé ne vaut pas forcément 0 en C !
  - Toujours initialiser les pointeurs à 0.
  - Et les remettre à 0 une fois qu'ils ne sont plus nécessaires

- Attention à l'arithmétique de pointeurs !
  - une source inépuisable de bugs.
  - utiliser plutôt du code déjà écrites (et déjà débuggé !)
  - vous avez peu de chances d'y échapper…

- Quand c'est possible, il faut préférer l'utilisation de *tableaux de taille fixe* (quitte à surestimer la taille), plutôt que de tableaux de taille variable alloués dynamiquement par *malloc()*.
  - Pour des questions de rapidité d'exécution.

- Pour afficher un pointeur, on peut utiliser le code *%p*:

```
printf( "%p", ptr );
```

# Fin